Practical Lab Manuals

WP1.1.1

“Comprehensive Embedded Software Security Evaluation Against Fault Injection Attacks”

By Zahra KAZEMI

Co-funded by the Erasmus+ Programme of the European Union

Table of Contents

Introduction

1. Hardware Attacks

In general, the attacks against IoT embedded devices can be classified into three main categories, including **1) Network**, **2) Software**, and **3) Hardware Attacks**. In practice, an attack can employ any or all of these approaches. In principle, the Network-based Attacks could be applied remotely at any point of the interconnected IoTs. Various studies show IoTs are susceptible to Network Attacks such as Denial of Service and Spoofing. The second class of security attacks against IoTs is applied at the software level. They can be applied at various software abstraction layers, such as high levels and low levels. For instance, some High-level Software Attacks are brute force attacks that target an application that consists of a pair of input/output to get authenticated or to reveal the information. Other examples aim to inject malware or manipulate the machine-level code at lower levels and hijack the application's execution flow. Besides software attacks, numerous security threats exist against the user-accessible targets named Hardware Attacks. Hardware attacks become critical when the attacker can have direct physical access to measure the device operating parameters or can tamper with the external inputs of the targeted embedded device.

2. Fault Injection Attacks

Fault attacks are the noticeable type of physical attacks, in which the expected and secure behavior of the targeted devices is liable to be jeopardized. Fault Injection Attacks have been designed and introduced in various methods, such as 1) By manipulating the inputs of the device (such as clock or voltage); 2) By stressing the target by changing its surrounding conditions (such as raising the temperature); 3) By emitting energy rays (such as electromagnetic or laser). They can either modify the process of software execution or the stored values inside the memory locations.

3. Clock Glitching Attack

Clock-based fault injection is a low-cost attack that can be applied by the attacker to devices supplied with an external clock. If the target uses an internal clock signal, this method is often not applicable.

In the clock glitching method, the attacker generates glitches in the clock signal. The induced glitches produce extra edges in the clock signal, resulting in an erroneous output as the timing inequality has been violated. Figure.1 shows a typical clock signal in which a glitch is induced. In this figure, T represents the normal clock period, and T_{glitch} is the width of the glitch signal. As it can be seen, an extra edge appears in the clock signal. Another important parameter is T_{min}, which is equal to the reciprocal of maximum frequency. In order to have erroneous behavior, T_{glitch} should be less than $T_{min.}$

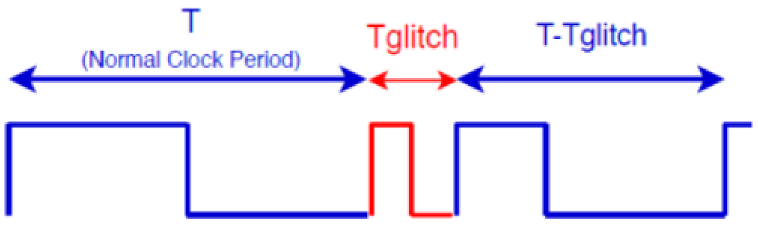


Figure 1. Violating Critical Path Delay by Insertion of Additional Positive Clock Edge

Multiple clock glitch parameters (Figure.2) must be tuned, such as:

- **Glitch Delay:** This parameter shows where to insert the glitch after the positive edge of a clock cycle.
- **Glitch Width:** This parameter describes the width from the point indicated by Glitch Delay to the right.
- **Glitch Temporal Location:** This variable shows the clock cycle (i.e., number of cycles) to insert the glitch after the trigger signal's positive edge.

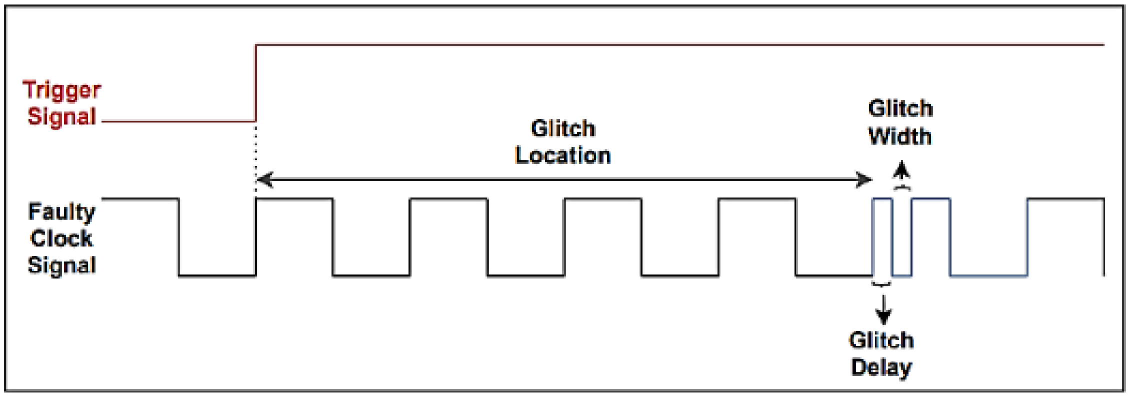


Figure 2. Clock Glitch Parameters

4. Application Level Analysis of Fault Effects

Discovering the impacts of physical FIA is not always straightforward. So, one needs to understand the FIA effects and their propagation through different levels. Figure.2 shows an example of the propagation of an injected fault through layers of an embedded system along with its effects on the target.

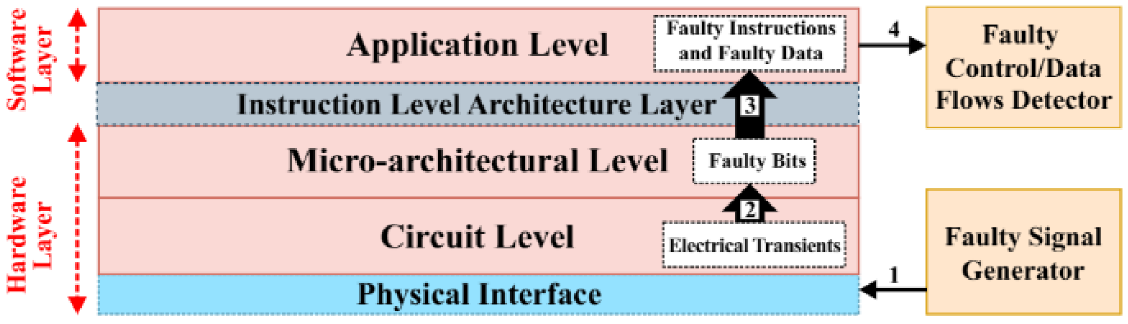


Figure 3. Fault propagation through different layers

These effects can be classified into different categories:

- **Faults at Circuit Level:** The physical stress on any target interface leads to transient electrical faults like transient voltage glitches or current spikes at the circuit level resulting in gate faulty behavior.
- **Faults at Micro-Architectural Level:** Transient electrical faults might be captured by the latches and flip-flops in the system's data or control paths, resulting in erroneous micro-architectural states or data.
- **Faults at Software/Application Level: Faulty** values captured by different micro-architectural blocks would cause errors in the control or data flow of the running software. In other words, a fault at the micro-architectural level manifests itself as a deviation in the correct instruction flow or as a faulty operand or opcode at the software level. Note that the faults at the software level can be exploited in different

In general, the exploited vulnerabilities at the application level can be modeled as **1) Control-Flow Corruptions (CF-Corrupt**) and/or **2) Data-Flow Corruptions (DF-Corrupt)** at the application level. The CF-Corrupt can occur by disrupting the intended order of instructions, branches, or statements of the embedded software. Accordingly, several works have shown that even non-invasive FIAs such as clock/voltage glitching attacks can lead to CF-Corrupt by skipping or repeating one instruction or by replacing that with another instruction. Other CF-Corrupt instances happen when the evaluation step of a conditional branch has been skipped, and the incorrect branch is taken. FIAs can alter the conditional branch instructions, which are used to implement loops and change conditions in security checks of embedded. The software-based countermeasures have significant performance overhead, and they cannot guarantee complete code integrity against fault injection attacks. However, in many non-critical cases, they can provide a good trade-off between hardware cost and security.

