University of Maryland Baltimore County
CMPE 419/691, ENEE 691
Hardware Security
Spring 2023

HW6: Concurrent Error Detection in AES

Due Date: 4/6/2023

Deliverables

- A .zip file containing
  1. Your source VHDL/Verilog files (Please make sure that the Makefile works properly**)**
  2. A ReadMe describing how to run your script

In this lab, you are to implement a parity-based Concurrent Error Detection (CED) scheme for the AES cipher to detect that faults that may have been injected in the output of any module in AES (sub-bytes, shift-rows, mix-columns, add-round-key) of rounds 1 to 9. The faults are not injected in the key-schedule module.

The code you were given is the VHDL implementation of AES (with 128-bit key). It includes a fault injector module that enables you to inject one or two faults in the module and round of your choice. Please note that, in case of injecting two faults, these two faults are injected into the same module and in the same round. The fault injection module has been instantiated in the top module (aes_128.vhd). If you are more comfortable with Verilog, you can use the package provided for Verilog. Note that in the Verilog package, only the files that you need to change are in Verilog and the rest are in VHDL. You can use VCS to simulate the package that includes both VHDL and Verilog files.

Fault Injector:

You were given the VHDL implementation of AES. It includes a module to inject toggling faults (a specific bit is inverted due to fault) in the output of one of the sub-bytes, shift-rows, mix-columns, or add-round-key modules in the round of your choice. This tool can inject up to two faults (either 0, 1, or 2). The tool reads the file called "in.txt". The "in.txt" includes the following information:

The first line in "in.txt" includes 6 numbers as below:

<faulty bit> <round number> <operation> <row offset1> <byte offset1> <bit offset1><row offset2> <byte offset2> <bit offset2>

- **<faulty bit>** can be '0' for disabling fault injection in which case the rest of the numbers in the first line are ignored, or '1' for enabling fault injection.
- **<round number>** can be from 1 to 9, to set the round in which the fault will be injected.
- **<operation>** specifies the module in which the fault will be injected: 0 for sbox, 1 for shiftRow, 2 for mixColumn, and 3 for keyXor. The fault will be injected in the output of the specified step.
- **<row offset1>** can be from **0** to **3** to specify the word (row) in which the fault will be injected.
- **<byte offset1>** can be from **0** to **3** to specify the byte in the specified row in which the fault will be injected.
- **<bit offset1>** can be **from 0** to **7** to specify the bit in the specified byte in which the fault will be injected.

- **<row offset2> <byte offset2> <bit offset2>** show the row, byte and bit location of the second fault. Note that the second fault is injected in the same module and at the same round as fault 1 and only the location is different. If you want to inject only one fault, these three values should be exactly similar to row, byte and bit location of the first fault.

The 2nd and 3rd lines in "in.txt" specify the key and plaintext respectively, and will be fed to the algorithm column-wise.

Example: The first line in Fig.1 shows that two faults are injected at round 1 in the output of the mix-column module. The first one is injected in the 3rd row, 1st byte, 5th bit, and the second fault is injected in the 1st row, 2nd byte, 3rd bit. Fig. 2 shows the locations where these faults are injected in "green".

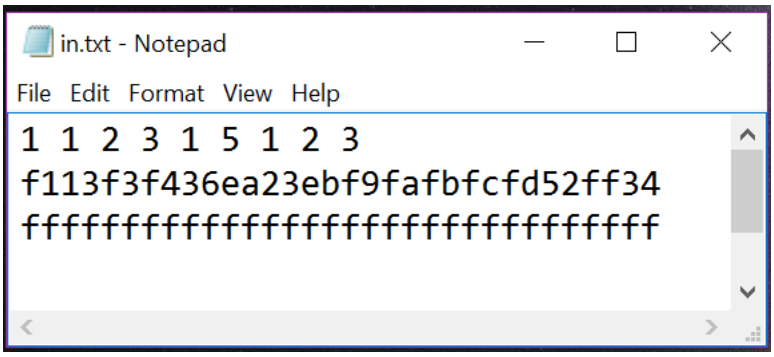

Figure 1: "in.txt"

