luminis
Conversing worlds

what every  Java developer needs to know about

cryptography

Angelo van der Sijpt
@_angelos

https://github.com/angelos/javacrypto

luminis
Conversing worlds

Home › Malware

Mirai Botnet Infects Devices in 164 Countries

By Ionut Arghire on October 28, 2016

Share 41 G+1 10 Tweet Recommend 42 RSS

Mirai, the infamous botnet used in the recent massive distributed denial of service (DDoS) attacks against Brian Krebs' blog and Dyn's DNS infrastructure, has ensnared Internet of Things (IoT) devices in 164 countries, researchers say.

Home › Virus & Threats

Many Joomla Sites Hacked via Recently Patched Flaws

By Eduard Kovacs on October 31, 2016

Share 21 G+1 5 Tweet Recommend 27 RSS

Less than 24 hours after Joomla released patches for a couple of critical account creation vulnerabilities, researchers noticed that malicious actors had already started exploiting the flaws in the wild.

Home › Data Protection

Australian Red Cross Leaks Blood Donor Data

By Eduard Kovacs on October 28, 2016

Share 20 G+1 8 Tweet Recommend 15 RSS

The Australian Red Cross Blood Service apologised on Friday to donors after one of its third-party service providers inadvertently made accessible a backup database containing the personal details of 550,000 individuals.

Angelo van der Sijpt

Fellow connected devices & security

Running, OCR, Krav Maga

Father of 1.9

@_angelos
angelo.vandersijpt@luminis.eu

luminis
Conversing worlds

abc

UTF-8

011000 010110 001001 100011

binary

YWJj

Base64

61 62 63

Hex

luminis
Conversing worlds

Hashing

Stable, fixed length

Quick

```java
        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])));

        // Most hashes in Java a (1) created, (2) updated, and (3) asked for a result.
        CRC32 crc = new CRC32();
        crc.update(data[0]);
        crc.update(data[1]);
        long checksum = crc.getValue();

        System.out.println(String.format("Checksum:\n  %s", Long.toHexString(checksum)));
```

```
Input data:
  57686F206C6976657320696E20612070696E656170706C6520756E646572207468652073656E3F
  4162736F7262656E7420616E642079656C6C6F7720616E6420706F726F7573206973206861721
Checksum:
  dd996490
```

CRC32
Checksum

Stable, fixed-length

Quick

Collision resistant

Pre-image resistant

luminis
Conversing worlds

```java
        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])));

        MessageDigest sha;
        try {
            sha = MessageDigest.getInstance("SHA-256");
        }
        catch (NoSuchAlgorithmException e) {
            // Won't happen.
            return;
        }

        sha.update(data[0]);
        sha.update(data[1]);
        byte[] finalHash = sha.digest();
        System.out.println(String.format("Hash:\n  %s", Util.toHex(finalHash)));
```

`MessageDigest` Algorithms

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

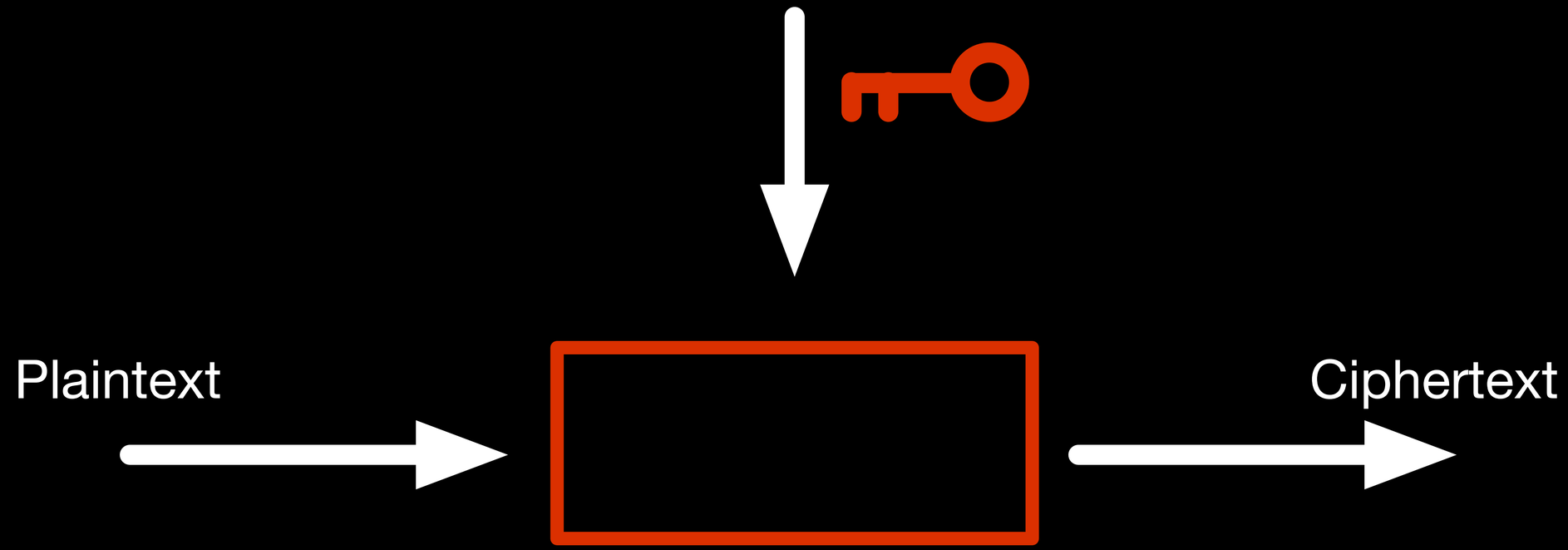
Algorithm Name	Description
MD2	The MD2 message digest algorithm as defined in RFC 1319.
MD5	The MD5 message digest algorithm as defined in RFC 1321.
SHA-1 SHA-224 SHA-256 SHA-384 SHA-512	Hash algorithms defined in the FIPS PUB 180-4. Secure hash algorithms - SHA-1, SHA-224, SHA-256, SHA-384, SHA-512 - for com representation 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:
  57686F206C6976657320696E20612070696E656170706C6520756E646572207468652073656E3F
  4162736F7262656E7420616E642079656C6C6F7720616E6420706F726F7573206973206865216
Hash:
  82CC75DC672B5D2C80C3DAA69F08776FDAB6EE0B09FFFA101BF07431D68F7F33
```

```
1         byte[][] smallData = new byte[][] {{0x00, 0x00, 0x00, 0x00},
2                                            {0x00, 0x00, 0x00, 0x01}};
3
4         MessageDigest sha = MessageDigest.getInstance("SHA-256");
5         System.out.println(String.format("Hash for %s: %s",
6                 Util.toHex(smallData[0]),
7                 Util.toHex(sha.digest(smallData[0]))));
8         System.out.println(String.format("Hash for %s: %s",
9                 Util.toHex(smallData[1]),
10                Util.toHex(sha.digest(smallData[1]))));
```

```
Hash for 00000000: DF3F619804A92FDB4057192DC43DD748EA778ADC52BC498CE80524C014B81119
Hash for 00000001: B40711A88C7039756FB8A73827EABE2C0FE5A0346CA7E0A104ADC0FC764F528D
```

SHA
Cryptographic hash



Stable

Quick

Collision resistant

Pre-image resistant

Authenticated

luminis
Conversing worlds

```java
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");

Key secret = new SecretKeySpec("nauticalnonsense".getBytes(), "HmacSHA256");
mac.init(secret);

mac.update(data[0]);
byte[] digest = mac.doFinal(data[1]);