5. The lab objectives

In this course, our objectives are:

1. To install and investigate an experimental evaluation platform (Chipwhisperer)
2. To understand the software-level evaluation process of different C-functions and patterns
3. To use chipwhisperer and high-level evaluation approaches to identify the security vulnerabilities of an IoT application (Sec-Pump as an example)

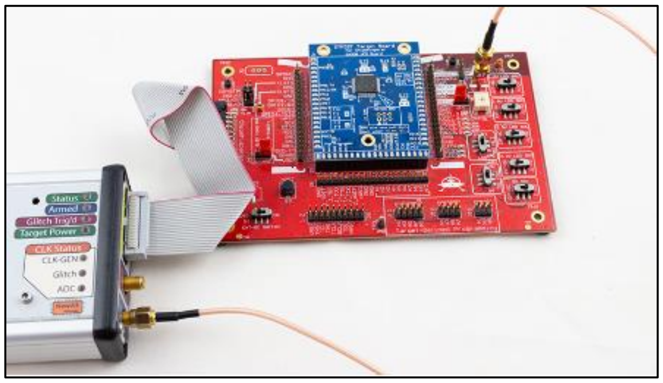
Session 1: Setup Preparation

6. An Introduction to Chipwhisperer Evaluation Platform

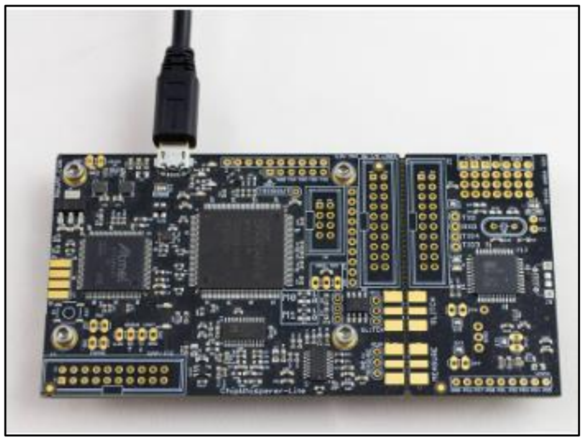
ChipWhisperer is an open-source toolchain that makes learning about fault injection and side-channel attacks easy and affordable. In the following, we are going to focus on the clock glitching part to evaluate the security of our embedded software.

There are different available types of Chipwshiperer boards in the market; For instance:

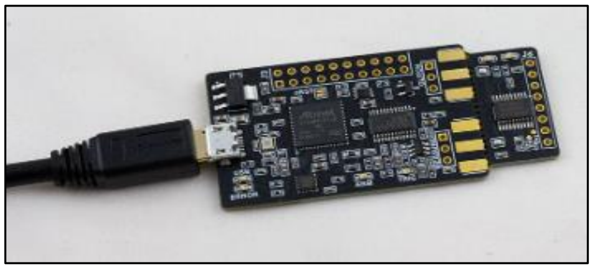
1) CW1200 ChipWhisperer-Pro (CWPro)



2) CW1173 ChipWhisperer-Lite (CWLite)



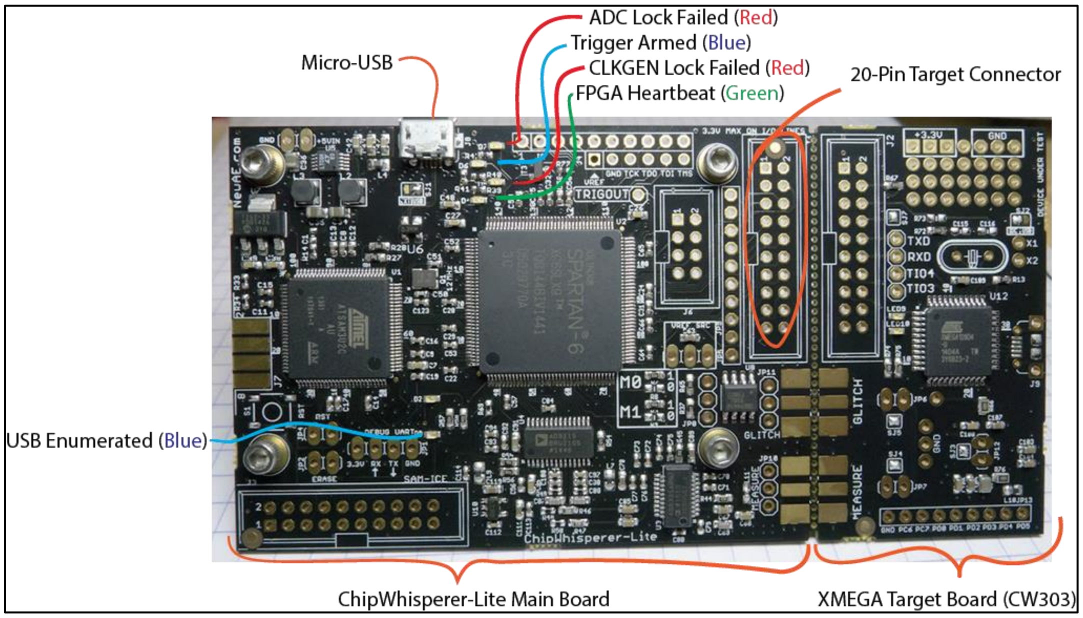
3) CW1101 ChipWhisperer-Nano (CWNano)



7. An Introduction to ChipWhisperer-Lite

In this lab, we are going to use CW1173 ChipWhisperer-Lite (CWLite) from NewAE Technology since it serves as a good middle ground between the full feature-set of the ChipWhisperer-Pro, and the affordability of the ChipWhisperer-Nano.

The ChipWhisperer-Lite typically comes with two main parts: a multi-purpose capture instrument, and a target board. The target board is a standard microcontroller (**XMEGA** or **ARM**) which you can implement algorithms onto.



The ChipWhisperer-Lite, as the name suggests, features the ChipWhisperer-Lite capture hardware. Its datasheet can be found on Mouser: https://www.mouser.ca/datasheet/2/894/NAE-CW1173_datasheet-1859842.pdf

Product Highlights:

- Synchronous (capture board and target board both use the same clock) capture and glitch architecture, offering vastly improved performance over a typical asynchronous oscilloscope setup
- 10-bit 105MS/s ADC for capturing power traces
- Can be clocked at both the same clock speed as the target and 4 times faster
- +55dB adjustable low noise gain, allowing the Lite to easily measure small signals
- Clock and voltage fault generation via FPGA-based pulse generation
- XMEGA (PDI), AVR (ISP), and STM32F (UART Serial) bootloader built-in

8. Practical Lab: ChipWhisprer-Lite Quick-Start Guide

Hardware Installation

The hardware setup is fast and easy! simply use a micro USB cable to connect the ChipWhisperer-Lite to your computer.

Once that's done, you can open the installed version of Chipwhisperer on your computer. If it is not installed yet, please follow this link to install it on Windows/Mac or Linux OS:

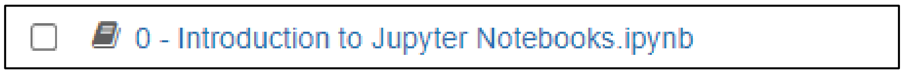
https://chipwhisperer.readthedocs.io/en/latest/index.html#overview

Software Start Guide

The software interface of the ChipWhisperer is basically based on a Python 3 package. In order to connect to the hardware, Jupyter is used. Jupyter also is used as a code editor.

