Coursework 2023-2024: Galois/Counter Mode COMP3077 Cryptography

Michael Pound
March 25, 2024

Introduction

In this coursework you will need to complete an implementation of AES-GCM, an Authenticated Encryption with Associated Data (AEAD) encryption scheme. AES-GCM builds upon the popular counter (CTR) mode of operation. It combines a 128-bit block cipher, in this case the Advanced Encryption Standard, with an authenticated tag computed in $GF(2^{128})$. The result is that both the ciphertext (CT) and additional authenticated data (AAD) are verified as not having been changed before decryption.

A full implementation of AES-GCM requires working implementations of both AES, and multiplication in $GF(2^{128})$. In order to make this coursework more manageable, I have provided both of these implementations for you. More details can be found below, but in essence you are combining building blocks together into a working protocol, you do not need to implement all of the components yourself.

The structure of this document is as follows: First there will be an explanation of the code from which you will work, followed by an overview of AES-GCM and some suggested resources. We will end with some suggestions for implementation, and instructions on submission.

Note: I have provided some hints and tips throughout this document. Read this document carefully, and in full, before beginning implementation.

Java Project

To be consistent with the labs you have already done, a base project has been created for you. You can download a zip file containing the project using the link on moodle. This project makes use of gradle, so can be built and run in a number of other IDEs or on the command line. The project should load without issue, you may need to set up the SDK as shown in the video on moodle, as with the labs. The code is fairly straightforward at a glance, it contains implementations of both AES and GF multiplication coded to the `AES128Encryptor` and `GF128Multiplier` interfaces respectively.

AES Implementation

The AES implementation uses an optimised version that makes heavy use of pre-computed lookup tables. You need not worry about the implementation, the coursework can be completed by utilising only the two functions provided in the interface:

- `void init(byte[] key)`: Initialises AES using the 128-bit key provided as a 16 byte array. This implementation of AES does not support 192 or 256 bit keys.
- `void encryptBlock(byte[] input, byte[] output)`: Encrypts a single 128-bit block and copies the output to an array of the same size.

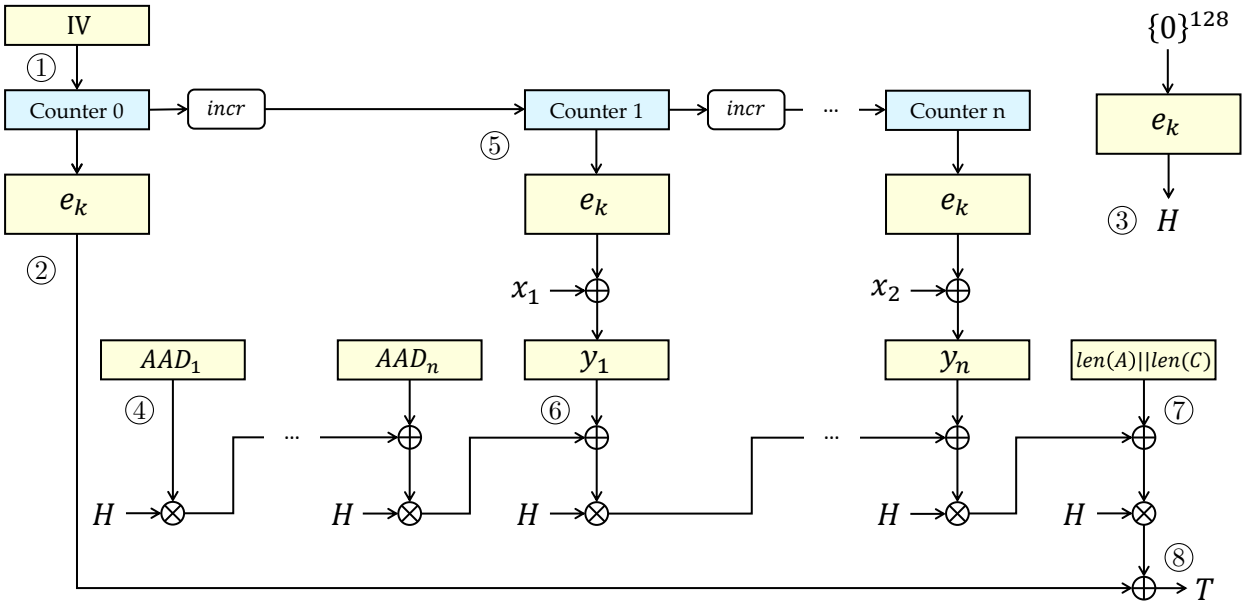

Figure 1: An diagram of the AES-GCM scheme. Circled numbers represent key steps described in this document, you can click them to see the corresponding note. Decryption using AES-GCM is almost identical; the ciphertext is added to the key stream to recover the message, but the ciphertext is still used to generate the tag.

