# DFA attack on AES-128

# Aims

The aim of this notebook is to show an alternative to DPA and CPA by using DFA to recover the secret key of an AES.

# Hardware assembly

This notebook is configure for the STM32F3 target.

**Summary**

- Identify the last rounds (r8, r9, r10)
- Make faults before MxC r9
- Perform Piret's Attack
- Make faults before MxC r8
- Perform Piret's Attack

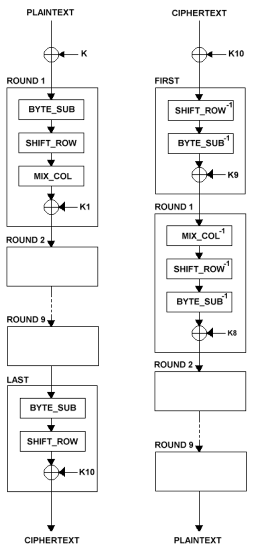
In [1]:
```
SCOPETYPE = "OPENADC"
PLATFORM = 'CW308_STM32F3'
CRYPTO_TARGET = "TINYAES128C"
```

# Prerequisites

## Brief introduction to Differential Fault Analysis

Each round is composed of 4 operations except for the last, which has no mixing column (this round is smaller than the others).

- Sub-bytes: passe in the S-box
- Shift rows: shifting of tines
- Mix Columns: Mixing of columns
- Add Round Key: Apply a xor between the intermediate state and the round key.




## Installing dependencies

Firstly, let's install `phoenixAES` in the current kernel environment:

In [2]:
```
import sys
!{sys.executable} -m pip install phoenixAES
```

```
Requirement already satisfied: phoenixAES in /usr/local/lib/python3.11/sit
e-packages (0.0.5)
WARNING: Running pip as the 'root' user can result in broken permissions a
nd conflicting behaviour with the system package manager, possibly renderi
ng your system unusable. It is recommended to use a virtual environment in
stead: https://pip.pypa.io/warnings/venv. Use the --root-user-action optio
n if you know what you are doing and want to suppress this warning.
```

## Building the target firmware

In [3]:
```
%%bash -s "$PLATFORM" "$CRYPTO_TARGET"
cd ../chipwhisperer/firmware/mcu/simpleserial-aes
make PLATFORM=$1 CRYPTO_TARGET=$2
```

```
Building for platform CW308_STM32F3 with CRYPTO_TARGET=TINYAES128C
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
Blank crypto options, building for AES128
.
Welcome to another exciting ChipWhisperer target build!!
arm-none-eabi-gcc (15:14.2.rel1-1) 14.2.1 20241119
Copyright (C) 2024 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOS
E.

mkdir -p objdir-CW308_STM32F3
.
Compiling:
-en     simpleserial-aes.c ...
-e Done!
.
Compiling:
-en     ../simpleserial/simpleserial.c ...
-e Done!
.
Compiling:
-en     ../hal/hal.c ...
-e Done!
.
Compiling:
-en     ../hal//stm32f3/stm32f3_hal.c ...
-e Done!
.
Compiling:
-en     ../hal//stm32f3/stm32f3_hal_lowlevel.c ...
-e Done!
.
Compiling:
-en     ../hal//stm32f3/stm32f3_sysmem.c ...
-e Done!
.
Compiling:
-en     ../crypto/tiny-AES128-C/aes.c ...
-e Done!
.
Compiling:
-en     ../crypto/aes-independant.c ...
-e Done!
.
Assembling: ../hal//stm32f3/stm32f3_startup.S
arm-none-eabi-gcc -c -mcpu=cortex-m4 -I. -x assembler-with-cpp -mthumb -mf
loat-abi=soft -fmessage-length=0 -ffunction-sections -DF_CPU=7372800 -Wa,-
gstabs,-adhlns=objdir-CW308_STM32F3/stm32f3_startup.lst -I../simpleseria
l/ -I../hal/ -I../hal/ -I../hal//stm32f3 -I../hal//stm32f3/CMSIS -
I../hal//stm32f3/CMSIS/core -I../hal//stm32f3/CMSIS/device -I../ha
l//stm32f4/Legacy -I../simpleserial/ -I../crypto/ -I../crypto/tiny-A
ES128-C ../hal//stm32f3/stm32f3_startup.S -o objdir-CW308_STM32F3/stm32f
3_startup.o
.
LINKING:
-en     simpleserial-aes-CW308_STM32F3.elf ...
Memory region         Used Size  Region Size  %age Used
             RAM:        2416 B        40 KB      5.90%
```

```
             ROM:        6084 B       256 KB      2.32%
-e Done!
.
Creating load file for Flash: simpleserial-aes-CW308_STM32F3.hex
arm-none-eabi-objcopy -O ihex -R .eeprom -R .fuse -R .lock -R .signature s
impleserial-aes-CW308_STM32F3.elf simpleserial-aes-CW308_STM32F3.hex
.
Creating load file for Flash: simpleserial-aes-CW308_STM32F3.bin
arm-none-eabi-objcopy -O binary -R .eeprom -R .fuse -R .lock -R .signature
simpleserial-aes-CW308_STM32F3.elf simpleserial-aes-CW308_STM32F3.bin
.
Creating load file for EEPROM: simpleserial-aes-CW308_STM32F3.eep
arm-none-eabi-objcopy -j .eeprom --set-section-flags=.eeprom="alloc,load"
\
--change-section-lma .eeprom=0 --no-change-warnings -O ihex simpleserial-a
es-CW308_STM32F3.elf simpleserial-aes-CW308_STM32F3.eep || exit 0
.
Creating Extended Listing: simpleserial-aes-CW308_STM32F3.lss
arm-none-eabi-objdump -h -S -z simpleserial-aes-CW308_STM32F3.elf > simple
serial-aes-CW308_STM32F3.lss
.
Creating Symbol Table: simpleserial-aes-CW308_STM32F3.sym
arm-none-eabi-nm -n simpleserial-aes-CW308_STM32F3.elf > simpleserial-aes-
CW308_STM32F3.sym
Size after:
   text    data     bss     dec     hex filename
   5548     536    1888    7972    1f24 simpleserial-aes-CW308_STM32F3.elf
+--------------------------------------------------------
+ Default target does full rebuild each time.
+ Specify buildtarget == allquick == to avoid full rebuild
+--------------------------------------------------------
+--------------------------------------------------------
+ Built for platform CW308T: STM32F3 Target with:
+ CRYPTO_TARGET = TINYAES128C
+ CRYPTO_OPTIONS = AES128C
+--------------------------------------------------------
```

