Todd Leonhardt
todd.leonhardt@gmail.com

Practical Cryptography Engineering

Basic Introduction to Cryptography

- With a focus on the practical aspects
- Example code on GitHub

Overview

- Topics Covered
  - The context of cryptography - how it fits in to a system
  - Introduction to cryptography - basic kinds of algorithms
  - Practical aspects many people mess up
  - Available libraries of cryptographic primitives
  - Focus on usage independent of platform or language
- What is NOT covered
  - Mathematics behind any of the algorithms
  - Cryptanalysis
  - Theory

The Context of Cryptography

Applies to cyber security in general

What is Cryptography?

- The art and science of encryption/decryption
- But nowadays it is much broader
- Also covers
  - Authentication
  - Digital Signatures
  - Secure key exchange algorithms

The Context of Cryptography

- Cryptography is not magic security dust
  - Can't sprinkle it over your software to make it secure
- Security is only as strong as the weakest link
  - Mathematics of crypto is almost never the weakest link
- If you strengthen one part of your crypto
  - An adversary will attack a weaker part
- Fundamentals of crypto are important
  - How they are implemented and used is more important
- It's the things around the crypto that make the crypto effective (or ineffective)

The Role of Cryptography

- Cryptography by itself is useless
  - It has to be part of a much larger system to be useful
- A lock by itself is a useless thing
  - It needs to be part of a larger system to be useful
    - A door on a building, a chain, a safe, etc.
  - Larger system even extends to the people who use it
    - They need to remember to actually lock it
    - Need to not leave the key around in the open
- The same goes for cryptography
  - It is just a small part of a much larger security system
    - **<u>But it is a very critical part</u>**

The Role of Cryptography (con't)

- Cryptography has to:
  - Provide access to some people but not to others
    - This is tricky. Much easier to keep everyone out.
  - Distinguish between "good" access and "bad" access
- Crypto provides a natural point of attack
  - Along with its surrounding elements
- Crypto only useful if rest of system is secure
  - Why attack crypto if easily exploitable vulns. exist
    - SQL Injection, Cross-site scripting, buffer overflow
  - Still important to get crypto right even if weaknesses
    - Low chance of detecting broken crypto

The Weakest Link Property

- **A security system is only as strong as its weakest link**
- Assumptions:
  - Every security system consists of multiple parts
  - Smart opponents will attack the weakest link
- Implications:
  - Security systems are fiendishly hard to get right
  - To improve the security of a system, we must improve the weakest link
    - To do that, we need to know what the links are and which ones are weak

The Adversarial Setting

- Big difference between security and other types of engineering
- Most engineers have to deal with problems:
  - Storms, heat, wear and tear, CPU usage, memory, etc.
  - All of these factors affect designs
    - But their effect is fairly predictable with experience
- Not so in security systems
  - Our opponents are intelligent, clever, and malicious
    - They will do things nobody ever thought of
    - They don’t play by the rules

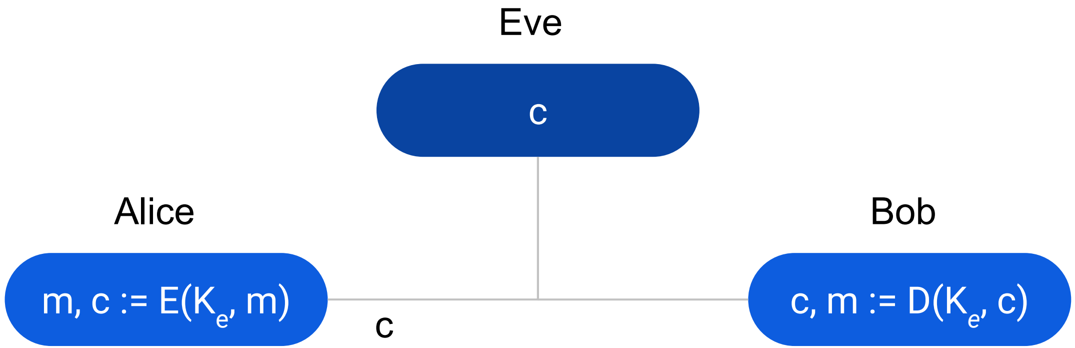
The Security Mindset

- To work in this field
  - You have to become devious yourself
  - You have to learn to think like a malicious attacker
    - To find weaknesses in your own work
  - You have to start thinking about how to attack systems
- Developing the professional paranoia mindset
  - Will help you observe things about systems and your environment that most people don’t notice
  - Assume that all parties involved other than yourself are not only malicious but are actively conspiring together against you

