Patrick Favre-Bulle Follow

Software Engineer currently working on Android Applications supporting Contactless EMVCo Payments with HCE/SE

Jan 6 · 9 min read

# Security Best Practices: Symmetric Encryption with AES in Java and Android

In this article I will give you a primer on the Advanced Encryption Standard (AES), common block modes, why you need padding and initialization vectors and how to protect your data against modification. Finally I will show you how to easily implement this with Java avoiding most security issues.


So, I made this "hacker-encryption-security-image" hope you like it.

## What every Software Engineer should know about AES

AES, also known by its original name Rijndael, was selected by the NIST in 2000 to find a successor for the dated Data Encryption Standard(DES). AES is a block cipher, that means encryption happens on fixed-length groups of bits. In our case the algorithm defines 128 bit blocks. AES supports key lengths of 128, 192 and 256 bit.

Every block goes through many cycles of transformation rounds. I will omit the details of the algorithm here, but the interested reader is referred to the Wikipedia article about AES. The important part is that the key length does not affect the block size but the number of repetitions of transformation rounds (128 bit key is 10 cycles, 256 bit is 14)

> *Until May 2009, the only successful published attacks against the full AES were side-channel attacks on some specific implementations. (Source)*

## Want to encrypt more than one Block?

So AES will only encrypt 128 bit of data, but if we want to encrypt whole messages we need to choose a block mode with which multiple blocks can be encrypted to a single cipher text. The simplest block mode is Electronic Codebook or ECB. It uses the same unaltered key on every block like this:

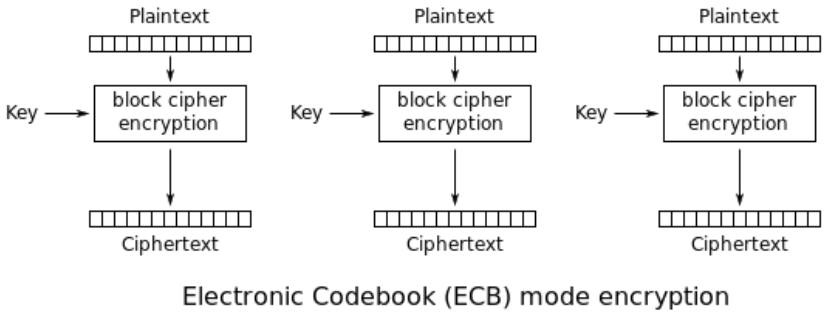



Image from Wikpedia

This is particularly bad since identical plaintext blocks are encrypted to identical ciphertext blocks.

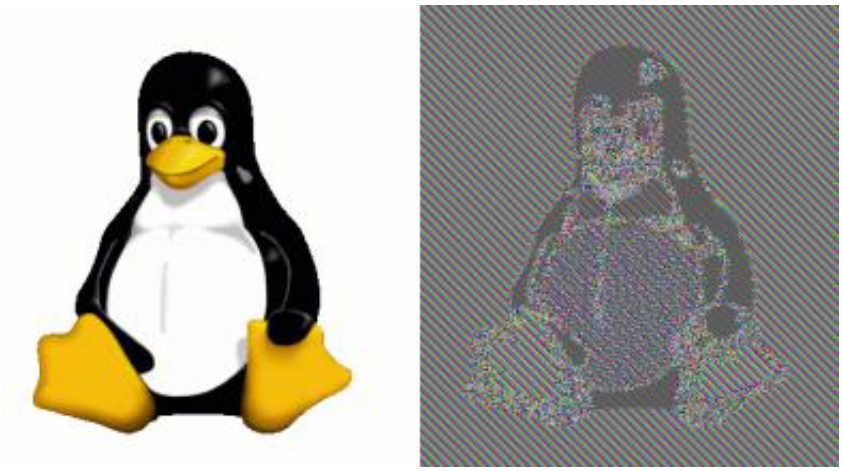


Image encrypted with ECB block mode reveals patterns of the original (Image from Wikpedia)

Remember to **never choose this mode unless you only encrypt data smaller than 128 bit.** Unfortunately it is still often misused because it does not require you to provide an initial vector (more about that later) and therefore *seems* to be easier to handle for a developer.

One case has to be handled with block modes though: what happens if the last block is not *exactly* 128 bit? That's where **padding** comes into play, that is, filling the missing bits of the block up. The simplest of which just fills the missing bits with zeros. There is practically no security implication in the choice of padding in AES.

