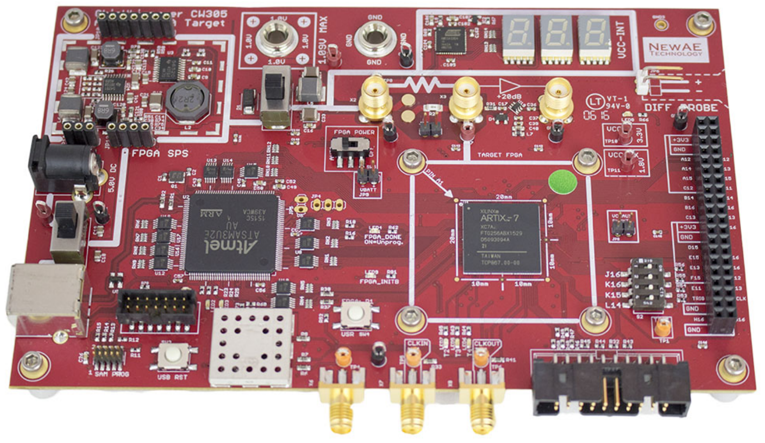
NAEAN0010: Power Analysis on FPGA Implementation of AES Using CW305 & ChipWhisperer®



Alex Dewar, Jean-Pierre Thibault, Colin O'Flynn

Abstract: *Side channel power analysis can be used to recover keys from cryptographic hardware blocks. Thus appnote demonstrates how to perform a power analysis attack on an AES block implemented on a FPGA, using the CW305 target board. Power analysis uses a Correlation Power Analysis (CPA) attack implemented in Python. Discussion of porting AES cores to the CW305 FPGA is also included*

Last Update: Oct 29, 2020

Contents

1 Power Analysis Background

It has been known that the power consumed by a digital device varies depending on the operations performed since at least 1998, when Kocher, Jaffe, and Jun showed the use of the power analysis for breaking cryptography.[1] The first example given was that of Simple Power Analysis (SPA), where knowing the sequence of operations would directly allow read-out of the secret key. Differences in power consumption for different operations allows breaking of cryptographic algorithms using SPA.

Fundamentally, this is due to physical effects of how digital devices are built. A data bus on a digital device is driven high or low to transmit signals between nodes. The bus line can be modeled as a capacitor, and we can see that changing the voltage (state) of a digital bus line takes some physical amount of energy, as it effectively involves changing the charge on a capacitor.

1.1 Power Analysis and AES

Simple Power Analysis, which allows different code flow to be seen via power consumption, is not typically applicable to AES. A more powerful model - that the power consumption of a device depends on the data that it's manipulating - gives rise to various other attacks that are relevant to AES.

Consider some operation, f, that a digital device is performing. Since the power consumption that a device is measuring depends on the data that it is manipulating, the output of f will be visible in the power traces. In the case of AES, $C = f(P, K)$, where P is a value known to the attacker (typically the plaintext or ciphertext), and K is a secret key that the attacker is trying to obtain. Assume we record the power consumption of the device while it is performing AES N times into N power traces, sampled with an analog to digital converter at some sampling frequency. Then, if we generate $C' = f(P, K')$, with K' as all possible values of K, we can compare C' to our recorded power traces and choose the K' that leads to the best comparison as our guess for the key. Here, f will determine the search space of K'. If $C = f(P, K)$ only depends on a single byte of K, then each byte of K can be attacked individually, with a search space of 2^8. In practice, it is also important that f be non-linear, as this will eliminate linear relationships between the input and intermediate values.

How C and the power consumption are compared is important. For a Differential Power Analysis attack, a single bit of C' is used to group power traces. A difference of means of the two groups is then taken, with the trace with the largest difference of means being our guess. If K' is wrong, or the device isn't manipulating the data at a point in the power trace, the traces should be randomly distributed and therefore have a similar mean, giving a low difference. Instead, if K' is correct, one group will have higher power consumption on aver-

[1] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 388–397. ISBN: 978-3-540-66347-8. URL: http://dl.acm.org/citation.cfm?id=646764.703989.

age than the other and there will therefore be a high average difference between the trace groups at that point.

A more powerful attack, Correlation Power Analysis, works off the assumption that each bit of C contributes roughly the same to the power consumption of the device. Stated another way, the power contributed by the data is proportional to the Hamming weight of C. The Hamming weight of a binary number is the number of bits set to 1. Here, the comparison is the linear correlation between C' and the recorded power traces. A wrong K' will give a low correlation and a correct K' will give a high correlation. Similarly, if f is not occurring at a particular point in the power trace, the correlation will be low.

It is important to note that these attacks assess each point in time separately. As such, it is key that the information leakage for f occur at the same sample point in each trace. If this is not the case, the traces will need to be resynchronized. Various resynchronization methods are available in ChipWhisperer, such as Sum of Absolute Difference and Dynamic Time Warp; however, they will not be covered in this document.

Test Vector Leakage Assessment (TVLA), a methodology for evaluating the side channel leakage of an embedded device, uses Welsh's T-Test and various trace groupings to assess if power leakage is present. TVLA will be covered in a later section.

In addition to the above attacks, more complex attacks such as template attacks and higher order DPA/CPA exist, with various advantages and disadvantages over normal CPA/DPA attacks. These attacks will not be covered in this white paper.

1.2 Hardware Leakage Models

Choosing an attack point, f, is key for a successful side channel attack. Software implementations of AES, for example, load values from memory onto a high-capacitance data bus, making the side channel information for associated operations particularly clear. For this reason, as well as the nonlinearity mentioned earlier, the SBox or T-Table lookup operations are ideal spots to attack a software AES implementation.

Hardware AES, on the other hand, varies much more, with the ideal attack point depending on the design of the AES block. For example, the Hamming weight of the SBox will still leak information as in the case of Software AES; however, a successful attack may take many orders of magnitude more traces to break than the software implementation. Often a better attack point to pick is registers used to store the AES state. The placement of these registers can vary between implementations, a key reason why leakage models for hardware AES vary. Another thing that must be taken into consideration is the previous state of the bus on which the new data we care about is being loaded to. For example, if a register previously had the value `0xFF`, and the SBox lookup also results in `0xFF`, no power will need to be consumed to change the state of the register. This can be incorporated into the leakage model by taking the Hamming distance between the previous value in the register and the new value; the Hamming weight of the two values XOR'd. The Hamming distance of registers is primarily the concern of attacks on hardware AES. Microcontrollers, on the other hand, typically have their register bits set to a value between 0 and 1 before being updated, as this will save power on average and reduce the voltage swing when changing a value.

A high level block diagram of an AES implementation that completes 1 round of AES per clock cycle is shown in Figure 1.

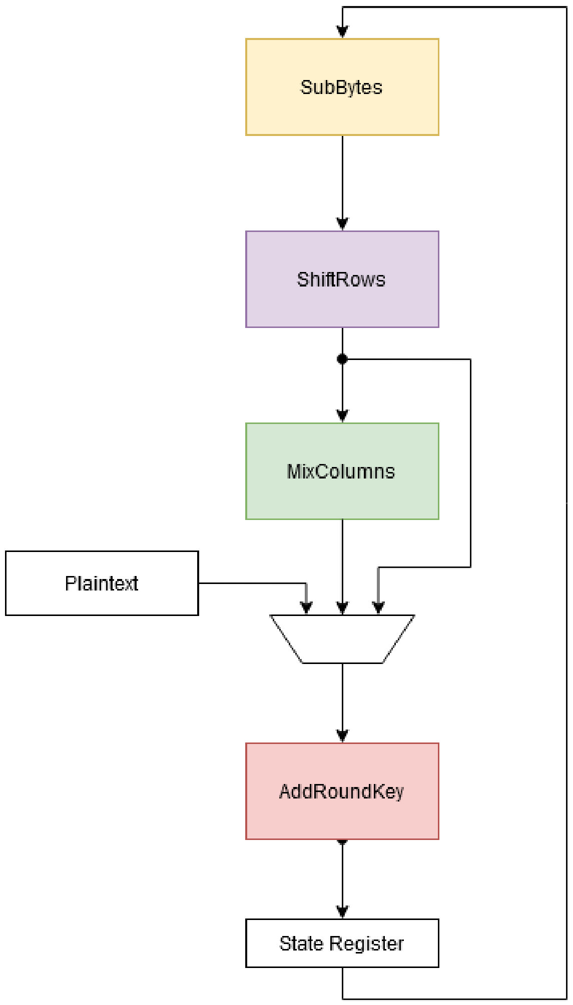


Figure 1: Hardware AES Block Diagram

The plaintext mixed with the key will be loaded into the State Register, then put through the required rounds of AES, finally resulting in the ciphertext again being put into the State Register. The first transition of the state register is not ideal - the presence of MixColumns and another AddRoundKey means the output state will not depend on a single byte of the key, greatly increasing the search space of the attack. Instead, the ideal leakage model to use for this implementation is the Hamming distance between the final two states - ciphertext and InvSubBytes(InvShiftRows(AddRoundKey(ciphertext))). Again, this is just one possible implementation of AES. If the implementation does not place the ciphertext in the state register, for example, this attack avenue will not be available.

2 ChipWhisperer Background

ChipWhisperer is a set of many tools useful for embedded hardware security research. In particular, there are the ChipWhisperer-Capture devices (which perform sampling of power measurements), the ChipWhisperer hardware targets, the ChipWhisperer target device firmware and target device FPGA blocks, and ChipWhisperer analysis software and libraries.

Taken as a whole, the ChipWhisperer platform includes tools for *all* aspects of side-channel power analysis and fault injection. This ecosystem makes ChipWhisperer unique, as it does not rely on external tooling. This also makes it ideal for environments where setups need to be replicated, as it makes minimal assumptions about existing tools.

Some of the tools within ChipWhisperer have not been optimized due to the very wide ranging nature of the ChipWhisperer system – for example ChipWhisperer does not currently include any high-performance acceleration of the analysis algorithms. ChipWhisperer *can*, however, easily interface to several other tools to fill those gaps. This whitepaper will highlight two such open-source tools that specifically emphasize analysis performance, and can easily perform several of the attacks several hundred times faster than the ChipWhisperer analysis tools.