Introduction to Cryptography

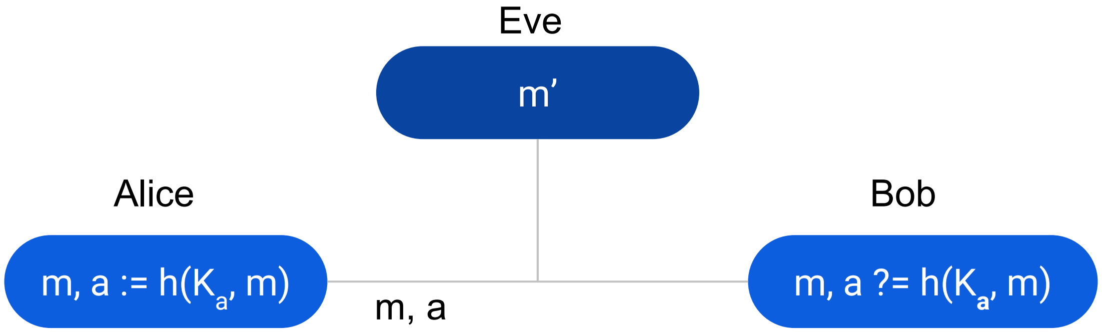
Encryption (symmetric)

- Alice and Bob want to communicate securely
- Eve is eavesdropping on the channel
- Alice sends message **m** to Bob, Eve intercepts
- To prevent Eve from understanding, Alice and Bob use encryption
- Alice and Bob first agree on a secret key, $\mathbf{K_e}$
  - Must do this via communication channel that Eve cannot eavesdrop on
- Alice first encrypts *plaintext* message **m** using encryption function $\mathbf{E(K_e, m)}$
  - Results is the *ciphertext* **c**
- When Bob receives **c**, he decrypts it with decryption function $\mathbf{D(K_e, c)}$



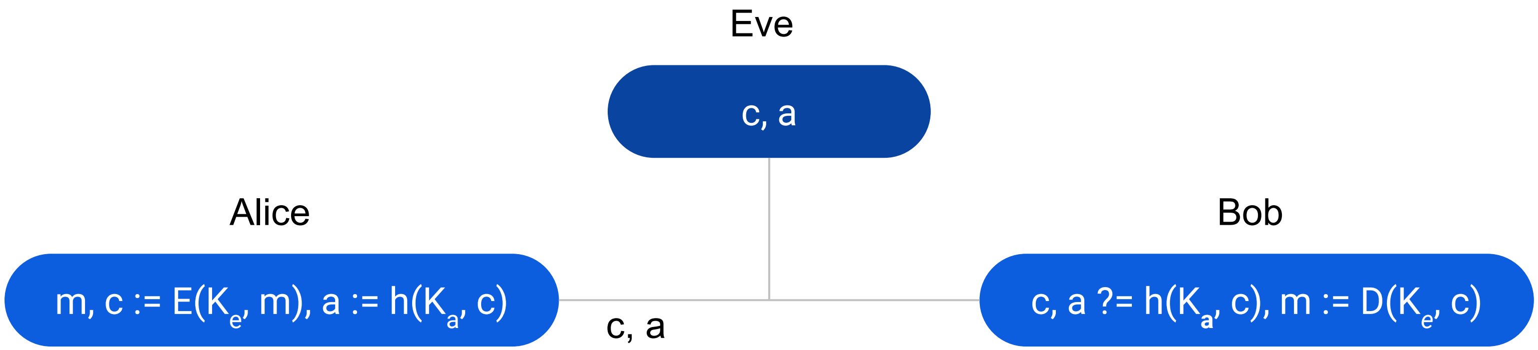
Authentication

- Eve could do more than just listen in on the message - she can change it
    - Alice sends message **m**, but Bob receives a different message **m’**
    - Eve could also delete a message or record it and send it to Bob later
- How does Bob know who sent the message he receives?
- Authentication resolves this problem and uses a secret key, $\mathbf{K_a}$
- When Alice sends **m**, she computes a *message authentication code* (MAC)
    - Compute MAC **a** as $\mathbf{a := h(K_a, m)}$, where h is the MAC function
- Alice now sends both **m** and **a** to Bob
- When Bob receives **m** and **a**, he recomputes what **a** should have been
- Bob will discard the message if **a** doesn’t match



