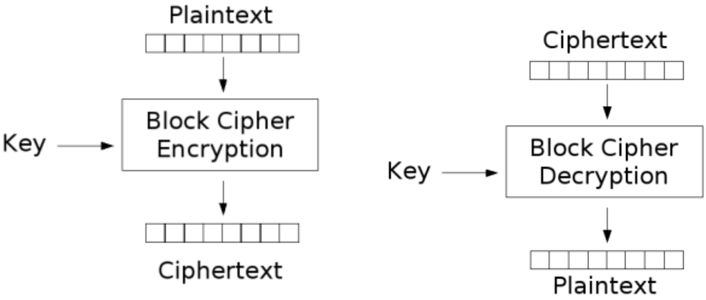
MTAT.07.017
Applied Cryptography

Block Ciphers (AES)

University of Tartu

Spring 2018

Block Ciphers



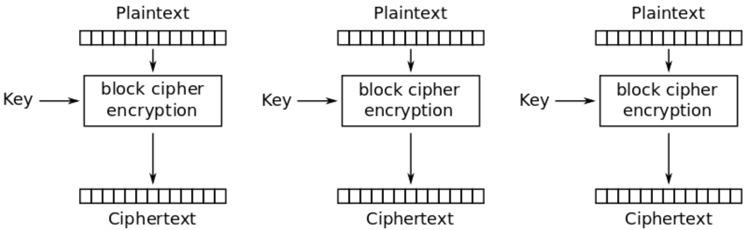
Properties:

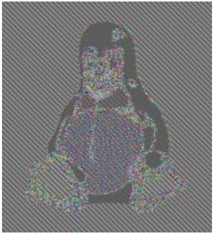
- Deterministic
- Without the key plaintext cannot be found
- Valid plaintext-ciphertext pairs do not leak the key
- Diffusion & Confusion

AES – Advanced Encryption Standard (NIST 2001)

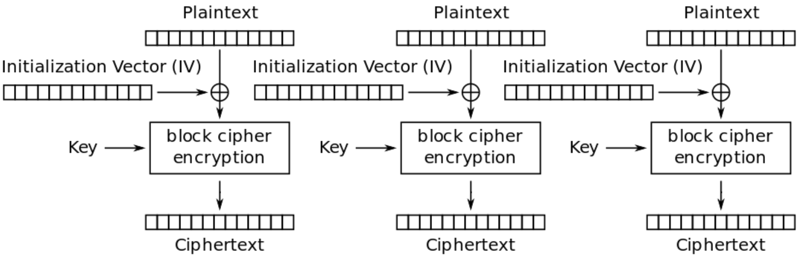
- 16 byte (128 bit) block size
- key sizes – 128/192/256 bits

Electronic Codebook (ECB) mode





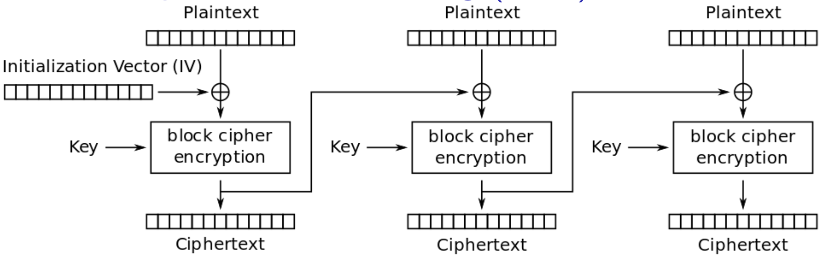
Initialization Vector (IV)

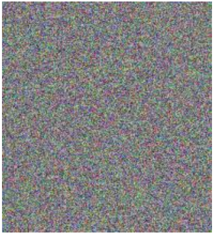


- On encryption XOR plaintext with random IV
- IV must not be secret
- IV must be stored along with ciphertext
- On decryption XOR plaintext with IV

Problem? Ciphertext two times larger than the plaintext

Cipher Block Chaining (CBC) mode





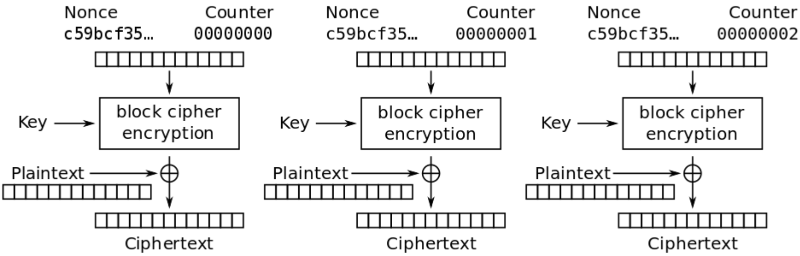
- Serial writes
- Parallel reads
- What about integrity (malleability)?

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**

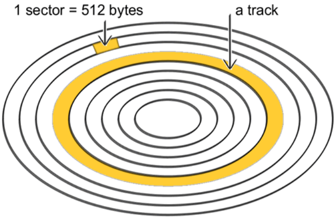
Padding is added even if the plaintext occupies full block.

