FRANKFURT UNIVERSITY OF APPLIED SCIENCES

MASTER THESIS

# Efficient, Large-Scale Computation Techniques For The Evaluation of Side Channel Attacks

by

Mohammed Mohiuddin

***A thesis submitted in partial fulfillment for the degree of Master of Science***

*under the guidance of*

*1st Academic Supervisor:* Prof. Dr. Christian Baun

*2nd Academic Supervisor:* Prof. Dr. Eicke Godehardt

*Industrial Supervisor:* Mr. Robert Szerwinski

*in*

High Integrity Systems

Informatik and Ingenieurwissenschaften

April 2016

# Declaration of Authorship

I, MOHAMMED MOHIUDDIN, declare that this thesis titled, Efficient, Large-Scale Computation Techniques For The Evaluation of Side Channel Attacks and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a Masters degree at Frankfurt University of Applied Sciences.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ______________________________

Date: ______________________________

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

# *Abstract*

High Integrity Systems
Informatik and Ingenieurwissenschaften

Master of Science

by Mohammed Mohiuddin

Internet has changed the way people think, learn, connect, communicate, and pretty much do anything. All sort of information whether it is personnel information or organisation's private data, travels over internet. There is a great threat with the security of the information. Cryptography has provided a way of secure communication. Cryptography used today is the implementation of a mathematical algorithm on a physical hardware device. Hardware resources consume time, release heat, electromagnetic radiations, if such information is collected and analysed to retrieve key, it is referred to as side channel attack. The less amount of time and effort required to perform the side channel attacks has increased the threat to the security of the cryptographic devices. Many different type of attacks like simple power analysis (SPA)[1], differential power analysis(DPA)[2] and mutual information analysis(MIA)[3] have proven to be feasible. It is possible to design an attack based on different hardware, intermediate values and hypothetical models. The existence of many attacks, models has made the job of testing the vulnerability of a cryptographic device difficult. Evaluation labs and producers need an efficient way of testing the cryptographic device with minimal effort and time. The Welch's t test has been recommended by Cryptographic Research Inc.[4]. Performing t test at different orders requires parameters namely means, variances and sample sizes. Because of the large size of the traces to be processed and multiple passes the test consumes more time. There is a need to use the existing one pass algorithms[5][6] and run the time consuming components of the algorithm on cluster to reduce the time. This paper presents the design and implementation of the t-test, by exploiting the state of the art algorithms and technologies, and hence thereby enables the process to test the exploitability of the device efficiently.

# *Acknowledgements*

I would like to take this opportunity to express my sincere gratitude to my industrial supervisor, Mr. Robert Szerwinski at Robert Bosch GmbH for providing me this opportunity and for his time, guidance and patience. His valuable inputs helped me in all the time of research and writing of this thesis. I would like to thank my academic supervisor, Prof. Dr. Christian Baun for undertaking this work and for his continuous support throughout the thesis. I am grateful to my academic supervisor, Prof. Dr. Eicke Godehardt for his encouragement and support.

I am truly grateful to my parents and brothers for their belief, prayers and altruistic support throughout the course of my academic career. I would also like to thank my friends, Tallapaneni Vikram Kumar, Sameer Sheikh, Arun Verma, Sneha Sahu, Mohammed Asim Azmi, Gulam Mohammed Moosa, Nalikala Srinivas, Vineet Talwar, and Ashwini Bhagat without whom it wouldn't have been possible to come this far.

Lastly, I would like to thank everyone who have directly or indirectly, helped me in any regard.

Mohammed Mohiuddin

# Contents

# List of Figures

# Abbreviations

| | |
|---|---|
| ANSSI | Agence nationale de la scurit des systmes d'information |
| BSI | Bundesamt fr Sicherheit in der Informationstechnik |
| NIST | National Institute of Standards and Technology |
| SPA | Simple Power Analysis |
| DPA | Differential Power Analysis |
| MIA | Mutual Information Analysis |
| RAM | Random Access Memory |
| SSD | Solid State Drive |
| GPU | Graphical Processing Unit |
| CPU | Central Processing Unit |
| DDR3 | Double Data Rate Type 3 |
| NIC | Network Interface Card |
| DUT | device under test |
| NUC | Next Unit of Computing |
| IT | Information Technology |
| $j_c$ | First Sample Point Number For a Given Chunk |
| $M_d$ | Raw Moment at Order d |
| $CM_d$ | Central Moment at Order d |
| $SM_d$ | Standard Moment at Order d |
| $V_d$ | Variance At Order d |
| t | Total Number of Traces in Power Traces |
| s | Total Number of Sample Points in Power Traces |
| $l_t$ | Length of Traces in a Chunk of Power Traces |
| $l_s$ | Length of Samples in a Chunk of Power Traces |
| b | Bandwidth in Gbits/sec |

| | |
|---|---|
| $d_t$ | Data Type |
| d | Order of Attack |
| o | order |
| n | Sample Size (Number of Traces Inserted) |
| y | New Trace |
| $\mu$ | Mean |
| $\Delta$ | y - $\mu$ |
| $b_t$ | Number of bytes transferred over the network |
| $t_{tc}$ | Time taken to transfer chunk over the network |

# Chapter 1

# Introduction

Internet has changed the way people think, communicate, learn, entertain and just about do anything in today's world. Exchange of information ranging from personnel information to organisation's private data is exchanged over the internet. There is a great threat with the security of the information. Cryptography has provided a way of secure communication. Cryptography is based on the difficulty of breaking the mathematical algorithms using state of the art knowledge and hardware in reasonable time. If there are loopholes in the algorithm, implementation of the algorithm using software and hardware, an attack is possible. Attacks on the cryptography can be classified into two. First is the study of the cryptographic system for the purpose of breaking into the cryptography or cryptographic systems, referred as cryptanalysis. cryptanalysis focusses on the theoretical weaknesses in the cryptographic algorithm. Second, the collection and analysis of the information gained through the side channel to retrieve key, called as side channel attacks. Many different types of side channel attacks[[1][2][3][7]] are proven to possible and many more hypothetical models can be created based on intermediate values, hardware etc. One standard approach to test the exploitability of the cryptographic device irrespective of type, model and hardware is required in feasible time. This paper presents how t-test can be performed to test the cryptographic device vulnerability in an optimised way.

