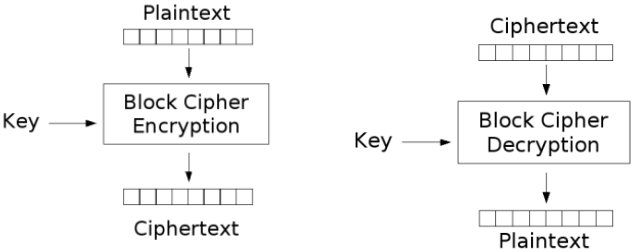
MTAT.07.017
Applied Cryptography

Block ciphers (AES)

University of Tartu

Spring 2023

Block cipher



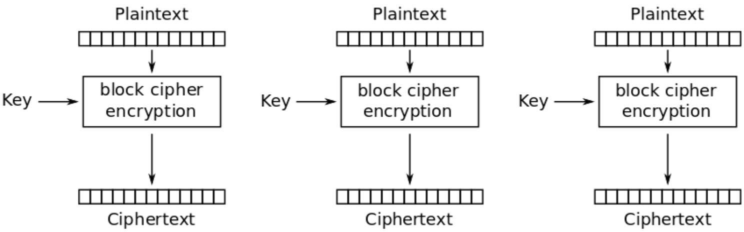
Properties:

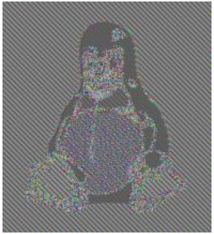
- Deterministic
- Without the key plaintext cannot be found
- Valid plaintext-ciphertext pairs do not leak the key
- Diffusion & Confusion

AES – Advanced Encryption Standard (NIST 2001)

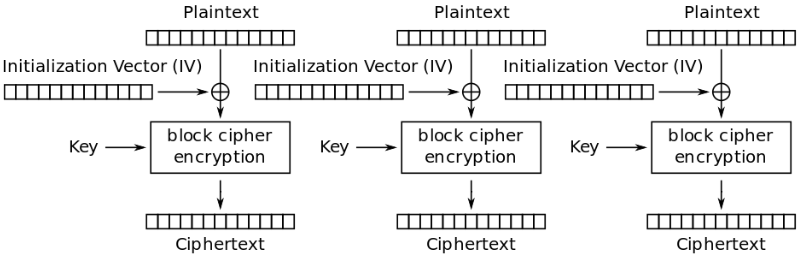
- 16-byte (128-bit) block size
- key sizes – 128/192/256 bits

Electronic Codebook (ECB) mode



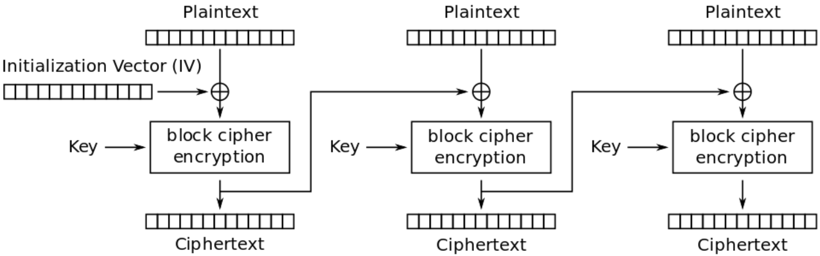
Initialization Vector (IV)

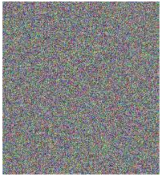


- On encryption XOR plaintext with random IV
- IV must be sent along with the ciphertext
- IV does not have to be secret
- On decryption XOR plaintext with IV

The problem? The ciphertext is two times larger than the plaintext

Cipher Block Chaining (CBC) mode





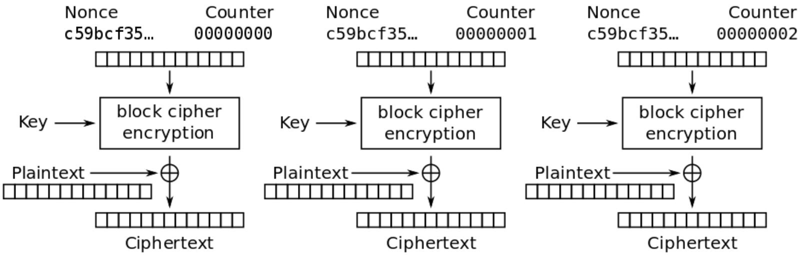
- Serial encryption
- Parallel decryption
- What about integrity (malleability)?