System.out.println(String.format("Hash for password \"nauticalnonsense\":\n  %s",
        Util.toHex(digest)));
```

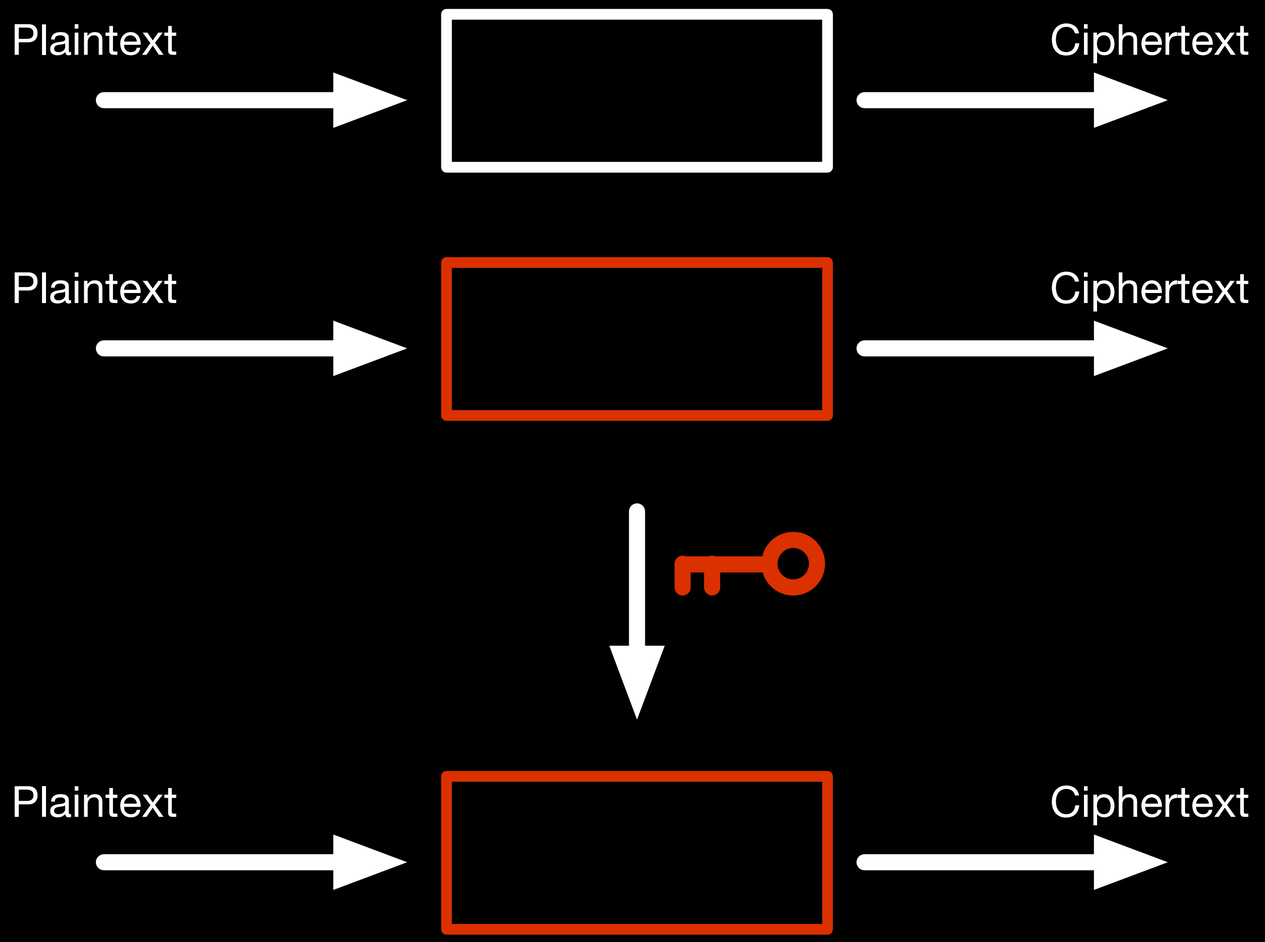
```
Input data:
  57686F206C6976657320696E20612070696E656170706C6520756E646572207468652073656E3F
  4162736F7262656E7420616E642079656C6C6F7720616E6420706F726F7573206973206865216
Hash for password "nauticalnonsense":
  212E0851609770C95991CD4F2EFCE7C428109F59F9F3AED1DFE7D9FCA5AF22AA
```

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

Keyed hash
(Hashed Message Authentication Code)

Hashing

luminis
Conversing worlds

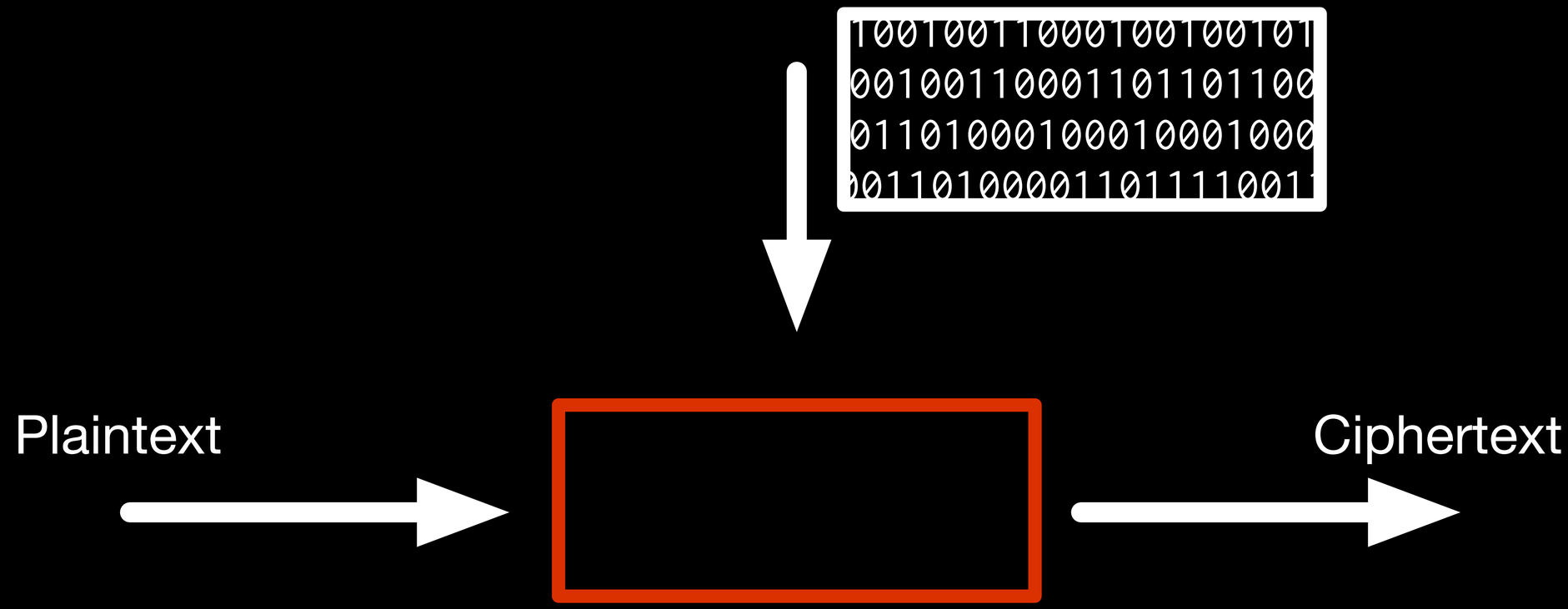


- Checksums
  - stable
  - quick
- Cryptographic hashes
  - collision resistant
  - pre-image resistant

- Keyed hashes
  - authentication

luminis
Conversing worlds

Keys



```
1         KeyGenerator aesGenerator = KeyGenerator.getInstance("AES");
2         javax.crypto.SecretKey key = aesGenerator.generateKey();
3         System.out.println(String.format("Generated secret key: %s",
4                 Util.toHex(key.getEncoded())));
```

```
Generated secret key: F2B8CDAF2E949210F45CE2E67A0A0A0B
```

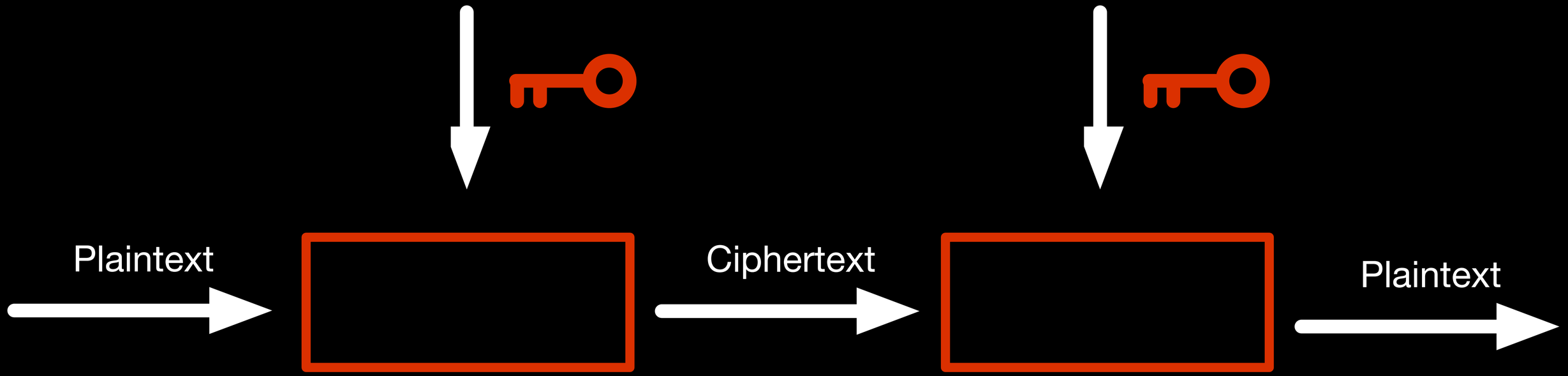
```
1         Key key = new SecretKeySpec(
2                 new byte[] {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
3                             0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
4                 "AES");
5         System.out.println(String.format("secret key from 0 input: %s",
6                         Util.toHex(key.getEncoded())));
```