This chapter goes through the existing work on the topic and introduces required background information. section 1 discusses previous work in this area. Section 2 explains different concepts needed to understand this paper. Last section gives outline on how the paper is organised.

## 1.1 Previous Work and Aim of thesis

To perform t test at higher orders, we need calculate central moments using one pass algorithms [6]. Philippe Pebay proposed one pass approach to calculate central moments at different orders, which can be applied for any order[5]. This paper uses the idea mentioned in the paper proposed by Tobias Schneider and Amir Moradi [6]. Paper presents an approach to calculate parameters of t test, perform test, and if test is positive stop the t test otherwise continue the computations and test. One major difference is that the algorithm is being designed and implemented on cluster(parallel computation) in contrast to implementation on one system. Parallel computation reduces the execution time of the testing.

To my knowledge there are no published papers on utilizing the existing one pass algorithms to calculate higher order parameters of the t test and distribute the tasks on the cluster to increase the performance and decrease the time of execution. The aim of the thesis is to identify the expensive components of the computation, develop an algorithm to distribute and combine the tasks in these components. This paper also presents the analysis on the performance of the algorithm with different parameters.

## 1.2 Background Information

This report is a combination of different concepts. It is necessary to have basic knowledge of these concepts, to be able to understand the report. In this section, we will go through the basics of cryptography, side channel attacks, t test in brief.

### 1.2.1 Cryptography

Cryptography is a process of storing and transmitting data in way that only the intended participants are able to understand the message using secret keys. According to the text book *Introduction to Modern Cryptography*[8], Cryptography is defined as *the scientific study of techniques for securing digital information, transactions, and distributed computations*. As discussed earlier, information security is an inevitable part of modern society. The key goals of information security solved by the cryptography are *Confidentiality*, *Data Integrity*, *Authentication*, *Non-repudiation* and *Authorization*. In order to be able understand this section, let us introduce some terminology. Original message, algorithm used to modify the message, modified message are called *plaintext*, *cipher* and *ciphertext* respectively. The process of converting the plaintext into ciphertext is called *encryption* and the process of converting the ciphertext back to plaintext is referred to as *decryption*. Key used for encryption and decryption are referred to as $k_e$ and $k_d$ respectively. Based on whether the sender and receiver uses same or different keys, the cryptography can be divided into two types. Brief discussion about these two is as follows.

#### Symmetric Cryptography

In symmetric key cryptography, sender and receiver use same key for encryption and decryption. Secrecy achieved through this encryption depends on how well the key can be kept private. Advanced Encryption Standard (AES), Data Encryption Standard are examples of the symmetric key cryptography. Symmetric cryptography is fast and efficient, hence it is used in the encryption of the message. Problems associated with this cryptography include key sharing, key management(need to create a new key for every participant), integrity(it is possible to detect if the information is altered) and repudiation[8].

#### Asymmetric Cryptography

Asymmetric cryptography is also known as public key cryptography. Different keys are used for encryption and decryption. Two keys are required to encrypt and decrypt, one key which is visible to everyone is called public key and the other key which is available only to individual is called private key. There is no need of creating a new key for every pair of participants. Most of the disadvantages associated with symmetric cryptography are solved by asymmetric key cryptography, but its execution is far slower than the symmetric key cryptography. In real world, a combination of both types of cryptography is used for achieving all the goals of information as mentioned earlier[8].

### 1.2.2 Side channel attacks

Two parts of cryptography used today are mathematical algorithm and its implementation on a physical hardware device. Cryptographic algorithms are assumed to mathematically secure when the key is sufficiently large, as the brute force attacks are infeasible. As discussed earlier, the study of finding the theoretical weaknesses in the algorithm to break in is referred to as cryptanalysis. Microprocessors need some time and power for performing the given task. They release heat, radiate electromagnetic field etc. Collection and analysis of the information that physically leak out of a device is referred to Side channel attacks. The extend of the success in an attack depends the measurement equipment used to collect information [9]. If proper care is not taken, side channel attacks can compromise the security of a cryptographic system. Side channels include amount of power consumed, electromagnetic emanation, execution time of the different instructions in a cryptographic algorithm. A brief description about the types of side channel attacks is as follows.

**Timing Attack** - In timing side channel attack, timing details collected from the implementation of the cryptosystem are utilized to break it. It is known that the execution time of algorithms may depend on the input given. By profiling the execution time of the algorithm for different inputs, it is possible to retrieve the key used by the cryptosystem. An example of such an attack was designed by Kocher to expose secret the used for RSA decryption[7][10].

**Power Monitoring Attack** - is a type of side channel attack, where an attacker analyses the power consumption of the cryptographic hardware device. It is possible to extract the key by exploiting the directly available interfaces i.e. non-invasive attack. Power analysis attack is based on the fact that the power consumed by a cryptographic hardware device might depend on the operation they perform and data they process. Simple power analysis (SPA)[11] and differential power analysis (DPA)[12] are examples of this type of attack.

**Electromagnetic attacks** This attack exploits the correlation between the secret data and variations in the radiations of the device under attack. Simple electromagnetic analysis (SEMA), where one or few traces are used to determine the key. Differential electromagnetic analysis (DEMA) is considered to be more powerful because the attacker doesnt need to know anything about the device but requires large number of traces[13].

### 1.2.3 Welch's t test

T test was developed by William Sealy Gosset in 1908. It is a hypothesis test, used to determine whether two given data sets are significantly different from each other. The assumption is that the two data sets follow normal distribution. It is used when the variances of the population are unknown or the sample sizes are small or unequal[14]. Two sample location t test, tests whether the means of two populations are equal. The t test used in this paper is a variant of two sample location t test, where variances of the two populations are assumed to be same. If the assumption is dropped, then it is called Welch's t test[15].

