Physical Attacks

Assignment 2: Leakage Assessment and Fault Injection

IMPORTANT: do not distribute the files provided for this assignment!

Teacher	Email
Omid Bazangani	`omid.bazangani@ru.nl`

1 Part A: Leakage Assessment on ByteMasked-AES

Goals: After finishing this assignment, you should be able to evaluate a crypto design for possible leaks using the first-order and second-order TVLA tests. You will learn how to conduct a TVLA test and interpret test results.

Grading: This assignment has 6.5 points, obtained by answering the assignment questions. Please check the questions for a breakdown of how the points are awarded.

Deadline: Thursday, 25th of MAY 2023, 23:59 sharp!

Handing in your answers:

- You can hand in your solutions via the assignment module in Brightspace.
- Make sure that your name and student number for all team members are on top of the first page!
- For submitting the answers, please submit a `.pdf` typeset solutions.
- Please write your code in Python and provide a Jupyter notebook you ran before and saved while the results are visible.

Your teacher will grade your assignment digitally in Brightspace, where he/she will leave feedback.

Before you start: The steps required for solving the assignment are the following:

1. This is a group assignment, so find a partner (or otherwise work alone).

2. Install the Python environment. The provided code snippets are written in a `jupyter notebook`. If you are unfamiliar with this concept, you can find out more here: https://realpython.com/jupyter-notebook-introduction/.We have tested the code on `python 3.7`, (you can select this with https://docs.anaconda.com/anaconda/install/), `Jupyter Lab 2.2.6`

3. Download the provided files (data and code).

4. Complete the assignment.

5. Submit the assignment solutions before the deadline, including your **code** and **report**.

1.1 Assignment Statement:

Imagine that you are working for a security company in the design department as part of the blue-team (defense side). Your supervisor asked you to evaluate the most recent product for possible leaks before launching. Your task is to secure the product against possible attacks. (including some that do not exist yet!)
Your goal is to find the root cause of the problem and the possible leaks in the design. The secret to finding these leaks is hidden in your course lecture.
You are correct: the answer is TVLA! This method helps you evaluate your design and find hidden leaks in it. TVLA uses Welch's t-test, which is as simple as comparing the difference between two samples (carefully chosen) to a threshold of 4.5 (which has a story behind it).
The company's developer team claims their design is secure against first-order attacks as they used masking countermeasures[1]. They implemented a ByteMasked-AES algorithm on an ARM Cortex-M3 Microcontroller. We provided the recorded dataset link in the table1.

First-order Analysis. You need to apply the first-order TVLA test (normal TVLA test) on the provided dataset to check if the claim of the developer team is correct!

Second-order Analysis. For a second-order assessment, you must combine the samples in pairs for each trace to make a new trace. The new trace length can be calculated by the following combination formula.

$$^{n}C_k = \frac{n!}{k!(n-k)!} \tag{1}$$

Here n is the number of samples and k is the number of combinations ($k = 2$ in second-order)

[1]Masking is one of the countermeasures against first-order analysis with the goal to break the secret information into different shares and process each share at different times (different CPU clocks).

As the length of the second-order trace would increase dramatically, we ask you to apply the combining method for a small window of samples. We suggest you combine the 50 samples around sample number 10 000 (from 9 975 to 10 025). Combine these samples by the multiplication function to create a new power trace.

$$New\_Trace\_sample_x = sample[i] \times sample[j] \tag{2}$$

$i = 9\,975 \ \ to \ \ 10\,025 \qquad j = (i+1) \ to \ \ 10\,025 \qquad x = 0 \ \ to \ \ 1\,225$

Apply the TVLA test on the new power trace, compare the result with first-order TVLA, and answer the following questions.

Table 1: Files to download for solving Part B.

File name	**Description**
`PA_Assignment_2.pdf`	Description of the assignment you are currently reading.
`TVLA _dataset .npy files`	[DATA] contains fixed and random datasets besides plaintext files (w.r.t ByteMaskedAES algorithm). Download: https://mega.nz/folder/48ZyFKhb#BzSyOLHR7phnIBR1qQl1Lw

1.2 Questions

To complete this assignment answer the following questions:

(1.0p) Q1: Code implementation.

a) Implement the first-order TVLA test.

b) Count the number of leaky points with respect to the threshold value of 4.5.

c) Plot the TVLA graph with the threshold lines (+/- 4.5).
Hint: your graph should look like usual TVLA graphs (e.g. see Fig. 9 and 11 here https://eprint.iacr.org/2019/1445.

(1.5p) Q2: Code implementation for second-order TVLA.

a) Implement the second-order TVLA test.

b) Count the number of leaky points with respect to the threshold value of 4.5.

c) Plot the TVLA graph with the threshold lines (+/- 4.5).
Hint: your graph should look like usual TVLA graphs (e.g. see Fig. 9 and 11 here https://eprint.iacr.org/2019/1445.

