Flexible Opensource BOard for Sidechannel analysis

FOBOS Version 0.1

Table of Contents

1 Flexible Opensource workBench fOr Sidechannel analysis - FOBOS 3

1 Introduction and Motivation 3

2 Previous Work 3

2.1 SCA - Hardware Platforms 3

2.2 SCA - Data Analysis Platforms 4

2.3 Drawbacks of Current SCA Evaluation Platforms 5

3 Our Approach 5

4 Architecture of FOBOS 7

4.1 FOBOS Hardware 7

4.2 FOBOS Software 9

5 FOBOS Data Analysis Module: 11

5.1 Signal Alignment Module 11

5.2 Post processing Module 11

5.3 SCA Module 14

5.4 Statistics Module 14

6 CPA Attack on AES using FOBOS 15

2 Module Descriptions 24

7 FOBOS - Capture Module 24

8 FOBOS - Analysis Module 24

Flexible Opensource workBench fOr Sidechannel analysis - FOBOS

1 Introduction and Motivation

While a few FPGA boards designed for SCA exist, many research groups from academia and industry use their own hardware harness, their own software for data acquisition and data analysis and sometimes their own FPGA boards or generic FPGA boards. This increases the complexity and effort needed to obtain a working SCA setup. Another, but costly option is the use of commercial SCA workstations.

Due to the importance of the topic of side channel attacks, they became part of the curriculum of cryptography courses in many universities. However, only very few have associated laboratory exercises and hands-on examples due to the cost and complexity of current SCA setups.

To our knowledge no complete software package exists that contains everything needed for evaluating the side-channel attack resistance of FPGA implementations from data acquisition to analysis (see Sect:2). In this chapter, we are presenting a framework for efficient side-channel evaluation of cryptographic implementations on hardware and software. Such an environment should be flexible, open-source and low cost and beneficial to both research and educational communities.

2 Previous Work

2.1 SCA - Hardware Platforms

The Side Channel Analysis Board (SCAB) introduced in [?], was one of the early efforts in developing evaluation platforms for conducting SCA attacks on implementations of cryptographic algorithms. This board housed an FPGA on which the cryptographic algorithms can be implemented along with an unrestricted access to power and clock pins to perform the following SCA attacks: Differential Power Analysis (DPA) and fault analysis. Information about the board design and the status of the project is currently not available.

The Side Channel Attack Standard Evaluation Board (SASEBO) [?], [?] was developed by the Research Center for Information (RCIS) of National Institute of Advanced Industrial Science and Technology (AIST) and Tohoku University as a common platform for evaluating side channel attacks. These boards were developed with the intent of performing side channel attacks on various hardware

Table 1. SASEBO boards with FPGAs as victims

Board	Control	**Victim**		**Wires Control–**	Host Data
	FPGA	FPGA	Techn.	**Victim**	Communication
SASEBO	Virtex-2 Pro	Virtex-2 Pro	130 nm	54	RS232
SASEBO-G	Virtex-2 Pro	Virtex-2 Pro	130 nm	53	RS232, FT245RL (USB)
SASEBO-GII	Spartan-3A	Virtex-5	65 nm	46	FT2232D (USB)
SASEBO-B	Stratix-2	Stratix-2	90 nm	53	RS232, FT245RL (USB)

platforms like FPGAs, ASICs and Smart cards. SASEBO boards are designed with two FPGAs, a cryptographic FPGA (or an ASIC/Smart card) where the algorithm can be implemented and a control FPGA which directs the data flow between the software and the cryptographic FPGA. The data acquisition software which comes with SASEBO is written in C#. It does not provide support for different brands of oscilloscopes. Hence the user is required to tweak the code to provide support for his/her own oscilloscope. Only four different types of SASEBO boards with FPGAs as victims (shown in Table 1) are available. Early this year, AIST announced that it discontinued support for the SASEBO project. Morita Tech [?] recently announced SAKURA as a successor to SASEBO project.

2.2 SCA - Data Analysis Platforms

The DPA Contest [?] organized jointly by VLSI research group of Telecom ParisTech university and AIST, is an online-based contest with the aim of having a fair confrontation between different attack methodologies. Currently three editions of this contest were introduced of which the first two deal primarily with attacking DES (v1) and AES (v2) using different techniques where as the goal of the third edition is to compare acquisition platforms and techniques. The results for the third edition was recently announced at COSADE 2012. For the v1 & v2 editions, the acquired data was provided by the contest organizers where as in v3 only the RTL description of AES was provided. Data acquisition was left to the the participant choice. This contest provides a wealth of information regarding DPA statistical techniques, although all the data acquisition is obtained from SASEBO GII only.

The OpenSCA Toolbox [?] is an open source project which consists of set of Matlab codes and objects to perform DPA attacks. Using this toolbox one can conduct not only first order power analysis attacks but also the higher order and template attacks. The toolbox also comes with several examples, demonstrating the attacks. Currently the supported statistical testing procedures are Difference-of-Means, Correlation Power Analysis and Baysian analysis. All codes are written in Matlab and does not include data acquisition. In short, we can perform only data analysis using OpenSCA.

The DPA Workstation™ [?] is a state-of-the art proprietary SCA testing platform by *Cryptography Research, Inc.* DPA Workstation™can perform data acquisition, processing and analysis and also has the ability to generate hypothesis models for a range of ciphers like AES, DES, RSA, ECC etc. It also provides support for data capture for a wide range of sampling devices like oscilloscopes and PCI A/D converters. Additionally, it supports multiple hardware platforms (FPGAs, SoC etc.) and different sensors (current, field probes) and hence both power and EM attacks can be performed using this workstation. The major drawback is that this tool is not freely available and licensing is very costly, thus not usable for educational purposes. Also, collaborations between research groups are difficult as they might not all have access to the DPA Workstation™.

2.3 Drawbacks of Current SCA Evaluation Platforms

An efficient SCA evaluation platform should have the following criteria:

– Flexibility: Able to support multiple hardware platforms/technologies/vendors.
– Open Source: Community support will allow for rapid development and adoption of the latest devices and technologies.
– Reproducibility: Results published in research should be reproducible to obtain a fair SCA analysis of cryptographic algorithms.
– Broad-Spectrum Acceptance: Should be accepted by both educational (low-cost) and research/industry (state-of-the-art) communities.

We have shown in Sect. 2.1 and Sect. 2.2 that a complete (acquisition to analysis), free and open source solution is not available. Therefore, research groups and industry who do not want to invest in the proprietary DPA Workstation™employ home grown scripts, programs and platforms. Their main disadvantages are that they are mostly written in an ad-hoc fashion and therefore difficult to maintain and extend. These scripts and platforms are also proprietary and hence, their results are not reproducible by other research groups. SASEBO currently has limited hardware support. OpenSCA toolbox can perform data analysis only. The DPA contest provides information about different attack strategies only.

Hence there is a need for a flexible and complete open-source framework for SCA that allows fair and comprehensive evaluation of implementations on hardware platforms with reproducible results.

3 Our Approach

We call our framework for efficient side-channel evaluation of hardware platforms - FOBOS. This abbreviation stands for Flexible Open-sources workBench fOr Side-channel analysis. FOBOS, loosely named after the Greek god Phobos ($\phi\acute{o}\beta o\varsigma$) who personifies fear and can pierce shields. FOBOS is designed to be an inexpensive side channel analysis setup that includes a complete software package with programs for victim control, data acquisition and data analysis. In order to evaluate side-channel leakage of hardware platforms, FOBOS uses off-the shelf FPGA boards as control and victim which are less expensive than the traditional setup. Thus, it enables universities to add active side channel analysis laboratory exercises to their cryptography classes. Furthermore, FOBOS is designed in a modular fashion to allow for a multitude of victim devices while maintaining the remainder of the setup, hence making FOBOS flexible. The FOBOS software package, documentation, and hardware components will be released as open-source for quick adaptation of newer technologies. Designers of cryptographic implementations and countermeasures against DPA and DEMA on FPGAs can test their design techniques on FPGAs from various vendors and with different technologies. As the hardware and software are open source, the results are reproducible by researchers from different groups.

