**Lab Manual**

***for***

**Security Property Development and**

**Fault-Injection Simulation**

**Objective**

The objective of this lab module is to understand a logical-level modeling approach of fault-injection attacks and practice a gate-level fault-injection simulation methodology. Participants will learn the basic categories of fault-injection attacks, their logical impact, how to model their behavior at front-end design stages like gate-level, how to properly simulate their propagations, and how to quantitatively analyze the fault-injection’s threat level by defining a set of security properties and fault-injection lists.

**Equipment needed**

1. Lab database file *lab\_db.zip*
2. Linux server with licensed access to *Synopsys VCS* functional simulator and *Synopsys Z01X* fault simulator

**1. Background**

**1.1 Fault-Injection Attacks**

Fault-injection techniques can maliciously alter the correct functionality of a computing device. It has 3 categories: 1) *non-invasive attacks*, which are inexpensive and more frequent, clock or voltage glitching attacks and electromagnetic (EM) attacks can be performed; 2) *semi-invasive attacks* require some supportive operations on the chip such as depackaging and thinning without harming the chip’s functionality, it utilizes optical fault injection techniques, including flashlight, focused light beam, lasers, etc. ; 3) *invasive attacks* include the powerful and most expensive approaches such as using Focused Ion Beam (FIB) for circuitry editing. In this lab, we discuss the most common fault-injection techniques are below:

**Clock Glitching:** One very low-cost technique to inject faults is to tamper with the clock signal to cause either setup or hold time violations. For example, the length of a clock cycle can be shortened by prematurely toggling the clock signal. In normal operation, the clock cycle should be longer than the maximum path delay of the combinational logic. However, when a clock glitch occurs, it results in a timing violation and the metastability of sequential components, where a fault might be injected and propagate through the circuit.

**Voltage Glitching:** Another fault-injection technique involves tampering with the power supply of a device. For instance, one can run the chip with a depleted power supply, causing high-threshold voltage transistors to remain non-functional. As a result, transient faults are injected into the device. Alternatively, power spikes can be leveraged to inject faults, potentially causing a processor to skip an instruction or a crypto engine to skip a round of encryption/decryption.

**Electromagnetic (EM) Fault Injection:** An external electromagnetic field can also be exploited to inject faults. This can cause a chip to malfunction or flip memory cell(s). Eddy currents on the chip surface can be induced by the EM field, resulting in single-bit faults, which can facilitate the propagation of secret data (e.g., keys) to observable nodes.

**Laser and Optical-based Fault Injection:** Laser-based fault injection attacks pose a severe threat to the security of a device due to their high attack precision from the backside of ICs without causing permanent damage. Laser fault injection techniques utilize the photoelectric effect, which generates electron-hole pairs by passing a laser beam through the silicon substrate. The generated electron-hole pairs create a current pulse through the transistor, further creating a voltage pulse (potentially, a transient fault) that propagates in the circuit. A strong and precisely focused light beam can be exploited to induce alterations in one or more logic gates.

Table 1 summarizes the characteristics of different fault injection techniques.

*Table 1. Summary of fault-injection techniques.*

|  |  |  |  |
| --- | --- | --- | --- |
| Technique | Accuracy | Cost | Damage to device |
| Voltage Glitching | low | low | no |
| Clock Glitching | low | low | no |
| EM | low | low | possibly |
| Light Beam | moderate | moderate | possibly |
| Laser Beam | high | high | possibly |

**1.2 Advanced Encryption Standard (AES)**

AES is a well-adopted encryption standard, implemented in both software and hardware IPs. AES has a fixed block (plaintext) size of 128 bits, and 3 variations of key size: 128, 192, or 256 bits. In this lab, we focus on security property definitions and fault-injection simulations on AES with a 128-bit key.

A screenshot of a computer

AI-generated content may be incorrect.As demonstrated on the right, AES-128 starts with a ‘round 0’ encryption: an XOR between plaintext and 128-bit original key to get an initially encrypted text. This text will then go through 10 rounds of encryption, each round consists of 3 operations (SubBytes, ShiftRows, MixColumns) in order, except for round 10, which only has 2 operations of SubBytes and ShiftRows.

Each encryption round ends with an XOR with the corresponding round key. These 10 round keys are derived from a key-expansion module, which basically utilizes each round’s previous round key and some constants to calculate a new sub-key. Therefore, the ‘round 0’ key (which is the original secret key) is the source of all following round keys.

