2022 Embedded Capture-the-Flag (eCTF) Competition

Side-Channel Emulator

Riverside Research

A side-channel (SC) emulator allows for the collection of power measurements from the virtualized hardware used in the eCTF competition. Typically, teams would collect real power, electromagnetic (EM), timing, or other SC measurements from the physical integrated circuit as it boots and performs its required tasks. The SC emulator collects waveforms derived from the operations and data processed by an emulated target device running another eCTF team's secured bootloader design. You can try using these measurements to perform side-channel analysis (SCA) attacks on cryptographic operations, passwords, and other security features.

This tutorial will teach you:

- How to collect traces from the SC emulator,
- How to import emulated SC data into Python and perform basic analysis, and
- Practical considerations for performing SCA during the eCTF competition.

This tutorial also includes an SC Emulator challenge with three tasks that can be completed to retrieve flags and score points for your team!

The SC challenge scenario emulates a microcontroller using the advanced encryption standard (AES) algorithm [1] to encrypt a binary file, shown in ***Figure 1***. Teams will configure the SC receiver to collect emulated power measurements as blocks of 16 bytes of data are encrypted by the AES-128 engine on the microcontroller. The side-channel measurements and input data will be used to perform a three-phased attack and recover the AES encryption key.

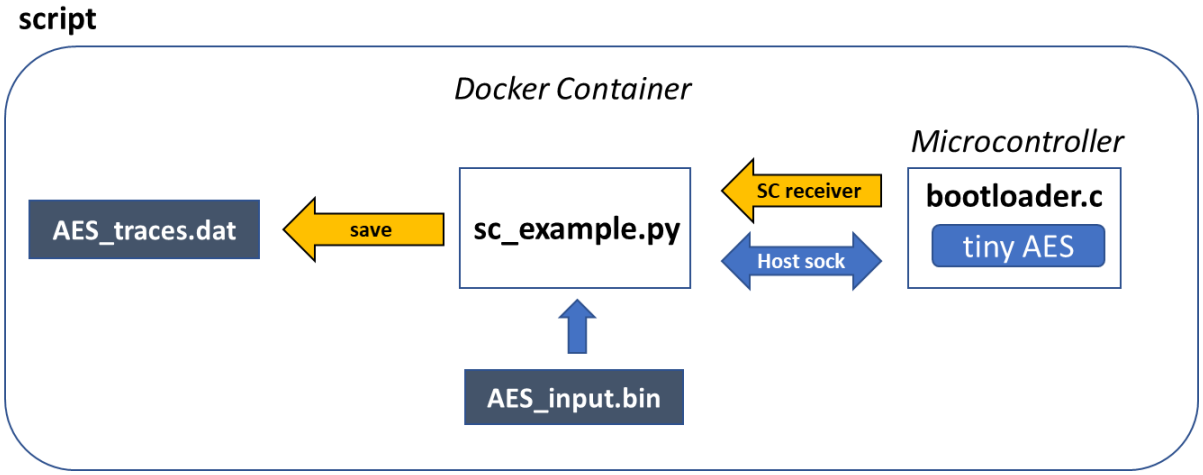


Figure 1: This challenge emulates a side-channel attack on AES-128 encryption.

What is Side-Channel Analysis?

Side-Channel Analysis (SCA) is a technique where unintended emissions from a microelectronic device are used to gain knowledge of privileged or sensitive information. SCA has been used to defeat encryption [2], steal passwords or personal identification numbers [3], and read the contents of memory using the highly publicized SPECTRE and MELTDOWN attacks [4]. The class of SCA attacks known as differential power analysis (DPA) is possible due to differences in the power required to set a binary data bit to a value of one versus zero in a digital circuit [5, 6]. An attacker exploits this imbalance by recording the power consumption from a device as it encrypts or decrypts sensitive information. Measurements will have higher power consumption if more bits are set to one and lower power consumption if more bits are set to zero.

Side-channel measurements are collected when a known input or output, such as the encrypted ciphertext, is combined with portions of the cryptographic key. An example of where key-dependent side-channel leakage can occur in an AES engine is shown in Figure 2. The leaked power signal is dependent on the key since the data value leaked after the Sbox will change based on the exclusive-or (XOR) between the *known text input* and the *fixed unknown key*. Different input and key combinations result in different number of bits set to one (higher power consumption) or bits set to zero (lower power consumption).

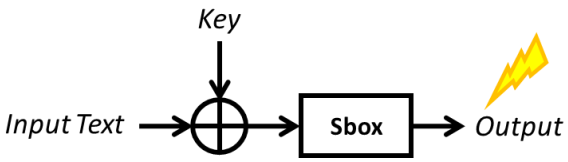


Figure 2 – An example of key dependent leakage from the first round of AES encryption.

SCA measurements are collected from voltage supply lines, in-line power resistors, electromagnetic (EM) emissions, and timing information. Further discussion on collecting SCA measurements from embedded devices can be found in published research articles [3]. In an SCA attack, a series of measurements are often collected with a fixed unknown key and varying known cipher inputs. An example of a power measurement from a cryptographic implementation is shown in ***Figure 3***.

