Practical Introduction to Hardware Security

# Task 2: Advanced Encryption Standard

Chair of Dependable Nano Computing, Karlsruhe Institute of Technology
Instructors: Dennis Gnad, Jonas Krautter, Mehdi Tahoori

November 21, 2019

## 1 Introduction and Supplementary Information

The Advanced Encryption Standard (AES) was introduced by the US National Institute of Standards and Technology (NIST) in 2000 as a replacement for the formerly used Data Encryption Standard (DES), which was considered insecure due to the short key length and increasing computational power. Originally, the algorithm was named Rijndael after its developers Joan Daemen and Vincent Rijmen, who submitted it to a competition which would determine the most suitable DES successor.

AES is a symmetric block cipher, meaning it operates on data blocks of 128 bit length and uses the same key for encryption and decryption. The encryption function $C = Enc(K, M)$ takes a 128 bit input plaintext together with the key, which can have a length of 128, 192 or 256 bit, and computes a 128 bit output ciphertext. If we want to encrypt longer messages, the data needs to be segmented and padded before submitting it to one of the various encryption modes, such as Electronic Codebook (ECB), Cipherblock Chaining (CBC), Cipher Feedback (CFB) or Output Feedback (OFB).

For this task, we focus on the block encryption only, without any encryption mode. You will implement a 128 bit key length AES block cipher module in hardware, that receives plaintexts through the UART interface and sends the respective ciphertext back afterwards. Therefore, you need a detailed understanding of the algorithm, which will also help you in later tasks on attacking the AES cipher. The most important reference document is the original NIST publication NIST.FIPS.197 [1], which is available online and has very helpful example cipher vectors for debugging in the appendix. You can focus on this document for most implementation needs, but you are also encouraged to look for further documentation of the AES online. Since it is a very well known and widely used algorithm, there is an abundance of documentation and examples available.

### 1.1 Examination of this and following Tasks

In this task we allow teamwork, but would like to limit the amount of people working together on a task to **two** students. If you would like to form a group of two, you must announce it to us **in advance** by e-mail. In that case, you will be examined together. Cooperation between all students of the class is also accepted, but **without sharing code**.

After the task is finished, you will have to hand it in by sending us all your material and appear in the lab. You should then explain us all the steps that you implemented, if possible by actually showing how/that it works. If you have been working in a group, both group members must be present for examination.

The deadline for Task 2 is **Thursday, December 12, 2019**.

### 1.2 Questions / Help

Do it! If you are really stuck and can't find your problem, it is usually better to ask us for help. As in the initial tests, feel free to write mails / come by to get assistance.

## 2 Task Details

We provide the completed outer framework of the AES hardware encryption module as well as a makefile to compile the project. You should not have to edit *top_level.v*, *aes_module.v* or *uart.v* **at all,** but are of course allowed to do so, if your implementation requires it. The outer framework handles the communication with the PC and the basic state machine. It waits until 16 bytes of plaintext data have been received, then activates the encryption and returns the ciphertext through UART after the encryption has finished. Your task is to implement the four main AES operations SubBytes, MixColumns, ShiftRows, AddRoundKey as well as the key expansion (key schedule) part.

We provide some small helper functions, which you may find useful for implementing the AES operations. We provide a simple AES sbox lookup table in *sbox.v*, a helper function for multiplication in $GF(2^8)$ in *xtime.v* as well as the round constants for the key scheduling in *rcon.v*.

The generic interface of each AES operation module is provided but the internal implementation of the submodules is completely up to you. You may use any of the helper functions and can add arbitrary additional modules as well. The makefile automatically includes all Verilog files in the source directory in the compilation. If you wish to change that behaviour, please adapt line 2 of the makefile accordingly. Furthermore, you might not want to implement some operation sequentially, in which case you can simply ignore the *clk* or *rst* signals. The AddRoundKey operation should be implemented in *aes.v* itself (see the TODO in the statemachine there). To summarize: The way you implement the AES operations is up to you, we only provide a helpful skeleton and of course your overall design is restricted by the size of our iCE40-HX8K FPGA.

Please note that the secret AES key is hard-coded into *aes_module.v* as the example key from NIST.FIPS.197. If you wish to evaluate a different key, you must change the file there accordingly.

We provide a python script *check_aes.py* to quickly verify the correct function of your finished AES design.

## 3 Development and Debugging Help

For debugging and development, it can be helpful to jump in the state machine (`always`-block) of *aes.v* to the last state (`DONE`) to get the result of an intermediate value to the PC, without going through the other states.

For example, in the beginning, you implement the `SUB_BYTES` state. From that state you can instead of going to the next state, go to the `DONE` state, and you will be able to check the result on the PC with the *check_aes.py* script, if it matches what you expect.

To get some helpful examples what values to try (with known-good results of the intermediate states), the NIST.FIPS.197 document has a very helpful appendix. **You just have to first understand the 4x4 matrix style of writing the `aes_state` register, which is essential to solve the whole task.**

Regarding potential size limitations: Instantiating a certain module multiple times will usually create that respective hardware structurally (i.e. instantiating the sbox five times, will use FPGA resources for five of them). If you want to perform operations sequentially, you have to schedule it yourself using an `always`-block sensitive to `clk`, and for example by a counting mechanism or state machine.

# References

[1] N. I. of Standards and T. (NIST). (2001) Announcing the advanced encryption standard (AES). [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf