ECE568 笔记汇总

- Cryptography - Block Ciphers
- Cryptography - Ciphers
- Cryptography - Hashes, MACs, and Digital-Signitures
- Cryptography - Public-Key Cryptography
- Cryptography - Stream Ciphers

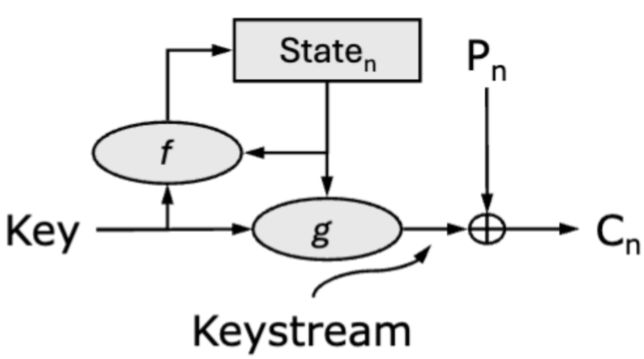
Table of Contents

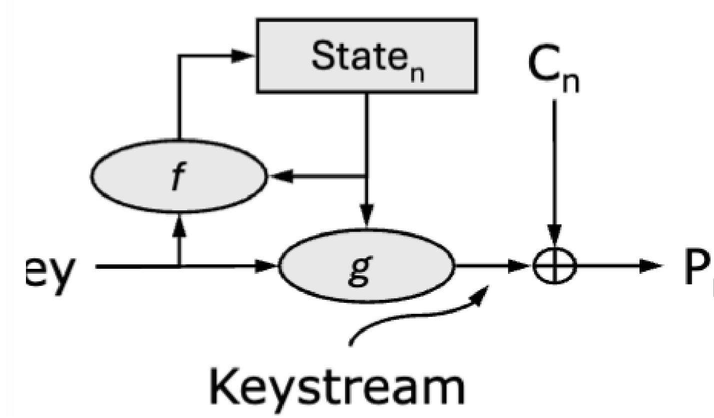
Stream Ciphers

- motivated by encryption/decryption with low latency
    - real-time systems
- operates a bit/byte at a time
    - produce CT exactly as long as PT
    - stream ciphers have **no modes**
- the pad (**key stream**) is a pseudo-random sequence of bits generated from a much shorter encryption key
    - stream of random bits is XORed with PT to create CT

Synchronous Stream Ciphers

- **key stream** is **independent** of the message text
    - state is modified by the function f and the key
    - each step uses **feedback**, in which f takes the current state to produce the new state





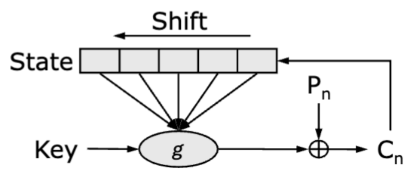

```
state = IV    # Initialize the state register
while read(plaintext):
        state = f(key, state)  # Update state
        keystream = g(key, state)  # Generate keystream
        cryptotext = XOR(plaintext, keystream)  # Encrypt plaintext
        write(cryptotext)
```

```
state = IV   # Initialize the state register
while read(cryptotext):
        state = f(key, state)  # Update state
        keystream = g(key, state)  # Generate keystream
        plaintext = XOR(cryptotext, keystream)  # Encrypt plaintext
        write(plaintext)
```

- encryption XOR key stream with PT
- decryption uses the key to produce the same key stream and XOR with CT

Self-Synchronizing Stream Ciphers

- key stream **depends** on the PT
  - state consists of a **shift register**
  - every CT bit created is shifted into the shift register and fed back as an input into **g**
  - Each CT bit has an effect on the next **n** bits (length of shift register)




```
state = IV   # Initialize the state register
while readByte(plaintext):
        keystream = g(key, state)  # Generate next keystream byte
        cryptotext = XOR(plaintext, keystream)  # Encrypt plaintext
        writeByte(cryptotext)
        state = cryptotext + state[1:]  # Shift cryptotext byte into state
```

Stream Cipher Properties

- **Security** - similar to OTP
  - dangerous to use same key stream to encrypt 2 different messages
    - synchronous ciphers, IV must be changed for new message
    - self-synchronizing ciphers, insert random data at the beginning
  - **malleable** - CT can be changed to generate related PT
  - with self-synchronizing ciphers, adversary can **replay** previously-sent CT into a stream, and cipher will re-sync #question/ece568
- **Performance** - better performance than block ciphers
  - especially for hardware implementation
  - key stream for synchronous stream ciphers can be pre-computed before the message arrives so encryption/decryption is simply an XOR (storage overhead?) #question/ece568
- **Error Propagation**
  - for synchronous stream ciphers, transmission error only affects corresponding PT bits
  - for self-synchronizing, error affects next **n** bits
- **Error Recovery**
  - synchronous, if a section of CT is lost, CT stream and key-stream become out-of-sync and recovery is *impossible* unless we know how much CT is lost
  - self-synchronizing, will recover after **n** bits have passed #question/ece568

Stream Cipher Implementation

RC4

- Ron's Code is a SC created in RSA Labs
  - good performance in software
- S is an array of size 256 that contains the state
  - always contains a permutation of 0...255
  - key-length is generally 5-16 bits
  - key scheduling algorithm initializes state S
  - PRGA generates key stream

```
for i from 0 to 255
  S[i] := I
endfor
j := 0
for i from 0 to 255
  j := (j + S[i] + key[i mod keylength]) mod 256
  swap(S[i],S[j])
endfor
```

Key Scheduling Algorithm

```
i := 0
j := 0
while GeneratingOutput:
  i := (i + 1) mod 256
  j := (j + S[i]) mod 256
  swap(S[i],S[j])
  output S[(S[i] + S[j]) mod 256]
endwhile
```

Pseudo-Random Generation Algo

Selecting Right Cipher

- SC = better performance, but difficult to use safely
  - vulnerable to replay, IV's need to be managed never repeat
  - repeating IV is more damaging than CBC
- block ciphers are easier and more commonly used
  - no reason to use DES (backward compatibility)
  - CBC is most common
  - ECB safe for short data