



Angelo van der Sijpt
@\_angelos

https://github.com/angelos/javacrypto

### abc

UTF-8

011000 010110 001001 100011 binary

MIIDsjCCApqgAwI BAYTAk5MMRAwDgY dW1pbmlzMQ8wDQY BAMTCWV2ZXJ5LmR CzAJBgNVBAYTAk5

YWJj

Base64

61 62 63



# Hashing





Quick



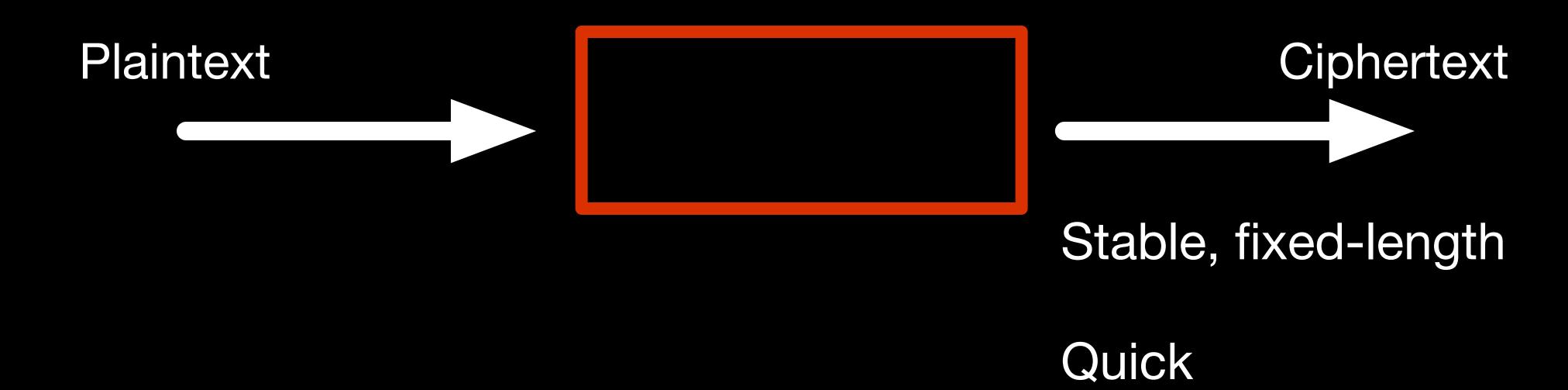
```
byte[][] data = new byte[][]{"Who lives in a pineapple under the sea?".getBytes(),
                   "Absorbent and yellow and porous is he!".getBytes()};
           System.out.println(String.format("Input data:\n %s\n %s", Util.toHex(data[0]), Util.
 5
                              toHex(data[1])));
 6
           <u>// Most ha</u>shes in Java a (1) created, (2) updated, and (3) asked for a result.
 8
           CRC32 crc = new CRC32();
           crc.update(data[0]);
10
           crc.update(data[1]);
           long checksum = crc.getValue();
12
           System.out.println(String.format("Checksum:\n %s", Long.toHexString(checksum)));
13
```

```
Input data:
   57686F206C6976657320696E20612070696E656170706C6520756E64657220746865207365613F
   4162736F7262656E7420616E642079656C6C6F7720616E6420706F726F757320697320686521
Checksum:
   dd996490
```



# CRC32 Checksum





Pre-image resistant lumins

Conversing worlds

Collision resistant

```
byte[][] data = new byte[][] {"Who lives in a pineapple under the sea?".getBytes(),
                                        "Absorbent and yellow and porous is he!".getBytes()};
           System.out.println(String.format("Input data:\n %s\n %s", Util.toHex(data[0]), Util.
 5
                               toHex(data[1])));
 6
           MessageDigest sha;
 8
           try {
               sha = MessageDigest.getInstance("SHA-256")
10
           catch (NoSuchAlgorithmException e) {
11
12
               // Won't happen.
13
               return;
14
15
           sha.update(data[0]);
16
           sha.update(data[1]);
17
           byte[] finalHash = sha.digest();
18
```

#### MessageDigest Algorithms

The algorithm names in this section can be specified when generating an instance of MessageDig

Description
The MD2 message digest algorithm as defined in RFC 1319.
The MD5 message digest algorithm as defined in RFC 1321.
Hash algorithms defined in the FIPS PUB 180-4.
Secure hash algorithms - SHA-1, SHA-224, SHA-256, SHA-384, SHA-512 - for correpresentation of electronic data (message). When a message of any length less th 1, SHA-224, and SHA-256) or less than 2^128 (for SHA-384 and SHA-512) is input the result is an output called a message digest. A message digest ranges in length depending on the algorithm.

#### Input data:

57686F206C6976657320696E20612070696E656170706C6520756E64657220746865207365613F 4162736F7262656E7420616E642079656C6C6F7720616E6420706F726F757320697320686521

System.out.println(String.format("Hash:\n %s", Util.toHex(finalHash)));

Hash:

19

82CC75DC672B5D2C80C3DAA69F08776FDAB6EE0B09FFFA101BF07431D68F7F33

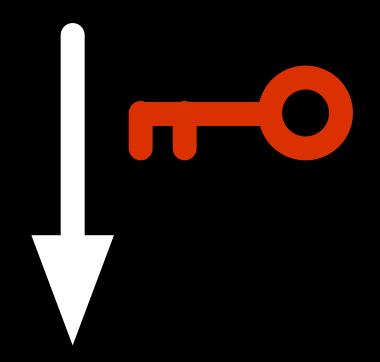


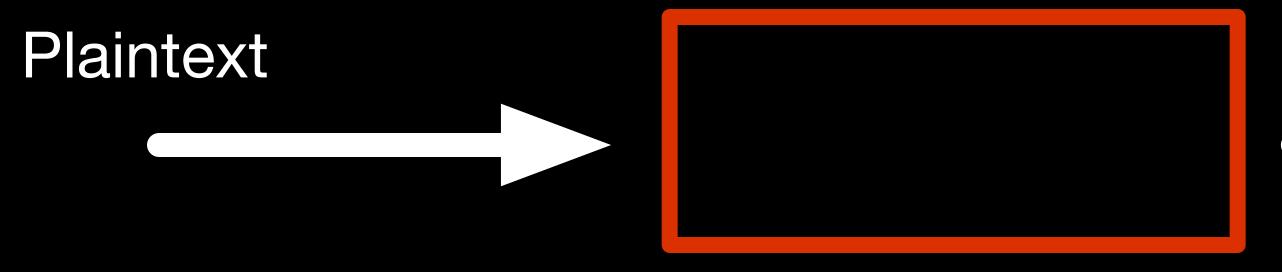
Hash for 00000000: DF3F619804A92FDB4057192DC43DD748EA778ADC52BC498CE80524C014B81119 Hash for 00000001: B40711A88C7039756FB8A73827EABE2C0FE5A0346CA7E0A104ADC0FC764F528D



# SHA Cryptographic hash







Ciphertext

Stable

Quick

Collision resistant

Pre-image resistant

Authenticated



```
byte[][] data = new byte[][]{"Who lives in a pineapple under the sea?".getBytes(),
                                         "Absorbent and yellow and porous is he!".getBytes()};
            System.out.println(String.format("Input data:\n %s\n %s",
                    Util.toHex(data[0]), Util.toHex(data[1])));
            Mac mac = Mac.getInstance("HmacSHA256");
  6
  8
            Key secret = new SecretKeySpec("nauticalnonsense".getBytes(), "HmacSHA256");
            mac.init(secret);
 10
            mac.update(data[0]);
            byte[] digest = mac.doFinal(data[1]);
 12
 13
            System.out.println(String.format("Hash for password \"nauticalnonsense\":\n %s",
 14
                    Util.toHex(digest)));
 15
Input data:
  57686F206C6976657320696E20612070696E656170706C6520756E64657220746865207365613F
  4162736F7262656E7420616E642079656C6C6F7720616E6420706F726F757320697320686521
Hash for password "nauticalnonsense":
  212E0851609770C95991CD4F2EFCE7C428109F59F9F3AED1DFE7D9FCA5AF22AA
```

```
Hash for password "floplikeafish": 2E4CA244C7439F39F8071841171265B292A90280222D7FD6987BBE31AB650484
```

