Advanced Topics in Cybersecurity 2 Elisabeth Oswald

AES — Implementations

2024

1 Introduction

You must read the introduction of every lab sheet. It explains if you must install something, and if so, what you must install. It also explains the first steps to get going with a lab.

1.1 Requirements

All example code that I provide here has been tested under Python 3, using the PyCharm community edition UI (`https://www.jetbrains.com/pycharm/download/`). The Community Edition (scroll on the linked website) is free and runs reliably under all major OSs. If you do not have Python installed on your system, please do so. I can not support other programming languages in this course. If you elect to use a different (G)UI, then you do this at your own risk (in the sense that I cannot offer help and advice for anything but PyCharm).

If you are unfamiliar with Python or PyCharm, use the wondrous wild web to find one of the various tutorials. PyCharm helpfully enables you to create different "virtual environments" for different projects: in case you mess something up in one project, provided that you don't reuse the same venv for all projects, your other projects won't be affected.

The lab sheet assumes, as does the corresponding slide set, that you are familiar with basic algebra, i.e. you know what is a group, a field, a prime number, and you are familiar with modular arithmetic (at least modulo a prime number).

1.2 What you should get out of this.

In the next part of this course we are going to apply attacks to implementations of AES. These attacks make (in part) use of structural properties of the AES encryption round, and countermeasures must be tailored to specific properties of the AES round functions. Consequently, this lab gives you the chance to "play a bit" with AES.

1.3 Getting going

Create a project under PyCharm (the default options from PyCharm should be fine, i.e. Python 3) and then add the supplied files for this lab (they are on Moodle) to your project. It is up to you whether or not you connect/set up PyCharm with any Version control system that you might be using (it natively integrates with Github), or whether you do version control outside of PyCharm. But ensure you have some form of version control, to avoid the "I lost all my files before the deadline." embarrassment.

The file `AES.py` includes a number of useful functions to implement an AES (Advanced Encryption Standard) round. The file should execute, albeit it does not yet do an AES round, because some components yet need to be implemented.

2 The mathematics behind AES

AES, as defined in FIPS 197, encrypts a 128 bits at a time. The standard defines three different key lengths; in this lab, we only consider 128 bit keys. The standard not only defines how the different steps in an encryption round work, how the round keys are generated, etc.; it also explains how data must be organised and interpreted.

2.1 Understanding the AES state

The input bits are organised into 16 bytes, whereby the byte order and the bit order are precisely defined in the standard.

Check out, using the official standard, how the bits in an input block are organised. How do the byte ordering and the ordering of the bits within each byte differ?

In the AES implementation (fragment) in the file`AES.py`, the AES state is mapped faithfully into a two dimensional array (in Python this translates into a nested list of integers). This may look like the only (or most natural option): consider this perhaps in the context of processors where loads/stores are very expensive (much more expensive than operations that can take within the ALU — if you don't know what an ALU is, then either chat with a colleague or check within the wondrous wild web).

Could there be (good) reasons to organise the AES state differently within the memory (of a computer)?

2.2 AES field arithmetic

Whilst AES can be implemented using “bit-level” operations (i.e. exclusive-or, and, shifts), some of the AES round transformations (AddRoundKe, SubBytes, MixColumns) are based on finite field arithmetic. The interpretation as doing arithmetic over a finite field is fundamental to understand why AES is so resistant against classical structural attacks (this is outside of the scope of this course/lab sheet). It is also important to find/utilise implementation options that are better (either by being faster, smaller, or more secure).

Each (state) byte can be understood as an element of a finite field (if you don’t know what a mathematical field is, find the definition now, don’t read on until you have refreshed your understanding). The size of the finite field is 256. This implies that we have a slightly more interesting field (because typically we build finite fields by choosing a prime modulus). The AES finite field is a so-called extension of the finite field with only two elements. An extension can be defined by “replicating” the finite field with two elements $\mathbb{F}_2$ eight times (somehow). We now illustrate how this works.

I want to do this via an analogy with simple prime finite fields. In a prime finite field, e.g. $\mathbb{F}_5$, we have the five elements $\{0, 1, 2, 3, 4\}$ which can also be understood as “residue classes”. This means that every number x in $\mathbb{N}$ can be “mapped” to one of these five numbers/residue classes by means of calculating what is the residue modulo 5 (we know that $x = k \cdot 5 + r$, and $r \leq 4$).

Now to replicate the prime finite field $\mathbb{F}_2$ we need some mathematical object that “extends” beyond being a single number. A concept that you should be familiar with is that of a vector space, where we organise elements into vectors. We thus want to have “vectors” with eight bits (this enables us to represent a byte). For computing with multiple eight-bit numbers,we need to define an operation over the vectors space that allows us to reduce any number that is larger than 255, and that preserves the useful property of having a finite field.

The way to go about solving this problem is by considering the vector elements as though they are “positions” (like with e.g. decimal numbers, each digit is associated with a position relative to power of 10): so any value a can be written as $a_7 \cdot x^7 + a_6 \cdot x^6 + \ldots a_0$, with a_i coming from the bit representation of a. If we think along these lines, then it is clear that we need some polynomial by which we can reduce numbers (aka polynomials) that are larger than 255. In other words, we need a polynomial “equivalent” to a prime number.

Such an "equivalent" is called an irreducible polynomial. In the case of AES, the irreducible polynomial was selected to be $x^8 + x^4 + x^3 + x + 1$.

Check out the implementation of the finite field arithmetic functions in `AES.py`. Explain addition and multiplication (this should be easy). Then look at the implementation of the finite field multiplication, find out how it works, so that you could produce a pen an paper example. Finally, consider the special case of `xTimes`: firstly find out what this is using the AES standard, and then provide an implementation that does not use the provided `GF28_multiply()` function.

2.3 Getting familiar with the AES round functions

The file `AES.py` already implements the `SubBytes, ShiftRows` and `AddRoundKey` functions. Check them out to see how the state is accessed. There is also a pretty print function for the state, to help you print intermediate steps to the console.

However, `MixColumns` is missing; perhaps there could be a more efficient way to do `SubBytes, ShiftRows`, and it is unclear where the `sbox` really comes from. Thus I want you to fill in the "gaps", using the FIPS 197 standard.

- Implement MixColumns
- Implement a new function that *generates the sbox* table
- Consider combining `SubBytes, ShiftRows`

If you do at least MixColumns, you can get the script to do one round of AES, and use the test vectors in the standard to check correctness!

If you want to have the entire encryption, you also need to implement the key schedule.

Finally, I want you to consider what happens, for each round function, if we wanted to apply it to inputs where each byte is exclusive-ored with a another value m. So consider what happens if we apply `AddRoundKey, SubBytes, ShiftRows` to $a \oplus m$ rather than a, and what happens if we apply `MixColumns` to $(a_0 \oplus m_0, a_1 \oplus m_1, a_2 \oplus m_2, a_3 \oplus m_3)$. It may not be obvious why this matters (although perhaps you want to speculate)?