

Encryption Deep Dive

Diego Pacheco



About me...



- ❑ Cat's Father
- ❑ Head of Software Architect
- ❑ Agile Coach
- ❑ SOA/Microservices Expert
- ❑ DevOps Practitioner
- ❑ Speaker
- ❑ Author



diegopacheco



@diego_pacheco



<http://diego-pacheco.blogspot.com.br/>




<https://diegopacheco.github.io/>


Security 101

diegopacheco diegopacheco tinyurl.com/diegoyoutube diego-pacheco.blogspot.com

Sec
101



Diego Pacheco



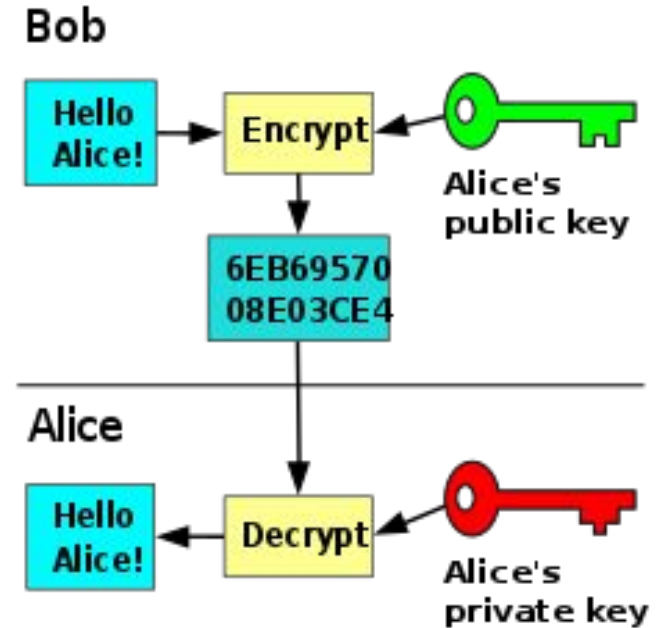
0:30 / 44:05

Sec 101

<https://www.youtube.com/watch?v=WnkCdQb7S9s>

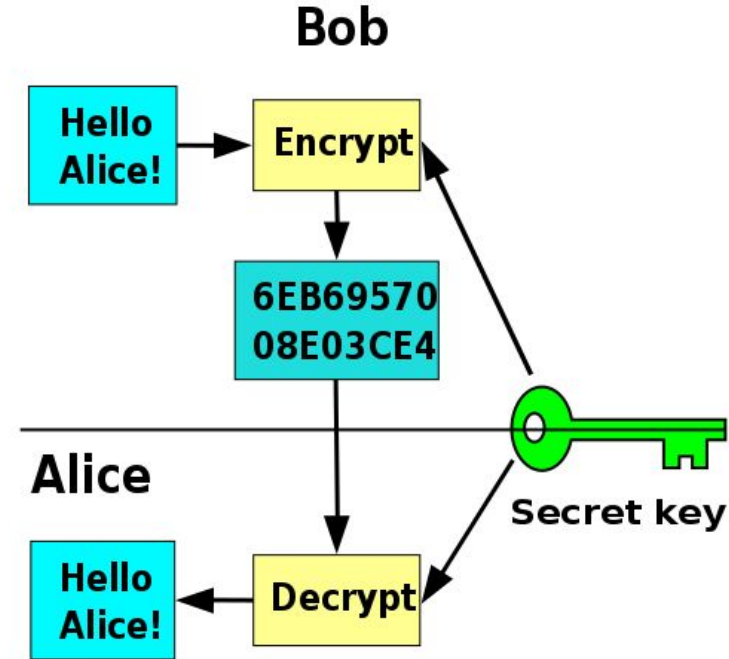
Asymmetrical Encryption

- ❑ Protect Data at Transit
- ❑ Public Key Cryptography
- ❑ Public Key does Encryption
- ❑ Private Key does Decryption
- ❑ Used on TLS/mTLS(HTTPS), SSH
- ❑ Algos: RSA, Diffie-Hellman, ECC
- ❑ Key Size often from 1 to 4kb



Symmetrical Encryption

- ❑ Protect Data at Rest (Payment, PII, etc...)
- ❑ Requires RNG and PRNG
- ❑ Biggest problem: Share / Access the encryption keys
- ❑ Block vs Stream Algos
- ❑ Algos: AES, RC4, RC5
- ❑ Key Sizes: 128, 192, 256
- ❑ $C = E(K, P)$ - $P = D(K, C)$



Symmetrical Encryption



- ❑ Ciphertext same or slightly bigger than the plaintext
- ❑ Ciphertext is never smaller than the plaintext
- ❑ Permutations should be determined by the key
- ❑ Different keys should result in different permutation
- ❑ The permutation should look random
- ❑ Kerckhoffs Principle - 1883 - Really on a secret key - not on the secrecy of the Cipher
- ❑ Storage vs Application level encryption

Application side encryption Drawbacks



- ❑ For sure more secure, however...
- ❑ Requires Engineering Discipline
- ❑ Requires constant Education / Training
- ❑ Cripple Features (Uniquines, Joins, Search)
- ❑ Bugs, Changes, Improvements Intrusive Require Migrations
- ❑ Key Scoping is mirror by Conway's law / Architecture