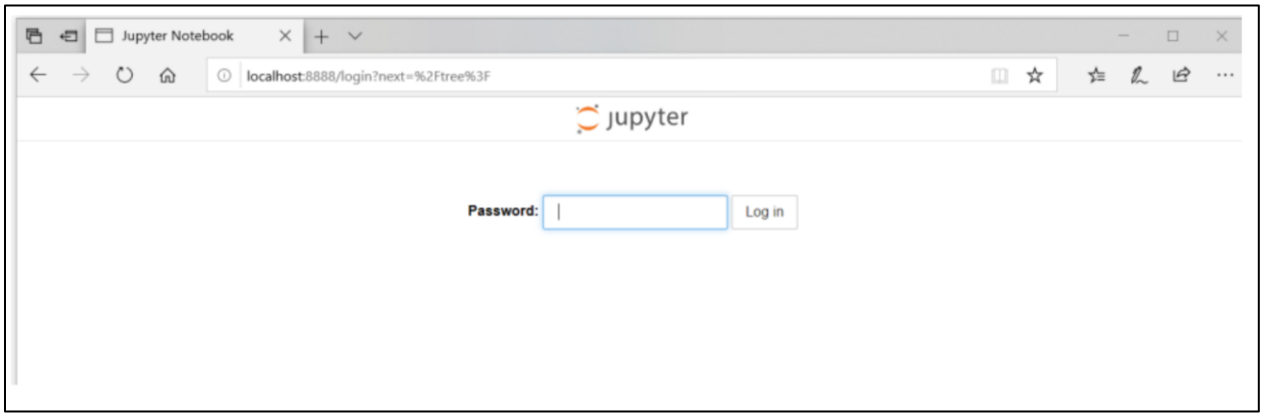
Some reminders for using a Jupyter notebook:

- **Ctrl+Enter** to execute the contents of a cell.
- **Shift+Enter** to execute the contents of a cell and move to the next cell.
- a cell is running when the symbol **[*]** is present to its left. When the execution is complete, this symbol becomes **[N]**, where **N** indicates that this cell is the Nth to have been executed.
- For more practice on Jupyter, please see **"Introduction to jupyter notebook. ipynb".**

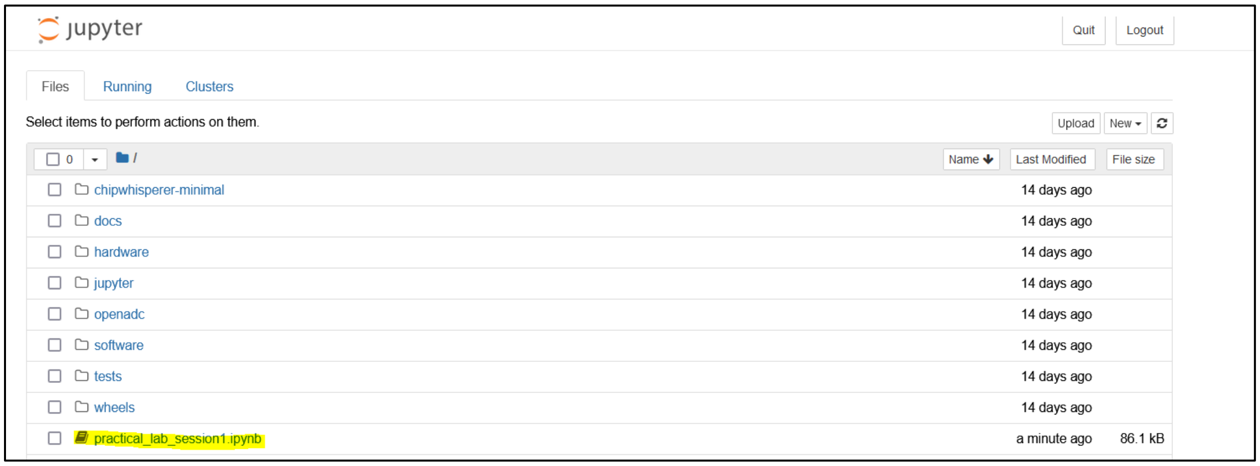


0 - Introduction to Jupyter Notebooks.ipynb

In order to open the software interface, you need to open a web browser (e.g., Chrome-firefox) and type in 127.0.0.1:8888 or localhost:8888 and then enter your password (initial password has been set to "**vagrant**"):



In jupyter, you can either go through the prepared version called **"practical_lab_session1.ipynb"** or create your own item and write the codes inside that:



Select items to perform actions on them.

Name	Last Modified	File size
chipwhisperer-minimal	14 days ago	
docs	14 days ago	
hardware	14 days ago	
jupyter	14 days ago	
openadc	14 days ago	
software	14 days ago	
tests	14 days ago	
wheels	14 days ago	
practical_lab_session1.ipynb	a minute ago	86.1 kB

Definition of variables:

We define here the global variables that will be useful for the project. The last CW_PATH is to be adapted to your ChipWhisperer installation: it specifies the directory where the module is installed.

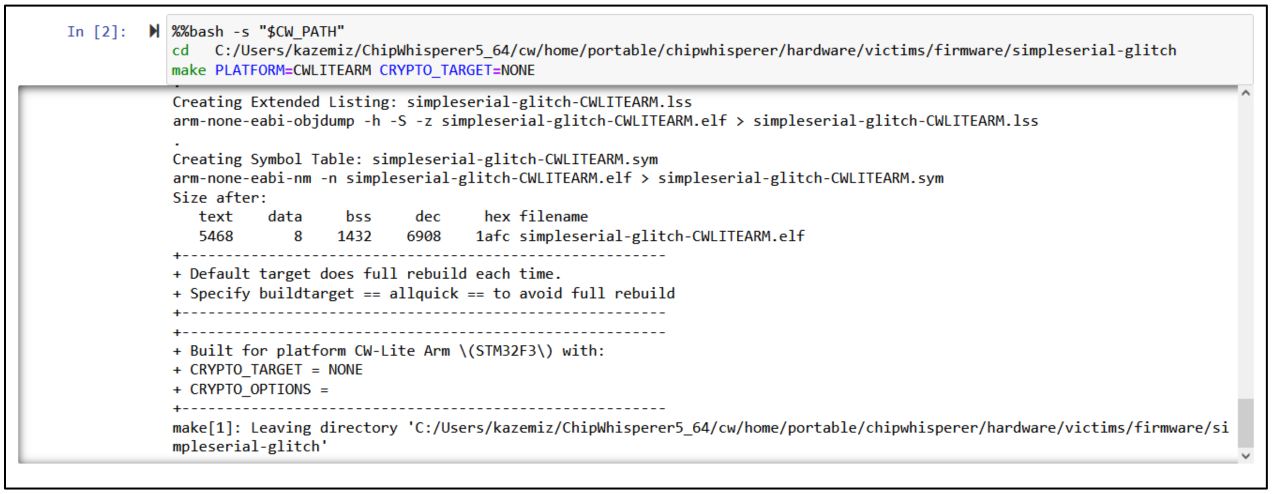
```
In [1]:  SCOPETYPE = 'OPENADC'
         PLATFORM = 'CWLITEARM'
         CW_PATH = '/home/vagrant/work/projects/chipwhisperer/'
```

Firmware Compilation:

We start by compiling the firmware that we are going to attack. The execution of the cell below should end with the following message:

```
+--------------------------------------------------------
+ Built for platform CW-Lite Arm \(STM32F3\) with:
+ CRYPTO_TARGET = NONE
+ CRYPTO_OPTIONS =
+--------------------------------------------------------
```

Here is an example of our firmware compilation:



```
In [2]:  %%bash -s "$CW_PATH"
         cd   C:/Users/kazemiz/ChipWhisperer5_64/cw/home/portable/chipwhisperer/hardware/victims/firmware/simpleserial-glitch
         make PLATFORM=CWLITEARM CRYPTO_TARGET=NONE

         Creating Extended Listing: simpleserial-glitch-CWLITEARM.lss
         arm-none-eabi-objdump -h -S -z simpleserial-glitch-CWLITEARM.elf > simpleserial-glitch-CWLITEARM.lss
         .
         Creating Symbol Table: simpleserial-glitch-CWLITEARM.sym
         arm-none-eabi-nm -n simpleserial-glitch-CWLITEARM.elf > simpleserial-glitch-CWLITEARM.sym
         Size after:
            text    data     bss     dec     hex filename
            5468       8    1432    6908    1afc simpleserial-glitch-CWLITEARM.elf
         +--------------------------------------------------------
         + Default target does full rebuild each time.
         + Specify buildtarget == allquick == to avoid full rebuild
         +--------------------------------------------------------
         +--------------------------------------------------------
         + Built for platform CW-Lite Arm \(STM32F3\) with:
         + CRYPTO_TARGET = NONE
         + CRYPTO_OPTIONS =
         +--------------------------------------------------------
         make[1]: Leaving directory 'C:/Users/kazemiz/ChipWhisperer5_64/cw/home/portable/chipwhisperer/hardware/victims/firmware/si
         mpleserial-glitch'
```

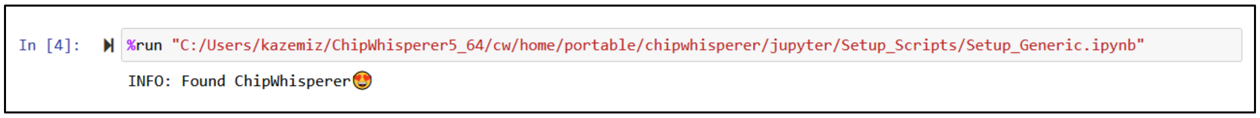
Target Connection:

Executing the cell below should display the following message:

"INFO: Found ChipWhisperer😍 !"