```
secret key from 0 input: 00000000000000000000000000000000
```

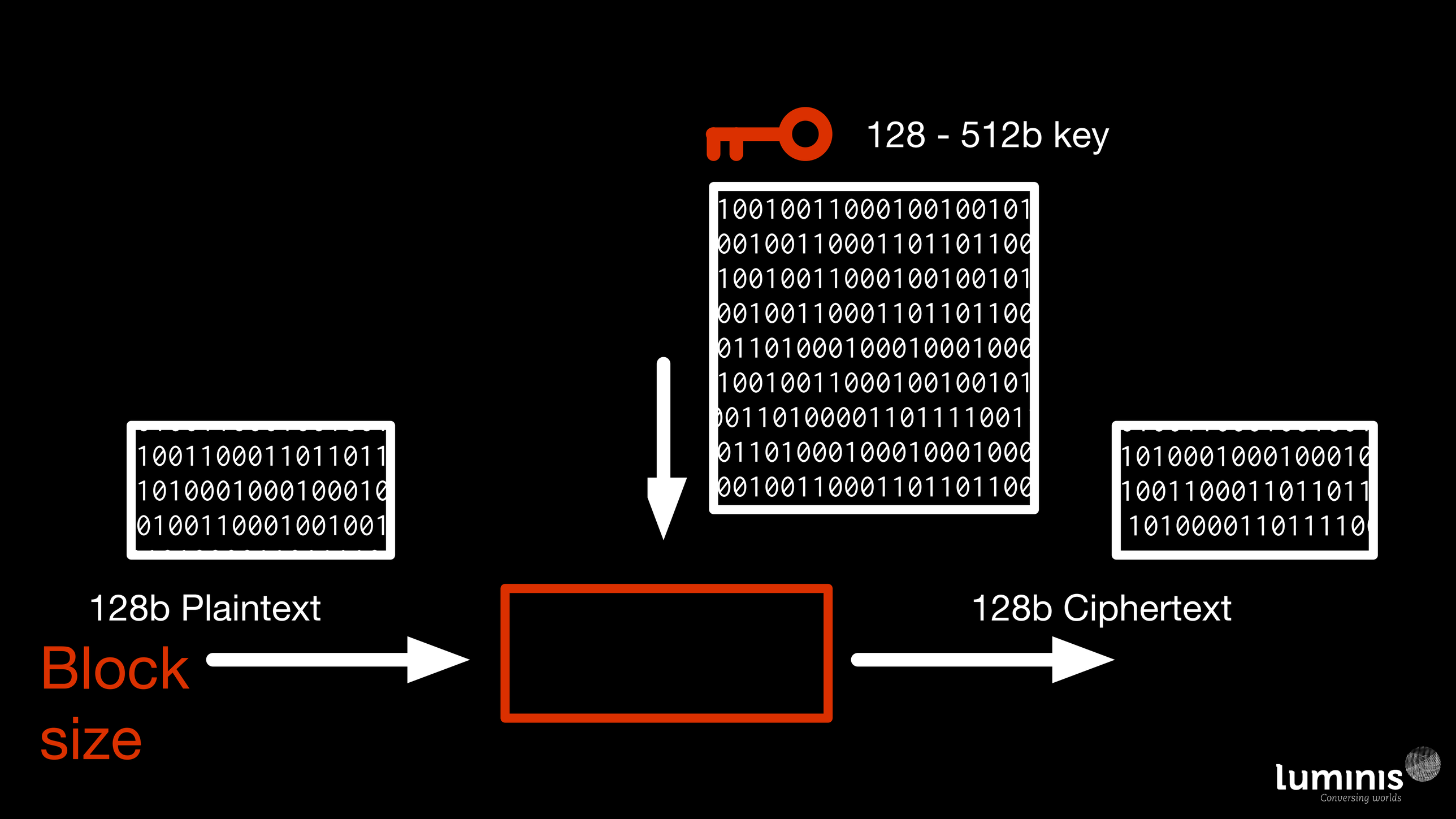
```
1         SecretKeyFactory skf = SecretKeyFactory.getInstance( "PBKDF2WithHmacSHA1" );
2
3         PBEKeySpec spec = new PBEKeySpec( "nauticalnonsense".toCharArray(), // the password
4                 new byte[] {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08}, // a salt
5                 50, // iteration count, hardness of the function
6                 128 ); // key length, should match the kind of encryption you're trying to do
7         Key key = skf.generateSecret(spec);
8         System.out.println(String.format("secret key from \"nauticalnonsense\" using PBKDF2 with
9                         HmacSHA256: %s", Util.toHex(key.getEncoded())));
```

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

Symmetric encryption



AES (Rijndael)
Block cipher



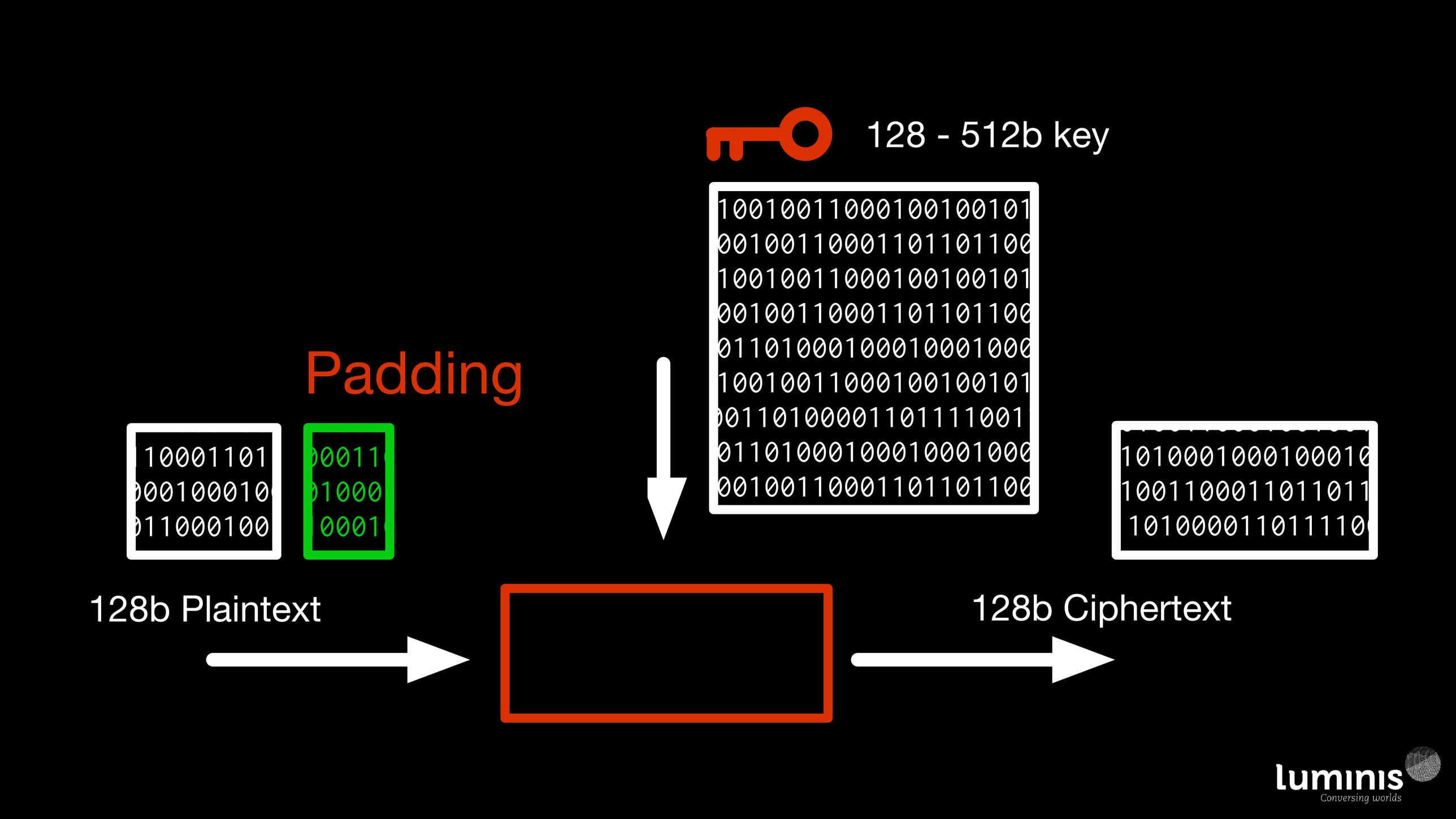
luminis
Conversing worlds

```java
        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);
        System.out.println(String.format("Encrypted version of \"%s\" with AES key 0: %s",
                new String(data),
                Util.toHex(encrypted)));

        cipher.init(Cipher.DECRYPT_MODE, key);
        byte[] decrypted = cipher.doFinal(encrypted);

        System.out.println(String.format("Decrypted: %s", new String(decrypted)));
```

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

```java
        try {
            System.out.println("Trying to encrypt 15 bytes...");
            Cipher cipher = Cipher.getInstance("AES/ECB/NoPadding");
            cipher.init(Cipher.ENCRYPT_MODE, key);
            cipher.doFinal("1234567890123456".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)
```



PKCS5 padding

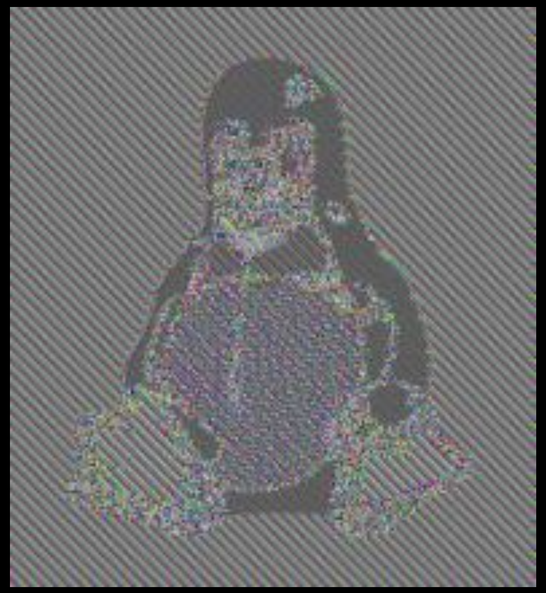
*Public Key Cryptography Standard