Protecting Keys

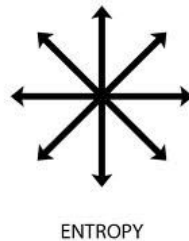
- ❑ Storing the key on a hardware token(Smart Card or USB dongle)
- ❑ On the fly generation from a password
- ❑ Wrapping - protecting the key with another key
- ❑ Key Management System

Randomness



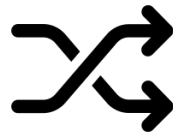
- ❑ Randomness is a probability distribution
- ❑ Probability is about the likelihood of something happening:
0 never, 1 certain.
- ❑ Uniform Distribution when all probabilities are equal
chance do happens
- ❑ Toss a coin (Tail or Head) is $1/2 + 1/2 = 1$, $1/2$ chance head,
 $1/2$ chance tail.

Entropy



- ❑ Entropy as a measure of Uncertain
 - ❑ The measure of disorder in a system
 - ❑ Entropy is the amount of surprise found in the result of a random process
 - ❑ Higher entropy less the certain found on the result
 - ❑ Entropy is maximized when the distribution is uniform

RNG

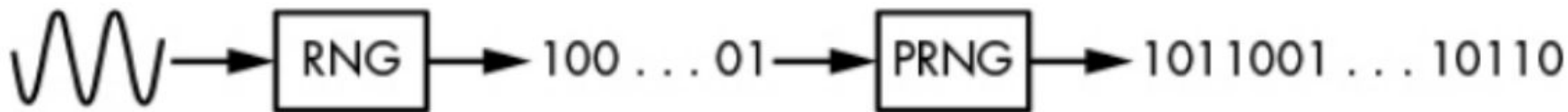


- ❑ A Source of incertain, or a source of entropy
- ❑ A Crypto algo to produce high-quality bits from the source of entropy
- ❑ RNG could be sampling from(Sources of entropy):
 - ❑ measurements of temperature, acoustic noise, air turbulence, or electric static (which are not always available)
 - ❑ Entropy from OS: sensors, I/O devices, network, disk, logs, running process (But can be fragile and manipulated by an attacker)

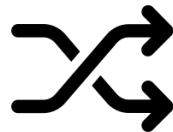
PRNG



- ❑ Rely on RNG
- ❑ RNG produce random bits(Analog) but slow and no deterministic and no guarantee of high entropy
- ❑ PRNG: Random looking data, Quickly from Digital Sources, deterministic and maximum entropy.



PRNG



- ❑ PRNG needs to ensure backtracking resistance (previously generated bits are impossible to recover)
- ❑ PRNG needs to ensure prediction resistance (impossible to predict future bits)
- ❑ `/dev/random` vs `/dev/urandom` (un-blocking)
- ❑ Java Secure Random

AES

- ❑ World defacto standard encryption
- ❑ NSA approved AES for TOP-Secret information
- ❑ AES is more Belgian than American
- ❑ Blocks of 128 bits (128, 192, 256)
- ❑ AES will never be broken
- ❑ All input bytes depends on output bytes in some complex pseudo random way



AES

- ❑ 2011 - People found a way to recover AES 128 bits key but require an insane amount of data plaintext ciphertext pairs 2^{88} bits (so we don't need to worry about it)
- ❑ The biggest threat is not the CORE algo but the Operation Modes
- ❑ Authentication vs Confidentiality modes