## 1.3 Thesis Outline

Chapter 2 goes through the existing solutions, problem statement, motivation of thesis and ends with an explanation of the tools and technologies used in the thesis. Readers, who are not interested in the actual implementation details can skip the fourth section of the second chapter. Mathematical background, theoretical analysis of the algorithm, design for implementation on single node and cluster are discussed in the third chapter. Chapter 4 has implementation details on single node, cluster and benchmarking scenarios. Readers interested in the implementation are encouraged to read the chapters 3 and 4 completely. Chapters 5 and 6 give details how the algorithm performs with different parameters and suggests optimal chunk size for a given set of hardware and algorithm parameters.

# Chapter 2

# State of The Art

This chapter explains the need for a design and implementation of a fast and efficient methodology to test the vulnerability of device under test (DUT), followed by existing solutions and goal of the thesis, and then the tools and technologies used in the thesis.

## 2.1 Existing Solutions

To abstract out dependency on the hardware architecture of the device under test, types of attacks and hypothetical models Welch's t test is used to perform a fixed vs random t test[4]. Though welch's t test has been used in research works[[16][17][18]], but they don't provide any information on the challenges and practical problems in performing the test at higher orders. Calculation of parameters requires mulitple accesses of the traces i.e. different passes for means, moments and variances. Philippe Pebay's work[5] on the calculation moments at higher orders, gave a way for the possibility of calculating the parameters in one pass. This work was not intended specifically for the t test and does give any information on using it to calculate t parameters or perform t test. Research work contributed for Amir Moradi and Tobias Schneider presents the theoretical background as well as the practical approach to be followed for performing the t test[6]. The calculation of parameters in univariate and multivariate settings, at different orders is clearly presented in the paper.

## 2.2 Problem Statement

Side channel attacks pose a serious threat to the information security. Integrating the countermeasures into the commercial security-enabled devices has become inevitable.

Many different type of attacks like simple power analysis (SPA)[1], differential power analysis(DPA)[2] and mutual information analysis(MIA)[3] have proven to be feasible. It is also possible to design an attack based on different hardware, intermediate values and hypothetical models. The existence of many attacks, models has made the job of testing the vulnerability of a cryptographic device difficult. Standardization bodies like NIST are looking for a standard methodology, which is independent of the types of attacks, underlying architecture of the device and hypothetical models. Governing bodies like BSI have common criteria evaluations, that the evaluation labs need to test the possibility of an attack on the device under test. The time and resources required to test cryptographic devices are making the job of testing infeasible for both the producers and standardization organisations.

The Welch's t test has been recommended by the Cryptographic Research Inc.[4] and used in research works[[16][17]] to check the efficiency of the countermeasures. To avoid the dependency on the hardware and types of attacks, t test is being used for testing the vulnerability of the cryptographic devices. Since, the traces are large and multiple passes are required to access the traces; calculation of parameters takes more time. Calculation of moments at different orders is presented in the paper contributed by Philippe pebay[5]. The detailed procedure of performing the t test at different at higher orders is available in the paper presented by Tobias Schneider and Amir Moradi[6]. Since the size of traces is large, usually millions of traces, time taken to calculate the parameters of t test is large. It might take days or months to compute the parameters depending on the size of the traces. This makes the testing infeasible for the evaluation labs and producers of security-enabled devices.

## 2.3 Technologies

### 2.3.1 Cluster

The CPU cluster[19] is made using the Ubuntus cluster boxes. Each cluster box contains 10 Intel NUCs (Next unit of computing). The configuration of each NUC is as follows [20].

- Intel @i5-3427U CPU
- 16 GB DDR3 RAM
- 120GB SSD Storage
- Intel HD4000 GPU

- Intel Gigabit NIC

### 2.3.2 Python Programming Language

Python being an interpreted language, gives an impression of being slow. It raises a question, then why it is used in high performance computing. It is more of used as a glue between highly optimised libraries built in different language rather than a core developing language. Scientific software developers need a language which is flexible, easy to learn and use, simple to maintain and easy to adapt to the new requirements and complexities of the problems, so less time on spent on the software and more on ideas[21]. In Bosch, one of the constraints of implementation is the usage of Python programming language, to be able to integrate with the existing modules. Optimised libraries like NumPy[22], SciPy[23], Matplotlib[24] are used in the implementation.

#### Doctest

Doctest, along with inbuilt modules are used to verify the results obtained by the implementation are correct. Doctest is a simple, easy to use unit testing framework, which can be included in docstring. When doctest is run with a small function at the end of the file, it searches for the text of code, which looks like python sessions, executes them and verifies with the expected output. For more information refer to the official documentation of python[25].

#### Profilers

In order to be able to decide which parts of the algorithm should be implemented on the cluster, information on the execution time is needed. Therefore, it is used in this thesis. Profilers give set of statistics on the execution of a program. Two types of profilers used often are time and memory. Many time profilers are available for python. We are going to discuss only about two of the mentioned in the python documentation[26]. They are,

- **profile** It is a pure python module. Profile interface is imitated by cProfile and adds significant overhead to the profiled programs.
- **cProfile** It can be used for long running programs with reasonable overhead. It is based on lsprof, contributed by Brett Rosen and Ted Czotter [27]. Since, it is efficient and already available with distribution, it is best choice for time profiling.

### 2.3.3 IPython

IPython refers Interactive Python. The goal of building IPython was to create a comprehensive environment for interactive computing. The main components of IPython are:

- An interactive powerful python shell
- IPython Web based notebook, which allows inclusion of code, text, inline plots, rich media along with the core features.
- High performance tools for parallel computing.

IPython notebook extends the console based interactive computing approach to provide web based application capable of capturing the whole computational process. Notebook can combine the developing, executing code, documenting, and results at one place.