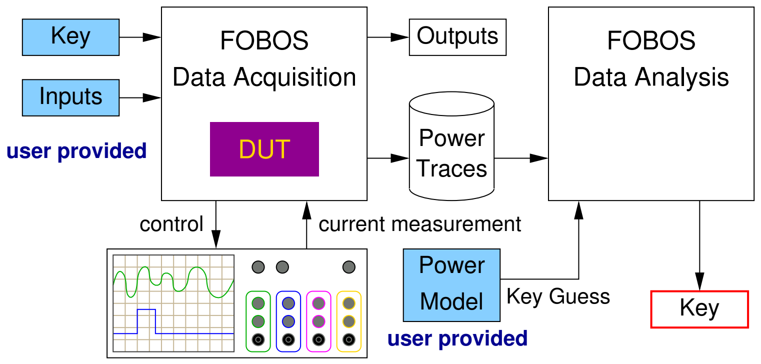

Fig. 1. Components of FOBOS

Figure 1 shows various components of FOBOS. It consists of the *FOBOS Hardware* as well as software for *Data Acquisition and Control* and *Data Analysis*. The FOBOS Hardware consists of two FPGA boards that are connected to each other. It is also possible to use the SASEBO GII board instead. The user has to provide the hardware description of the cipher under investigation, the key, a set of inputs and a power model. The Data Acquisition and Control module configures and controls the FOBOS Hardware and the Oscilloscope. It takes the user provided key and inputs and sends them to the FOBOS Hardware which in turn encrypts the inputs with the key and returns the outputs (i.e. ciphertext). As soon as the FOBOS Hardware starts with the encryption, it sends a trigger signal to start data acquisition of the oscilloscope. The Data Analysis module uses the user supplied power model, which can be based on inputs and/or outputs, and the power traces collected by the oscilloscope to recover the key.

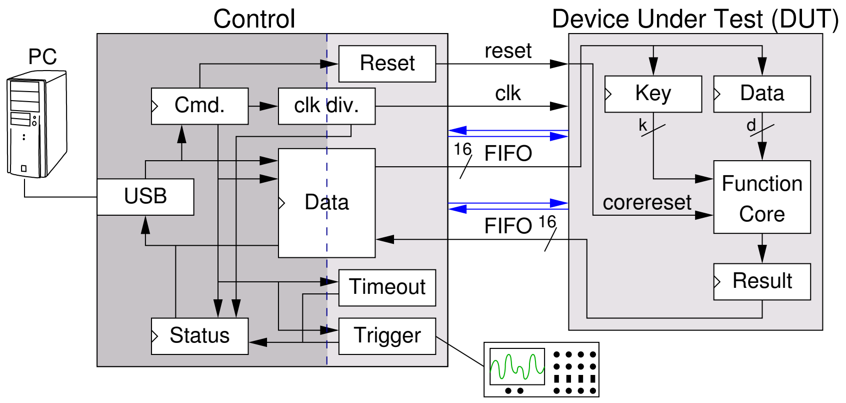

Fig. 2. Schematic Diagram of FOBOS Hardware

4 Architecture of FOBOS

FOBOS has two parts, the *FOBOS Hardware* and the *FOBOS Software*. The following sections describe the functionality of various components of FOBOS.

4.1 FOBOS Hardware

A schematic diagram of the FOBOS hardware is shown in Fig. 2. It consists of two boards *Victim Board* & *Control Board* connected together by the so called bridge connector. The cryptographic algorithms whose security needs to be evaluated are to be implemented on the FPGA of the Victim board. Data i.e. plaintext and/or key is sent from the PC via USB to the control FPGA, which then forwards the data to the Victim FPGA. After processing, the Victim FPGA sends the results back to the Control FPGA which in turn forwards the results to the PC for verification. The control board also sends a trigger signal to the oscilloscope to capture power measurement data.

Control Board: The control board used by FOBOS is either a Nexys2 or a Nexys3 board. Table 2 shows details of the both boards. The control board contains several modules (see Fig. 2) and two clock domains. It uses the on-board 50 MHz oscillator as base clock for the USB communication. The second clock is generated through a clock divider circuit which uses the Digital Clock Managers (DCMs) to generate a clock in the range of 350 KHz $\sim$ 50 MHz from the 50 MHz oscillator on board depending upon the user's choice and the oscilloscope specification. This clock is used for communication with the victim FPGA and also provided to the victim FPGA board.

The control board receives commands from the PC and returns a status. This is facilitated through the 8-bit Command and Status registers. We use them to implement a simple protocol between PC and Control FPGA which is explained in Sect. 4.2.

Table 2. FOBOS FPGA Control Boards

Board	FPGA	Technology	Connector	PC-Control	Cost
Nexys 2	Spartan-3E	90 nm	Hirose FX2 (43)	USB2	$149
Nexys 3	Spartan 6	45 nm	VHDC (40)	USB2	$199

The Trigger module generates a reference point from which the oscilloscope should start measuring the power consumption of the victim FPGA. Depending upon the user's requirement, this reference point can be set through a command to the beginning of the cryptographic operation or to specific clock cycle during the computation. This reference point is later used to perform signal alignment of several power traces.

A Timeout module makes sure that PC receives a status (of TIMEOUT) if an exception occurs during the communication with the victim or if the victim does not respond within a given time. This timeout value can be specified through a command. The timeout counter is automatically reset each time the victim returns data.

The Reset module is used to send a reset signal to the crypto core implemented on the victim FPGA depending upon the value specified by the user. This is useful if for example a cryptographic operation takes 1,000 clock cycles to complete, however, the interesting event happens in the 30th clock cycle. The user can then reset the victim automatically every 35 clock cycles and start a new encryption without having to wait for the encryption to complete.

Victim Board: We are investigating several FPGA boards available in the market, which can be used as Victim boards for FOBOS. Table 3 shows some potential Victim boards. The column "V_{Core} Jumper" indicates whether the board contains a jumper on the core power line which allows for by-passing the on board core power supply and inserting a current sensor (resistor or current probe) to measure the power consumption of the victim FPGA. So far, we have successfully used the Spartan 3E Starter Kit, Spartan 3E-1600 Developer Board, and the Altera DE1 board as FOBOS Victim boards. As the Altera DE1 does not have V_{Core} Jumper, we had to de-solder the voltage regulator for core voltage. On all boards we also removed several capacitors. Our preliminary investigation (shown in Table 3) into the other boards have shown that it is possible to modify them in order to measure the current of the core supply.

For each victim board we plan on publishing instructions on how to modify it for DPA and the printed circuit board (PCB) layout of the bridge connector which connects the victim board securely to the control board.

FOBOS Control-Victim Protocol The FOBOS Control-Victim Protocol uses a simple FIFO interface to transfer data to and from the control and victim FPGAs. The functionality of the input and output ports of the protocol is described in [**?**], [**?**]. All data and key to and from the FPGA is broken into segments. The first 2 bytes (16-bit) of each segment is a command word, which decides the nature of the segment and the number of bytes being sent.

Table 3. FOBOS FPGA victims

Board	FPGA	Techn-ology	$\mathbf{V_{Core}}$ Jumper	Cost
Spartan 3E Starter	Spartan-3E	90 nm	yes	$159
Spartan 3E-1600 Dvlp.	Spartan-3E	90 nm	yes	$225
Altera DE1	Cyclone-II	90 nm	no	$150
Cyclone III Starter	Cyclone-III	65 nm	yes	$199
Genesys Board	Virtex-5	65 nm	no	$449
Altera DE2-115	Cyclone-IV	60 nm	no	$299
Altys Board	Spartan-6	45 nm	no	$199
Altera DE4	Stratix-IV	40 nm	no	$2,995
Xilinx ML605	Virtex-6	40 nm	no	$1,795
Xilinx KC705	Virtex-7	28 nm	no	$1,695

The format of the 16-bit command words is shown in Fig 3. A '0' value in the LSB and a '0' value in the MSB of the command word indicates that a key is being sent. Similarly a '1' value in the LSB indicates that data is send. The bit in position '1' indicates with a '0' that more segments are following the current one, a '1' indicates that the current segment is the last. The MSB bit value '1' for a 16-bit command for loading the key is left explicitly for future use. This protocol does not require the control board to know what the block size of the cryptographic function is. The widths of the buses for 'k' and data 'd' indicated in Fig 2 can be defined by the user according to the requirement of the cryptographic implementation.

