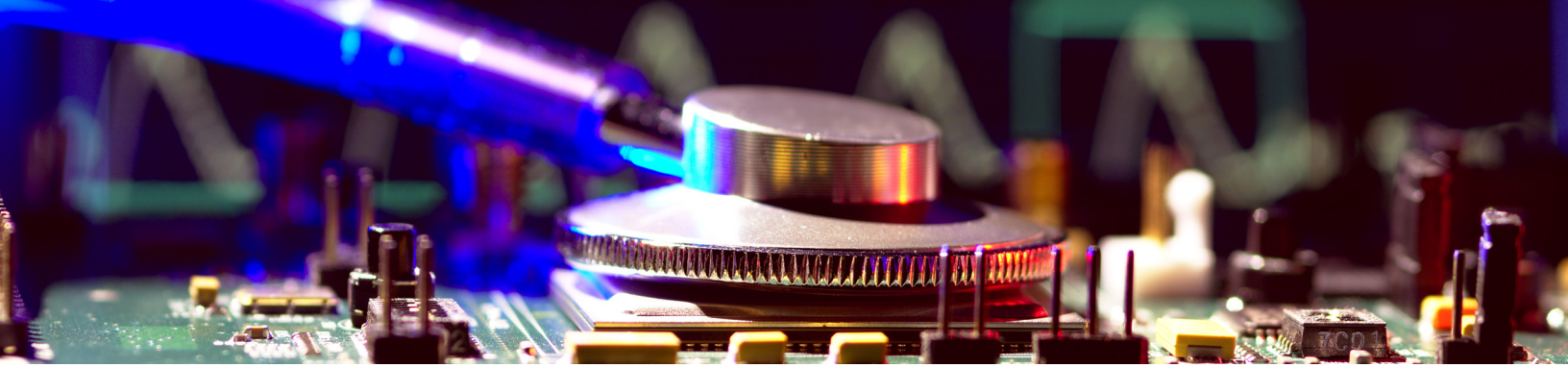


Flexible, Opensource workBench fOr Side-channel analysis (FOBOS)

FOBOS User Guide v2.0

Abubakr Abdulgadir **William Diehl** **Rajesh Velegalati**
Jens-Peter Kaps
[aabdulga, wdiehl, jkaps]@gmu.edu

August 3, 2018
George Mason University
Fairfax, Virginia

cryptography.gmu.edu

www.gmu.edu

Cryptographic Engineering Research Group
Department of Electrical and Computer Engineering George Mason University
3100 Engineering Building, 4400 University Drive, Fairfax, VA 22030-4444, USA
Voice: (703) 993-1561, Fax: (703) 993-1601

Contents

1 FOBOS Overview **4**
- 1.1 Introduction . . . 4
- 1.2 Hardware Setup . . . 4
- 1.3 Trace Acquisition . . . 4
- 1.4 Data Analysis . . . 4

2 Software Installation **6**

3 DUT Development **7**
- 3.1 Data Flow Description . . . 7
- 3.2 The DUT Wrapper - DUT interface . . . 7

4 FOBOS Hardware Configuration **10**
- 4.1 Introduction . . . 10
- 4.2 Hardware List . . . 10
- 4.3 Programming the Control Board . . . 10
  - 4.3.1 Programming Steps . . . 10
- 4.4 DUT Board Programming . . . 12
  - 4.4.1 Programming Steps . . . 14
- 4.5 Oscilloscope Configuration . . . 17
- 4.6 Connecting the Hardware . . . 17
  - 4.6.1 Nexys3 Control Board - Nexys3 DUT . . . 18

5 Test Vector Generation **19**
- 5.1 Test Vector Format . . . 19
  - 5.1.1 Supported commands . . . 19
  - 5.1.2 FOBOS Protocol Example . . . 19
- 5.2 Using the blockCipherTVGen.py script . . . 20

6 Data Acquisition **21**
- 6.1 Data Aquisition Configuration . . . 21
  - 6.1.1 General Settings . . . 21
  - 6.1.2 Control Board Settings . . . 21
  - 6.1.3 Trigger Settings . . . 22
  - 6.1.4 Data Settings . . . 22
  - 6.1.5 Capture Settings . . . 23
  - 6.1.6 Oscilloscope Settings . . . 23

6.2 Sample Configuration File . . . 24
6.3 Running Data Acquisition . . . 25

7 Correlation Power Analysis 26
7.1 Steps to perform CPA using FOBOS . . . 26
7.1.1 Hypothetical power calculation . . . 26
7.2 CPA configuration . . . 26
7.2.1 Data Analysis Parameters . . . 26
7.2.2 Sample Space Disposition . . . 27
7.2.3 Compression Parameters . . . 27
7.2.4 Sample Analysis Configuration Files . . . 27
7.3 Running Data Analysis . . . 28

8 Welch's T-test 29
8.1 Background . . . 29
8.2 Test vector generation . . . 30
8.3 T-test module cofiguration . . . 30
8.4 Performing a t-test . . . 31

9 Using the FOBOS Profiler 32
9.1 FOBOS trigger settings . . . 32
9.2 Generating State Files . . . 32
9.3 Profiler configuration . . . 33
9.4 Running the profiler . . . 33
9.5 Using the plot_t_values.py script . . . 34

10 Power Measurement 35
10.1 Trace collection . . . 35
10.2 Power Measurement Configuration . . . 35
10.3 Running the Power Measurement Script . . . 35

A Function Descriptions 38
A.1 FOBOS - Analysis Module . . . 38
A.2 FOBOS - Capture Module . . . 38
A.3 FOBOS - Other Functions . . . 38

1 FOBOS Overview

1.1 Introduction

FOBOS is a Side-Channel Analysis (SCA) platform that is flexible, open-source and includes tools needed for power data acqusition and analysis. In this chapter, we briefly discuss FOBOS features. To get more details about any feature, please refer to the corresponding chapter in this document.

1.2 Hardware Setup

FOBOS hardware consist of the following components:

1. The control board (A standard FPGA board)
2. The DUT board (A standard FPGA board). This board hosts the DUT Wrapper and the DUT.
3. An oscilloscope (Agilent DSO6054A)
4. Power supply
5. Clock generator
6. Control PC
7. Current probe

FOBOS control and DUT boards are standard FPGA boards that need to be configured. FOBOS control board hardware description is provided along with a DUT wrapper. However, DUT is user provided. FPGA boards need to be connected to other components. For details, please refer to Chapter 3 and Chapter 4.

1.3 Trace Acquisition

After FOBOS software and hardware are prepared, operations (e.g. encryptions) can be run on the DUT board and traces collected. This process is called Data Acquisition. To run Data Acquisition, user must prepare test vectors, configure FOBOS and run the Data Acquisition script. Data Acquisition will send test vectors to FOBOS hardware and traces will be collected form the oscilloscope. For more information about this, please refere to Chapter 6

1.4 Data Analysis

FOBOS comes with multiple analysis modules including:

1. Correlation Power Analysis (CPA). See Chapter 7 for more information.
2. Leakage Assesment Using Welch's T-Test. See Chapter 8 for more information.
3. Time domain analysis that maps trace samples to internal DUT state (FOBOS Profiler). See Chapter 9 for more information.

2 Software Installation

FOBOS consists of Python scripts for acquisition and analysis and hardware for controller and DUT. This chapter describes installation of the software part (the Python scripts). Since Python is used, the software can run in Linux, Windows and other operating systems.

The following installation procedure was tested on Linux Ubuntu 16.04. It is assumed that Pytho2.7 is already installed.

1. Download FOBOS from `https://cryptography.gmu.edu/fobos/getfobos.php`
2. Extract the archive into the directory of your choice

   ```
   $ tar xvfz fobos-v1.0.tgz
   ```

3. Use the following commands to install the necessary Python packages:

   ```
   $ sudo apt-get install python-pip
   $ sudo pip install pycrypto
   $ sudo pip install numpy
   $ sudo pip install matplotlib
   $ sudo apt-get install python-tk
   $ sudo pip install scipy
   $ sudo pip install configparser
   ```

4. Install Digilent Adept from `https://reference.digilentinc.com/reference/software/adept/start?redirect=1#software_downloads` This will install libraries needed for the Digilent DEP protocol used for PC - Control Board communication.
5. Install Xilinx ISE. Make sure to install Cable Drivers in Xilinx ISE installation. If not installed, you will encounter issues using Impact to program the boards.

3 DUT Development

This chapter describes how to interface the DUT (Design Under Test) a.k.a victim, to the DUT Wrapper. The DUT Wrapper is included with FOBOS. However, the DUT is user provided. The DUT Wrapper handels communication to the Control Board and includes FIFOs to store input data for the DUT along with output FIFO.

3.1 Data Flow Description

Test vectors are sent form PC one at a time to the Control Board which stores them briefly in a FIFO. The PC sends a command indicating test vector is complete. This will initiates the process of sending the data from the Control Board to the DUT through the interface shown in the figure above. The DUT wrapper then puts data in the correct FIFOs (PDI, SDI and RDI). Once the DUT wrapper receives the start command from the controller, it de-assersts the reset signal and the DUT will run and use the data in the FIFOs. The output from the DUT is stored in the DO FIFO. Once the DO FIFO accumulates EXPECTED_OUTPUT bytes, the DUT wrapper will send this data to the control board which forwards it to the PC. This process repeats unitl all traces have been collected.