```
1           Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
2           byte[] data = "1234567890123456".getBytes();
3           cipher.init(Cipher.ENCRYPT_MODE, key);
4           byte[] encrypted = cipher.doFinal(data);
5           System.out.println(String.format("Encrypted version of \"%s\" with AES key 0: %s",
6                   new String(data),
7                   Util.toHex(encrypted)));
```

```
Encrypted version of "1234567890123456" with AES key 0: 9BF579AE2C77467D96E4A7CA2E0ED95C
```

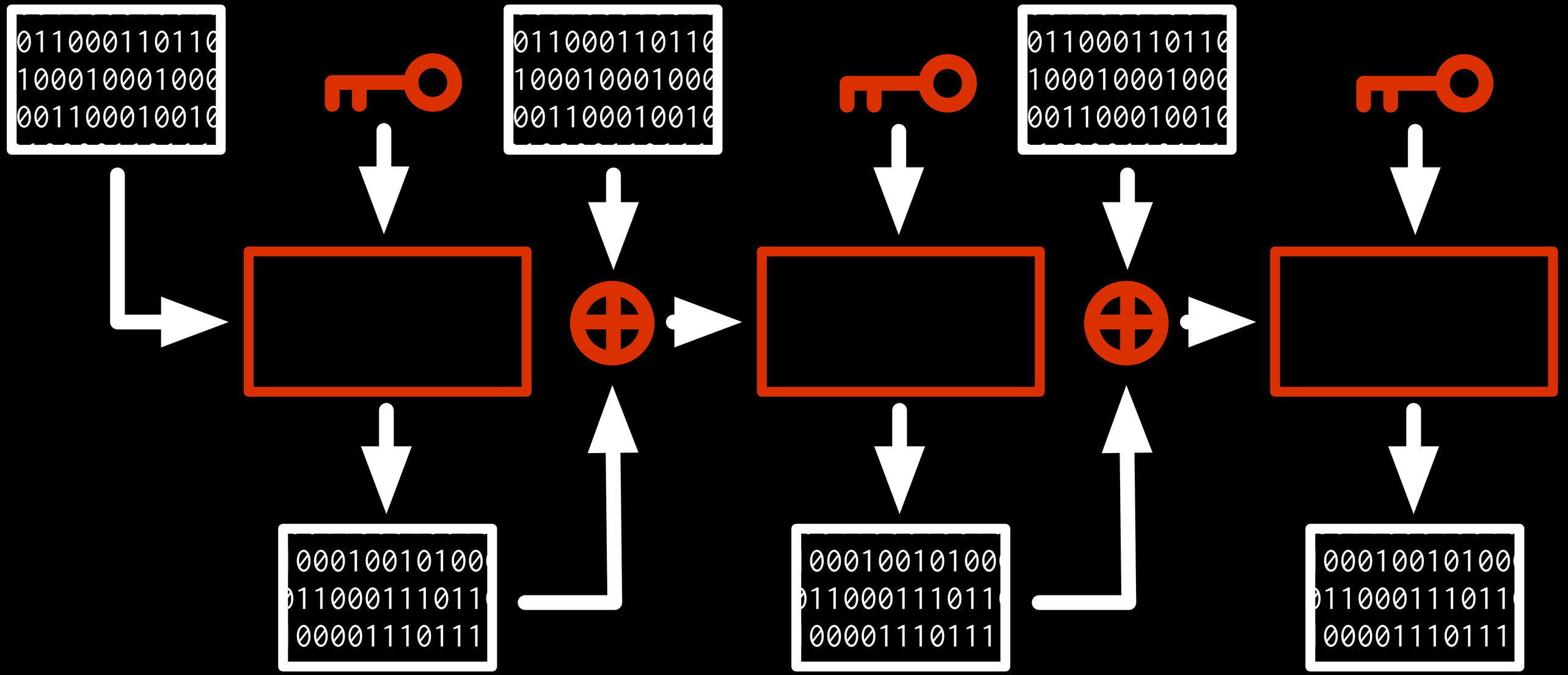
```
1         Cipher cipher = Cipher.getInstance("AES/ECB/NoPadding");
2         byte[] data = "1234567890123456123456789012345612345678901234561234567890123456"
3                 .getBytes();
4         cipher.init(Cipher.ENCRYPT_MODE, key);
5         byte[] encrypted = cipher.doFinal(data);
6         System.out.println(String.format("Encrypted version of \"%s\" with AES key 0:\n%s",
7                 new String(data),
8                 breakStringAt(Util.toHex(encrypted), 32)));
```

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



luminis
Conversing worlds

https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#ECB



Cipher Block Chaining

```
1         Cipher cipher = Cipher.getInstance("AES/CBC/NoPadding");
2         byte[] data = "1234567890123456123456789012345612345678901234561234567890123456"
3                 .getBytes(); // just four times the same text
4         cipher.init(Cipher.ENCRYPT_MODE, key);
5         byte[] encrypted = cipher.doFinal(data);
6         System.out.println(String.format("Encrypted version of \"%s\" with AES key 0:\n%s",
7                 new String(data),
8                 breakStringAt(Util.toHex(encrypted), 32)));
```

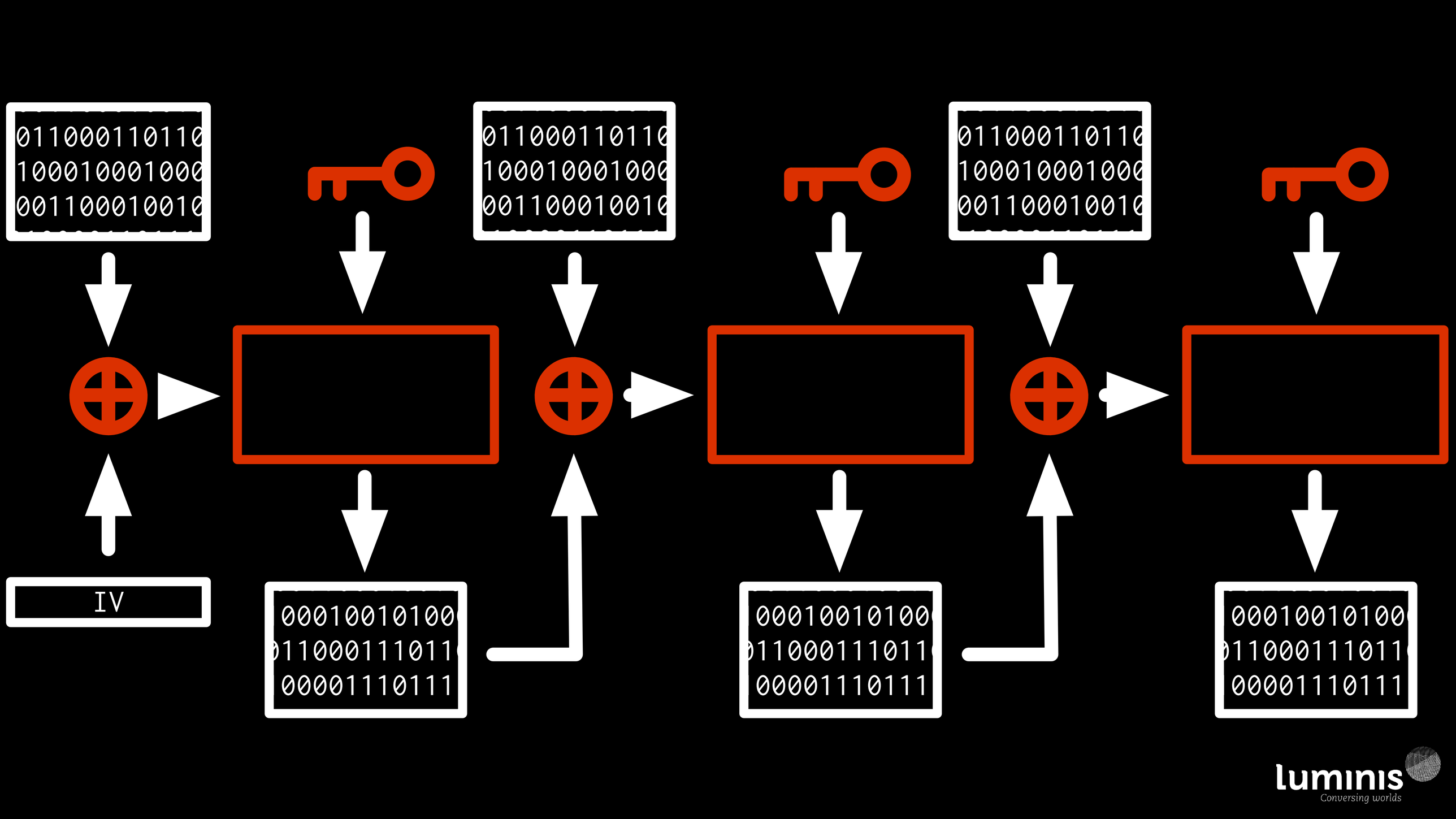
```
Encrypted version of "1234567890123456123456789012345612345678901234561234567890123456" CBC:
1F3C8C921D5278C929F965D964FE3F4C
68DAEDEEDA5E1B825DCAB2EBADF9369A
518ADB2208C0741CA3215DE137F02604
881758BBF2E52FAF57E902C6BF27B46B
```

