

## COMS30901 lab. worksheet #3

- Both the hardware and software in MVB-2.11 is managed by the IT Services Zone E team. If you encounter a problem (e.g., a workstation that fails to boot, an error when you try to use some software, or you just cannot log into your account), they can help: either talk to them directly in room MVB-3.41, or submit a service request online via

<http://servicedesk.bristol.ac.uk>

- We intend this worksheet to be attempted, at least partially, in the associated lab. session. Your attendance is important, since this session represents the primary source of formative feedback and help for COMS30901. Perhaps more so than in units from earlier years, *you* need to actively ask questions of and seek help from either the lectures and/or lab. demonstrators present.
- The worksheet is not assessed *at all*: the purpose is to provide help via a tutorial-style overview of selected technologies and concepts. If you are confident you already understand the content, there is no problem with nor penalty for totally ignoring it.

Before you start work, download (and, if need be, unarchive<sup>a</sup>) the file

<http://tinyurl.com/zetunur/csdsp/crypto/sheet/lab-3.tar.gz>

somewhere secure<sup>b</sup> in your file system: it is intended to act as a starting point for your own work, and will be referred to in what follows.

<sup>a</sup>Use the `gz` and `tar` commands within a BASH shell (e.g., in a terminal window), or the archive manager GUI (available either via the menu Applications→Accessories→Archive Manager or by directly executing `file-roller`) if you prefer.

<sup>b</sup>For example, the Private sub-directory in your home directory.

The AES block cipher is *everywhere*: you can argue the details, but it is fair to say *any* security-critical task you undertake is likely to involve AES in some way. Partly due to this ubiquity, a good software library, such as OpenSSL, will provide an AES implementation you can use. Therefore, treating such an implementation as a black-box (i.e., by ignoring the internal detail) is already enough in many contexts. On the other hand however, investigating this detail can provide a range of more general insight, and often represents a useful learning exercise.

AES is a fairly “clean” design, combining a simple high-level structure (based on an SP-network) with close connection to underlying Mathematics at lower-levels. Numerous resources may be used to support study of the design, ranging from

- the definitive, but formal FIPS 197 [1] standard available at

<http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>

- relevant video-based overviews, such as

<http://www.youtube.com/channel/UC1usFRN4LCMcIV7UjHNuQg>

presented by Christof Paar, that includes block cipher material in

[http://www.youtube.com/watch?v=x1v2tX4\\_dkQ](http://www.youtube.com/watch?v=x1v2tX4_dkQ)

[http://www.youtube.com/watch?v=NHuibtoL\\_qk](http://www.youtube.com/watch?v=NHuibtoL_qk)

<http://www.youtube.com/watch?v=4FBgb2uobWI>

and

- less formal introductions such as

<http://www.moserware.com/2009/09/stick-figure-guide-to-advanced.html>

An arguably simpler, more hands-on approach would be to develop your own AES implementation. The goal of this worksheet is to do precisely this, using a programming language of your choice to develop an implementation based on FIPS 197 [1] (which is used as a reference throughout).

## Step-by-step implementation of AES using FIPS 197

The remit in what follows is deliberately limited: although you will produce a working implementation of AES, it will

1. support AES-128 (i.e., a key size of  $n_k = 128$  bits) encryption only, and
2. represent a specific implementation strategy often used on constrained platforms (a typical example being an 8-bit micro-controller, such as an Atmel AVR or Intel 8051): this implies a trade-off favouring memory footprint over efficiency.

This is partly intended to manage the workload involved (since there are a *huge* range of alternatives), but also to maximise utility within the corresponding coursework assignment (more specifically any attack that targets AES).

### Step #1: support functionality

The design of AES is fundamentally based on the structure provided by  $\mathbb{F}_{2^8}$ , a finite field of  $2^8 = 256$  elements. Mirroring the lecture(s), [1, Section 3 and 4] provides a brief overview split into roughly two parts. First, one needs to consider the representation of data. The pertinent points can be summarised as

- construction of each  $\mathbb{F}_{2^8}$  using binary polynomials modulo a fixed irreducible polynomial,
- representation of field elements in  $\mathbb{F}_{2^8}$  using an unsigned 8-bit integer, or byte, data type (e.g., `uint8_t` in C), and
- representation of  $(4 \times 4)$ -element state and round key matrices as (column-major), 16-element arrays (e.g., `uint8_t s[ 16 ]` in C).

Second, one needs to consider how to perform the required computation with said data: this aspect introduces a range of options, again covered by the lecture(s). In short, AES depends on three central operations whose implementation (in isolation) is your first task.