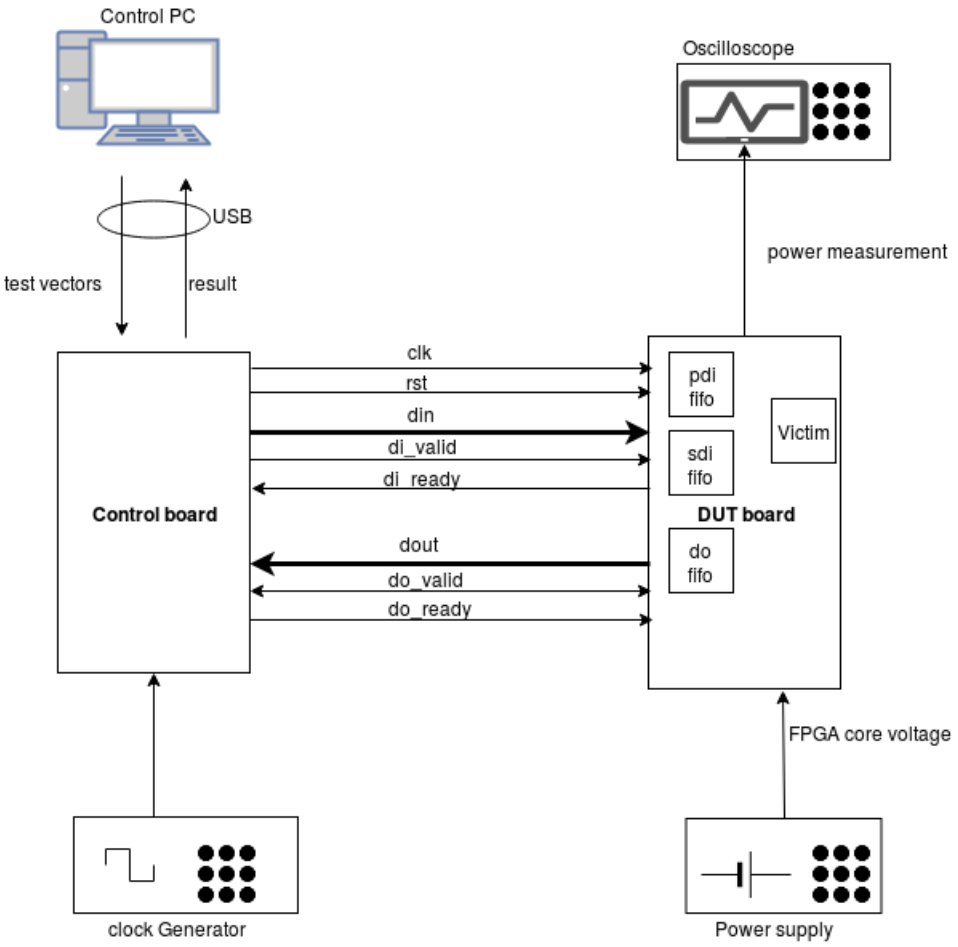
3.2 The DUT Wrapper - DUT interface

The protocol follows AXI stream protocol. The DUT (victim) is instantiated as follows in the FOBOS_DUT.vhd file. The following is an example of how the DUT is instantiated:

```
victim: entity work.victim(behav)
   -- Choices for W and SW are independently any multiple of 4, defined in generics above

   generic map  (
      G_W   => W, -- pdi and do width (mulltiple of 4)
      G_SW  => SW -- sdi width (multiple of 4)
   )
   port map(
      clk => clk,
      rst => start,  --! The FOBOS_DUT start signal meets requirements for
      --synchronous resets used in
      --! CAESAR HW Development Package AEAD
      --data signals
      pdi_data  => pdi_data,
      pdi_valid => pdi_valid,
      pdi_ready => pdi_ready,
```

Figure 3.1: FOBOS Hardware



```
      sdi_data => sdi_data,
      sdi_valid => sdi_valid,
      sdi_ready => sdi_ready,
      do_data => result_data,
      do_ready => result_ready,
      do_valid => result_valid

--! if rdi_interface for side-channel protected versions is
--- required, uncomment the rdi interface
--    ,rdi_data => rdi_data,
--    rdi_ready => rdi_ready,
--    rdi_valid => rdi_valid
);
```

There are four interfaces for four types of data that can be sent to the DUT:

1. Public Data Input (PDI): This is the data that the DUT will process. For example if the DUT is performing encryption, PDI will transfer plaintext.
2. Secret Data Input (SDI): Secret data interface. For example, encryption keys.
3. Random Data Input (RDI): Random data interface. This can be used when running protected ciphers that use masking.
4. Data Out (DO): Output data from the DUT. For example this can be ciphertext.

The generic W corresponds to the PDI and DO width in bits. The generic SW corresponds to the SDI width in bits.

It is highly recommended that the DUT is tested using the `sources/dut/fobos_dut_tb.vhd` test bench and ensure that the result data in the `do` port is correct. This testbench needs one test vector to be stored in the file `dinFile.txt` (See Chapter 5 for information about test-vector generation).

4 FOBOS Hardware Configuration

4.1 Introduction

FOBOS consists of Python scripts for acquisition and analysis and hardware for controller and DUT. This chapter describes the harware setup and descibes preparing and programming the controller and DUT boards. This chapter assumes familiarity with Xilinx ISE and Xilinx Imapct tools.

FOBOS hardware is composed of two boards and other devices. The first board is the Control Board which handles communication with the control PC, triggering the oscilloscope and clocking the DUT. The other board is the DUT which hosts the DUT Wrapper and the DUT. An oscilloscope captures the power traces and the control PC collects them for analysis. A power supply is used to power the DUT and a clock generator might be used to provide clock for the Control Board which provides it to the DUT.

4.2 Hardware List

1. The control board (A standard FPGA board)
2. The DUT board (A standard FPGA board)
3. An oscilloscope (Agilent Technologies DSO6054A)
4. DC Power supply (Agilent E3620A)
5. Clock generator (Instek SFG-2120 20 MHz)
6. Control PC (A standard PC running Linux)
7. Current probe (1mV/mA Tektronics CT-2 current sensing probe)

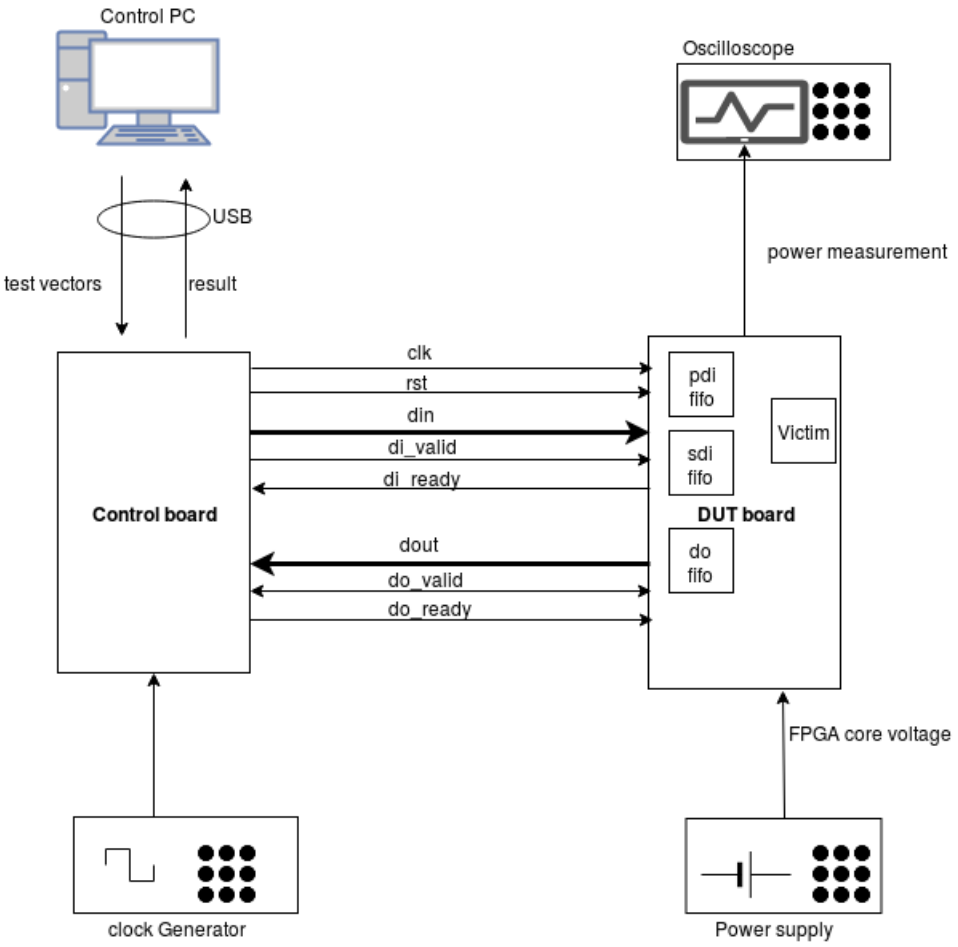
4.3 Programming the Control Board

The source VHDL files for the controller are located at `fobos/source/vhdl`. You need to use Xilinx ISE to generate the bitstream file and use Xilinx Impact to load the bitstream into the Control Board.

4.3.1 Programming Steps

1. Create a new project using Xilinx ISE. In the New Project wizard, set the Project Settings per the control board used. Make sure to select the values for Family, Device, Package and Speed (See Fig 4.2 for an example).

Figure 4.1: FOBOS Hardware



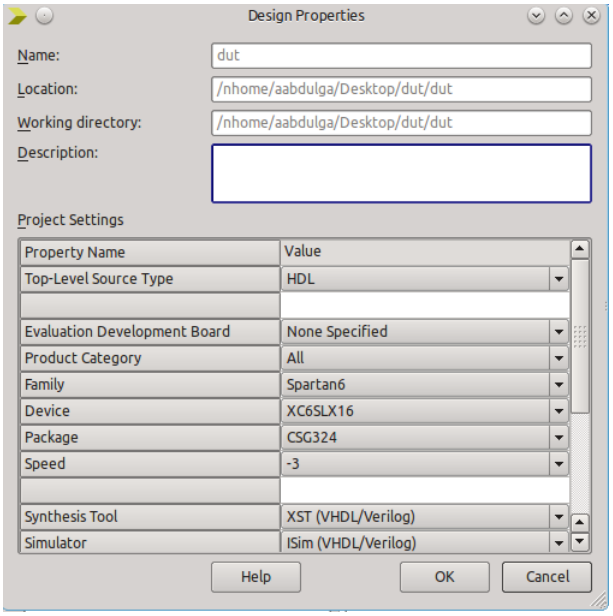


Figure 4.2: FOBOS Controller Desgin Properties

2. From the Project menu, select Add Source... and add all files from `$fobos/sources/common`.

3. Repeat the previous step to add all vhdl files from `$fobos/sources/vhdl/control`. Also, add the appropriate UCF file depending on the Control Board used (See Fig 4.3) .

4. Set the `fobosControlTopLevel` module as the top-level module for this project (See Fig 4.4).

5. Generate the programming bit file for the control board by clicking "Generate Programming File" in the Processes window.