If not, the card is not recognized by the PC. No need to go any further until you have solved the problem.

Here is an example of our target connection:

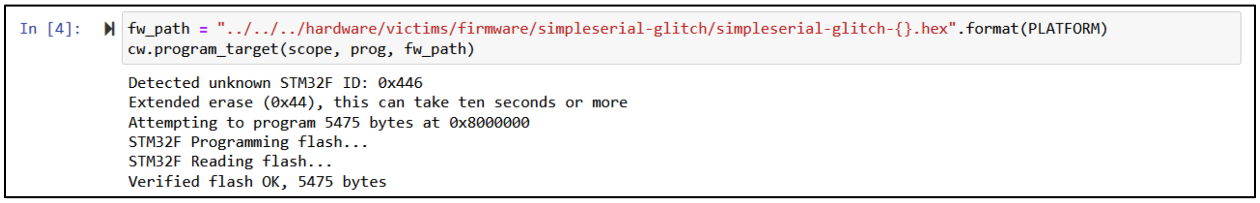


```
In [4]:  %run "C:/Users/kazemiz/ChipWhisperer5_64/cw/home/portable/chipwhisperer/jupyter/Setup_Scripts/Setup_Generic.ipynb"

         INFO: Found ChipWhisperer😍
```

Programming of the microcontroller and reset of the card:

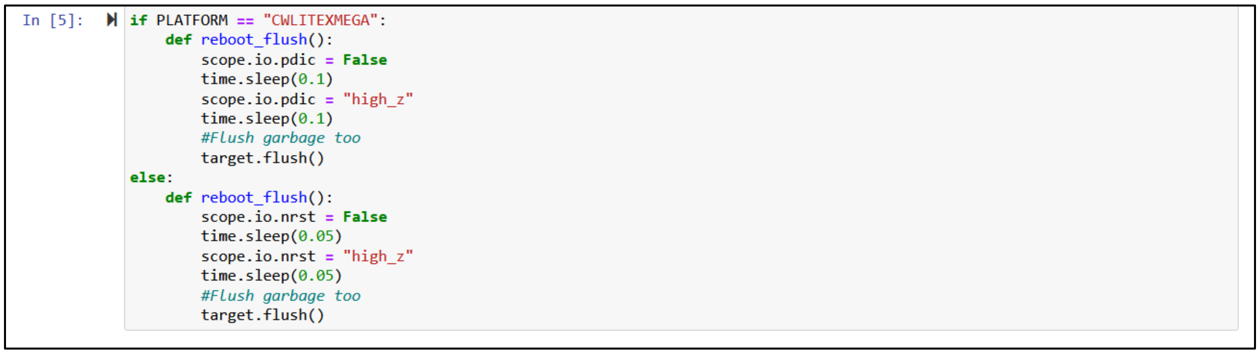
Here, we are going to program the microcontroller with the firmware that we compiled above and write the previously compiled code in the flash memory of the microcontroller. Then, we reset the card before using it. After the reset, the microcontroller will load the firmware from its flash memory and run it.

Executing the cell below should give the following message: **"Verified flash OK".**



```
In [4]:  fw_path = "../../../hardware/victims/firmware/simpleserial-glitch/simpleserial-glitch-{}.hex".format(PLATFORM)
         cw.program_target(scope, prog, fw_path)

         Detected unknown STM32F ID: 0x446
         Extended erase (0x44), this can take ten seconds or more
         Attempting to program 5475 bytes at 0x8000000
         STM32F Programming flash...
         STM32F Reading flash...
         Verified flash OK, 5475 bytes
```

Note that, whenever you want to reset the target you can use the code block below:



```
In [5]:  if PLATFORM == "CWLITEXMEGA":
             def reboot_flush():
                 scope.io.pdic = False
                 time.sleep(0.1)
                 scope.io.pdic = "high_z"
                 time.sleep(0.1)
                 #Flush garbage too
                 target.flush()
         else:
             def reboot_flush():
                 scope.io.nrst = False
                 time.sleep(0.05)
                 scope.io.nrst = "high_z"
                 time.sleep(0.05)
                 #Flush garbage too
                 target.flush()
```

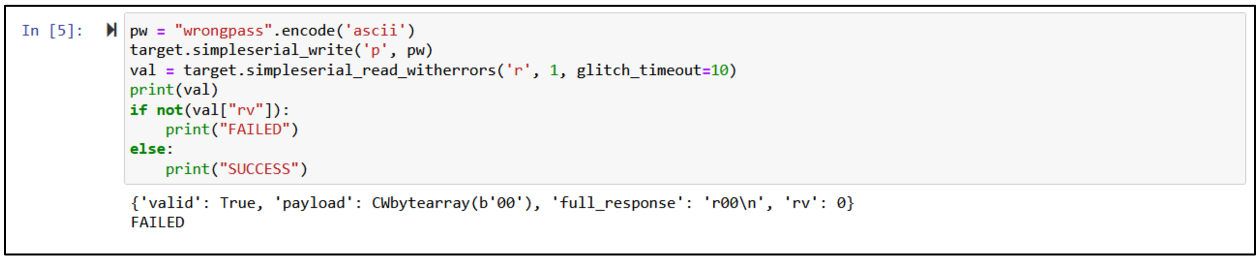
9. Student Activity: a simple example of cock glitching fault injection attack

At the end of this session, the goal is to use Chipwhisprer to perform a simple clock glitching attack against a password checking function. This function is described below:

This function accepts a correct PIN and refuses a false PIN. Since our goal is to be accepted with a wrong PIN, we have not implemented the functionality of locking the system after 3 wrong PINs. The correct password is "**correctpass**".

First Scenario: normal functionality with the wrong password

We start by checking that an incorrect password is indeed refused. The password is defined in the pw variable and explicitly encoded in ascii. We communicate via a serial link with the card. We, therefore, send the password **"wrongpass"** to the card, preceded by the letter p with the simpleserial_write command.

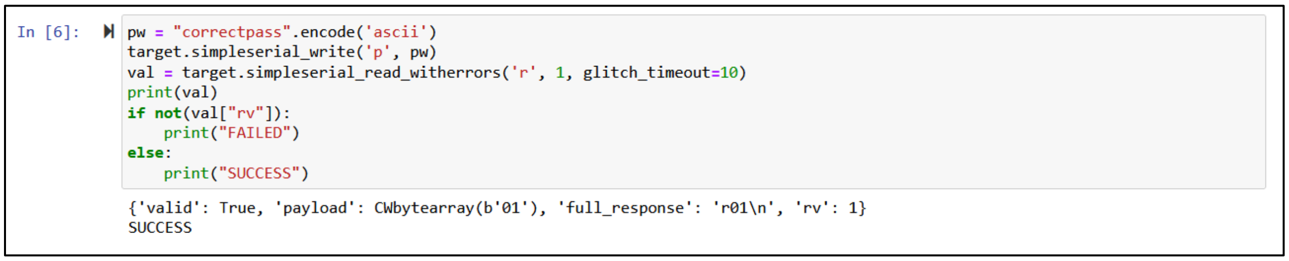


```
In [5]:  pw = "wrongpass".encode('ascii')
         target.simpleserial_write('p', pw)
         val = target.simpleserial_read_witherrors('r', 1, glitch_timeout=10)
         print(val)
         if not(val["rv"]):
             print("FAILED")
         else:
             print("SUCCESS")

         {'valid': True, 'payload': CWbytearray(b'00'), 'full_response': 'r00\n', 'rv': 0}
         FAILED
```

We then read the response from the card, with the start by sending the letter r with the simpleserial_read_witherrors command. The result obtained is stored in the variable val. The result obtained is a data structure. The field that interests us is rv (return value) which indicates the return value obtained following the sending of the password. Here, since we sent an incorrect password, we should have rv=0. We check that this is indeed the case by displaying FAILED.

