In [1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

CS529-HWS: Hardware Security - Lab 3

Aneesh Kandi | [kandia@purdue.edu]

Connecting to the ChipWhisperer Nano

In [ ]:
```
SCOPETYPE = 'CWNANO'
PLATFORM = 'CWNANO'
CRYPTO_TARGET = 'TINYAES128C'
SS_VER='SS_VER_1_1'
```

The following code will build the firmware for the target.

In [ ]:
```
%run "../../Setup_Scripts/Setup_Generic.ipynb"
```

```
INFO: Found ChipWhisperer😍
```

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

```
Building for platform CWNANO 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
arm-none-eabi-gcc (15:10.3-2021.07-4) 10.3.1 20210621 (release)
Copyright (C) 2020 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-CWNANO
.
Welcome to another exciting ChipWhisperer target build!!
.
.
.
Compiling:
Compiling:
.
Compiling:
-en     simpleserial-aes.c ...
-en     .././simpleserial/simpleserial.c ...
-en     .././hal/hal.c ...
Compiling:
-en     .././hal//stm32f0_nano/stm32f0_hal_nano.c ...
.
.
Compiling:
Compiling:
-en     .././hal//stm32f0/stm32f0_hal_lowlevel.c ...
.
-en     .././crypto/tiny-AES128-C/aes.c ...
Compiling:
-en     .././crypto/aes-independant.c ...
.
Assembling: .././hal//stm32f0/stm32f0_startup.S
arm-none-eabi-gcc -c -mcpu=cortex-m0 -I. -x assembler-with-cpp -mthumb -mf
loat-abi=soft -ffunction-sections -DF_CPU=7372800 -Wa,-gstabs,-adhlns=objd
ir-CWNANO/stm32f0_startup.lst -I.././simpleserial/ -I.././hal/ -I.././hal/
-I.././hal//stm32f0 -I.././hal//stm32f0/CMSIS -I.././hal//stm32f0/CMSIS/co
re -I.././hal//stm32f0/CMSIS/device -I.././hal//stm32f0/Legacy -I.././simp
leserial/ -I.././crypto/ -I.././crypto/tiny-AES128-C .././hal//stm32f0/stm
32f0_startup.S -o objdir-CWNANO/stm32f0_startup.o
-e Done!
-e Done!
-e Done!
-e Done!
-e Done!
-e Done!
-e Done!
.
LINKING:
-en     simpleserial-aes-CWNANO.elf ...
-e Done!
.
.
.
Creating load file for Flash: simpleserial-aes-CWNANO.hex
.
arm-none-eabi-objcopy -O ihex -R .eeprom -R .fuse -R .lock -R .signature s
```

```
impleserial-aes-CWNANO.elf simpleserial-aes-CWNANO.hex
Creating load file for Flash: simpleserial-aes-CWNANO.bin
Creating load file for EEPROM: simpleserial-aes-CWNANO.eep
arm-none-eabi-objcopy -O binary -R .eeprom -R .fuse -R .lock -R .signature
simpleserial-aes-CWNANO.elf simpleserial-aes-CWNANO.bin
Creating Extended Listing: simpleserial-aes-CWNANO.lss
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-CWNANO.elf simpleserial-aes-CWNANO.eep || exit 0
arm-none-eabi-objdump -h -S -z simpleserial-aes-CWNANO.elf > simpleserial-
aes-CWNANO.lss
.
Creating Symbol Table: simpleserial-aes-CWNANO.sym
arm-none-eabi-nm -n simpleserial-aes-CWNANO.elf > simpleserial-aes-CWNANO.
sym
Size after:
   text    data     bss     dec     hex filename
   5648     540    1884    8072    1f88 simpleserial-aes-CWNANO.elf
+--------------------------------------------------------
+ Default target does full rebuild each time.
+ Specify buildtarget == allquick == to avoid full rebuild
+--------------------------------------------------------
+--------------------------------------------------------
+ Built for platform CWNANO Built-in Target (STM32F030) with:
+ CRYPTO_TARGET = TINYAES128C
+ CRYPTO_OPTIONS = AES128C
+--------------------------------------------------------
```

In [ ]:
```
cw.program_target(scope, prog, "../../../firmware/mcu/simpleserial-aes/si
```

```
Detected known STMF32: STM32F03xx4/03xx6
Extended erase (0x44), this can take ten seconds or more
Attempting to program 6187 bytes at 0x8000000
STM32F Programming flash...
STM32F Reading flash...
Verified flash OK, 6187 bytes
```

Capturing Traces

In [ ]:
```
from tqdm.notebook import trange
import numpy as np
import time

ktp = cw.ktp.Basic()

trace_counts = [1000, 2000, 2500, 5000]
captured_traces = []

for N in trace_counts:
    trace_array = []
    textin_array = []

    key, text = ktp.next()

    target.set_key(key)

    for i in trange(N, desc='Capturing traces'):
        scope.arm()
```

```
        target.simpleserial_write('p', text)

        ret = scope.capture()
        if ret:
            print("Target timed out!")
            continue

        response = target.simpleserial_read('r', 16)

        trace_array.append(scope.get_last_trace())
        textin_array.append(text)

        key, text = ktp.next()

    captured_traces.append([trace_array, textin_array])
```

```
Capturing traces:   0%|          | 0/1000 [00:00<?, ?it/s]
Capturing traces:   0%|          | 0/2000 [00:00<?, ?it/s]
Capturing traces:   0%|          | 0/2500 [00:00<?, ?it/s]
Capturing traces:   0%|          | 0/5000 [00:00<?, ?it/s]
```