6. Program the control board using Xilinx Impact. In the Processes window, click Configure Target Device (See Fig 4.5).

7. In the Impact window, click "Boundary Scan" then form the File menu, click "Initialize Chain" and assign the bit file to the FPGA. Now you may right-click the FPGA and click "Program".

4.4 DUT Board Programming

The DUT board hosts two components:

1. The DUT Wrapper : handles communication with the Control Board.

2. The DUT (Victim) : the Device-Under-Test is user provided. This section assumes that this component is developed, instantiated and tested. Please see Chapter 3 for more details.

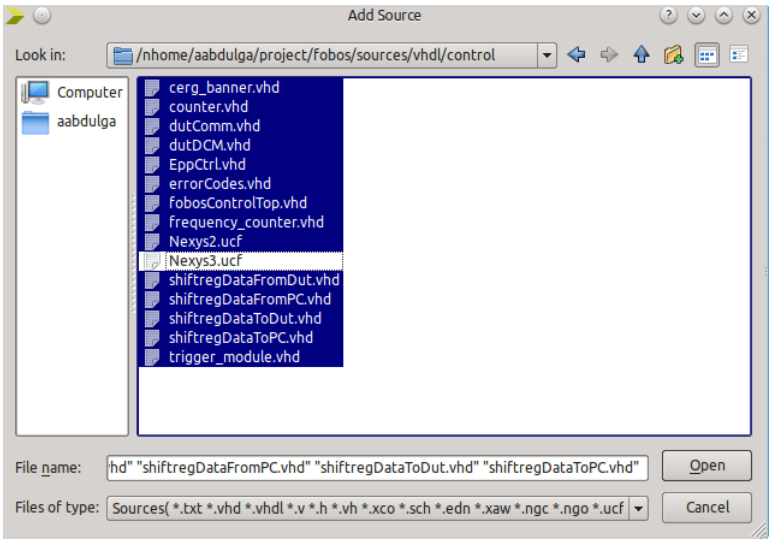


Figure 4.3: Adding Source Files to FOBOS Controller

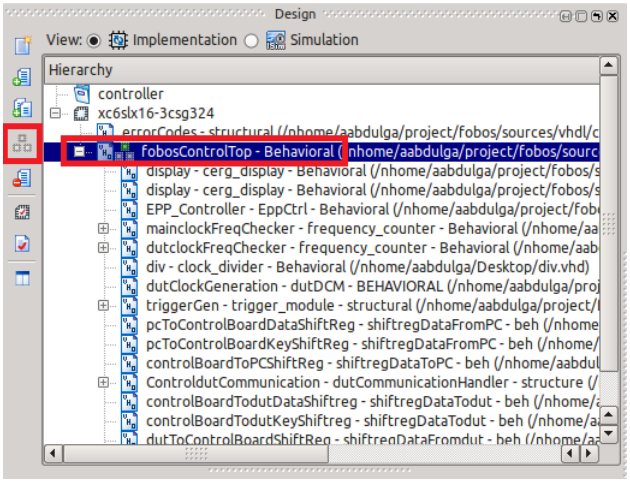


Figure 4.4: Setting Top-level Module

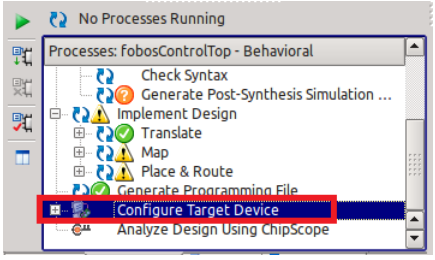


Figure 4.5:

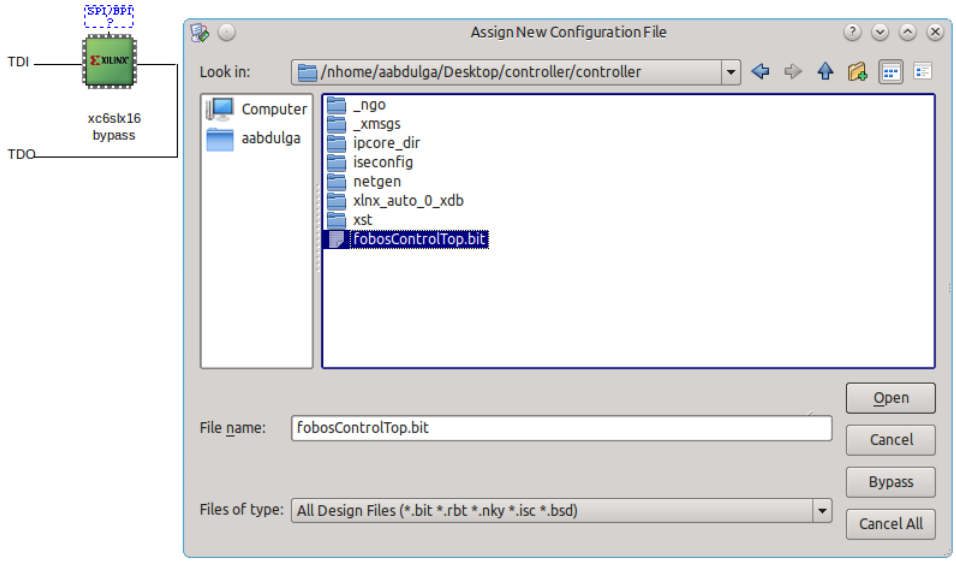


Figure 4.6: Progrmamming the Control Board

4.4.1 Programming Steps

1. Create a new project using Xilinx ISE. In the New Project wizard set the Project Settings per the DUT Board used. Make sure to select the values for Family, Device, Package and Speed (See Fig 4.7 for an example).
2. From the Project menu select Add Source... and add all files from `$fobos/sources/common`.
3. Repeat the previous process to add all vhdl files from `$fobos/sources/vhdl/DUT` and make sure to add the appropriate UCF file depending on the DUT board used (See Fig 4.8).
4. Add the DUT (victim) vhdl files to the project (user provided).
5. (Optional) You may choose to avoid using block RAMs in the implementation since this may affect attack difficulty.
   (a) Make sure to select the "Implementation" view.
   (b) Right-click the Synthesize-XST process.
   (c) In the Preocess Properties window, select HDL Options and select "Distributed" for the RAM Style property (See Fig 4.9).
   (d) Click OK.
6. Set `FOBOS_DUT` as the top-level module in this project.
7. Generate the programming bit file for the DUT by clicking "Generate Programming File" in the Processes window.
8. Program the DUT using Xilinx Impact. In the Processes window, click Configure Target Device (See Fig 4.11).