```
Encrypted version of "1234567890123456123456789012345612345678901234561234567890123456" CBC:
34DA8ED866CF25078D087AC5D22F93FE
7B334FA40B5FA539FA393BF5FBEE7EAA
0ABA95EAA72758F35609C75CE52567DD
A1FB1D781F7F01AA65389021B43C4A84
```

```
Encrypted version of "1234567890123456123456789012345612345678901234561234567890123456" CBC:
D5C12E6B8B89AE91D04BAFE4E9E18273
934AB4551F916171B58E7FF2C69DF9C5
C0BAF06EF4A68ABFD662DA0421E44AA9
C55FDC4D12D9B2153E2CCF1876DDCBD6
```

```
1         Cipher cipher = Cipher.getInstance("AES/CBC/NoPadding");
2         byte[] data = "1234567890123456123456789012345612345678901234561234567890123456"
3                 .getBytes();
4         cipher.init(Cipher.ENCRYPT_MODE, key);
5         byte[] encrypted = cipher.doFinal(data);
6         System.out.println(String.format("Encrypted version of \"%s\" CBC:\n%s",
7                 new String(data), breakStringAt(Util.toHex(encrypted), 32)));
8
9         try {
10            System.out.println("Decrypting...");
11            cipher.init(Cipher.DECRYPT_MODE, key);
12        }
13        catch (InvalidKeyException e) {
14            e.printStackTrace(System.out);
15        }
```

```
Encrypted version of "1234567890123456123456789012345612345678901234561234567890123456" 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 javax.crypto.Cipher.init(Cipher.java:1226)
```



http://www.isaac.cs.berkeley.edu/isaac/wep-faq.html

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

```
1         Cipher cipher = Cipher.getInstance("AES/CBC/NoPadding");
2         byte[] data = "123456789012345612345678901234561234567890123456123456789012345612345678901234561234567890123456"
3                 .getBytes();
4
5         IvParameterSpec iv = new IvParameterSpec(
6                 new byte[] {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
7                             0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F});
8
9         cipher.init(Cipher.ENCRYPT_MODE, key, iv);
10        byte[] encrypted = cipher.doFinal(data);
11        System.out.println(String.format("Encrypted version of \"%s\" with AES key 0:\n%s",
12                new String(data),
13                breakStringAt(Util.toHex(encrypted), 32)));
14
15        cipher.init(Cipher.DECRYPT_MODE, key, iv);
16        System.out.println(String.format("Decrypted :\n%s", new String(cipher.doFinal(encrypted))));
```

```
Encrypted version of "123456789012345612345678901234561234567890123456123456789012345612345678901234561234567890123456" with AES key 0:
466243EC0D98F0AD82AE3263F7A36D0D
31738A94186E76120011DFF92113D73B
8E387C7DF0B22428ED6BD7B9E17A4F06
6C034BFAAF6387BB0F2BC0FCC492F082

Decrypted :
123456789012345612345678901234561234567890123456123456789012345612345678901234561234567890123456
```

Symmetric crypto

Block size

Padding

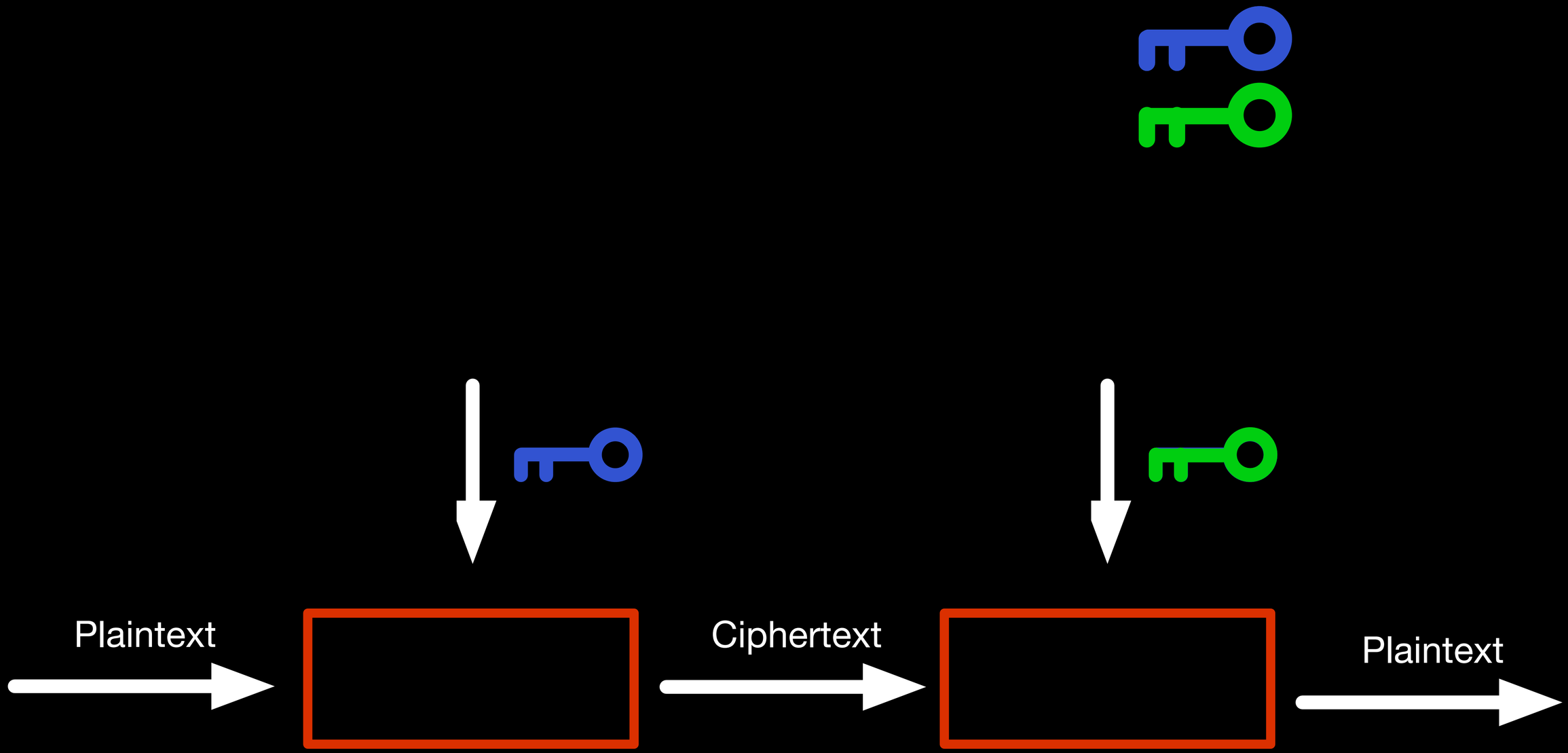
Cipher modes
- ECB
- CBC

Initialization Vector

luminis
Conversing worlds

http://arstechnica.com/security/2015/07/what-amateurs-can-learn-from-security-pros-about-staying-safe-online/

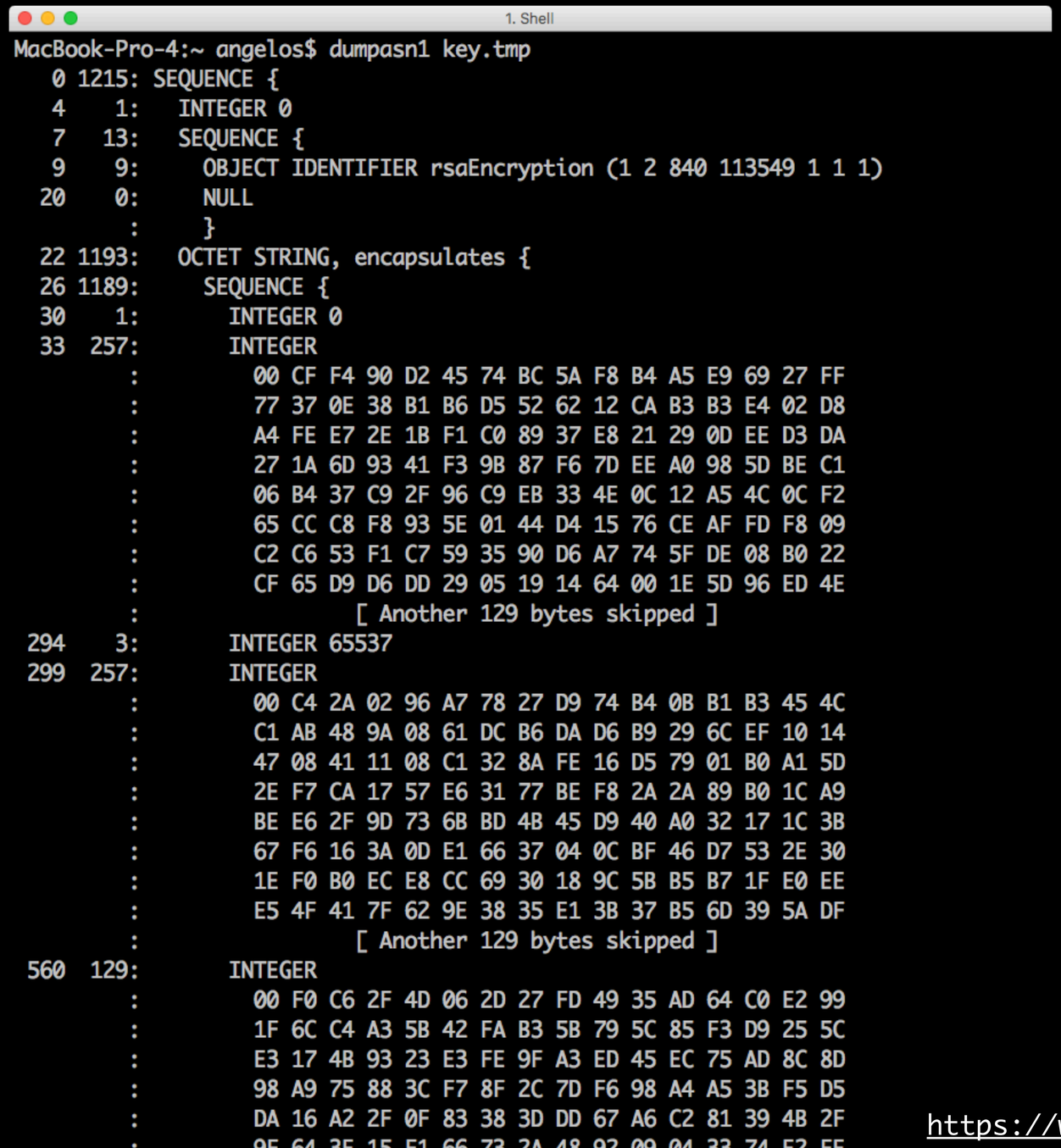
Asymmetric encryption