(2.0p) Q3: Is it possible to distinguish in the power traces where the first two bytes of the key are being processed? Explain your answer. Hint you could use SNR (Hint: `https://ileanabuhan.github.io/general/2021/05/07/SNR-tutorial.html`)

(1.0p) Q4: Compare the results of first-order and second-order analysis in terms of the number of leaky points and explain the results.

Hint: For a fair comparison, you should compare the number of leaky points in the sample range of 9 975 to 10 025 with the number of leaky points you find in the second-order trace that you made.

(1.0p) Q5: How do you interpret the *t*-test result if the threshold changes to +/-5 instead of +/-4.5?

Hint: explain your answer with the help of α.

2 Part B: Simulation of Fault Injection Attacks

Goals: After completing this assignment, you can analyze the resilience of a code snippet to Fault Injection (FI) attacks. During the assignment, you will use an open-source simulator and learn how to limit the amount of FI vulnerabilities in software implementations.

Grading: This assignment has 3.5 points, obtained by answering the questions. Please check the questions for a breakdown of how the points are awarded.

Deadline: Thursday, 25th of MAY 2023, 23:59 sharp!

Handing in your answers:

- You can hand in your solutions via the assignment module in Brightspace.
- Make sure your name and student number are on the top of the first page!
- For submitting the answers, please submit a `.pdf` typeset solutions.
- For this assignment, you must deliver your code in C/C++ format compatible with the provided software environment.

Before you start: To complete this assignment, download the related software and the provided example. To solve the assignment, follow the below steps:

1. Unzip the FiSim software and run the GUI. Although FiSim is an open-source software (that you can build for your machine), it is only fully supported on the Windows operating system. Therefore, download the released version here: `https://github.com/Riscure/FiSim/releases`.
2. You can learn about FiSim here: `https://github.com/Riscure/FiSim`.
   To learn even more, you can follow a presentation covering fault attacks here:
   - `https://www.youtube.com/watch?v=_ZLJra0trDA`, and
   - `https://www.youtube.com/watch?v=7DYkV4uZqS8`.
3. Download the provided example code here:
   `https://mega.nz/file/h1QCRDAR#8NYZMnIfw09J_fxPbrlDI_vNm_mIxcY8WeQPIdts57E`.
   After downloading and unzipping the example, please copy the files in the "./Content" directory, where the "." represents the FiSim root directory.
4. Do the assignment.
5. Submit your solution, including your code and report.