Second Scenario: normal functionality with the correct password

This time we send the correct password ("**correctpass**"), following the same procedure as above. Here, since we sent the correct password, we should have `rv=1`. We verify that this is indeed the case by displaying `SUCCESS`.

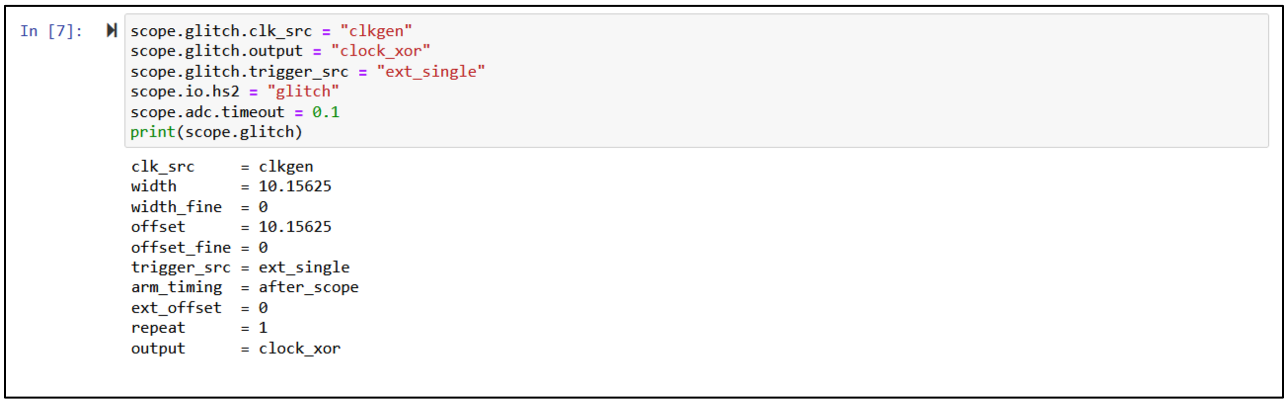


```
In [6]:  pw = "correctpass".encode('ascii')
         target.simpleserial_write('p', pw)
         val = target.simpleserial_read_witherrors('r', 1, glitch_timeout=10)
         print(val)
         if not(val["rv"]):
             print("FAILED")
         else:
             print("SUCCESS")

         {'valid': True, 'payload': CWbytearray(b'01'), 'full_response': 'r01\n', 'rv': 1}
         SUCCESS
```

Third Scenario: Bypassing the password checking with a wrong password

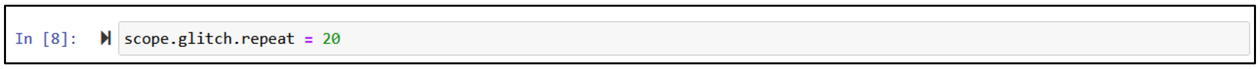
After having validated the nominal operation of our code, we will try to disturb it by injecting a fault into a clock glitch.

The cell below sets the parameters required for fault injection (**width**: glitch width, **offset**: glitch offset, **ext_offset**: glitch delay). It is therefore these three parameters that we will have to adjust. The call to print displays the parameters.



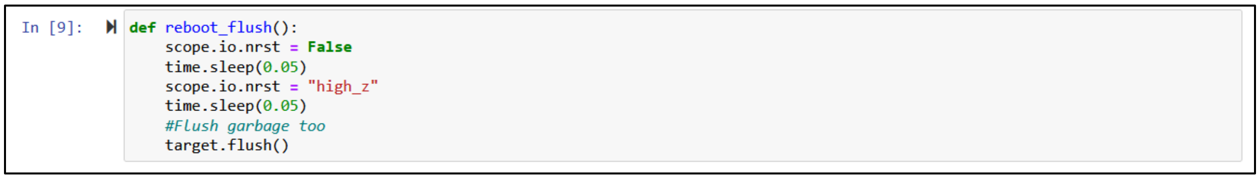
```
In [7]:  scope.glitch.clk_src = "clkgen"
         scope.glitch.output = "clock_xor"
         scope.glitch.trigger_src = "ext_single"
         scope.io.hs2 = "glitch"
         scope.adc.timeout = 0.1
         print(scope.glitch)

         clk_src     = clkgen
         width       = 10.15625
         width_fine  = 0
         offset      = 10.15625
         offset_fine = 0
         trigger_src = ext_single
         arm_timing  = after_scope
         ext_offset  = 0
         repeat      = 1
         output      = clock_xor
```

- **Note:** The goal in the following is going to be to find the width and offset values that lead to a successful attack. In order to reduce the space of parameters to be explored, we are going to fix the glitch delay setting by performing multiple consecutive clock glitches. For this, we will set the repeat parameter to 20 for example, to glitch 20 consecutive clock cycles:



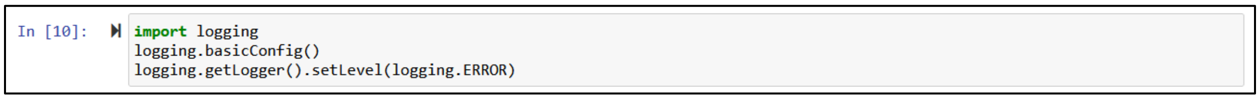
```
In [8]:  scope.glitch.repeat = 20
```

- **Note:** Sometimes the fault injection attack can result in the crash of the card. In this case, it will be necessary to do a *reset*. We define here a function that is responsible for performing a *reset* of the card in case it no longer responds.



```
In [9]:  def reboot_flush():
              scope.io.nrst = False
              time.sleep(0.05)
              scope.io.nrst = "high_z"
              time.sleep(0.05)
              #Flush garbage too
              target.flush()
```

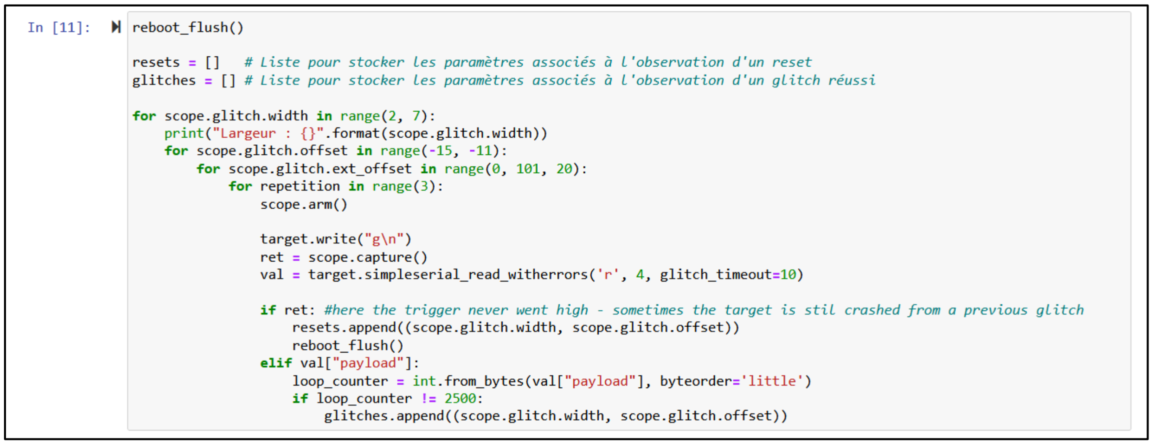
- **Note:** We need also to filter the *warnings* emitted during the execution of the characterization code so as not to fill the terminal output too much.



```
In [10]:  import logging
          logging.basicConfig()
          logging.getLogger().setLevel(logging.ERROR)
```