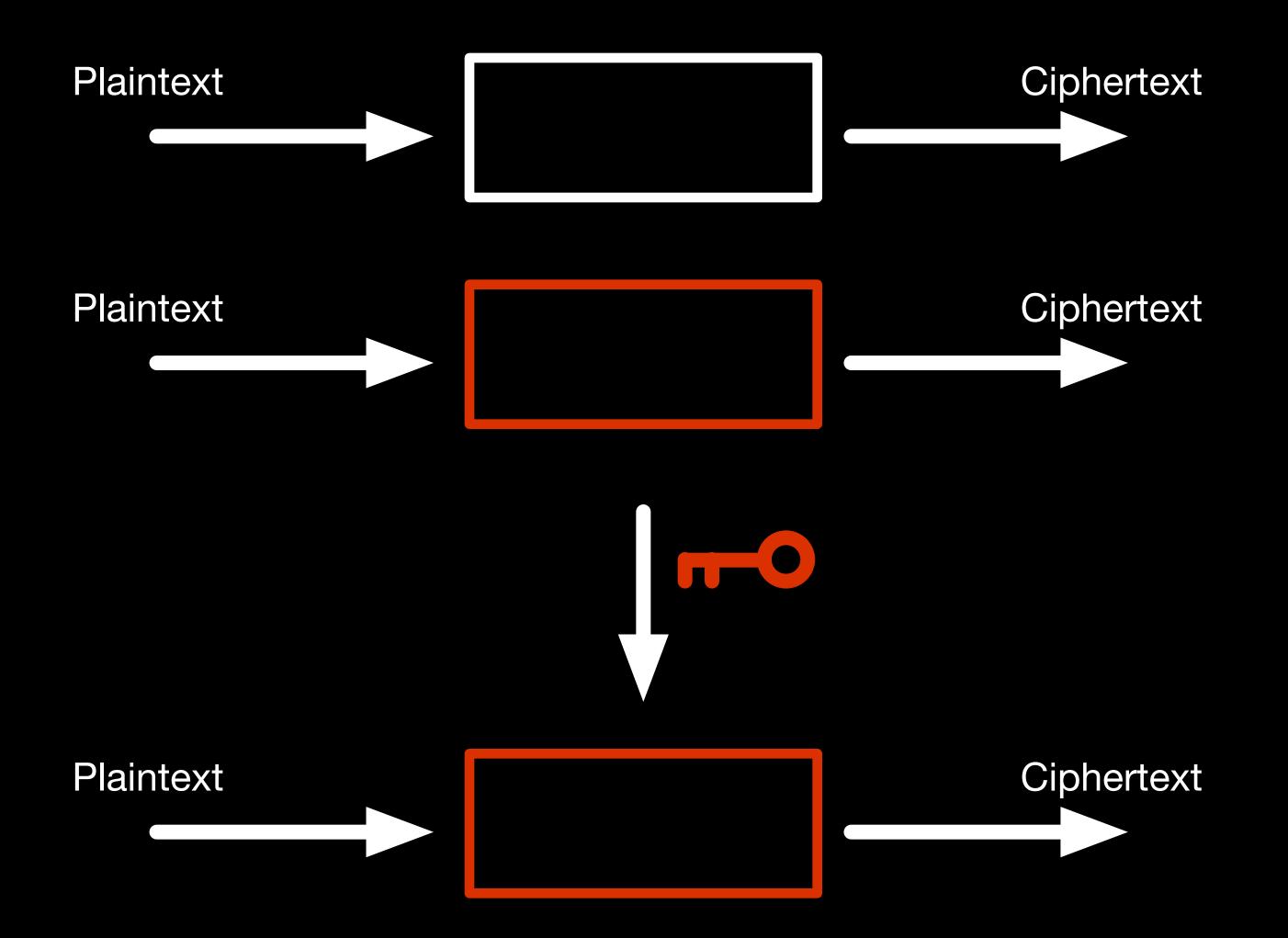


# Keyed hash (Hashed Message Authentication Code)



## Hashing





Checksums stable quick

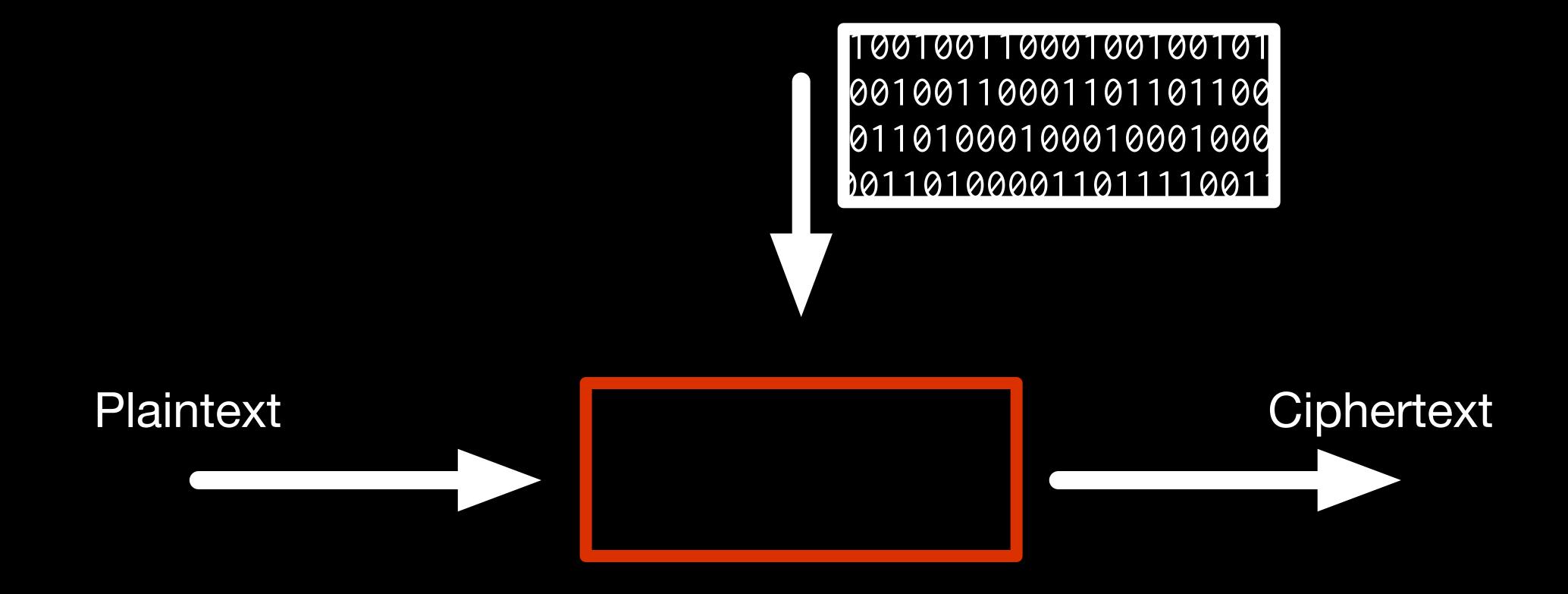
Cryptographic hashes collision resistant pre-image resistant