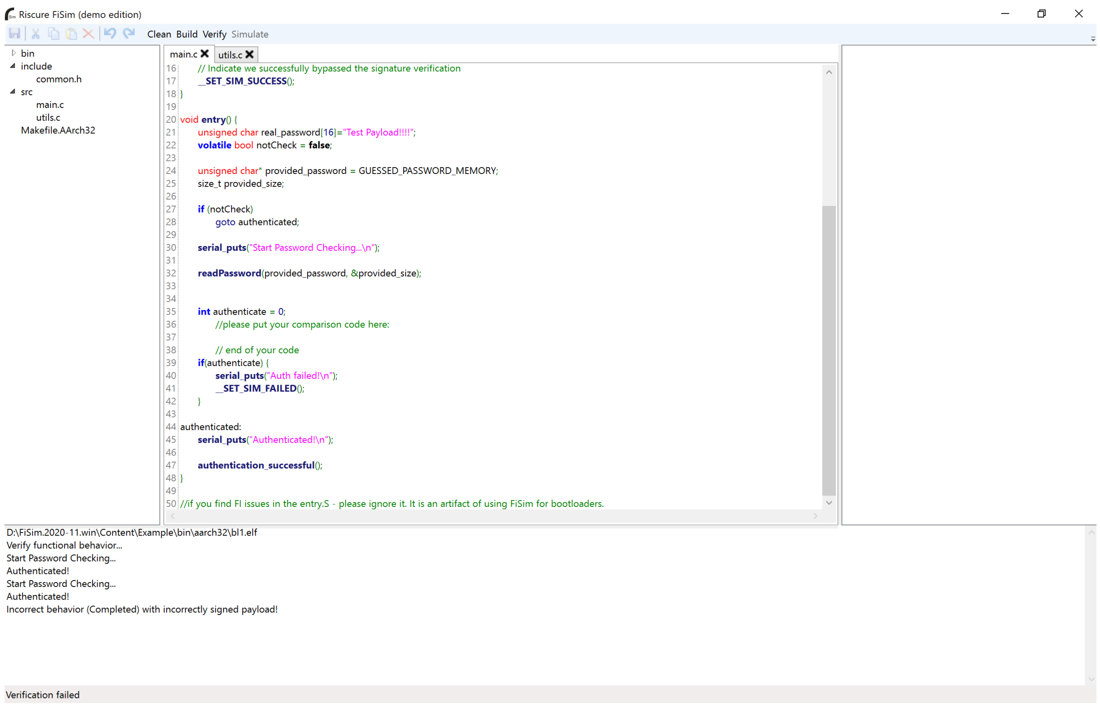
3 Simulation of Fault Injection Attacks

The provided example code is an authentication procedure (password checker) for a secure boot-loader in an ARM microprocessor. After loading the provided example code, do the build and verify steps, respectively. The verification step is aimed at checking the functionality of the code. As the example code is incomplete, you will get an "authenticated" message in the software console, which is an incorrect behavior of the boot-loader; see Fig 1. Your goal is to write code to compare two variables, simulate your code using FiSim, analyze the simulation results for FI vulnerabilities, and try to remove them from your code using simulator feedback.

3.1 Complete the code

The first step of your assignment is to write a code snippet that compares the two variables, `provided_password` and `real_password`. You must assign the value "0" to the `authenticate` variable if they are equal. A section to write your code is provided, as shown in Fig 1.

Note 0: Every time you change the code in the IDE, please save the file (as it's not done automatically), then clean, build and verify your code respectively.



```
    // Indicate we successfully bypassed the signature verification
    __SET_SIM_SUCCESS();
}

void entry() {
    unsigned char real_password[16]="Test Payload!!!!";
    volatile bool notCheck = false;

    unsigned char* provided_password = GUESSED_PASSWORD_MEMORY;
    size_t provided_size;

    if (notCheck)
        goto authenticated;

    serial_puts("Start Password Checking...\n");

    readPassword(provided_password, &provided_size);


    int authenticate = 0;
        //please put your comparison code here:

        // end of your code
    if(authenticate) {
        serial_puts("Auth failed!\n");
        __SET_SIM_FAILED();
    }

authenticated:
    serial_puts("Authenticated!\n");

    authentication_successful();
}

//if you find FI issues in the entry.S - please ignore it. It is an artifact of using FiSim for bootloaders.
```

```
D:\FiSim.2020-11.win\Content\Example\bin\aarch32\bl1.elf
Verify functional behavior...
Start Password Checking...
Authenticated!
Start Password Checking...
Authenticated!
Incorrect behavior (Completed) with incorrectly signed payload!
```

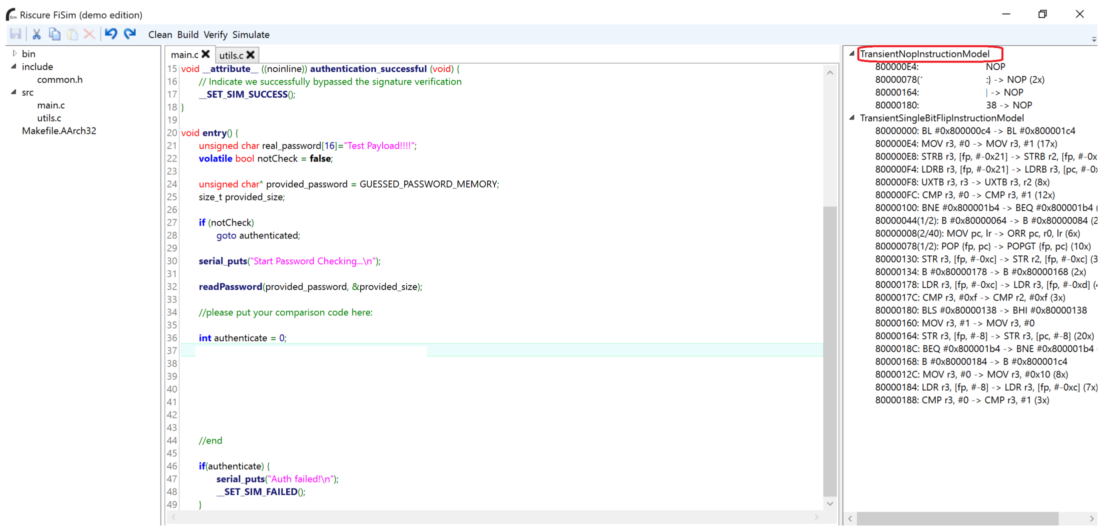
```
Verification failed
```

Figure 1: FiSim Example Code

3.2 Code Simulation

If your code works properly, you will get the “Auth failed!” message in the console as the provided password is not correct, and you can see the “verification finished ” message in the console.
As the next step, please simulate your code and find FI vulnerabilities. When you find vulnerabilities, please explain the root cause for each vulnerability in your code.