$GF(2^{128})$ Implementation

The arithmetic during creation of the message tag in Galois/Counter mode is performed in $GF(2^{128})$, modulo an irreducible polynomial $P(x) = x^{128} + x^7 + x^2 + x + 1$. Addition is therefore XOR, and the multiplication implementation is very similar to the AES `gmul()` function from the labs, made more challenging primarily because there are now 128 bits rather than 8. The NIST spec also defines the polynomial in reverse order, which makes the implementation harder. The interface defines the following functions:

- `void init(byte[] H)`: Initialises the multiplier with the value H.
- `void multiplyByH(byte[] X)`: Multiplies the 128 bit polynomial represented by X by H, and stores the result in the original X array. I.e. $X = X \cdot H \pmod{P(x)}$. This function will do nothing if H has not been initialised.
- `void multiply(byte[] X, byte[] Y)`: A more general multiplication of two polynomials represented by X and Y. The result is stored in the original X array. I.e. $X = X \cdot Y \pmod{P(x)}$. This function can also be used to square a polynomial if the same parameter is passed for both X and Y.

Coursework Implementation

The project contains an interface `AEADCipher`, and a bare implementation `AESGCM` where you will add your code. As you complete parts of the implementation, further tests will pass, and you can also use these as a guide for what you should expect parts of the algorithm to do. Various test vectors are also included in this document.

AES-GCM Overview

A full description of AES-GCM can be found in the reference documentation from NIST. Important details of the algorithm are described below, but you may also find the official document useful, for example the test vectors in Appendix B. A diagram of AES-GCM encryption can be found in Figure 1. Circled numbers represent the key steps described here:

① A 96 bit IV is concatenated with a 32 bit counter that begins at 1. This is incremented for each new block required.

② The first counter is encrypted using AES under a 128 bit key to be used later as an additional mask applied to the final tag.

③ An all zero block is encrypted to produce the hash key H in $GF(2^{128})$.

④ Each block of AAD is added to the existing tag using XOR, and then multiplied by the constant H.

⑤ The counter is incremented and encrypted to produce new keystream blocks. These are added using XOR to the corresponding plaintext blocks to produce each ciphertext block.

⑥ Each block of ciphertext is added to the existing tag using XOR, and then multiplied by H.

⑦ The bit length of the AAD and ciphertext (represented as 64 bits each are concatenated together, added to the existing tag using XOR, and then multiplied by H.

⑧ The initial mask is added to the existing tag using XOR to produce the final tag T. Some implementations may truncate the tag, we will not do this.

Instructions

Use the provided gradle project as a starting point. The project contains numerous tests which will evaluate the functionality of your implementation. You should run the tests regularly to see how you are progressing, you will find that the tests provide a logical progression of steps from basic functionality through to more advanced encryption and decryption of arbitrary messages. You may also like to make use of the `main()` method if you wish to manually test various functions. The provided functions in `HexUtils` may be of particular use in parsing what is happening with the various intermediate values. Avoid the use of external libraries (except JUnit) that are not directly part of the JDK. I will be running your code through my own test suite, and if I have to hunt down other dependencies this may cost marks. You can add and implement any functions, classes etc you think are relevant to coding the cipher.

You should add your code into the `AESGCM` class, which implements the `AEADCipher` interface. You can add any other helper methods or code you wish into this file, as long as the interface remains as the tests use this. The primary functions you will need to implement are as follows:

- `void init(AEADParams params)`: Initialises the cipher using the key, IV and CipherMode supplied in the `AEADParams` class. The `CipherMode` enum specifies either encryption or decryption. Ciphers that have finished after a call to `finalise` or `verify` should be re-usable following a call to `init`.
- `void updateAAD(byte[] data)`: Adds a block of AAD for authentication. This AAD block should not be encrypted, but should be incorporated into the final tag as described in the overview above.

- `void processBlock(byte[] data)`: Encrypts or decrypts a single block of data inline, storing the resulting plaintext or ciphertext bytes into the same `data` array. The ciphertext should be incorporated into the final tag as appropriate.
- `void finalise(byte[] out)`: Finishes by calculating the final tag, storing it in the supplied `out` array. No tests will call any functions except `init` after calling finalise.
- `void verify(byte[] tag)`: Finishes by calculating the final tag, and comparing it to the supplied `tag` array. Should throw an `InvalidTagException` if the tag supplied as a parameter does not match the final tag computed by the AEAD scheme. As with `finalise`, you can assume that no further calls except reinitialisation will occur after `verify` has been called.

Test Classes

It may seem that there is no obvious starting point for the `AEADCipher` interface, with all of the functions looking equally important. However, some of the tests are much easier to pass than others, depending on the nature of the messages they require you to handle. You may find it easier to code your cipher to handle the easier tests first, and then systematically add functions later. Below is a short description of some of test classes you may wish to start on, in an *approximate* order of challenge.

- `NoTagTests`: These tests perform simple encryption using a few blocks of data, and do not require any complex tag implementation at all.
- `EmptyTests`: These tests require you to calculate a valid tag on an empty message.
- `AADTests`: These tests require you to calculate a tag that authenticates some AAD, but with no encryption.
- `PlaintextTests`: These tests require you encrypt blocks of plaintext, and authenticate the corresponding ciphertext within the tag, but provide no AAD blocks.
- `EncryptTests`: These tests require you to encrypt and calculate the correct tag given blocks of both plaintext and AAD.