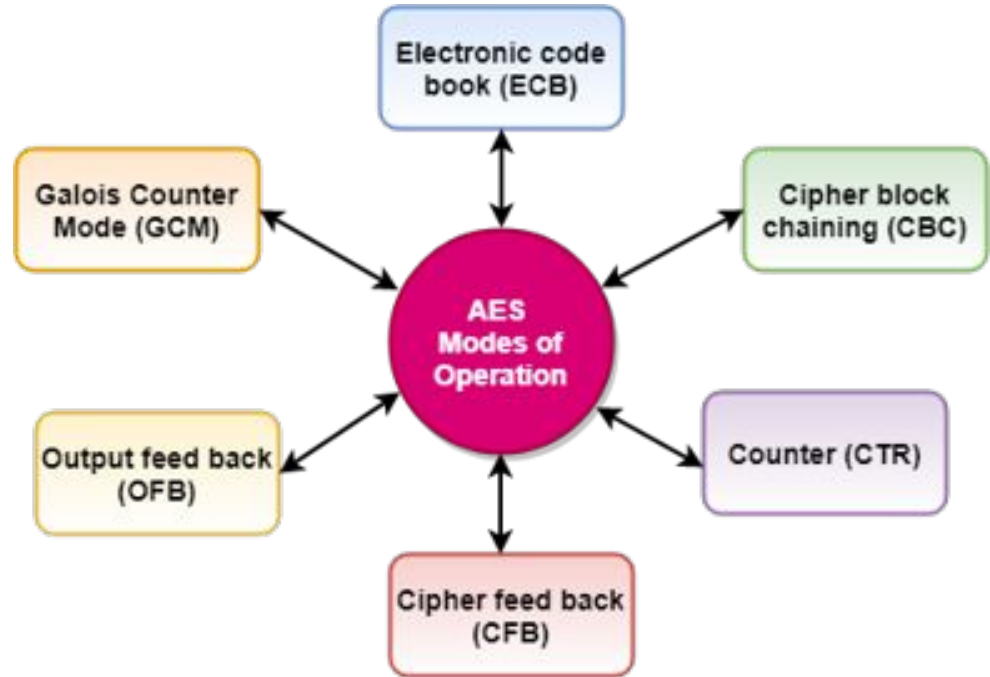
AES Operational Modes

❑ Authenticated Encryption

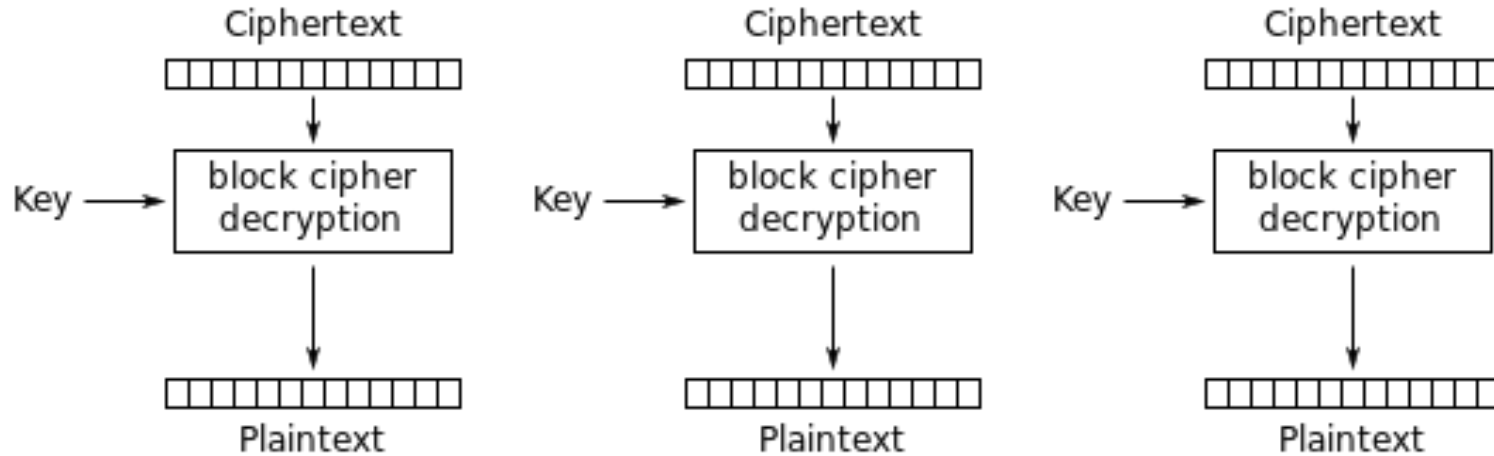
- ❑ GCM
- ❑ CCM
- ❑ SIV
- ❑ AES-GCM-SIV

❑ Confidentiality only

- ❑ ECB
- ❑ CBC
- ❑ PCBC
- ❑ CFB, OFB and CTR



AES Operational Modes - ECB (Electronic Code Book)

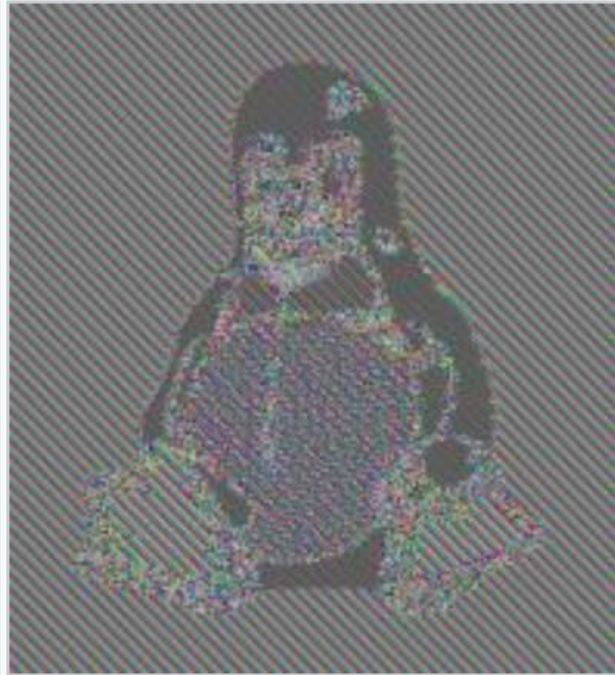


Electronic Codebook (ECB) mode decryption

AES Operational Modes - ECB (Electronic Code Book)