	0	1	2	3
0	11010011	00110011	01011010	10100101
1	10100101	01100100	11111100	00101000
2	01010000	01010001	10111101	01010011
3	01010001	01010111	01010100	01011100

Figure 2. The location of the injected fault in Example 1 is shown in green

As mentioned, the 2nd and 3rd lines in "in.txt" specify the key and plaintext respectively, and will be fed to the algorithm column-wise. In the example shown in Fig. 1, the key would be:

f1 36 f9 fd
13 ea fa 52
f3 23 fb ff
f4 eb fc 34

Note: As mentioned, when a fault is injected, the selected bit will be toggled.

In this project, you need to implement a parity-based error detection scheme for each of the sub-bytes, shift-rows, mix-columns, add-round-key steps as discussed in class (Please check Lecture 6 slides).

For shift-rows, you need to consider a parity for each row. For mix-columns you need to consider a parity for each column, for s-box, as you know (from slides), you need to modify the s-box module you were given and add 1 bit to each output of each byte. Note that as you change the sbox, you need to change the related component instantiation in AES. For s-box and add-round-key, you need to consider a parity for each byte.

After implementing the parity-based error-detection, you generate a 1-bit signal called "fault_detected", which should remain low while operation is correct, and goes high whenever a fault is detected. You add this signal to the primary output of the AES module (which is instantiated in the testbench)

Note that you should build your parity checkers around the input and output signals of the fault injector as shown in Table 1. As mentioned for sbox, you need to change the module itself.

Step	Input Signal	Output Signal
sbox	data_reg	sbox_out_eff_sig
shiftRow	sbox_out_eff_sig	shiftRow_out_eff_sig
mixColumn	shiftRow_out_eff_sig	mixColumn_out_eff_sig
keyXor	mixColumn_out_eff_sig	keyXor_out_eff_sig

Table 1: Input-Output Signals of all AES Steps

- Based on the assignment instruction, You just need to modify the aes_128.v, sbox.v, and aes_128_tb.v are provided in the assignment folder. In case of using VHDL, aes_128.vhd, sbox.vhd, and aes_128_tb.vhd.

- It is recommended to use Synopsys VCS Compiler to do this assignment. However, if you do not want to work with Synopsys VCS compiler, please do not forget that in each VHDL file, you need to change every instance of "use work…." to "use ieee….." except for the "use work.aes128Pkg.all;"
  For example:
  use work.std_logic_arith.all; ====> use ieee.std_logic_arith.all;

- As Verilog does not support multidimension array as input and output port, either use Snopsys VCS (which does not need any change, and you can use the code as is) or if you use a different compiler/simulator, you need to change the file extensions from .v to .sv.

- As you can see in the testbench, after simulating, it reports the AES output in out.txt file. So, when you modify your design you should print the new 1-bit output (fault_detected) which indicates if the output is faulty or not. So, please print the new output signal in the out.txt file with some space with AES ciphertext.

-

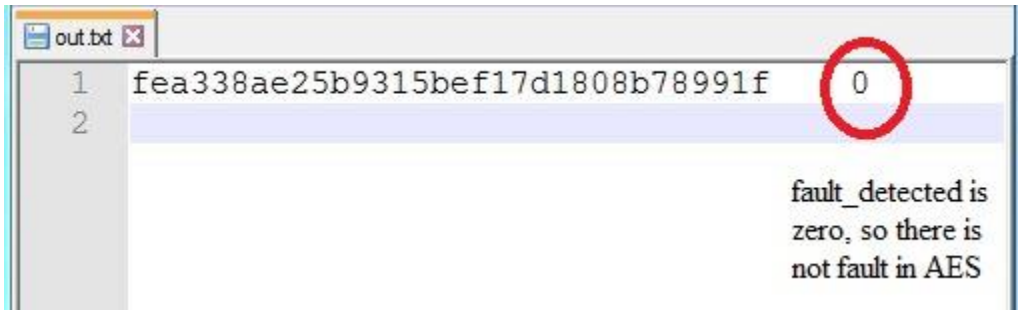

Figure 1: Sample output