Theoretical AES Implementation

In [ ]:

```
sbox = [
    # 0    1    2    3    4    5    6    7    8    9    a    b    c    d
    0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7
    0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4
    0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8
    0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27
    0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3
    0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c
    0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c
    0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff
    0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d
    0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e
    0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95
    0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a
    0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd
    0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1
    0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55
    0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54
]

def aes_internal(inputdata, key):
    return sbox[inputdata ^ key]
```

DPA Implementation

In [ ]:

```
def calculate_diffs(guess, byteindex, bitnum, textin_array, trace_array):

    one_list = []
    zero_list = []
    numtraces = np.shape(trace_array)[0] #total number of traces
    numpoints = np.shape(trace_array)[1] #samples per trace

    for trace_index in range(numtraces):
```

```
        hypothetical_leakage = aes_internal(guess, textin_array[trace_ind

        #Mask off the requested bit
        if hypothetical_leakage & (1<<bitnum):
            one_list.append(trace_array[trace_index])
        else:
            zero_list.append(trace_array[trace_index])

    one_avg = np.asarray(one_list).mean(axis=0)
    zero_avg = np.asarray(zero_list).mean(axis=0)
    return abs(one_avg - zero_avg)
```

In [ ]:

```
from tqdm.notebook import tnrange
import numpy as np

# Known key for validation
known_key = [0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
             0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c]

# Store results
key_guesses = [[0]*8 for _ in range(16)]  # best guess per bitnum
bit_rankings = [[0]*8 for _ in range(16)] # rank of correct key for each
best_bits = [0]*16  # best bitnum per subkey

for i in range(len(trace_counts)):
    print(f"\n--- Running DPA with {trace_counts[i]} traces ---")
    trace_array, textin_array = captured_traces[i]

    for subkey in tnrange(0, 4, desc=f"Subkey"):
        best_bit = 0
        best_diff = -1

        for bitnum in range(8):
            max_diffs = [0]*256

            for guess in range(256):
                full_diff_trace = calculate_diffs(guess, subkey, bitnum,
                full_diff_trace = full_diff_trace[(1100 + subkey*60):]
                max_diffs[guess] = np.max(full_diff_trace)

            # Sorted guesses (highest to lowest diff)
            sorted_args = np.argsort(max_diffs)[::-1]
            top_guess = sorted_args[0]
            rank_of_correct = list(sorted_args).index(known_key[subkey])
            correct_diff = max_diffs[known_key[subkey]]

            key_guesses[subkey][bitnum] = top_guess
            bit_rankings[subkey][bitnum] = rank_of_correct

            if correct_diff > best_diff:
                best_diff = correct_diff
                best_bit = bitnum

        best_bits[subkey] = best_bit
        print(f"Subkey {subkey:02d}: Best Bit = {best_bit}, Rank of corre
```

```
--- Running DPA with 1000 traces ---
Subkey:   0%|          | 0/4 [00:00<?, ?it/s]
```

```
Subkey 00: Best Bit = 7, Rank of correct key at that bit = 12
Subkey 01: Best Bit = 4, Rank of correct key at that bit = 0
Subkey 02: Best Bit = 2, Rank of correct key at that bit = 37
Subkey 03: Best Bit = 1, Rank of correct key at that bit = 17

--- Running DPA with 2000 traces ---
Subkey:   0%|          | 0/4 [00:00<?, ?it/s]
Subkey 00: Best Bit = 5, Rank of correct key at that bit = 9
Subkey 01: Best Bit = 4, Rank of correct key at that bit = 0
Subkey 02: Best Bit = 1, Rank of correct key at that bit = 31
Subkey 03: Best Bit = 7, Rank of correct key at that bit = 13

--- Running DPA with 2500 traces ---
Subkey:   0%|          | 0/4 [00:00<?, ?it/s]
Subkey 00: Best Bit = 1, Rank of correct key at that bit = 8
Subkey 01: Best Bit = 4, Rank of correct key at that bit = 0
Subkey 02: Best Bit = 7, Rank of correct key at that bit = 50
Subkey 03: Best Bit = 3, Rank of correct key at that bit = 88

--- Running DPA with 5000 traces ---
Subkey:   0%|          | 0/4 [00:00<?, ?it/s]
Subkey 00: Best Bit = 5, Rank of correct key at that bit = 5
Subkey 01: Best Bit = 3, Rank of correct key at that bit = 0
Subkey 02: Best Bit = 6, Rank of correct key at that bit = 50
Subkey 03: Best Bit = 0, Rank of correct key at that bit = 2
```