```java
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
        kpg.initialize(2048); // Key size

        KeyPair keyPair = kpg.generateKeyPair();

        System.out.println(String.format("Generated a private key: %s",
                Util.toHex(keyPair.getPrivate().getEncoded())));
        System.out.println(String.format("Generated a public  key: %s",
                Util.toHex(keyPair.getPrivate().getEncoded())));
```

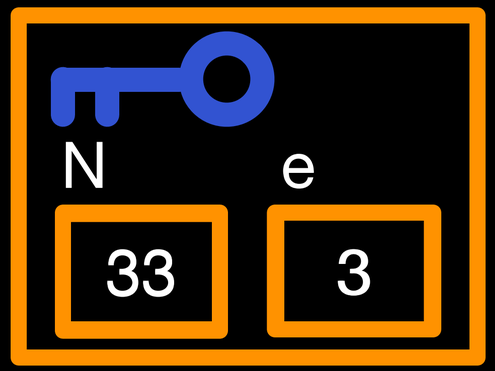
```
Generated a private key:
3082[redacted]4BD020100300D06092A864886F70D0101010500048204A7308204A30201000282010100E368C1A59F49F8517A39F4724407
45705E902E104792F5A1CB41D4C1D237FF4AF10254A93E7A9F56F1654761C8636D4A5C1384663950DFABC40B8A814B7F4966D9CA
F960067C6735A7D49F8A2135CA018C41FDC555C25B53715CEF968B553FD287A6E0640C94D98166E4EF269B562E558BA65B...
Generated a public  key:
3082[redacted]122300D06092A864886F70D0101010500038201 0F003082010A0282010100E368C1A59F49F8517A39F4724407B745705E90
104792F5A1CB41D4C1D237FF4AF10254A93E7A9F56F1654761C8636D4A5C1384663950DFABC40B8A814B7F4966D9CA69F960067C
35A7D49F8A2135CA018C41FDC555C25B53715CEF968B553FD287A6E0640C94D98166E4EF269B562E558BA65BCC564CB58D...
```

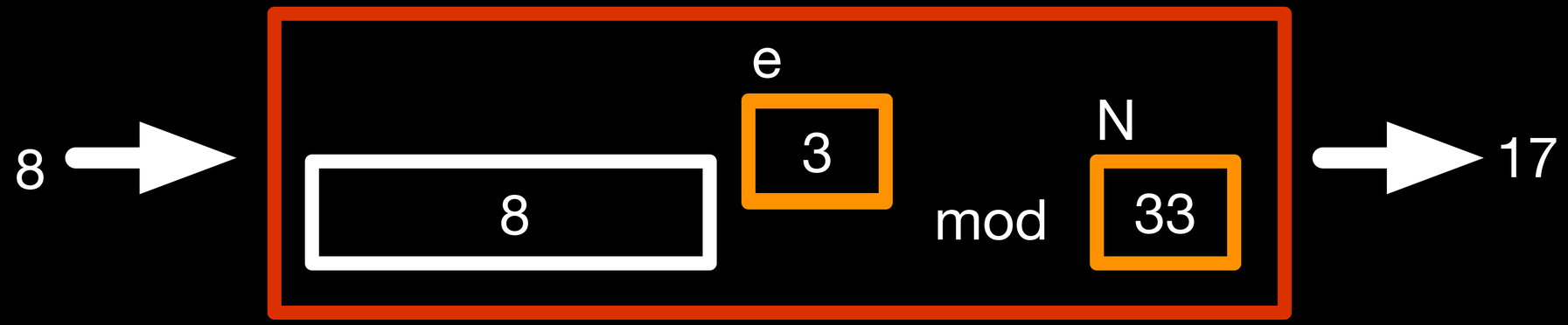
luminis
Conversing worlds

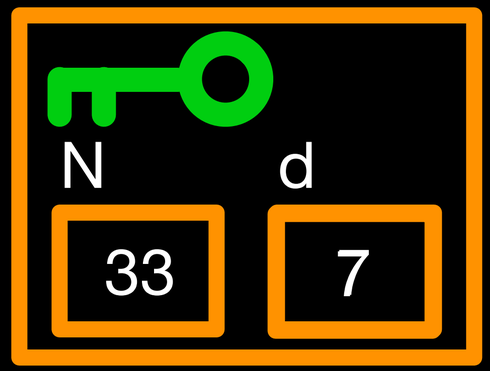


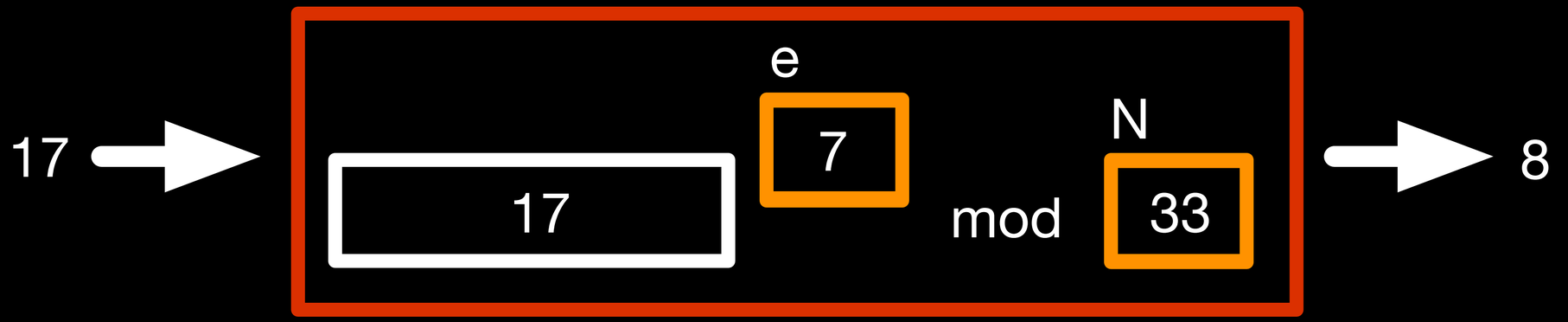
https://www.cs.auckland.ac.nz/~pgut001/dumpasn1.c

RSA









```
1         byte[] data = "1234567890123456".getBytes();
2
3         Cipher rsa = Cipher.getInstance("RSA");
4
5         rsa.init(Cipher.ENCRYPT_MODE, keyPair.getPublic());
6         byte[] encrypted = rsa.doFinal(data);
7
8         System.out.println(String.format("Encrypted version of \"%s\" with generated RSA key:\n%s",
9                 new String(data),
10                breakStringAt(Util.toHex(encrypted), 64)));
```

```
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
```

```
1          byte[] data = "1234567890123456".getBytes();
2
3          Cipher rsa = Cipher.getInstance("RSA");
4
5          rsa.init(Cipher.ENCRYPT_MODE, keyPair.getPublic());
6          byte[] encrypted = rsa.doFinal(data);
7
8          System.out.println(String.format("Encrypted version of \"%s\" with generated RSA key:\n%s",
9                  new String(data),
10                 breakStringAt(Util.toHex(encrypted), 64)));
11
12         rsa.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());
13         byte[] decrypted = rsa.doFinal(encrypted);
14         System.out.println(String.format("Decrypted: %s",
15                 new String(decrypted)));
```