2.1 ChipWhisperer Capture Synchronous Sampling

Commercial oscilloscopes typically provide their own sampling clock which is not synchronized to the device clock. In the ChipWhisperer-Capture system, the sample clock is instead derived from the device clock to measure a consistent point; for example it can be used to measure the power consumption on the clock edge. A comparison of measurements taken with an unsynchronized and synchronized sample clock is shown in Fig. 2. This relaxes the sample rate requirements – that is instead of requiring 1 – 5 GS/s, we can perform attacks at the same speed as the target device (or some multiple of it). This sample synchronization is a unique feature of the ChipWhisperer platform.

Note that sample clock synchronization is different from the trigger input that all oscilloscopes provide. With a real-time oscilloscope, the internal sample clock of the oscilloscope will be running at all times, and the sample occurs at the next clock edge after the trigger. Thus even though the oscilloscope is triggered at a repeatable time, there will be some random jitter between when the first sample occurs relative to this trigger for unsynchronized (free-running) sample clocks.

The capture board also adds an adjustable delay (phase shift) between the input clock and the actual sample point, which can be used to fine-tune the location of the sample.

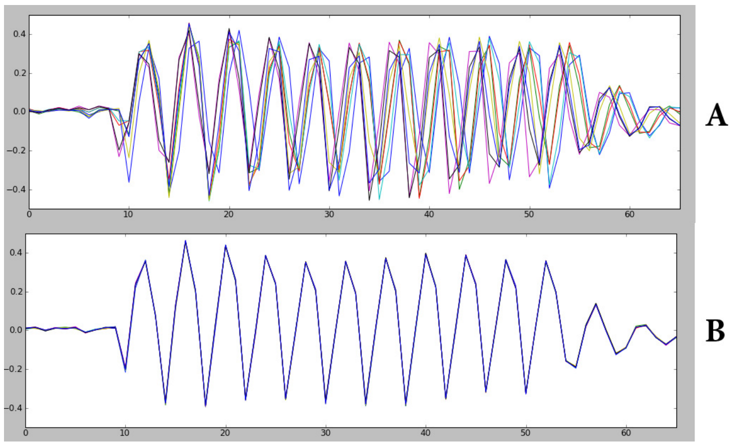


Figure 2: Eight power samples with the same input are taken and overlaid to show consistency of measurements. In *A* the sample clock is 100 MHz but not synchronized to the device clock, whereas in *B* the sample clock is 96 MHz, but synchronized with the device clock.

2.2 ChipWhisperer Target Boards

The ChipWhisperer ecosystem includes several “target boards”. These boards contain various types of devices such as Arm Cortex-M microcontrollers, small FPGAs, PowerPC devices, etc. They can be used during development of secure algorithms to validate the algorithms on various target boards.

The ChipWhisperer CW308 “UFO Boards” are the most flexible target, as they include many different target boards that can fit on the CW308 baseboard. However, these target boards *do not* include large FPGAs due to the higher power requirements of these FPGAs. The CW305 board is a stand-alone target which allows usage of a larger FPGA target to implement cores such as AES, ECC, etc.

2.3 CW305 Overview

The NewAE `NAE-CW305` is a target board containing an Artix A100 or Artix A35 FPGA, which is instrumented to simplify side-channel power analysis work. A photo of the board is shown in Figure 3.

A custom USB interface chip means you can trivially send and receive data to your FPGA design, while also performing FPGA configuration and adjusting external PLL operating frequencies all from the same interface. ESD protection on all I/O lines allows you to perform glitch insertion safely, and an optional BGA socket is perfect for comparing effects across many physical devices.

In order to use this board, you will typically provide:

1. A USB-A Connection used to power the board & provide communications to your FPGA core.

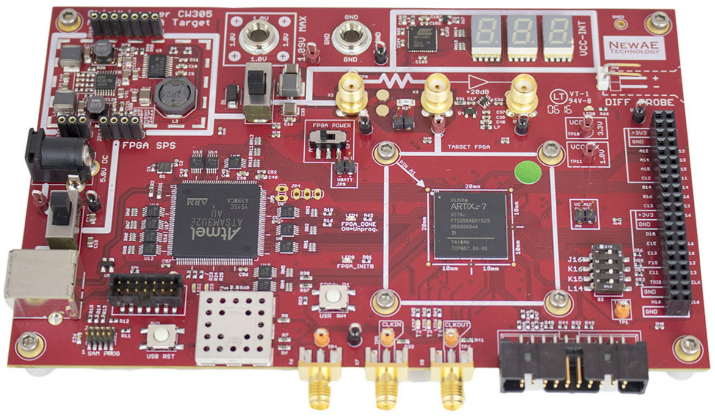


Figure 3: The NAE-CW305-04-A100-X-0.10

2. A bitstream programmed into the target FPGA, implementing your cryptographic core.
3. A connection at JP1, the 20-pin connector, to a ChipWhisperer capture platform (such as NAE-CWLITE-CAPTURE or NAE-CW1200) which provides clock and triggering.
4. A connection at X4, the SMA connector, to a ChipWhisperer capture platform (such as NAE-CWLITE-CAPTURE or NAE-CW1200) which provides the power measurement.

We will see more details of this in Section 3.

3 Hardware Setup

In this section, we will use the CW305 board and ensure it is correctly setup for power capture.

3.1 Overview

The objective of our side-channel measurement is ultimately to provide the framework shown in Figure 4. In this example, the *Algorithm Under Test* is the algorithm you want to test. The rest of the circuitry supports the objective of performing side-channel power analysis on this algorithm.

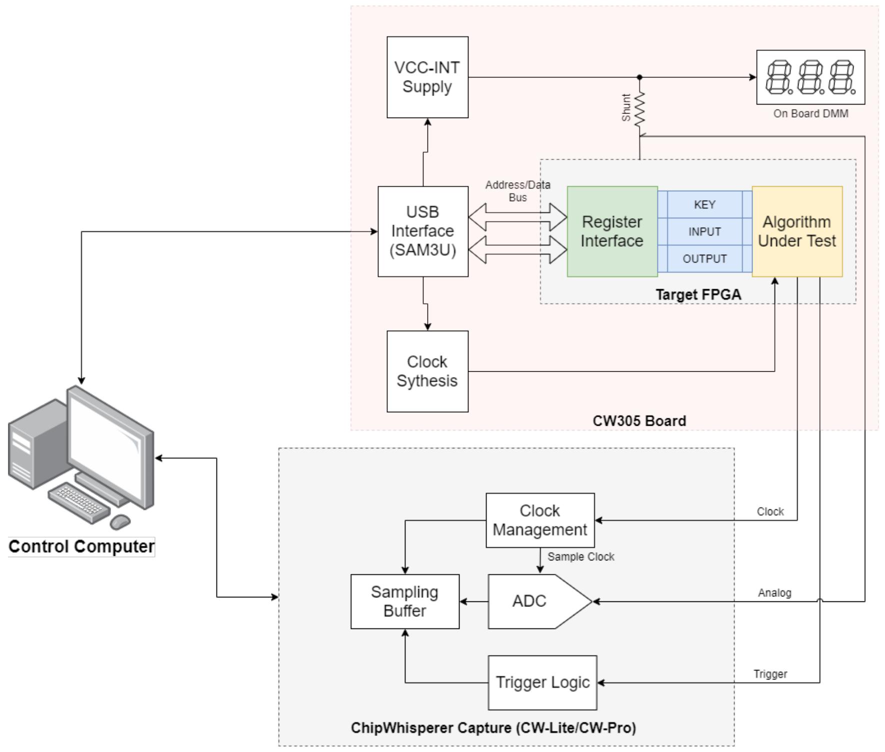


Figure 4: The CW305 allows you to spend time implementing your *Algorithm Under Test*, and let ChipWhisperer provide the supporting framework.

Note that the *Register Interface* is provided as a sample Verilog RTL from NewAE Technology Inc – you can of course use your own interface to your core. Using our *Register Interface* means you can use the matching open-source Python code on the *Control Computer* to perform operations such as loading input, keys, output, or triggering operations with minimal effort.

At a physical level, the CW305 provides an *Address/Data Bus* between the USB interface microcontroller and the FPGA. This address/data bus allows you to define a typical address/data bus on the FPGA instead, and write arbitrary data into the FPGA.

As the code running on the USB interface microcontroller (a Microchip SAM3U) is open-source, you can freely change this interface to anything you choose. Such work can be performed under a consulting or support contract if you wish of course too. For most users, however, we highly recommend using instead a shim layer inside the FPGA, as this will reduce your effort and maintenance, and you be able to take advantage of future firmware updates to the SAM3U microcontroller.

3.2 CW305 Default Setup

The CW305 should come with the following jumper and switches configured already. If you are interested in the function of all the switches, see the full documentation on https://rtfm.newae.com.

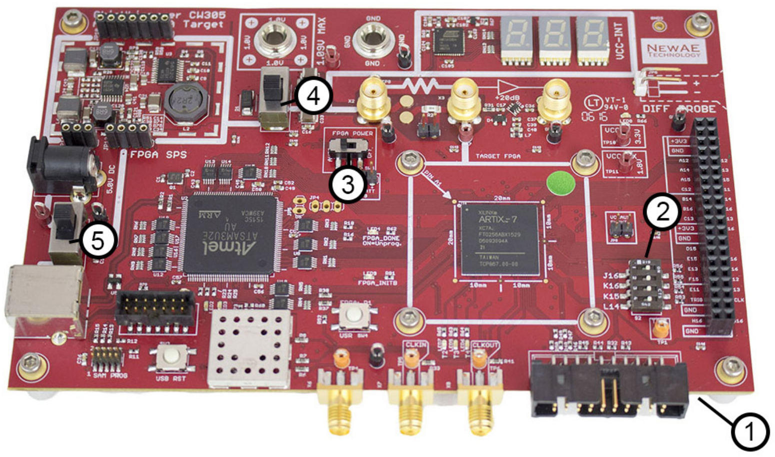


Figure 5: The CW305 configuration switches.

① **S1 (DIP switch, bottom-side, lower-left corner)**

The DIP `S1` switch configures the FPGA bitstream mode: the `M2`, `M1`, `M0` match exactly with mode pins on the Artix-7 FPGA. Normally these are set to `111` which will allow the microcontroller on the CW305 to configure the FPGA bitstream with your design.