Counter (CTR) mode



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

Disk Encryption



- Encrypt whole disk using CBC?
  - Every sector is encrypted separately
  - Use sector number as IV
- Password change without disk reenc
  - Master password is used to encrypt data
  - Master password is stored encrypted in disk header
  - User's password decrypts master password
  - Enterprise solutions have several encryptions of master key
- Prevent meaningful malleability attacks
  - MAC is not an option
  - XTS mode

Password-based encryption

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

- Use hash of the password
  - Increases security level by one bit
- Use salt as an addition to password
  - Prevents precomputation attacks
- Use iterated hash to slow down brute-force
  - Adds arbitrary number of operations to brute-force

PBKDF2

Password Based Key Derivation Function 2

```
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)
- kLen – desired length of the derived key

For example, WPA2 uses:

```
key = PBKDF2(HMAC-SHA1, passphrase, ssid, 4096, 256)
```

Truecrypt uses PBKDF2 with 2000 iterations

Task: Password-based file encryption

Implement utility that encrypts and decrypts files using a password:

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

$ ./aes.py -encrypt plain plain.enc
[+] Benchmark: 34856 PBKDF2 iterations in 1 second
[?] Enter password: asd

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

Encryption parameters are stored ASN.1 DER-encoded as a header of the ciphertext file.

Task: Password-based file encryption

```
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,
}
```

```
$ dumpasn1 plain.enc
  0  86: SEQUENCE {
  2  18:   SEQUENCE {
  4   8:     OCTET STRING 7D 64 F8 30 70 5B AE 73
 14   3:     INTEGER 34856
 19   1:     INTEGER 36
       :     }
 22  29:   SEQUENCE {
 24   9:     OBJECT IDENTIFIER aes128-CBC (2 16 840 1 101 3 4 1 2)
 35  16:     OCTET STRING 03 C3 62 AC 25 C2 25 48 E4 B8 11 38 25 D5 2C 25
       :     }
 53  33:   SEQUENCE {
 55   9:     SEQUENCE {
 57   5:       OBJECT IDENTIFIER sha1 (1 3 14 3 2 26)
 64   0:       NULL
       :       }
 66  20:     OCTET STRING 85 DF 81 4E C2 32 2A CC C8 BC 4B 36 C5 30 46 2C 4A F1 29 15
       :     }
       :   }
Warning: Further data follows ASN.1 data at position 88.
```

Task: Test cases

```
$ echo -n "hello world" > plain
$ ./aes.py -encrypt plain plain.enc
[+] Benchmark: 35229 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 \x01\x01\x02\x02" > plain
$ ./aes.py -encrypt plain plain.enc
[+] Benchmark: 34856 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 01 01 02 02  |hello world ....|
00000010

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

```
$ wget https://bitbucket.org/appcrypto/2018/raw/master/04/big.enc
$ ./aes.py -decrypt big.enc big
[?] Enter password: bigfilepassword
$ openssl dgst -sha1 big
SHA1(big)= 34edb7d89a791969d710283c7464a80fe2e39249
```

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 = datetime.datetime.now()
...
stop = datetime.datetime.now()
time = (stop-start).total_seconds()
```

- Use the default PBKDF2 (HMAC-SHA1) to obtain 36 bytes
  - `PBKDF2(password, salt, 36, iter)`
  - Use first 16 bytes as AES-128 key
  - Next 20 bytes as HMAC-SHA1 key
- Generate IV (16 bytes) and salt (8 bytes) randomly
- Implement CBC mode using *pure* AES-128 (ECB mode)

```
cipher = AES.new(key_aes)
cipher.encrypt(plaintext_block)
cipher.decrypt(ciphertext_block)
```

- Use PKCS#5 padding
- Read DER header by parsing length bytes
- Process plaintext/ciphertext in 512 byte chunks
- Verify HMAC-SHA1 *before* starting decryption

Other key derivation functions

The attacker should not be able to gain advantage by optimizing derivation more than the legitimate user.

- PBKDF2 (attackable using GPU, ASIC, FPGA)
- bcrypt (attackable using FPGA)
- scrypt (uses memory bound cryptographic functions)

Side channel attack

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

- Function vulnerable to timing attack
  - Comparison stops on first incorrect byte
    - password 'aaaaaa' – 1ms
    - password 'baaaaa' – 1ms
    - password 'qaaaaa' – 2ms
    - password 'qwaaaa' – 3ms
    - password 'qweaaa' – 4ms
    - password 'qweraa' – 5ms
- Using `sleep(random())` before `return` will not help
- Constant-time string comparison function needed

```
def is_equal(a, b):
    result = 0
    for x, y in zip(a, b):
        result |= ord(x) ^ ord(y)
    return result == 0
```

Your HW 03 (hmac.py) solution is vulnerable to timing attack

Questions

- How block cipher works (takes as an input, returns)?
- What happens to ciphertext if single plaintext or key bit is changed?
- Why encrypting every block of the file independently is not secure?
- Why do we apply initialization vector (IV) to plaintext block?
- How to provide integrity of ciphertext?
- When should we use stream cipher and when block cipher?
- How to convert short password to 128-bit encryption key?
- What is side-channel vulnerability?