Plaintext padding

- Random padding:
  1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a **XX YY ZZ**
- Zero padding:
  1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a **00 00 00**
- ANSI X.923:
  1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a **00 00 03**
- ISO/IEC 10126:
  1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a **F1 A6 03**
- ISO/IEC 7816-4:
  1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a **80 00 00**
- PKCS#5/PKCS#7:
  1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a **03 03 03**
  1a 1a 1a 1a 1a 1a 1a 1a 1a 1a 1a **05 05 05 05 05**

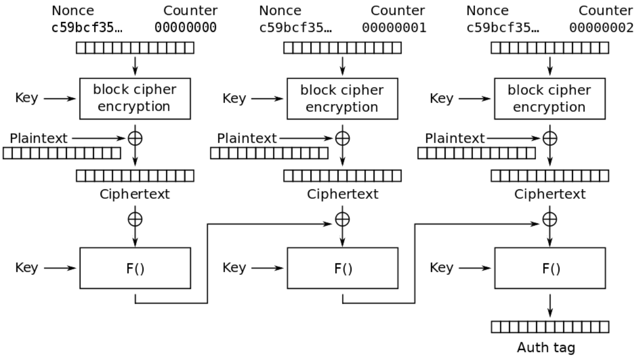
Padding is added even if the plaintext occupies a full block

Counter (CTR) mode



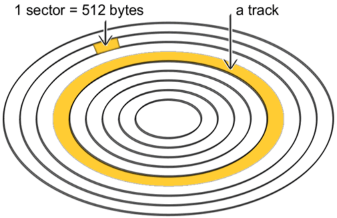
- Block cipher-based PRNG
- Turns block cipher into a stream cipher
- Nonce must never be reused!
- Block ciphers vs stream ciphers

Galois/Counter Mode (GCM)



- Authentication tag is 16 bytes (128 bits) long
- Standardized for 16-byte block size
- Nonce 12 bytes, counter 4 bytes (max 64GB of data)
- Authenticated Encryption with Associated Data (AEAD)

Disk encryption



- Encrypt the whole disk using CBC?
  - Each sector is encrypted separately
  - Use sector number as IV
- How to provide integrity?
  - MAC is not an option
  - XTS mode
- Password change without disk reencryption
  - Random master key is used to encrypt data
  - Master key is stored encrypted in disk header
  - User's password decrypts master key
  - Enterprise solutions have several encryptions of master key

Password-based encryption

Deriving strong (128-bit) keys from short, low entropy passwords?

- Use hash of the password
  - Distributes entropy over all 128 bits
- Use salt as an addition to password
  - Prevents multi-target attacks
- Use iterated hash to slow down brute-force
  - Adds arbitrary number of operations to brute-force

Password-Based Key Derivation Function 2 (PBKDF2)

`key = PBKDF2(PRF, Password, Salt, iter, kLen)`

- `PRF` – pseudorandom function (e.g., HMAC-MD5, HMAC-SHA1)
- `Password` – password entered by the user
- `Salt` – random cryptographic salt
  - Recommended at least 64 bits
- `iter` – number of iterations desired
  - Recommended at least 1'000 iterations (increases the security level by 10 bits)
  - NIST recommends 10'000'000 for critical keys (increases the security level by 23 bits)
  - Ubuntu disk encryption selects iteration count that corresponds to 1 second
- `kLen` – desired length of the derived key

- WPA2 uses `key = PBKDF2(HMAC-SHA1, passphrase, ssid, 4096, 256)`
- Truecrypt uses PBKDF2 with 2000 iterations

New schemes should use scrypt / Argon2id

Task: Password-based file encryption

Implement a utility that encrypts and decrypts files using a password:

```
$ ./aes.py
Usage:
-encrypt <plaintextfile> <ciphertextfile>
-decrypt <ciphertextfile> <plaintextfile>

$ ./aes.py -encrypt plain.txt plain.txt.enc
[+] Benchmark: 806972 PBKDF2 iterations in 1 second
[?] Enter password: asd

$ ./aes.py -decrypt plain.txt.enc plain.txt
[?] Enter password: asd
```

- The integrity of the ciphertext is provided using HMAC
- Parameters are ASN.1 DER-encoded and stored as a header of the ciphertext file

Task: Password-based file encryption

```
$ dumpasn1 plain.txt.enc
  0 102: SEQUENCE {
  2  18:   SEQUENCE {
  4   8:     OCTET STRING 59 9C B8 63 65 82 FA 39
 14   3:     INTEGER 806972
 19   1:     INTEGER 48
       :     }
 22  29:   SEQUENCE {
 24   9:     OBJECT IDENTIFIER aes128-CBC (2 16 840 1 101 3 4 1 2)
 35  16:     OCTET STRING 7A AD 60 A6 50 F2 42 49 9E 7A 34 70 39 37 CF C6
       :     }
 53  49:   SEQUENCE {
 55  13:     SEQUENCE {
 57   9:       OBJECT IDENTIFIER sha-256 (2 16 840 1 101 3 4 2 1)
 68   0:       NULL
       :       }
 70  32:     OCTET STRING
       :       FC 71 A6 AC D5 35 52 1C C7 BF 13 15 AB B8 FA 85
       :       87 81 1E C9 8F 3F 83 91 0B D7 B3 86 A2 ED 8B C3
       :     }
       :   }
Warning: Further data follows ASN.1 data at position 104.

EncInfo ::= SEQUENCE {
    kdfInfo pbkdf2params,
    cipherInfo aesInfo,
    hmacInfo DigestInfo
}
pbkdf2params ::= SEQUENCE {
    salt OCTET STRING,
    iterationCount INTEGER (1..MAX),
    keyLength INTEGER (1..MAX)
}
aesInfo :: = SEQUENCE {
    algorithm OBJECT IDENTIFIER,
    iv OCTET STRING OPTIONAL,
}
```