```
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

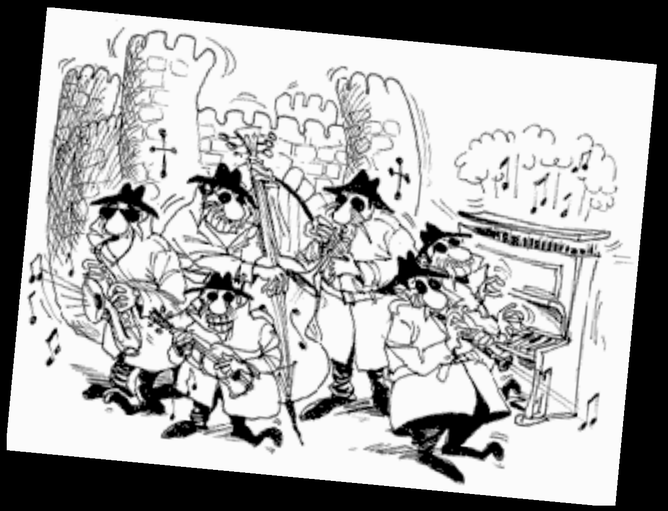
Private & public key

Simple but expensive math

Java Cryptography Architecture

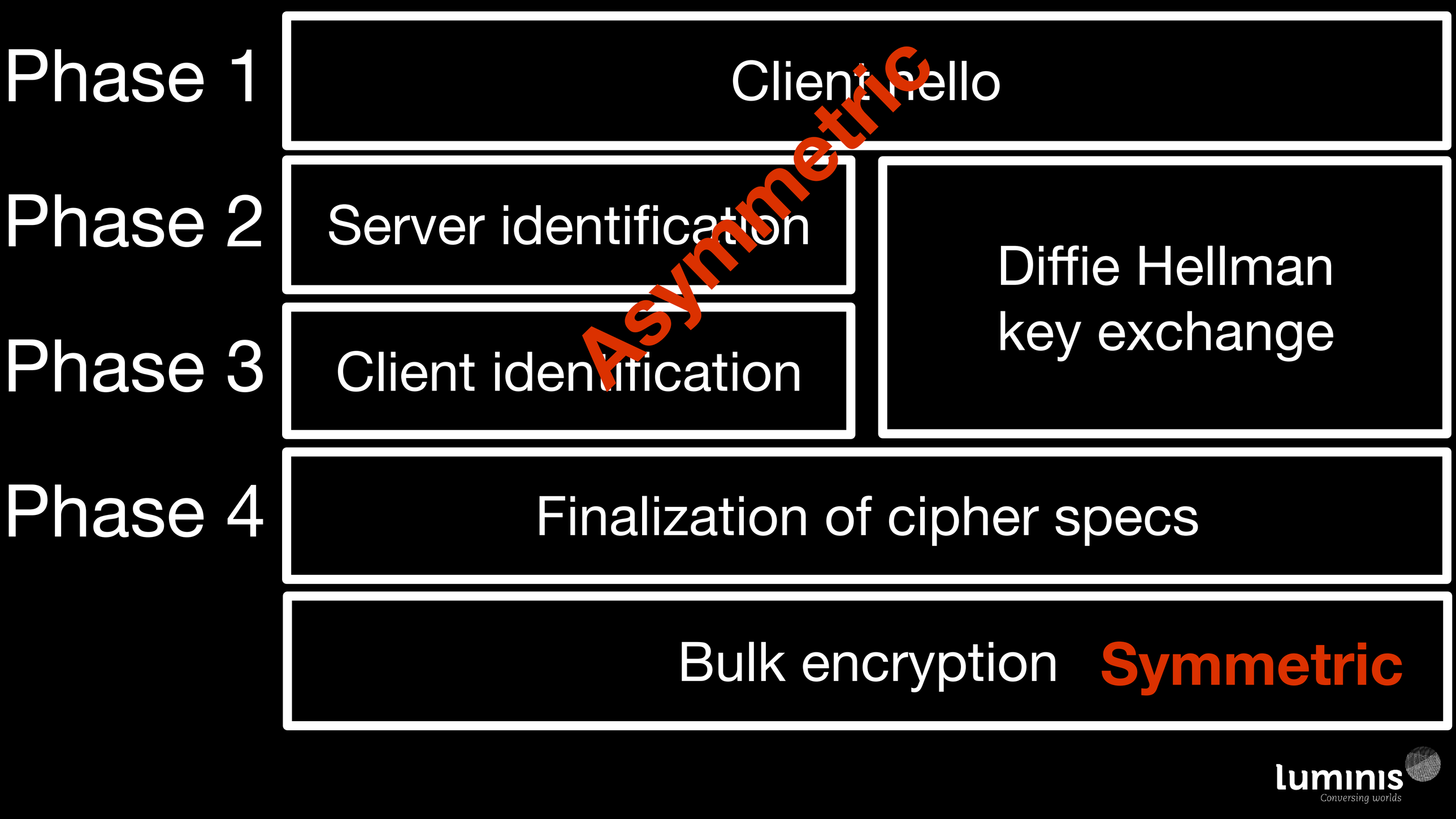
```
1 static EngineClassName getInstance(String algorithm)
2     throws NoSuchAlgorithmException
3
4 static EngineClassName getInstance(String algorithm, String provider)
5     throws NoSuchAlgorithmException, NoSuchProviderException
6
7 static EngineClassName getInstance(String algorithm, Provider provider)
8     throws NoSuchAlgorithmException
```

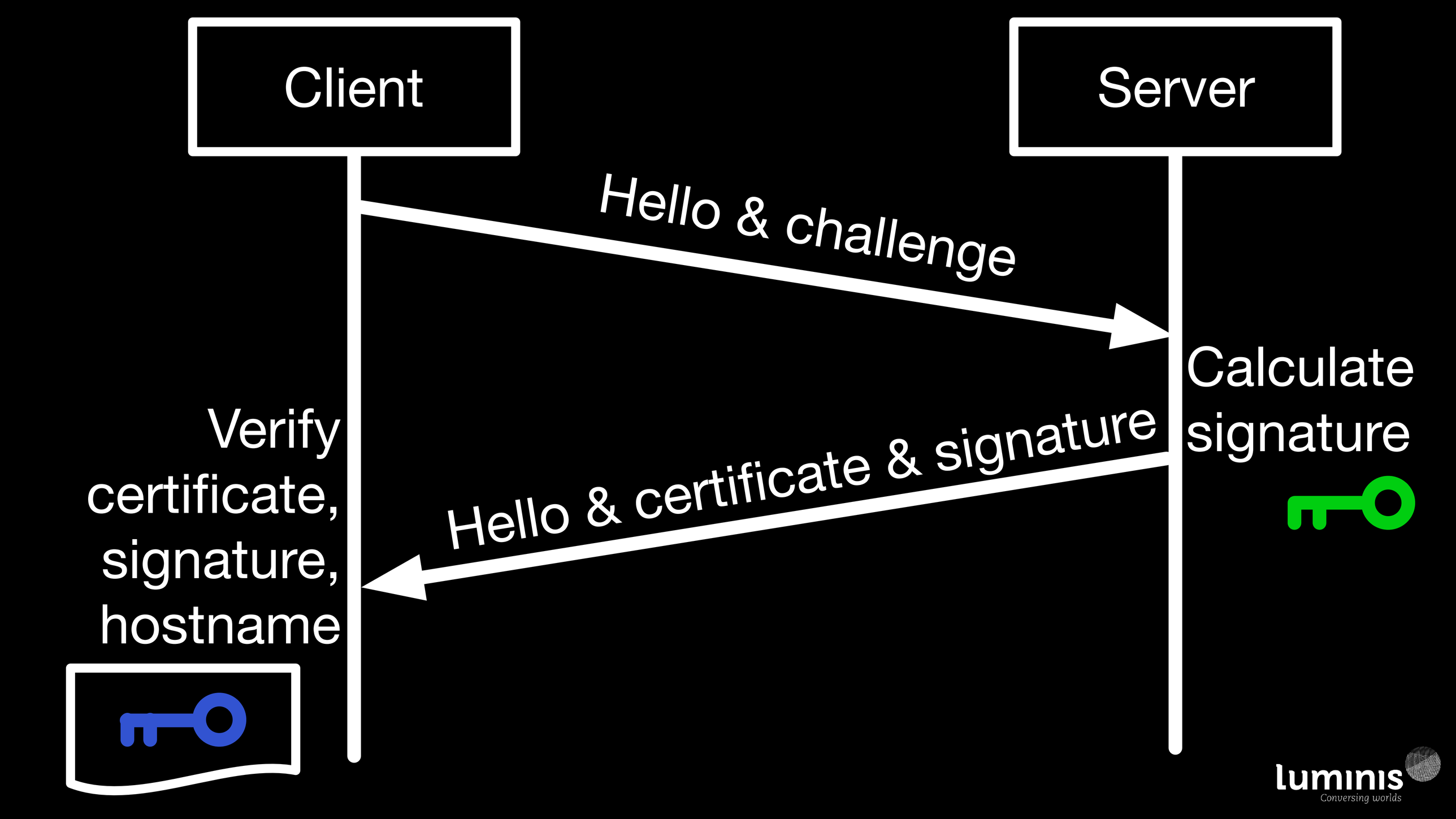
```
1 Security.addProvider(new MyProvider());
```