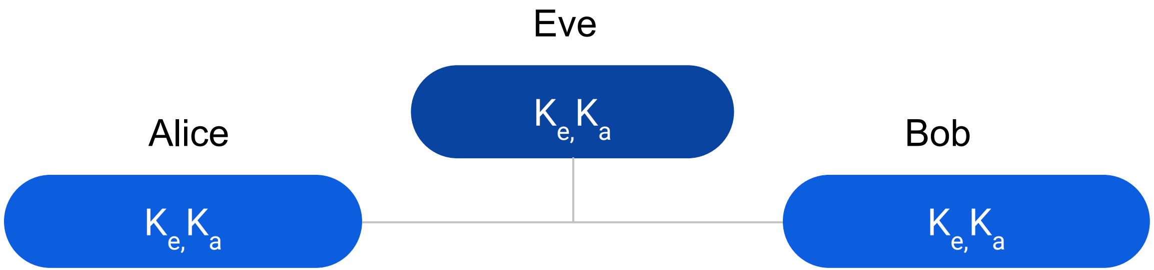
Encryption + Authentication

- In most situations Alice and Bob will want both encryption and authentication
- Never confuse encryption and authentication
    - Encrypting a message doesn't stop manipulation of its contents
    - Authenticating a message doesn't keep the message secret
    - Even though they both use a secret key, they are very different beasts
        - NOTE: It is generally important that $\mathbf{K_e}$ and $\mathbf{K_a}$ are <u>different</u> keys
- Always encrypt first and then compute the MAC on the ciphertext
    - Only way to guarantee integrity of the ciphertext



Key Distribution

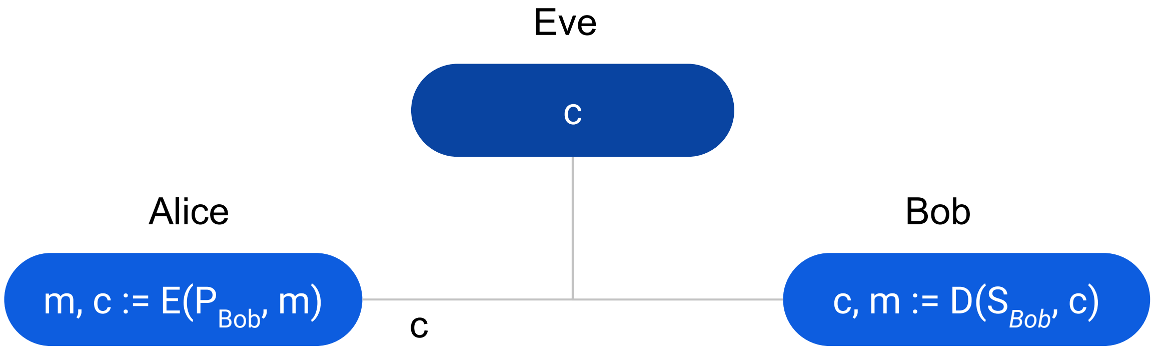
- Alice and Bob must share the secret keys $\mathbf{K_e}$ and $\mathbf{K_a}$
    - Alice can't just send the key to Bob over the communication channel
        - Eve would then have the key too



- Alice and Bob could exchange the key when they meet in person
    - A group of 20 friends - each member would have to exchange 19 keys
        - All in all, the group would have to exchange a total of 190 keys
    - Group of size N, need a total of N * (N - 1) / 2 keys ≅$N^2/2$

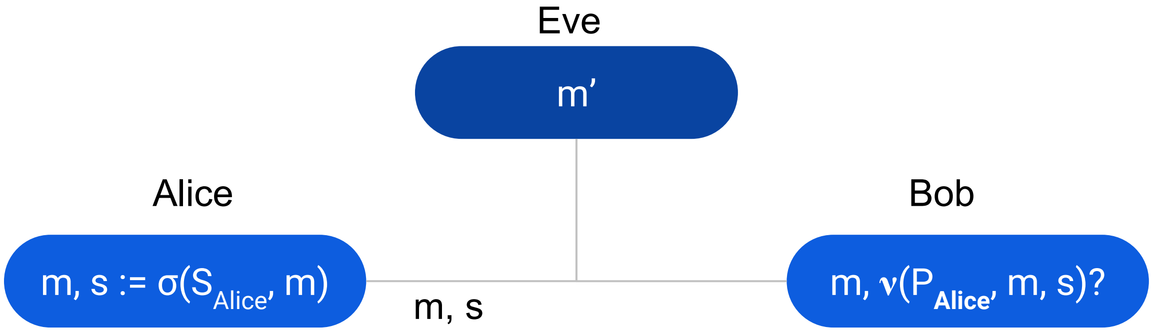
Public-key Encryption (asymmetric)