1. M2: Set to `1`
2. M1: Set to `1`
3. M0: Set to `1`

② **S2 (DIP switch, top-side, lower-right corner)**

The DIP `S2` switch are routed to four FPGA pins. With the default usage, we use them to configure if the clock comes from the on-board PLL, or from an external clock (such as the ChipWhisperer). By default, we will configure them to use the clock from the on-board PLL, as well as route that clock *out* to the ChipWhisperer-Capture.

1. J16: Set to `0`
2. K16: Set to `1`
3. K15: Set to `1`
4. L14: Set to `1`

③ **SW5, "FPGA POWER", small surface-mount switch**

Set this switch to `AUTO`.

④ **VCC-INT power source selection, through-hole SPDT switch**

This switch should be set down, which selects the on-board switch-mode power supply for the VCC-INT supply.

⑤ **Input power source selection, through-hole SPDT switch**

This switch can be used as a power switch. It selects to power the board via the USB-A connector, or the DC power jack. Most users can simply use the USB-A connector as a power source.

3.3 Power On/Off with SMA Connector Removals

It is important to avoid accidentally shorting out the board, as can often happen due to the conductive (and often grounded) external SMA cable that could touch portions of the CW305.

☞ `WARNING:` Always power off the CW305 board when connecting or disconnecting the SMA cable. This can be easily done with power switch by the USB port. It is *very easy* to accidentally short the power using the conductive outside of the SMA cable, which can *permanently damage* the CW305.

3.4 CW305 to ChipWhisperer-Pro

☞ Follow this section if you are using a ChipWhisperer-Pro.

With the CW305 board set to the defaults above, the additional work is simply to connect a ChipWhisperer-Pro Capture box, which is part of `NAE-CW1200-KIT`, to the CW305 board.

In this case, you simply need to perform the following:

1. Turn off the CW305 board (if already plugged in – see warning above).
2. Connect the 20-pin "Target Connector" from the ChipWhisperer Capture to JP1 on the CW305.
3. Connect the SMA cable from the "Measure" SMA on the ChipWhisperer Capture to X5 (amplified shunt output) on the CW305.
4. Connect the ChipWhisperer-Pro to a computer using a USB-A cable. You may need to also provide DC power to the ChipWhisperer-Pro with some versions.
5. Turn on the CW305 board (or plug in if not plugged in yet).

An example of this is shown in Figure 6.

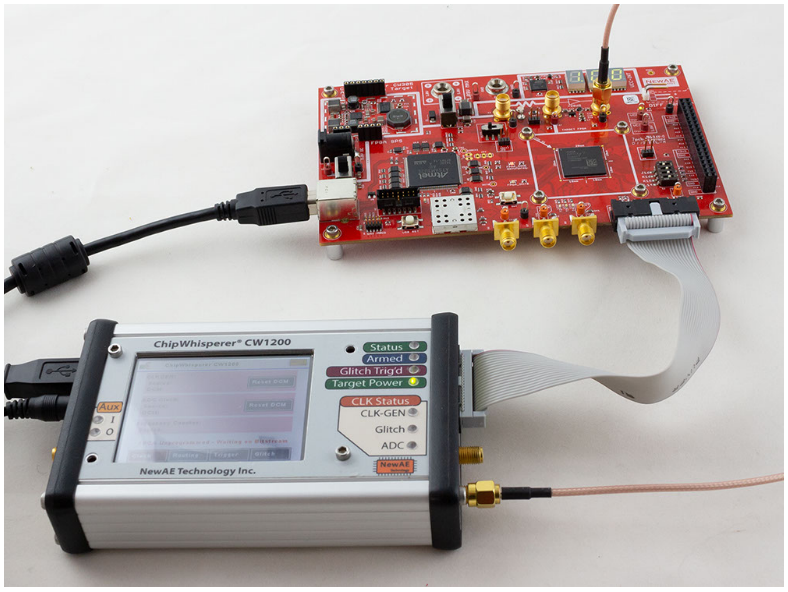


Figure 6: The CW305 can be easily interconnected to the ChipWhisperer-Pro Capture system.

3.5 CW305 to ChipWhisperer-Lite

☞ Follow this section if you are using a ChipWhisperer-Lite capture board.

With the CW305 board setup as defaults require, the additional work is simply to connect a ChipWhisperer-Lite Capture board (part number `NAE-CWLITE-CAPTURE`, which is part of `NAE-SCAPACK-L1`, `NAE-SCAPACK-L2`, and `NAE-CWLITE-2PART`) to the CW305 board.

In this case, you simply need to perform the following:

1. Turn off the CW305 board (if already plugged in – see warning above).
2. Connect the 20-pin "Target Connector" from the ChipWhisperer Capture to JP1 on the CW305.
3. Connect the SMA cable from the "Measure" SMA on the ChipWhisperer Capture to X5 (amplified shunt output) on the CW305.
4. Connect the ChipWhisperer-Lite to a computer using a Mini-USB.
5. Turn on the CW305 board (or plug in if not plugged in yet).

An example of this is shown in Figure 7.

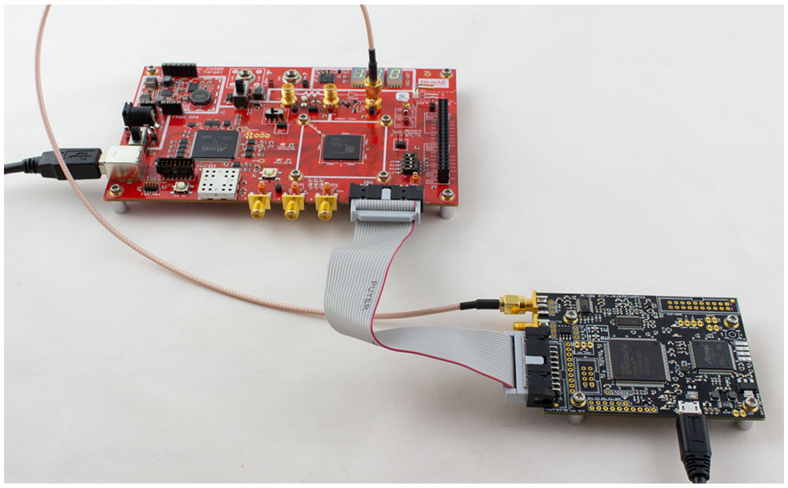


Figure 7: The CW305 can be easily interconnected to the ChipWhisperer-Lite Capture board.

3.6 CW305 to Oscilloscope

The final example is one in which a ChipWhisperer capture platform is not used. In this case you cannot complete the rest of the power analysis tutorial by following this white paper, but we include this section as a reference for users who wish to interface the CW305 with external tools.

In this case, you simply need to perform the following:

1. Turn off the CW305 board (if already plugged in – see warning above).
2. Connect a SMA to BNC cable from the oscilloscope to X5 (amplified shunt output) on the CW305.
3. Connect an oscilloscope probe to test point `TP1`, labeled `TRIG`.
4. Turn on the CW305 board (or plug in if not plugged in yet).
5. Ensure the probe connected to X5 is AC coupled

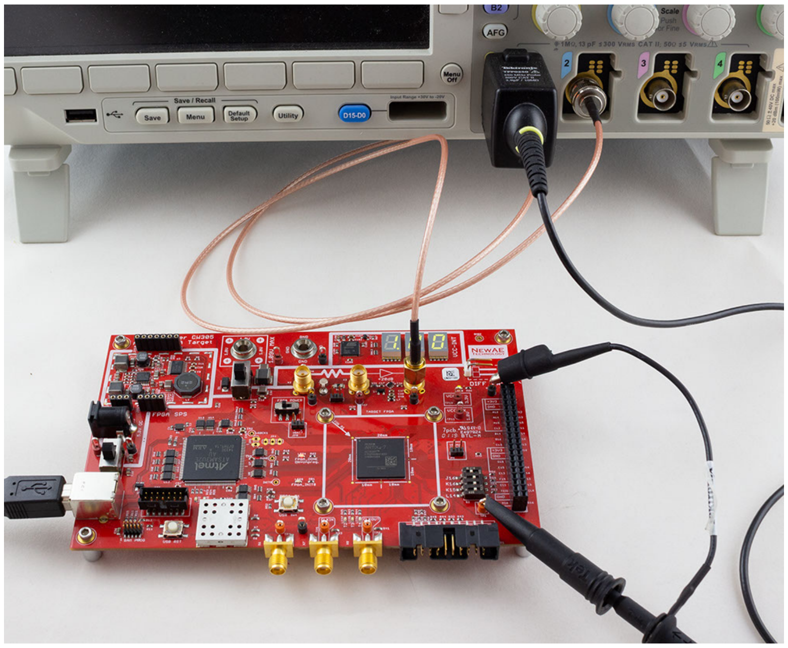


Figure 8: The CW305 can be easily connected to a regular oscilloscope.

3.7 CW305 LED Indicators

The CW305 target board includes three user LEDs (LED5/LED6/LED7), along with several LEDs to indicate the status of the FPGA configuration. The most important of those is LED4, called `FPGA_DONE`. This has the output of the `FPGA_DONE` pin.

If this led is `ON`, it means the FPGA is unconfigured. This happens on power-on, or if you have attempted to load an invalid bistream. The most common cause of this is using a bitstream built for the Artix A35 instead of the Artix A100 (or vice versa).

The three user LEDs (LED5/6/7, located to the next of the 20-pin connector, above the three side-mounted SMAs) are controlled by the FPGA. With the reference bitfile, they indicate the following:

- LED7 (red): USB clock heartbeat
- LED5 (green): cryptography (target) clock heartbeat
- LED6 (blue): trace capture underway; controlled by Python via the `REG_USER_LED` register.

4 Software Setup

This section will include setup of the toolchain to build the FPGA target, along with the setup of the toolchain for the power analysis.

4.1 Firmware Setup

Building firmware for the CW305 requires either the Xilinx Vivado toolchain (CW305-7A35 or CW305-7A100) or the Xilinx ISE toolchain (CW305-7A100 only). We currently only support the Xilinx Vivado toolchain in this tutorial.

Note that we have prebuilt bitstreams checked into the CW305 Examples of the ChipWhisperer Github. You can use these to perform the first setup to ensure your CW305 and ChipWhisperer setup has been done correctly before rebuilding the FPGA image itself.

4.2 Python Library

Communication with both NewAE Capture and Target boards can be done via the chipwhisperer Python library. The library is available on Github and can be run natively on Windows 7 and above, Mac OSX, and Linux. The Python library requires Python 3.6 or later. The library is documented on ReadTheDocs, including installation instructions.