#### IPython Parallel Computing

IPython parallel tools are used to implement the parallelization part of the thesis. IPython provides sophisticated and powerful architecture for distributed computing. It supports many different types of parallelism, which include Single program multiple data parallelism, Multiple program multiple data parallelism, Message passing using MPI, Task farming, Data parallel and combinations of these. It allows parallel applications to be developed, executed, monitored and debugged interactively. Basics of the IPython parallel required to understand the thesis are described below 2.1.

IPython architecture comprises of four components.

- IPython engine
- IPython hub
- IPython Scheduler
- Controller client

All the components are available in IPython.parallel package and are installed with IPython. For more information on the usage see the official documentation of IPython [29]. IPython supports all types of parallel applications to be developed, executed, monitored and debugged interactively.

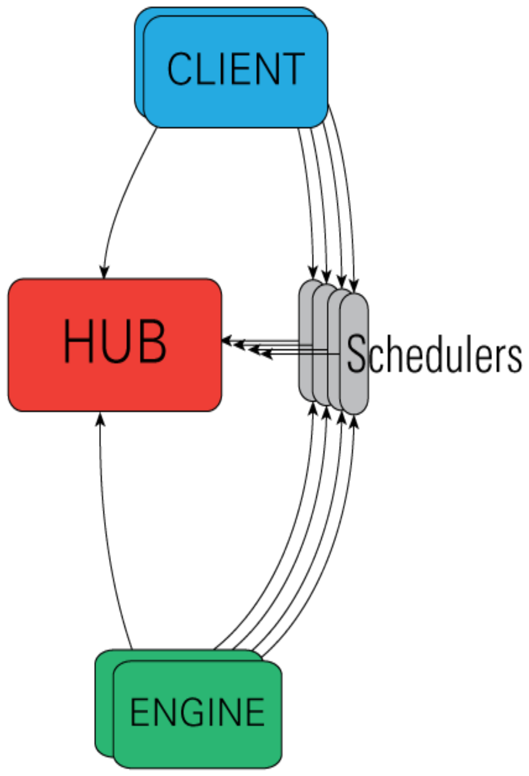


Figure 2.1: IPython Parallel Architecture[28]

### IPython Engine

The IPython engine is like a normal python instance that takes python commands over a network. The engine is capable of handling objects that are sent over the network. When multiple engines are started together, parallel and distributed computing is possible. IPython engines get blocked while executing the user code but the controller handles the requests and provides a asynchronous interface to the users. In description of the implementation we use engine instead of the IPython engine.

### IPython Controller

The IPython controller provides an interface to work with group of engines. IPython controller is an accumulation of processes to which IPython clients and engines can connect. Controller consists of a Hub and a set of Schedulers. Schedulers and hub run in separate processes but on the same machine. Schedulers take commands from the local as well as remote machines. Engines connect to the schedulers. Users who want to utilize the engines, can send the commands to the scheduler. Basically there are two different ways of working with controller. A view object needs to be created with a subset of engines or all engines. First way, called Direct interface is used to run commands on specific engines by explicitly addressing them. Second way, called LoadBalanced interface is used to dynamically assign the tasks on the available engines through the scheduler. Scheduler handles all the issues like load balancing, node failures by itself, giving freedom to the user. In both ways, View.apply() method is used. In this report, direct view means an object of the Direct interface and load balanced view means an object of LoadBalanced interface.

#### Hub

It is the center of IPython cluster. Hub keeps track of all the details of the cluster, which includes clients, engine connections, schedulers, task requests and results. Main purpose of the hub is provide the state of the cluster and reduce the information required to establish connections between client and engines.

#### Schedulers

All the actions that are performed on the engines go through the scheduler. Engines block when executing the user code, but the scheduler provides a asynchronous interface to the user.

## 2.4 Motivation and Goal

As discussed earlier, if proper care is not taken, side channel attacks pose a serious threat to the security of the cryptographic devices. Evaluation labs and producers of cryptographic-enabled devices need a fast, robust approach of testing to test the devices. Robert Bosch GmbH is a global company producing wide range of products. The security and IT systems group at Bosch is investigating a systematic approach to penetration testing. In particular the resistance level of Bosch products against the side channel attacks. One pillar of approach is the implementation of efficient large scale computations of the necessary algorithms. The state-of-the-art algorithms are available for performing Welch's t test, but there is no existing work on the implementation of these algorithms in distributed environment. Parallelization of the algorithms, reduces the execution time, making the testing of cryptographic devices fast.

# Chapter 3

# Design

## 3.1 Characteristics Of Traces

Power traces are vectors of voltage values recorded at different points of time with a digital sampling oscilloscope. An oscilloscope is connected to an electromagnetic probe or measurement circuit and its power is equal to the product of voltage and current. As a result of this, the power consumption of the cryptographic hardware device is proportional to the measured voltages. Power Traces can be thought as a matrix of elements, where each row corresponds to a single *trace*, each column to *sample point* and element as *power trace*. Every value in a particular trace is independent of each other because they belong to different points of time. However, every value in a particular sample point is related to each other. The value might depends on the data being executed by device or operation being executed by the device[30].

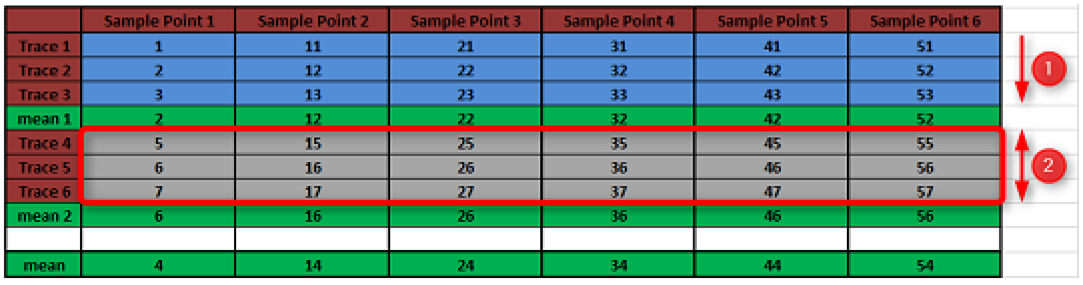
### 3.1.1 Parallel Properties of Traces

Let us consider the below set of variables -

- t: the total number of traces in the power traces
- s: total number of sample points in the power traces
- $l_t$: number of traces in the chunk
- $l_s$: number sample points in the chunk
- $j_c$: first sample point number in the chunk
- n(sample size): number of traces inserted before the chunk

Now, suppose Q and Q' be two sets with different sample sizes. Then a chunk is a subset of power traces consisting of - parts of traces and sample points. Also as shown in figure 3.1 the following matrix shows power traces and we are interested in finding mean of all the sample points. The actual power traces are floating point values, but we have selected integers for the sake of better understanding.