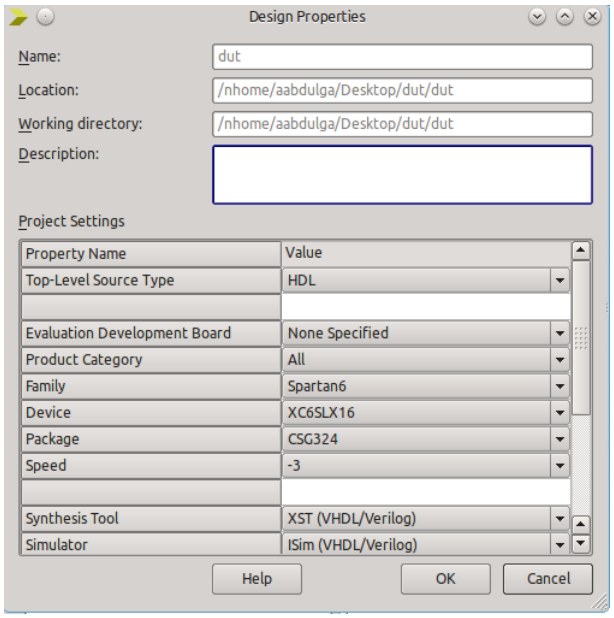


Figure 4.7: DUT Design Properties

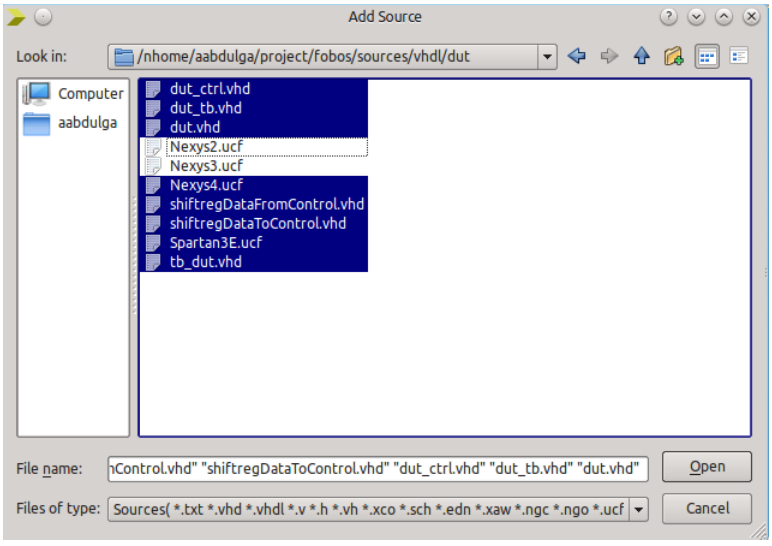


Figure 4.8: DUT Add Sources

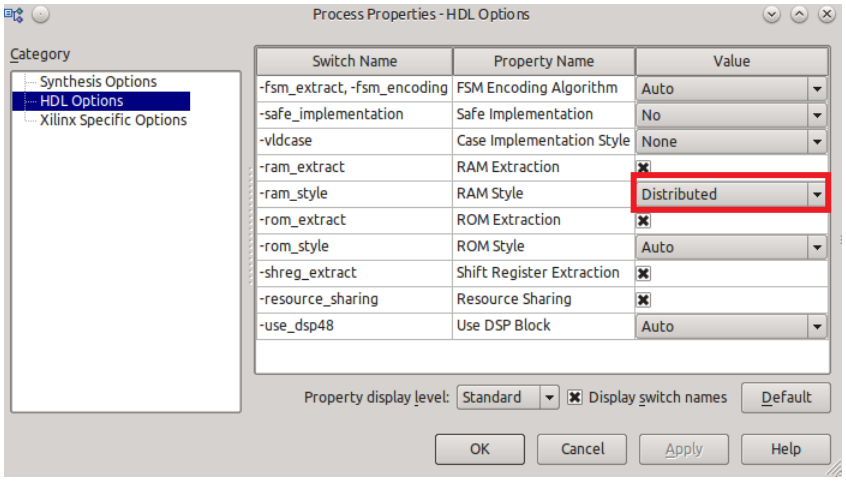


Figure 4.9: DUT RAM Style

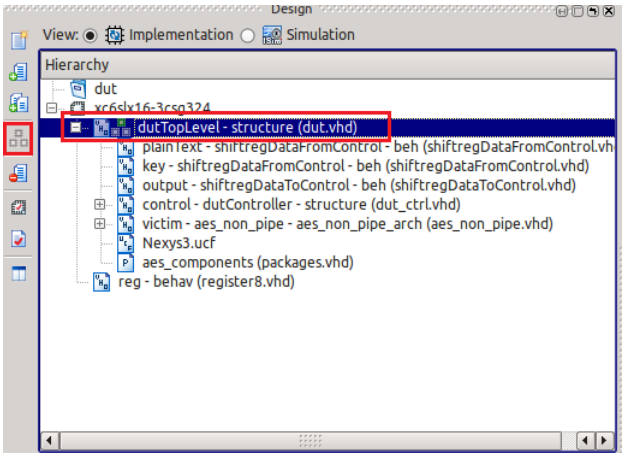


Figure 4.10: Set DUT Top-level

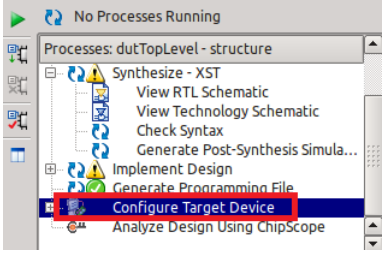


Figure 4.11:

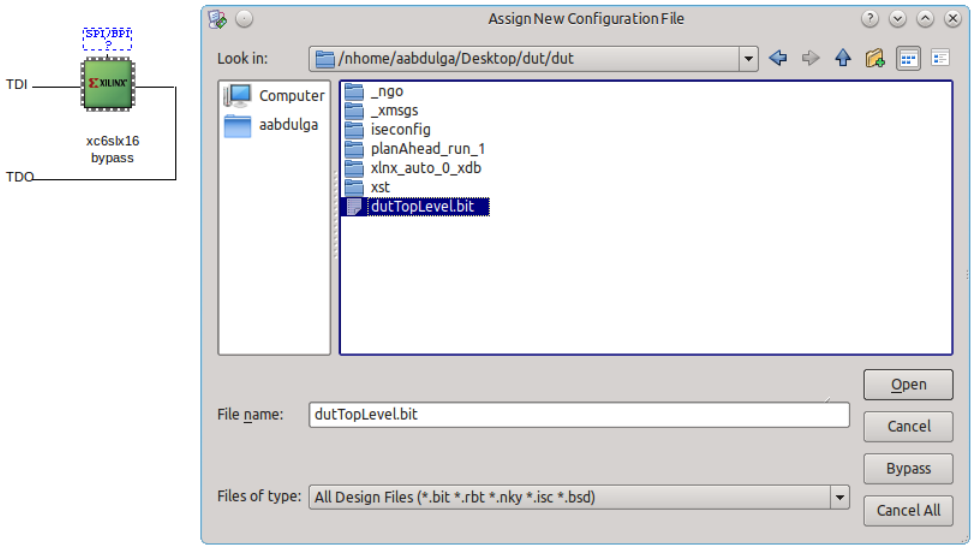


Figure 4.12: Progrmamming DUT

9. In the Impact window, click "Boundary Scan" then form the File menu, click "Initialize Chain" and assign the bit file to the FPGA. Now you may right-click the FPGA and click "Program".

4.5 Oscilloscope Configuration

Ocsilloscope is connected to the PC via Ethernet. IP configuration must be completed on the oscilloscope for the PC to be able to collect traces. The oscillopscope used in current setup is Agilent DSO6054A. To configure IP on this oscilloscope, please refer to vendor's documentation.

4.6 Connecting the Hardware

To connect FOBOS hardware, follow these steps:

1. Connect control board to power and ground.
2. Connect control board tirgger output the oscilloscope trigger channel.
3. Connect the control board to the DUT (see details below).
4. Connect the control board to the PC running FOBOS software using USB.
5. Connect clock generator to control board. Set the clock generator to desired clock.
6. Connect the current probe to the oscilloscope.
7. Connect the current probe to the DUT's ground.
8. Connect DUT to power making sure that the current probe is measuring the current.

9. Connect DUT to power supply ground.

The following subsections ilustrate how to connect different FPGA boards as control and victim boards.

4.6.1 Nexys3 Control Board - Nexys3 DUT

This section describes connecting a Digilent Nexys3 Control board to a Nexys3 DUT board. Nexys3 board includes a Xilinx Spartan6 FPGA. This board can use PMOD connectors for I/O. It also descibes connections to other devices like the oscillopscope. For a high-level FOBOS hardware diagram, please refer to Figure 4.1.

The two boards are connected using the PMOD port C and D. Some pins in port A and B are used to connect to the oscillopscope and the clock generator.
Note that the two boards must share GND. PMOD ports C and D are connected, each pin to its corresponding pin in the other board (e.g. JA1 $->$ JA1 and GND$->$ GND). However the 3.3 V pin is not connected in the PMOD connector.

PMOD connector pin assignment on the control board

$PORTA$	$PORTB$	$PORTC$	$PORTD$
$JA1:$	$JB1: EXTClock$	$JC1: DUTClock$	$JD1: reset$
$JA2:$	$JB2:$	$JC2: do\_ready$	$JD2: di\_ready$
$JA3: trigger$	$JB3:$	$JC3: din(3)$	$JD3: dout(3)$
$JA4:$	$JB4:$	$JC4: din(1)$	$JD4: dout(1)$
$JA7:$	$JB7:$	$JC7: do\_valid$	$JD7:$
$JA8:$	$JB8:$	$JC8:$	$JD8: di\_valid$
$JA9:$	$JB9:$	$JC9: din(2)$	$JD9: dout(2)$
$JA10:$	$JB10:$	$JC10: din(0)$	$JD10: dout(0)$

PMOD connector pin assignment on the DUT board

$PORTA$	$PORTB$	$PORTC$	$PORTD$
$JA1:$	$JB1:$	$JC1: DUTClock$	$JD1: reset$
$JA2:$	$JB2:$	$JC2: do\_ready$	$JD2: di\_ready$
$JA3:$	$JB3:$	$JC3: din(3)$	$JD3: dout(3)$
$JA4:$	$JB4:$	$JC4: din(1)$	$JD4: dout(1)$
$JA7:$	$JB7:$	$JC7: do\_valid$	$JD7:$
$JA8:$	$JB8:$	$JC8:$	$JD8: di\_valid$
$JA9:$	$JB9:$	$JC9: din(2)$	$JD9: dout(2)$
$JA10:$	$JB10:$	$JC10: din(0)$	$JD10: dout(0)$

5 Test Vector Generation

The user must prepare test vectors before running data acquisition. User defined scripts or scripts provided with FOBOS can be used. The dataAcquisition.py script will send the test vectors one at a time and collect traces from the oscilloscope. The blockCipherTVGen.py can be used to generate test vectors to be used by block ciphers. The script is located at `fobos/sources`.

5.1 Test Vector Format

5.1.1 Supported commands

00C0 pdi fifo (length in bytes to follow)

00C1 sdi fifo (length in bytes to follow)

00C2 rdi fifo (length in bytes to follow)

0081 store expected output size (expected output size in bytes to follow)

0080 select command register (command to follow)

5.1.2 FOBOS Protocol Example

Here is an example of a signle test vector of the FOBOS protocol format:

```
00C0 # pdi fifo (length in bytes to follow);
     #FOBOS Controller must assert di_valid
0008 # 8 bytes
FFFF
FFFF
FFFF
FFFF # 8 bytes, 16 bits at a time
00C1 # sdi fifo (length in bytes to follow)
000A # 10 bytes
0000
0000
0000
0000
0000 # 10 bytes, 16 bits at a time
0081 # store expected output size
0008 # 8 bytes of output expected
0080 # select command register
0001 # "start signal"
```

5.2 Using the blockCipherTVGen.py script

The blockCipherTVGen.py (located at `fobos/sources`) can be used to generate test vectors for block ciphers. It can generate random key/palintext or fixed key/random plaintext. The script will add necessary headers per FOBOS protocol.

There are two steps to use the script:

1. Set user defined parameters.
2. Run the script. It will generate the test vector file and plaintext file (not required for acquisition).

User Defined Parameters:

```
############user defined settings
TRACE_NUM = <>       #Number of traces (plaintexts blocks to be sent to DUT)
                     #e.g 1000
PDI_LENGTH = <>      #Plaintext length in byets e.g 16
SDI_LENGTH = <>      #Key length in bytes e.g. 16
EXPECTED_OUT = <>    #Expected output in bytes i.e ciphertext length e.g 16
DIN_FILE = <>        #desitination file name e.g dinFile.txt
FIXED_KEY = <>       #Fixed key = yes | no e.g 'no'
KEY =  <>            # Fixed key (if needed)
                     #e.g. '00112233445566778899AABBCCDDEEFF'
```

Here is an example to generate 4 test vectors with 16 byte blocks, key and ciphertext. Key is fixed in this case:

```
############user defined settings
TRACE_NUM = 4            #Number of traces
PDI_LENGTH = 16          #In byets
SDI_LENGTH = 16          #In bytes
EXPECTED_OUT = 16        #expected output in bytes
DIN_FILE = 'dinFile.txt' #desitination file name
FIXED_KEY = 'yes'        #Fixed key = yes | no
KEY =  '00112233445566778899AABBCCDDEEFF' # Fixed key (if needed)
```

`$ python blockCipherTVGen.py`

Here is how the generated dinFile.txt looks like. `$ cat dinFile.txt`

```
00C00010B4900D0ECC646D4858C9125B3B61F76700C10010001122...BBCCDDEEFF0081001000800001
00C00010E1361496BA8F078ED02DC1283C2F98C200C10010001122...BBCCDDEEFF0081001000800001
00C00010150424A6E6EEED8C15D7DB737771FBE7400C10010001122...BBCCDDEEFF0081001000800001
00C00010FCC8863498CD255ED57F864FD02824A800C10010001122...BBCCDDEEFF0081001000800001
```

This file can now be used in FOBOS as a test vector file.