4.2 FOBOS Software

FOBOS Software Control Flow: The FOBOS control flow is shown in Fig. 4. The control script parses the configuration files and initializes the FOBOS environment. It performs a simple tool check to verify whether the necessary library files essential for data transfer and oscilloscope control are installed and only continues when the check passes successfully. The control script then assigns the hardware and oscilloscope attribute values as specified by the user in the configuration files. The FOBOS hardware then performs a built-in self test to check whether all the attributes are set accordingly and issues an appropriate status message to the control script. The status message can be a success or an error code. If the control receives an error code it exits the program displaying proper error message. On receiving a success code, the control script instructs the oscilloscope to digitize its analog inputs which then in turn waits for the trigger signal from the control board to start capturing data. The plaintext and

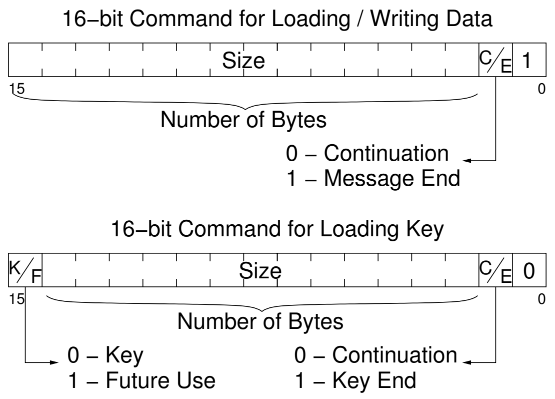

Fig. 3. FOBOS Protocol

the key are then transferred to the FOBOS hardware and the control script waits until it receives data from the oscilloscope. Once the oscilloscope data is captured, the control script writes the outputs from the FOBOS hardware to a file.

FOBOS has support for two data capturing modes, called *Single Capture* and *Multi Capture* to capture the power traces. Single Capture mode, as shown in Fig. 5a), assumes that a power trace contains a single encryption whereas Multi Capture mode, as shown in Fig. 5b), contains multiple encryptions per power trace. Once all data has been captured the control is transferred to data analysis module.

FOBOS PC- Control Communication Protocol: FOBOS uses the command & status registers to control the PC- Control communication. The command register is used (shown in Fig. 2) to pass the option values to the modules inside the control FPGA and to signal the control board that PC is ready to transmit the data. The status register (shown in Fig. 2) on the other hand, is used for signaling the PC that the control FPGA is ready to transmit the data obtained from victim FPGA or to report errors.

FOBOS Data Acquisition Module: The data acquisition module configures the oscilloscope and retrieves its data. Its behaviour is determined by a configuration file which uses a generic, oscilloscope brand independent description. A special, oscilloscope dependent sub-module translates the configuration file to commands which are oscilloscope specific. The sub-module of our prototype uses the Virtual Instrument Software Architecture (VISA) library which is a standard for configuring and programming instruments using a variety of interfaces. Presently, the FOBOS prototype supports communication for oscilloscopes from Agilent Technologies. In future we plan to provide support for oscilloscopes from other manufacturers.

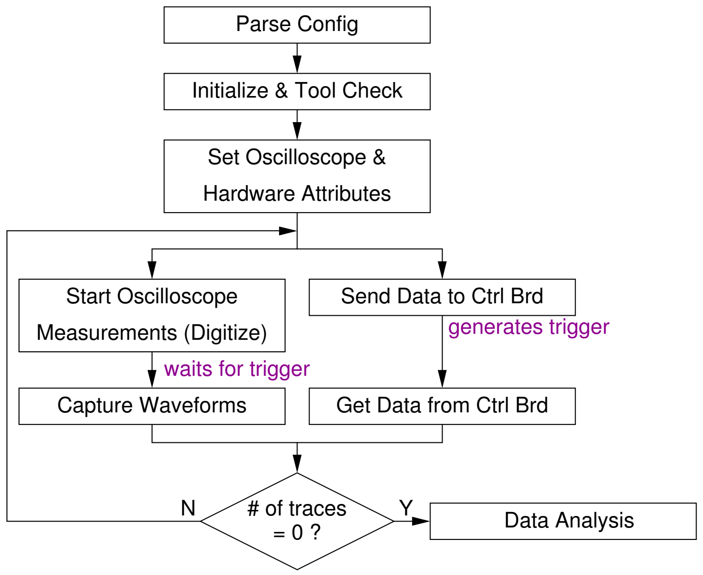

Fig. 4. FOBOS Control Flow

5 FOBOS Data Analysis Module:

The Data Analysis module consists of 3 sub-modules as shown below:

– Signal Alignment Module
– Post processing Module
– SCA Module
– Statistics Module

5.1 Signal Alignment Module

As the name indicates, the main function of this module is to align all the measured power traces with respect to time using a reference signal called the "Trigger" signal. Depending upon the type of capture mode used i.e. Single capture or Multi capture its functionality varies. In Single capture mode as each measurement trace contains only one encryption, the power traces are aligned when the "Trigger" signal becomes logic high. In the Multi capture mode, as each measurement trace contains multiple encryptions, the start of each encryption is indicates by the Trigger signal. To be exact, the measurement point from one trigger high to the subsequent trigger high is equivalent to one encryption. The measurement module chops up the trace accordingly and aligns them.

5.2 Post processing Module

Currently FOBOS supports three Post processing sub modules. They are:

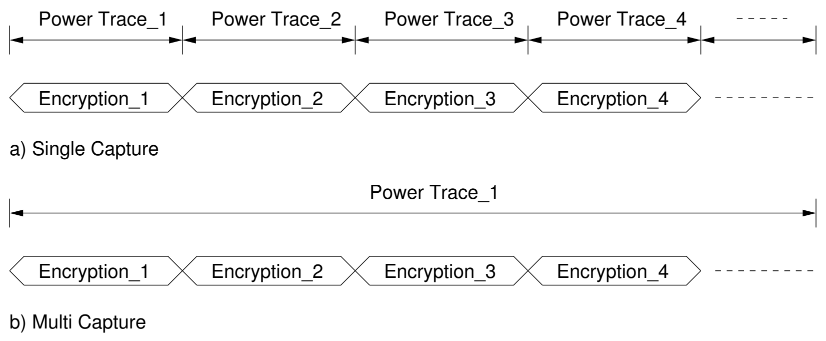

Fig. 5. Capture Modes

1. Sample Space Disposition
2. Compression
3. Trace Expunge

Sample Space Disposition allows user to select a part of the trace to perform further post processing or to perform statistical testing. This also reduces computation time during statistical testing.

Fig. 6. Sample Space Disposition

The parameters which facilitate this selection are "WINDOW START POINT" and "SAMPLE WINDOW SIZE". As shown in Fig 6 "WINDOW START POINT" parameter indicate which point in time to further sample the trace and the "SAMPLE WINDOW SIZE" parameter indicates the number of points to be sampled and stored to a new trace.

Compression allows user compress the power trace in chunks or as a whole depending upon the user requirement by using the "COMPRESSION LENGTH" parameter as shown in Fig 7 The user can also the type of compression to be used. They are:

– Max: Can compress the user specified sample size to the maximum of the given sample set.
– Min: Can compress the user specified sample size to the minimum of the given sample set.
– Mean: Can compress the user specified sample size to the average of the given sample set.

This also reduces computation time during statistical testing.

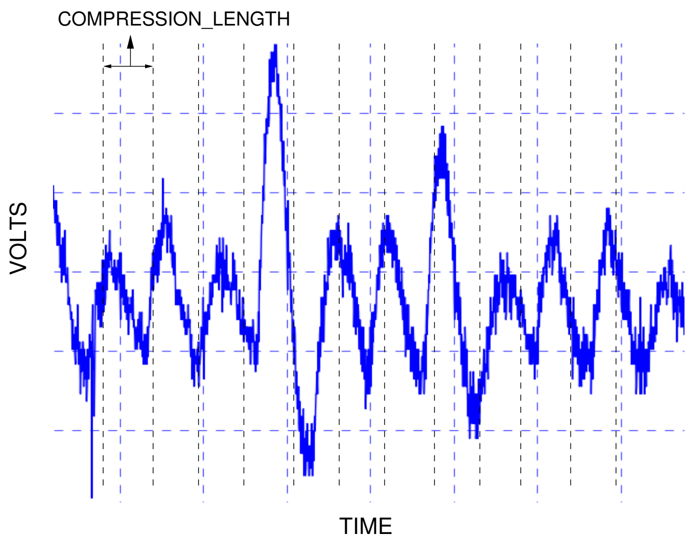

Fig. 7. Compression

Trace Expunge allows user to selectively ”expunge” or discard one or more traces depending upon two different selection criteria:

- Variance: User specifies Upper and Lower bound values. A given trace is removed if its variance (of the entire trace) is above upper limit or below lower limit.
- Standard Deviation: User specifies Upper and Lower bound values. A given trace is removed if its standard deviation (of the entire trace) is above upper limit or below lower limit.