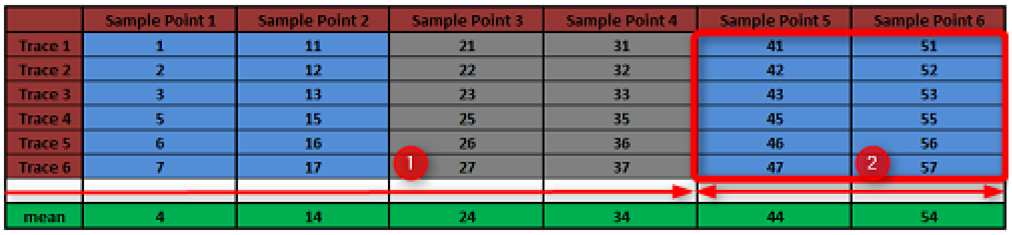
### Division Of Power Traces in First Dimension (Traces)



| | Sample Point 1 | Sample Point 2 | Sample Point 3 | Sample Point 4 | Sample Point 5 | Sample Point 6 |
|---|---|---|---|---|---|---|
| Trace 1 | 1 | 11 | 21 | 31 | 41 | 51 |
| Trace 2 | 2 | 12 | 22 | 32 | 42 | 52 |
| Trace 3 | 3 | 13 | 23 | 33 | 43 | 53 |
| mean 1 | 2 | 12 | 22 | 32 | 42 | 52 |
| Trace 4 | 5 | 15 | 25 | 35 | 45 | 55 |
| Trace 5 | 6 | 16 | 26 | 36 | 46 | 56 |
| Trace 6 | 7 | 17 | 27 | 37 | 47 | 57 |
| mean 2 | 6 | 16 | 26 | 36 | 46 | 56 |
| | | | | | | |
| mean | 4 | 14 | 24 | 34 | 44 | 54 |

FIGURE 3.1: Division Of Power Traces in First Dimension (Traces).

The red rectangle in figure 3.1, represents new chunk. We can divide the task based on the traces. The traces 1, 2 and 3 can be assigned to one process and the traces 4, 5 and 6 can be assigned to another process as shown in the figure 3.1. While merging the results we need to spend additional resources. The order of merging results is not important. In given example, process one returns the mean of the first three traces for all sample points. Second process returns the mean of the next three traces for all of the six sample points. The extra step here is to use sets properties to combine the results. The mean of all sample points, for the first three traces can be calculated by one process. The mean of all sample points, for the second three traces can be calculated by any other process. Two means of the first sample point are calculated to be 2 and 6. These means can be merged using the formula $(n_1 * \mu_1 + n_2 * \mu_2)/2$. Where n1,n2 and $\mu_1, \mu_2$ are number of traces(sample sizes) and means of sets $Q_1, Q_2$ respectively. In given example, the number of chunks are 2 and number of traces(sample size) are equal to 3, but it is not neccesarily true always. The number of chunks and their size may change. Excluding means, information of the number of traces already inserted and number of traces in the current chunk are required. Detailed explanation about combining the results is explained in the next section. Different colors are used for trace points to differentiate power traces into chunks. Please note that means(green color) are not part of the traces.



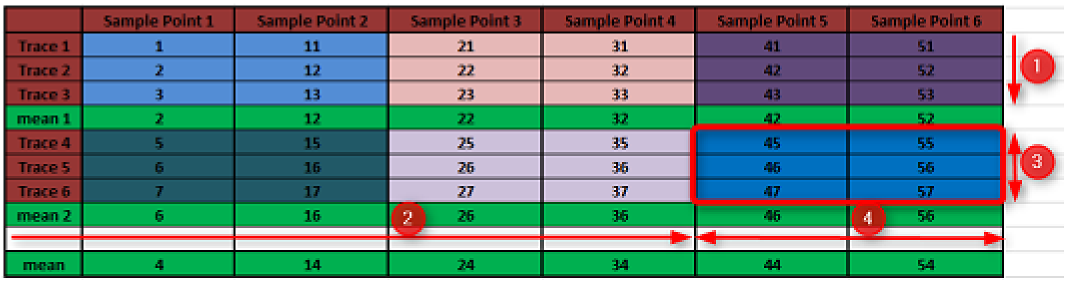
| | Sample Point 1 | Sample Point 2 | Sample Point 3 | Sample Point 4 | Sample Point 5 | Sample Point 6 |
|---|---|---|---|---|---|---|
| Trace 1 | 1 | 11 | 21 | 31 | 41 | 51 |
| Trace 2 | 2 | 12 | 22 | 32 | 42 | 52 |
| Trace 3 | 3 | 13 | 23 | 33 | 43 | 53 |
| Trace 4 | 5 | 15 | 25 | 35 | 45 | 55 |
| Trace 5 | 6 | 16 | 26 | 36 | 46 | 56 |
| Trace 6 | 7 | 17 | 27 | 37 | 47 | 57 |
| | | | | | | |
| mean | 4 | 14 | 24 | 34 | 44 | 54 |

FIGURE 3.2: Division Of Power Traces in Second Dimension (Sample Points).