Keyed hashes authentication



# Keys







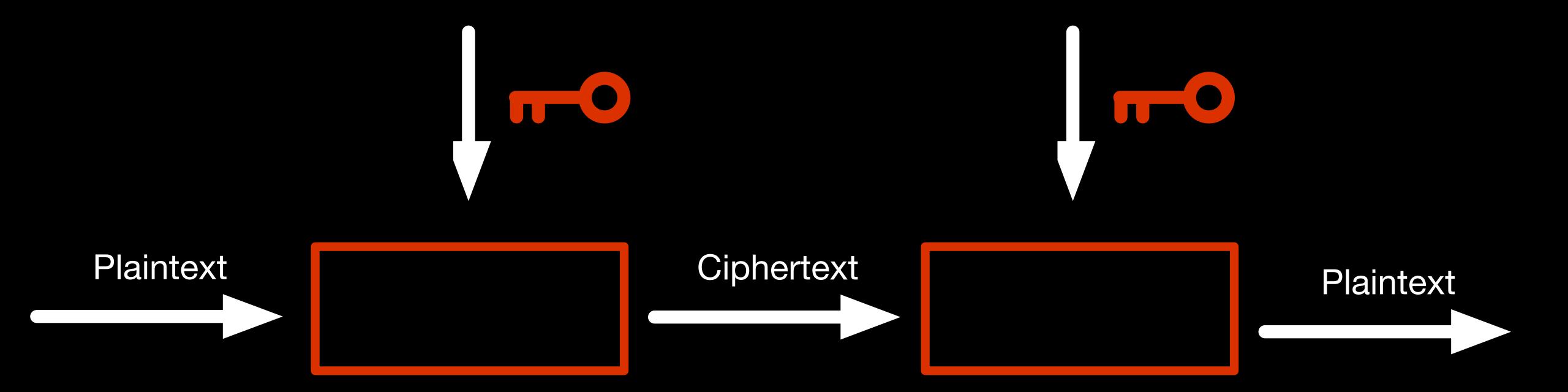
```
KeyGenerator aesGenerator = KeyGenerator.getInstance("AES");
            javax.crypto.SecretKey key = aesGenerator.generateKey();
            System.out.println(String.format("Generated secret key: %s",
                    Util.toHex(key.getEncoded())));
Generated secret key: F2B8CDAF2E949210F45CE2E67A0A0A0B
            Key key = new SecretKeySpec(
                    new byte[] {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                0 \times 00, 0 \times 00},
                    "AES");
            System.out.println(String.format("secret key from 0 input: %s",
  6
                               Util.toHex(key.getEncoded()));
SecretKeyFactory skf = SecretKeyFactory.getInstance( "PBKDF2WithHmacSHA1" );
            PBEKeySpec spec = new PBEKeySpec( "nauticalnonsense".toCharArray(), // the password
                    new byte[] \{0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08\}, // a salt
  5
                    50, // iteration count, hardness of the function
                    128); // key length, should match the kind of encryption you're trying to do
  6
            Key key = skf.generateSecret(spec);
            System.out.println(String.format("secret key from \"nauticalnonsense\" using PBKDF2 with
  8
                               HmacSHA256: %s", Util.toHex(key.getEncoded())));
```

secret key from "nauticalnonsense" using PBKDF2 with HmacSHA256: 367CDD70CF460068316FB0BA94DBE0CC



# Symmetric encryption







# AES (Rijndael) Block cipher



128 - 512b key

128b Plaintext

1001100011011011

1010001000100010

0100110001001001

Block

size





```
byte[] data = "1234567890123456".getBytes(); // exactly 16 bytes or 128 bits long.
           Cipher cipher = Cipher.getInstance("AES/ECB/NoPadding");
           cipher.init(Cipher.ENCRYPT_MODE, key);
           byte[] encrypted = cipher.doFinal(data);
 6
           System.out.println(String.format("Encrypted version of \"%s\" with AES key 0: %s",
                   new String(data),
 8
                   Util.toHex(encrypted)));
 9
10
           cipher.init(Cipher.DECRYPT_MODE, key);
           byte[] decrypted = cipher.doFinal(encrypted);
11
12
13
           System.out.println(String.format("Decrypted: %s", new String(decrypted)));
```

Encrypted version of "1234567890123456" with AES key 0: 41F813ECA9DDD189F7FF3280ADA72C8A Decrypted: 1234567890123456



```
try {
    System.out.println("Trying to encrypt 15 bytes...");
    Cipher cipher = Cipher.getInstance("AES/ECB/NoPadding");
    cipher.init(Cipher.ENCRYPT_MODE, key);
    cipher.doFinal("123456789012345".getBytes()); // that's one byte too few
}
catch (IllegalBlockSizeException e) {
    e.printStackTrace(System.out);
}
```

```
Trying to encrypt 15 bytes...
javax.crypto.IllegalBlockSizeException: Input length not multiple of 16 bytes
 at com.sun.crypto.provider.CipherCore.finalNoPadding(CipherCore.java:1016)
 at com.sun.crypto.provider.CipherCore.doFinal(CipherCore.java:984)
 at com.sun.crypto.provider.CipherCore.doFinal(CipherCore.java:824)
 at com.sun.crypto.provider.AESCipher.engineDoFinal(AESCipher.java:436)
 at javax.crypto.Cipher.doFinal(Cipher.java:2121)
 at crypto4.encryption1.SymmetricCrypto.blockCipher2(SymmetricCrypto.java:75)
 at crypto4.encryption1.SymmetricCrypto.main(SymmetricCrypto.java:20)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
 at java.lang.reflect.Method.invoke(Method.java:483)
 at com.intellij.rt.execution.application.AppMain.main(AppMain.java:147)
```

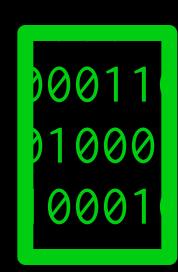




128 - 512b key

### Padding

10001101 00010010 011000100



128b Plaintext



128b Ciphertext



## PKCS5 padding

\*Public Key Cryptography Standard



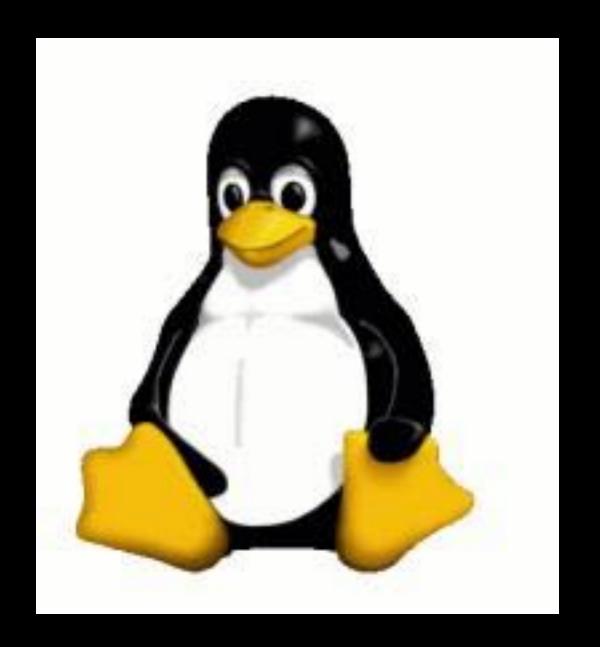
```
Cipher cipher = Cipher.getInstance("AES/ECB 'PKCS5Padding');
byte[] data = "123456789012345".getBytes();
cipher.init(Cipher.ENCRYPT_MODE, key);
byte[] encrypted = cipher.doFinal(data);
System.out.println(String.format("Encrypted version of \"%s\" with AES key 0: %s",
new String(data),
Util.toHex(encrypted)));
```

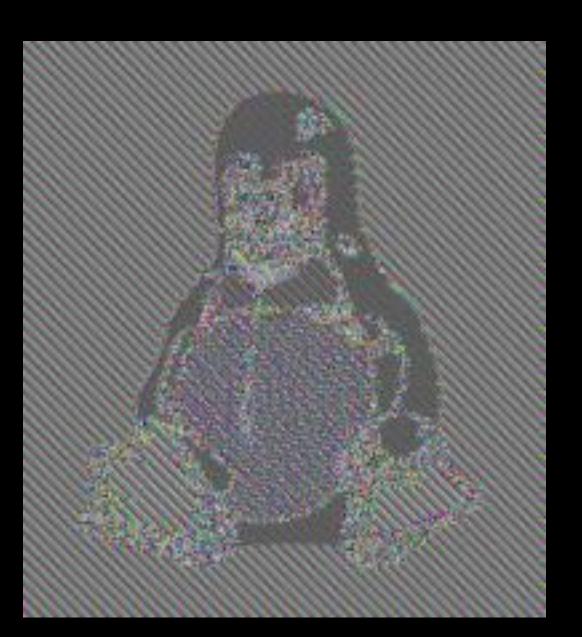
Encrypted version of "123456789012345" with AES key 0: 9BF579AE2C77467D96E4A7CA2E0ED95C