The plaintext’s transformed forms at every ending points of round 0 to round 9, are called *intermediate states*, and the text of round 10’s ending point is called *ciphertext*.

**1.3 Attacks on AES**

A diagram of a security system

AI-generated content may be incorrect.Mathematically, breaking AES is not feasible due to the 128-bit key space and well-designed encryption algorithm. However, unprotected AES implementations on hardware are subject to key leakage due to fault-injection attacks, side-channel attacks, etc. In this lab, we focus the following fault-injection attack scenarios on AES:

**Attack scenario:** In the normal operations of AES implementation 1, a ‘ld’ signal raise indicates a successful load of plaintext and secret key. After 10 rounds of encryptions, a ‘done’ signal should be raised by the hardware to indicate the completion. However, a fault-injection attack such as clock/voltage glitching or laser injections could flip the logic values stored in the circuits’ sequential components. The register of ‘done’ signal, if being flipped, could be raised before the completion of 10 rounds encryption. In the worst case, if ‘done’ signal is raised before round 1 is completed, which marks the intermediate state at this stage is ready to be read, the attacker could simply XOR the intermediate state with plaintext to derive the secret key.

2. Experimental Setup

**Check your software installation**

If you are on a UF ECE server, do:

* source /apps/settings

Use the following command to confirm if the necessary tools are installed at any path. If you are not on UF ECE server, make sure you get the pointer to these tools’ installations. Contact your server administrator for tool installation issues.

* which zoix
* which fmsh
* which vcs

**Check your software license**

Synopsys license server on UF ECE is 27020@licsvr.ece.ufl.edu, if you are not on UF ECE server, use command *lmstat* to find out your Synopsys license server. Contact your server administrator for licensing issues.

Use the following command to make sure you have valid license to access the tools, and there are available tokens remaining.

* lmstat -c <license server name> -S | grep -i <tool name: zoix | fmsh | vcs>

**Prepare your work folder**

Use the following command to unzip the lab database on your linux server and make it your work folder.

* unzip lab\_db.zip
* cd ./lab\_db

**3. Lab 1: Functional verification and security property definition**

Read the background section to understand the basics of AES encryption algorithm, as well as the attack scenario introduced in Section 1.3.

Although fault-injection attacks possess the potential to alter the logical values within the circuit (i.e., creating a fault), note that not all *faults* could become a *threat* to the hardware security. E.g., a byte fault injected at the end point of the 10th round AES encryption could only result in an incorrect ciphertext without anything harmful to security.

Therefore, it is important to define the *margin* of security, which means that before we do any simulations or estimations on how much a fault-injection incident could badly impact the circuit, we should first define what exactly scenarios cannot be permitted (i.e., threatening the security).

**Functional Verification on AES**

We need to perform functional verification first to understand the normal operations of AES. This is necessary for later steps in defining security properties. Under the path lab\_db/VCS/, do:

* source /apps/settings # Skip this line if not using UF ECE server
* export SNPS\_SIM\_DEFAULT\_GUI=dve # Use dve GUI for the simulation
* vcs -kdb tb\_top.v ../designs/done\_fanin.v -v ../designs/saed32nm.v -o tb\_top.vvp +v2k -debug\_acc+all -debug\_region+cell+encrypt -full64

For the 3rd command using VCS, we are doing a gate-level functional verification for an AES submodule with ‘done’ signal together with the Synopsys 32 nm generic library (saed32nm).

A black text with red letters

AI-generated content may be incorrect.In this lab, since our goal is not to learn writing the testbench for designs, we provide you with a testbench (*tb\_top.v*) for testing this AES. It basically generates random plaintext for the AES to encrypt and finish after a certain period of time. There is an *initial* block in this testbench dumping the stimulus waveform into an *input.vcd* file, which will be automatically generated after the simulation is done, and this *.vcd* file will be used as the stimulus for further fault simulations.

All the commands above are summarized in the *command* script, so alternatively, you could simply do the following for functional simulation:

* ./command

**Expected outcome:** a *tb\_top.vvp* executable file should be generated. Do the following to start dve for simulation:

* ./tb\_top.vvp -gui

Dve window opened with simulation settings loaded:

A computer screen shot of a computer

AI-generated content may be incorrect.

Note that at this point, the simulation is not started yet.

