CITS3007 Secure Coding
Cryptography best practices

Unit coordinator: Arran Stewart

Highlights

- XOR operations
- Block ciphers and modes
- Encryption worst practices

XOR operations

Encrypting with a key

Suppose we have a plaintext of some length n bytes, and a key of the same length.

Is there a way of combining the two to encrypt the message?

XOR operation

One of the most convenient and flexible ways of doing so is to use the XOR logical operation (sometimes indicated using the $\oplus$ symbol).

Applied to single bits, the XOR operation is defined as follows:

A	**B**	**Output**
0	0	0
0	1	1
1	0	1
1	1	0

XOR in C

In C, the "^" (circumflex) operator performs bitwise XOR-ing on two integers.

Bitwise operations can be applied to both signed and unsigned integer types (but we usually stick to unsigned).

```
uint8_t a   = 0b11011010;   // 218 in decimal
uint8_t b   = 0b01110100;   // 116 in decimal
uint8_t res = a ^ b;        // 174 in decimal
```

XOR in C

```
uint8_t a   = 0b11011010;   // 218 in decimal
uint8_t b   = 0b01110100;   // 116 in decimal
uint8_t res = a ^ b;        // 174 in decimal
```

a	1	1	0	1	1	0	1	0
b	0	1	1	1	0	1	0	0
a ^ b	1	0	1	0	1	1	1	0

XOR in Python

- Python adopts the `^` operator from C (as do many other languages)
- It also provides convenient functions for displaying the bits in a number.

```
>>> a   = 0b11011010   # 218 in decimal
>>> b   = 0b01110100   # 116 in decimal
>>> res = a ^ b
>>> bin(res)
'0b10101110'
```

Properties of bitwise XOR

Some useful properties of bitwise XOR are:[1]

- It is symmetric – $A \oplus B$ is the same as $B \oplus A$
- If we take some number, XOR it with a second number K, and then XOR it with K again – we get back our original number.
- In other words, for any numbers A and K

$$(A \oplus K) \oplus K = A$$

[1] These assume both our inputs are of the same size. If our inputs are of different sizes, the smaller will be "widened" so that it's of the same type as the larger.

One-time pads

The property that

$$(A \oplus K) \oplus K = A$$

means we can use the XOR operation to encrypt *and* decrypt using a key.

One-time pads

- Assume our plaintext is a byte sequence P
- Assume our key is a completely random byte sequence K of the same length.
- We can encrypt P by XORing it with K
- and if the recipient has the key, they can decrypt by XORing it with K again

plaintext	1	1	0	1	1	0	1	0
key	0	1	1	1	0	1	0	0
ciphertext	1	0	1	0	1	1	1	0

Security of one-time pads

One-time pads are one type of cryptography that we can *prove* are information-theoretically secure, and impossible to break.[2]

We need the following conditions for that to be true:

- The key must be at least as long as the plaintext.
- The key must be generated from a random, uniformly distributed non-algorithmic source (e.g. a hardware random number generator).
- The key must never be reused
- The key must be kept completely secret by the parties using the encryption.

[2] Claude Shannon proved in 1949 that one-time pads have a property he called "perfect secrecy" – they give a cryptanalyst no information about the plaintext, except its maximum possible length.

Security of one-time pads

- Why is a one-time pad unbreakable?
- Because if the key was perfectly random, the ciphertext, too, will be indistinguishable from a random sequency of bytes.
- There literally is no pattern in the ciphertext that could be used to break it.
- For, say, a 20-byte message, there are $2^{8\times20}$ possible keys it could have (around 10^{48}).
- No way to break encryption
  - If we try all possible keys, we'll generate all possible plaintexts of 20 bytes length
  - We have no way of knowing which is correct.

Drawbacks of one-time pads

So why not use one-time pads for all encryption?

The main drawback is we need to have some way of distributing the keys to all recipients of a message, and we need some way of agreeing with them which key to use.

XOR uses

One-time pads have fallen out of use in modern times, but we still use the XOR operation for encryption and decryption.

Some examples:

Stream ciphers In a stream cipher, we have some continuous source (the keystream) of pseudorandom bytes.

To encrypt, we XOR our plaintext with this keystream.

At the other end, the recipient needs some way of calculating the same keystream, and can use it to decrypt the ciphertext.

Cryptographic primitve XOR is used a building block in more complex cryptographic algorithms. It's used in the AES (Advanced Encryption Standard) and in block cipher "modes".

Block ciphers and modes

Cipher block modes

When using block ciphers, the cipher operate on chunks or **blocks** of fixed length.

Let's suppose we've split our plaintext into a number of blocks. How do we actually encrypt them?