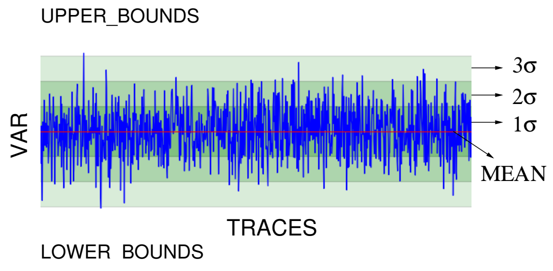

Fig. 8. Trace Expunge

This will help in removing traces which are heavily influenced by factors like measurement artefacts etc,.

5.3 SCA Module

Currently FOBOS supports following Side-channel distinguishers:

– Pearson's r
– Spearman's RHO

Pearson's r We assume a linear relationship between the power consumption of the device and the data being processed. Hence we use Pearson product-moment correlation coefficient (r), commonly known as Pearson's correlation to correlate instantaneous power consumption with hamming distance model [**?**].

The Pearson's correlation (r) between the the power consumption of the device P and the hypothetical power model H is given by the Eq. 1

$$r(P,H) = \frac{n\sum_i^n P_i H_i - \sum_i^n P_i \sum_i^n H_i}{\sqrt{n\sum_i^n P_i^2 - (\sum_i^n P_i)^2}\sqrt{n\sum_i^n H_i^2 - (\sum_i^n H_i)^2}} \tag{1}$$

Correlation Power Analysis (CPA) is a form of DPA where we use a different statistical test to obtain the secret key. Henceforth, we use the term DPA and CPA alternatively throughout this document.

Spearman Rank coefficient , on the other hand, is a measure of monotonic relationship between two variables, in this case between power consumption and power model. The Rank correlation between power consumption samples P and power consumption hypothesis samples G is given by

$$RHO(P,G) = 1 - \frac{6\sum_i^n d_i^2}{n(n^2-1)} \tag{2}$$

where d_i =P_i - G_i, and P_i, G_i are ranks of the variables P and G

5.4 Statistics Module

FOBOS currently supports following statistic functions.

– Mean : Calculates mean of the trace, both trace and sample wise, as shown in Fig 9.
– Standard Deviation : Calculates Standard Deviation of the trace, both trace and sample wise, as shown in Fig 9.
– Variance : Calculates Variance of the trace, both trace and sample wise, as shown in Fig 9.

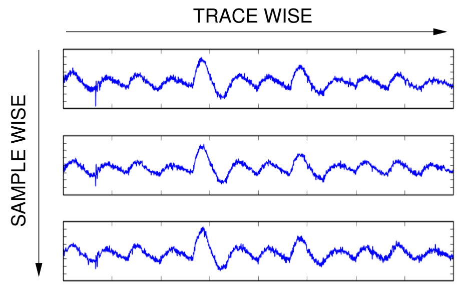

Fig. 9. Trace wise vs Sample wise

6 CPA Attack on AES using FOBOS

This section describes a Correlation Power Analysis (CPA) attack of an implementation of the Advanced Encryption Standard (AES) [?] using FOBOS. AES is an symmetric-key cipher used extensively in security sensitive applications world wide. AES applies four different transformations, SubBytes, ShiftRows, MixColumns, and AddRoundKey, per round and iterates through several such rounds depending upon the key size. An intermediate key called "round key" is generated and used per round which is derived from the original key through a reversible key scheduling function. We have implemented a basic iterative architecture i of AES with 128-bit key length and 128-bit wide datapath requiring 11 clock cycles for one encryption. Key scheduling is done on-the-fly and the SubBytes function is realized through look-up-tables. The block diagram for this design is shown in Fig. 10.

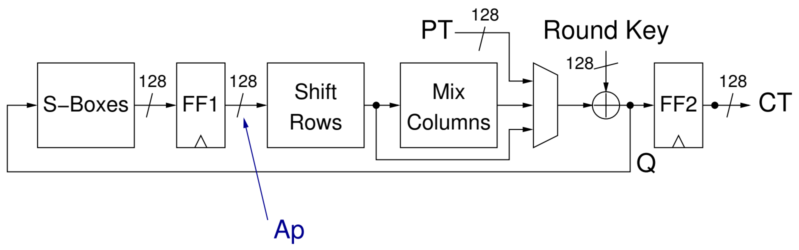

Fig. 10. Block Diagram of the AES Core

We attack our AES design during the first round at the output of the register FF1 indicated by Ap in Fig. 10. The equation for calculating the Hamming Distance (HD) is shown in 3. We use Pearson's Correlation to correlate the instantaneous power consumption with the HD model.

$$P_{est.} = \mathrm{HD}(\mathrm{SBOX}(CT_i), \mathrm{SBOX}(k_{guess} \oplus PT_{i+1})) \tag{3}$$

Fig. 11. AES Core with Wrapper on Victim FPGA

Figure 12 shows a snippet of hardware attributes specified in the FOBOS configuration file. FOBOS Control sends data from datain.txt and a key from keyin.txt, which are both in the format of ASCII coded Hexadecimal values, to the victim. A snapshot of these files is shown in Fig. 13. FOBOS Control sets the timeout to 30,000 clock cycles and the trigger to 4 clock cycles after processing starts. The victim clock is set to run at 500 KHz and the result will be stored in hexadecimal values in the file outputs.txt

```
DATA_FILE = datain.txt
KEY_FILE  = keyin.txt
CLK_FREQ = 500 KHz
TIME_OUT = 30000
TRIGGER = 4
CAPTURE_MODE = multi
```

Fig. 12. Snippet of config.txt

```
. . . .
40 F6 BB C7 94 78 0B D7 99 C3 5F 6A 77 8F 05 D8
A5 34 8B CC 02 EE C0 68 B4 9E 29 A5 22 B8 EF 54
CB 00 B7 22 F8 36 F9 E4 40 E2 EE BD 1B 13 BA A3
. . . .
```

```
2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
```

```
. . . .
0A 59 8B A5 3D B3 0D B6 34 B2 C2 7E 98 A8 DB 71
2E 13 7A 5F E2 F9 86 C0 15 9A 69 AB 6E 3F 04 01
FB D0 09 43 E7 71 59 4A 15 37 53 33 A3 EF 74 1B
. . . .
```

Fig. 13. Plaintext, Key & Ciphertext sent to FOBOS in hex format

A snippet of oscilloscope attributes from osc_config.txt file is shown in Fig. 14. The FOBOS control connects to the instrument specified by the VISA address from the RESOURCE attribute. The voltage ranges of the channels of the oscilloscope are specified in terms of vertical full-scale value in volts. The time range of the channels are specified in terms of horizontal full-scale value in seconds. In general oscilloscopes are configured in 8x10 graticule, therefore channel-1 range is 0.0125 Volts/div, channel-2 range is set to 2 Volts/div. The time range is set to 0.01 Sec/div. We also set the trigger source to be channel-2 and the condition on trigger to be positive edge.

```
RESOURCE  = GPIB0::7::INSTR    #Instrument Resource
CHANNEL_RANGE1 = 0.1V          #Specified in Volts per screen
CHANNEL_RANGE2 = 16V           #Specified in Volts per screen
TIME_RANGE = 0.001             #Specified in seconds per screen
TRIGGER_SOURCE =  CHANNEL2
TRIGGER_MODE =  EDGE
TRIGGER_SLOPE = POSITIVE
```

Fig. 14. Snippet of osc_config.txt

The FOBOS control sends the data from the oscilloscope i.e. the power traces, inputs, and outputs to the data analysis module. The first step involves processing the raw power trace using the preamble information to obtain the *measured_power_trace*. We use Multi Capture mode to obtain the Raw power trace. We use the Signal alignment module to process and align the traces as shown in Fig. 15

The parameters used for the signal processing module is shown in Fig. 16.