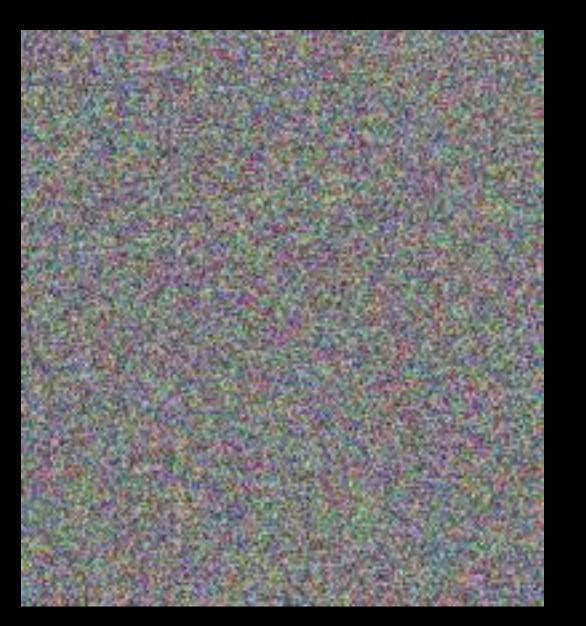


```
Cipher cipher = Cipher.getInstance("AES/ECB/NoPadding");
byte[] data = "1234567890123456123456789012345612345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567
```

Encrypted version of "1234567890123456123456789012345612345678901234561234567890123456" with AES key 0: 41F813ECA9DDD189F7FF3280ADA72C8A 41F813ECA9DDD189F7FF3280ADA72C8A 41F813ECA9DDD189F7FF3280ADA72C8A 41F813ECA9DDD189F7FF3280ADA72C8A 41F813ECA9DDD189F7FF3280ADA72C8A