We first choose the signals we want to observe. As demonstrated below, we right-click on the *test* module on the left (the *hierarchy* column) and choose *Add To Waves -> New Wave View*, in this way we added all the signals from testbench into a waveform for watch.

A screenshot of a computer

AI-generated content may be incorrect.

Optionally, one could click on the small ‘+’ next to the module name *test* on the left (the *hierarchy* column) to open a list of lower level modules, and choose any interested signals on the *variable* column to add to waveforms for watching.

Click the downward arrow in the popped-out waveform window to run the simulation.

A screenshot of a computer

AI-generated content may be incorrect.

Press ‘F’ to Zoom to the full waveforms.

A screenshot of a computer

AI-generated content may be incorrect.

Hold and drag on the simulation timeline using left mouse button to Zoom in any portions of the waveform, press ‘F’ to go back full.

Click and hold here

A screenshot of a computer

AI-generated content may be incorrect.

Then drag to here

Based on your observations from the waveform, and the information from Sections 1.2 and 1.3, **answer the following questions:**

* Q1: A full AES-128 encryption completes when a ciphertext is ready. Then how many encryptions did this testbench test?
* Q2: How many clock cycles did one encryption take?
* Q3: If the ‘done’ signal is forced to be raised one cycle earlier than its current raising cycle. Do you think this AES encryption is still secure? Why?
* Q4: Under what circumstances do you think the AES is not secure? Why? (This is an open question)

**Security Property Definition**

A *Security Property* (SP) can be stated at a very high level, even in natural language, to define a security margin. For example:

* AES key cannot be leaked.
* Software instructions executing on processors should not be tampered.

However, these definitions could result in ambiguous explanations and infeasible protection plan against hardware attacks. Regarding the two SPs above, we will naturally have the following questions:

1. How could a hardware attacker leak the AES key?
2. How could they tamper the instructions?
3. To protect the AES key from being leaked, should we protect the whole AES module?
4. Should we duplicate all the instructions on processors to avoid them suffering from potential tampering?

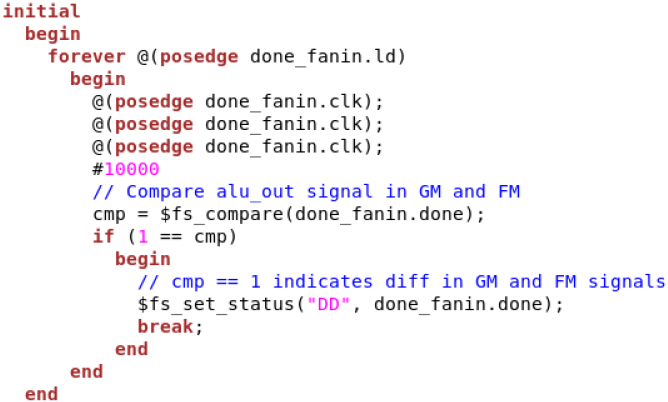
Questions 1 and 2 are open-ended, as hardware attackers can employ various techniques to leak the AES key or tamper with software instructions. Each approach may target different signals, utilize distinct experimental setups, and exploit different attack mechanisms. For Question 3, existing hardware attack countermeasures often rely on duplicating the entire encryption module, which, while effective, introduces significant overhead. Similarly, for Question 4, duplicating all instructions to counteract hardware attacks is impractical due to resource constraints.

Therefore, in security verifications, security properties should be explicitly defined to reflect the attack scenario and attack timing. In this lab, we define our security properties in the form of *strobe* file, which is used by Synopsys Z01X fault simulations.

Section 1.3 described a specific scenario that could lead to AES key leaking. We first make the following assertion in natural language to reflect this security property:

* **SP 1:** The ‘done’ signal indicating the completion of AES encryption should not be raised during the 1st round of encryption.