We perform Trace Expunge sub routine on the processed power traces using the parameters shown in Fig. 16. The resultant power trace is shown in Fig. 17

We further process the resultant power trace using sample space disposition and compression using the parameters shown in Fig. 16. The resultant power traces are shown in Fig. 18 and Fig. 19

The CPA attack is conducted on a sub-byte of the key depending upon the user choice. Hence there are 256 different key guess values and correspondingly 9 different HD values i.e. $0 \rightarrow 8$. A snippet of the *est_power_traces* per key guess value is shown in Fig 20.

The sca module than calculates the Pearson's Correlation and Spearman's Rank correlation for all the key guesses by correlating the *est_power_traces* and *processed_power_trace*. Snippets of the output logs of the Pearson's and Spearman's Correlation are shown in Fig. 21 and Fig. 22 respectively.

The sca module also plots two graphs, called the Correlation Plot shown in Fig. 23 for Pearson's r and Fig. 24 for Spearman's RHO respectively. The correlation plot shows how well each individual key guess correlates with the power trace. The peak value of this plot indicates the correct sub-key byte.

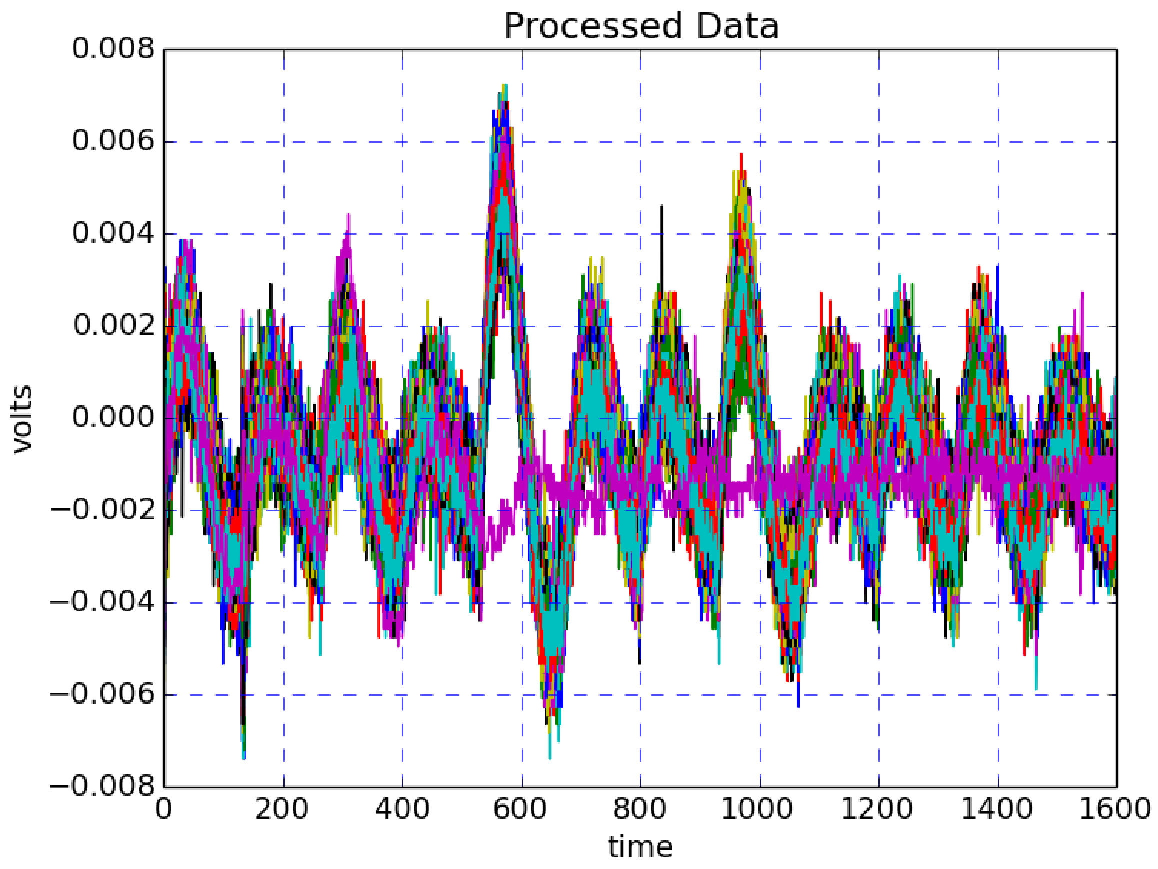

Fig. 15. Aligned Power Trace

```
CAPTURE_MODE = MULTI # MULTI|SINGLE
TRIGGER_THRESHOLD = 1.8
TRACE_EXPUNGE_PARAMS = VAR:0.0000025:0.0000035
SAMPLE_WINDOW = 1000
WINDOW_START_POINT = 100
COMPRESSION_LENGTH = 40
COMPRESSION_TYPE = MAX
```