## Cipher Block Chaining (CBC)

So what alternatives to ECB are there? For one there is CBC which XORs the current plaintext block with the previous ciphertext block. This way, each ciphertext block depends on all plaintext blocks processed up to that point. Using the same image as before the result would be noise not distinguishable from random data:

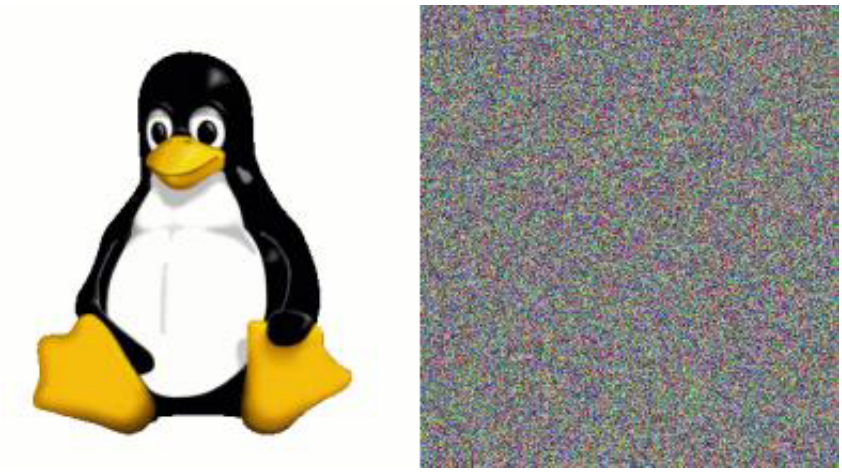


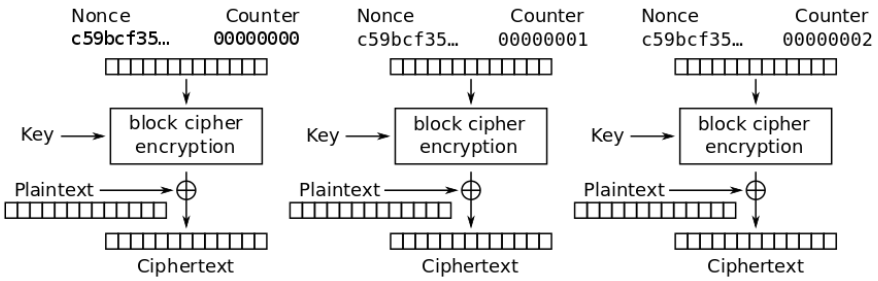
Image encrypted with CBC block mode looks random (Image from Wikpedia)

So what about the first block? The easiest way is to just use a block full of e.g. zeros, but then every encryption with the same key and plaintext would result in the same ciphertext. Also if you reuse the same key for different plaintexts it would make it easier to recover the key. A better way is to use a random **initialization vector (IV).** This is just a fancy word for random data that is about the size of one block (128 bit). Think about it like the **salt** of the encryption, that is, a IV can be public, should be random and only used one time. Mind though, that not knowing the IV will only hinder the decryption of the first block since the CBC XORs the ciphertext not the plaintext of the previous one.

When transmitting or persisting the data it is common to just prepend the IV to the actual cipher message.

## Counter Mode (CTR)

Another option is to use CTR mode. This block mode is interesting because it turns a block cipher into a stream cipher which means no padding is required. In its basic form all blocks are numbered from 0 to n. Every block will now be encrypted with the key, the IV (also called *nonce* here) and the counter value.




Counter (CTR) mode encryption

Image from Wikpedia

The advantage is, unlike CBC, encryption can be done in parallel and all blocks are depended on the IV not only the first one. A big caveat is, that an IV **must never be reused** with the same key because an attacker can trivially calculate the used key from that.

## Can I be sure that nobody altered my message?

The hard truth: *encryption does not automatically protect* against data modification. It is actually a pretty common attack. Read here on a more thorough discussion about this issue.

So what can we do? We just add Message Authentication Code (MAC) to the encrypted message. A MAC is similar to a digital signature, with the difference that the verifying and authenticating key are practically the same. There are different variations of this method, the mode that is recommend by most researchers is called Encrypt-then-Mac. That is, after encryption a MAC is calculated on the cipher text and appended. You would usually use Hash-based message authentication code (HMAC) as type of MAC.