Cipher block modes

- We choose what's called a *block mode*, which is a way of applying the cipher to the blocks.
- There are many modes – we will just look at a couple of illustrative examples.

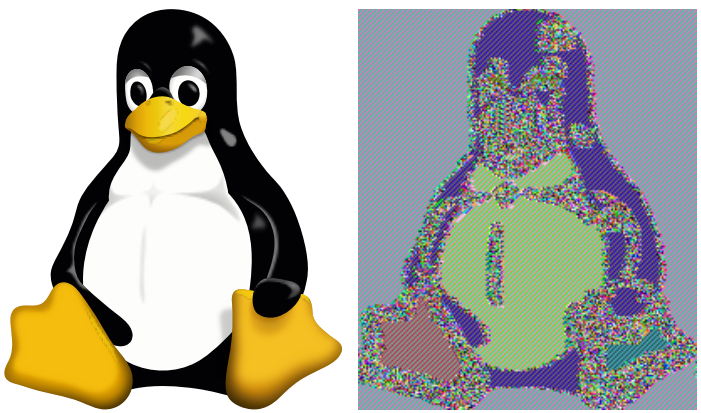
ECB ("Electronic Code Book")

ECB mode is a mode you almost certainly *shouldn't* choose.

In ECB

- Each block is encrypted independently, using the same encryption algorithm and key
- There is no feedback or "chaining" between blocks

As a result, two plaintext blocks with the same content will encrypt to the same ciphertext – patterns will be clearly visible.



ECB ("Electronic Code Book")

So why does ECB exist?

It's intended for secure transmission of *single* values (e.g. a key to be used with some other encryption method/protocol) – not for encrypting messages.

IV (Initialization Vector)

Block cipher modes other than ECB typically require a random **initialization vector** (IV) (also as nonce or salt).

- ECB is one of the few block modes that *doesn't* require an IV (part of what makes it insecure)
- IV serves similar purpose as password salts
- ensures the same plaintext does not encrypt to the same ciphertext when encrypted multiple times with the same key.
- prevents patterns from appearing in the ciphertext, which could be exploited by attackers

IV security requirements

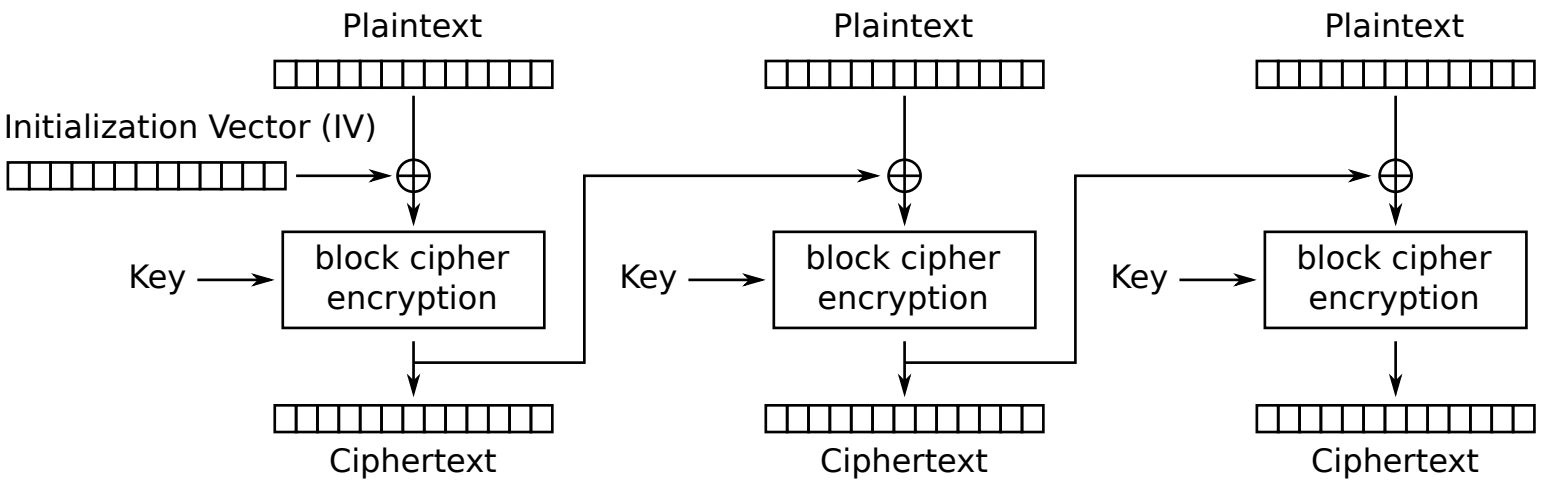
An IV has different security requirements from a key.