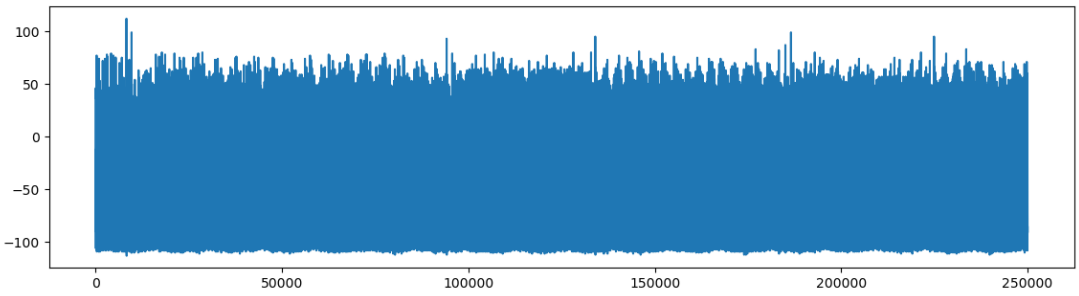


Figure 3: A power measurement from a microcontroller with peaks indicating differences in code execution [7].

An attacker recovers each byte of the AES key separately in a divide-and-conquer attack until the entire key is recovered. For example, SCA can exploit algorithms that were architected to operate on 8-bit microcontrollers, and hence perform operations on subsets of the 128-bit key. AES includes operations that use 8 bits of the key at a time, such as the Sbox operation shown in Figure 2, that allow the attacker to estimate the power consumption for a single byte of the key (2^8 or 256 possible key values). Without isolating subsets of the key, the attacker would have to contend with all 2^{128} possible key values for AES-128, which is not computationally feasible.

The analysis portion of the attack involves the adversary making guesses on the power consumption for all possible values of a single key byte. Statistical techniques are used to check the estimated power for each possible key guess against the collected measurements. The correct key results in the closest match between estimated and measured power, often identified as the maximum statistical test score or correlation value as show in Figure 4. The attack is repeated for each key byte until all 16 bytes (128 bits) of the key are recovered.

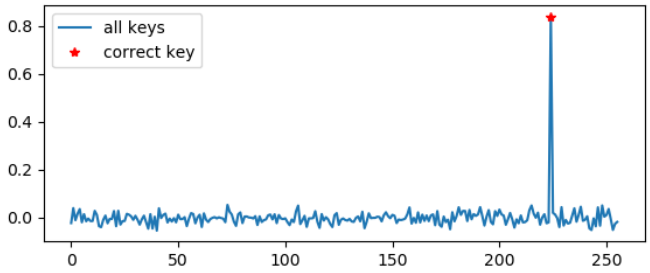


Figure 4 – A successful side-channel attack on AES recovers the correct key, which is indicated by the highest correlation score (y-axis) of all 256 key hypotheses (x-axis). The attack is repeated to recover each of the 16 AES-128 key bytes.

Launching the Device with the SC Emulator Enabled

To launch a bootloader container with the SC emulator, use the *`launch-bootloader-sc`* command for *`tools/run_saffire.py`* instead of the regular *`launch-bootloader`* command.

A bootloader image has been posted to DockerHub for this challenge. To pull it on to your development server, choose a unique *`sysname`* and run the following commands from the top folder of the example repository replacing `sc-challenge` with the *`sysname`*:

```
docker pull ectf/bootloader:sc
```

```
docker image tag ectf/bootloader:sc sc-challenge/bootloader:sc
```

Then, launch the bootloader with the SC emulator enabled. This example command uses the same 1337 UART socket from the "getting started" instructions. Replace it with the UART socket port number you have been using during development:

```
python3 tools/run_saffire.py launch-bootloader-sc --emulated --sysname
sc-challenge --uart-sock 1337 --sock-root socks
```

After running this, you should see a socket called *`sc_probe.sock`* in the *`socks`* directory. This is where side-channel traces will be collected from. A complete list of instructions for running the side channel emulator can be found in the 'sc_challenge_instructions.md' markdown.