In [ ]:

```
trace_array, textin_array = captured_traces[-1]
for subkey in tnrange(0, 16, desc=f"Subkey"):
    best_bit = 0
    best_diff = -1

    for bitnum in range(8):
        max_diffs = [0]*256

        for guess in range(256):
            full_diff_trace = calculate_diffs(guess, subkey, bitnum, text
            full_diff_trace = full_diff_trace[(1100 + subkey*60):]
            max_diffs[guess] = np.max(full_diff_trace)

        # Sorted guesses (highest to lowest diff)
        sorted_args = np.argsort(max_diffs)[::-1]
        top_guess = sorted_args[0]
        rank_of_correct = list(sorted_args).index(known_key[subkey])
        correct_diff = max_diffs[known_key[subkey]]

        key_guesses[subkey][bitnum] = top_guess
        bit_rankings[subkey][bitnum] = rank_of_correct

        if correct_diff > best_diff:
            best_diff = correct_diff
            best_bit = bitnum

    best_bits[subkey] = best_bit
    print(f"Subkey {subkey:02d}: Best Bit = {best_bit}, Rank of correct k
```

```
Subkey:   0%|          | 0/16 [00:00<?, ?it/s]
```

```
Subkey 00: Best Bit = 5, Rank of correct key at that bit = 5
Subkey 01: Best Bit = 3, Rank of correct key at that bit = 0
Subkey 02: Best Bit = 6, Rank of correct key at that bit = 50
Subkey 03: Best Bit = 0, Rank of correct key at that bit = 2
Subkey 04: Best Bit = 1, Rank of correct key at that bit = 57
Subkey 05: Best Bit = 7, Rank of correct key at that bit = 72
Subkey 06: Best Bit = 2, Rank of correct key at that bit = 9
Subkey 07: Best Bit = 4, Rank of correct key at that bit = 19
Subkey 08: Best Bit = 2, Rank of correct key at that bit = 29
Subkey 09: Best Bit = 6, Rank of correct key at that bit = 166
Subkey 10: Best Bit = 5, Rank of correct key at that bit = 12
Subkey 11: Best Bit = 4, Rank of correct key at that bit = 84
Subkey 12: Best Bit = 6, Rank of correct key at that bit = 14
Subkey 13: Best Bit = 1, Rank of correct key at that bit = 20
Subkey 14: Best Bit = 2, Rank of correct key at that bit = 23
Subkey 15: Best Bit = 6, Rank of correct key at that bit = 7
```

Questions from Lab 3

What about using other bits in the output as the rule to partition power traces, e.g., bit-X?

Using other bits (bit 0 to bit 7) in the intermediate value (e.g., S-box output) to partition traces is a valid and often useful strategy. This generalizes the traditional DPA approach, which often focuses only on bit-0. Each bit of the S-box output leaks differently based on circuit design, operation timing, and power measurement noise. Some bits may leak more reliably depending on how the microcontroller processes those bits internally.

For the recovery of each subkey (byte, or round key), what would be the best bit-X, e.g., yielding the most significant differences compared to other guesses?

The above output shows that the best bit is different for each subkey. I ran for only 4 subkeys because it was taking a huge amount of time for 16.

Does it exist the best bit-X that is simply the best for each subkey recovery? No, there is usually not a single global best bit for all subkeys which is clearly evident from the above results. This is due to how AES processes data and how leakage manifests across instructions and cycles. You can determine this by analyzing the distribution of `best_bits[subkey]`:

- If one bit dominates (e.g., bit 0 is best for most subkeys), then it's a good default.
- If the best bit is spread out evenly, per-subkey selection is better.

If bit-0 is not the best global partition policy, how would you choose your partition policy?

We can adaptively select the bit per subkey by running DPA on all bits for every subkey. Then measure the difference peak. Select the bit-X with the highest leakage as the partition rule for that subkey.

Explore all the questions above with the number of traces in 1K, 2K, 2.5K (default), and 5K.

Based on the results of running DPA with varying numbers of traces, we observed the following trends:

- As the number of traces increases, the DPA attack becomes more effective. The correct key bytes are more likely to be ranked at the top.
- The ranking of the correct key byte improves as the trace count (N) increases. Lower rank values indicate higher confidence in the guessed key byte.
- While Bit-0 is commonly used in DPA, it is not always the optimal choice. At lower trace counts, certain bits—such as Bit-0 or Bit-7—may exhibit stronger leakage, but this varies between subkeys.
- **At Low Trace Counts (e.g., 1,000 traces):**
  - Greater variability is observed in which bit provides the best results per subkey.
  - The correct key may not rank highly, leading to inaccurate or failed key recovery for some subkeys.
- **At Higher Trace Counts (e.g., 5,000 traces):**
  - The choice of best-leaking bit for each subkey becomes more consistent.
  - The correct key is more reliably identified, with improved ranking across all subkeys.

Summary Table

Traces	Subkey	Best Bit	Rank of Correct Key
1000	00	7	12
1000	01	4	0
1000	02	2	37
1000	03	1	17
2000	00	5	9
2000	01	4	0
2000	02	1	31
2000	03	7	13
2500	00	1	8
2500	01	4	0
2500	02	7	50
2500	03	3	88
5000	00	5	5
5000	01	3	0
5000	02	6	50
5000	03	0	2