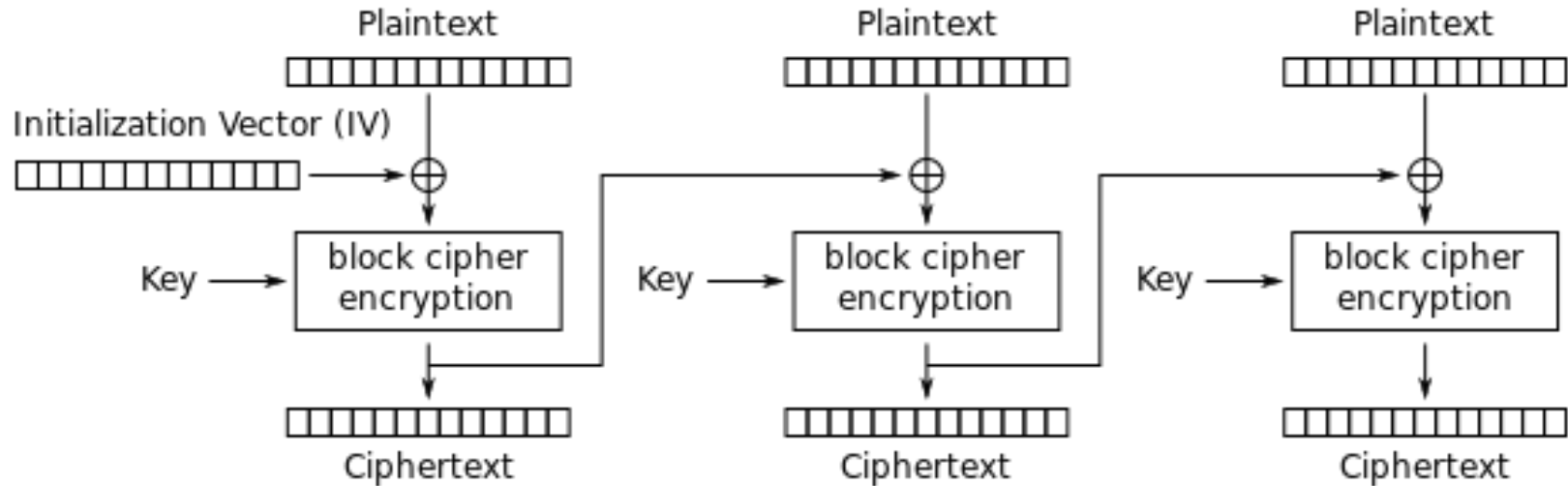


Original image



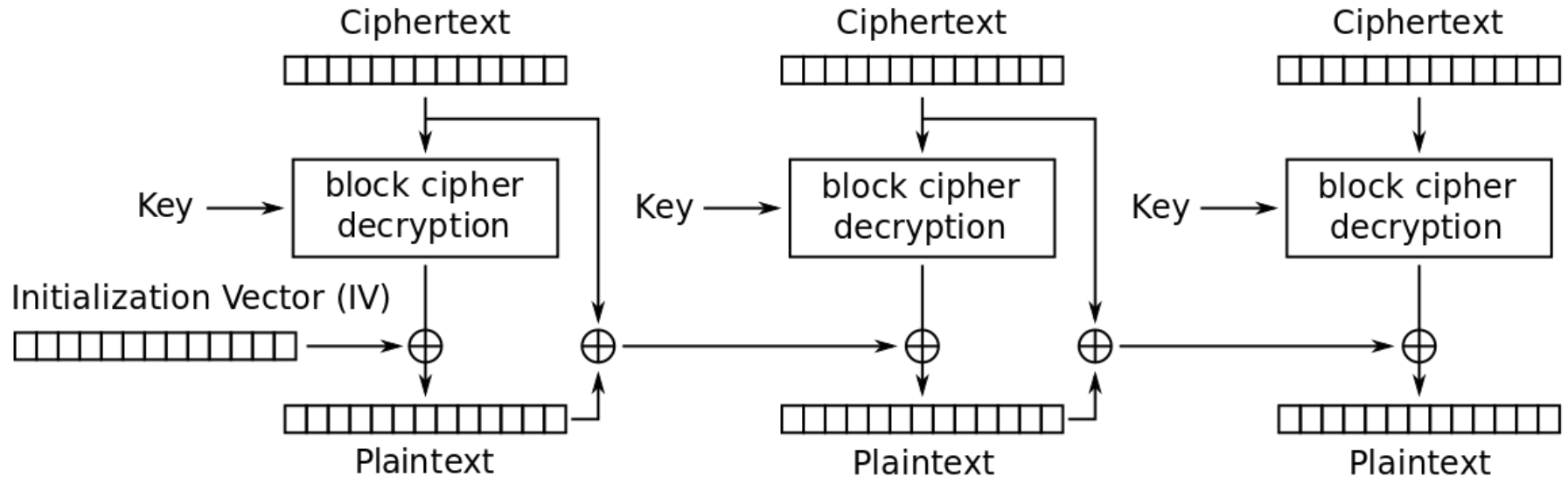
Encrypted using ECB mode

AES Operational Modes - CBC (Cipher Block Chaining)



Cipher Block Chaining (CBC) mode encryption

AES Operational Modes - PCBC (Propagate Cipher Block Chaining)



Propagating Cipher Block Chaining (PCBC) mode decryption

Each Block XORed plaintext + ciphertext

AES Operational Modes - CFB (Cipher Feedback)