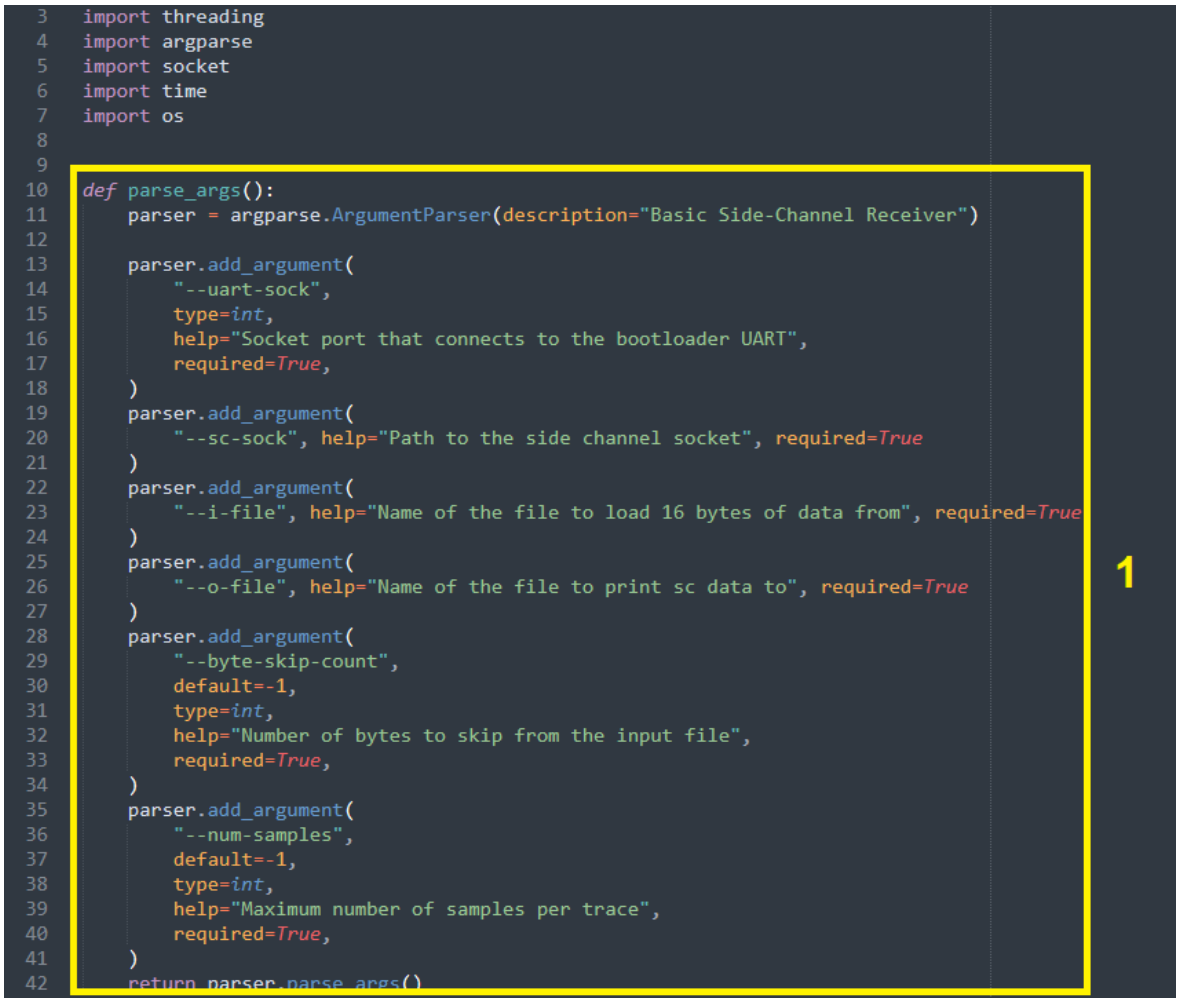
Side-Channel Trace Collector

An example side-channel collector has been provided in the `tools` folder called `sc_example.py`. This script takes the following arguments:

- *'--uart-sock'* is the bootloader UART socket to connect to.
- *'--sc-sock'* is the path to the side-channel socket (*'socks/sc_probe.sock'*).
- '*--i-file*' is a binary file that contains input data to send to the microcontroller for encryption.
- '*--o-file*' is a binary file where the collected trace data is saved.
- '*--byte_skip_count*' is the number of bytes to skip from the start of *`aes_input.bin`* before writing data over the UART socket to the microcontroller.
- '*--num_samples*' is the number of SC data samples to save to the output file. Students will need to decide how many data samples are sufficient to capture the target signal. If they capture too many data samples, then they will have extraneous samples to remove. Capture too few samples and they risk not collecting SC data while the cryptographic engine is running.

Collection in the provided script starts with the first 16 bytes of data in the input file (by skipping 0 bytes), captures 200000 points of data, and saves the data to an output file. Teams will most likely want to create their own sets of inputs stored in a similar input file and make their collection script loop and save each run to a new, unique output file.

The example SC receiver is implemented in a Python script that captures data from the collector and packages it into the saved file. Teams are free to modify sc_example.py as they wish, including adding looping, adding/removing input arguments, adding new functions, or triggering off of specific events. The example SC receiver is shown in Figures 6-8 with various major portions highlighted and numbered. Teams will need to come up with innovative ways to deal with the copious amounts of noise added into the challenge. There will be noise added both before and after every AES operation. The noise added before each AES operation will be a variable amount and will cause misalignment. Teams may want to consider downsampling or decimating traces to reduce file size, but the main consideration should be the removal of noise.