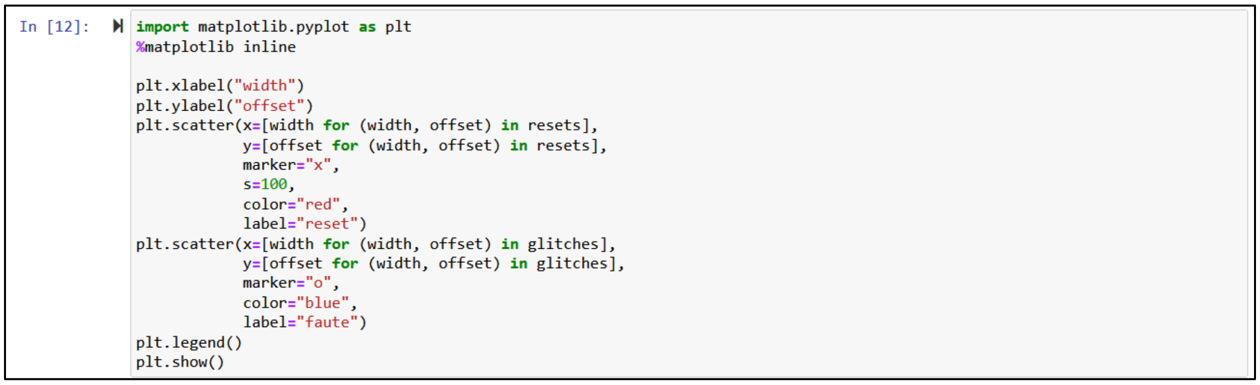
In the following, we are trying to explore the clock glitch offset and width for the successful fault injection attack:

Based on the needed precision, you choose the value ranges to explore: width: possible values are in the range [0:50]. If the glitch width is so small, a glitch would not even be detected. On the other hand, a very large glitch width will result in a glitch that will have no effect because the instruction will have time to execute normally. Possible values for glitch offset are in the range of [-50,50]. It is better to limit ourselves to negative values so that the glitch arrives before the rising edge. It should be mentioned that very small glitch offsets will not be detected. On the other hand, a very large glitch offset will result in a glitch that will have no effect because the instruction will have time to execute normally. It's up to you to set the correct intervals in the cell below and then run it.



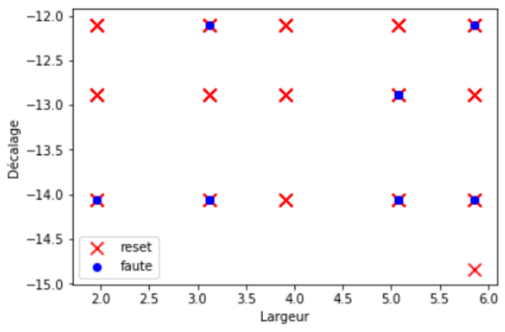
```
In [11]:  reboot_flush()

          resets = []   # Liste pour stocker les paramètres associés à l'observation d'un reset
          glitches = [] # Liste pour stocker les paramètres associés à l'observation d'un glitch réussi

          for scope.glitch.width in range(2, 7):
              print("Largeur : {}".format(scope.glitch.width))
              for scope.glitch.offset in range(-15, -11):
                  for scope.glitch.ext_offset in range(0, 101, 20):
                      for repetition in range(3):
                          scope.arm()

                          target.write("g\n")
                          ret = scope.capture()
                          val = target.simpleserial_read_witherrors('r', 4, glitch_timeout=10)

                          if ret: #here the trigger never went high - sometimes the target is stil crashed from a previous glitch
                              resets.append((scope.glitch.width, scope.glitch.offset))
                              reboot_flush()
                          elif val["payload"]:
                              loop_counter = int.from_bytes(val["payload"], byteorder='little')
                              if loop_counter != 2500:
                                  glitches.append((scope.glitch.width, scope.glitch.offset))
```

To identify the parameters giving a successful attack, we can draw a graph from the data collected The next cell plots this scatter plot, from the `glitches` and `resets` lists constructed previously.

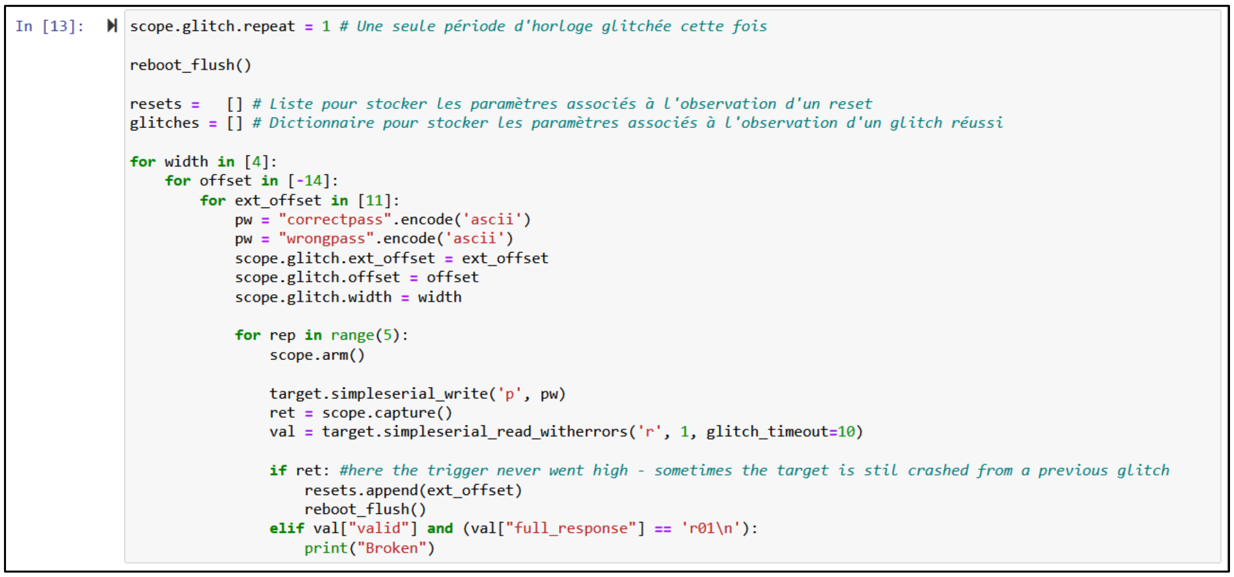


```
In [12]:  import matplotlib.pyplot as plt
          %matplotlib inline

          plt.xlabel("width")
          plt.ylabel("offset")
          plt.scatter(x=[width for (width, offset) in resets],
                      y=[offset for (width, offset) in resets],
                      marker="x",
                      s=100,
                      color="red",
                      label="reset")
          plt.scatter(x=[width for (width, offset) in glitches],
                      y=[offset for (width, offset) in glitches],
                      marker="o",
                      color="blue",
                      label="faute")
          plt.legend()
          plt.show()
```

Here is an example of plotted graph:



We have now identified the faulty values for the width and offset parameters. To perform a fault injection attack on a given code, we only have to adjust the delay parameter:



```
In [13]:  scope.glitch.repeat = 1 # Une seule période d'horloge glitchée cette fois

          reboot_flush()

          resets =   [] # Liste pour stocker les paramètres associés à l'observation d'un reset
          glitches = [] # Dictionnaire pour stocker les paramètres associés à l'observation d'un glitch réussi

          for width in [4]:
              for offset in [-14]:
                  for ext_offset in [11]:
                      pw = "correctpass".encode('ascii')
                      pw = "wrongpass".encode('ascii')
                      scope.glitch.ext_offset = ext_offset
                      scope.glitch.offset = offset
                      scope.glitch.width = width

                      for rep in range(5):
                          scope.arm()

                          target.simpleserial_write('p', pw)
                          ret = scope.capture()
                          val = target.simpleserial_read_witherrors('r', 1, glitch_timeout=10)

                          if ret: #here the trigger never went high - sometimes the target is stil crashed from a previous glitch
                              resets.append(ext_offset)
                              reboot_flush()
                          elif val["valid"] and (val["full_response"] == 'r01\n'):
                              print("Broken")
```

```
Broken!
```

When the broken is being printed, it means that you were successful in bypassing the password checking function with the wrong password.