The function of each set of tests should be clear from the name, but you can also look inside at the code they use. Within each test class you will find that the tests also vary in difficulty. The simplest are so-called `SingleBlock` tests, comprising only a single block of exactly 16 bytes that must be processed. Next we have `MultiBlock` tests that will supply multiple blocks of 16 bytes one at a time. Next we have `PartialBlock` tests that will supply a single short block of less than 16 bytes. Finally we have `MultiThenPartialBlock` tests which supply some number of full blocks, followed by a single partial block. None of these tests will require you to handle partial blocks in the middle of either AAD or encrypt/decrypt calls, and AAD will always be supplied first, followed by plaintext or ciphertext. No test will initialise the cipher using `CipherMode.ENCRYPT` and then pass ciphertext to `processBlock`, or vice versa. To avoid other unnecessary boilerplate you can also assume there will be no tests requiring you handle null data or zero length data.

Challenging Tests

A small selection of more challenging tests are included – which are ignored by default – for those who are feeling adventurous. Implementing the features required to pass any or all of these tests will be worth a

limited number of additional marks. The majority of marks will be given for the primary tests that are enabled by default, and for the quality of submissions in terms of code quality, clarity and efficiency.

The challenging tests are given below, and there are also comments within each text file identifying them. JUnit will ignore these tests by default, to enable them simply change the boolean `testsEnabled` to true at the top of the corresponding file.

- `ArbitraryIVTests`: GCM actually supports IVs of any length. This set of tests uses various sizes of IV down to a minimum of 8 bytes. Initialisation with IVs that are not exactly 12 bytes is handled differently. Information on this can be found in the GCM specification, but in essence any IV that is not exactly 12 bytes must be initialised by computing a GHASH, so in essence you are initialising GCM with a variant of GCM!
- `FastGFTests`: The provided `GFMultiplierImpl` is a basic implementation of multiplication in $GF(2^{128})$. Within the project is an empty `GFFastImpl` class, which you can implement if you want to try. Details on strategies for optimisation are in the GCM specification.
- `LateAADTests`: In all other tests AAD is passed in its entirety before any encryption or decryption is performed. this makes calculating the tag easier. The tests here require you to handle late calls to `updateAAD` after some calls to `processBlock` have already been performed. Doing this efficiently is a challenging mathematical problem. To solve it, consider the entire equation for the tag, which is actually a polynomial evaluated at `H`. For example, the equation for the tag for two blocks of AAD and three blocks of ciphertext is:

$$T = \underset{\text{Original}}{AAD_1 \cdot H^6 + AAD_2 \cdot H^5} + \underset{\text{Final}}{CT_1 \cdot H^4 + CT_2 \cdot H^3 + CT_3 \cdot H^2 + L \cdot H + E_k(ctr)}$$

  If AAD_2 was submitted late, then the difference between the final AAD tag and the original one passed to the ciphertext must be added back in, multiplied by some power of H.

Tips and Information

Unlike the labs I will be somewhat more vague about suggestions for the coursework! Of course, please do ask if you have a question. Here are some pointers to get you started:

- As above, make use of the tests I have supplied and the official test vectors if you wish to debug your code. In the `HexUtils` class within the tests you will find some hex-to-byte and vice versa functions that might be useful.
- As mentioned above, the structure of the interface and my tests assumes a somewhat reusable implementation of the cipher. I.e. that the cipher can reinitialise at any time if provided a new IV and key through a call to `init()`.
- Despite being labelled clearly as `Counter 0` in the official documentation, the first counter value is 1!
- I will be marking on a number of criteria, and there are many marks available simply for a working implementation, however it is implemented. This said, there are also plenty of marks available for good code, efficiency, readability, etc. Do not worry if you don't pass all of the tests, particularly the harder ones. I will still mark your work if some or all of the tests fail.

- **Do not plagiarise.** Discussing how AES-GCM works at an overview level, such as using the diagram above, is absolutely fine. Pseudo code is likely to be crossing a line, and definitely do not copy or share code amongst yourselves. It will be obvious, and will also flag up under a code plagiarism checker. Online implementations of AES-GCM obviously exist, most are hard to read and embedded in other complex libraries that are nothing like ours. I will likely run all submissions and online examples through JPlag. You will find that ChatGPT's efforts are hilariously bad.

Submission

Use gradle to compile, test and run your code. Once you are finished and ready to submit, please submit your full project as a zip file to moodle. You can do everything required in IntelliJ, or build using gradle off the command line from the project directory:

- **gradlew build**: builds the project
- **gradlew test**: reruns the tests
- **gradlew jar**: builds the jar file.

You can create and implement additional class files if you wish, but do not rename or change any interface e.g. `AEADCipher`, or `HexUtils`. These are used in both your and my test suites, as you can see if you read the test files.