Fig. 16. Snippet of Signal Processing module Parameters

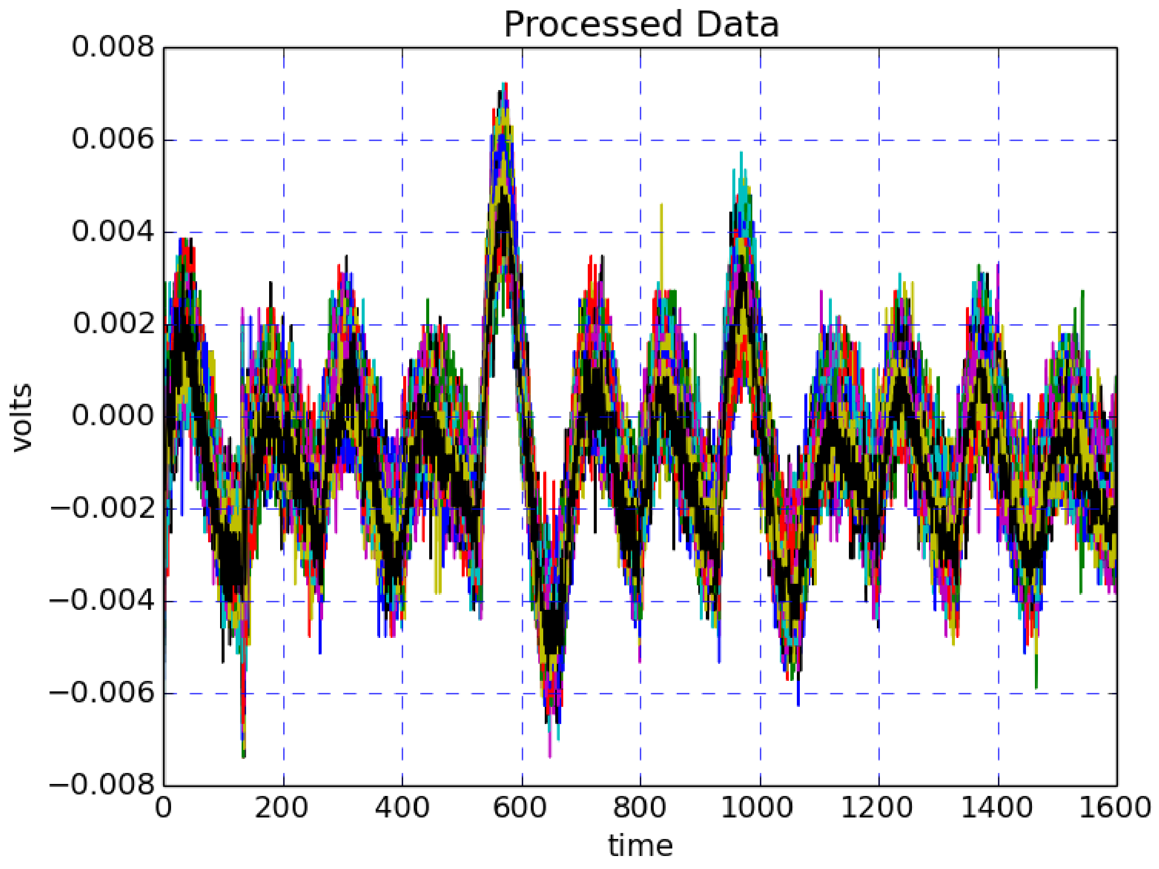

Fig. 17. Power Trace after Trace Expunge

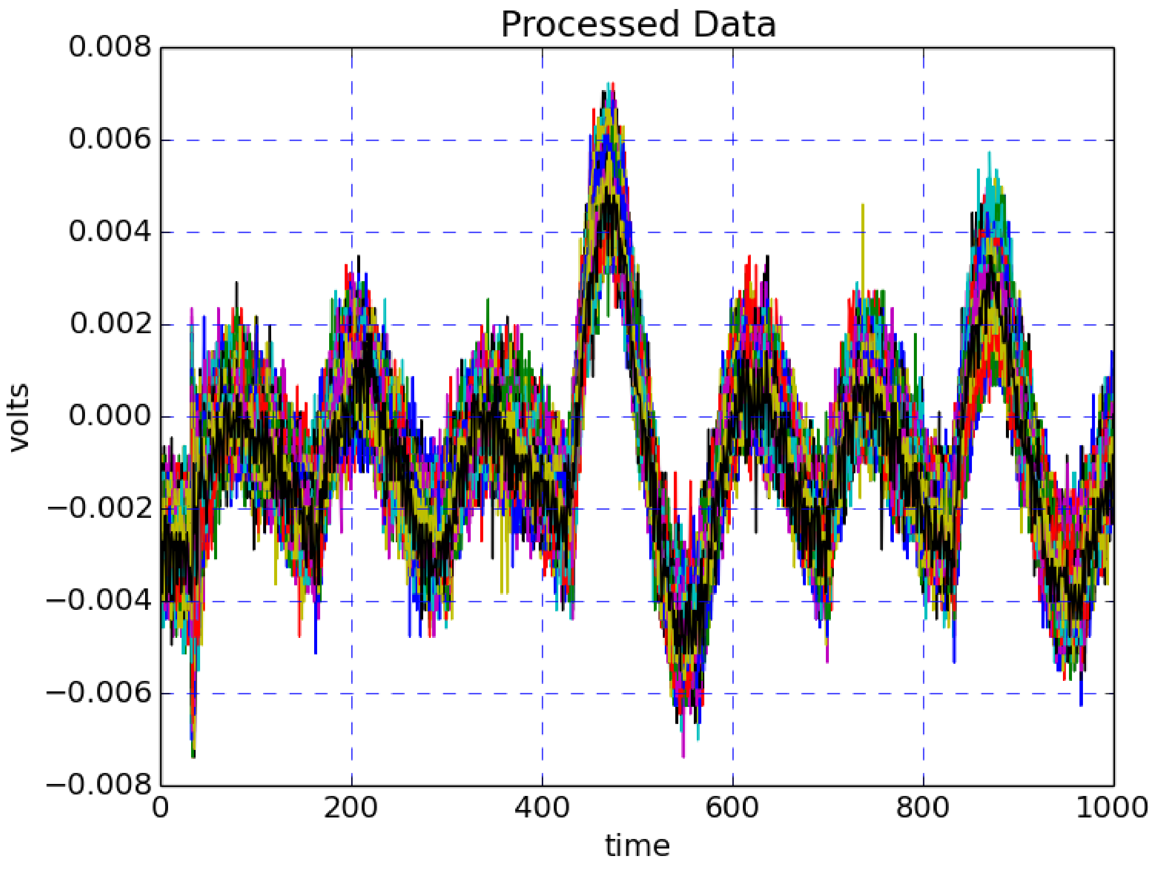

Fig. 18. Power Trace after Sample Space Disposition

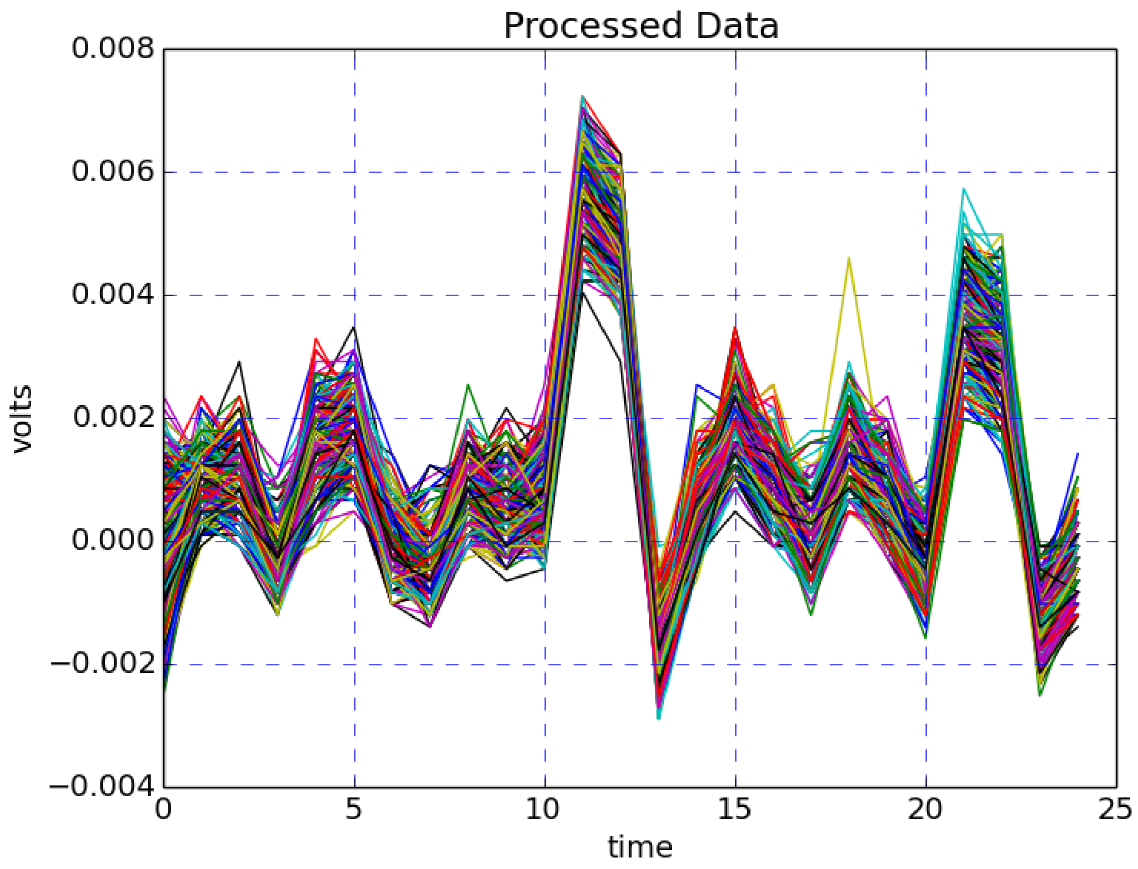

Fig. 19. Power Trace after Compression

```
4 6 3 4 5 1 5 4 4 2 4 3 5 6 3 4 4 3 4 2 . . . .
3 2 7 6 5 6 5 5 7 3 5 4 4 2 6 5 2 3 2 5 . . . .
5 4 3 6 4 5 3 4 3 3 6 5 3 0 2 5 6 5 6 5 . . . .
. . . .
```

Fig. 20. Hypothetical Power Model

```
Window[0] Key Byte- 0x5d [93] Correlation- 0.080
Window[1] Key Byte- 0x9c [156] Correlation- 0.078
Window[2] Key Byte- 0x9c [156] Correlation- 0.086
Window[3] Key Byte- 0x6 [6] Correlation- 0.086
Window[4] Key Byte- 0x16 [22] Correlation- 0.109
Window[5] Key Byte- 0x16 [22] Correlation- 0.175
Window[6] Key Byte- 0xa3 [163] Correlation- 0.087
Window[7] Key Byte- 0xd7 [215] Correlation- 0.082
```