# Attack setup

## CW-lite connection and target flashing

Connect to the Chipwhisperer:

In [4]:
```
%run "../chipwhisperer/chipwhisperer-jupyter/Setup_Scripts/Setup_Generic.
```

```
(ChipWhisperer Other WARNING|File __init__.py:69) ChipWhisperer update ava
ilable! See https://chipwhisperer.readthedocs.io/en/latest/installing.html
for updating instructions
INFO: Found ChipWhisperer😍
```

Flash the target:

In [5]:
```
fw_path = "../chipwhisperer/firmware/mcu/simpleserial-aes/simpleserial-ae
cw.program_target(scope, prog, fw_path)
```

```
Detected known STMF32: STM32F302xB(C)/303xB(C)
Extended erase (0x44), this can take ten seconds or more
Attempting to program 6083 bytes at 0x8000000
STM32F Programming flash...
STM32F Reading flash...
Verified flash OK, 6083 bytes
```

In [6]:
```
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()
```

## First execution

For the DFA attack, we need a **constant plaintext** (and **constant key** of course).

In [7]:
```
ktp = cw.ktp.Basic()
ktp.fixed_text = True
ktp.fixed_key = True
key, text = ktp.next()
```

In [8]:
```
scope.clock.adc_src = "clkgen_x1"
scope.adc.samples = 8000
```

Let's test our setup with a first execution, without fault. It will give us the **golden reference output**.

In [9]:
```
# make sure glitches are disabled (in case cells are re-run)
scope.io.hs2 = "clkgen"

trace = cw.capture_trace(scope, target, text, key)
goldciph = trace.textout
master_key = key.hex()
print("Plaintext: {}".format(text.hex()))
print("Key:       {}".format(key.hex()))
print("Ciphertext:{}".format(goldciph.hex()))
```

```
Plaintext: 000102030405060708090a0b0c0d0e0f
Key:       2b7e151628aed2a6abf7158809cf4f3c
Ciphertext:50fe67cc996d32b6da0937e99bafec60
```

In [10]:
```
reset_target(scope)
```

In [11]:
```
from Crypto.Cipher import AES
aes = AES.new(bytes(key), AES.MODE_ECB)
goldciph2 = aes.encrypt(bytes(text))
print("Expected ciphertext:  {}".format(goldciph2.hex()))
```

```
Expected ciphertext:  50fe67cc996d32b6da0937e99bafec60
```