Devices with firmware versions lower than 0.21 (ChipWhisperer-Lite and ChipWhisperer-Nano), 0.31 (CW305), or 1.21 (ChipWhisperer-Pro) require manual installation of USB drivers. ChipWhisperer will function with generic WinUSB or libusb0 drivers, which can be installed using Zadig or via provided libusb0 drivers in `chipwhisperer/hardware/newae_windowsusb_drivers.zip`, respectively.

Devices running on Linux will need to add a `rules.d` entry and add their user to an appropriate group to have permission to read and write to NewAE's USB devices. More details are available on ReadTheDocs.

Once installed, the Python library can be used to update the ChipWhisperer firmware:

```
import chipwhisperer as cw
# Capture FPGA automatically
# programmed on connection if unprogrammed
scope = cw.scope()

# erase USB firmware
prog = cw.SAMFWLoader(scope)
prog.enter_bootloader(True)

# or "cw1200" or "cwnano"
prog.program("/path/name/of/serial/device",
             hardware_type="cwlite")
```

ChipWhisperer can be run in a regular Python environment. Some additional features, however, are available if the Python library is used with Jupyter notebooks. An example of this is an attack result table. Many attack examples are also available in the form of Jupyter Notebooks from ChipWhisperer Jupyter.

5 Capturing Power Traces

Capturing those traces!

5.1 Setup

```
import chipwhisperer as cw
scope = cw.scope()

# scope is optional
target = cw.target(scope, cw.targets.CW305,
            "/path/to/bitstream", force=True)
```

Here the force parameter will cause the FPGA to be reprogrammed, even if it has already been programmed. If this is not desired, the force parameter can be omitted.

☞ Bitstreams stored in the FPGA are volatile, meaning they will be lost once the FPGA loses power. Persistent storage is available on the CW305 in the form of a SPI flash chip. This flash chip can be programmed via the Python API as well. Programming the SPI flash requires a shim bitstream that connects appropriate pins on the on board SAM3U to the SPI flash pins. If the `fpga_id` parameter indicating the FPGA present on the CW305 is given when calling `cw.target()`, the CW305 will automatically be programmed with the correct SPI shim bitstream when calling `spi_mode()`. However, the bitfile that you wish to write to the SPI flash must still be explicitly given to `spi.program()`.

```
import chipwhisperer as cw

target = cw.target(None, cw.targets.CW305,
            fpga_id="100t") # or '35t'

spi = target.spi_mode()

spi.erase_chip() # erase full chip
with open("/path/to/bitfile", "rb") as f:
    data = list(f.read())
    spi.program(data)
```

You will need to set the switches on the bottom of the CW305 to SPI mode to force reading the bitstream from the SPI memory chip. To reprogram the SPI flash itself you will need to change them back to USB mode. For most users you *do not need* to use the SPI boot option, as the USB configuration takes only a few seconds.

The CW305 FPGA's VCC-INT pin is powered by an adjustable voltage regulator. The regulator can be set to output 0.80V to 1.10V. Typically, the default of 1.00V is sufficient:

```
target.vccint_set(1.0)
```

The CW305 contains a 3 output PLL. Its architecture is shown in Figure 9

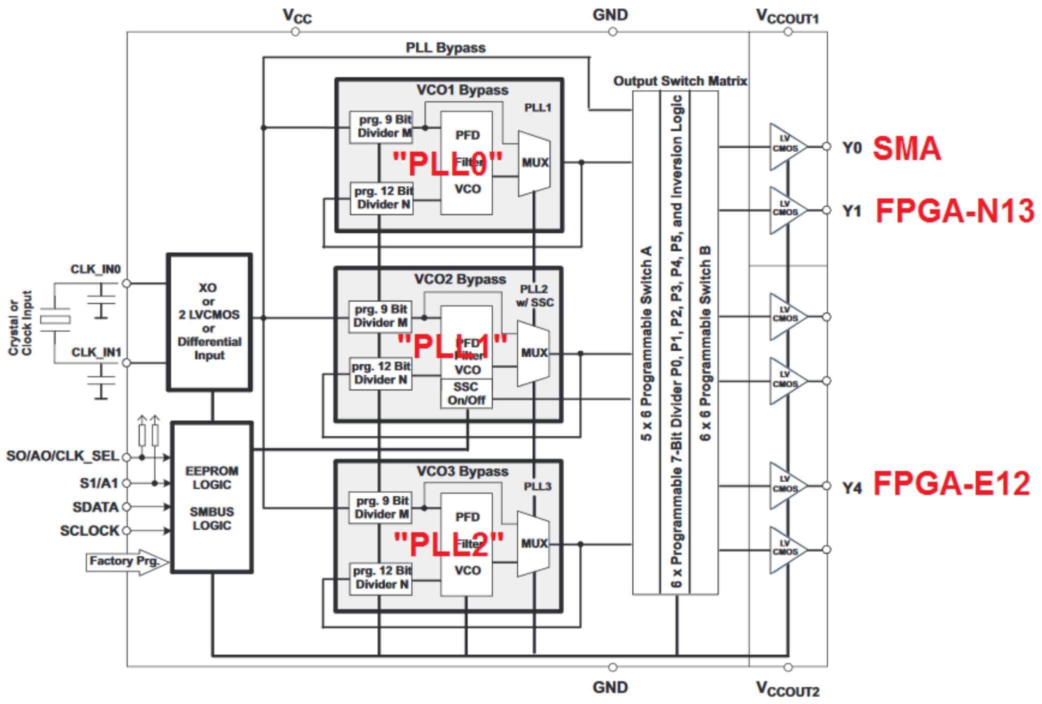


Figure 9: CW305 PLL Architecture

Here Y1 and Y4 are fixed to PLL1 and PLL2, respectively. Y0 can be connected to PLL0, PLL1, or PLL2 via the Python API:

```
# connect PLL0 and Y0
target.pll.pll_enable_set(True) # enable PLL
target.pll.pll_outsource_set("PLL0", 0)
```

PLL output frequency is configurable from 630kHz to 167MHz:

```
# set PLL 0 to 10MHz
target.pll.pll_outfreq_set(0, 10E6)
```

For an application based on the example ChipWhisperer AES core, setting PLL1 to 10MHz is sufficient. It is important to note that the ChipWhisperer-Lite and ChipWhisperer-Pro are limited to a sample rate of 105MS/s. For best results, do not set PLL1 above 26MHz, as this will allow the ChipWhisperer to sample at 4x the clock rate of the target.

ChipWhisperer (Lite or Pro) setup can begin with the following:

```
import chipwhisperer as cw
scope = cw.scope()
scope.gain.db = 25
scope.adc.samples = 129
scope.adc.offset = 0
scope.adc.basic_mode = "rising_edge"
scope.clock.adc_src = "extclk_x4"
scope.io.hs2 = "disabled"
```

When both the CW305 and the ChipWhisperer capture board are setup, ensure the ADC is locked:

```
# ensure ADC is locked:
scope.clock.reset_adc()
assert (scope.clock.adc_locked), "ADC failed to lock"
```

5.2 Capturing Traces

A ChipWhisperer project can be used to store traces:

```
proj = cw.project("desired_path.cwp", overwrite=True)
```

Data can be written and read from the FPGA as follows:

```
target.fpga_write(<address>, [<data>])
data = target.fpga_read(<address>, <number of bytes>)
```

Before capture, you should:

1. Get a key text pair object using ChipWhisperer
2. Ensure a valid key is loaded on the target

A basic capture loop typically has the following steps:

1. Arm the ChipWhisperer
2. Write the plaintext to the target
3. Trigger the encryption
4. Capture the trace
5. Read the ciphertext back from the target
6. Organize and store the data

For example:

```
import time
N_traces = 5000 # change as required
ktp = cw.ktp.Basic()
key, text = ktp.next()
target.fpga_write(target.REG_CRYPT_KEY, key)

for i in range(N_traces):
    #arm the scope
    scope.arm()

    # write the plaintext
    target.fpga_write(target.REG_CRYPT_TEXTIN, text)

    key, text = ktp.next()

    # Trigger the encryption
    target.fpga_write(target.REG_USER_LED, [0x01])
    target.usb_trigger_toggle()

    # Capture the trace
    ret = scope.capture()

    if ret:
        print("Capure timeout")
        continue

    # organize and store the data
    output = target.fpga_read(target.REG_CRYPT_CIPHEROUT, 16)
    wave = cw.get_last_trace()
    trace = cw.Trace(wave, text, output, key)

    proj.traces.append(trace)


proj.save()
```

You may notice in the example above that ChipWhisperer allows you to refer to FPGA registers by name, rather than by address. `CW305.py` actually parses the Verilog source code in order to make this possible; it makes it easier to keep the Verilog and Python in sync, which is very handy during development. If `fpga_id` is specified in the call to `cw.target()`, then `CW305.py` will automatically find the correct Verilog source file for the register definitions. It is also possible to pass the definition file explicitly to `cw.target()` , which may be useful during development:

```
target = cw.target(scope, cw.targets.CW305, \
                   defines_files=["my_def.v"])
```

Multiple source files may be passed to `defines_files`, which is why they must be passed as a list.

The example above showed the details of the capture routine, to show what happens under the hood; the following shorthand can be used to achieve the same result:

```
N_traces = 5000 # change as required
ktp = cw.ktp.Basic()
for i in range(N_traces):
    key, text = ktp.next()
    trace = cw.capture(scope, target, text, key)
    if trace is None:
        continue
    proj.traces.append(trace)

proj.save()
```

6 Performing AES CPA Attack

6.1 Opening the Project

If the project has not been closed, either by closing Python, deleting the project object, or by calling `proj.close()`, then the project can continue on to be used for analysis. Otherwise, it must be reopened as follows:

```
proj = cw.open_project("/path/to/project.cwp")
```

6.2 Running the Attack

CPA analysis can be setup as follows:

```
import chipwhisperer.analyzer as cwa
leak_model = cwa.leakage_models.sbox_output
attack = cwa.cpa(project, leak_model)
```

ChipWhisperer has leakage models available from `cwa.leakage_models`. These can be seen by printing that class, or via ChipWhisperer's API Documentation. Not all leakage models available are sensible. For example some, such as `cwa.leakage_models.round_1_2_state_diff_text` requires the key to be known. Otherwise, the attack must be perform over 4 key bytes instead of 1. ChipWhisperer does not currently support attacks over multiple key bytes, or attacks with a known key.