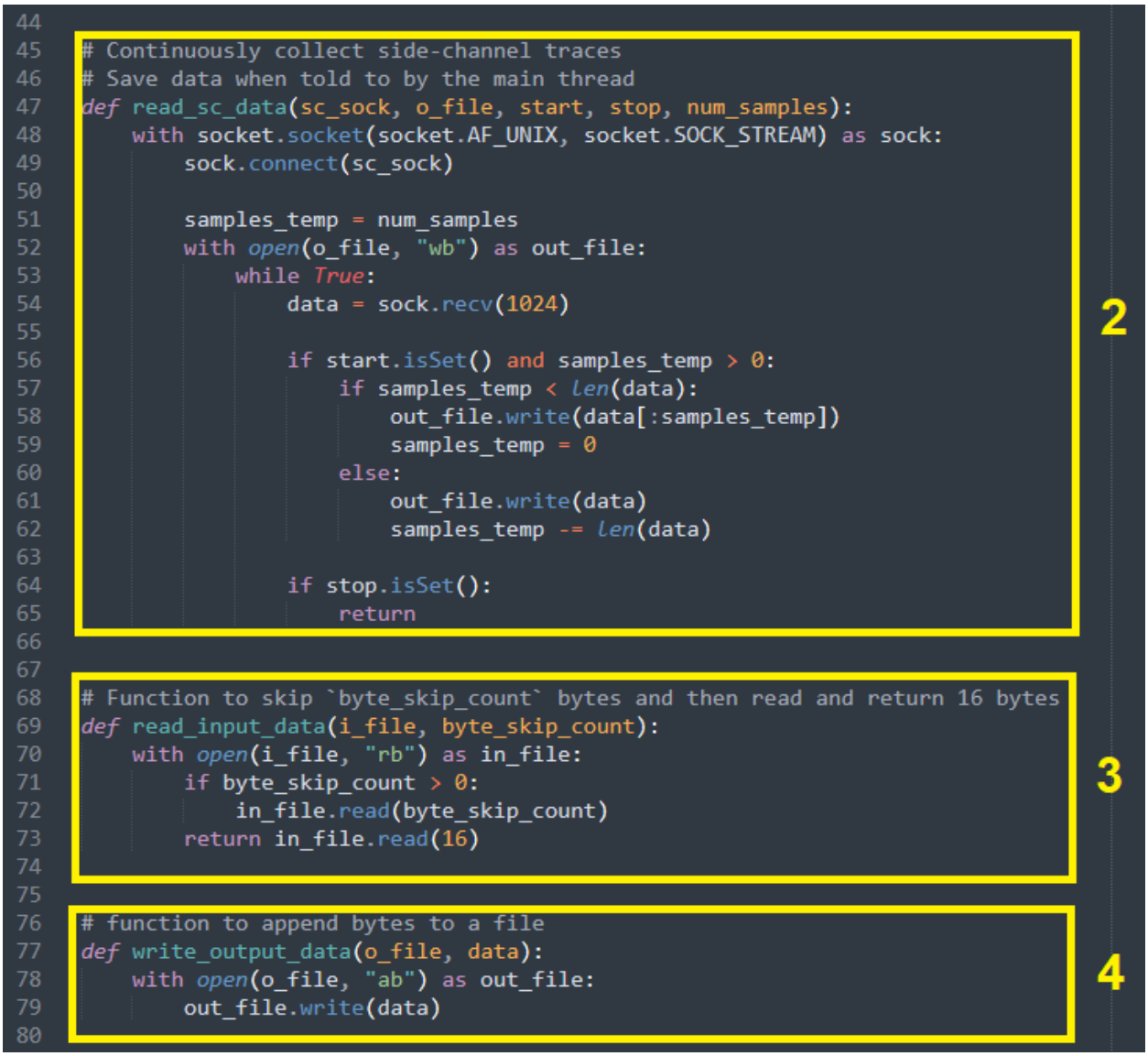

```python
import threading
import argparse
import socket
import time
import os


def parse_args():
    parser = argparse.ArgumentParser(description="Basic Side-Channel Receiver")

    parser.add_argument(
        "--uart-sock",
        type=int,
        help="Socket port that connects to the bootloader UART",
        required=True,
    )
    parser.add_argument(
        "--sc-sock", help="Path to the side channel socket", required=True
    )
    parser.add_argument(
        "--i-file", help="Name of the file to load 16 bytes of data from", required=True
    )
    parser.add_argument(
        "--o-file", help="Name of the file to print sc data to", required=True
    )
    parser.add_argument(
        "--byte-skip-count",
        default=-1,
        type=int,
        help="Number of bytes to skip from the input file",
        required=True,
    )
    parser.add_argument(
        "--num-samples",
        default=-1,
        type=int,
        help="Maximum number of samples per trace",
        required=True,
    )
    return parser.parse_args()
```

Figure 5: The SC python example argument list handles important test parameters.

1. SC input arguments. The input arguments have already been touched upon during the prior explanation of the sample script which called the SC python example. They are listed again in Figure 5. All input arguments are required; however, teams are free to edit the SC python example however they wish.

There are three main SC functions used to gather data for AES encryption and to write data to external files, including SC receiver measurements. These three functions are listed in Figure 6.

2. The first function '*read_sc_data*', is used to continuously send collected data across the open probe socket until the specified number of samples to collect by '*num_samples*' has been met.
3. The '*read_input_data*' function is used to read in 16 bytes of data from an input file and send them to the bootloader for AES encryption. This function grabs a contiguous 16-byte block of data after it has met the number of bytes to skip as specified by the input argument '*byte_skip_count*'.
4. The *'write_output_data'* function is not used in the basic example but could be used to append the collected data to the output file specified by the 'o_file' input argument.




```python
# Continuously collect side-channel traces
# Save data when told to by the main thread
def read_sc_data(sc_sock, o_file, start, stop, num_samples):
    with socket.socket(socket.AF_UNIX, socket.SOCK_STREAM) as sock:
        sock.connect(sc_sock)

        samples_temp = num_samples
        with open(o_file, "wb") as out_file:
            while True:
                data = sock.recv(1024)

                if start.isSet() and samples_temp > 0:
                    if samples_temp < len(data):
                        out_file.write(data[:samples_temp])
                        samples_temp = 0
                    else:
                        out_file.write(data)
                        samples_temp -= len(data)

                if stop.isSet():
                    return
```

```python
# Function to skip `byte_skip_count` bytes and then read and return 16 bytes
def read_input_data(i_file, byte_skip_count):
    with open(i_file, "rb") as in_file:
        if byte_skip_count > 0:
            in_file.read(byte_skip_count)
        return in_file.read(16)
```

```python
# function to append bytes to a file
def write_output_data(o_file, data):
    with open(o_file, "ab") as out_file:
        out_file.write(data)
```

Figure 6: The SC python example functions control data communications with the target device and saving test measurements to external files.