Session 2: High-Level Analysis of Important Patterns and Functions

In this session, our goal is to explain test scenarios and approaches to be able to catch the fault effects at the software level. The following focuses on analyzing the most prominent patterns in the program control flow and common functions.

1. Main Control Flow Patterns and Their Evaluation Methods

In this part, to present the evaluation approach, the important control flow statements are categorized into three main classes, namely: 1) Unconditional Branches, 2) Decision Makings, and 3) Iterative Controls. Table.1 shows these statements with some C code examples.

Table 1. Important Control Flow Statements

Control Flow Statements	Type	Examples
Branching/Skipping	Unconditional	Continue/Break/Go
Decision Making	Conditional	If/If-else
Iterative	Conditional	For/Do-While/While

Unconditional Branches

An unconditional branch is a basic control flow and contains an outgoing edge from a node in a control flow graph (Figure. 4 (a)). In order to catch the fault effect on an unconditional branch, we need to insert a checkpoint (like in Figure. 4 (b)). and then run the program in the presence of the clock glitching attacks.

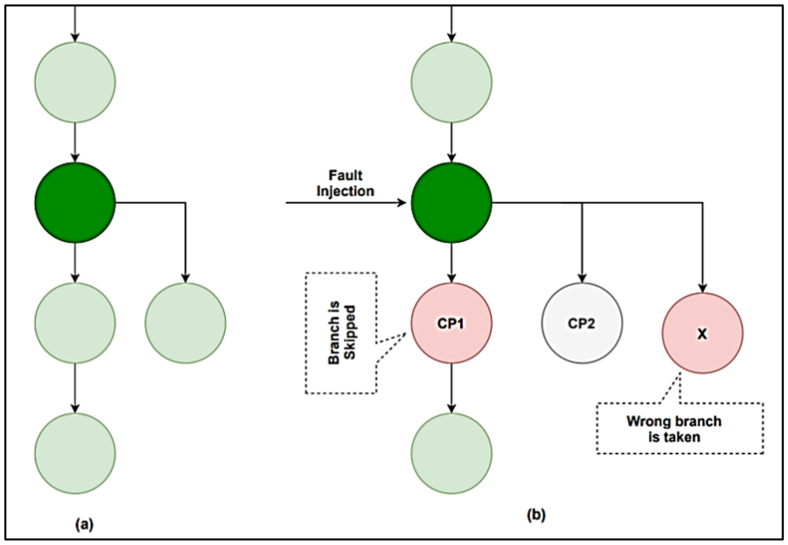


Figure 4 Control Flow Evaluation for Unconditional Branch

When CP1 was activated, the injected fault did not affect the correct execution of the branch, and when the CP2 was set, it showed the branch was corrupted. When none of the checkpoints are activated, it implies that the PC register contains an incorrect instruction memory address (represented as X).

Decision Making or Conditional Branches

The second important category of control flow patterns is the decision-makings such as **(if-else)** which contains a decision node with two control branches (Figure.5 (a)). The conditional control branches are merged after executing the statements of the selected branch (white circles). In order to evaluate these patterns against clock glitching attacks, we need to first set the condition to a state which leads to a known result. Then, we need to insert checkpoints to catch the fault effects and monitor the consequences. For example, in Figure.5 (b) then CP1 and CP2 are inserted to monitor the fault effects. In this example, it is expected that the condition is false, so when the fault injection is unsuccessful, the CP2 is activated. When the CP1 is set, one can detect the skipped conditional test.

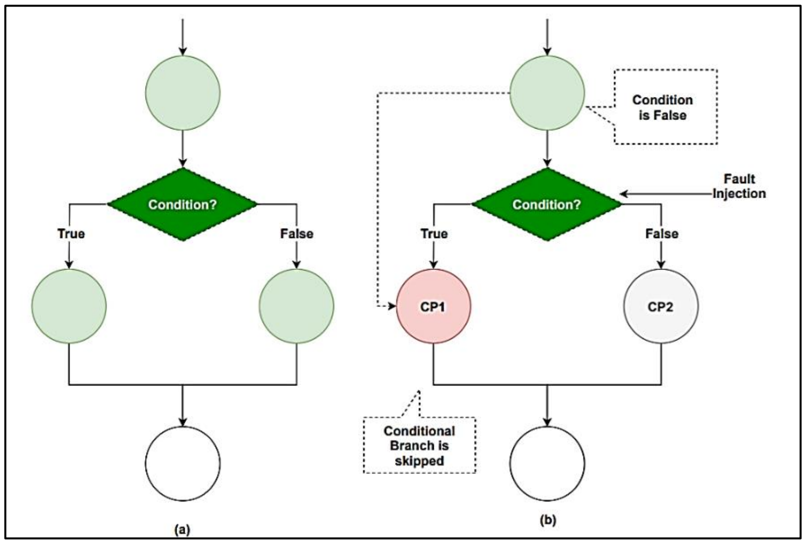


Figure 5 Control Flow Evaluation of Single Conditional Branch

Iterative patterns

Iterative control statements are playing an important role in the correct application execution. For example, the while loop is one of the iterative control statements. First, we need to set an always true condition such as while (1), then insert the CP1 as a watchdog flag to detect the skip from the loop. In this case, if CP1 becomes active, it demonstrates that the fault injection has manipulated the correct execution of the loop.

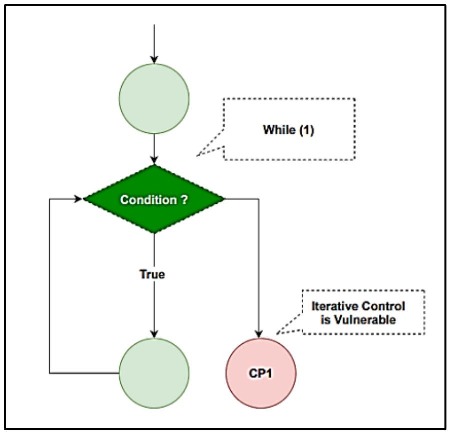


Figure 6. Control Flow Evaluation of an Iterative Control

2. C-Functions and Their Evaluation Methods

This section aims to explain more general evaluation scenarios to exploit the vulnerabilities of standard high-level C functions. First, the standard embedded C-functions are categorized, including **1) Type Casting, 2) String Manipulation, 3) Memory-Based,** and **4) Searching and Sorting Functions**.

There are different evaluation scenarios for different C-functions. In the following, we will explain these approaches to evaluate these functions.

Type Casting Functions

These functions are used to perform data type conversion from one type to another. Two important examples are **atoi** and **itoa**, which convert string to int and int to string, respectively.

Evaluation Method: to test the Type Casting Functions, an (input, output) pair is selected and the function runs in the presence of FIA. If the generated result differs from the expected one, a successful attack is reported.

String Manipulation Functions

These functions are used to modify the strings. There are various functions in this class, including **strcp** and **strncpy** to copy a string to another.

Evaluation Method: For the Sring Manipulation Functions that transfer or copy data from a source to a destination, the results can be compared to detect any possible mismatch.

Memory-Based Functions

These functions manipulate the data inside the memory and are specifically vital for a system's initialization. Important examples of these functions are **memset** and **memcpy,** which are used to set all the bytes in a block of memory to a particular value and to copy a block of data from a source address to a destination address.

Evaluation Method: The Memory-Based Functions are assessed by feeding them with known values and checking the specific memory location(s) related to the operation. A fault-affected function will result in an incorrect memory address or value.

Searching and Sorting Functions

These functions include examples such as **bsearch** and **qsort**. A bsearch sorts an array and then searches the desired record based on the binary search tree algorithm. A qsort function sorts an array of numbers. To evaluate it, each element is weighed at the output array.