Fig. 21. CPA using Pearson's r Log file

```
Window[0] Key Byte- 0x5d [93] Correlation- 0.082
Window[1] Key Byte- 0x9c [156] Correlation- 0.083
Window[2] Key Byte- 0x78 [120] Correlation- 0.109
Window[3] Key Byte- 0x6 [6] Correlation- 0.083
Window[4] Key Byte- 0x16 [22] Correlation- 0.105
Window[5] Key Byte- 0x16 [22] Correlation- 0.176
Window[6] Key Byte- 0xa3 [163] Correlation- 0.086
Window[7] Key Byte- 0xd7 [215] Correlation- 0.091
```

Fig. 22. CPA using Spearman's RHO Log file

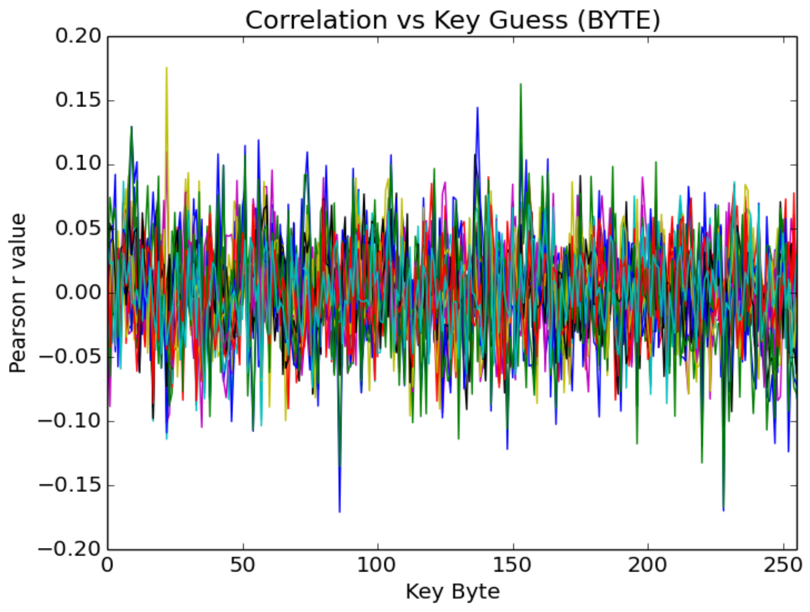

Fig. 23. Results of Pearson's r

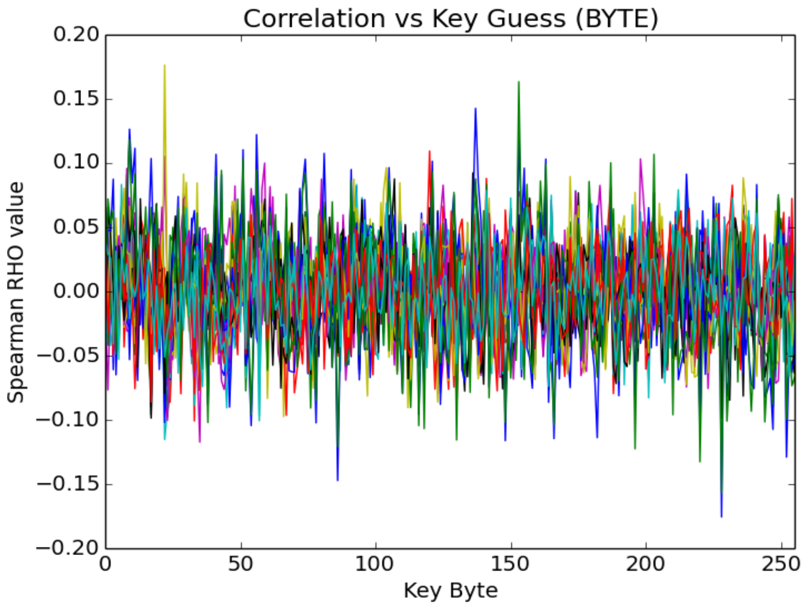

Fig. 24. Results of Spearman's RHO

Module Descriptions

7 FOBOS - Capture Module

8 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:

Table 4. Config Extract Functions

configExtract.extractAnalysisConfigAttributes()	
Usage	`$configExtract.extractAnalysisConfigAttributes(filename)`
Description	Loads the configuration attributes required for various analysis submodules
Inputs	file-name
Outputs	None

Table 5. Signal Alignment Functions

signalAlignmentModule.getAlignedMeasuredPowerData()	
Usage	`$signalAlignmentModule.getAlignedMeasuredPowerData()`
Inputs	None
Outputs	An M x N numpy array matrix
Description	Aligns all the raw measured data obtained from the oscilloscope with respect to trigger signal. This function returns a M x N numpy array matrix where there are M encryptions/decryptions and N oscilloscope sample points per measurement

Table 6. Signal Alignment Functions

signalAlignmentModule.acquireHypotheticalValues()	
Usage	`$signalAlignmentModule.acquireHypotheticalValues(filename)`
Inputs	filename
Outputs	An M x N numpy array matrix
Description	Loads the hypothetical power model into an M x N numpy array where there are M secret key guesses and N encryptions. This file is to be placed in `$fobos\powermodels` directory.

Table 7. getAlignedMeasuredPowerData Function

signalAnalysisModule.getAlignedMeasuredPowerData()	
Usage	`$signalAnalysisModule.getAlignedMeasuredPowerData()`
Inputs	N/A
Outputs	An M x N Numpy matrix that hold aligned traces. Where M is the number of traces and N is number of samples per trace.
Description	Reads aligned traces from the $workspace/$projectName/Measurements directory and loads it to M x N Numpy array where M is the number of traces and N is number of samples per trace. This function calls signal-AnalysisModule.readAlignedDataFromFile() to read data from file.

Table 8. readAlignedDataFromFile Functions

signalAnalysisModule.readAlignedDataFromFile()	
Usage	`signalAnalysisModule.readAlignedDataFromFile()`
Inputs	N/A
Outputs	An M x N Numpy matrix that hold 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 where M is the number of traces and N is number of samples per trace. File name is read from cf . **revise**

Table 9. getAlignedMeasuredPowerData Function

signalAnalysisModule.getAlignedMeasuredPowerData()	
Usage	`$signalAnalysisModule.getAlignedMeasuredPowerData()`
Inputs	N/A
Outputs	An M x N Numpy matrix that hold aligned traces. Where M is the number of traces and N is number of samples per trace.
Description	Reads aligned traces from the $workspace/$projectName/Measurements directory and loads it to M x N Numpy array where M is the number of traces and N is number of samples per trace. This function calls signalAnalysisModule.readAlignedDataFromFile() to read data from file.

Table 10. readAlignedDataFromFile Functions

signalAnalysisModule.readAlignedDataFromFile()	
Usage	`signalAnalysisModule.readAlignedDataFromFile()`
Inputs	N/A
Outputs	An M x N Numpy matrix that hold 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 where M is the number of traces and N is number of samples per trace. File name is read from cf . **revise**

Table 11. detectSampleSize Functions

signalAnalysisModule.detectSampleSize()	
Usage	`signalAnalysisModule.detectSampleSize(fileName)`
Inputs	The 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 due to acquisition timing.

Table 12. adjustSampleSize Functions

signalAnalysisModule.adjustSampleSize()	
Usage	`signalAnalysisModule.adjustSampleSize(sampleLength, dataArray)`
Inputs	SampleLength, 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 13. signalAnalysisModule.acquirePowerModel Function

signalAnalysisModule.acquirePowerModel()	
Usage	`signalAnalysisModule.acquirePowerModel (HypthecticalDataFileName,globals.ADAPTIVE_CPA)`
Inputs	Power model file name, correlation type
Outputs	An M x N Numpy array that holds the hypothetical power traces.
Description	Reads hypothetical power data from the HypthecticalDataFileName file in cfg.POWERMODELDIR directory. Returns M x N Numpy array that holds the hypothetical power traces.

Table 14. pdatePowerModelUsingBCPA Function

signalAnalysisModule.updatePowerModelUsingBCPA(myArray1, myArray2, rowA)	
Usage	`signalAnalysisModule.updatePowerModelUsingBCPA(myArray1, myArray2, rowA)`
Inputs	
Outputs	
Description	Called from signalAnalysisModule.acquirePowerModel.