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

TLS




Client
Server
Hello & challenge
Calculate signature
Hello & certificate & signature
Verify certificate, signature, hostname

```java
// Initialize the Server Socket
SSLServerSocketFactory sslServerSocketfactory =
        (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
SSLServerSocket sslServerSocket =
        (SSLServerSocket) sslServerSocketfactory.createServerSocket(8443);
SSLSocket sslSocket = (SSLSocket) sslServerSocket.accept();

PrintWriter out = new PrintWriter(sslSocket.getOutputStream(), true);
BufferedReader in = new BufferedReader(
        new InputStreamReader(sslSocket.getInputStream()));

String input;
while ((input = in.readLine()) != null) {
    out.println(input);
    System.out.println(input);
}

// IO cleanup omitted for readability
```

private.key

```
-----BEGIN RSA
MIIEogIBAAKCAQE
nrJK6S2hnFpRFP0
XCIVm5tLQ7/s1QT
yb0H62Pg8p5K7Ok
```

certificate.cer

```
-----BEGIN CERT
MIIDsjCCApqgAwI
BAYTAk5MMRAwDgY
BAMTCWV2ZXJ5LmR
CzAJBgNVBAYTAk5
```

keystore.p12

```
0010010100100110
0010010100100110
1000100010010011
1111001101101000
0001000100100110
```

```
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
```

```
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
```

luminis
Conversing worlds

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
```

luminis
Conversing worlds

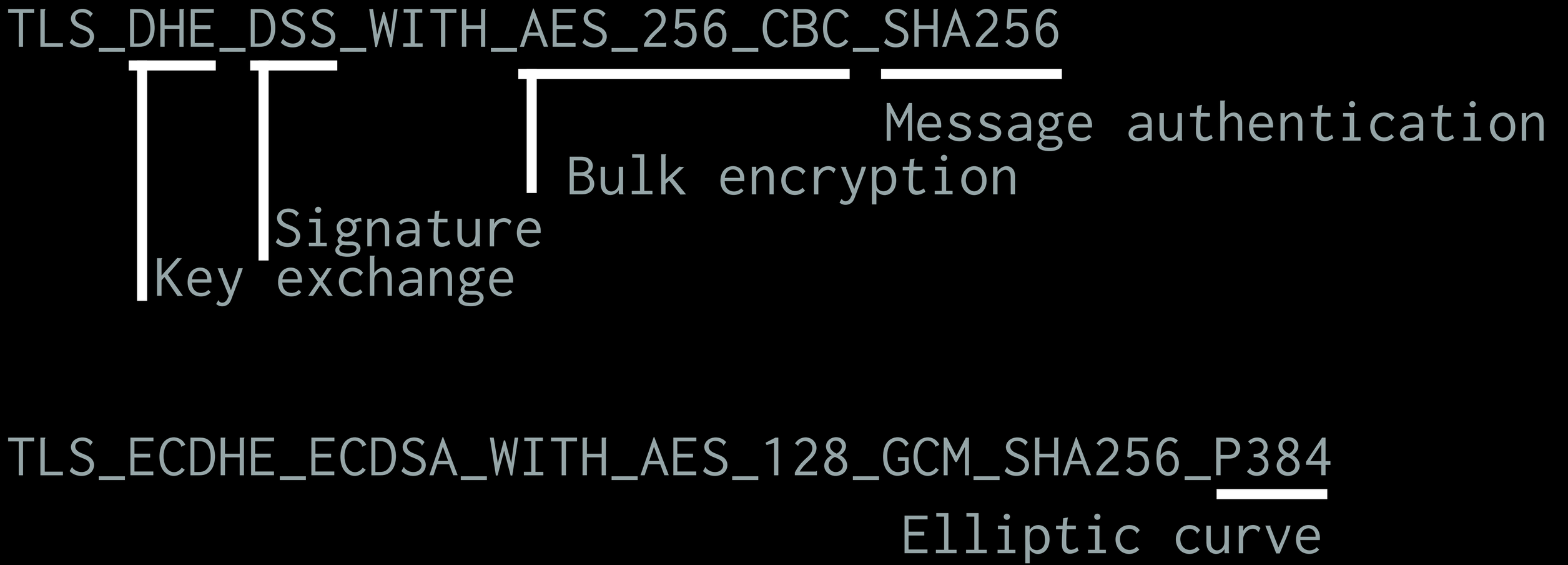
```
1 java \
2 -Djavax.net.debug=all \
3 -Djavax.net.ssl.keyStore=keystore.p12 \
4 -Djavax.net.ssl.keyStorePassword=123456 \
5 -cp src \
6 crypto6.ssl1.SSL
```

Cipher suites

```
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384_P384,
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA_P384,
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA_P384,
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA_P256,
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA_P256,
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256_P384,
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256_P384,
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256_P256,
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256_P256,
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA_P384,
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA_P256,
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA_P384,
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA_P256,
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256_P384,
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256_P256,
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256,
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256
TLS_DHE_DSS_WITH_AES_128_CBC_SHA256,
TLS_DHE_DSS_WITH_AES_128_CBC_SHA,
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA,
TLS_RSA_WITH_AES_256_CBC_SHA256,
TLS_RSA_WITH_AES_256_CBC_SHA,
TLS_RSA_WITH_AES_128_CBC_SHA256,
TLS_RSA_WITH_AES_128_CBC_SHA,
TLS_RSA_WITH_3DES_EDE_CBC_SHA,
SSL_CK_DES_192_EDE3_CBC_WITH_MD5
```

luminis
Conversing worlds

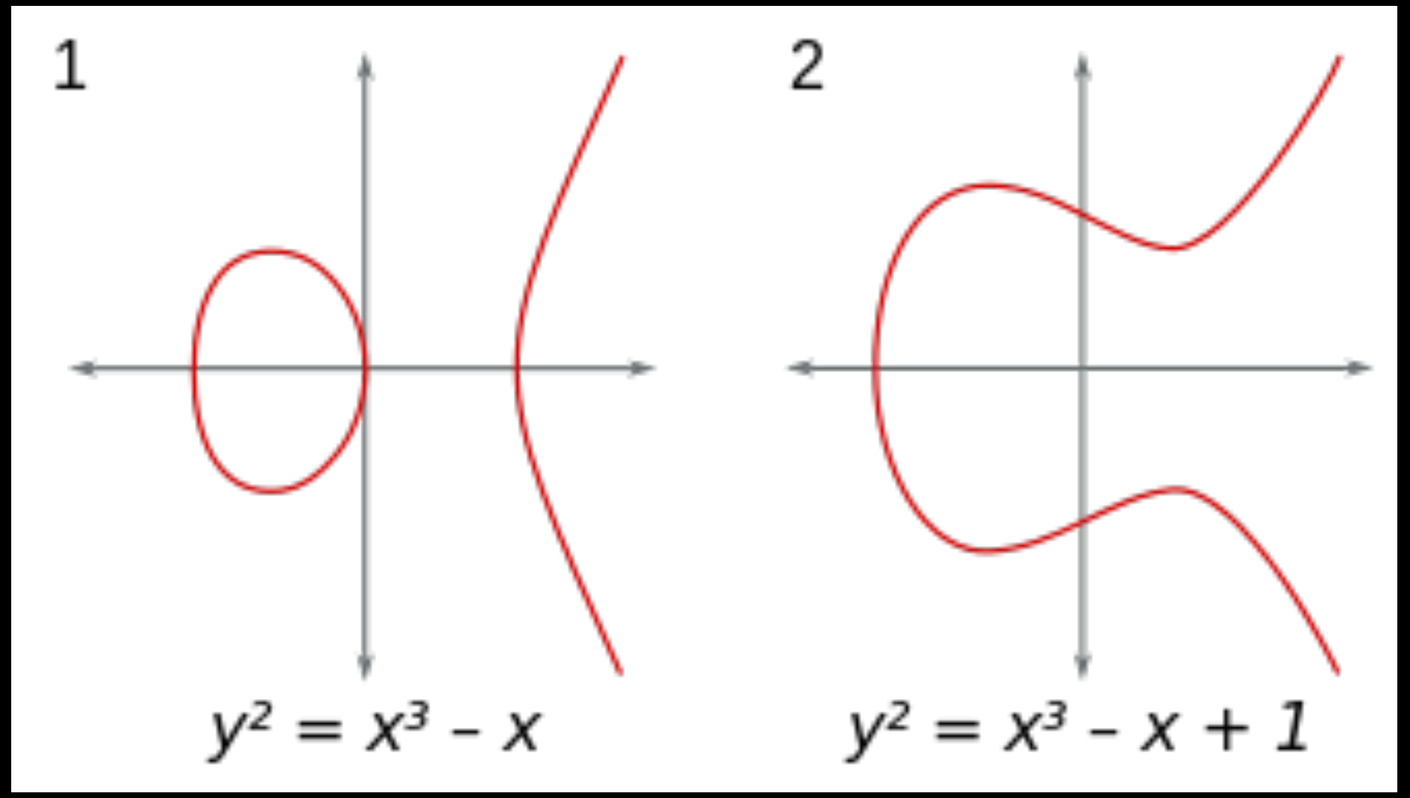
https://www.grc.com/miscfiles/SChannel_Cipher_Suites.txt



New developments

luminis
Conversing worlds

Elliptic Curve



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

Hashing
Checksum
Crypto hash
HMAC
PBKDF

Symmetric crypto
Block size
Padding
Cipher modes
- ECB
- CBC

Initialization Vector
AES

Asymmetric crypto
Public vs private key
Certificates
RSA

TLS
File formats (PEM, P12)
Message authentication
Signature
Bulk encryption
Cipher suites

Pick up your cheat sheet at the front!

luminis
Conversing worlds

NEVER

build your own cryptography

ALWAYS

use built-in primitives
patch your systems
rely on experts

luminis
Conversing worlds

what every  developer needs to know about

cryptography

https://github.com/angelos/javacrypto

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

Angelo van der Sijpt
@_angelos