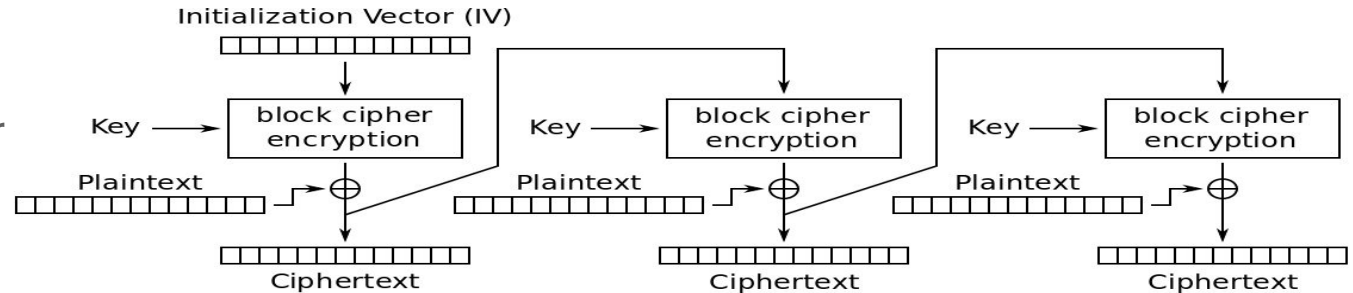
❏ Stream Cipher

❏ Decrypt

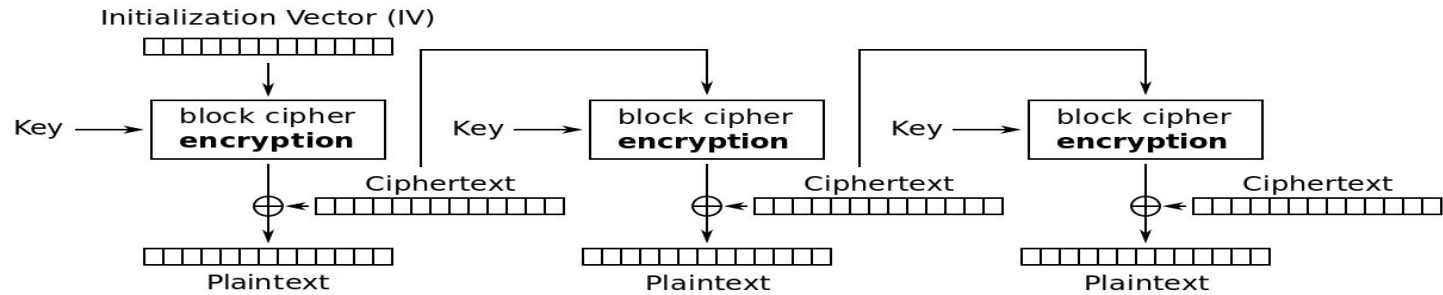
Parallel

❏ Encryption

Blocking



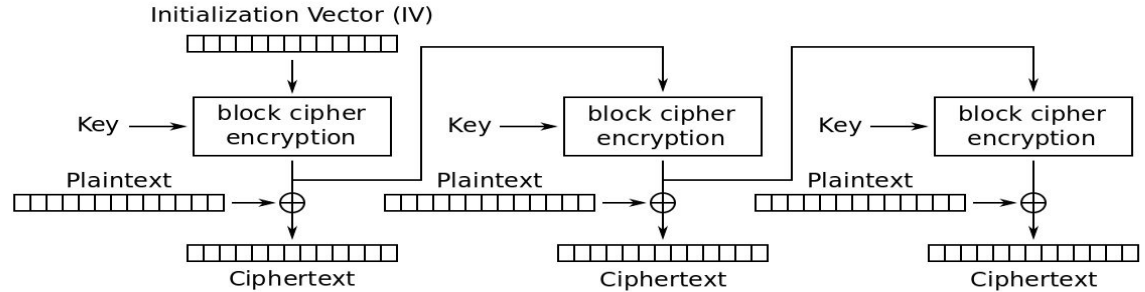
Cipher Feedback (CFB) mode encryption



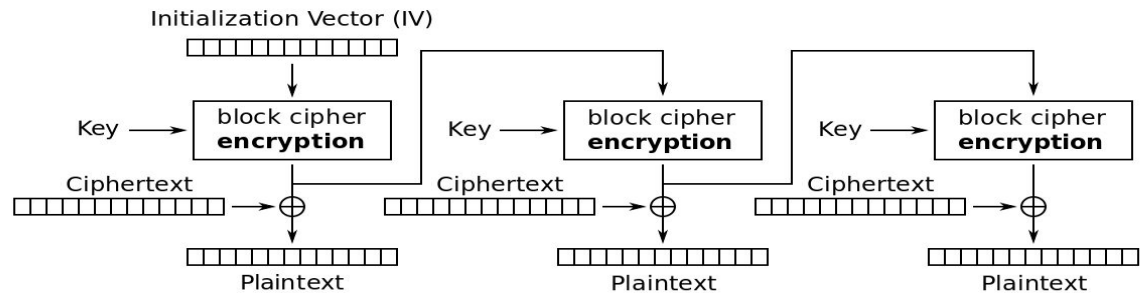
Cipher Feedback (CFB) mode decryption

AES Operational Modes - OFB (Output Feedback)

- ❑ Stream Cipher
- ❑ Encrypt IV
- ❑ XOR plaintext
- ❑ Does not require Padding
- Data