Table 15. computeAlignedData Function

signalAnalysisModule.computeAlignedData(adjustedPowerData, adjustedTriggerData)	
Usage	`signalAnalysisModule.computeAlignedData(adjustedPowerData, adjustedTriggerData)`
Inputs	PowerData, TriggerData
Outputs	Aligned trace
Description	Uses the 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 16. correlation$_{p}earsonFunction$

sca.correlation_pearson(compressedData, hypotheticalPowerData)	
Usage	`M x N numpy matrix for power traces. Where M is the number of traces, N the number of samples per trace. M x L array the represents the hypothetical power values. Where M is the number of encryptions/Dycryptions. L is the number of key guesses (i.e for byte guess, L=256)`
Inputs	PowerData, TriggerData
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. See Fig -1.

Table 17. findMinimumGuessingEntropy Function

sca.findMinimumGuessingEntropy()	
Usage	`signalAnalysisModule.computeAlignedData(adjustedPowerData, adjustedTriggerData)`
Inputs	measuredData, hypotheticalPowerData,correlationType,stepSize,knownKey
Outputs	
Description	

Table 18. calculate$_a utocorrelationFunction$

sca.calculate$_a utocorrelation(alignedData)$	
Usage	`signalAnalysisModule.computeAlignedData(adjustedPowerData, adjustedTriggerData)`
Inputs	measuredData, hypotheticalPowerData,correlationType,stepSize,knownKey
Outputs	
Description	

Table 19. plottingModule.plotTrace Function

plottingModule.plotTrace(alignedData, ’ALL’, ’OVERLAY’)	
Usage	`plottingModule.plotTrace(alignedData, ’ALL’, ’OVERLAY’)`
Inputs	N x m numpy array.
Outputs	N/A
Description	Plots the numpy array provided see fig () for example.

Table 20. plottingModule.plotHist Function

plottingModule.plotHist(correlationData, globals.PEARSON)	
Usage	`plottingModule.plotHist(correlationData, globals.PEARSON)`
Inputs	correlationData, globals.PEARSON
Outputs	N/A
Description	Plots a histogram. See fig ()

Table 21. plottingModule.plotCorr Function

plottingModule.plotCorr(corrMatrix, corrType)	
Usage	`plottingModule.plotCorr(corrMatrix, corrType)`
Inputs	correlationData, globals.PEARSON
Outputs	N/A
Description	Plots correlation data. See fig()

Table 22. sampleSpaceDisp Function

postProcessingModule.sampleSpaceDisp(alignedData)	
Usage	`postProcessingModule.sampleSpaceDisp(alignedData)`
Inputs	M x N Numpy array 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

Table 23. compressData Function

compressData(windowedData)	
Usage	`compressData(windowedData)`
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. K is the number of samples per trace after compression.
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 24. compress Function

postProcessingModule.compress ()	
Usage	`postProcessingModule.compress (a, compressionLen, compressionType)`
Inputs	A, CompressionLen, CompressionType
Outputs	
Description	This function is called by postProcessingModule.compressData to do the compression.

Table 25. traceExpunge Function

postProcessingModule. traceExpunge(measuredData)	
Usage	`postProcessingModule. traceExpunge(measuredData)`
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 rang. 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 26. getKeyForAnalysis Function

dataGenerator.getKeyForAnalysis()	
Usage	`dataGenerator.getKeyForAnalysis()`
Inputs	N/A
Outputs	Key formatted as a hexadecimal string.
Description	Reads the key from file to be used in encrypting data. The key is saved in \$worksapce/\$projectName/\$output/

Table 27. getPlainText Function

dataGenerator.getPlainText()	
Usage	`dataGenerator.getPlainText()`
Inputs	N/A
Outputs	A list of blocks that represents plain text
Description	Generates random plain text or reads from file depending on configuration.

Table 28. generateRandomKey Function

dataGenerator.generateRandomKey()	
Usage	`dataGenerator.generateRandomKey()`
Inputs	N/A
Outputs	A A list of key bytes. Each byte is represented as a hexadecimal string.
Description	

Table 29. convertToHex(hexString) Function

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

Table 30. convertToHex(hexString) Function

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

Table 31. openOscilloscopeConnection Function

Oscilloscope_core.openOscilloscopeConnection()	
Usage	`Oscilloscope_core.openOscilloscopeConnection()`
Inputs	N/A
Outputs	N/A
Description	Connects to oscilloscope. It opens a socket using OSCILLOSCOPE_IP and OSCILLOSCPOE_PORT. Also gets the oscilloscope identifier.

Table 32. setOscilloscopeConfigAttributes Function

Oscilloscope_core.setOscilloscopeConfigAttributes()	
Usage	`Oscilloscope_core.setOscilloscopeConfigAttributes()`
Inputs	N/A
Outputs	N/A
Description	Configures the oscilloscope by sending commands (in text format) to the oscilloscope.

Table 33. initializeOscilloscopeDataStorage Function

Oscilloscope_core.initializeOscilloscopeDataStorage()	
Usage	`Oscilloscope_core.initializeOscilloscopeDataStorage()`
Inputs	N/A
Outputs	N/A
Description	Creates empty Numpy arrays for each enabled oscilloscope channel.

Table 34. Oscilloscope$_core.armOscilloscope()Function$

Oscilloscope_core armOscilloscope()	
Usage	`Oscilloscope_core armOscilloscope()`
Inputs	N/A
Outputs	N/A
Description	Instructs the oscilloscope to digitize channel specified in FOBOS configuration.

Table 35. populateOscilloscopeDataStorageAndAlign Function

Oscilloscope_core.populateOscilloscopeDataStorageAndAlign(traceCount)	
Usage	`Oscilloscope_core.populateOscilloscopeDataStorageAndAlign(traceCount)`
Inputs	traceCount
Outputs	N/A
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 36. closeOscilloscopeConnection Function

Oscilloscope_core.closeOscilloscopeConnection()	
Usage	`Oscilloscope_core.closeOscilloscopeConnection()`
Inputs	N/A
Outputs	N/A
Description	Closes socket that connects to oscilloscope

Table 37. openControlBoardConnection Function

Oscilloscope_core.openControlBoardConnection()	
Usage	`Oscilloscope_core.openControlBoardConnection()`
Inputs	N/A
Outputs	N/A
Description	Initializes connection to controlboarb, resets control board and reads control board and victim clocks

Table 38. initializeControlBoardConnection Function

usbcomm_core.initializeControlBoardConnection()	
Usage	`usbcomm_core.initializeControlBoardConnection()`
Inputs	N/A
Outputs	N/A
Description	Initializes the USB connection to the board. Called from OpenControlBoardConnection()

Table 39. usbcomm_core.sendTriggerParamsToControlBoard() Function

usbcomm_core.sendTriggerParamsToControlBoard()	
Usage	`usbcomm_core.sendTriggerParamsToControlBoard()`
Inputs	N/A
Outputs	N/A
Description	Reads the configuration and sent control message to oscilloscope to digitize channels per configration

Table 40. runEncrytionOnControlBoard Function

usbcomm_core.runEncrytionOnControlBoard()	
Usage	`usbcomm_core.runEncrytionOnControlBoard(traceCount)`
Inputs	the number of block used in encryption
Outputs	N/A
Description	Send key and data to control board to do encryption

Table 41. sendKeyToControlBoard Function

usbcomm_core.sendKeyToControlBoard()	
Usage	`usbcomm_core.runEncrytion0`
Inputs	the number of block used in encryption
Outputs	N/A
Description	

Table 42. sendBlockOfDataToControlBoard Function

usbcomm_core.sendBlockOfDataToControlBoard(traceCount)	
Usage	`usbcomm_core.usbcomm_core.sendBlockOfDataToControlBoard(traceCount)`
Inputs	the number of block used in encryption
Outputs	N/A
Description	runEncrytionOnControlBoard(traceCount) to send a block of data to control board

Table 43. saveControlBoardOutputDataStorage Function

Oscilloscope_core.saveControlBoardOutputDataStorage()	
Usage	`Oscilloscope_core.saveControlBoardOutputDataStorage()`
Inputs	N/A
Outputs	N/A
Description	Saves the trace Numpy array to disk.