- The IV *usually* does not need to be kept secret.
- Nearly always, it's important never to re-use the same IV with the same key.
  $\Rightarrow$ Re-using an IV is a common security mistake
- The IV normally must be random.
  $\Rightarrow$ Using an IV that is predictable in some way is another common mistake.

CBC (Cipher Block Chaining)

In CBC mode:

- Each plaintext block is XORed with the ciphertext of the previous block, before being enctypted.
- For the first block, the IV is XORed with the plaintext block.



Cipher Block Chaining (CBC) mode encryption

Image source: Wikipedia

CBC (Cipher Block Chaining)

- This mode is very secure against eavesdroppers, even when the adversary can conduct a "chosen plaintext" attack (i.e. force encryption of particular chosen plaintexts to see how they're encrypted)

But must satisfy the following:

- Must use a secure block cipher
- Must generate a new, random IV for each messag

What cipher block mode to use?

The following block modes are commonly used, and secure when used properly:

- CBC (Cipher Block Chaining)
- CTR (Counter)

CTR (Counter)

- Each block of plaintext is XORed with an encrypted counter
- the counter is incremented for each subsequent block

Counter incrementing is fast, so useful where speed is a concern.

What cipher block mode to use?

Typical application of different modes:

ECB ("Electronic Code Book") Secure transmission of single values (e.g. a key)

CBC (Cipher Block Chaining) General-purpose block-oriented transmission, authentication

CTR (Counter) General-purpose block-oriented transmission, where high speed is required

Encryption worst practices

Not using encryption when you should

Example: Greg Myre, "How does Ukraine keep intercepting Russian military communications?" (NPR, 26 April 2022)

- Although Russia has a modern, secured radio system for military use, it often hasn't been used
- Russian troops brought their own mobile phones into Ukraine
- If mobile phone infrastructure is controlled by an adversary, they can disable particular numbers (availability) and breach confidentiality

Security through obscurity

Kerckhoff's Principle (Auguste Kerckhoff, 19th C)

- Security should depend only on the secrecy of the secret (private) key.
- Security should not depend on the secrecy of the algorithm itself.
  "[The cipher] must not be required to be secret, and it must be able to fall into the hands of the enemy without inconvenience."

Claude Shannon's take on it: assume that "the enemy knows the system" (Shannon's maxim).

If your security depends only on your keeping the algorithm obscure, that's poor security.

Security through obscurity

"Isn't relying on a secret key/password 'security through obscurity?"

No! It's the exact opposite.

Kerckhoff's Principle says that the key is the one thing that you *do* need to keep secure.

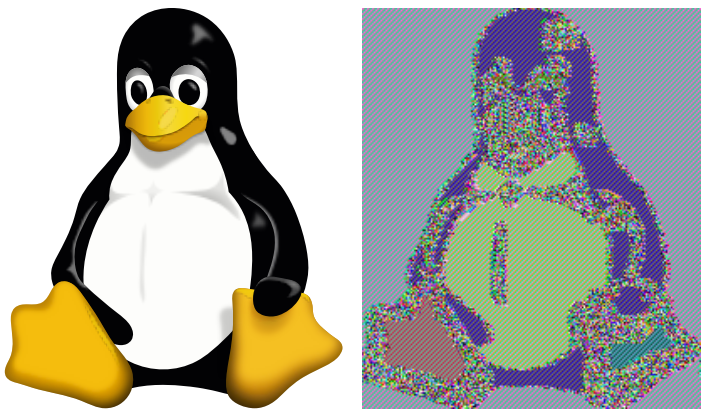
Hard-coded keys

- It can be very tempting to hard-code passwords and/or keys into the source of a program.
- However, anyone who can get hold of the binary will probably be able to work out the value of the key (through disassembly and reverse engineering)
- If secrecy of the key is breached ...
  - Your only way of updating the key is by releasing new versions of the program
  - Old versions will continue to use the now insecure key
- **Key management** is a complex area – usually easiest to use an off-the-shelf system which does key management for you

ECB mode for block ciphers

We have already seen this – ECB is a very bad choice of block mode.