To execute our DFA attack, we need to identify 8th round, 9th round and the 10th round

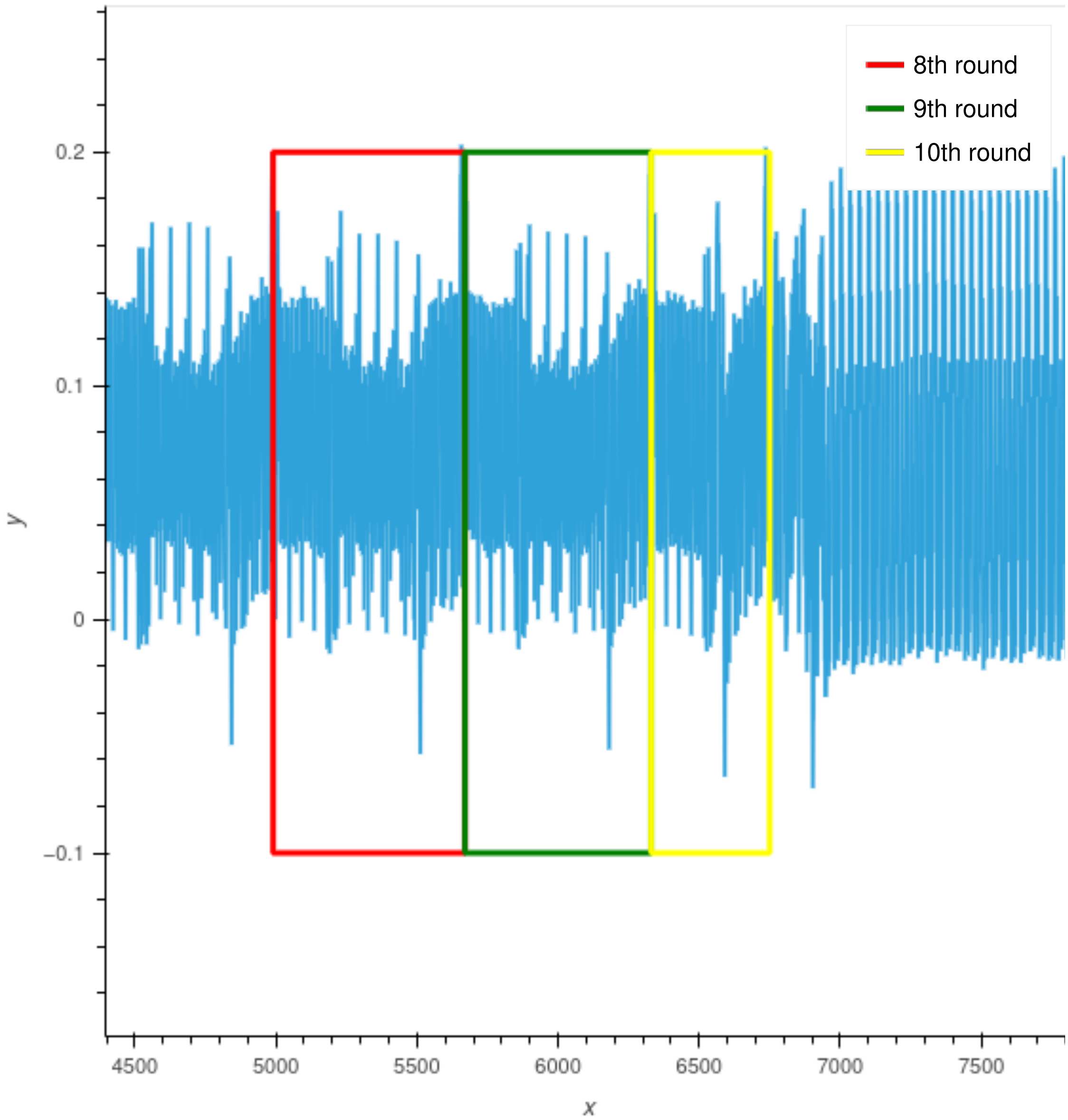
In [13]:
```
import holoviews as hv
from holoviews import opts
hv.extension('bokeh')
```

```
curve = hv.Curve(trace.wave).opts(width=600, height=600)


line = hv.Path([(4990, -0.1), (4990, 0.2), (5670, 0.2), (5670, -0.1), (49
            hv.Path([(5670, -0.1), (5670, 0.2), (6330, 0.2), (6330, -0.1)
            hv.Path([(6330, -0.1), (6330, 0.2), (6750, 0.2), (6750, -0.1)


#plt.show()
(curve * line).opts(opts.Path(line_width=3)).opts(width=600, height=600)
```

Out[13]:




We see clearly the 10 AES-128 rounds, **the 10th round being smaller than the others** as there is no *MixColumn*.