The main loop of the SC python example is shown in Figure 7.

5. First, the python example fetches 16 bytes of data from the input file.
6. Next, trace collection is started by setting start.set() as 16 bytes of data are sent across the socket to the bootloader for AES encryption.
7. After AES is finished, a message is printed across the terminal and the data collection is stopped.


```python
def main():
    args = parse_args()

    uart_sock = args.uart_sock
    sc_sock = os.path.abspath(args.sc_sock)
    num_samples = args.num_samples
    byte_skip_count = args.byte_skip_count

    start = threading.Event()
    stop = threading.Event()

    sc_thread = threading.Thread(
        target=read_sc_data,
        args=(
            sc_sock,
            args.o_file,
            start,
            stop,
            num_samples,
        ),
    )
    sc_thread.start()

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect(("0.0.0.0", uart_sock))

        print("\nGetting plaintext...")
        text = read_input_data(args.i_file, byte_skip_count)

        print("Collecting traces...")
        start.set()
        sock.send(text)

        print("Finishing")
        time.sleep(0.1)
        data = sock.recv(1)

        if data[0] != 0x6:
            print("Error. Bootloader did not respond with 0x6")

        stop.set()
```

Figure 7: The SC Python example implements a side-channel collection.

Some sample input data is shown in Figure 8 and is sent to the bootloader 16 bytes at a time for AES encryption. Teams will create their own binary input data to support their data collection efforts. The input data consists of raw byte values and should be in a binary file format.

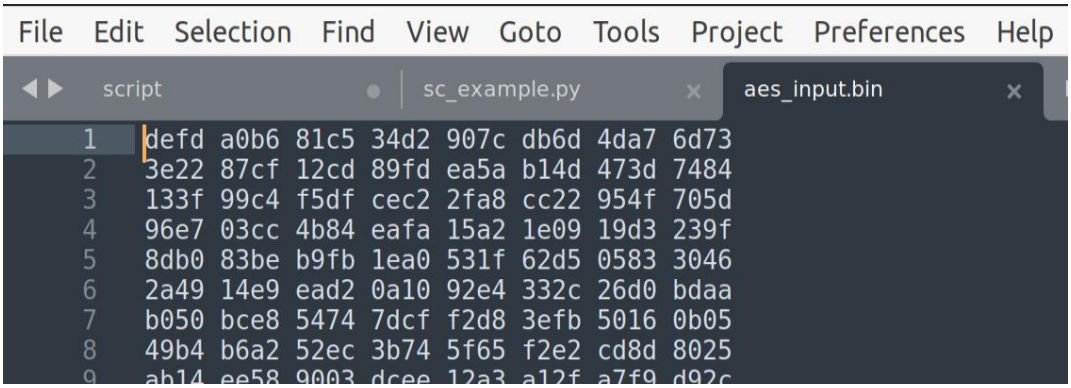


Figure 8: The sample AES input data are randomly generated byte values.

SC Collector Usage

All commands in this section assume you are running from the top folder in the reference design repo. After starting the bootloader with the side-channel emulator enabled, you will need to generate data to send to the bootloader. A quick way to generate random bytes is by reading from *'/dev/urandom'*:

```
dd if=/dev/urandom of=aes_input.bin count=128 bs=1
```

This will put 128 random bytes into *'aes_input.bin'*. Then, run the example side-channel collector using the following (remember to change the *'uart-sock'* argument to match your port:

```
python3 tools/sc_example.py --uart-sock 1337 --sc-sock
socks/sc_probe.py --i-file aes_input.bin --o-file aes_traces.dat
--byte-skip-count 0 --num-samples 200000
```

The script will print three messages. The "Getting plaintext…" is printed when 16 bytes of plaintext are loaded from the input file. "Collecting traces…" is printed before sending the plaintext to the bootloader and collecting power data as AES is performed. "Finishing" is printed when data collection has concluded.

1. After data has been collected, the output file '*aes_traces.dat*' is transferred from the server to a local computer using the 'scp' command for graphical plotting in a local environment. See the next section for instructions on how to do this.
2. The data is plotted using Spyder in the local environment as shown in Figure 9. A simple plotting python code is shown for reference purposes, teams are free to use it as a starting point.