Evaluation Method: The output of the Searching Functions can be observed when a known array is given to the function, and when they return a null value, it means that the attack was successful. To evaluate the Sorting Functions, each element of the output array is weighed. Then, when the sorted array is generated, the sum of all the multiplication of weights and related elements of the arranged array are compared. If these two are not equal, it means wrong sorting.

3. Student Activities: Experimental Evaluaiton of Patterns and Functions

Evaluating different pattrens

- Using the project that you have done in the first session, try to evaluate and report the successful clock glitch configuration for three important patterns: 1) unconditional branch, 2) simple conditional branch (if-else), and 3) nested conditional branch (nested ifs)

Note: you need to draw the graph for successful glitch width and offset

Comparing the security of different patterns

- Based on the obtained result of the first activity, try to explain and compare the security of different patterns.

Two-step password checking function

- Try to extend the first session example and write a two-step password checking (with nested conditional branch) and compare the results.

Reviewing the important C-Functions

Table.2 summarizes different categories of the standard library functions with their normal behavior.

- Please complete the last column for their faulty behavior in front of FIA. As an example, the faulty behavior of atoi function has been given.

Table 2. The behavior of different high-level C-functions

C-Function	Normal Behavior	Faulty behavior in front of FIA
atoi	Convert an ASCII array to an integer value	Returns un-expected and wrong integer value for a known ASCII array
itoa	Convert an integer value to an ASCII array	…
memset	Saves a value in memory	

memcpy	Compares the values in two different memory locations	
strcpy	Copies from one character array to another	
strncpy	Copies a portion (n-bit) of one character array to another	
strchr	Finds the first occurrence of a character in a string	
strtod	Converts string to a double value	
qsort	Sorts an input array	
bsearch	Searches an array to find a value	

Find the glitch configurations to attack different functions

- Try to evaluate all the functions in the table.2 using Chipwhisprer and report all the successful glitch configurations.

Compare the security of different functions

- Based on the obtained results from the previous activity, try to compare the security of different functional categories.
- Which category is the most difficult one to be analyzed?
- From the string manipulation category, can you compare the exploited vulnerability of strcpy and strncpy? Which one is more vulnerable?
    - Try to explain your comparison by studing these functions.

Session 3: Evaluation of an Embedded Application

Different high-level analysis methods have been proposed in the previous session. In this session, our goal is to apply these approaches to assess an embedded application which is more complex and has different modules. Then, we need to show how a clock gliching attack can affect the credential information, the data/control flow integrity, and the availability of an embedded application. As a case study we will focus on a medical IoT application named Sec-Pump application. This example shows the potential vulnerabilities of a critiacal application which result in dangrous consequences for the patients. The final goal of this part is to propose a few software-level examples tomitigate the mentioned impacts on an application's security level.

1. An Introduction to Sec-Pump Application

An excellent example of a critical embedded application exists as an infusion pump installed in hospitals to deliver doses of drugs to patients and monitor their health status. Sec-Pump is an open-source and security oriented medical application was selected to model the behavior of a life-critical infusion pump. Sec-Pump has been designed to be used for both research and teaching activities related to embedded system security and runs on both ARM and RISC-V microprocessors.

All the necessary information can be found in " https://github.com/r3glisss/SecPump ". Please, try to carefully read this github page and download the code. You can download the version to be executed on ARM processor. However, if you have access to risc-v target board of Chipwhisprer Pro., you can also use it.

2. Student Activities: Exploit the Vulnerabilities of Sec-Pump Application

In this session, you need to evaluate different parts of the sec-pump application.

Evaluation of the Sec-Pump's Authentication Module

Sec-Pump has a single-step authentication process. When the Sec-Pumpboots, it enters an infinite loop for password entry. If an attacker can bypass this password checking step, he/she can access the entire system.

- Try to find the related code block to the authentication module. The try to perform clock glitching attack using chipwhisprer and report the configurations for the successful attacks.

Evaluation of the Sec-Pump's Drug Manager Module

DrugManagement is another critical module that manipulates the central data/control flows of the Sec-Pump application. To evaluate this module, first, the main functions and variables in this module are determined. The Drug Management module's core functions are 1) Create-Cure, 2) Modify-Cure, and 3) Delete-Cure. These three functions operate on three variables: 1) Cure-Name, 2) Cure-Volume, and 3) Cure-Duration. Cure-Volume and Cure-Duration directly impact the Sec-Pump's critical functionalities because they have a direct relation with the time and amount of injected medicine. Consequently, they need to be protected against any kind of attacks, including FIAs.

- First, try to review the Create-Cure part. It forms a flexible mechanism to initiate a cure with a Cure-Name to receive the inputs (Cure-Volume, and Cure-Duration) and to initialize them.
- Which functions in this Create-Cure part seem to be vulnerable?
- The strcpy function copies the string from the name to the Cure-Name variable and returns the copied string. Try to evaluate the strcpy function in this part and report the successful attack's configurations.
- What are the results of the faulty behavior of strcpy in the Sec-Pump's expected functionality?
- The atoi function in the Create-Cure module converts a string, such as entered volume and duration, to a number (specifically an integer). Try to evaluate one of the atoi functions in this part and report the successful attack's configurations.
- What are the results of the faulty behavior of atoi in the Sec-Pump's expected functionality?
- Try to review the Delete-Cure part. This part is responsible for eliminating a cure and its related data (Cure-Name, Cure-Volume, and Cure-Duration) from memory. Erasing the Cure-name is done by filling its memory block with zeros. Accordingly, a C library function named memset has been called to copy 0x0 to the 32 first characters

of the memory block where the Cure-Named is pointed to. Try to attack memset function in the Delete-Cure part and report the successful attack's configurations.

Securing the Sec-Pump's Authentication Module

It was shown that the single-step authentication module is vulnerable and may be bypassed by a simple clock glitching FIA. Therefore, it is needed to use an alternative for it.

- Try to modify the authentication module and include a two steps authentication using nested conditional branch. After performing the clock glithcing and reporting the configurations for the successful attack, try to compare the results with the previous task.

Securing the Sec-Pump's Drug Manager Module

In this part, some robust software-level alternatives are proposed to mitigate the impact of existing vulnerabilities in different high-level instructions of the Drug-Management module.

One of the vulnerable functions in the Drug-Management module is the strcpy function. Generally, it is used to copy a string fromthe source to the destinationwith a null character termination. This operation does not specify the length of the copied string. The strcpy function in the Drug Management module is used to copy "name" to a destination "cure.name". This vulnerability originates from the fact that the strcpy operation continues to copy into the destination location until it reaches the null character. In the presence of a clock glitch, the processor may not have enough time to detect the transferred NULL character and continues to copy the string. This is the reason behind the unexpected copied string in the curename of the Sec-Pump.

- Try to suggest an alternative function for strcpy in Drug Management Module and experimentally show that this alternative is less vulnerable against clock glitching attack. (you can perform clock glithcing and compare the success rate of fault injection attacks)

Another vulnerable function in the Drug-Management module is atoi. This function converts a string into its integer numerical representation. The atoi function has been used in theDrug

Management module to convert the related ASCII string as an argument (e.g., Volume and Duration values) to integer form. At some point, in the presence of clock glitching FIA, this function returns an undefined integer or zero, which does not correspond to the received argument. This faulty behavior can originate from the fact that this function works iteratively (convert characters from lef to right), and in the presence of a clock glitch, it may miss one character and return the first valid number that can be converted from the received string. Also, atoi expects a null-terminated string as an input, and a clock glitch can affect the observation of the null character.

- Try to suggest an alternative function for atoi to increase the security of the sec-pump application.
- Can we include redundancy to secure such functions? Try to add a redundant atoi function and report the attack results. Then compare with the vulnerability of a single atoi function.

Final Solution: Random Delays and Make the Execution Timing Unpredictable

Another solution for vulnerable function is to make the execution timing unpredictable. in this case, the programmer needs to consider a loop with random delays. Like so, the probability of successful synchronization for clockglitching FIA is too low.

- Try to add random delays in Authentication Module and compare with the initial version.
- Try to add random delays in Drug Manager Module and compare with the initial version.