6 Data Acquisition

After test vectors have been generated, user can run dataAcquisition.py. The PC will send one test vector at a time to the control board, which sends it to DUT. The control board will trigger the oscilloscope to capture the power trace. The process will be repeated unitl all traces are collected.

The DUT Wrapper uses the header information in the test vector to put the data (plaintext, key etc.) into the correct FIFOs. The DUT Wrapper then allows the DUT (victim) algorithm to run by setting the victim reset to zero. The victim then drains the FIFOs (sdi,pdi and rdi FIFOs) and stores the output in the dout FIFO. Once the dout FIFO accumulates the expected amount of data, the DUT wrapper sends data to the controller which sends it to the PC.

Figure 4.1 shows the components of FOBOS including the handshake signals used. Before running the `dataAcquisstion.py` script, the user must modify the configuration files at `config/config.txt` and `confi/acquisitionconfig.txt`. Here is sample for `acquisitionConfig.txt` file.

6.1 Data Aquisition Configuration

6.1.1 General Settings

- MEASUREMENT_FORMAT
  Possible values: dat
  The format to store power measurements. Only dat is supported.

- LOGGING
  Possible values: INFO|DEBUG
  The logging level. DEBUG is more verbose.

6.1.2 Control Board Settings

- CONTROL_BOARD
  Possible values: Nexys3|Nexys2
  The Control Board type.

- VICTIM_RESET
  Possible values: Integer
  Reset the DUT after the specified clock cycles.

- TIME_OUT
  Possible values: Integer
  If the DUT will not return data after the specified clock cycles, a time-out signal is returned to the PC.

6.1.3 Trigger Settings

The Control Board can send a trigger to the Oscilloscope once the DUT starts processing the data (ie. di_ready = 0). Or it can be configured to trigger any number of clock cycles after this event occurs.

- TRIGGER_WAIT_CYCLES
  The number of clock cycles after which the trigger is asserted (after di_ready goes to zero).
- TRIGGER_LENGTH_CYCLES
  The time the trigger signal is asserted.
- TRIGGER_TYPE
  possible values: TRG_NORM, TRG_FULL, TRG_NORM_CLK, TRG_FULL_CLK
  - TRG_NORM
    Normal trigger mode. in this mode the TRIGGER_WAIT_CYCLES and TRIGGER_LENGTH_CYCLES are applied.
  - TRG_FULL
    Full trigger mode. While DUT is running (between di_ready = 0 and do_valid = 1) the trigger is asserted.
  - TRG_NORM_CLK
    Similar to TRG_NORM but the trigger signal is anded with the clock.
  - TRG_FULL_CLK
    Similar to TRG_FULL but the trigger signal is anded with the clock.
- CUT_MODE
  Controls how the trace retreived from the scope will be processed. possible values: FULL, TRIG_HIGH
  - FULL
    The trace is cut starting at the rising edge of the trigger to the end of the screen.
  - TRIG_HIGH
    the trace is cut from the rising edge to the falling edge of the trigger ie. the trace where the trigger is high will be saved.

6.1.4 Data Settings

- DATA_FILE
  Possible values: file name (string)
  The test-vector file name.
- EXPECTED_OUTPUT
  Possible values: Integer
  Expected output size in bytes.
- OUTPUT_FORMAT
  Possible values: hex
  Output format. 'hex' is the only supported format.

6.1.5 Capture Settings

- NUMBER_OF_TRACES
  Number of traces to collect.
- CAPTURE_MODE
  Possible values: MULTI|SINGLE
  Sigle encryption per traces or mutiple encryptions per trace.

6.1.6 Oscilloscope Settings

- OSCILLOSCOPE
  Possible values: AGILENT|OPENADC
  Oscilloscope type.
- OSCILLOSCOPE_IP
  Possible values: IP address
  Oscilloscope IP address.
- OSCILLOSCOPE_PORT
  Oscilloscope port number.
- AUTOSCALE = YES|NO
- IMPEDANCE
  Possible values: FIFTY|ONEMEG
  Oscilloscope input impedance.
- CHANNELx_RANGE
  Possible values: ON|OFF|voltage range
  Oscilloscope channel voltage range (Full screen range) in Volts.
- TIME_RANGE
  Possible values: float (seconds) Oscilloscope time range in seconds (full screen time).
- TIMEBASE_REF = LEFT
- TRIGGER_THRESHOLD
  Minimum voltage for valid trigger.
- TRIGGER_SOURCE
  Possible values: CHANNEL1|CHANNEL2|CHANNEL3|CHANNEL4
  Specifies the rigger channel.
- TRIGGER_MODE = EDGE
- TRIGGER_SWEEP = NORM
- TRIGGER_LEVEL = 1

- TRIGGER_SLOPE = POSITIVE
- ACQUIRE_TYPE = NORM|PEAK|HRES|AVER
- ACQUIRE_MODE = RTIM|ETIM|SEG

6.2 Sample Configuration File

```
 # ============================================
# Global Settings
# ============================================
# ============================================
MEASUREMENT_FORMAT = dat # Default => dat
LOGGING = INFO # INFO|DEBUG
# ============================================
# ============================================
# Control Board Settings
# ============================================
# ============================================
CONTROL_BOARD = Nexys3
TRIGGER_WAIT_CYCLES = 0 #@VICTIM CLOCK
TRIGGER_LENGTH_CYCLES = 1 #@VICTIM CLOCK
TRIGGER_TYPE = TRG_FULL #TRG_NORM | TRG_FULL | TRG_NORM_CLK | TRG_FULL_CLK
CUT_MODE = TRIG_HIGH #FULL | TRIG_HIGH
# ============================================
# ============================================
# Test Data Generation Settings
# ============================================
# ============================================
DATA_FILE = dinFile.txt
EXPECTED_OUTPUT = 16 # Expected output size in bytes
OUTPUT_FORMAT = hex # Default => hex
NUMBER_OF_ENCRYPTIONS_PER_TRACE = 1
BLOCK_SIZE = 16 # In Bytes
# ============================================
# ============================================
# FOBOS Capture Settings
# ============================================
# ============================================
DUMMY_RUN = NO #YES/NO
NUMBER_OF_TRACES = 50000
#####################################################
######## Signal Alignment Module Parameters ########
#####################################################
CAPTURE_MODE = SINGLE # MULTI|SINGLE
TRIGGER_THRESHOLD = 1.0
```

```
# ==========================================
# ==========================================
# FOBOS Oscilloscope Settings
# ==========================================
# ==========================================
# INTIALIZATION OPTIONS
OSCILLOSCOPE = AGILENT #AGILENT|OPENADC
OSCILLOSCOPE_IP = 192.168.10.10
OSCILLOSCOPE_PORT = 5025
AUTOSCALE = NO # YES|NO
IMPEDANCE = ONEMEG #FIFTY|ONEMEG
# VOLTAGE AND TIME RANGE OPTIONS
CHANNEL1_RANGE = 0.060V
CHANNEL2_RANGE = 6V
CHANNEL3_RANGE = OFF # ON|OFF|voltage range
CHANNEL4_RANGE = OFF # ON|OFF|voltage range
TIME_RANGE = 0.000050
TIMEBASE_REF = LEFT
# TRIGGER OPTIONS
TRIGGER_SOURCE = CHANNEL2
TRIGGER_MODE = EDGE
TRIGGER_SWEEP = NORM
TRIGGER_LEVEL = 1
TRIGGER_SLOPE = POSITIVE
# ACQUIRE OPTIONS
ACQUIRE_TYPE = NORM # NORM|PEAK|HRES|AVER
ACQUIRE_MODE = RTIM # RTIM | ETIM| SEG
```

6.3 Running Data Acquisition

Once the configuration is done, user can run

`python dataAcquisition.py`

The output will be saved in `workspace/<projectName>/measurements`. The traces are stored in a Numpy array called `rawDataAligned.npy`.

7 Correlation Power Analysis

Once acquisition is complete, Correlation Power Analysis CPA can be performed. User must provide their own power model and calculate hypothetical power for each key guess. FOBOS takes the hypothetical power for each key guess and use a correlation method e.g. Pearsons coefficient and calculates the correlation values. The key guess that achieves the highest correlation is a candidate for correct key.

The Analysis module is used to perform DPA attack on power traces collected by the Acquisition module. To Perform CPA, two inputs are needed, the power traces and and hypothetical power.

7.1 Steps to perform CPA using FOBOS

7.1.1 Hypothetical power calculation

Note: This description uses an example case where key is guessed one byte at a time.
Power model or hypothetical power data is user provided. FOBOS uses a text file for each key byte. This file includes a line for each key guess value (i.e. 0-255). Each line inculdes hypothetical power value for the specific key guess for all encryptions. Each value is an integer and separated from the next value by a space. The text below is an example for one key byte. The first number is the hypthetical power when the byte equals zero for the frist encryption, first value in the second line is the estimated power when the key byte equals one for the first encryption and so on. FOBOS expects to find these files at `$fobos/data/`. File names should be HPower_byte_<BYTE NUMBER>.txt.

Here is sample of `Hpower_byte_0.txt`

```
6 4 4 2 5 4 3 5 7 4 5 5 7 3 4 6 5 2 4 5 3 4 3 4 7 4 .
5 3 4 5 4 2 5 7 4 4 2 4 3 2 4 4 3 4 2 4 3 6 3 2 1 5 .
.
.
.
```

7.2 CPA configuration

There are few configuration files that controls the CPA analysis. Here we list all the configuration parameters used in the FOBOS Analysis and description of their usage.

7.2.1 Data Analysis Parameters

File: `fobos/config/dataAnalysisParams.txt`

- WORK_DIR
  Directory to save analysis files(Inside the measurements directory).