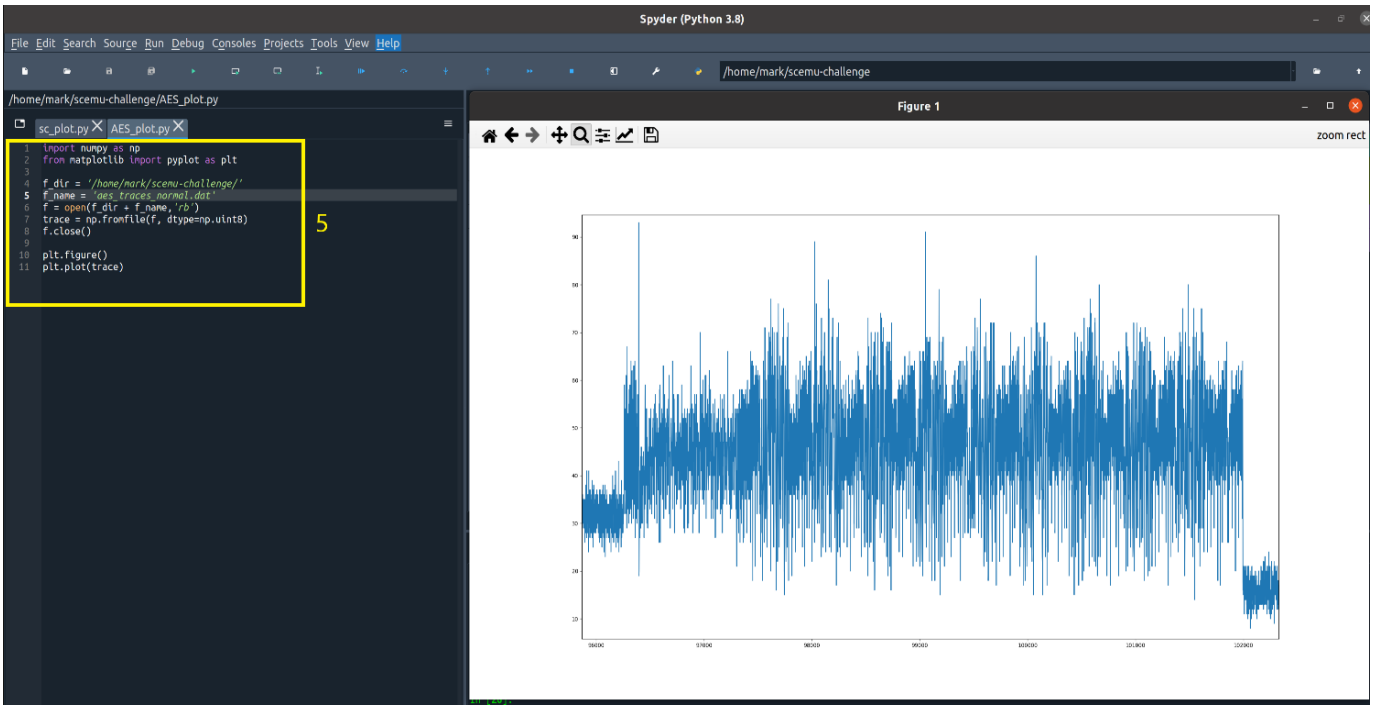


Figure 9: An example plot shows the SC waveform with repeated AES round structure that teams will want to find in emulated SC challenge traces.

Trace Plotting in Spyder

The scripts are configured so the emulated SC waveform is saved to the root directory of the deployment repository. When hosting the deployment on a remote server, teams will want to transfer the emulated data to a local machine for analysis. Our example transfers over SSH using the following scp command, invoked from a Linux terminal or a windows command prompt.

```
scp <<username>>@<<server IP address>>:<<path and filename>> <<local
path for storing file>>
```

Python is a powerful tool often used for scientific computing and is free to use. This tutorial uses Python from a Spyder environment and several libraries including NumPy. These libraries contain many useful math functions, and Matplotlib for visualizing data. First, the necessary libraries are loaded:

```
import numpy as np
from matplotlib import pyplot as plt
```

Next, point to the data file and read binary data using NumPy. Since the SC Emulator saves data to a flat binary file as 8-bit unsigned integers, we have to be careful how the data is read such that it is not misrepresented, or the analysis will be thrown off.

```
f_dir = '<<path to your file>>/'
f_name = 'example.traces'
f = open(f_dir + f_name,'rb')
trace = np.fromfile(f, dtype=np.uint8)
f.close()
```

The f_dir variable should end in '/' since it will be concatenated with the file name during open. The 'rb' command opens the file for reading binary data and setting the type on np.fromfile() ensures we read the correct number of bytes at a time.

The emulated data will be stored in Spyder as an array, so you can perform basic inspections by viewing the variable explorer tab. Next, you can visualize the data. These steps as well as the graph are shown in Figure 9.

```
plt.figure()
plt.plot(trace)
```

Since we are using Spyder development environment for this demo, you may want to change the default plot settings so that plots are generated in a separate interactive and re-sizable window. Go to the menu item 'tools', then click 'Preferences'. In the window that pops up, select 'IPython console', then the 'Graphics' tab, then under 'Graphics backend' change this to 'Automatic'. Click Apply. These steps are shown in Figure 10 below.