## First glitches

In [14]:
```
import time

# These width/offset numbers are for CW-Lite/Pro; for CW-Husky, convert a
def test_glitches():
```

```
    scope.io.hs2 = "glitch"
    scope.glitch.clk_src = "clkgen"
    scope.glitch.width = -10.15625
    scope.glitch.offset = -13.84
    scope.glitch.trigger_src='continuous'

def stop_test_glitches():
    scope.glitch.trigger_src='ext_single'

test_glitches()
time.sleep(2)
stop_test_glitches()
```

In [15]:
```
print(scope.glitch)
```

```
clk_src     = clkgen
width       = -10.15625
width_fine  = 0
offset      = -13.671875
offset_fine = 0
trigger_src = ext_single
arm_timing  = after_scope
ext_offset  = 0
repeat      = 1
output      = clock_xor
```

Let's see the effect of clock glitches on the AES execution.

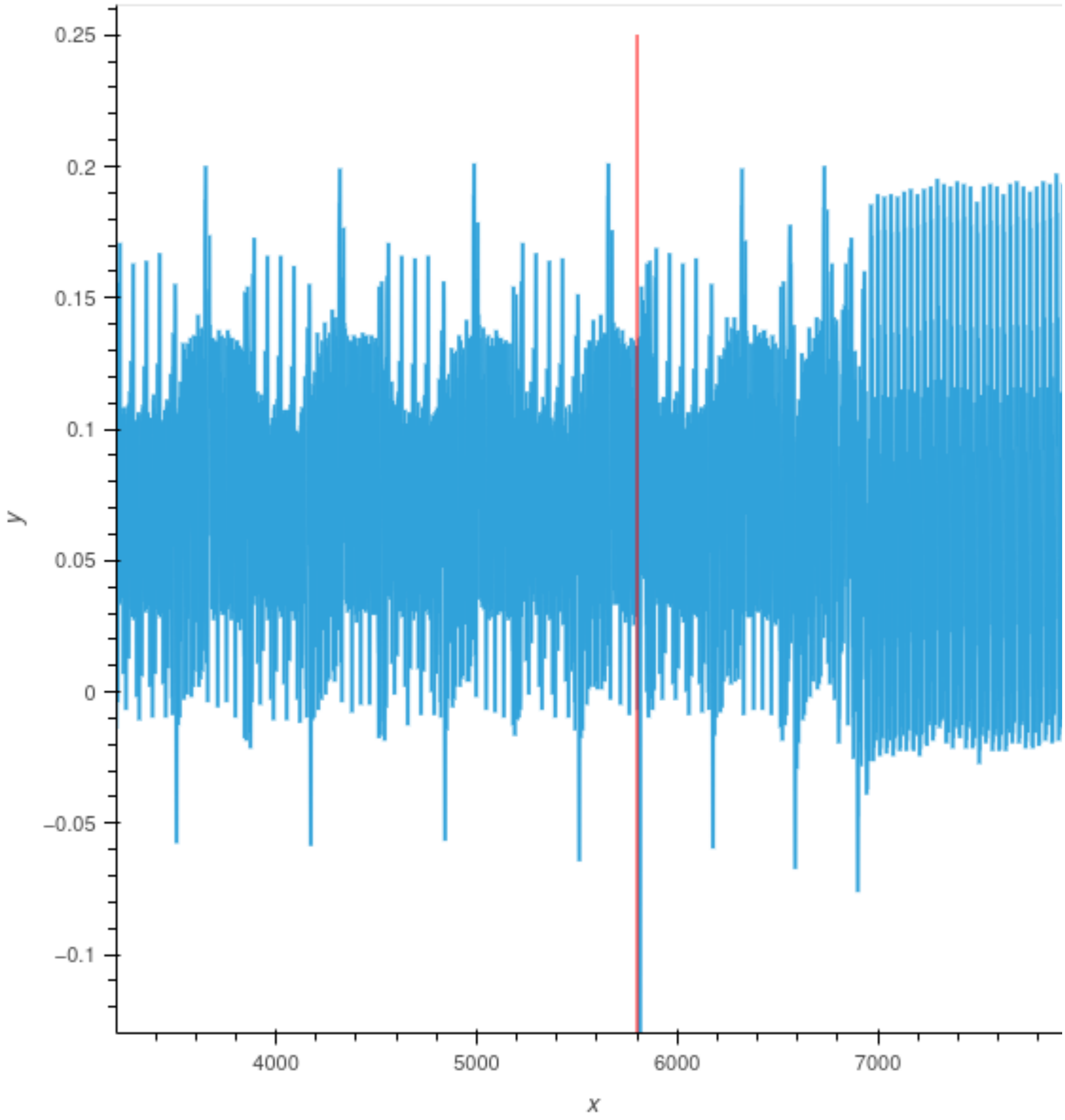
In [21]:
```
scope.io.hs2 = "glitch"
scope.glitch.clk_src = "clkgen"
scope.glitch.width = -10.15625 + 1
scope.glitch.offset = -13.84
scope.glitch.ext_offset = 5800 #5400
scope.glitch.repeat = 3
scope.glitch.trigger_src='ext_single'

# reset target
reset_target(scope)
time.sleep(0.1)

trace = cw.capture_trace(scope, target, text, key)
```