Possible Values: file name
Example: analysis

- MEASUREMENT_WORK_DIR
  The name of the measurement directory.
  Possible Values: directory name.
  Example: workspace

- TAG
  The type of prefix for the directory name. Used to distinguish different runs.
  Possible Values: counter
  Example: counter

7.2.2 Sample Space Disposition

File: samplesSpacesDisp.txt

- SAMPLE_WINDOW_SIZE
  Number of samples (per trace) to be used in analysis.
  Possible values : Number (e.g. 2000)

- SAMPLE_WINDOW_START
  The number of the first sample in the window.
  Possible values: Number (e.g. 100)

7.2.3 Compression Parameters

File: signalAlignmentParams.txt

- COMPRESSION_LENGTH
  Number of samples to be compressed into one samples.
  Possible values : Number (e.g. 10)

- COMPRESSION_TYPE
  The operation to be performed to generate the compressed sample.
  Possible values : MEAN | MAX | MIN

7.2.4 Sample Analysis Configuration Files

```
dataAnalysisParams.txt
#####################################################
WORK_DIR = FOBOSAnalysis
MEASUREMENT_WORK_DIR = FOBOSWorkspace
TAG = counter


compressionParams.txt
#####################################################
######## Compression Module Parameters #############
```

```
#####################################################
COMPRESSION_LENGTH = 10
COMPRESSION_TYPE = MEAN # MAX|MIN|MEAN

sampleSpaceDispParams.txt
#####################################################
#### Sample Space Disposition Module Parameters ####
#####################################################
SAMPLE_WINDOW_SIZE = 3000
SAMPLE_WINDOW_START = 3000
```

7.3 Running Data Analysis

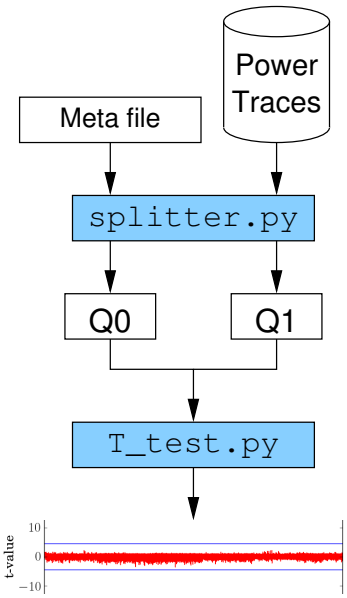
Data Analysis can be run as follows:

```
$cd $fobos
$python dataAnalysis.py
```

Once this is done, the script reads the measured and hypothetical power data, runs CPA and produces several output files. The script prompts the user for the directory that contains the traces and then uses the power data in the measurements directory as input. The script will create a new directory each time it runs. This directory is created in the project directory. Output files will be save in `fobos/<workspace>/<projectDir>/analysis/`.

8 Welch's T-test

Welchs T-test is used as a tool for leakage assessment. This chapter describes using FOBOS to perform a fixed-vs-random t-test.



8.1 Background

Usage of Welch's t-test for leakage assessment was introduced in [GOODWILL]. There are two methods to perform a t-test, specific and non-specific T-test. In specific t-test, like DPA an intermediate bit is trageted. We perform multiple operations on the DUT using random input and power traces are collected. The traces are then partitioned into two sets. The first set is the set of traces where the target bit equals one. We then run T-test on the two populations and see if they are distiguishable. This will indicate if DPA attck is likely to scucceed. In the non-specific t-test, a fixed palintext D is selected and randomly interleaved with random data. This interleaved data is processed by the DUT and traces are collected. The traces are partioned into two sets, one for the fixed palintext and the other for the random. T-test is then performed on the two sets. If the traces are distingushable, this indicates that the device is leaking information.

The equation used in t-test is as follows [GOODWILL]:

$$\frac{\mu_0 - \mu_1}{\sqrt{\frac{s_0^2}{n_0} - \frac{s_1^2}{n_1}}}$$

Where μ_0 and μ_1 are means of distributions 0 and 1, s_0 and s_1 are standard deviations, and n_0 and n_1 are the cardinality of the distributions, or the number of samples.

8.2 Test vector generation

Fixed-vs-random t-test uses interleaved fixed and random test vectors. These vectors should be generated by the user. We can select a fixed test vector D and create s set of test vectors that interleaves D and a randomly selected test vector. The interleaving is random.

For example the following test vector has been used to perform a t-test on an algorithm implemented on FOBOS DUT.

```
52000000830000010744451618AB57B8FFA4..6C48B60F02688701E000000043000000F0000000
52000000830000103B1C4EE0D776710B905..C2A8258DAF9BD939E000000043000000F0000000
52000000830000108EAB864A60C8CED8AC8..4B5D7CE3BBDB4666E000000043000000F0000000
52000000830000010589A762E564D5077BB3..B26D711E98989827E000000043000000F0000000
52000000830000103AEDF14E7998D7471FB..3DA0E90D53046D92E000000043000000F0000000
52000000830000106255E0D9D6FC53E0BBF..4A992A8EAF5B0646E000000043000000F0000000
52000000830000108EAB864A60C8CED8AC8..4B5D7CE3BBDB4666E000000043000000F0000000
52000000830000105E1868742DCE8785194..17EC1AE1861553D0E000000043000000F0000000
52000000830000100EEF4E5C15D888121FA..FFAED2002AC180ABE000000043000000F0000000
52000000830000108EAB864A60C8CED8AC8..4B5D7CE3BBDB4666E000000043000000F0000000
52000000830000108EAB864A60C8CED8AC8..4B5D7CE3BBDB4666E000000043000000F0000000
```

A meta file specifies which trace is random and which is fixed. A one represents a random trace and a zero represents a fixed trace. Below is an example of this file. Note how the bits correspond to random and fixed traces from the test vectors above.

```
11011101100100111111111110101010010
```

8.3 T-test module cofiguration

The T-test module needs two inputs:

1. The trace file. The traces collected using the test vectors described above.
2. The fixed vs random meta file. This file tells which trace is fixed and which trace is random.

T-test cofiguration is loacted at `fobos/config/analysis.ini`. Here is a sample file with explanation of each prameter:

```
[tTest]
#Number of traces to use in t-test (Both Q0 and Q1)
traceCount      = 2000
#File to store the t-t_values (output)
tValuesFile     = t_values.npy
#files to stor the distributions (random or fixed)
Q0File          = traces0.npy
Q1File          = traces1.npy
#Meta file that determines which trace is random
#and which is fixed (input). The file is expected
#to be at fobos/sources
```

```
fvrFile         = fvrchoicefile.txt
#file to store the t-t_values vs sample no. plot (output)
```

Most of the configuration parameters are file names for input and output files. The most important parameter to configure is the traceCount which is the total number of traces. Note that the fvrchoicefile is expected to be at `/fobos/sources`. However, the trace file is in the measurement file (where the acquisition module stores it).

8.4 Performing a t-test

Run the `fobos/bin/tTest.py` script. This will display a menu showing all the measurements done in the current project. You can select a measurement and the t-test will be performed on it. A new analysis directroy will be created under the selected project directroy to store output files.

9 Using the FOBOS Profiler

FOBOS profiler can be used to map time domain events or clock cycles to the specific sample in a power trace (or t-value). This can be use in applications like:

- Determine when leakage on a t-test occurs (clock cycle).
- Determining instruction power consumption.
- Correlating the state of a state machine to specific power trace samples.

9.1 FOBOS trigger settings

Once the DUT starts the crypto operation, it sets `di_ready` to zero. The Control Board can trigger the oscilloscope immediately when `di_ready=0` or after this event by any number of clock cycles. The trigger signal can be active for any number of clock cycles or until the DUT is done with its operation. The falling and rising edges of the trigger signal can be used later to truncated the trace properly to be used by the profiler.
optionally, a user may generate a file that maps the internal state of the DUT to clock cycles. This is called a 'state file'. This file can be generated by modifying the test bench to write a value corresponding to each state to a text file. The profiler script can take this state file as input along with the power traces (or t-values) and display clock transitions and the DUT state on each clock cycle.

9.2 Generating State Files

The following is an example on how to modify the fobos_dut_tb.vhd to write the state of the DUT to a text file. Writing starts just after di_ready becomes zero until `do_valid` goes to one. That is the time when the DUT is processing the data.
In fobos_dut_tb.vhd, we add this process:

```
--Process to write states
writeState: PROCESS(clk)
  VARIABLE VectorLine: LINE;
BEGIN
IF (rising_edge(CLK)) THEN
        IF (di_ready = '0' and do_valid = '0') THEN
          hwrite(VectorLine, state_debug);
          writeline(stateFile, VectorLine);
        END IF;
END IF;
ASSERT False
Report "Writing States"
```

```
SEVERITY NOTE;
END Process;
--------
```

We also add these lines in architecture declarations section:

```
---
FILE stateFile: TEXT OPEN WRITE_MODE is "state_file.txt";
signal state_debug : std_logic_vector(7 downto 0);
---
```

In this example, state is assumed to be 8 bit std_logic_vector reported by the DUT (victim) controller (i.e FOBOS_DUT have a port called `state_debug`) The file generated is a list of numbers each representing the state of the controller at the clock cycle identified by the line number.

9.3 Profiler configuration

Below is an example profiler configuration with description of each parameter. This configuration is located at `fobos/config/analysis.ini`

```
[profiler]
#File to read the t-t_values (input)
#Expected at the measurements directory
srcFile     = t_values.npy
#Enable/Disable clock plotting
display_clk     = YES
#num of clocks to in trace
num_of_clks     = 10
#high clock voltage for plotting
clk_high        = 5
#low clock voltage for plotting
clk_low         = -5
#File name to save plot
profilerPlot    = profiler_plot.png
#File that stors clock number vs state mapping
stateFile       = state_file.txt
```

9.4 Running the profiler

Run the `fobos/bin/runProfiler.py` script. This will display a menu showing all the measurements done in the current project. You can select a measurement and the profiler will take it as input. A new analysis directory will be created under the selected project directory to store output files.

9.5 Using the plot_t_values.py script

The `plot_t_values.py` is a script that can be used to run the profiler on t-values. Here is usage description.
Command line arguments:

```
$ python plot_t_values.py -h
usage: plot_t_values.py [-h] t_values plot_file

positional arguments:
  t_values    A .npy file that store traces as Nx1 Nupmy array that contains
              t-values.
  plot_file   File name where the plot is saved

optional arguments:
  -h, --help  show this help message and exit
```

Also there are settings on the script that need to be set. Here is a snippet of the configuration along with brief explanation for the configuration parameters:

```
##########GENERAL SETTINGS
start_ylim  = -40  #Plot ymlim range
end_ylim    = 40
##########END GENERAL SETTINGS
###########CLK GRAPH SETTINGS
display_clk = 'YES' #Enable/Disable clock plotting
num_of_clks = 18   #num of clocks to in trace. Known from behavioral simulation.
clk_high    = 10   #high clock amplitude for plotting
clk_low     = -10  #low clock amplitude for plotting
state_file  = 'state_map.txt' #text file that includes states. One state in each line.
###END CLK GRAPH SETTING
```

10 Power Measurement

10.1 Trace collection

Power traces need to be collected using a board called XBP. Figure 10.1 shows how this device is connected to the power supply and the DUT. The XBP includes a shunt resistor and an amplifier with selectable gain. The XBP is connected to the digital oscilloscope using an SMA connector. Power is supplied through the XBP to the DUT. The voltage drop across the shunt resistor is amplified and the oscilloscope records the amplified trace. Figure 10.2 shows an example trace collected using this setup.