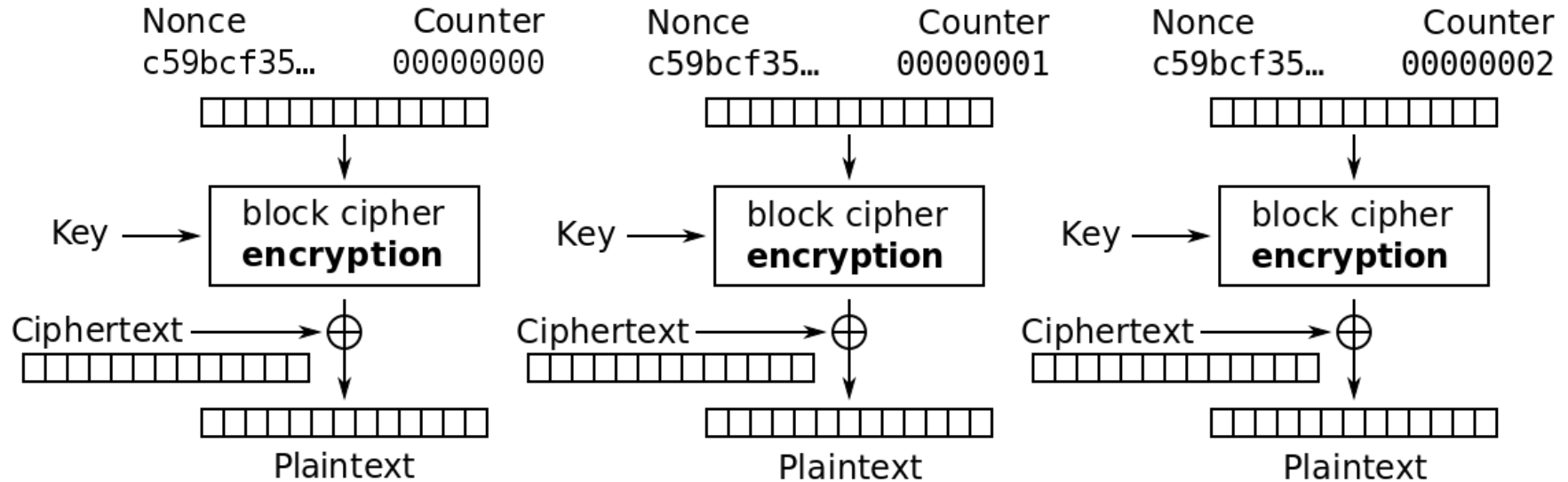


Output Feedback (OFB) mode encryption



Output Feedback (OFB) mode decryption

AES Operational Modes - CTR



Counter (CTR) mode decryption

encryption/decryption parallelization

Java Secure Random



Algorithm Name	Description
NativePRNG	Obtains random numbers from the underlying native OS. No assertions are made as to the blocking nature of generating these numbers.
NativePRNGBlocking	Obtains random numbers from the underlying native OS, blocking if necessary. For example, <code>/dev/random</code> on UNIX-like systems.
NativePRNGNonBlocking	Obtains random numbers from the underlying native OS, without blocking to prevent applications from excessive stalling. For example, <code>/dev/urandom</code> on UNIX-like systems.
PKCS11	Obtains random numbers from the underlying installed and configured PKCS11 library.
SHA1PRNG	The name of the pseudo-random number generation (PRNG) algorithm supplied by the SUN provider. This algorithm uses SHA-1 as the foundation of the PRNG. It computes the SHA-1 hash over a true-random seed value concatenated with a 64-bit counter which is incremented by 1 for each operation. From the 160-bit SHA-1 output, only 64 bits are used.
Windows-PRNG	Obtains random numbers from the underlying Windows OS.

<https://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html#SecureRandom>

Java Secure Random



```
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.util.Base64;

public class SecureRandomGenerator {

    public static byte[] generate(){
        SecureRandom secureRandom = null;
        try {
            secureRandom = SecureRandom.getInstance("NativePRNG");
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
        byte[] rndBytes = new byte[16];
        secureRandom.nextBytes(rndBytes);
        return rndBytes;
    }

    public static void main(String[] args) throws Exception{
        // something like: yMvz6QeIjkBkcSiYgZvfMQ==
        System.out.println(new String(Base64.getEncoder().
            encode(SecureRandomGenerator.generate()), charsetName: "UTF-8"));
    }
}
```

Java Generate Keys



```
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import java.security.NoSuchAlgorithmException;
import java.util.Base64;

public class SimpleKeyGenerator {
    public static SecretKey generate256Key(){
        KeyGenerator generator = null;
        try {
            generator = KeyGenerator.getInstance("AES");
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
        generator.init( keysize: 256);
        SecretKey key = generator.generateKey();
        return key;
    }

    public static void main(String[] args) throws Exception{
        // something like: Et3U0I0z2fQyeBoDVW1jMIKMRBNRFoJIvBP3N2V+2hk=
        System.out.println(new String(Base64.getEncoder().
            encode(SimpleKeyGenerator.generate256Key().getEncoded()
            ), charsetName: "UTF-8"));
    }
}
```

Java Encrypt



```
import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import java.nio.charset.StandardCharsets;
import java.util.Base64;

public class Encryptor {
    public static String encrypt(String plaintext, SecretKey key, byte[] iv){
        try{
            Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
            SecretKeySpec spec = new SecretKeySpec(key.getEncoded(), "AES");
            IvParameterSpec ivSpec = new IvParameterSpec(iv);
            cipher.init(Cipher.ENCRYPT_MODE, spec, ivSpec);
            byte[] cipherText = cipher.doFinal(plaintext.getBytes(StandardCharsets.UTF_8));
            return new String(Base64.getEncoder().
                encode(cipherText), StandardCharsets.UTF_8);
        }catch(Exception e){
            throw new RuntimeException(e);
        }
    }
}
```