- Alice and Bob no longer use the same key - they use different keys
  - The key to encrypt a message is different from the key to decrypt it
- Bob first generates a pair of keys ($\mathbf{S_{Bob}}$, $\mathbf{P_{Bob}}$) using a special algorithm
  - The two keys are the secret key $\mathbf{S_{Bob}}$ and the public key $\mathbf{P_{Bob}}$
- Bob then publishes $\mathbf{P_{Bob}}$ as his public key
  - This makes $\mathbf{P_{Bob}}$ accessible to everyone, including both Alice and Eve
- When Alice wants to send a message to Bob, she first obtains $\mathbf{P_{Bob}}$
  - Alice encrypts the message **m** with the public key $\mathbf{P_{Bob}}$ to get ciphertext **c**
- Bob uses his secret key $\mathbf{S_{Bob}}$ and the decryption algorithm to decrypt **c**



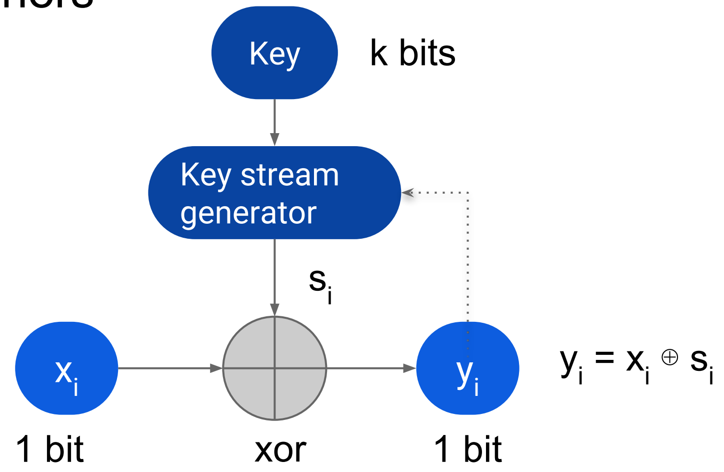
Digital Signatures

- Digital signatures are public-key equivalent of message authentication codes
  - This time it is Alice who generates a key pair ($\mathbf{S}_{\mathbf{Alice}}$, $\mathbf{P}_{\mathbf{Alice}}$)
  - Alice then publishes her public key $\mathbf{P}_{\mathbf{Alice}}$
- When Alice wants to send a signed message **m** to Bob
  - She computes a signature $\mathbf{s := \sigma(S_{Alice}, m)}$ and sends **m** and **s** to Bob
- Bob uses verification algorithm that uses Alices' public key to verify signature
  - $\boldsymbol{\nu}\mathbf{(P_{Alice}, m, s)}$
- The signature works just like a MAC
  - Except that Bob can verify it with the public key
  - Whereas the secret key is required to create a new signature



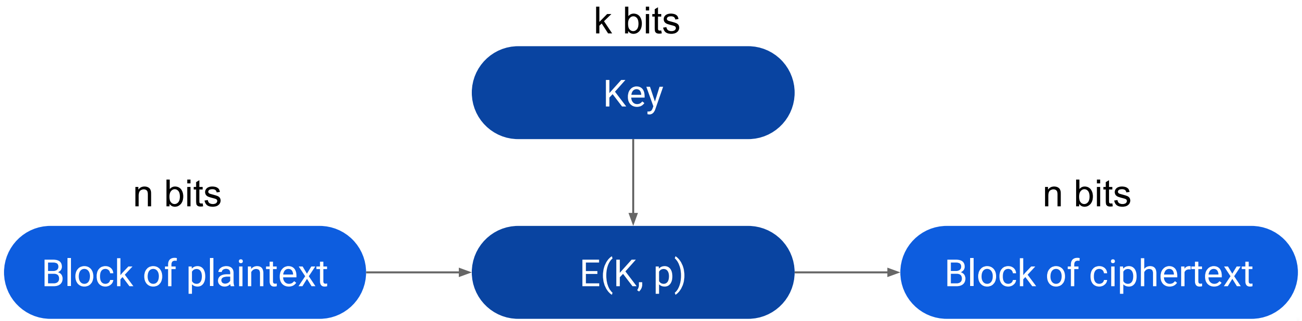
Stream Ciphers

- A *stream cipher* is an encryption function that encrypts bits individually
  - Achieved by adding a bit from a *key stream* to a plaintext bit
- Two types of stream ciphers
  - Synchronous - key stream depends only on the key
  - Asynchronous - key stream depends on the key and on the ciphertext
    - If the dotted line below is present, it is asynchronous
- Most practical stream ciphers are synchronous ones
- Example stream ciphers
  - [Salsa20]
  - [ChaCha20]
  - [XChaCha20]