10.2 Power Measurement Configuration

Before running the power measurement script, few configuration parameters must be set. The configuration is located at `/fobos/config/analysis.ini`. Below is an example of the configuration and brief description for the parameters.

```
[power]
#Source trace file (input)
srcFile         = rawDataAligned.npy
#File name to write the output
dstFile         = powerResults.txt
#Number of traces used in power calculation
numTraces       = 20
#sample number to start/end. Samples before and after
#will be ignored.
startSample     = 0
endSample       = 15000
```

10.3 Running the Power Measurement Script

Run the `fobos/bin/calcPower.py` script. This will display a menu showing all the measurements done in the current project. You can select a measurement and the script will take it as input. A new analysis directory will be created under the selected project directory to store output files. Power measurement will be displayed on the screen that shows minimum and average power. Below is a sample output file.

```
Source File: rawDataAligned.npy
Start sample (for truncated traces): 1
...
XBP Shunt Resistance (ohms): 1
...
```

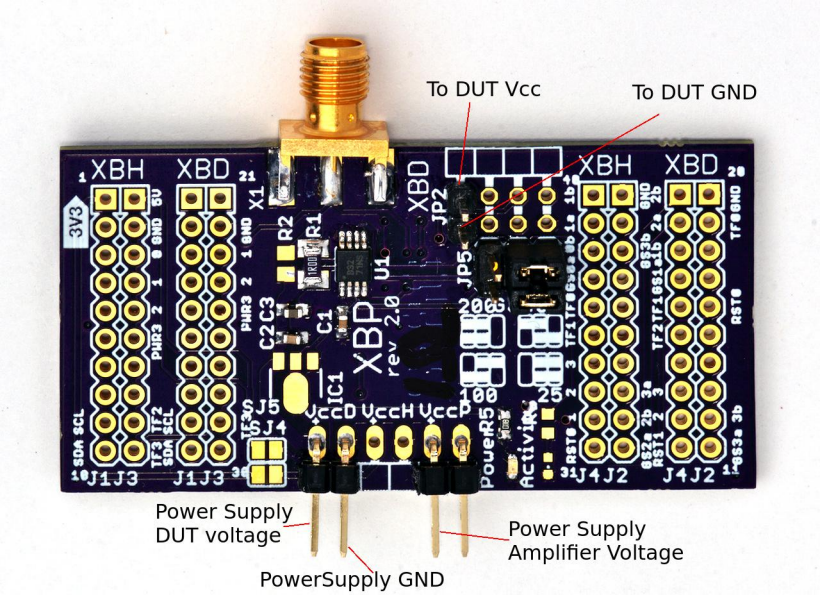


Figure 10.1: The XBP Board

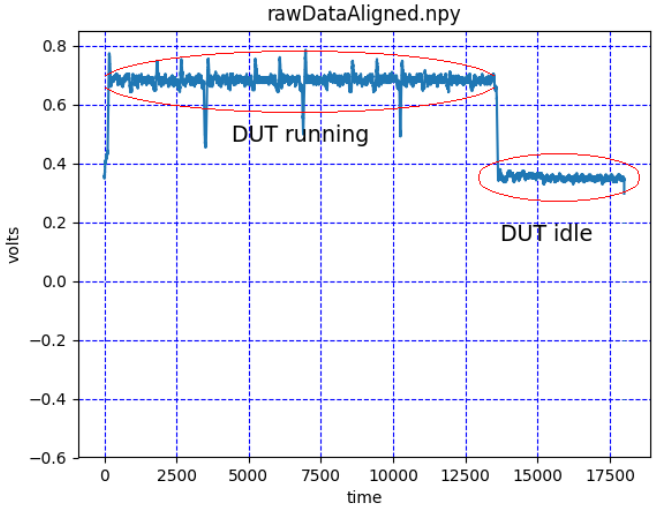


Figure 10.2: The XBP Board

```
Mean voltage for truncated traces is: 0.67885V
Mean power for truncated traces is: 0.0326W
```

A Function Descriptions

A.1 FOBOS - Analysis Module

FOBOS's analysis module uses a set of python scripts to post process the raw measurement data obtained from the oscilloscope and perform analysis on the obtained data. Various functions implemented in the Analysis module is described below:

A.2 FOBOS - Capture Module

The Capture module is used to run encryptions on hardware and capture traces from the ocsilloscope.

A.3 FOBOS - Other Functions

Here we list helper functions that are used by the Analysis and Capture modules.

Table A.1: getAlignedMeasuredPowerData Function

signalAnalysisModule.getAlignedMeasuredPowerData	
Usage	`signalAnalysisModule.getAlignedMeasuredPowerData()`
Inputs	None
Outputs	M x N Numpy matrix that holds aligned traces. Where M is the number of traces and N is the number of samples per trace.
Description	Reads aligned traces from the $Workspace/$projectName/$attempt/Measurements directory and loads it to an M x N Numpy matrix. This function calls signalAnalysisModule.readAlignedDataFromFile().

Table A.2: readAlignedDataFromFile Function

signalAnalysisModule.readAlignedDataFromFile	
Usage	`signalAnalysisModule.readAlignedDataFromFile()`
Inputs	None
Outputs	M x N Numpy matrix that holds aligned traces. Where M is the number of traces and N is number of samples per trace.
Description	Reads aligned data from file and returns M x N Numpy matrix.

Table A.3: detectSampleSize Function

signalAnalysisModule.detectSampleSize	
Usage	`signalAnalysisModule.detectSampleSize(fileName)`
Inputs	Aligned traces file name.
Outputs	Number of samples in a trace.
Description	Reads the first 10 traces and returns the number of samples in the largest trace. If the number of traces is less than 10 all traces are read. This is done to be able to adjust all traces to the same number of samples if some do not have the same size due to acquisition timing.

Table A.4: adjustSampleSize Function

signalAnalysisModule.adjustSampleSize	
Usage	`signalAnalysisModule.adjustSampleSize(sampleLength, dataArray)`
Inputs	• sample length to adjust to. • N x 1 nympy array that represents one trace where N is the number of samples in the trace.
Outputs	SampleLenght x 1 Numpy array that represents the adjusted trace.
Description	Used to modify the number of samples in a trace. If the number of samples is less than SampleLength, the array is padded with zeros. If the number of samples is more than SampleLength, the array is truncated. The function does nothing if the number of samples is equal to SampleLength.

Table A.5: acquirePowerModel Function

signalAnalysisModule.acquirePowerModel	
Usage	`signalAnalysisModule.acquirePowerModel (HyptheticalDataFileName,globals.ADAPTIVE_CPA)`
Inputs	• Power model file name • Correlation type
Outputs	M x N Numpy array that holds the hypothetical power traces. Where N is the number of encryptions/decryptions and M is the number of key guesses.
Description	Reads the hypothetical power data from file in $fobos/data. Returns M x N Numpy array that holds the hypothetical power traces. Where N is the number of encryptions/decryptions and M is the number of key guesses.

Table A.6: computeAlignedData Function

signalAnalysisModule.computeAlignedData	
Usage	`signalAnalysisModule.computeAlignedData(powerData, triggerData)`
Inputs	• N x 1 Numpy array that holds measured power data. Where N is the number of samples. • N x 1 Numpy array that holds trigger power data. Where N is the number of samples.
Outputs	Aligned power trace (K x 1 Numpy array where K is the number of samples).
Description	Uses powerData and triggerData to generate the aligned trace. The function looks for the rising edge of the trigger signal to determine the start of the trace.

Table A.7: correlation_pearson Function

sca.correlation_pearson	
Usage	`sca.correlation_pearson(measuredPowerData, hypotheticalPowerData)`
Inputs	• M x N Numpy matrix for power traces. Where M is the number of encryptions/decryptions, N the number of samples per trace. • L x M array the represents the hypothetical power values. Where M is the number of encryptions/decryptions and L is the number of key guesses (i.e for byte guess, L=256).
Outputs	N x L Numpy correlation matrix where N is the number of samples per trace.
Description	Calculates Pearson correlation between the hypothetical data and measured data. Returns an N x L correlation matrix. When we guess byte values L = 256.

Table A.8: findMinimumGuessingEntropy Function

sca.findMinimumGuessingEntropy	
Usage	`signalAnalysisModule.computeAlignedData(measuredPowerData, triggerData)`
Inputs	• measured power data. • hypotheticalPowerData.
Outputs	Minimum Guessing Entropy graph
Description	calculates the find Minimum Guessing Entropy

Table A.9: plotTrace Function

plottingModule.plotTrace	
Usage	`plottingModule.plotTrace(dataToPlot, traceNos, plotType)`
Inputs	• M x N Numpy matrix that holds traces to plot. Where M is the number of traces and N is the number of samples. • Trace number • Plot type per trace. The traces to be plotted.
Outputs	None
Description	Plots the traces represented by the Numpy matrix. The x-axis represents time and the y-axis represents voltage.

Table A.10: plotHist Function

plottingModule.plotHist	
Usage	`plottingModule.plotHist(corrMatrix, corrType)`
Inputs	• M x N Correlation Numpy matrix. • Correlation type.
Outputs	None
Description	Plots a histogram. The x-axis represents the key guess and the y-axis represenst the number of occurrences.

Table A.11: plotCorr Function

plottingModule.plotCorr	
Usage	`plottingModule.plotCorr(corrMatrix, corrType)`
Inputs	• M x N Numpy correlation matrix • Correlation type
Outputs	None
Description	Plots correlation data.