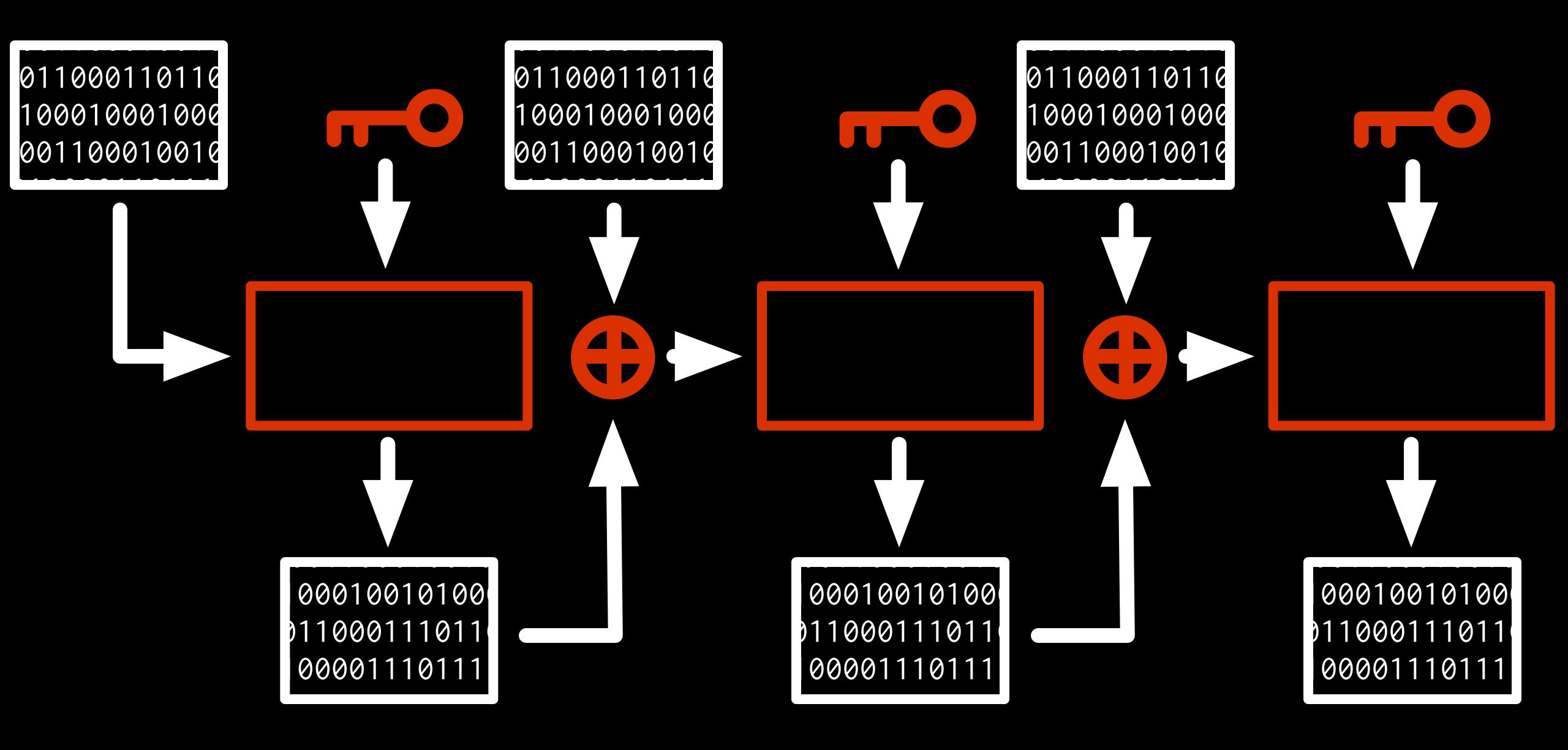








https://en.wikipedia.org/wiki/Block\_cipher\_mode\_of\_operation#ECB

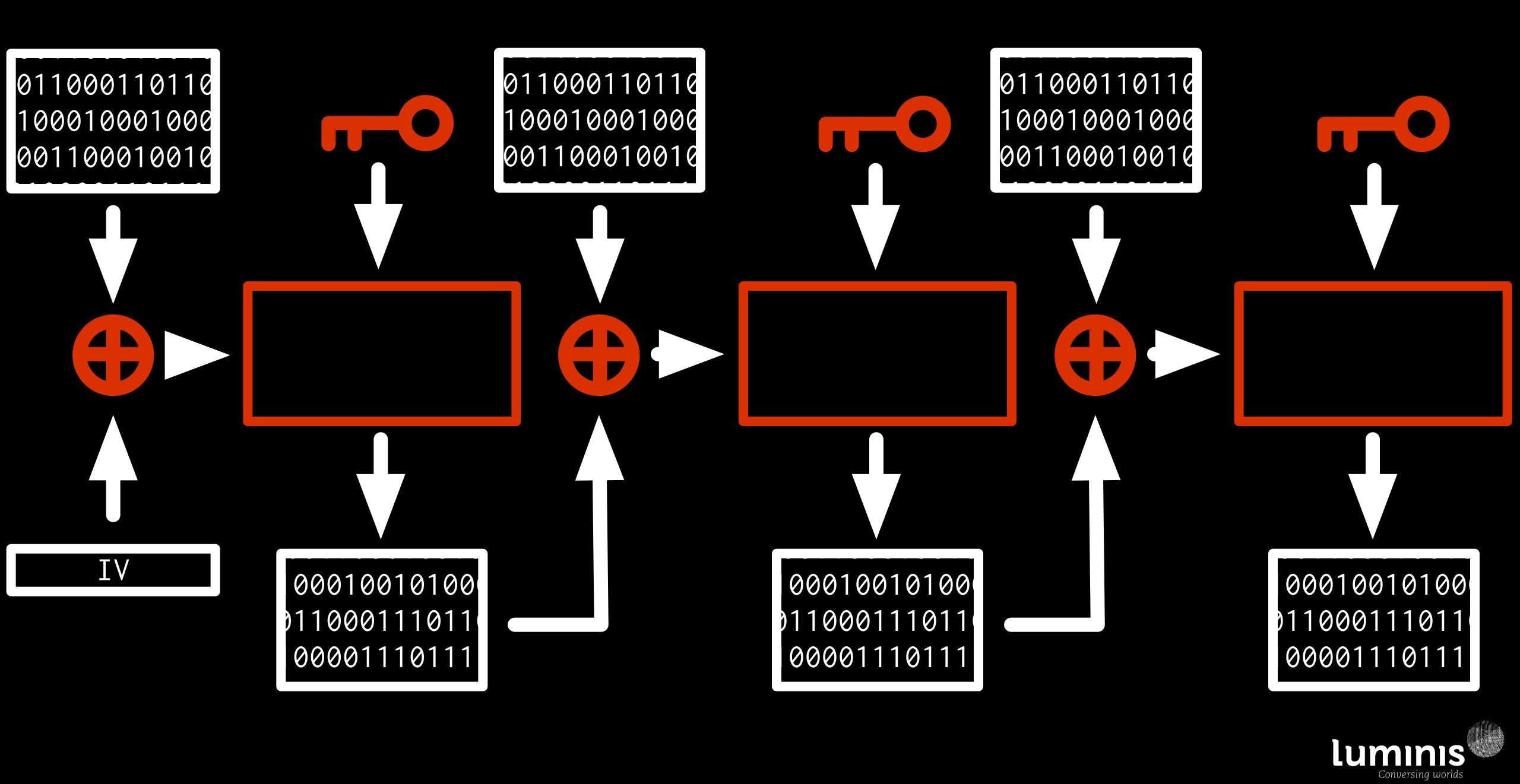


# Cipher Block Chaining



```
Cipher cipher = Cipher.getInstance("AES CBC NoPadding");
            byte[] data = "1234567890123456123456789012345612345678901234561234561234567890123456"
                     .getBytes(); // just four times the same text
            cipher.init(Cipher.ENCRYPT_MODE, key);
            byte[] encrypted = cipher.doFinal(data);
            System.out.println(String.format("Encrypted version of \"%s\" with AES key 0:\n%s",
                    new String(data),
                    breakStringAt(Util.toHex(encrypted), 32)));
Encrypted version of "1234567890123456123456789012345612345678901234561234561234567890123456" CBC:
1F3C8C921D5278C929F965D964FE3F4C
68DAEDEEDA5E1B825DCAB2EBADF9369A
518ADB2208C0741CA3215DE137F02604
881758BBF2E52FAF57E902C6BF27B46B
Encrypted version of "1234567890123456123456789012345612345678901234561234561234567890123456" CBC:
34DA8ED866CF25078D087AC5D22F93FE
7B334FA40B5FA539FA393BF5FBEE7EAA
0ABA95EAA72758F35609C75CE52567DD
A1FB1D781F7F01AA65389021B43C4A84
Encrypted version of "12345678901234561234567890123456123456789012345678901234561234567890123456" CBC:
D5C12E6B8B89AE91D04BAFE4E9E18273
934AB4551F916171B58E7FF2C69DF9C5
C0BAF06EF4A68ABFD662DA0421E44AA9
C55FDC4D12D9B2153E2CCF1876DDCBD6
                                                                                              luminis
```

```
Cipher cipher = Cipher.getInstance("AES/CBC/NoPadding");
            byte[] data = "12345678901234561234567890123456123456789012345678901234561234567890123456"
                     .getBytes();
            cipher.init(Cipher.ENCRYPT_MODE, key);
  5
            byte[] encrypted = cipher.doFinal(data);
            System.out.println(String.format("Encrypted version of \"%s\" CBC:\n%s",
  6
                     new String(data), breakStringAt(Util.toHex(encrypted), 32)));
  8
            try {
                System.out.println("Decrypting...");
 10
                cipher.init(Cipher.DECRYPT_MODE, key);
 12
            catch (InvalidKeyException e) {
 13
 14
                e.printStackTrace(System.out);
 15
Encrypted version of "123456789012345612345678901234561234567890123456789012345678901234567890123456" CBC:
500327C5C8AF0D51FA4EF2314701FE06
E1C58D710196E73E9FD8E91E81E57D55
910C76B3F5D9EE86C365CA963D0FC4C9
DFA8CBC6FDFFDF2AEF8C5C97BFE4513D
Decrypting...
java.security.InvalidKeyException: Parameters missing
  at com.sun.crypto.provider.CipherCore.init(CipherCore.java:460)
  at com.sun.crypto.provider.AESCipher.engineInit(AESCipher.java:307)
  at javay crypto Cipher init(Cipher java:1226)
```



### Initialization Vector

The initialization vector in WEP is a 24-bit field, which is sent in the cleartext part of a message. Such a small space of initialization vectors *guarantees* the reuse of the same key stream. A busy access point, which constantly sends 1500 byte packets at 11Mbps, will exhaust the space of IVs after 1500\*8/(11\*10^6)\*2^24 = ~18000 seconds, or 5 hours. (The amount of time may be even smaller, since many packets are smaller than 1500 bytes.) This allows an attacker to collect two ciphertexts that are encrypted with the same key stream and perform statistical attacks to recover the plaintext. Worse, when



```
Cipher cipher = Cipher.getInstance("AES/CBC/NoPadding");
           byte[] data = "1234567890123456123456789012345612345678901234561234561234567890123456"
                   .getBytes();
 5
           IvParameterSpec iv = new IvParameterSpec(
 6
                   new byte[] \{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                               0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F);
 8
           cipher.init(Cipher.ENCRYPT_MODE, key, iv);
           byte[] encrypted = cipher.doFinal(data);
10
11
           System.out.println(String.format("Encrypted version of \"%s\" with AES key 0:\n%s",
                   new String(data),
12
                   breakStringAt(Util.toHex(encrypted), 32)));
13
14
           cipher.init(Cipher.DECRYPT_MODE, key, iv);
15
           System.out.println(String.format("Decrypted:\n%s", new String(cipher.doFinal(encrypted))));
16
```

```
Encrypted version of "12345678901234561234567890123456123456789012345612345678901234567890123456" with AES key 0: 466243EC0D98F0AD82AE3263F7A36D0D 31738A94186E76120011DFF92113D73B 8E387C7DF0B22428ED6BD7B9E17A4F06 6C034BFAAF6387BB0F2BC0FCC492F082
```