Once the attack is setup, it can be run with:

```
results = attack.run()
```

If you're running in Jupyter, you can run instead with:

```
results = attack.run(cwa.get_jupyter_callback(attack, 25))
```

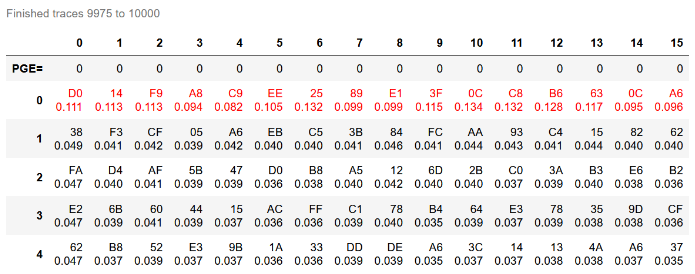
This will display the results of the attack in a nicely formatted table. The table is updated every time 25 traces are processed by default. By changing the 25 in the code above, other callback rates can be used. The table after an attack on AES running in the CW305 is shown in Figure 10.

6.3 Interpreting Results

One of the simplest results from a CPA attack is the guessed key:

```
results.key_guess()
```

However, there's much more you can learn from a CPA attack besides just whether or not you recovered the key. One important question for an unsuccessful attack is "how close was the attack to being successful?". Partial guessing entropy (PGE) is a useful measure in this case. PGE is how many wrong key



Finished traces 9975 to 10000

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PGE=	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	D0 0.111	14 0.113	F9 0.113	A8 0.094	C9 0.082	EE 0.105	25 0.132	89 0.099	E1 0.099	3F 0.115	0C 0.134	C8 0.132	B6 0.128	63 0.117	0C 0.095	A6 0.096
1	38 0.049	F3 0.041	CF 0.042	05 0.039	A6 0.042	EB 0.040	C5 0.040	3B 0.041	84 0.046	FC 0.041	AA 0.044	93 0.043	C4 0.041	15 0.044	82 0.040	62 0.040
2	FA 0.047	D4 0.040	AF 0.041	5B 0.039	47 0.039	D0 0.036	B8 0.038	A5 0.040	12 0.042	6D 0.040	2B 0.040	C0 0.037	3A 0.039	B3 0.040	E6 0.038	B2 0.036
3	E2 0.047	6B 0.039	60 0.041	44 0.039	15 0.037	AC 0.036	FF 0.036	C1 0.039	78 0.040	B4 0.035	64 0.039	E3 0.037	78 0.039	35 0.038	9D 0.038	CF 0.036
4	62 0.047	B8 0.037	52 0.039	E3 0.037	9B 0.037	1A 0.036	33 0.036	DD 0.039	DE 0.039	A6 0.035	3C 0.037	14 0.037	13 0.038	4A 0.038	A6 0.037	37 0.035

Figure 10: Jupyter CPA Table

guesses are ahead of the correct key. This isn't available in a real attack since you need to know the key to calculate it, but it's extremely useful in understanding whether the attack is partially successful, meaning more traces will lead to a successful attack, as well as if a brute force attack will be tractable. The PGE for a particular byte can be found as follows:

```
# get PGE for byte 0
results.simple_pge(0)
```

A full dataset sorted by their max correlation can be acquired via the `results.find_maximums()` method. This method returns a tuple of `(key_guess, loc_of_max, correlation)` for each sorted guess for each byte in the key. For example, to see the best correlation for the 4th key byte:

```
print(results.find_maximums()[4][0][2])
```

Note that `loc_of_max` is not normally calculated and therefore returns as 0.

6.4 Graphical Results

6.4.1 Output Vs. Time

ChipWhisperer also includes a class to graphically display the results:

```
plot_data = cwa.analyzer_plots(results)
```

which has three different data plots available to us. The first is output vs. time, which displays the calculated correlation at each point in time for the correct key, as well as the max positive and negative correlations at each point in time. If you've gotten the right key, you should see a large correlation spike at a point in time. This spike might be the same for each key byte, typical for hardware AES, or at different times for each key byte, typical for a software implementation. We'll use holoviews/bokeh for plotting the data since it deals well with large datasets like this; however, you can use any Python plotting library:

```
import holoviews as hv
from holoviews.operation.datashader import datashade, shade, dynspread, rasterize
from holoviews.operation import decimate
import pandas as pd, numpy as np
def byte_to_color(idx):
    return hv.Palette.colormaps['Category20'](idx/16.0)

a = []
b = []
hv.extension('bokeh')
for key_byte in range(0, 16):
    data = plot_data.output_vs_time(i)
    a.append(np.array(data[1]))
    b.append(np.array(data[2]))
    b.append(np.array(data[3]))

pda = pd.DataFrame(a).transpose().rename(str, axis='columns')
pdb = pd.DataFrame(b).transpose().rename(str, axis='columns')
curve = hv.Curve(pdb['0'], "Sample").options(color='black')
for key_byte in range(1, 16):
    curve *= hv.Curve(pdb[str(i)]).options(color='black')

for key_byte in range(0, 16):
    curve *= hv.Curve(pda[str(i)]).options(color=byte_to_color(i))
decimate(curve.opts(width=900, height=600))
```

ChipWhisperer versions 5.3.2 and above can simply do the following for the same effect if using Jupyter:

```
plot_data.plot_output_vs_traces()
```

The correct key correlation will be a color depending on the key byte, while incorrect keys will be in black. If an attack is successful, correlation peaks should be present for the correct key, indicating where the encryption operation is happening in time. Depending on the implementation, these correlation peaks may be at the same point in time (basic software implementation), all at the same time (typically one round per clock cycle), or somewhere in between.

An Output Vs. Time plot for a the TinyAES128C AES implementation running on an STM32F3 microcontroller is shown in Figure 11, showcasing that each key byte has a correlation peak in a different location.

In Figure 12 the Output Vs. Time plot for a 1 round/cycle AES implementation is shown. As can be seen, all correlation peaks are at the same time.

6.4.2 Windowing

Sometimes, with hardware AES attacks, so-called ”ghost peaks” will occur. These peaks appear at different time points than the operation we are attacking,

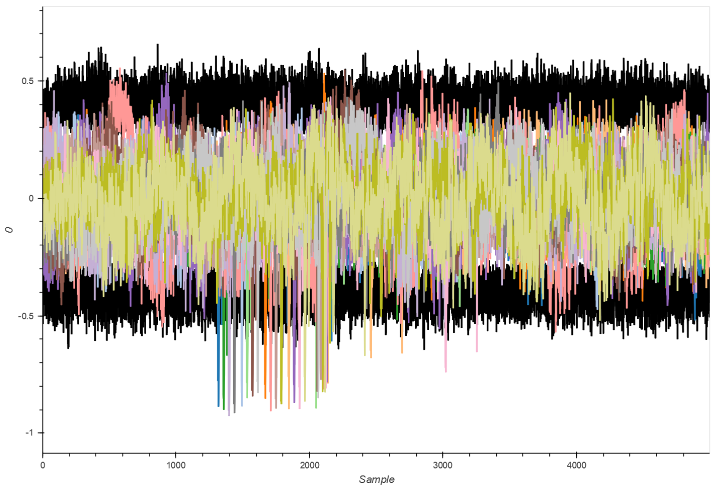


Figure 11: STM32F3 Output Vs. Time Plot

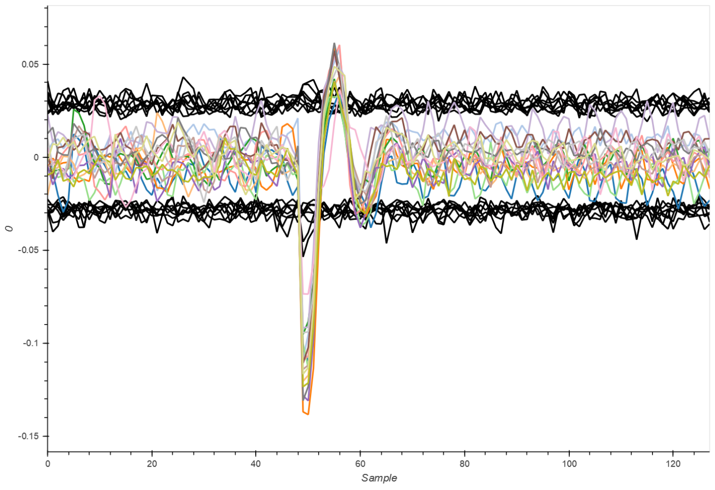


Figure 12: CW305 Output Vs. Time Plot

with high correlation with an incorrect key guess. As such, it can be important to window trace data to remove areas with ghost peaks. If you know the key, this easy using the output vs. time plot. However, for an attack with the correct knowledge, these false peaks can be easily identified as well. For example, an attacker might use the output vs. time plot to look for:

- Correlation peaks occurring outside the encryption window. Often this window can be identified by a clear pattern of 10 rounds (AES128) or 14 rounds (AES256)

- In some cases, ghost peaks will become less of an issue as more traces are collected.
- Locations where correlation is only present for some of the key bytes

6.4.3 PGE vs. Traces

The PGE vs. traces plot can be useful for figuring out how many traces were required to break an AES implementation. The following code will display the partial guessing entropy (PGE) for each key byte on the Y-axis, and the number of traces on the X-axis:

```
ret = plot_data.pge_vs_trace(0)
curve = hv.Curve((ret[0],ret[1]), "Traces Used in Calculation",
                                  "Partial Guessing Entrop of Byte")
for bnum in range(1, 16):
    ret = plot_data.pge_vs_trace(bnum)
    curve *= hv.Curve((ret[0],ret[1])).opts(
                                    color=byte_to_color(bnum))
curve.opts(width=900, height=600)
```

ChipWhisperer versions 5.3.2 and above can simply do the following for the same effect if using Jupyter:

```
plot_data.plot_pge_vs_traces()
```

For example, assume 100k traces are captured on a device. After analyzing them with ChipWhisperer, you recover the entire key. However, you may also want to know if this is possible at a lower number of traces, say 10k. Rather than running the attack again with a lower number of traces, you can inspect the PGE vs. Traces plot to see when the PGE for all the key bytes reached zero.