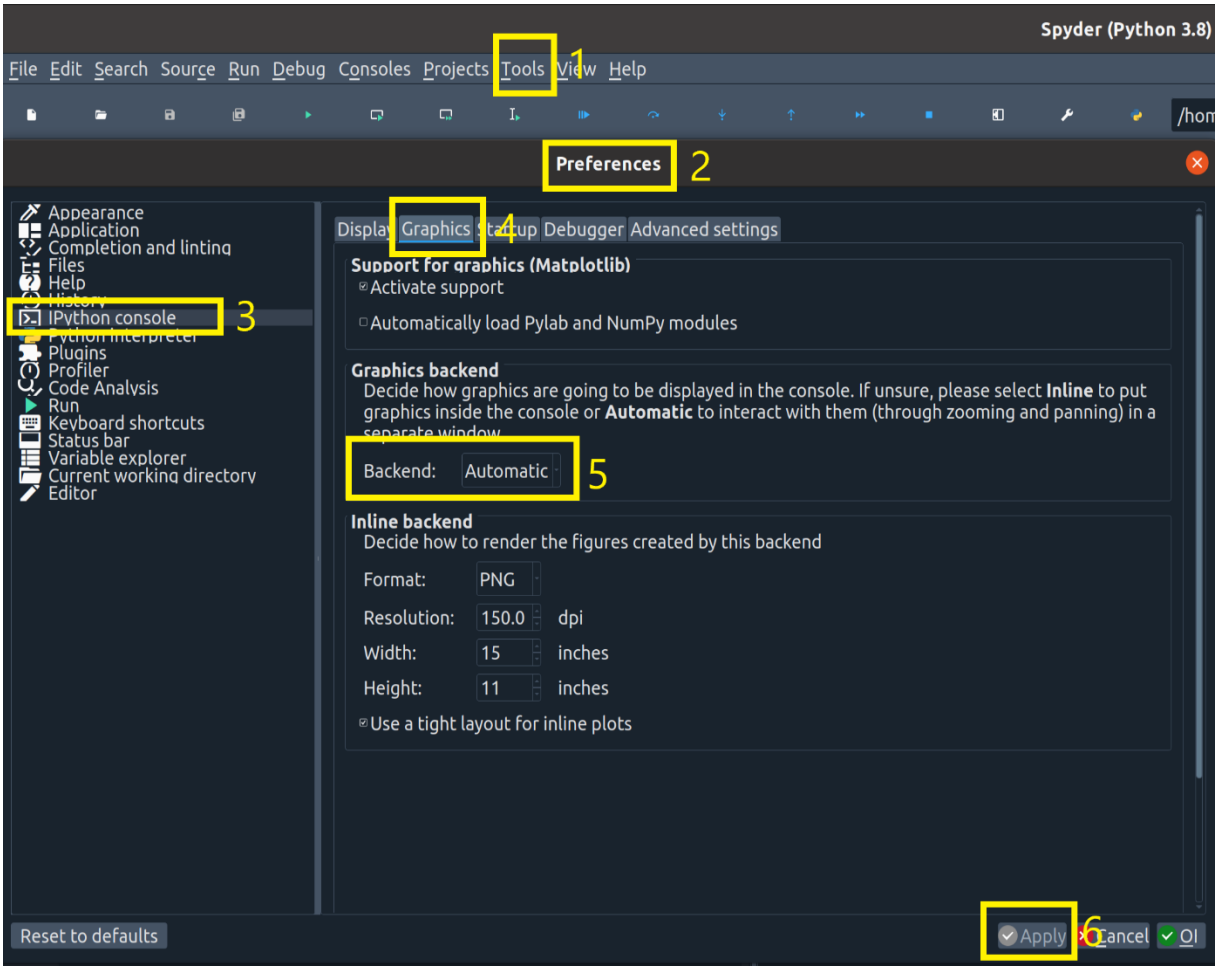


Figure 10: Spyder graphic preferences can be adjusted to aid in analysis.

At first glance, we notice several features in the Matplotlib figure. There are buttons that allow zooming in and scrolling through the waveform. Students should use the features to discover and clearly capture the AES waveform so that it is easily discernible from the noise.

SC Emulator Challenge

There are three main phases of the SC emulator challenge:

1) The first phase requires sifting through the noise and locating the AES encryption signal. This should be fairly straightforward in Spyder or a similar development environment. *Students will need to save a clear plot of the AES operations and post them in their teams' slack channel with an @organizers message.* An example plot of the emulated AES structure is shown in Figure 11. SC challenge waveforms may look different since noise is added before and after the AES encryptions.

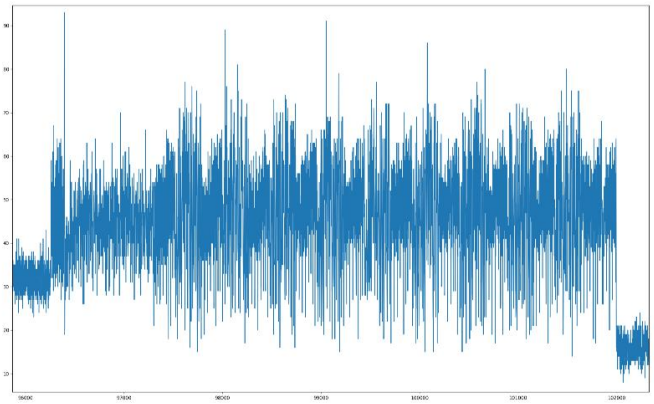


Figure 11: The first SC challenge flag is captured by identifying AES in the emulated power traces.

2) The second phase requires removing trace misalignment. All collected traces will have a variable amount of misalignment added before the AES operation is performed that must be removed. Aligned traces should be sent to the organizers. Figure 12 shows several traces with a variable amount of misalignment added before the signal of interest. *Submissions posted in the team's slack channel with an @organizers tag should include at least 5 traces plotted overlaid in different colors, similar to Figure 13.*