Note 1: Depending on how you wrote your code, there might be instructions for which it is hard to determine the cause of a fault (like NOP(2x)). We do not expect you to find the exact reason for the issue, but we encourage you to experiment and find as much as possible (preferably also to avoid glitches).
Note 2: In the simulation, there are two fields for the result. You need to focus on the “TransientNopInstructionModel” part, which is indicated by a red box in Fig 2.
Note 3: The code is compiled with -O0 flag. Please do not change it.



```
void __attribute__ ((noinline)) authentication_successful (void) {
    // Indicate we successfully bypassed the signature verification
    __SET_SIM_SUCCESS();
}

void entry() {
    unsigned char real_password[16]="Test Payload!!!!";
    volatile bool notCheck = false;

    unsigned char* provided_password = GUESSED_PASSWORD_MEMORY;
    size_t provided_size;

    if (notCheck)
        goto authenticated;

    serial_puts("Start Password Checking...\n");

    readPassword(provided_password, &provided_size);

    //please put your comparison code here:

    int authenticate = 0;







    //end

    if(authenticate) {
        serial_puts("Auth failed!\n");
        __SET_SIM_FAILED();
    }
```

```
TransientNopInstructionModel
    800000E4:                   NOP
    80000078(                   :} -> NOP (2x)
    80000164:                   | -> NOP
    80000180:                   38 -> NOP
TransientSingleBitFlipInstructionModel
    80000000: BL #0x800000c4 -> BL #0x800001c4
    800000E4: MOV r3, #0 -> MOV r3, #1 (17x)
    800000E8: STRB r3, [fp, #-0x21] -> STRB r2, [fp, #-0x
    800000F4: LDRB r3, [fp, #-0x21] -> LDRB r3, [pc, #-0:
    800000F8: UXTB r3, r3 -> UXTB r3, r2 (8x)
    800000FC: CMP r3, #0 -> CMP r3, #1 (12x)
    80000100: BNE #0x800001b4 -> BEQ #0x800001b4 (
    80000044(1/2): B #0x80000064 -> B #0x80000084 (2
    80000008(2/40): MOV pc, lr -> ORR pc, r0, lr (6x)
    80000078(1/2): POP {fp, pc} -> POPGT {fp, pc} (10x)
    80000130: STR r3, [fp, #-0xc] -> STR r2, [fp, #-0xc] (3
    80000134: B #0x80000178 -> B #0x80000168 (2x)
    80000178: LDR r3, [fp, #-0xc] -> LDR r3, [fp, #-0xd] (
    8000017C: CMP r3, #0xf -> CMP r2, #0xf (3x)
    80000180: BLS #0x80000138 -> BHI #0x80000138
    80000160: MOV r3, #1 -> MOV r3, #0
    80000164: STR r3, [fp, #-8] -> STR r3, [pc, #-8] (20x)
    8000018C: BEQ #0x800001b4 -> BNE #0x800001b4
    80000168: B #0x80000184 -> B #0x800001c4
    8000012C: MOV r3, #0 -> MOV r3, #0x10 (8x)
    80000184: LDR r3, [fp, #-8] -> LDR r3, [fp, #-0xc] (7x)
    80000188: CMP r3, #0 -> CMP r3, #1 (3x)
```

Figure 2: FiSim Result

3.3 Code Robustness

In this step, please modify your code to make it more robust against FI vulnerabilities (if any). As an example, you might use redundancy.

Note 4: There might be some instructions you cannot modify easily to make the code 100% robust. As mentioned, we do not expect you to analyze these instructions in detail but try to eliminate possible glitches.

3.4 Questions

(0.75p) Q1: Implement the code to compare two variables (`provided_password` and `real_password`). Please provide your code in the report you will submit.

(1.0p) Q2: Simulate your code with FiSim and (try to) explain the root cause for the vulnerabilities found by the software for the FI attack.

(1.0p) Q3: Modify your code to make it more robust by removing FI vulnerabilities. Please provide your code for **modified** implementation in your report and explain it.

(0.75p) Q4: What can you say about faults listed in"TransientSingleBitFlipInstructionModel"? please explain as many as you can. You do not need to eliminate them.

Note 5: For the code submission, please submit two files containing the initial and the modified version of your code in addition to the report.

Some small clarifications:

1. To harden the code for fault-injection resistance, it is allowed to modify the code outside the original comments, namely, outside `//please put your comparison code here: ... //end`.
   The robust code must be functionally equivalent to your code containing the password check.

2. We recommend not using any external libraries to implement the password comparison. This would make it hard to determine where faults are really happening. Therefore, it suggests writing your own code for password comparison.

3. Using "Verify" is an excellent first step to check whether your new code is functionally correct.