So now it starts getting complicated. For integrity/authenticity we have to choose a MAC algorithm, choose an encryption tag mode, calculate the mac and append it. This is also slow since the whole message must be processed twice. The opposite side has to to the same but for decrypting and verifying.

## Authenticated Encryption with GCM

Wouldn't it be great if there were modes which handles all the authentication stuff for you? Fortunately there is a thing called authenticated encryption which simultaneously provides confidentiality, integrity, and authenticity assurances on the data. One of the most popular block modes that supports this is called **Galois/Counter Mode or GCM** for short (it is e.g. also available as a cipher suite in TLS v1.2)

GCM is basically CTR mode which also calculates an authentication tag sequentially during encryption. This authentication tag is then usually appended to the cipher text. Its size is an important security property, so it should be at least 128 bit long.

It is also possible to authenticate additional information not included in the plaintext. This data is called **associated data.** Why is this useful? For example the encrypted data has a meta property, the creation date, which is used to check if the content must be re-encrypted. An attacker could now trivially change the creation date, but if it is added as associated data, GCM will also verify this piece of information and recognize the change.

## A heated discussion: What Key Size to use?

So intuition says: the bigger the better—it is obvious that it is harder to brute force a 256 bit random value than a 128 bit. With our current understanding brute forcing through all values of a 128 bit long word would require astronomically amount of energy, not realistic for anyone in sensible time (looking at you, NSA). So the decision is basically between infinite and infinite times $2^{128}$.

> *AES actually has three distinct key sizes because it has been chosen as a US Federal Algorithm Apt at being used in various areas under the control of the US federal government [including the military]. (…) So the fine military brains came up with the idea that there should be **three** "security levels", so that the most important secrets were encrypted with the heavy methods that they deserved, but the data of lower tactical value could be encrypted with more practical, if weaker, algorithms. (…) So the NIST decided to formally follow the regulations (ask for three key sizes) but to also do the smart thing (the lowest level had to be unbreakable with foreseeable technology) (Source)*

The argument follows: an AES encrypted message probably won't be broken by brute forcing the key, but by other less expensive attacks (not currently known). These attacks will be as harmful to 128 bit key mode

as to the 256 bit mode, so choosing a bigger key size doesn't help in this case.

So basically 128 bit key is enough security for most of every use case with the exception of quantum computer protection. Also 128 bit is faster than 256 bit.

. . .

## Implementing AES-GCM in Java and Android

So finally it gets practical. Modern Java has all the tools we need, but the crypto API might not be the most straigt forward one. A mindful developer might also be unsure what length/sizes/defaults to use. *Note: if not stated otherwise everything applies equally to Java and Android.*

In our example we use a randomly generated 128 bit key. Java will automatically choose the correct mode when you pass a key with 192 and 256 bit length. Note however, 256 bit encryption usually requires the JCE Unlimited Strength Jurisdiction Policy installed in your JRE (Android is fine).

```
SecureRandom secureRandom = new SecureRandom();
byte[] key= new byte[16];
secureRandom.nextBytes(key);
SecretKey secretKey = SecretKeySpec(key, "AES");
```

Then we have to create our initialization vector. For GCM a 12 byte (not 16!) random byte-array is recommend by NIST because it's faster and more secure. Be mindful to always use a strong pseudorandom number generator (PRNG) like SecureRandom.

```
byte[] iv = new byte[12]; //NEVER REUSE THIS IV WITH SAME
KEY
secureRandom.nextBytes(iv);
```

Then initialize your cipher. AES-GCM mode should be available to most modern JREs and Android newer than v2.3. If it happens to be not available install a custom crypto provider like BouncyCastle, but the

default provider is usually preferred. We choose an authentication tag of size 128 bit

```
final Cipher cipher =
Cipher.getInstance("AES/GCM/NoPadding");
GCMParameterSpec parameterSpec = new GCMParameterSpec(128,
iv); //128 bit auth tag length
cipher.init(Cipher.ENCRYPT_MODE, secretKey, parameterSpec);
```

Add optional associated data if you want (for instance meta data)

```
if (associatedData != null) {
    cipher.updateAAD(associatedData);
}
```

Encrypt; if you are encrypting big chunks of data look into CipherInputStream so the whole thing doesn't need to be loaded to the heap.

```
byte[] cipherText = cipher.doFinal(plainText);
```

Now concat all of it to a single message

```
ByteBuffer byteBuffer = ByteBuffer.allocate(4 + iv.length +
cipherText.length);
byteBuffer.putInt(iv.length);
byteBuffer.put(iv);
byteBuffer.put(cipherText);
byte[] cipherMessage = byteBuffer.array();
```

Optionally encode it with e.g. Base64 if you require a string representation. Android does have a standard implementation of this encoding, the JDK only from version 8 on. As an alternative check out Apache Commons Codec.