#### Decrypted

1234567890123456123456789012345612345678901234561234567890123456



## Symmetric crypto

Block size

Padding

Cipher modes
ECB
CBC

Initialization Vector









### SECURITY NONEXPERTS' TOP ONLINE SAFETY PRACTICES



1. USE ANTIVIRUS SOFTWARE



1. INSTALL SOFTWARE UPDATES

2. USE STRONG PASSWORDS



2. USE UNIQUE PASSWORDS





2

3. USE TWO-FACTOR AUTHENTICATION





4. USE STRONG PASSWORDS





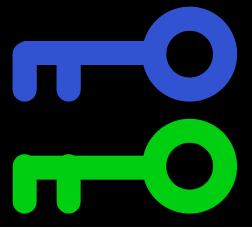


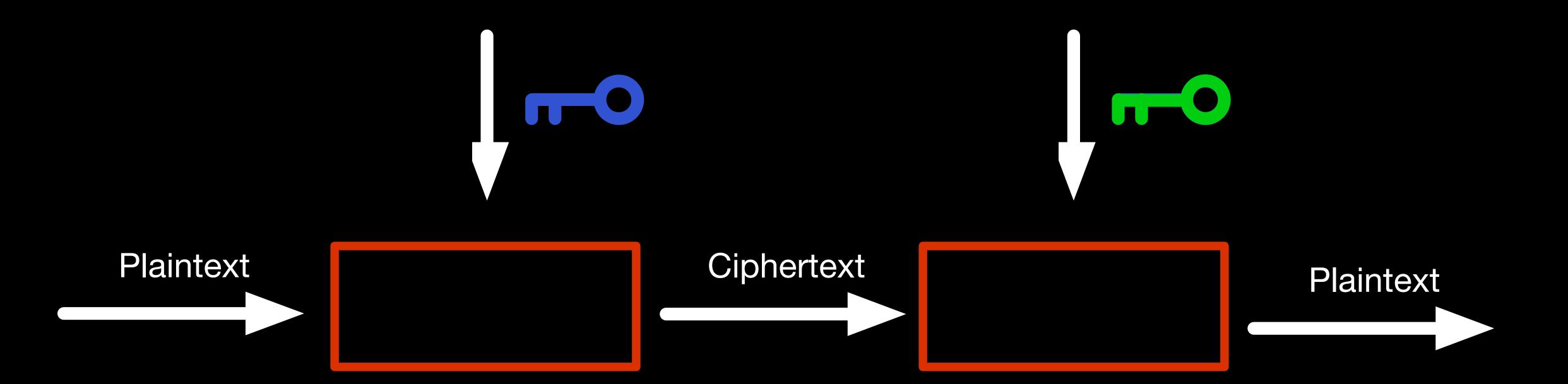
5. USE A PASSWORD MANAGER



# Asymmetric encryption









```
Generated a private key:

30820
4BD020100300D06092A864886F70D0101010500048204A7308204A30201000282010100E368C1A59F49F8517A39F4724407
45705E902E104792F5A1CB41D4C1D237FF4AF10254A93E7A9F56F1654761C8636D4A5C1384663950DFABC40B8A814B7F4966D9CA
F960067C6735A7D49F8A2135CA018C41FDC555C25B53715CEF968B553FD287A6E0640C94D98166E4EF269B562E558BA65B...

Generated a public key:
308201223300D066092A864886E70D01010105000382010E003082010A0282010100E368C1A59E49E8517A39E4724407B745705E90
```

3082<mark>(</mark>122300D06092A864886F70D01010105000382010F003082010A0282010100E368C1A59F49F8517A39F4724407B745705E90 104792F5A1CB41D4C1D237FF4AF10254A93E7A9F56F1654761C8636D4A5C1384663950DFABC40B8A814B7F4966D9CA69F960067C 35A7D49F8A2135CA018C41FDC555C25B53715CE<u>F968B553FD287A6E0640C94D98166E4EF269B562E558BA65BCC564CB58D...</u>



```
1. Shell
MacBook-Pro-4:~ angelos$ dumpasn1 key.tmp
  0 1215: SEQUENCE {
       1: INTEGER 0
      13: SEQUENCE {
       9:
              OBJECT IDENTIFIER rsaEncryption (1 2 840 113549 1 1 1)
 20
       0:
              NULL
            OCTET STRING, encapsulates {
 22 1193:
 26 1189:
              SEQUENCE {
  30
      1:
                INTEGER 0
                INTEGER
 33 257:
                  00 CF F4 90 D2 45 74 BC 5A F8 B4 A5 E9 69 27 FF
                  77 37 0E 38 B1 B6 D5 52 62 12 CA B3 B3 E4 02 D8
                  A4 FE E7 2E 1B F1 C0 89 37 E8 21 29 0D EE D3 DA
                  27 1A 6D 93 41 F3 9B 87 F6 7D EE A0 98 5D BE C1
                  06 B4 37 C9 2F 96 C9 EB 33 4E 0C 12 A5 4C 0C F2
                  65 CC C8 F8 93 5E 01 44 D4 15 76 CE AF FD F8 09
                  C2 C6 53 F1 C7 59 35 90 D6 A7 74 5F DE 08 B0 22
                  CF 65 D9 D6 DD 29 05 19 14 64 00 1E 5D 96 ED 4E
                          [ Another 129 bytes skipped ]
       3:
 294
                INTEGER 65537
 299
     257:
                INTEGER
                  00 C4 2A 02 96 A7 78 27 D9 74 B4 0B B1 B3 45 4C
                  C1 AB 48 9A 08 61 DC B6 DA D6 B9 29 6C EF 10 14
                  47 08 41 11 08 C1 32 8A FE 16 D5 79 01 B0 A1 5D
                  2E F7 CA 17 57 E6 31 77 BE F8 2A 2A 89 B0 1C A9
                  BE E6 2F 9D 73 6B BD 4B 45 D9 40 A0 32 17 1C 3B
                  67 F6 16 3A 0D E1 66 37 04 0C BF 46 D7 53 2E 30
                  1E F0 B0 EC E8 CC 69 30 18 9C 5B B5 B7 1F E0 EE
                  E5 4F 41 7F 62 9E 38 35 E1 3B 37 B5 6D 39 5A DF
                          [ Another 129 bytes skipped ]
 560 129:
                INTEGER
                  00 F0 C6 2F 4D 06 2D 27 FD 49 35 AD 64 C0 E2 99
                  1F 6C C4 A3 5B 42 FA B3 5B 79 5C 85 F3 D9 25 5C
                  E3 17 4B 93 23 E3 FE 9F A3 ED 45 EC 75 AD 8C 8D
                  98 A9 75 88 3C F7 8F 2C 7D F6 98 A4 A5 3B F5 D5
```

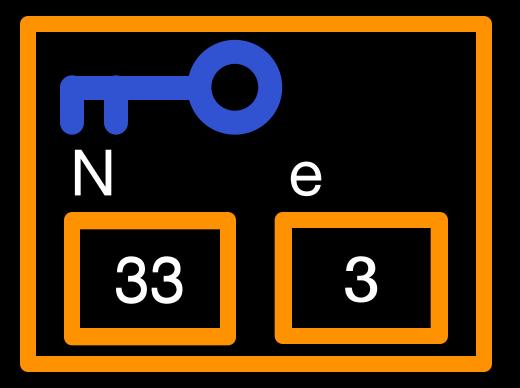
DA 16 A2 2F 0F 83 38 3D DD 67 A6 C2 81 39 4B 2F

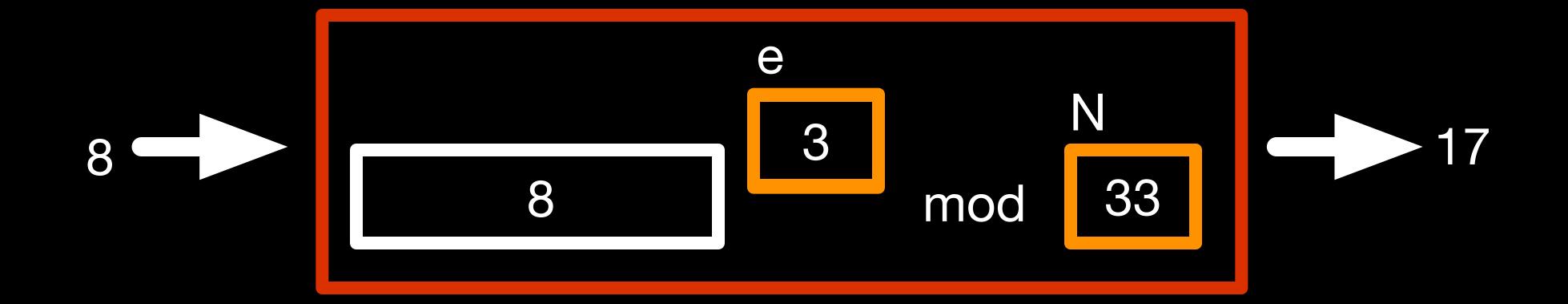
2E 1E E1 66 72 24 49 02 00 04 22 74 E2 EE