### Division Of Power Traces in Second Dimension (Sample Points)

We can assign any sample point or a set (chunk) of sample points to any process, since they are independent from each other with respect to time. The process can perform the computations independent of the other. There are six sample points in the table 3.2 and mean of each sample point can be calculated by any process or node. The important thing to note here is the order of merging the results. The means of first chunk are 27, 37 and second chunk are 47 and 57, respectively, if they are merged in reverse order, it would be wrong. While merging the results, first sample point number and the number of sample points in a chunk are required. Suppose that the results of the last chunk shown in figure are to be merged. The first sample point number is 5 and number of sample points in chunk are two, then the results can be stored at the locations 5 and 6 of means.

### Division Of Power Traces in Both Dimensions



| | Sample Point 1 | Sample Point 2 | Sample Point 3 | Sample Point 4 | Sample Point 5 | Sample Point 6 |
|---|---|---|---|---|---|---|
| Trace 1 | 1 | 11 | 21 | 31 | 41 | 51 |
| Trace 2 | 2 | 12 | 22 | 32 | 42 | 52 |
| Trace 3 | 3 | 13 | 23 | 33 | 43 | 53 |
| mean 1 | 2 | 12 | 22 | 32 | 42 | 52 |
| Trace 4 | 5 | 15 | 25 | 35 | 45 | 55 |
| Trace 5 | 6 | 16 | 26 | 36 | 46 | 56 |
| Trace 6 | 7 | 17 | 27 | 37 | 47 | 57 |
| mean 2 | 6 | 16 | 26 | 36 | 46 | 56 |
| | | | | | | |
| mean | 4 | 14 | 24 | 34 | 44 | 54 |

FIGURE 3.3: Division Of Power Traces in Both Dimensions.

Division in both dimensions gives more flexibility in experimenting and hence improve the performance. Consider the chunk shown in the figure 3.3. Suppose the chunk (sample point 5,6 and traces 4,5,6) shown in red rectangle as the new chunk and chunk (sample point 5,6 and traces 1,2,3) as the existing chunk (in this example number of traces in existing chunk is 3 but in actual, it will vary depending on the number of chunks before the new chunk). To merge the new results of the remote execution, identification of chunk and existing results details are required. Chunk can be identified by first sample point number and the number of traces inserted before the chunk. In given example, $j_c$ is 5 and n is 3, $l_t$ is 3 and $l_s$ is 2, using these details highlighted in the figure 3.3 can be identified as the sixth chunk. After identification, we can access the means (42, 52) and sample size(3) of the old chunk and update it according to the new chunk using sets properties. Only the means of sample points 5 and 6 are updated.

## 3.2 Concept and Design

To perform t test we need means, variances, sample sizes and degrees of freedom at different orders for every sample point. When the standard deviation and sample size of two samples is nearly same, degrees of freedom can be calculated as the sum of sample sizes. If the absolute value of t is greater 4.5 and degrees of freedom is more than 1000 the probability to reject the null hypothesis is less than 0.00001. Which results in a confidence of greater than 0.99999. In case of traces, the number of traces considered is usually large than 1000, so we can neglect degrees of freedom [6].

So, we require only three parameters to decide the exploitability of the device. Normally we need multiple passes to fetch the required data and calculate parameters. One pass to calculate the means of the power traces, since means are requires in the calculation of variance. Second pass for the calculation of variance. Earlier it was not possible to calculate means and variances in one pass using a single formula. Now it is possible because of the contributions of Philippe Pebay, Amir Moradi and Tobias Schneider.

**Shape of Distributions**

**Moment** - The d-th moment of a real valued continuous function about a value c can be written as

$$\mu_n = \int_{\infty}^{\infty} (x-c)^d f(x)dx$$

**Raw Moment** - In the above formula if the value of c is replaced with zero i.e. the moment of the function about zero is called raw moment. First raw moment is mean.

**Central Moment** - The moment about the mean i.e. if the value of c used is mean it is referred to as central moment.

Design consists of six steps. The first four contribute for the calculations of means and variances. The last two methods actually perform the test. The procedure can be divided into two classes. First one is called *Parameters* and second as *Ttest*. One important question is how often we want to perform t test. The algorithm is designed