And that's basically it for encryption. For constructing the message, the length of the IV, the IV, the encrypted data and the authentication tag are appended to a single byte array. (in Java the authentication tag is automatically appended to the message, there is no way to handle it yourself with the standard crypto API).

It is best practice to try to wipe sensible data like a cryptographic key or IV from memory as fast as possible. Since Java is a language with automatic memory management, we don't have any guarantees that the following works as intended, but is should in most cases:

```
Arrays.fill(key,(byte) 0); //overwrite the content of key
with zeros
```

Be mindful to not overwrite data that is still used somewhere else.

Now to the **decrypt** part: It works similar to the encryption; first deconstruct the message:

```
ByteBuffer byteBuffer = ByteBuffer.wrap(cipherMessage);
int ivLength = byteBuffer.getInt();
byte[] iv = new byte[ivLength];
byteBuffer.get(iv);
byte[] cipherText = new byte[byteBuffer.remaining()];
byteBuffer.get(cipherText);
```

Initialize the cipher and add the optional associated data and decrypt:

```
final Cipher cipher =
Cipher.getInstance("AES/GCM/NoPadding");
cipher.init(Cipher.DECRYPT_MODE, new SecretKeySpec(key,
"AES"), new GCMParameterSpec(128, iv));
if (associatedData != null) {
    cipher.updateAAD(associatedData);
}
byte[] plainText= cipher.doFinal(cipherText);
```

That's it! If you like to see a full example check out my Github project Armadillo where I use AES-GCM.. . .

## Summary

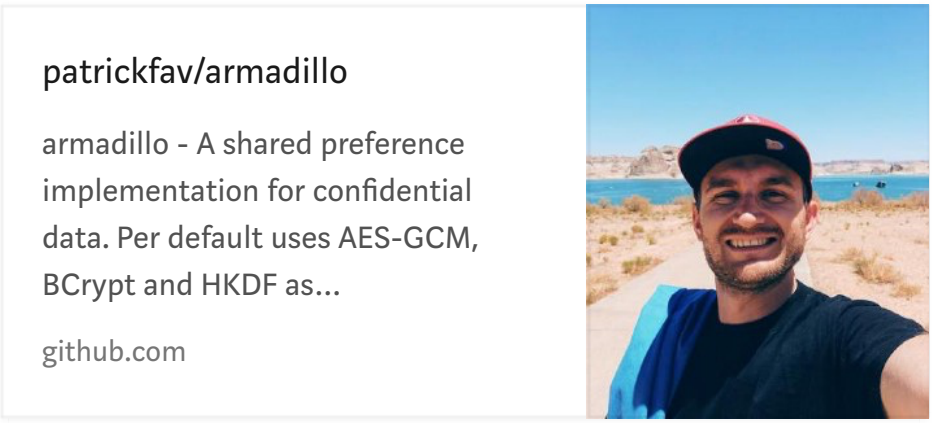
There are 3 properties we want for securing our data

- **Confidentiality:** The ability to prevent eavesdroppers from discovering the plaintext message, or information about the plaintext message.
- **Integrity:** The ability to prevent an active attacker from modifying the message without the legitimate users noticing.
- **Authenticity**—The ability to prove that a message was generated by a particular party, and prevent forgery of new messages. This is usually provided via a Message Authentication Code (MAC). Note that authenticity automatically implies integrity.

AES with Galois/Counter Mode (GCM) block mode provides all those properties and is fairly easy to use and is available in most Java/Android environments. Just consider the following:

- Use 12 byte initialization vector that is never reused with the same key (use a strong pseudorandom number generator like SecureRandom)
- Use 128 bit authentication tag length
- Use 128 bit key length (you will be fine!)
- Pack everything together into a single message

## References



## Advanced Encryption Standard - Wikipedia

The Advanced Encryption Standard ( AES), also known by its original name Rijndael ( Dutch pronunciation: ), is a...

en.wikipedia.org