In [22]:
```
curve = hv.Curve(trace.wave)
curve *= hv.Path([(scope.glitch.ext_offset, 0.25), (scope.glitch.ext_offs
curve.opts(width=600, height=600)
```

Out[22]:




You should see a glitch in the power trace (blue) when the clock was glitched (red dotted line).

## Campaign setup

In [23]:
```
# get control over logging in order to be able to mask target execution e
# which can easily happen when glitching the target!
import logging
import chipwhisperer.common.results.glitch as glitch

# can
gc = glitch.GlitchController(groups=["column0", "column1", "column2", "co
```

The next cell defines the glitches campaign.

In [24]:
```
#check if a single column is glitched
def check_column_glitch(glitched_ct, gold_ct, column):
    column_lookup = [[0, 13, 10, 7], [4, 1, 14, 11], [8, 5, 2, 15], [12,
    for byte in column_lookup[column]:
        if glitched_ct[byte] == gold_ct[byte]:
            return False
```

```
    return True

outputs = []
results = []
obf = []
def campaign():
    # Initial glitch parameters
    global outputs
    global results
    global obf

    #reset results arrays
    outputs = []
    results = results = [['target output', 'width', 'offset', 'extoffset'
    obf = []

    #glitch setup
    scope.io.hs2 = "glitch"
    scope.glitch.clk_src = 'clkgen'
    scope.glitch.trigger_src = 'ext_single'
    key, text = ktp.next()

    #make sure correct key is loaded on target
    reboot_flush()
    target.simpleserial_write('k', key)
    target.simpleserial_wait_ack()

    for glitch_setting in gc.glitch_values():
        # set glitch settings
        scope.glitch.offset = glitch_setting[1]
        scope.glitch.width = glitch_setting[0]
        scope.glitch.ext_offset = glitch_setting[2]

        #do glitch
        target.flush()
        key, text = ktp.next()
        logging.getLogger().setLevel(logging.ERROR)

        scope.arm()
        target.simpleserial_write('p', text)

        ret = scope.capture()
        if ret:
            print("timeout!")
            reboot_flush()
            target.simpleserial_write('k', key)
            target.simpleserial_wait_ack()
            continue

        #record output
        output = target.simpleserial_read_witherrors('r', 16, timeout=100

        #handle invalid output
        if not output['valid']:
            gc.add("reset", (scope.glitch.width, scope.glitch.offset, sco
            reboot_flush()
            target.simpleserial_write('k', key)
            target.simpleserial_wait_ack()
            continue
```

```
        data = [bytes(output['payload']).hex(), scope.glitch.width, scope

        #normal output
        if output['payload'] == goldciph:
            gc.add("normal", (scope.glitch.width, scope.glitch.offset, sc
            data.append(None)
            results.append(data)
            continue

        outputs.append(output['payload'])

        #check for a glitch in each column of AES
        column_glitches = []
        for column in range(4):
            if check_column_glitch(output['payload'], goldciph, column):
                column_glitches.append(column)

        #We're looking for single column glitches here
        if len(column_glitches) == 1:
            gc.add("column{}".format(column_glitches[0]), (scope.glitch.w
            obf.append(output['payload'])
            data.append(column_glitches[0])
        else:
            gc.add("other", (scope.glitch.width, scope.glitch.offset, sco
            data.append(0xFF)

        #for display in ascii table
        results.append(data)
```