Implement a function to support each of the following:

Implement  
(task #1)

1. given a field element  $a \in \mathbb{F}_{2^8}$ , produce  $a(x) \cdot x \pmod{p(x)}$  (i.e., multiply  $a$  by  $x$ ),
2. given a field element  $a \in \mathbb{F}_{2^8}$ , produce  $S\text{-Box}(a)$  (i.e., apply the S-box), and
3. given an integer  $i \in \mathbb{Z}$ , produce  $rc_i = x^i \pmod{p(x)}$  (i.e., the  $i$ -th round constant).

Although at this stage efficiency is not crucial, once each function above works you *could* pre-compute a look-up table that supports the same operation. Using such a look-up table downloaded from an external source, e.g.,

[http://en.wikipedia.org/wiki/Rijndael\\_S-box](http://en.wikipedia.org/wiki/Rijndael_S-box)

is a useful way to test your implementation: you can simply test whether the output of your function matches the table for every possible input.

### Step #2: round key generation

The next step is to provide functionality that can, given a cipher key, compute the rounds keys used during encryption: AES-128 requires a total of 11, the 0-th of which is  $rk^{(0)} = k$  and then each next,  $(r + 1)$ -th round key  $rk^{(r+1)}$  for  $0 < r < 11$  is produced as a function of the previous,  $r$ -th round key  $rk^{(r)}$ . There are various strategies for (pre-)computing the round keys, and the task below attempts to permit whichever you prefer (i.e., either compute and use them step-by-step when required, or pre-compute them in one step and use them when required).

Using [1, Section 5.2] as a guide, implement

Implement  
(task #2)

1. a function that accepts the  $r$ -th round key as input, and produces the next,  $(r + 1)$ -th round key as output (to do so, you will need to make use of the function from Task 1 to compute the associated round constants), and
2. a function that iterates use of the above to pre-compute *all* 11 round keys given an initial cipher key.

```

1 import struct, Crypto.Cipher.AES as AES
2
3 if ( __name__ == "__main__" ) :
4     k = [ 0x2B, 0x7E, 0x15, 0x16, 0x28, 0xAE, 0xD2, 0xA6, \
5           0xAB, 0xF7, 0x15, 0x88, 0x09, 0xCF, 0x4F, 0x3C ]
6     m = [ 0x32, 0x43, 0xF6, 0xA8, 0x88, 0x5A, 0x30, 0x8D, \
7           0x31, 0x31, 0x98, 0xA2, 0xE0, 0x37, 0x07, 0x34 ]
8     c = [ 0x39, 0x25, 0x84, 0x1D, 0x02, 0xDC, 0x09, 0xFB, \
9           0xDC, 0x11, 0x85, 0x97, 0x19, 0x6A, 0x0B, 0x32 ]
10
11     k = struct.pack( 16 * "B", *k )
12     m = struct.pack( 16 * "B", *m )
13     c = struct.pack( 16 * "B", *c )
14
15     t = AES.new( k ).encrypt( m )
16
17     if( t == c ) :
18         print "AES.Enc( k, m ) == c"
19     else :
20         print "AES.Enc( k, m ) != c"

```

**Figure 1:** Source code to perform AES encryption on the test vector from [1, Appendix B].

Note that [1, Appendix A.1] includes a worked example based on the test vector

$$k = \langle 2B, 7E, 15, 16, 28, AE, D2, A6, AB, F7, 15, 88, 09, CF, 4F, 3C \rangle_{(2^8)}$$

This represents a useful way to test your implementation: it allows you to compare intermediate output produced with a trusted reference, and thereby quickly localise and diagnose any error (versus comparison with the final output alone, which simply shows whether an error occurred or not).

### Step #3: round functions and encryption

Finally, we use the accumulated functionality to implement the encryption operation itself: the most sensible approach is to work step-by-step by initially focusing on the round functions, *then* their composition to form encryption.

Using [1, Section 5.1.1-4] as a guide, implement each of the following round functions:

1. ADD-ROUNDKEY,
2. SUB-BYTES,
3. SHIFT-ROWS, and
4. MIX-COLUMNS.

Implement  
(task #3)

Each function should accept a state matrix as input (with ADD-ROUNDKEY accepting an additional round key matrix that represents the appropriate round key), and transform it into an output state matrix.

Using [1, Section 5.1] as a guide, implement a function to perform AES-128 encryption: your implementation should, roughly,

- accept a 16-byte plaintext and 16-byte cipher key as input, and transforms them into an initial state and round key matrix,
- expand the initial round key matrix into all 11 round key matrices iff. appropriate,
- apply round functions from Task 3 to transform the initial state step-by-step into a final state, and finally
- transforms the final state into a 16-byte ciphertext, which is produced as output.