We then define this SP 1 in the file *strobe.sv* in SystemVerilog. *This strobe.sv file will be checked every time when the Z01X fault simulator runs a simulation with one fault injected.*



**4**

**3**

**2**

**1**

Assuming the fault simulator has simulated a waveform with a fault injected somewhere, how does it determine whether this faulty waveform could threaten the security? Let’s break this file into 4 sections:

* First, it will start checking the following statements only when a rise of ‘ld’ signal is detected.
* Secondly, the tool will wait for 3 cycles from ‘ld’ rise. This is the endpoint of the 1st AES round based on previous observations from functional verification steps. It will wait for a half more cycle (#10000 because the clock period is 20000 time unit, see *tb\_top.v*), to ensure the following checking is happening *after any fault-injections intended for the same cycle is finished.*
* Thirdly, at this specific time (3.5 cycles after ‘ld’ is raised), the *$fs\_compare* function will compare between the ‘done\_fanin.done’ signal’s value in fault-free simulation (good machine, i.e., GM) and the one in faulty simulation (fault machine, i.e., FM).
* The return value of *$fs\_compare* determines whether a security property violation is found. *A return value of 1 means GM and FM mismatches (a violation) due to the fault-injection in this simulation, we then mark this fault as ‘DD’ (Dropped Detected). A return value of 0 means ‘ND’ (Not detected), which does not need to state in the strobe.sv since it’s set by the tool.*

Based on your observations of the AES waveforms, **answer the following question:**

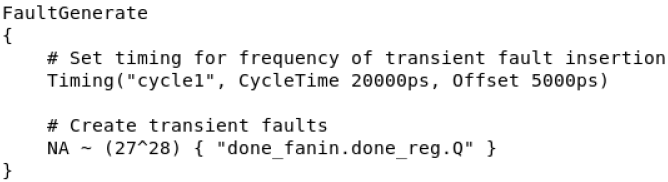
* Q5: Is SP1 rigorous enough to define the attack scenario in Section 1.3? Why? If not, could you write another security property file to define this scenario?

**4. Lab 2: Fault list definition and fault simulation**

The Synopsys Z01X tool picks one fault, inject it into the simulation, and run security property check based on this faulty simulation and the good simulation. This flow will be performed iteratively trying every fault in the fault list, thus, each run is an independent run. After sweeping the whole fault list, those faults causing security property violations are marked differently from others.

**Fault List Definition**

From the SP 1 definition in *strobe.sv* above, we already know that the ‘done’ signal is critical (because it’s being checked) when it’s at the timing when 3 cycles after ‘ld’ is raised. Therefore, if an attacker is able to precisely inject a fault on ‘done’ signal’s flip-flop at this specific timing, it should definitely violate the security property. This kind of “perfect fault” should be first considered in our fault list pending test. In our case, this specific timing is cycle 27, we have the following fault list, with only one fault.



**5**

**4**

**3**

**2**

**1**

Let’s break down this file into 5 sections:

* Section 1 states the clock *period* and *offset*.
  + The clock period should align with the stimulus for the previous functional verification. Check *tb\_top.v*, find a line “*always #10 clk = ~clk*;”, since the timescale is 1 ns according to the beginning of the testbench “*`timescale 1ns / 1ps*”, we could derive the clock period is 20 ns, i.e., 20000 ps. The finial stimulus for Z01X fault simulation is *input.vcd*, but it’s auto-dumped due to the first initial block in *tb\_top.v*, so the timescale and clock period will follow *tb\_top.v*.
  + The *offset* in the fault list file is to add a slight delay in case the fault value is not caught by your target when running a delay-file-involving gate-level simulation. We set it to ¼ of clock period because we intend to ensure this “perfect fault” can be detected by security property checking, which waited for a half more period to start (see *strobe.sv*).
* Section 2: *NA* stands for *Not Attempted*, it marks the status of this fault, since this is a fault list file that includes all the fault pending to be tested. The final report file has the same format, in that file this NA will be substituted by simulated labels such as DD or ND.
* Section 3: “~” stands for transient faults. The Z01X tool also supports other types such as stuck-at-fault, which appears during manufacturing. Transient faults are the only type here we consider in fault-injection simulations.
* Section 4 means this fault is injected at cycle 27, and disappear when simulation reaches cycle 28. This setting matches the nature of fault-injection behavior. This section could include more than one cycle, e.g., (27^28, 28^29, 29^30) will be treated as 3 *independent faults* at 3 different cycles, each of them disappears in the next cycle.
* Section 5 states the fault-injection target, where the value will be flipped.
  + Since our experiment is at gate-level, a port name should be given in the form of: <module name>.<cell name>.<port name>. In RTL simulations, simply substitute this with RTL variable names in the form of <module name>.<signal name>
  + Multiple signals can be involved in one fault by connecting them with “+”. The format is: “<module name>.<cell name>.<port name>” + “<module name>.<cell name>.<port name>”. Note that no matter how many signals are involved by this fault, *it is still one fault.* Therefore, they will be flipped simultaneously *in one signal run.*

This fault list format is *.sff* (standard fault format), check *user.sff* in *lab\_db.zip*. The purpose of this section is to get a better understanding of Z01X tool’s behavior: *Each simulation of Z01X runs one fault, and check strobe.sv for any security property violation, then label this one fault.*

Based on the formatting information about .sff, **answer the following questions:**

Q6: How many faults this .sff line in contains?