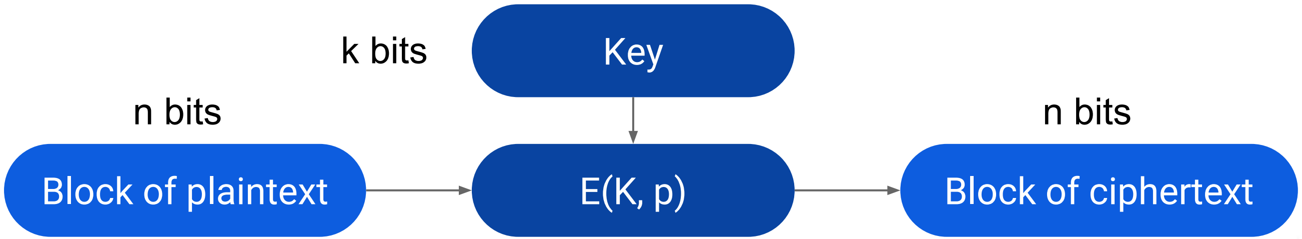
Block Ciphers

- A *block cipher* is an encryption function for fixed-size blocks of data
- The current generation of block ciphers has a block size of 128 bits (16 bytes)
  - They encrypt a 128-bit plaintext and generate a 128-bit ciphertext
  - They are reversible - a decryption function exists
- Encryption with a block cipher requires a secret key
  - Common key sizes are 128, 192, and 256 bits
  - Encryption: **c = E(K, p)** for plaintext p with key K
  - Decryption: **p = D(K, c)** for ciphertext c with key K
- Block ciphers should never be used directly
  - Instead, you should use a *block cipher mode*



Block Ciphers - Examples

- Canonical examples
  - AES (Advanced Encryption Standard)
    - n = 128 bits, k = 128, 192, or 256 bits
    - **AES is the current "gold standard"**
  - 3DES (Triple DES), where DES = Data Encryption Standard
    - n = 64 bits, k = 168 bits (3 x 56 bits)
    - Never use 3DES, it is a very old and very weak algorithm
- Alternative block ciphers to AES that might be more secure, but are slower
  - Serpent (n = 128 bits, k = 128, 192, or 256 bits) - 32 rounds vs AES 14
  - Twofish (n = 128 bits, k = 128, 192, or 256 bits) - key-dependent S-boxes
  - Rijndael (AES algorithm) with a bock size larger than 128 bits (256 bits)

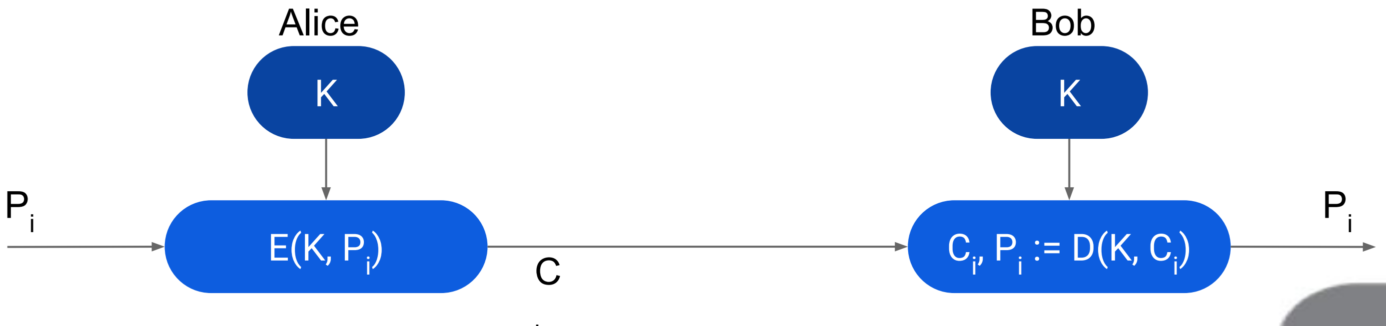


Block Cipher Modes

- Block ciphers encrypt only fixed-size blocks.
- To encrypt something longer than a block, you need a *block cipher mode*
    - Just a name for an encryption function built using a block cipher
- Many modes require length of plaintext to be an exact multiple of block size
    - These modes require some padding - many different ways to do this
        - Must be reversible - must be possible to determine length of original
        - Padding makes the ciphertext longer than the un-padded plaintext
        - After decrypting the ciphertext, the padding has to be removed
- Common Block Cipher Modes
    - ECB (Electronic Code Book) - always use same key, **NEVER USE THIS**
    - CBC (Cipher Block Chaining)
    - CTR (Counter)
    - AEAD Modes which combine Encryption + Authentication
        - GCM (Galois Counter Mode = Counter mode with GMAC)
        - CCM (Counter mode with CBC-MAC)