Task: Test cases

```
$ echo -n "hello world" > plain
$ ./aes.py -encrypt plain plain.enc
[+] Benchmark: 818397 PBKDF2 iterations in 1 second
[?] Enter password: asd
$ ./aes.py -decrypt plain.enc plain.new
[?] Enter password: asd
$ hexdump -C plain.new
00000000  68 65 6c 6c 6f 20 77 6f  72 6c 64                 |hello world|
0000000b

$ echo -e -n "hello world \xc3\x28\x02\x02" > plain
$ ./aes.py -encrypt plain plain.enc
[+] Benchmark: 816193 PBKDF2 iterations in 1 second
[?] Enter password: asd
$ ./aes.py -decrypt plain.enc plain.new
[?] Enter password: asd
$ hexdump -C plain.new
00000000  68 65 6c 6c 6f 20 77 6f  72 6c 64 20 c3 28 02 02  |hello world .(..|
00000010


$ ./aes.py -decrypt plain.enc plain.new
[?] Enter password: asdd
[-] HMAC verification failure: wrong password or modified ciphertext!

$ ./aes.py -decrypt big.enc big
[?] Enter password: bigfilepassword
                              |  |__ key AES: c1dcf40379d38d8a9ec9204fcf9652cf
                              |_____ key HMAC: 53d8b0bb34cc72cfb0689b6bd74abb4a35afa495ed054c283a8b5b16c1b93968

$ openssl dgst -sha256 big
SHA256(big)= 066090dceeece702a28c9ab08677bd91f7e53fa6b5a69d1d4c3e9a2d556e4cee
```

Task: Password-based file encryption

- Slow down password brute-force attacks to 1 try/second
  - Benchmark the time required for 10 000 iterations
  - Extrapolate the iteration count to 1 second

```
start = time.time()
...
stop = time.time()
took = stop-start
```

- Use PBKDF2 with HMAC-SHA1 to derive 48 bytes:

```
pbkdf2_hmac('sha1', password, salt, iter, 48)
```

  - Use the first 16 bytes as AES-128 key
  - The next 32 bytes as HMAC-SHA256 key
- Generate IV (16 bytes) and salt (8 bytes) randomly
- Implement CBC mode using *pure* AES-128 (ECB mode)

```
cipher = AES.new(key_aes, AES.MODE_ECB)
cipher.encrypt(strxor(plaintext_block, iv_current))
strxor(cipher.decrypt(ciphertext_block), iv_current)
```

- Use PKCS#5 padding

Task: Password-based file encryption

- The entire plaintext/ciphertext can be stored and processed in memory
- Calculate the MAC on the ciphertext
  - Prepend IV to ciphertext when calculating MAC
  - Use `hmac.compare_digest(mac_calculated, mac_from_der)`

Side channel attacks

```
def authenticate_admin(submitted_password):
    hardcoded_password = 'qwerty'
    if submitted_password == hardcoded_password:
        return 1 # access granted
    return 0 # access denied
```

- Function is vulnerable to a timing attack (comparison stops after the first incorrect byte):
    - password 'aaaaaa' – 1ms
    - password 'baaaaa' – 1ms
    - password 'qaaaaa' – 2ms
    - password 'qwaaaa' – 3ms
    - password 'qweaaa' – 4ms
    - password 'qweraa' – 5ms
- Exploitable over low-latency networks
- Adding `sleep(random())` before `return` will not help
- Constant-time string comparison function needed

Your `hmac.py` solution (homework 03) is vulnerable to a timing attack

`hmac.compare_digest(mac_calculated, mac_from_der)`

Questions

- How does a block cipher work (e.g., takes as input, returns)?
- What happens to the ciphertext if a single plaintext or key bit is changed?
- Why is encrypting every block of a file independently not secure?
- Why do we apply an initialization vector (IV) to plaintext?
- How can integrity be provided for ciphertext?
- When should a stream cipher be used and when should we use a block cipher?
- How should a short password be converted to a 128-bit encryption key?
- What is a side-channel vulnerability?