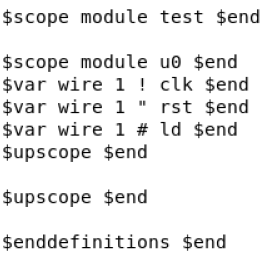
NA ~ (25^26) { "done\_fanin.done\_reg.Q" + "done\_fanin.dcnt\_reg\_0\_.Q" }

Q7: How many faults this .sff line contains?

NA ~ (22^23, 23^24, 24^25, 26^27, 27^28) { "done\_fanin.done\_reg.Q" + "done\_fanin.dcnt\_reg\_2\_.Q" + "done\_fanin.dcnt\_reg\_1\_.Q" }

**Fault Simulation**

**Step 1:** Prepare the stimulus *input.vcd*. In previous functional simulation using VCS, the testbench dumped a *input.vcd* file. Find this section in that file’s beginning:



This hierarchy is from the testbench: “test” (testbench top module name) -> “u0” (the name of instantiated design module). However, for Z01X fault simulation, we will need this vcd file to directly serve as stimulus of our design file. Since the design file’s top module name is “done\_fanin” instead of “u0”, and there is no “test” module, so under work path we do:

* cp ./VCS/input.vcd ./
* gedit ./input.vcd

We delete the first and last line of this section, and change the lower module name from “u0” to our design’s top module name “done\_fanin”:

**Delete ->**

A white background with black text

AI-generated content may be incorrect. A black text on a white background

AI-generated content may be incorrect.

**Delete ->**

Save and exit the file, our stimulus for fault simulation is ready.

**Step 2:** File consistency check. Next, we check if the design top module name “done\_fanin” from *input.vcd* is also the top module name when appearing in every design signal within:

* Security property file (*strobe.sv*): check the signal in your *$fs\_compare* input parameter.
* Fault list file (*user.sff*): check every fault’s target signal name.

Also check if the timescale and clock period are consistent across tb\_top.v, user.sff, and strobe.sv.

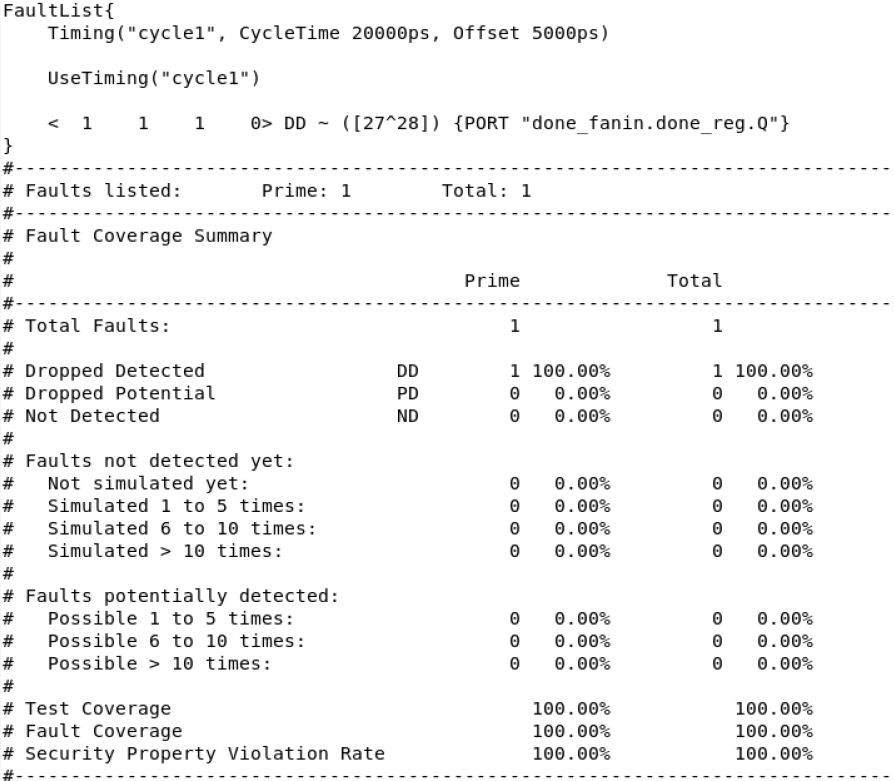
**Step 3:** File path check.