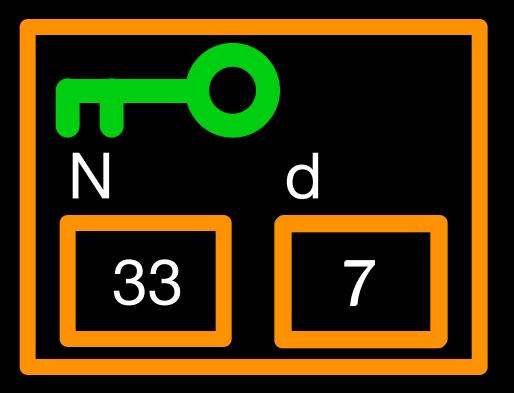
## RSA

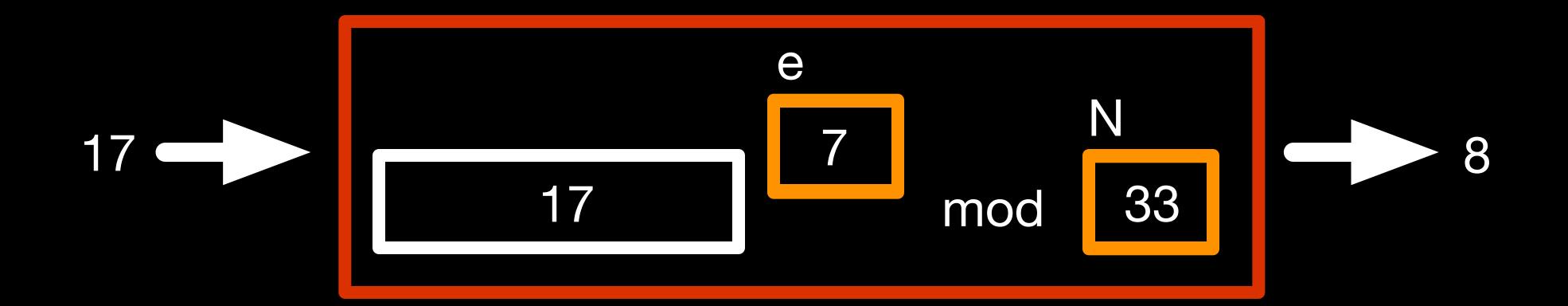














```
byte[] data = "1234567890123456".getBytes();
            Cipher rsa = Cipher.getInstance("RSA");
            rsa.init(Cipher.ENCRYPT_MODE, keyPair.getPublic());
            byte[] encrypted = rsa.doFinal(data);
            System.out.println(String.format("Encrypted version of \"%s\" with generated RSA key:\n%s",
  8
                    new String(data),
                    breakStringAt(Util.toHex(encrypted), 64)));
 10
Generated a private key: 308204BD020100300D06092A864886F70D010101...
Generated a public key: 30820122300D06092A864886F70D010101050003...
Encrypted version of "1234567890123456" with generated RSA key:
3621C9564A74D50D6EA054EBAE6B1E931F64D6C2177ED43A7FF94CC82BC433CD
9D0F75F51A9C84B96DF8617CCFB4278F05557518B64405FB9C06FB97C17660C1
8E6B3671D543489237FACA5685572970C603F97EA094FC9D85BD06D949EDCE02
CB1BE13659BEEF0DA47CDB7568515F96A336DE9F0C0C8705068EDAB480210EDD
723EC690ACC732657363A75F66F5A2ED08AFC7742535A436D00FC0D702DEB741
3172DDD47B306F612DAF649FA59A005CCEC3FAE163F4F08D008F3C496C8DB890
6A17ABE7E8FC3CEADDB5AFA92B2FDBDDAB357931E6B776AB9392EEBC872902BD
9FA7547F55EFE56D6847CB83200BF2DFD682C6656F5CA90758B7D438CDB1B3FA
```

```
byte[] data = "1234567890123456".getBytes();
           Cipher rsa = Cipher.getInstance("RSA");
           rsa.init(Cipher.ENCRYPT_MODE, keyPair.getPublic());
           byte[] encrypted = rsa.doFinal(data);
 6
 8
           System.out.println(String.format("Encrypted version of \"%s\" with generated RSA key:\n%s",
                   new String(data),
                   breakStringAt(Util.toHex(encrypted), 64)));
10
           rsa.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());
12
13
           byte[] decrypted = rsa.doFinal(encrypted);
           System.out.println(String.format("Decrypted: %s",
14
                   new String(decrypted)));
15
```

Encrypted version of "1234567890123456" with generated RSA key: B38687399634D817C1C57131D401832E2CCEA90CF9069AEBC4EBDF4C98D66C68 CA136DCE8D1E7297DAC8705E6EFFFAD6B57CE85AB5F86C46938918D9BCD28712 4B3F1EC2DD2ECF4E2EB194B6528402D003524D810521804B0D0CCE3F2D7D9A60 2A3F8AFFA529BF845FBEAB5B18E49785A516B08795F60E6C9AF864615DB5A896 F05548AB3F3FD747ACB2E0CEFC22C63CB203DFD31F3697629A37CD1C4F2655FE 5CBF5256F0E05424051CB5B21CCFC1A1F3F96F8FA344D81505A28159849687F2 3B4001F31E6E0D0ED233F54BED9EB12281F641D00F1D5F098CA59F294AEB3F43 F495BE08FA9CB0319C9CB91B95BC994D11E3042A86A71193F1B2D669BB78CCD9

Decrypted: 1234567890123456



Cipher rsa = Cipher.getInstance("RSA/ECB/PKCS1Padding");



## Asymmetric encryption

Private & public key

Simple but expensive math



## Java Cryptography Architecture



Security.addProvider(new MyProvider());



```
1 Set<String> digests = Security.getAlgorithms("MessageDigest");
2 Set<String> ciphers = Security.getAlgorithms("Cipher");
```



# TLS



Phase 1 Client fello Phase 2 Server identification Diffie Hellman key exchange Phase 3 Client identification Phase 4 Finalization of cipher specs



Bulk encryption Symmetric

## Client

## Server

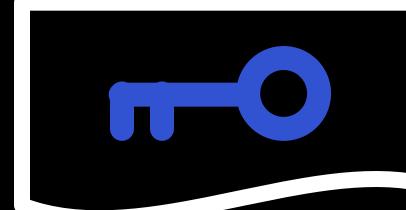
Hello & challenge

Verify certificate, signature, hostname

Hello & certificate & signature

Calculate signature







```
// Initialize the Server Socket
           SSLServerSocketFactory sslServerSocketfactory =
                   (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
           SSLServerSocket sslServerSocket =
4
5
                   (SSLServerSocket) sslServerSocketfactory.createServerSocket(8443);
           SSLSocket sslSocket = (SSLSocket) sslServerSocket.accept();
6
8
           PrintWriter out = new PrintWriter(sslSocket.getOutputStream(), true);
           BufferedReader in = new BufferedReader(
10
                   new InputStreamReader(sslSocket.getInputStream()));
11
12
          String input;
          while ((input = in.readLine()) != null) {
13
              out.println(input);
14
              System.out.println(input);
15
16
17
18
           // IO cleanup omitted for readability
```



### private.key

----BEGIN RSA
MIIEogIBAAKCAQE
nrJK6S2hnFpRFP0
XCIVm5tLQ7/s1QT
yb0H62Pg8p5K70k

### certificate.cer

----BEGIN CERT MIIDsjCCApqgAwI BAYTAk5MMRAwDgY BAMTCWV2ZXJ5LmR CzAJBgNVBAYTAk5

```
1 openssl req \
2 -x509 \
3 -sha256 \
4 -newkey rsa:2048 \
5 -keyout private.key \
6 -out certificate.cer \
7 -subj "/C=NL/O=Luminis/OU=Crypto/CN=every.dev" \
8 -nodes
```



### keystore.p12



```
1 openssl pkcs12 \
2 -export \
3 -nodes \
4 -out keystore.p12 \
5 -inkey private.key \
6 -in certificate.cer \
7 -passin pass:123456 \
8 -passout pass:123456
```

```
1 java \
2 -Djavax.net.ssl.keyStore=keystore.p12 \
3 -Djavax.net.ssl.keyStorePassword=123456 \
4 -cp src \
5 crypto6.ssl1.SSL
```



#### 1. Shell

MacBook-Pro-4:JavaCryptography angelos\$ java \
> -Djavax.net.ssl.keyStore=keystore.p12 \
> -Djavax.net.ssl.keyStorePassword=123456 \
> -cp src \
> crypto6.ssl1.SSL
GET / HTTP/1.1
User-Agent: Wget/1.17.1 (darwin14.5.0)

Accept: \*/\*

Accept-Encoding: identity

Host: localhost:8443 Connection: Keep-Alive



#### 3. Shell

MacBook-Pro-4:JavaCryptography angelos\$ wget https://localhost:8443 --no-check-c ertificate

--2016-11-01 19:40:44-- https://localhost:8443/

Resolving localhost... ::1, fe80::1, 127.0.0.1

Connecting to localhost!::1:8443... connected.

WARNING: cannot verify localhost's certificate, issued by 'CN=every.dev,OU=Crypt o,O=Luminis,C=NL':

Self-signed certificate encountered.

WARNING: certificate common name 'every.dev' doesn't match requested host na me 'localhost'.