Implement  
(task #4)

Note that [1, Appendix B and C] include worked examples based on the test vectors

```

k = < 2B, 7E, 15, 16, 28, AE, D2, A6, AB, F7, 15, 88, 09, CF, 4F, 3C >_{2^8}
m = < 32, 43, F6, A8, 88, 5A, 30, 8D, 31, 31, 98, A2, E0, 37, 07, 34 >_{2^8}
c = < 39, 25, 84, 1D, 02, DC, 09, FB, DC, 11, 85, 97, 19, 6A, 0B, 32 >_{2^8}

k = < 00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0A, 0B, 0C, 0D, 0E, 0F >_{2^8}
m = < 00, 11, 22, 33, 44, 55, 66, 77, 88, 99, AA, BB, CC, DD, EE, FF >_{2^8}
c = < 69, C4, E0, D8, 6A, 7B, 04, 30, D8, CD, B7, 80, 70, B4, C5, 5A >_{2^8}

```

This represents a useful way to test your implementation: it allows you to compare intermediate output produced with a trusted reference, and thereby quickly localise and diagnose any error (versus comparison with the final output alone, which simply shows whether an error occurred or not).

An obvious alternative is to compare your implementation with an existing reference implementation, or oracle<sup>1</sup>. The arguable advantage of this approach is that you no longer need any known test vectors: denoting the reference implementation by  $O$ , the idea is to select a random  $k$  and  $m$  then simply check whether

$$\text{AES-128.ENC}(k, m) \stackrel{?}{=} O(k, m).$$

The material cited at the start of the worksheet includes various examples demonstrating how to perform AES encryption in different languages. A Python implementation is shown in Figure 1 since this is arguably the “cleanest” (so illustrates the concepts above most clearly). Beyond this, some language-specific detail follows:

- `encrypt.py` is a Python-based implementation which can be compiled and executed from a BASH shell as follows:

```

bash$ make clean all
bash$ python ./encrypt.py

```

- `encrypt.java` is a Java-based implementation which can be compiled and executed from a BASH shell as follows:

```

bash$ make clean all
bash$ java encrypt

```

- `encrypt.[ch]` is a C-based implementation which can be compiled and executed from a BASH shell as follows:

```

bash$ make clean all
bash$ ./encrypt

```

## What next?

This worksheet, and indeed the associated lecture(s), focused exclusively on AES *encryption*, i.e., computing

$$c = \text{AES-128.ENC}(k, m).$$

Although this simplification might seem very restrictive, when used in CTR mode [2, Section 6.5], e.g.,

$$\begin{aligned} c_i &= m_i \oplus \text{AES-128.ENC}(k, i) \\ m_i &= c_i \oplus \text{AES-128.ENC}(k, i) \end{aligned}$$


Implement  
(task #5)

AES encryption is in fact all we require to both encrypt plaintexts *and* decrypt ciphertexts. In other applications however, explicit decryption can be important. [1, Section 5.3] details how decryption works, essentially describing how each step in encryption can be inverted.

Following the same step-by-step approach as you did for encryption within the tasks above, develop an AES decryption implementation. Although you can use the same approach to debugging via test vectors, note you can also test *both* implementations (effectively using each as an oracle to test the other) by selecting random  $k$  and  $m$  and checking whether

$$m \stackrel{?}{=} \text{AES-128.DEC}(k, \text{AES-128.ENC}(k, m)).$$

<sup>1</sup> [http://en.wikipedia.org/wiki/Oracle\\_\(software\\_testing\)](http://en.wikipedia.org/wiki/Oracle_(software_testing))



Implement  
(task #6)

This worksheet focused on a single, specific implementation strategy. For different platforms however, the trade-offs made may be unattractive: the obvious example is where efficiency is critical, e.g., in applications such as encryption of network traffic and full-disk encryption. With this in mind, [1, Section 6.4] briefly refers to other strategies including use of T-tables also covered in the lecture(s). Recall that the idea is to pack columns of the state and key matrices into 32-bit words, then pre-compute as much of each round function as is possible so their application requires less computation.

Using your implementation from the tasks above as a starting point, try to develop an alternative based on T-tables. Compare the two strategies in terms of their memory footprint and latency: how much more memory does the alternative use, and how much faster can it perform encryption operations as a result?

## References

- [1] *Advanced Encryption Standard (AES)*. National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 197. <http://csrc.nist.gov>. 2001 (see pp. 1–5).
- [2] *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*. National Institute of Standards and Technology (NIST) Special Publication 800-38A. <http://csrc.nist.gov>. 2001 (see p. 4).