* Make sure you have *./clean* file.
* Make sure design files and saed32nm.v are under ./designs, and this path is correctly reflected in ./run.f
* Make sure stimulus file and fault list file’s names are correctly reflected in ./user.fmsh file.
* Lastly, open ./run file, make sure run.f and user.fmsh are correctly taken by commands *$ZOIXHOME/bin/zoix* and *$ZOIXHOME/bin/fmsh.* If you are not on UF ECE server,substitute *$ZOIXHOME* with your Synopsys Z01X installation path.

**Step 4:** Run the fault simulation

* chmod +x ./run
* ./run

A fault coverage report *user\_coverage.sff* should be generated. It starts with a run info section, following is the fault info of every fault in the fault list, this section could become very long if we have a long fault list or even just a long fault-injection window, e.g, faults happening within clock cycles (1^2, 2^3, 3^4, …… 27^28), the tool will separate it and print each fault in a single line in the final report. Scroll or jump to the end to see the coverage report:



**Discussions:** Note that unlike ATPG tasks, the intention of this fault simulation is not necessarily to achieve better coverage, since a high coverage here only means that we designed our fault list efficiently, a combination of very low coverage % and large number of total faults only indicates that we might waste some simulation time to check on those safe modules and signals, it might be better to narrow the target cells and prepare a more focused fault list. *But the high or low coverage is never a critical issue.*

The main intention is to use this fault simulation to identify all signals that are violating the security properties. Security threats are first categorized and defined by specific scenarios, then for each scenario (security property), we could quantitatively measure their vulnerability level under a specific set of fault-injection inputs (fault list) based on the coverage % report, and enable the possibility of effective and low-cost protections on the entire design by locally protecting the critical signals.

**5. Lab 3: (Optional) build a longer fault list**

Z01X supports using wildcards in *strobe.sv* or *fault list* file (.sff). We could exploit these features to build a longer fault list and run a more comprehensive fault simulation. The following is an example fault list that generates 81 faults, covering all registers and a wide fault-injection window from 1st cycle to 27th cycle. A new section “PORT” is added before the target name to specify the target’s type.

NA ~ (1^2, 2^3, 3^4, 4^5, 5^6, 6^7, 7^8, 8^9, 9^10, 10^11, 11^12, 12^13, 13^14, 14^15, 15^16, 16^17, 17^18, 18^19, 19^20, 20^21, 21^22, 22^23, 23^24, 24^25, 25^26, 26^27, 27^28) { PORT "done\_fanin.\*.Q" }

By observing this fault list and the previous information, **answer the following questions:**

Q8: Do we need to cover a longer fault-injection window to include cycles > 27? Why?

Q9: Are faults on combinational cells included in this fault list? If not, can you develop one?

Q10: Based on the background knowledge in Section 1.1, can you briefly describe under what fault-injection scenario we should include combinational cells into fault lists?

**Reference**

1. H. Wang, H. Li, F. Rahman, M. M. Tehranipoor and F. Farahmandi, "SoFI: Security Property-Driven Vulnerability Assessments of ICs Against Fault-Injection Attacks," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 41, no. 3, pp. 452-465, March 2022, doi: 10.1109/TCAD.2021.3063998.
2. Z01X, Synopsys, CA, USA, Mar. 2025, [online] Available: https://www.synopsys.com/verification/simulation/z01x-functional-safety.html
3. S. Bhunia and M. Tehranipoor, Hardware Security—A Hands on Learning Approach, Cambridge, MA, USA:Morgan Kaufmann, 2019.
4. AES\_core, OpenCores, Mar. 2025, [online] Available: https://opencores.org/projects/aes\_core
5. A. Barenghi, L. Breveglieri, I. Koren and D. Naccache, "Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures," in Proceedings of the IEEE, vol. 100, no. 11, pp. 3056-3076, Nov. 2012, doi: 10.1109/JPROC.2012.2188769.