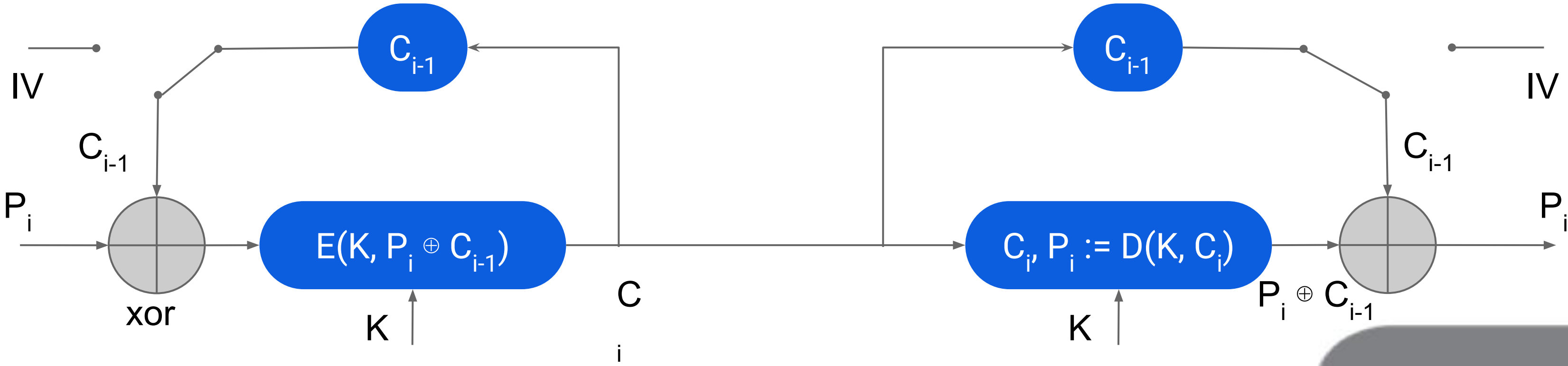
ECB (Electronic Code Book) Mode

- ECB is the simplest method to encrypt a longer plaintext
  - $\mathbf{C_i = E(K, P_i)}$ for $i = 1, \ldots, k$
  - You just encrypt each block separately using the same key
- **<u>DO NOT EVER USE ECB</u>** for anything
  - It has serious weaknesses
- What is the trouble with ECB
  - If two plaintexts are the same, then corresponding ciphertexts are same
    - This is visible to an attacker
    - This can leak quite a lot of information to the attacker



CBC (Cipher Block Chaining) Mode

- CBC is one of the most widely used block cipher modes
  - ECB problems avoided by XORing each plaintext with previous ciphertext
  - $\mathbf{C_i = E(K, P_i \oplus C_{i-1})}$ for $i = 1, \ldots, k$
- Problems of ECB avoided by “randomizing” plaintext using previous ciphertext
  - Equal plaintext blocks will encrypt to different ciphertext blocks
    - Significantly reduces information available to an attacker
- We are still left with the question of which value to use for C_0
  - This value is called the *initialization vector* or IV
  - There are many different strategies for picking the IV

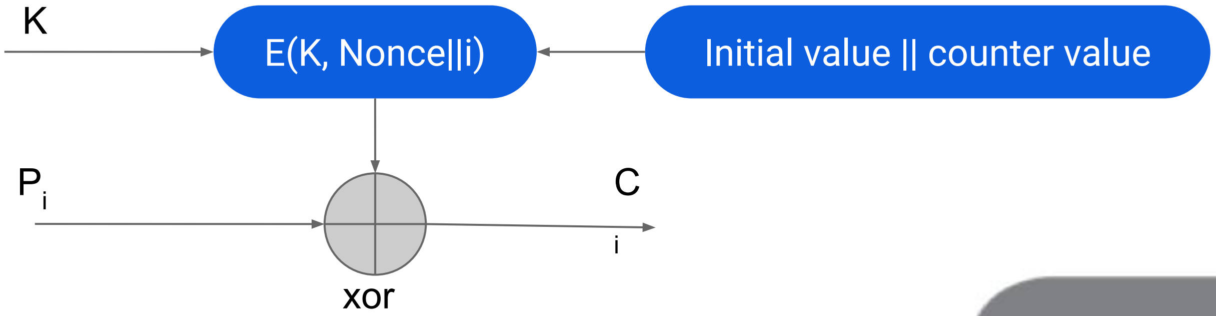


How to choose an IV

1. Fixed IV
   - You should never use a fixed IV - ECB problem for the 1st block
2. Counter IV
   - Use IV = 0 for 1st message, IV = 1 for 2nd message, etc.
   - Not a very good idea - many messages start in similar ways
3. Random IV
   - Recipient of the message needs to know the IV
     - Need to send it unencrypted before the encrypted message
   - Main disadvantage is that the ciphertext is one block longer than plaintext
4. Nonce-Generated IV
   - First each message is given a unique number called a *nonce*
     - Nonce = number used once - critical that it is unique
     - Should never use same nonce twice with the same key
     - Typically the nonce is a message number of some sort
     - Nonce does not need to be secret, just unique
   - Then IV is generated by encrypting the nonce