# Attacking the 9th round

## R9: Collecting faulty outputs

In [25]:

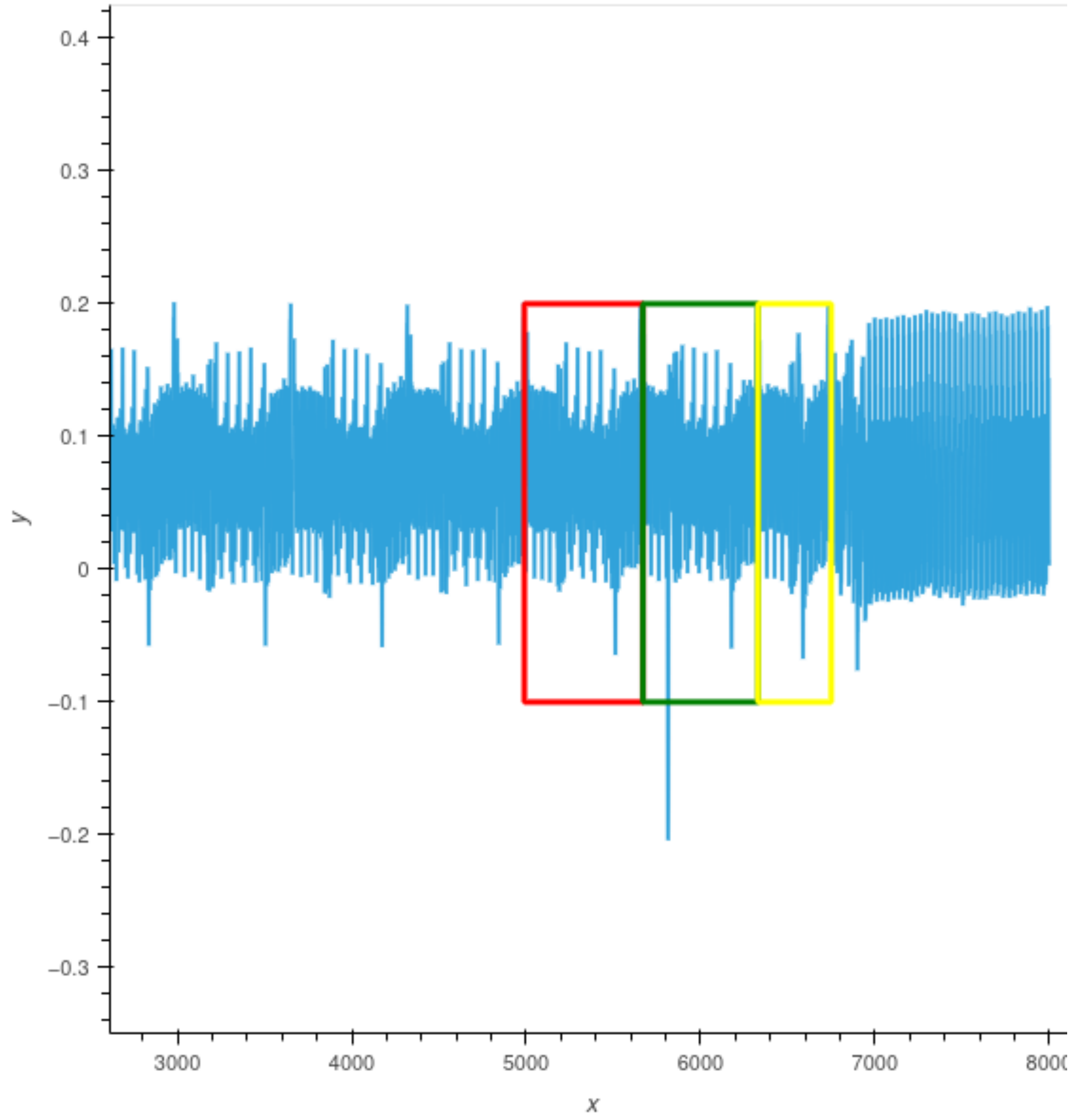
```
import holoviews as hv
from holoviews import opts
hv.extension('bokeh')
curve = hv.Curve(trace.wave).opts(width=600, height=600)

# add boxes around last rounds

line = hv.Path([(4990, -0.1), (4990, 0.2), (5670, 0.2), (5670, -0.1), (49
            hv.Path([(5670, -0.1), (5670, 0.2), (6330, 0.2), (6330, -0.1)
            hv.Path([(6330, -0.1), (6330, 0.2), (6750, 0.2), (6750, -0.1)


(curve * line).opts(opts.Path(line_width=3)).opts(width=600, height=600)
```