Table A.12: sampleSpaceDisp Function

postProcessingModule.sampleSpaceDisp	
Usage	`postProcessingModule.sampleSpaceDisp(alignedData)`
Inputs	M x N Numpy matrix that holds aligned data. Where M is the number of traces and N is the number of samples per trace.
Outputs	M x Window_Size Numpy array that holds the aligned data after removing samples
Description	Removes samples before WINDOW_START and after WINDOW_START + WINDOW_SIZE - 1 from each trace.

Table A.13: compressData Function

compressData	
Usage	`compressData(measuredPowerData)`
Inputs	M x N Numpy array that represents traces. Where M is the number of traces and N is the number of samples per trace.
Outputs	M x K Numpy array that represents compressed traces. Where M is the number of traces and $K = \frac{N}{COMPRESSION\_LENGTH}$.
Description	Summarizes COMPRESSION_LENGTH samples into one sample. The summarization type depends on the COMPRESSION_TYPE configuration parameter which can be MEAN, MIN or Max.

Table A.14: compress Function

postProcessingModule.compress	
Usage	`postProcessingModule.compress (a, compressionLenght, compressionType)`
Inputs	• M x N Numpy array that represents traces. Where M is the number of traces and N is the number of samples per trace. • Number of samples to compress into one sample. • Compression type.
Outputs	M x K Numpy array that represents compressed traces. Where M is the number of traces and $K = \frac{N}{COMPRESSION\_LENGTH}$.
Description	This function is called by postProcessingModule.compressData() to do the compression.

Table A.15: traceExpunge Function

postProcessingModule.traceExpunge	
Usage	`postProcessingModule.traceExpunge(measuredPowerData)`
Inputs	M x N Numpy array that represents traces. Where M is the number of traces and N is the number of samples per trace.
Outputs	L x N Numpy array that represents traces after removing the traces that do not fall in acceptable range. Where L is the number of traces and N is the number of samples per trace.
Description	Removes traces that do not fall in acceptable range.

Table A.16: openOscilloscopeConnection Function

Oscilloscope_core.openOscilloscopeConnection	
Usage	`Oscilloscope_core.openOscilloscopeConnection()`
Inputs	None
Outputs	None
Description	Connects to oscilloscope. It opens a socket using the IP address OSCILLOSCOPE_IP and port number OSCILLOSCPOE_PORT. Also gets the oscilloscope identifier.

Table A.17: setOscilloscopeConfigAttributes Function

Oscilloscope_core.setOscilloscopeConfigAttributes	
Usage	`Oscilloscope_core.setOscilloscopeConfigAttributes()`
Inputs	None
Outputs	None
Description	Configures the oscilloscope by sending commands (in text format) to the oscilloscope.

Table A.18: initializeOscilloscopeDataStorage Function

Oscilloscope_core.initializeOscilloscopeDataStorage	
Usage	`Oscilloscope_core.initializeOscilloscopeDataStorage()`
Inputs	None
Outputs	None
Description	Creates empty Numpy arrays for each enabled oscilloscope channel.

Table A.19: armOscilloscope Function

Oscilloscope_core armOscilloscope	
Usage	`Oscilloscope_core.armOscilloscope()`
Inputs	None
Outputs	None
Description	Instructs the oscilloscope to digitize channels specified in FOBOS configuration.

Table A.20: populateOscilloscopeDataStorageAndAlign Function

Oscilloscope_core.populateOscilloscopeDataStorageAndAlign	
Usage	`Oscilloscope_core.populateOscilloscopeDataStorageAndAlign(traceCount)`
Inputs	The number of current trace.
Outputs	None
Description	Reads power data trace from oscilloscope and trigger signal trace. It then aligns the trace to the trigger signal and saves the aligned trace to file.

Table A.21: closeOscilloscopeConnection Function

Oscilloscope_core.closeOscilloscopeConnection	
Usage	`Oscilloscope_core.closeOscilloscopeConnection()`
Inputs	None
Outputs	None
Description	Closes socket that connects to oscilloscope.

Table A.22: openControlBoardConnection Function

Oscilloscope_core.openControlBoardConnection	
Usage	`usbcomm_core.openControlBoardConnection()`
Inputs	None
Outputs	None
Description	Initializes connection to control board, resets control board and reads control board and victim clocks.

Table A.23: initializeControlBoardConnection Function

usbcomm_core.initializeControlBoardConnection	
Usage	`usbcomm_core.initializeControlBoardConnection()`
Inputs	None
Outputs	None
Description	Initializes the USB connection to the board. Called from OpenControlBoardConnection().

Table A.24: sendTriggerParamsToControlBoard Function

usbcomm_core.sendTriggerParamsToControlBoard	
Usage	`usbcomm_core.sendTriggerParamsToControlBoard()`
Inputs	None
Outputs	None
Description	Sends the trigger parameters to the control boards. Parameters are: TRIGGER_WAIT_CYCLES and TRIGGER_LENGTH_CYCLES.

Table A.25: runEncrytionOnControlBoard Function

usbcomm_core.runEncrytionOnControlBoard	
Usage	`usbcomm_core.runEncrytionOnControlBoard(traceCount)`
Inputs	The number of block used in encryption.
Outputs	None
Description	Sends a block of data to control board to do encryption. The key is sent before sending the frist block.

Table A.26: sendKeyToControlBoard Function

usbcomm_core.sendKeyToControlBoard()	
Usage	`usbcomm_core.sendKeyToControlBoard()`
Inputs	None
Outputs	None
Description	Sends the key to control board. This function is called from usbcomm_core.runEncrytionOnControlBoard().

Table A.27: sendBlockOfDataToControlBoard Function

usbcomm_core.sendBlockOfDataToControlBoard	
Usage	`usbcomm_core.usbcomm_core.sendBlockOfDataToControlBoard (traceCount)`
Inputs	The number of block used in encryption
Outputs	None
Description	Sends a block of data to the control board. This function is called from usbcomm_core.runEncrytionOnControlBoard().

Table A.28: saveControlBoardOutputDataStorage Function

usbcomm_core_core.saveControlBoardOutputDataStorage	
Usage	`Oscilloscope_core.saveControlBoardOutputDataStorage()`
Inputs	None
Outputs	None
Description	Saves output from control board (cipher text) to file. File is stored in $Workspace/$projectName/$attempt/output/

Table A.29: getKeyForAnalysis Function

dataGenerator.getKeyForAnalysis	
Usage	`dataGenerator.getKeyForAnalysis()`
Inputs	None
Outputs	Key formated as a list of hexadecimal bytes.
Description	Reads the key from file in $Workspace/$project/$attempt/output/ directory.

Table A.30: getPlainText Function

dataGenerator.getPlainText	
Usage	`dataGenerator.getPlainText()`
Inputs	None
Outputs	A list of blocks that represents plain text.
Description	Generates random plain text or reads from file depending on configuration. Plain text file is located in $fobos/$sources/ directory.

Table A.31: generateRandomKey Function

dataGenerator.generateRandomKey	
Usage	`dataGenerator.generateRandomKey()`
Inputs	None
Outputs	A list of key bytes. Each byte is represented as a hexadecimal string.
Description	Generates a random key in hexadecimal format. Key size is read form the KEY_SIZE configuration parameter.

Table A.32: convertToHex Function

dataGenerator.convertToHex	
Usage	`dataGenerator.convertToHex(hexString)`
Inputs	Hexadecimal string
Outputs	
Description	

Table A.33: configureWorkspace Function

configExtract.configureWorkspace()	
Usage	`configExtract.configureWorkspace()`
Inputs	None
Outputs	None
Description	Creates the project directory in the workspace and creates directories to store measured power data, cipher text and plain text etc. It also copies some configuration files and other files into the project directory.

Table A.34: extractConfigAttributes Function

configExtract.extractConfigAttributes()	
Usage	`configExtract.extractConfigAttributes()`
Inputs	None
Outputs	None
Description	Reads the main configuration file to get configuration attributes. It also reads the acquisition configuration file and extracts configuration attributes.

Table A.35: updatePowerAndTriggerFileNames Function

configExtract.updatePowerAndTriggerFileNames	
Usage	`configExtract.updatePowerAndTriggerFileNames()`
Inputs	None
Outputs	None
Description	Checks for the existance of measured data files and trigger data file and sets variables to the file names.

Table A.36: configureAnalysisWorkspace Function

configExtract.configureAnalysisWorkspace	
Usage	`configExtract.configureAnalysisWorkspace()`
Inputs	None
Outputs	None
Description	Configures the analysis workspace directory by creating directories to store analysis results and copies configuration files.

Table A.37: extractAnalysisConfigAttributes Function

configExtract.extractAnalysisConfigAttributes	
Usage	`configExtract.extractAnalysisConfigAttributes(fileName)`
Inputs	Configuration file name.
Outputs	None
Description	Reads the file provided and gets configuration attributes. Also, copies the file to the projects local configuration directory for future reference.

Table A.38: goToSleep Function

support.goToSleep	
Usage	`support.goToSleep(value)`
Inputs	Time to sleep in seconds.
Outputs	None
Description	Sleep for number of seconds.

Table A.39: exitProgram Function

support.exitProgram	
Usage	`support.exitProgram()`
Inputs	None
Outputs	None
Description	Self-explanatory

Table A.40: wait Function

support.wait	
Usage	`support.wait()`
Inputs	None
Outputs	None
Description	Waits for the user to press Enter to continue program execution.

Table A.41: clear_screen Function

support.clear_screen	
Usage	`support.clear_screen()`
Inputs	None
Outputs	None
Description	Self-explanatory.

Table A.42: convertToByteArray Function

support.convertToByteArray	
Usage	`support.convertToByteArray(hexString)`
Inputs	Hexadecimal string
Outputs	A byte array
Description	Converts a hexadecimal string to a byte array.

Table A.43: arrayToString Function

support.arrayToString(array)	
Usage	`support.arrayToString(array)`
Inputs	Array to convert
Outputs	A string that consist of array elements
Description	Self-explanatory.

Table A.44: readFile Function

support.readFile	
Usage	`support.readFile(fileName)`
Inputs	File name
Outputs	A string that holds file content.
Description	Self-explanatory.

Table A.45: removeFile Function

Support.removeFile	
Usage	`Support.removeFile(fileName)`
Inputs	File name
Outputs	None
Description	Self-explanatory.

Table A.46: removeComments Function

Support.removeComments(datalist)	
Usage	`Support.removeComments(datalist)`
Inputs	Data list
Outputs	
Description	Removes comments (anything after a '#' sign) from a list of strings.