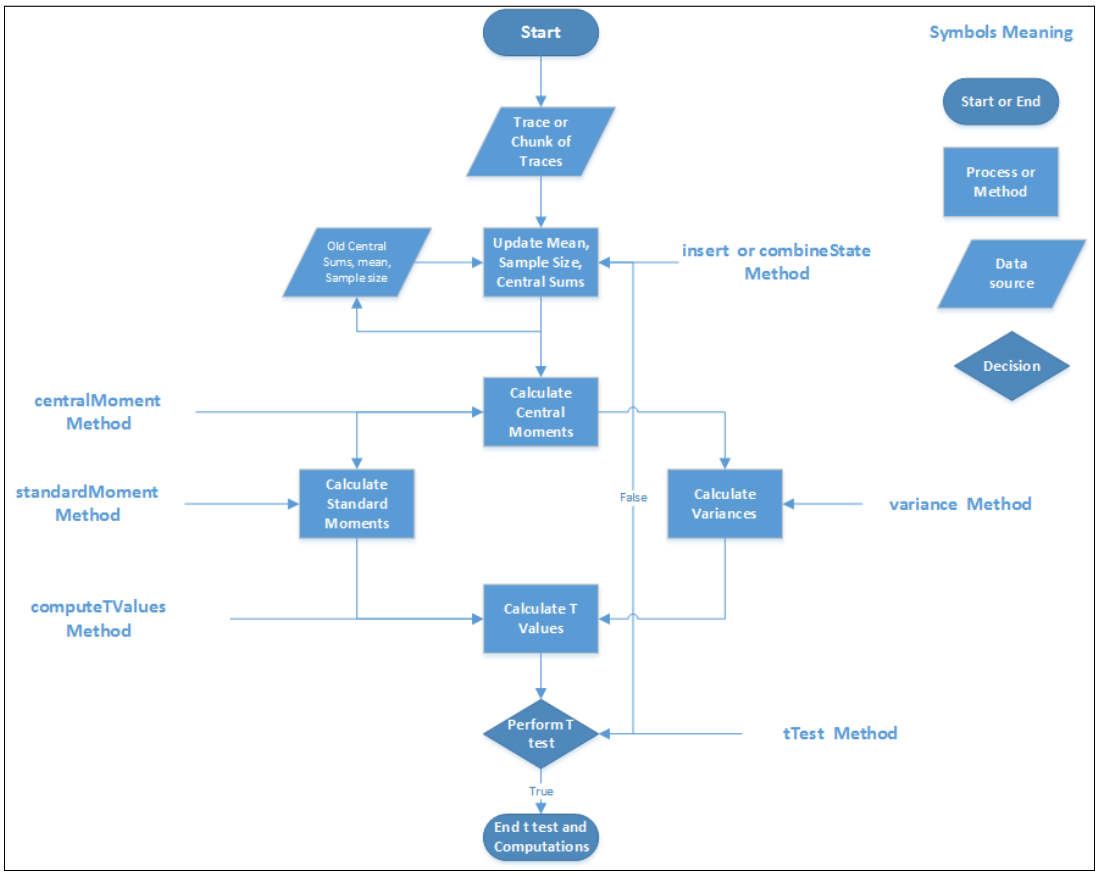


FIGURE 3.4: Flow Diagram of Algorithm.

that it can update parameters and test for every new trace. The amount of traces are very large and it is not efficient to process the whole traces at a time. Suppose, we have 10,000,000 traces, if it is possible to conclude that the device is exploitable by using just first 10,000 traces, it would save time and effort.

The developed algorithm would insert traces one by one but it should be possible to perform the test for specific number of traces, say, *size* which we should be able to decide during run time. Here the traces are being divided in one dimension i.e. traces and second dimension sample points is not divided. Every chunk of data has some fixed number of traces except the last chunk and all sample points. So the test is not performed for every trace, it means the second part (calculating t values and testing) of calculation is done only once for *size* number of traces. But what about the other steps in first part of the algorithm, do we need to run all methods for every trace? The answer is no. The calculation of central sums is the only step which should run iteratively for every trace. There is no need to update central moments, standard moments and variances, when the test is performed once for *size* number of elements. So out of the six steps, only one step (central sums) runs iteratively.

Approach Summary:

- Update central sums iteratively for every trace.
- Calculate means and variances for every *size* number of traces.
- Calculate t values and run t test for every *size* number of traces.

#### 3.2.0.1 Central Sums

Central Sums is defined as the sum of the d-th power of difference of the random variable from the mean. Where o is the order of the of the central sum. Mathematically it can be expressed as follows 3.1[5].

$$CS_o = \sum_i (x_i - \mu)^o \tag{3.1}$$

The iterative formula for updating the central sums for every trace when o>2 is as follows[5].

$$CS_{o,Q'} = CS_{o,Q} + \sum_{k=1}^{o-2} \binom{o}{k} CS_{o-k,Q}(\frac{-\Delta}{n})^k + (\frac{n-1}{n}\Delta)^o[1 - (\frac{-1}{n-1})^{o-1}] \tag{3.2}$$

where Q is a set of traces with cardinality of n-1 and Q'(includes the new trace, y) is a set of traces with cardinality of n. Suppose the name of the implementation of this step to be *insert*. The input to this method would be a chunk of power traces and it is expected to return the updated central sums and sample size. Please note the word update, means there is something already and we are updating it. In the beginning, since there are no traces, central sum and sample size would be zero. The state of the central sums and sample size needs to stored, so that it is accessed and updated for every trace in one specific chunk and for different chunks. The state can be stored in instance variables. Sample size is the number of traces inserted, this value can be stored in a single integer type of variable. As it can be seen from the equation number 3.6, the calculation of variance requires central moments up-to double the order of attack. The size of the central sums variable depends on the order and type would be a floating point array (floating point because the traces used and central sums calculated are floats). Suppose if the maximum order of attack is 5, the data structure used to store central sums for one sample point would be a floating point array of ten elements.

#### 3.2.0.2 Combine state

Combine state is used combine the results of two independent central sums. Suppose if the central sums are calculated for every chunk of 1000 traces. The central sums are updated for every new chunk. The size of already considered chunk (multiple of 1000 traces e.g. 10,000) and the new chunk (1000 traces) may be different. These two different central sums can be combined to update the state (central sums of 11,000 traces)[6]. Following formula 3.3 can be used to combine the state.

$$CS_{o,Q'} = CS_{o,Q} + \sum_{k=1}^{o-2} \binom{o}{k} CS_{o-k,Q} (\frac{-\Delta}{n})^k + (\frac{n-1}{n}\Delta)^o [1 - (\frac{-1}{n-1})^{o-1}] \tag{3.3}$$

#### 3.2.0.3 Central Moments

Central moments is obtained by the division of central sums and the sample size of the set used in the calculation of central sums. Calculation of central moments is referred by *centralMoment* in the flow diagram and implementation of algorithm.

$$CM_o = \frac{CS_o}{n} \tag{3.4}$$

Calculation of central moments is straight forward. It is obtained by dividing the central sums and sample size. The input to this method would be central sums and sample size and it would return central moments. The size and data structure used to store central moments is same as central sums.

#### 3.2.0.4 Standard Moments

Standard moments are normalized central moments by the variance. Standard moments can be derived from the following formula 3.5. These are used as means of the traces while performing t test. Where d is order of attack. This step of calculation is referred by *standardMoment* in the flow diagram.

$$SM_d = \frac{CM_d}{(\sqrt{CM_2})^d} \tag{3.5}$$

There are two exceptions to this formula i.e., standard moments at order one and two. Standard moment at order one is raw moment at order 1. Standard moment at order

2 is central moment at order 2. Standard moments can be calculated from the central moments. Standard moments at d requires central moments up-to order d. Please note that central sums and central moments are calculated up-to order 2d because calculation of variances at order d requires central moments upto order 2d. The input to the method would be central moments and it would return standard moments.