Figure 13 shows that although 50 traces were captured, 30 or fewer were required (the plot has a resolution of 10 traces) were required to break the implementation.

6.4.4 Correlation Vs. Traces

As mentioned previously, PGE is a metric only available if you have the key, not if you're an attacker. So, for example, while you might technically be able to break an AES implementation in 10k traces, an attacker won't know they've broken the key until they try it. Instead, they might use the difference in correlation between the best guess and the next best one to know when they've recovered a byte of a key. The Correlation Vs. Traces can be useful for understanding for when an attacker is sure to have broken a key. The following code will plot the correct key in multiple colours and the next best key in black:

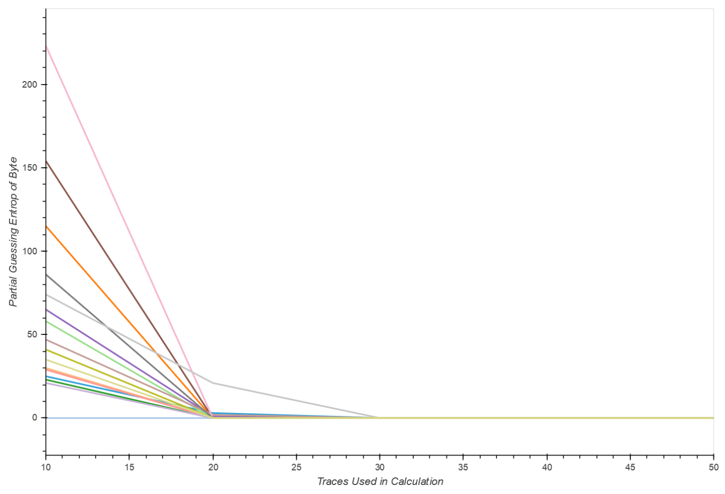


Figure 13: STM32F3 PGE Vs. Traces

```
a = []
b = []
for bnum in range(0, 16):
    data = plot_data.corr_vs_trace(bnum)
    best = [0] * len(data[1][0])
    for i in range(256):
        if i == key[bnum]:
            a.append(np.array(data[1][i]))
        else:
            if max(best) < max(data[1][i]): best = data[1][i]
    b.append(np.array(best))

pda = pd.DataFrame(a).transpose().rename(str, axis='columns')
pdb = pd.DataFrame(b).transpose().rename(str, axis='columns')
curve = hv.Curve(pdb['0'].tolist(), "Iteration Number",
                "Max Correlation").options(color='black')
for i in range(1,len(pdb.columns)):
    curve *= hv.Curve(pdb[str(i)]).options(color='black')

for i in range(len(pda.columns)):
    curve *= hv.Curve(pda[str(i)]).options(color=byte_to_color(i))

curve.opts(width=900, height=600)
```

ChipWhisperer versions 5.3.2 and above can simply do the following for the same effect if using Jupyter:

```
plot_data.plot_corr_vs_traces()
```

Figure 13 shows that TinyAES128C was broken by trace 30. Figure 14 shows that an attacker would clearly know by trace 30 that they had recovered the AES key.

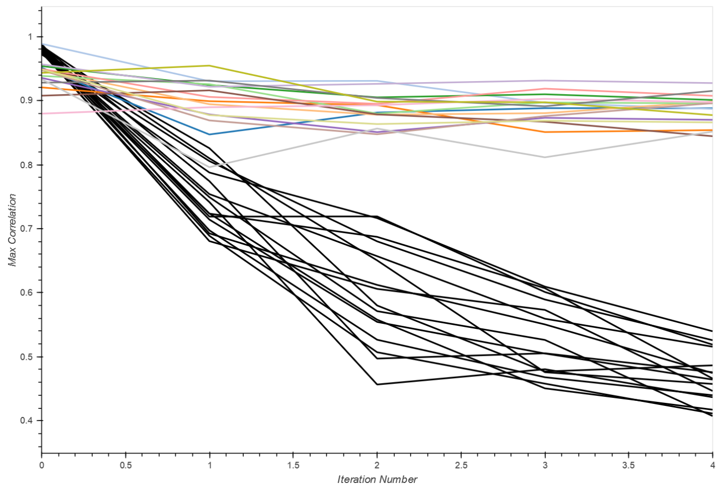


Figure 14: STM32F3 Correlation Vs. Traces

6.5 Faster Analysis Libraries

ChipWhisperer Analyzer was not built for simplicity and ease of use above speed. As such, there are a few CPA analysis libraries available that offer large speed increases over ChipWhisperer. Two, LASCAR and SCARED, will be shown in this section.

6.5.1 LASCAR

LASCAR, an open source Python library from Ledger, is a fully featured side channel analysis tool. It has much better performance than ChipWhisperer and is pure Python, meaning it has good compatibility with different operating systems. We'll be focusing on only its CPA attack capabilities, though it is capable of much more, such as classic DPA attacks and TVLA tests. ChipWhisperer has some basic compatibility functions and classes to make using data collected with ChipWhisperer with LASCAR as easy as possible. Instructions for how to install LASCAR are available on its Github Page.

A LASCAR project/container can be created as follows:

```
import chipwhisperer.common.api.lascar as cw_lascar
from lascar import *
cw_container = cw_lascar.CWContainer(project, project.textins)
```

The second argument for this container should be the known information from the project. For some leakage models, this is the plaintext, but for others it is the ciphertext.

An attack class list (one per byte) can be created follows:

```
leakage_model = cw_lascar.sbox_HW_gen
cpa_engines = [CpaEngine("cpa_%02d" % i,
                leakage_model(i, range(256))) for i in range(16)]
```

Three LASCAR compatable leakage models are available: `sbox_HW_gen`, `sboxInOut_HD_gen`, and `lastround_HD_gen`, corresponding to ChipWhisperer's `sbox_output`, `sbox_in_out_diff`, and `last_round_state_diff`, respectively. See LASCAR's CPA example to see how to create a different leakage model.

Once the attack is created, it can be run as follows:

```
session = Session(cw_container,
            engines=cpa_engines).run(batch_size=50)
```

See LASCAR's documentation for how to extract useful mesaures from the attack. ChipWhisperer also includes a class to display results in the same table as ChipWhisperer's analysis results:

```
disp = cw_lascar.LascarDisplay(cpa_engines, list(project.keys[0]))
disp.show_pge()
```

6.5.2 SCARED

Like LASCAR, scared is a general side channel analysis library (though again we'll just be focusing on its CPA capabilities). Unlike LASCAR, it's got C++ acceleration. This has the advantage of providing a sizable speedup over LASCAR, but has the disadvantage of limiting its compatibility with different operating systems. At the time of writing, SCARED only supports Linux and OSX. To install on Windows, you must have the Microsoft C compiler installed to build the C accelerated functions.

ChipWhisperer doesn't have any builtin compatibility functions for SCARED, but using your ChipWhisperer data with SCARED is easy. To put your ChipWhisperer traces in a SCARED compatible format:

```
import estraces, scared, numpy as np
cw_traces = estraces.read_ths_from_ram(np.array(proj.waves),
                            plaintext=np.array(proj.textins),
                            ciphertext=np.array(proj.textouts))
```

The known data name here is actually important, as it will need to match up to the name of the known variable in the leakage model. For example, the following leakage model will use the `plaintext/proj.textins` variable in `cw_traces`:

```python
@scared.attack_selection_function
def sbox_output(plaintext, guesses):
    res = np.empty((plaintext.shape[0], len(guesses),
            plaintext.shape[1]), dtype='uint8')
    for i, guess in enumerate(guesses):
        res[:, i, :] = scared.aes.sub_bytes(
                        np.bitwise_xor(plaintext, guess))
    return res
```

while the following will use the `ciphertext/proj.textouts`:

```python
@scared.attack_selection_function
def last_round_state_diff(ciphertext, guesses):
    res = np.empty((ciphertext.shape[0], len(guesses),
            ciphertext.shape[1]), dtype='uint8')
    for i, guess in enumerate(guesses):
        res[:, i, :] = np.bitwise_xor(scared.aes.sub_bytes(
                        np.bitwise_xor(ciphertext, guess)),
                        scared.aes.shift_rows(ciphertext))
    return res
```

Once you have your traces and leakage model, the attack can be run:

```python
container = scared.Container(cw_traces)
a = scared.CPAAttack(selection_function=sbox_output,
                    model=scared.HammingWeight(),
                    discriminant=scared.maxabs)

a.run(container)
```

There are two primary pieces of feedback that SCARED gives about the attack. The first is `a.scores`, which gives the maximum correlation for each guess for each key byte. For example:

```python
# max correlation for key guess 0x2b
# for byte 0
a.scores[0x2b][0]

# best guess for full key
key_guess = np.argmax(a.scores, axis=0)
```

The second is `a.results`, which gives the full correlation data for each key guess, key, and point. For example:

```python
# correlation for guess 0x2b
# for key byte 0
# at point 1317
a.results[0x2b][0][1317]
```

7 Performing TVLA Testing

As you've seen, full CPA attacks can be very complicated; selecting the wrong leakage model or misinterpreting the results can give a false sense of security for a vulnerable AES implementation. We can make investigating side channel leakage easier by asking a broader question. Instead of trying to fully break the implementation, we can assess whether or not there is measurable leakage. One method of detecting the presence of measurable leakage is the Test Vector Leakage Assessment (TVLA). The basic idea is as follows:

1. Collect a trace dataset with specific requirements. These requirements vary and will be discussed later
2. Partition the dataset into two groups, G_1 and G_2, based on specific requirements.
3. Split each group in half: For example, G_1 will become G_{1A} and G_{1B}.
4. Use Welsh's T-Test to test whether or not G_{1A} and G_{2A} have different means. The T-Test should be computed at each point along the traces. Repeat with G_{1B} and G_{2B}. If $\sigma \geq 4.5$ or $\sigma \leq -4.5$ for both T-Tests at the same point, the target is considered to have failed the TVLA.