Java Encrypt



```
public static void main(String[] args) throws Exception{
    SecretKey key = SimpleKeyGenerator.generate256Key();
    System.out.println("Key: " + new String(
        Base64.getEncoder().encode(key.getEncoded()), StandardCharsets.UTF_8));
    byte[] iv = SecureRandomGenerator.generate();
    System.out.println("IV: " + new String(
        Base64.getEncoder().encode(iv), StandardCharsets.UTF_8));
    System.out.println("Ciphertext: " + Encryptor.encrypt(
        plaintext: "this is a test message!",
        key, iv));
    // it will print something like:
    // Key: oVMYqd8m0qQ7dfhtvLow4aKbmcNt6VeuDq7L9Hnn/kQ=
    // IV: sl/btZyVkW02bMnyESCh/A==
    // Ciphertext: kfweYmu0G5UcJ0mrQ7MeKTy1l5HPynRf8zYcDjvX/4c=
}
```

Java Decrypt



```
import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import java.nio.charset.StandardCharsets;
import java.util.Base64;

public class Decryptor {
    public static String decrypt(String ciphertext, SecretKey key, byte[] iv){
        try{
            Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
            SecretKeySpec spec = new SecretKeySpec(Base64.getDecoder().decode
                (key.getEncoded()), algorithm: "AES");
            IvParameterSpec ivSpec = new IvParameterSpec(Base64.getDecoder().decode(iv));
            cipher.init(Cipher.DECRYPT_MODE, spec, ivSpec);
            byte[] plaintext = cipher.doFinal(Base64.getDecoder().
                decode(ciphertext.getBytes(StandardCharsets.UTF_8)));
            return new String(plaintext, StandardCharsets.UTF_8);
        }catch(Exception e){
            throw new RuntimeException(e);
        }
    }
}
```

Java Decrypt



```
public static void main(String[] args) {  
    SecretKey key =  
        new SecretKeySpec(  
            new String( original: "oVMYqd8m0qQ7dfhtvLow4aKbmcNt6VeuDq7L9Hnn/kQ=").  
                getBytes(StandardCharsets.UTF_8), algorithm: "AES"  
        );  
    byte[] iv = new String( original: "sL/btZyVkW02bMnyESCh/A==").getBytes(StandardCharsets.UTF_8);  
    System.out.println("Plaintext: " +  
        Decryptor.decrypt(  
            ciphertext: "kfweYmu0G5UcJ0mrQ7MeKTy1L5HPynRf8zYcDjvX/4c=",  
            key, iv  
        ));  
}
```



Keys & Data Rotations

- ❑ Use cases: Regular or Breach
- ❑ Standard Industry Practice
- ❑ In AVG around 90 days
- ❑ Downtime vs No-Downtime
- ❑ Can have reliability implications
- ❑ Requires Data Catalogs
- ❑ Operationality, Observability, Rowbacability challenges

Envelope Encryption



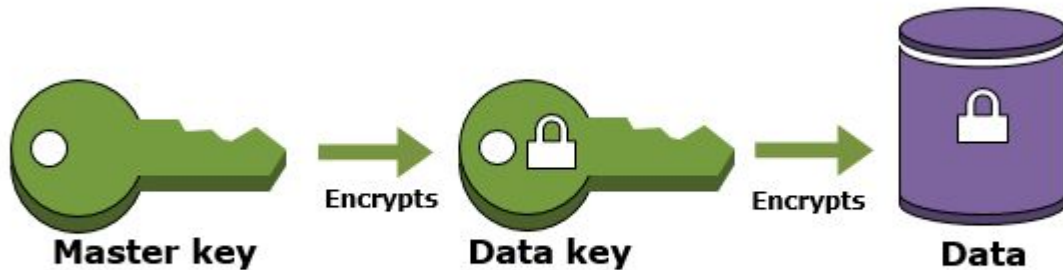
- ❑ More Secure
- ❑ Industry Standard
- ❑ Keys with the Data
- ❑ Key Encrypting: Keys, Metadata and Data
- ❑ Bootstrapping problem
- ❑ KMS



AWS KMS



- ❑ Uses Cloud HSM to protect the Keys
- ❑ Keys never leave AWS
- ❑ Remote Encryption always
- ❑ Symmetrical and Asymmetrical encryption
- ❑ Integrated with IAM and other services(EBS, S3, Kinesis, RedShift, etc...)
- ❑ Cloud Trail