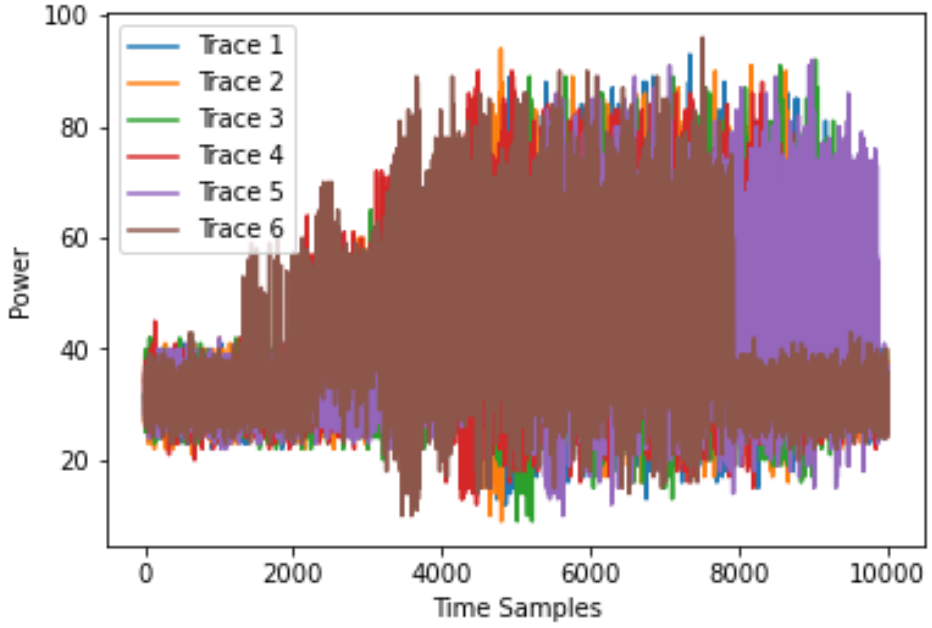


Figure 12: Misaligned traces returned by the SC receiver will be aligned prior to analysis.

Figure 13 shows those same traces with the misalignment removed. Teams can deal with misalignment however they choose.

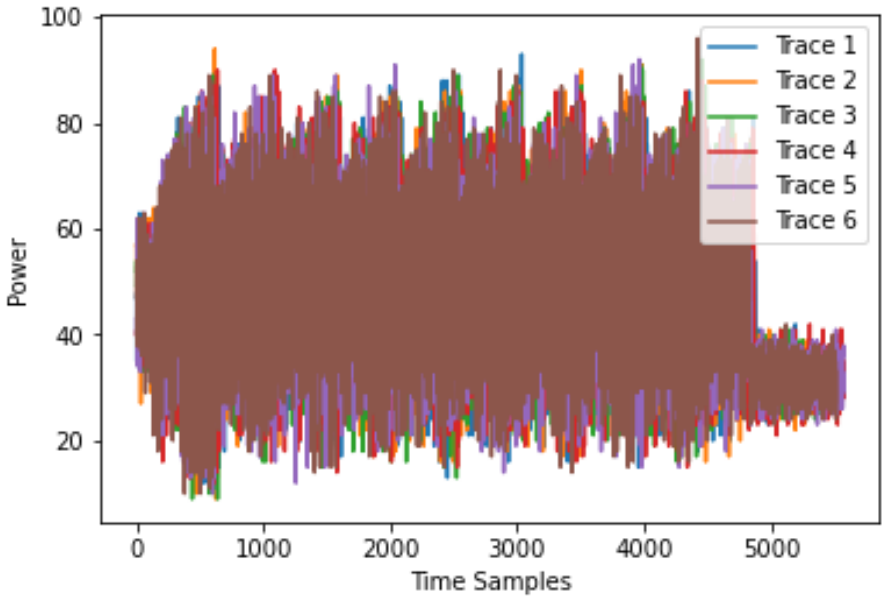


Figure 13: The aligned traces will be used in an SCA attack.

3) The third phase requires recovering the full AES key. After teams successfully align the traces, they will need to find the AES input file containing all data sent to the microcontroller for encryption. Teams will synchronize the 16-byte AES inputs to different emulated power traces returned by the SC receiver. To summarize, here are the items needed for the SCA attack.
    - Aligned SC receiver traces collected, each with a different 16-byte input
    - AES_input.bin file retrieved from server

    The mathematics of SCA are explained in numerous journal publications and textbooks [2]. Example measurements with MATLAB and Octave attack code can also be found online [9].

    Teams will post the recovered key in their teams' slack channels with an @organizers message. The key must be printed in hex, such as: `defda0b681c534d2907cdb6d4da76d73`.

References

[1] NIST, Advanced Encryption Standard (AES) (FIPS 197) (Nov. 26, 2001).

[2] S. Mangard, E. Oswald, E, and T. Popp, 2010. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (1st. ed.)*. Springer Publishing Company, Incorporated.

[3] F. Durvaux and M. Durvaux. 2020. *SCA-Pitaya: A Practical and Affordable Side-Channel Attack Setup for Power Leakage--Based Evaluations*. Digital Threats: Research and Practice 1, 1, Article 3 (March 2020), 16 pages. DOI:https://doi.org/10.1145/3371393

[4] https://meltdownattack.com/

[5] P. Kocher, J. Jaffe, B. Jun, et al., Introduction to differential power analysis. *J Cryptogr Eng* **1,** 5–27 (2011). https://doi.org/10.1007/s13389-011-0006-y

[6] F.-X. Standaert. *Introduction to Side-Channel Attacks*. https://perso.uclouvain.be/fstandae/PUBLIS/42.pdf
[7] https://github.com/ANSSI-FR/ASCAD
[8] http://dpabook.iaik.tugraz.at/