There are a variety of methods to collect and partition the data. An easy to apply and powerful one is the Fixed Vs. Random Text dataset. Here the trace dataset is made up of two sets: The first, fixed dataset, has the following settings for AES128, using I_{fixed} for the plaintext and K_{dev} for the key:

$$I_{fixed} = \texttt{0xda39a3ee5e6b4b0d3255bfef95601890}$$
$$K_{dev} = \texttt{0x0123456789abcdef123456789abcdef0}$$

The second, random dataset, has the following settings, using I_j for the plaintext and K_{dev} for the key:

$$K_{gen} = \texttt{0x123456789abcdef123456789abcde0f0}$$
$$I_0 = \texttt{0x00000000000000000000000000000000}$$
$$I_{j+1} = \mathtt{AES}(\mathtt{I_j}, \mathtt{K_{gen}})$$
$$K_{dev} = \texttt{0x0123456789abcdef123456789abcdef0}$$

For best results, intersperse capture of the fixed and random datasets. For the two half groups, it is sufficient to split them in half chronologically.

The following document from Rambus contains details on AES192 and AES256 Fixed Vs. Random Text TVLA, as well as other TVLA datasets: `https://www.rambus.com/wp-content/uploads/2015/08/TVLA-DTR-with-AES.pdf`. Compared to other TVLA tests, Fixed Vs. Random Text is very good at detecting general side channel leakage. It, however, can pick up undesired data as well.

For example, the device loading plaintext will often show up very strongly in the T-Test; however, this leakage isn't useful to an attacker and is therefore a false positive. Due to this, can be advantageous to plot the data from the T-Test. A TVLA test on AES running on the CW305 can be seen in Figure 15, showcasing that the unprotected core has clearly visible side channel leakage.

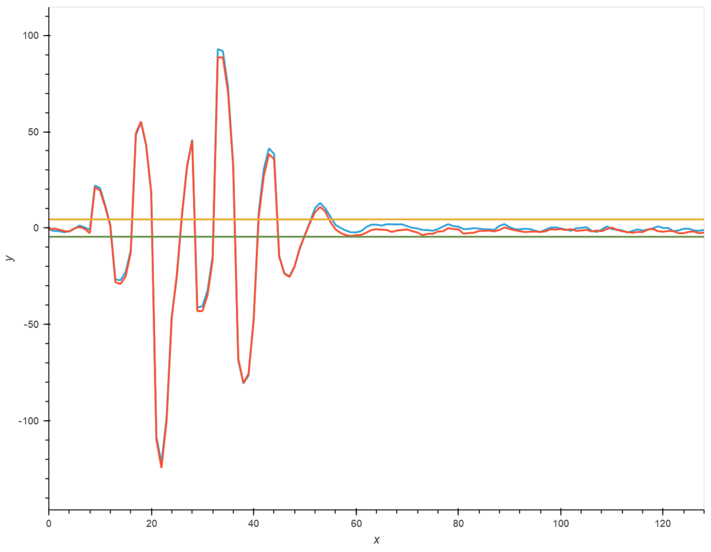


Figure 15: CW305 Fixed Vs. Rand Text TVLA

8 Porting your Own AES Core

While we provide a reference bitfile with an AES implementation for the CW305, ultimately the purpose of the CW305 platform is to host your own implementation, be it AES or something else, in order to study its resiliency to side-channel attacks.

In addition to providing a defined target for our tutorials which show side-channel attacks on hardware implementations of cryptographic algorithms, our reference target design provides a worked example of how to interface a hardware AES core with the ChipWhisperer capture hardware and software.

The reference target is structured in a way to make it easy to modify for different target cores and requirements. The hierarchical structure of the design, along with high-level connectivity, is shown in Figure 16.

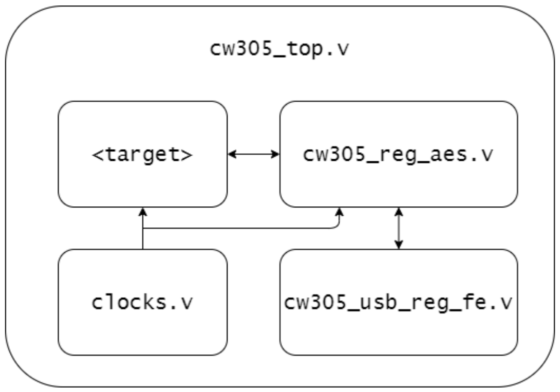


Figure 16: Verilog hierarchy.

In addition to the top-level `cw305_top.v` wrapper, there are three Verilog modules:

- `cw305_usb_reg_fe`: Front-end to the SAM3U USB interface; provides a simple interface to the register block. This module was created in order to keep all of the logic specific to the SAM3U USB interface out of the register block. If you are not changing the SAM3U firmware, you should not need to modify this module.
- `clocks`: Routing of internal and external clocks, controlled by the J16 and K16 DIP switches and by a register. If you do not require any changes to the clocking options provided by the reference design, you should not need to modify this module.
- `cw305_reg_aes`: Register block. All the control and status registers that the ChipWhisperer software interacts with in order to control the target are located here.

8.1 Clock Domains

The reference CW305 design has two clock domains: the USB clock domain, for all the control and status registers that ChipWhisperer software interacts with directly; and the cryptography clock domain, for the target logic. The main purpose of having these two clock domains is to allow the USB clock to be disabled when collecting power measurements, which is useful for avoiding the hassle of filtering out the USB clock noise from captured traces.

The second purpose of the two clock domains is to allow some flexibility for the cryptography clock's frequency. The USB clock is fixed at 96 MHz; the cryptography clock can, in theory, be anything you want it to be. In practice, constraints on the cryptography clock are imposed by:

- Your capture equipment: the ChipWhisperer-Lite and Pro have a restricted range of supported sampling rates.
- How fast you can implement the target cryptography logic.
- The clock ranges supported by any PLLs you require.

The reference design includes clock-domain-crossing logic so that the Vivado implementation runs cleanly, without any timing violations.

8.2 Register Block

When porting a new AES core to the CW305, the `c305_reg_aes` register block is where most (if not all) modifications should be required. It is a simple module and should be easy to understand and modify. All registers are straightforward, with the exception of the `REG_CRYPT_GO` register. Writing this register launches the target operation, and reading it indicates whether the operation is done or not; the logic around this register is a bit more tricky because the USB clock may be disabled during the target operation, and because the target may also be launched via a separate control signal.

8.2.1 Register Addressing

Register address definitions are located in `cw305_defines.v`. When connecting to the target using `CW305.py`, this Verilog defines file is automatically parsed so that registers may be refered to by their Verilog `‘define`’d name. For example, the key register can be referenced as `target.REG_CRYPT_KEY` in Python.

The USB interface between the SAM3U and the FPGA has an 8-bit data bus and a 21-bit address bus. In order to facilitate reading and writing the long cryptographic inputs and outputs from Python, the address is composed of a 14-bit register identifier (most significant bits) and a 7-bit byte count offset (least significant bits); every register can therefore contain up to 128 bytes. The byte count offset width can be adjusted via the `pBYTECNT_SIZE` parameter in Verilog, and via the `bytecount_size` property in `CW305.py`. `cw305_reg_aes.v` contains several examples of both single- and multi-byte registers.

8.2.2 Interface Signals

The waveform in Figure 17 shows the timing requirements for the register block's interface with the `cw305_usb_reg_fe` block:

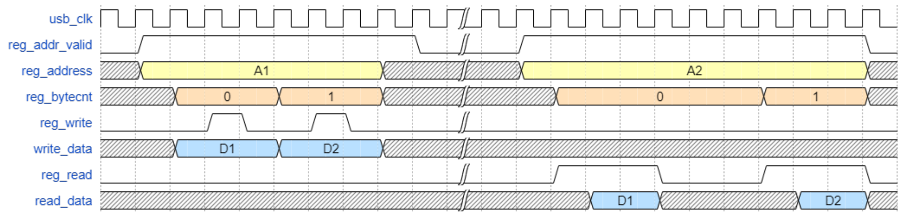


Figure 17: Register read and write timing.

The waveform in Figure 18 shows the timing requirements for the register block's interface with the target core. There are two options for the `I_done` signal timing; this is controlled by the `pDONE_EDGE_SENSITIVE` instantiation parameter. The reference design uses version 1, which corresponds to `pDONE_EDGE_SENSITIVE = 1`. The USB clock is not shown here, but keep in mind that it's possible for the USB clock to be disabled during the target operation. The USB clock can be re-enabled at any time, either during the target operation, or after its completion.

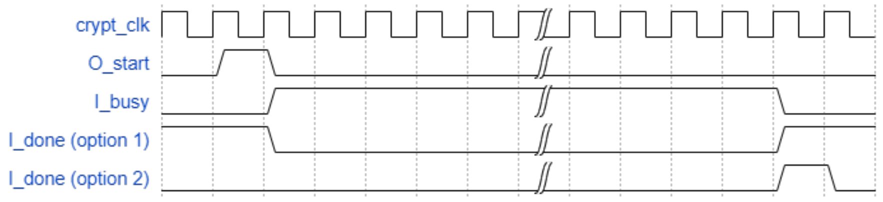


Figure 18: Go, done, and busy timing.

The `I_busy` signal should be connected to the 20-pin connector's `IO4` (trigger) line, and serves two purposes: its rising edge sets off the synchronized start of the power trace capture, and the number of cycles that it's held high is measured and available to Python to easily learn exactly how many clock cycles the target operation ran for.

8.2.3 Making it Go

There are two mechanisms to launch the target operation:

1. Write the `REG_CRYPT_GO` register.
2. Set the `usb_trigger` input high.

The second option exists because the disabling of the USB clock is done from Python; once the USB clock is disabled, it's impossible to write the `REG_CRYPT_GO` register, so the second mechanism must be used.

8.3 Testing

A basic Verilog testbench is provided here:
`hardware/victims/cw305_artixtarget/fpga/vivado_examples/aes128_verilog/sim/`
Simply run "make" to execute it. By default, Icarus Verilog ("iverilog") is used, but the makefile can easily be adapted to your simulator of choice. The testbench is by no means exhaustive; it only serves to verify basic operation and signs of life (also, any syntax errors tend to be picked up much faster with this than they are by Vivado!). Simulation waveforms may be viewed with gtkwave.

8.4 Other External Interfaces

The reference target uses the USB interface via the SAM3U. There are other options. If the target incorporates a soft core processor (for example, see our DesignStartTrace repository), it may make sense to use a UART interface instead; the 20-pin connector's `IO1` and `IO2` lines are available to the FPGA for this purpose. The CW305's JP3 header has several more FPGA pins broken out to it to allow even more interface possibilities.

A standard JTAG header can be broken out from these (an adapter board is available) for connecting of tools such as a J-LINK Segger to the soft-core processor.