AWS KMS

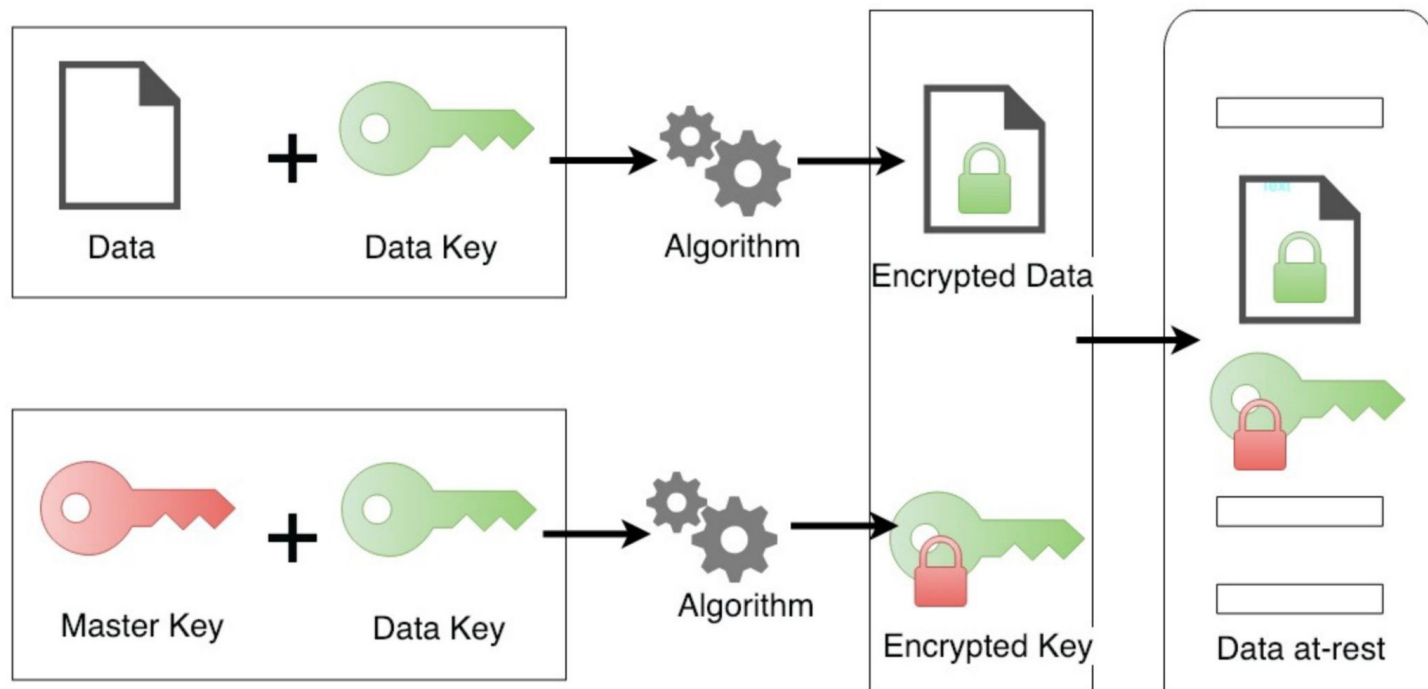
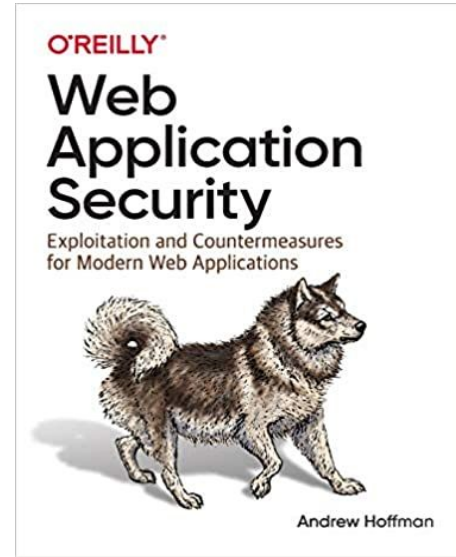
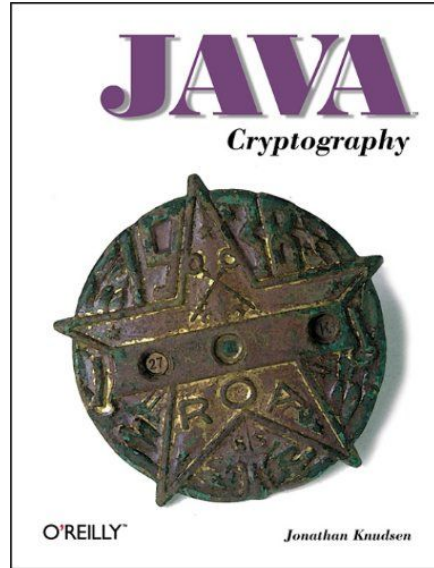
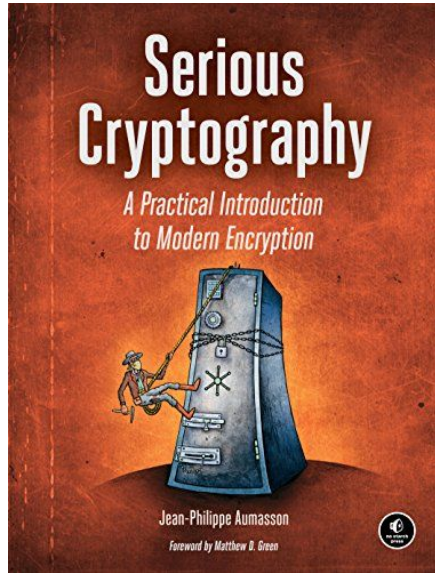


Figure 1: Envelope Encryption

Books



Encryption Deep Dive

Diego Pacheco