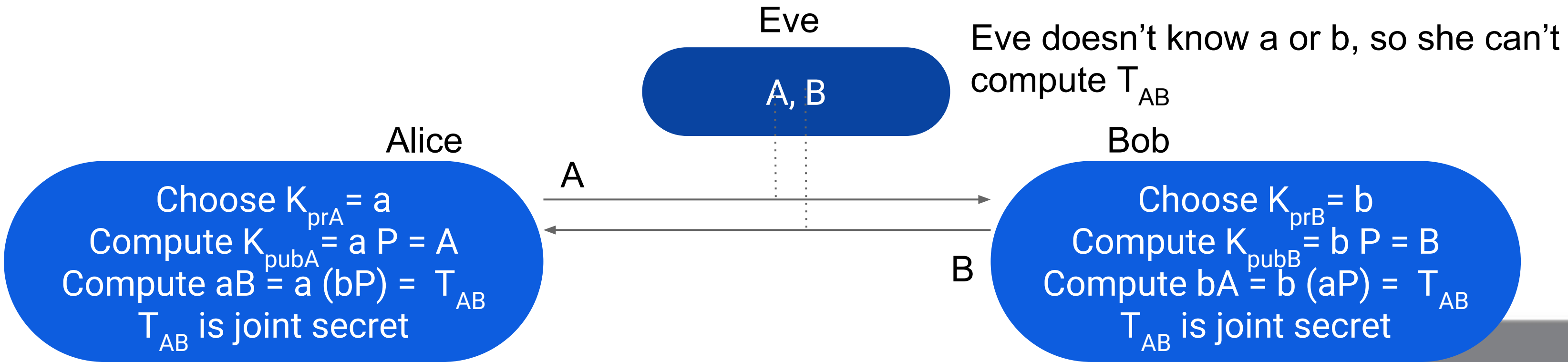
CTR (Counter) Mode

- CTR is a block cipher mode which actually creates a stream cipher
  - $\mathbf{K_i = E(K, Nonce||i)}$ for i = 1, …, k; where || represents concatenation
  - $\mathbf{C_i = P_i \oplus K_i}$
  - Like any stream cipher, you must supply a unique nonce of some form
- The message itself if never used as an input to the block cipher
  - The block cipher is used to generate a pseudo-random stream of bytes
    - Called the *key stream*
- CTR uses a remarkably simple method to generate the key stream
  - It concatenates the nonce with the counter value and encrypts it
    - Requires that the counter and the nonce fit in a single block



Elliptic-Curve Diffie-Hellman (ECDH)

- ECDH is a practical real-world solution to the key distribution problem
  - Enables two parties to derive a common secret key
    - By communicating over an insecure channel!
  - Agree in advance on a specific elliptic curve w/ primitive element **P**
- Mathematics behind it involves use of a one-way function
  - Easy to compute in one direction: A = a P
  - Extremely hard to invert: a = A/P
- The shared secret established can be used directly as a symmetric key
  - Or it can be used with a key-derivation function (KDF) to derive a key



Real World Crypto Attack Example

Nintendo Switch Hack of 1.0.0 firmware

Nintendo Switch Console Security

- Technical writeup on reddit detailing TrustZone code execution exploit
- TrustZone is highest privilege level in newer ARM SoCs (above kernel)
  - Provides hardware-based security and cryptography
- Switch uses an Nvidia Tegra X1 processor (Quad-core ARM Cortex-A57)
  - Contains Boot and Power Management Processor (BPMP)
    - Manages bootup and deep sleep mode
- In deep sleep, power is cut to the SoC and everything other than main DRAM
  - TrustZone Secure RAM (TZRAM) gets backed up to DRAM (encrypted)
    - Including all TrustZone code and state
  - A short warmboot firmware restores TrustZone code/state on wakeup
    - If you have a copy of BPMP firmware, you know how
      - BPMP runs LK littlekernel embedded kernel firmware (GitHub)
        - Wakeup firmware is stored at fixed DRAM address
- On wakeup, it copies the TrustZone context back into TZRAM
  - And decrypts it in-place using a fixed keyslot (#2) and AES-256-CBC