The “ECB penguin”:



ECB mode for block ciphers

ECB mode was used in Zoom Bill Marczak and John Scott-Railton, "Move Fast and Roll Your Own Crypto – A Quick Look at the Confidentiality of Zoom Meetings" (April 3, 2020)

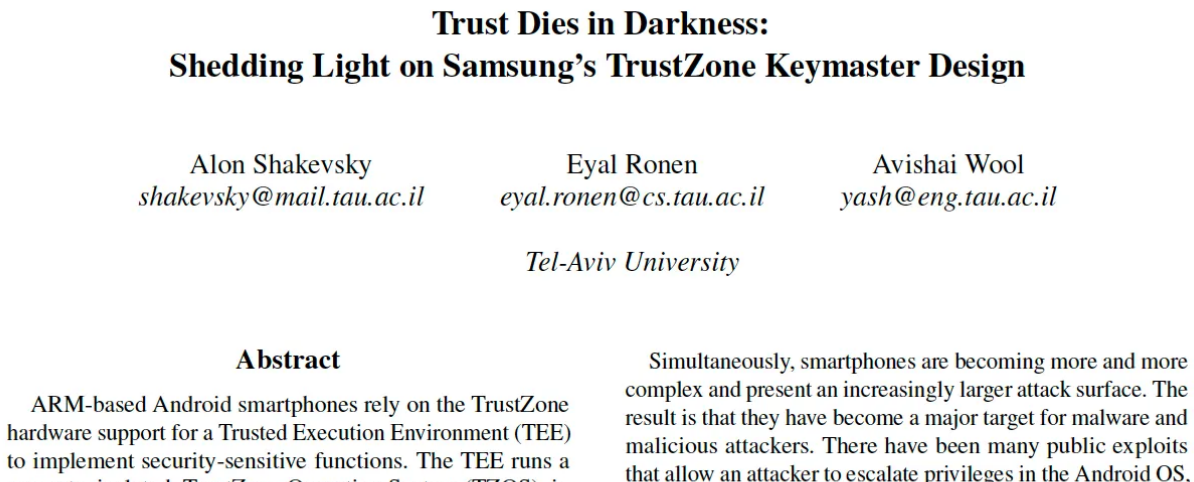
> *Zoom documentation claims that the app uses "AES-256" encryption for meetings where possible. However, we find that in each Zoom meeting, a single AES-128 key is used in ECB mode by all participants to encrypt and decrypt audio and video. The use of ECB mode is not recommended because patterns present in the plaintext are preserved during encryption.*

Poor IV practices – IV re-use

If an initialization vector (IV) is re-used, that can allow encryption to be broken.

Example: Crypto Bug in Samsung Galaxy Devices: Breaking Trusted Execution Environments (TEEs)

- Researchers analysed hardware backup store, reverse-engineered code, and found IVs were re-used and subject to attack
- Samsung Galaxy S8, S9, S10, S20, and S21 devices were analysed
- Attacks are CVE-2021–25490 and CVE-2021–25444.



Trust Dies in Darkness:
Shedding Light on Samsung's TrustZone Keymaster Design

Alon Shakevsky *shakevsky@mail.tau.ac.il*
Eyal Ronen *eyal.ronen@cs.tau.ac.il*
Avishai Wool *yash@eng.tau.ac.il*

Tel-Aviv University

Abstract

ARM-based Android smartphones rely on the TrustZone hardware support for a Trusted Execution Environment (TEE) to implement security-sensitive functions. The TEE runs a

Simultaneously, smartphones are becoming more and more complex and present an increasingly larger attack surface. The result is that they have become a major target for malware and malicious attackers. There have been many public exploits that allow an attacker to escalate privileges in the Android OS,

Poor IV practices – IV re-use/non-random IV

- We saw an example last lecture of very poor use of the PyCrypto library – SaltStack developers chose a poor public exponent for public-key ceyptography
- But also, the library API is fairly badly designed in a way that encourages misuse
- Example: to encrypt using symmetric-key encrpytion, PyCrypto has an optional parameter for the IV
- But the default value is IV $= 0$, which leads to encryption being insecure.
- Better API design: never allow a default value for the IV

Poor choice of hash functions for password storage

Recall that good hash functions should have the following properties

- Difficult to invert (also called "preimage resistance")
- Difficult to find a second input that "collides" with the first (also called "second preimage resistance")
- Slow to compute (because the faster it is to compute, the easier for attackers to run brute-force attacks)

Poor choice of hash functions for password storage

There are hash functions specifically designed for password hash storage:

- pbkdf2, bcrypt, scrypt and argon2

But many systems use hash functions (e.g. MD5, SHA1) which *aren't* designed for password storage.

Those hash functions may be fine for error detection, but typically lack one of the properties we want for a password hash.

Passwords != keys

A **password** is something memorable to people, normally consists of printable characters only, and can be of arbitrary length.

A **key** is a sequence of *bytes* (not necessarily printable), and is of some fixed length. (E.g. 128-bit key)

If you need to turn a password into a key, typically you need to apply a password hash function to it

- one of pbkdf2, bcrypt, scrypt or argon2