Out[25]:




In [26]:

```
# Parameters to make fault before MxColumn R9
gc.set_range("width", 3, 4)
gc.set_range("offset", 10, 10)
gc.set_range("ext_offset", 5700, 5900)
gc.set_global_step(0.5)
scope.glitch.repeat = 1
```

Execute the campaign

In [30]:

```
gc.display_stats()
campaign()
```

```
IntText(value=0, description='column0 count:', disabled=True)
IntText(value=0, description='column1 count:', disabled=True)
IntText(value=0, description='column2 count:', disabled=True)
IntText(value=0, description='column3 count:', disabled=True)
IntText(value=0, description='other count:', disabled=True)
IntText(value=0, description='reset count:', disabled=True)
IntText(value=0, description='normal count:', disabled=True)
FloatSlider(value=3.0, continuous_update=False, description='width setting
:', disabled=True, max=4.0, min=3.0,…
FloatSlider(value=10.0, continuous_update=False, description='offset setti
ng:', disabled=True, max=10.0, min=1…
```

```
FloatSlider(value=5700.0, continuous_update=False, description='ext_offset
setting:', disabled=True, max=5900.…
```

## R9: Cryptanalysis of the faulty outputs

We'll use `phoenixAES` to perform the DFA against the collected ciphertexts.

In [31]:

```
import phoenixAES
r9=phoenixAES.crack_bytes(outputs, goldciph, encrypt=True, verbose=2)
```

```
50fe67c3996d8eb6dae737e96cafec60: group 3
50fe67c3996d8eb6dae737e96cafec60: group 3
50fe67f3996d6fb6da9837e9d2afec60: group 3
Round key bytes recovered:
......A8....25....3F....B6......
50fe67f3996d6fb6da9837e9d2afec60: group 3
b3fe6784996d1e4ada35c3e9db48ec60: group None
b3fe6784996d1e4ada35c3e9db48ec60: group None
50b267cc116d32b6da0937d39bafd760: group 1
69ea67cc1e6d3253da094b5d9b8d3460: group None
69ea67cc1e6d3253da094b5d9b8d3460: group None
50fe5dcc997632b6d50937e99bafec82: group 2
50fe6712996de5b6dab337e9f8afec60: group 3
50fe6712996de5b6dab337e9f8afec60: group 3
38fe67cc996d3287da0942e99b86ec60: group 0
38fe67cc996d3287da0942e99b86ec60: group 0
72fede52995bb4e2e6b70de91cc3ece8: group None
72fede52995bb4e2e6b70de91cc3ece8: group None
503767cc106d32b6da0937889bafbb60: group 1
Round key bytes recovered:
..14..A8C9..25....3F..C8B6..0C..
503767cc106d32b6da0937889bafbb60: group 1
50e267ccb56d32b6da0937fa9baffc60: group 1
50e267ccb56d32b6da0937fa9baffc60: group 1
50fe67ef996d76b6dac637e91fafec60: group 3
50fe67ef996d76b6dac637e91fafec60: group 3
50fe042e997633b666c137e903afec1e: group None
50fe042e997633b666c137e903afec1e: group None
6144068813d9ea8d4c4cf43f11a46136: group None
6144068813d9ea8d4c4cf43f11a46136: group None
1cfe10b6994bbd8eb81783e94a00ecab: group None
1cfe10b6994bbd8eb81783e94a00ecab: group None
50806701a36d84b6dac6374cceafe460: group None
50886701886d84b6dac63717ceafd060: group None
50fe75cc997b32b6e70937e99bafec54: group 2
Round key bytes recovered:
..14F9A8C9EE25..E13F..C8B6..0CA6
0efe67cc996d32b6da0937e99bafec60: group None
46fe67cc996d32b6da0937e99bafec60: group None
46fe67cc996d32b6da0937e99bafec60: group None
46fe67cc996d323eda0925e99bbbec60: group 0
Round key bytes recovered:
D014F9A8C9EE2589E13F0CC8B6630CA6
Last round key #N found:
D014F9A8C9EE2589E13F0CC8B6630CA6
```