Chain of Epic Crypto Fails

- TZRAM decryption uses AES-256-CBC with a **fixed all-zeros IV**
- Then uses same slot to calculate an AES-256-CMAC over the decrypted blob
  - i.e. uses **MAC-then-Encrypt** instead of Encrypt-then-MAC
- Warmboot doesn’t initialize the keyslot it decrypts Trustzone with
  - That keyslot must be set prior to deep sleep and is restored at wakeup
- Bootrom restores Security Engine from fixed DRAM address (encrypted)
  - Decrypts **fixed-size blob** using AES-128-CBC with **fixed all-zeros IV**
  - **Validates** by verifying **last plaintext block matches fixed pattern**
    - If the “known pattern” matches, bootrom loads the context
    - **Otherwise** it sets the engine’s **context to be entirely zero**
- How to control the key used to decrypt TrustZone
  - **Corrupt the last block**, TrustZone will be **decrypted with all zeros key**
  - But we can do better!: AES-CBC decryption is a random-access cipher
    - Plaintext of block i depends only on ciphertext for blocks i and i-1
    - Security Engine won’t detect modifications to blocks other than last 2
    - 2nd-to-last block stores unused data, so don’t matter

Cryptographic Libraries

Never implement cryptographic primitives!

Cryptographic Libraries

OS Library	Windows	macOS	Linux	Android	iOS	Embedded
OpenSSL	X	X	X	X	X	X
libsodium	X	X	X	X	X	X
cryptography (PyCA)	X	X	X			X*
mbedTLS (ARM)	X	X	X	X	X	X
Common Crypto (Apple)		X			X	
CryptoNG (Microsoft)	X					
LibreSSL (OpenBSD)	X	X	X	X	X	
BoringSSL (Google)	X	X	X	X	X	X

Examples

Example code on GitHub:

https://github.com/tleonhardt/practical_cryptography_engineering

Final Thoughts

Which crypto should I use?

- Encryption
  - Use an AE algorithm which combines authentication
    - XSalsa20-Poly1305
  - Or an AEAD mode also authenticates added data
    - AES256-GCM or AES256-CCM
    - ChaCha20-Poly1305 or XChaCha20-Poly1305
- Key Exchange
  - Use Elliptic-curve Diffie-Hellman (ECDH)
- Digital Signatures
  - Use Elliptic-curve algorithm: ECDSA or Ed25519
  - Or use RSA-PSS probabilistic signatures

Where to learn more

- Books
  - Cryptography Engineering (2010) by Neils Ferguson, Bruce Schneier, and Tadayoshi Kohno
  - Understanding Cryptography (2010) by Christoph Paar, Jan Pelzl, and Bart Preneel
    - Website: http://www.crypto-textbook.com
    - YouTube lecture videos
    - Solutions Manual, Lecture Slides
- Online Courses
  - Cryptography I by Stanford on Coursera
  - Applied Cryptography by Univ. Virginia on Udacity

Backup Slides

Kerckhoffs's Principle

"A cryptosystem should be secure even if everything about the system, except the key, is public knowledge."

- Auguste Kerckhoffs (19th century Dutch cryptographer)

- The enemy knows your system/algorithm
  - They are capable of reverse engineering it quickly
- Security through obscurity doesn't work
- Only trust published and analyzed crypto algorithms
  - Don't trust secret algorithms

Authentication is a partial solution

- Eve can still record messages and their MACs
  - And then replay them by sending them to Bob at a later time
  - Eve can also delete or reorder messages from Alice to Bob
- Authentication is almost always combined with a numbering scheme
  - Number the messages sequentially
  - Then Bob is not fooled by Eve when she replays old messages
- Authentication combined with message numbering solves *most* of the problem
  - Eve can still stop Alice and Bob from communicating by deleting
  - Or delay messages by first deleting them and then sending them later
  - But deleting or delaying messages is about the extent of what she can do
- To avoid loss of information
  - Alice and Bob will often use a scheme of resending messages
    - This is more application specific
    - And outside the realm of cryptography

Public-Key Key Distribution

- Public-key cryptography makes the problem of distributing keys a lot simpler
  - Now Bob only has to distribute a single public key that everybody can use
  - Alice publishes her public key in the same way
  - N people can communicate securely by sharing only N keys
    - These keys do NOT need to be shared in secret!
- Why do we bother with secret-key encryption if public-key is so much easier?
  - Because public-key encryption is much less efficient
    - By several orders of magnitude
- In practice, almost always see a mix of public-key and secret-key algorithms
  - Public-key algorithms are used to establish a secret key
  - This secret key is in turn used to encrypt the actual data
  - This combines the flexibility of public-key with efficiency of secret-key
- Other reasons not to use asymmetric (public-key) cryptography:
  - A quantum computer will demolish all widely used public-key algorithms
  - Major governments will likely have quantum computers by late 2020's!
  - NIST is developing post-quantum cryptography by 2025