HTTP request sent, awaiting response... 200 No headers, assuming HTTP/0.9

Length: unspecified

Saving to: 'index.html'

index.html



137 --.-KB/s



```
1 java \
2 -Djavax.net.debug=all \
3 -Djavax.net.ssl.keyStore=keystore.p12 \
4 -Djavax.net.ssl.keyStorePassword=123456 \
5 -cp src \
6 crypto6.ssl1.SSL
```



# Cipher suites



- 1 TLS\_ECDHE\_ECDSA\_WITH\_AES\_256\_GCM\_SHA384\_P384,
- 2 TLS\_ECDHE\_ECDSA\_WITH\_AES\_256\_CBC\_SHA\_P384,
- 3 TLS\_ECDHE\_RSA\_WITH\_AES\_256\_CBC\_SHA\_P384,
- 4 TLS\_ECDHE\_ECDSA\_WITH\_AES\_256\_CBC\_SHA\_P256,
- 5 TLS\_ECDHE\_RSA\_WITH\_AES\_256\_CBC\_SHA\_P256,
- 6 TLS\_ECDHE\_ECDSA\_WITH\_AES\_128\_GCM\_SHA256\_P384,
- 7 TLS\_ECDHE\_ECDSA\_WITH\_AES\_128\_CBC\_SHA256\_P384,
- 8 TLS\_ECDHE\_ECDSA\_WITH\_AES\_128\_GCM\_SHA256\_P256,
- 9 TLS\_ECDHE\_ECDSA\_WITH\_AES\_128\_CBC\_SHA256\_P256,
- 10 TLS\_ECDHE\_ECDSA\_WITH\_AES\_128\_CBC\_SHA\_P384,
- 11 TLS\_ECDHE\_ECDSA\_WITH\_AES\_128\_CBC\_SHA\_P256,
- 12 TLS\_ECDHE\_RSA\_WITH\_AES\_128\_CBC\_SHA\_P384,
- 13 TLS\_ECDHE\_RSA\_WITH\_AES\_128\_CBC\_SHA\_P256,
- 14 TLS\_ECDHE\_RSA\_WITH\_AES\_128\_CBC\_SHA256\_P384,
- 15 TLS\_ECDHE\_RSA\_WITH\_AES\_128\_CBC\_SHA256\_P256,
- 16 TLS\_DHE\_DSS\_WITH\_AES\_256\_CBC\_SHA256,
- 17 TLS\_DHE\_DSS\_WITH\_AES\_256\_CBC\_SHA256
- 18 TLS\_DHE\_DSS\_WITH\_AES\_128\_CBC\_SHA256,
- 19 TLS\_DHE\_DSS\_WITH\_AES\_128\_CBC\_SHA,
- 20 TLS\_DHE\_DSS\_WITH\_3DES\_EDE\_CBC\_SHA,
- 21 TLS\_RSA\_WITH\_AES\_256\_CBC\_SHA256,
- 22 TLS\_RSA\_WITH\_AES\_256\_CBC\_SHA,
- 23 TLS\_RSA\_WITH\_AES\_128\_CBC\_SHA256,
- 24 TLS\_RSA\_WITH\_AES\_128\_CBC\_SHA,
- 25 TLS\_RSA\_WITH\_3DES\_EDE\_CBC\_SHA,
- 26 SSL\_CK\_DES\_192\_EDE3\_CBC\_WITH\_MD5



TLS\_DHE\_DSS\_WITH\_AES\_256\_CBC\_SHA256

Message authentication
Bulk encryption
Signature
Key exchange

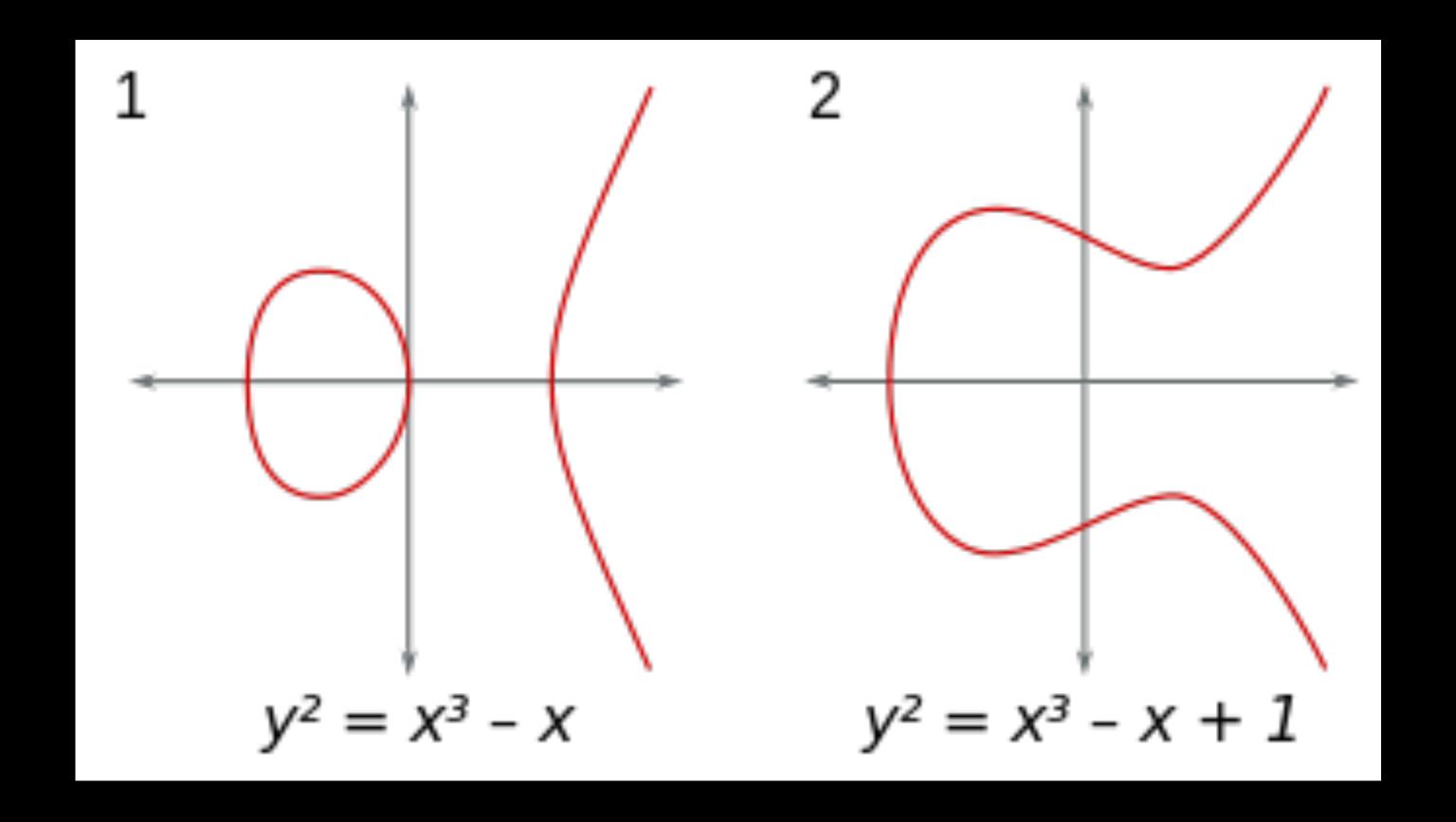
TLS\_ECDHE\_ECDSA\_WITH\_AES\_128\_GCM\_SHA256\_P384 Elliptic curve



## New developments



## Elliptic Curve



Drop-in replacement with (EC)DHE, (EC)DSA



Hashing Checksum Crypto hash Padding HMAC PBKDF

Block size

Cipher modes

ECB

CBC

Initialization Vector

AES

Symmetric crypto Asymmetric crypto TLS

Certificates

RSA

Public vs private key File formats (PEM, P12)

Message authentication

Signature

Bulk encryption

Cipher suites



build your own cryptography

# ALWAYS

use built-in primitives patch your systems rely on experts









https://github.com/angelos/javacrypto

Visit the Luminis booth at 1st floor, ask about our Developer Security Awareness training!