Once the **last round key is recovered**, you can **revert the AES keyscheduling** and reveal the actual AES

In [36]:
```
# Print the master key
from chipwhisperer.analyzer.attacks.models.aes.key_schedule import key_sc
print("Found master key:", end="")
r9_key = key_schedule_rounds(bytearray.fromhex(r9), 10, 0)
print(''.join("%02x" % x for x in r9_key))
print(f"Master key: {master_key}")
```

```
Found master key:2b7e151628aed2a6abf7158809cf4f3c
Master key: 2b7e151628aed2a6abf7158809cf4f3c
```

# Attacking the 8th round

## R8: Collecting faulty outputs

Focus faults before MixColumn R8

In [37]:
```
#Change parameters
gc.set_range("ext_offset", 5050, 5250)
gc.display_stats()
campaign()
```

```
IntText(value=0, description='column0 count:', disabled=True)
IntText(value=0, description='column1 count:', disabled=True)
IntText(value=0, description='column2 count:', disabled=True)
IntText(value=0, description='column3 count:', disabled=True)
IntText(value=0, description='other count:', disabled=True)
IntText(value=0, description='reset count:', disabled=True)
IntText(value=0, description='normal count:', disabled=True)
FloatSlider(value=3.0, continuous_update=False, description='width setting
:', disabled=True, max=4.0, min=3.0,…
FloatSlider(value=10.0, continuous_update=False, description='offset setti
ng:', disabled=True, max=10.0, min=1…
FloatSlider(value=5050.0, continuous_update=False, description='ext_offset
setting:', disabled=True, max=5250.…
```

## Piret’s attack R-8

In this second attack, we assume the fault was injected *before* the last two *MixColumn* operations.

In [39]:
```
outputs2=phoenixAES.convert_r8faults_bytes(outputs, goldciph, encrypt=Tru
r8=phoenixAES.crack_bytes(outputs2, goldciph, encrypt=True, verbose=2)
```

```
2cfe67cc996d32eada095ce99b5fec60: group 0
50b367ccf96d32b6da0937109baf4360: group 1
50fe03cc994f32b6780937e99bafecc3: group 2
50fe67e2996dceb6dabb37e953afec60: group 3
7cfe67cc996d3219da097ce99bfeec60: group 0
505f67ccd86d32b6da0937f49bafa960: group 1
50fe77cc996f32b6d90937e99bafec0e: group 2
50fe6742996d97b6da5937e9a4afec60: group 3
61fe67cc996d3240da0944e99b1bec60: group 0
50a067cc286d32b6da0937f29baf1a60: group 1
50fe98cc994232b6d80937e99bafec6a: group 2
50fe6787996d40b6da0537e956afec60: group 3
61fe67cc996d3240da0944e99b1bec60: group 0
50a067cc286d32b6da0937f29baf1a60: group 1
50fe98cc994232b6d80937e99bafec6a: group 2
50fe6787996d40b6da0537e956afec60: group 3
37fe67cc996d32f0da0956e99b59ec60: group 0
501f67cc3f6d32b6da0937a19baf7960: group 1
50fec5cc99e332b6e40937e99bafecc1: group 2
50fe6722996d60b6dafb37e907afec60: group 3
19fe67cc996d3214da09cbe99b15ec60: group 0
Round key bytes recovered:
D0............89....0C....63....
50ae67cca76d32b6da0937c89baf8560: group 1
Round key bytes recovered:
D014....C9....89....0CC8..630C..
50fe64cc994532b6700937e99bafec9a: group 2
Round key bytes recovered:
D014F9..C9EE..89E1..0CC8..630CA6
50fe67fc996de7b6da1437e9ffafec60: group 3
Round key bytes recovered:
D014F9A8C9EE2589E13F0CC8B6630CA6
Last round key #N found:
D014F9A8C9EE2589E13F0CC8B6630CA6
```

Once the **last round key is recovered**, you can **revert the AES keyscheduling** and reveal the actual AES

In [40]:
```python
print("Found master key:", end="")
key = key_schedule_rounds(bytearray.fromhex(r8), 10, 0)
print(''.join("%02x" % x for x in key))
print(f"Master key: {master_key}")
```

```
Found master key:2b7e151628aed2a6abf7158809cf4f3c
Master key: 2b7e151628aed2a6abf7158809cf4f3c
```

# The end

Once you're done, clean up the connection to the scope and target.

In [41]:
```python
scope.dis()
target.dis()
```

In [ ]:
```python

```