8.5 Other USB Interfaces

The default firmware on the CW305 board assumes the external memory bus is used to transfer data to registers inside the FPGA implementation. As a convenience, we also allow you to override these pins to use them as GPIO pins via the `FPGAIO` module.

☞ Using the `FPGAIO` module implies you are *not* using the data bus feature or register block discussed in Section 8.2, but instead defining your own interface and high-level Python control module.

These GPIO pins can implement almost any interface from the Python control computer, including a SPI interface that can be mapped to any of the interconnect pins. See the FPGAIO class documentation for more details. An example of the usage is shown here:

```python
fpga = cw.target(None, cw.targets.CW305)
io = target.gpio_mode()

# Example - toggle pin associated with FPGA pin C1 (USB_A11)
import time
io.pin_set_output("C1")
io.pin_set_state("C1", 0)
time.sleep(0.1)
io.pin_set_state("C1", 1)
```

```
# Setup a SPI interface based on schematic net names
io.spi1_setpins(mosi="USB_A20", miso="USB_A19", sck="USB_A18", cs="USB_A17")
io.spi1_enable(True)

somedata = [0x11, 0x22, 0x33]

response = io.spi1_transfer(somedata)
print(response)
```

8.6 Example of non-AES Core

For an example of a different target core, refer to ECDSA target here:
`hardware/victims/cw305_artixtarget/fpga/vivado_examples/ecc_p256_pmul/`.

On the Python side, this target required new functionality, which was done by extending the `CW305.py` class with `CW305_ECC.py`.

9 Alternative CPA Across MixColumns

While the typical last round state diff leakage model will work for many hardware implementations of hardware AES, it is important to note that this attack will not work for all implementations. In particular, if the ciphertext is not stored in the state register, the earliest possible difference in that register will be between the 9th and 8th rounds, which involves a MixColumns, a linear combination of 4 state bytes. This will change the CPA attack from 256 possible key bytes on each of the 16 key bytes to 4,294,967,296 possible values of 4 key words. There is also a nonlinear mix of another round of the AES key, further complicating the attack. The beginning of the encryption is not much better since the MixColumns operation is present between rounds 1 and 2 as well.

This might appear to completely thwart our attack, but we can actually do a modified CPA attack on the beginning of the encryption to completely recover the key. First detailed in,[2] we begin by examining the MixColumns operation:

$$\begin{bmatrix} s'_0 \\ s'_1 \\ s'_2 \\ s'_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix}$$

With s' being the output of MixColumns and s being the input. Note that s is also the output of the SubBytes operation (we're ignoring ShiftRows here since it just moves bytes around). Again, the problem here is that each byte of s' is comprised of 4 bytes of s and therefore 4 bytes of the key. For example, for s'_0:

$$s'_0 = 2s_0 + 3s_1 + s_2 + s_3$$

Next, let's examine the case where 3 bytes of s are constant and one is variable:

$$\begin{bmatrix} s'_0 \\ s'_1 \\ s'_2 \\ s'_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} v \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

s'_0 then becomes:

$$s'_0 = 2v + 3c_1 + c_2 + c_3 = 2v + c_a$$

Now, instead of s'_0 depending on 4 independent bytes, it now only depends on 2, bringing the search space for the attack down from 2^{32} to 2^{16}. This also

[2] Amir Moradi and Tobias Schneider. "Improved Side-Channel Analysis Attacks on Xilinx Bitstream Encryption of 5, 6, and 7 Series". en. In: *Constructive Side-Channel Analysis and Secure Design*. Ed. by François-Xavier Standaert and Elisabeth Oswald. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 71–87. ISBN: 978-3-319-43283-0. DOI: 10.1007/978-3-319-43283-0_5.

allows the attack to extend past the next AddRoundKey, since the XOR of the next round key, a constant, can be incorporated into c_a.

While this seems very promising, it suffers from a few major issues:

1. 2^{16} is still a large search space
2. Since c_a is mixed linearly with v, the difference in correlation between a correct and incorrect value for c_a is very low
3. This linear mix also results in ghost peaks

A better idea is to attack v one bit at a time. This is very similar to a single bit DPA attack, except we're still using correlation instead of the difference. Since c_a is constant, this will result in two scenarios:

1. The targeted bit of c_a is 0, causing the correlation to be negative (as in a normal CPA attack)
2. The targeted bit of c_a is 1, causing the correlation to be positive (opposite to a normal CPA attack)

Taking the absolute value of this correlation then completely removes the effect of c_a, bringing us back to a search space of 2^8 and removing the hard to distinguish linear mix of v and c_a.

The final hurdle that must be overcome is that a single bit is not enough information for a CPA attack. This can be solved by repeating the single bit attack on each bit of s'_0, as well as on $s'_1 = v + c_b$, $s'_2 = v + c_c$, and $s'_3 = 3v + c_d$; a total of 32 CPA attacks. These absolute correlations are then summed together and this sum is used to distinguish the correct key.

The attack can be done at the same time on the other columns, recovering 4 key bytes in total. 3 more acquisition campaigns will then be needed to recover the final 12 bytes of the key.

9.1 Running the Attack

Since the details of this attack can be fairly complicated, ChipWhisperer includes a special key text pair and attack object (requires SCARED). A typical trace acquisition looks like:

```
from tqdm.autonotebook import trange
ktp = cw.ktp.VarVec('row')
key, pt = ktp.next()
N = 5000
projects = []

for cmpgn in trange(4):
    project = cw.create_project(f"Var_Vec_{cmpgn}", \
                                overwrite=True)
    projects.append(project)
    for i in trange(N, leave=False):
        ktp.var_vec = cmpgn
        key, text = ktp.next()
        trace = cw.capture_trace(scope, target, text, key)
        if trace is None:
            continue
        project.traces.append(trace)
    project.save()
```

While the analysis can be done as follows:

```
from chipwhisperer.analyzer.attacks.attack_mix_columns \
                            import AttackMixColumns
attack = AttackMixColumns(projects, vec_type='row', hd=False)
results = attack.run(n_traces=None, trace_slice=None)
```

Two important parameters of the attack are `vec_type` and `hd`. For `vec_type`, a value of 'column' will make a single column of the plaintext variable, while a value of 'row' will make a single row of the plaintext variable. Both result in a single byte of each MixColumn being variable; however, attack on some targets may work better with one instead of the other. For example, the reference CW305 AES implementation is more vulnerable to a variable 'row'. Ensure that this variable is matched between the acquisition and the final attack. If you forget, all key bytes except 4 will be `0xFF`.

The second parameter in the attack, hd, switches between using the Hamming distance between the plaintext and output of MixColumns (hd=True) and just the Hamming weight of the MixColumns (hd=False). Again, some implementations of AES will work better using the Hamming weight instead of the Hamming distance, while others, like the CW305 AES implementation, will have the opposite be true.

The attack can be easily windowed via the `trace_slice` parameter and you can use a smaller number of traces via the `n_traces` parameter.

9.2 Interpreting Results

To print the guessed key and the actual key:

```
print(bytearray(results["guess"]))
print(bytearray(projects[0].keys[0]))
```

To plot an output vs. time plot, with the correct byte in red and the best guess that isn't correct in green:

```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
```

```
plt.figure()
for i in range(0,16):
    c = results["corr"][i]
    maxes = np.max(c, axis=1)
    guess = np.argsort(maxes)[-1]
    guess2 = np.argsort(maxes)[-2]
    actual = projects[0].keys[0][i]
    x = np.argmax(c[actual])
    if guess != actual:
        plt.plot(c[guess], "g-")
    else:
        plt.plot(c[guess2], "g-")
    plt.plot(c[actual], "r--")
    plt.plot(x, c[actual][x], "ro")
    print(f"Best guess {hex(guess)} (corr={maxes[guess]}), \
        next best = {maxes[guess2]}, real = {maxes[actual]}")
plt.show()
```

An example output vs. time plot from an analysis windowed between sample 5 and 20 is shown in Figure 19.

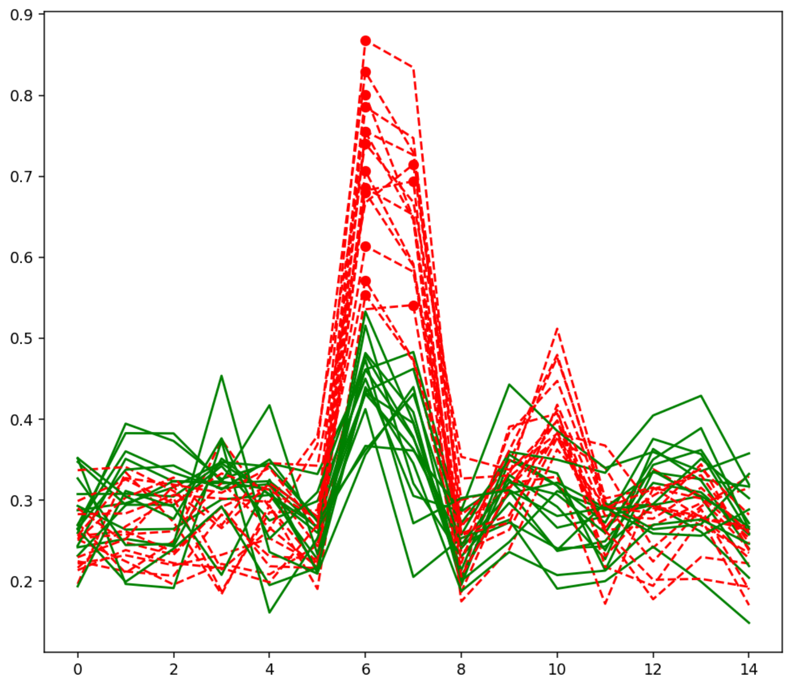


Figure 19: Mix Columns Output Vs. Time

10 Conclusions and Next Steps

Performing an power analysis attack on a FPGA implementation of AES can be accomplished with low-cost tools. This whitepaper has demonstrated how power analysis attacks work in theory, and then applied this to a FPGA implementation of an AES core.

This work has been done with the ChipWhisperer system, while also highlighting several other higher-performance open-source tools you can interface to ChipWhisperer for larger analysis work.

This setup can be used for a variety of other targets, including implementation of other hardware cores (ECC, SHA, etc), and implementation of soft-core processors such as Arm DesignStart or various RISC-V based cores.