#### 3.2.0.5 Variances

After the calculation of means and variances for the given number of traces. T values and degrees of freedom can be obtained by using the following formulas. Calculation of variance is called *variance* in flow diagram and implementation.

$$V_d = \frac{CM_{2d} - (CM_d)^2}{CM_2^d} \tag{3.6}$$

As it can be seen from the equation, calculation of variances requires central moments up-to order 2d. This is the reason for finding central sums and central moments up-to order 2d. The expected input given to the method is central moments up-to double the order and it would return variances.

**Compute Mean and Variances**

From the discussion in the beginning of this section, second step is the calculation mean and variances once for every chunk. It is better to combine the three methods i.e. calculation of central moments, standard moments and variances into one method say *computeMeanVariance*, to provide an public interface. So that, when this method is called by passing central sums, it should internally call three methods and return means and variances. This would make the process simple. The process would be to update central sums for every chunk, when this updated value is passed as input to the *computeMeanVariance*, it would return means and variances of all the samples at different orders.

#### 3.2.0.6 T values

The variance at order one is second central moment $CM_2$. Variance at orders two and more can be calculated using second and fourth central moments using the formula.

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}} \tag{3.7}$$

We are not interested in finding the degrees of freedom due to the reasons mentioned in the beginning of this section. t values can be calculated from the means, variances and sample sizes of random and fixed traces. There is no need of calculating the degrees of freedom as mentioned earlier. This method would accept means, variances and sample size as input parameters and return t values up-to order d for each sample point. This method is called *computeTValues* in the flow diagram and implementation of algorithm.

#### 3.2.0.7 Perform Test

Check whether the absolute value of t for different samples at different orders is greater than 4.5. If t value is greater than 4.5, it can be declared that the device under test might be exploitable. Instead of displaying message that device is exploitable, it would be better to return sample number and order as it would be helpful to the producers.

This would be the actual public method called from the client. This method would internally call the t values method to get t values and compare them. The input to this method would means, variances and sample sizes of random and fixed data and output would be a print statement, which displays the whether the device is exploitable and if yes at which order.

## 3.3 Single Node

The algorithm is capable of performing iteration for every trace, but it is not recommended to perform test for every trace. Because algorithm would spend more time in the calculation of central moments, standard moments, variances, t values and perform t test for every trace. It is not efficient to perform the test for complete power traces, as the leakage can be detected by processing a fraction of total traces. So a better approach would be perform test for every chunk of specific size, size may be decided according the requirement of the client. One more reason for chunking is that the complete power traces dont fit in the main memory. Next question is in how many dimensions do we want to chunk? As mentioned in the section 3.1 chunking is possible in both dimensions. Since we are chunking to avoid memory problems and save time, it is fine if we are chunking in one dimension i.e. first dimension. If there are memory problems with chunking in one dimension, then number of traces per chunk can be decreased or use the chunking in both dimensions. Comparing with the three steps in the algorithm mentioned in the section 3.2, the steps two and three not repeated for every trace but for every chunk.

## 3.4 Parallel

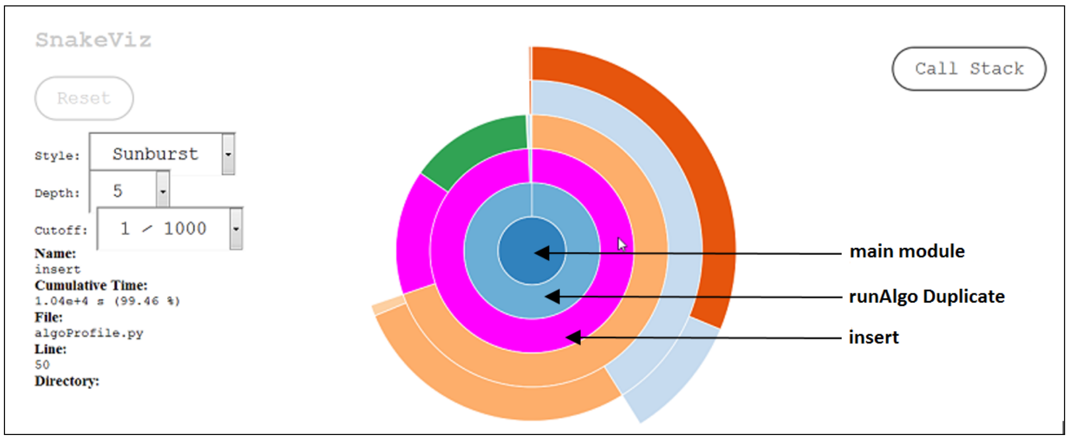
### 3.4.1 Analysis of Parallel Components of Algorithm

With the perspective of methods, there are six methods. If the point of view is steps involved in performing the test, then there are three steps. The decision of which methods or steps are executed in parallel is based on how expensive each method or step is compared to others in the algorithm. Profilers can be used to analyse the time and memory consumed by each method. Python profiler cProfile is used to analyse the execution time of the algorithm. The profiling data can be viewed on the console or stored in a file for analysis or view it on some available graphical user interfaces. The probable options for viewing the data graphically were ***runsnakerun*** and ***snakeviz***. Runsnakerun was not available in the distribution installed on the system. Whereas snakeviz available and hence used in the the analysis.

A couple of changes have been made for the sake of profiling. First, the normal task of the tTest method is to check t values and if any value is greater then 4.5 return the sample point and order. Here the method is modified to skip the body of the method using pass keyword, since we are interested in the complete execution of the algorithm to measure relative time consumed. The combineState method is called twice to make execution timings accurate. Though the central sums calculated are incorrect, it is not in the interest of the experiment.

The test is performed in a ipython notebook. The combineState method code can be seen in the code 4.1. The test case is written in a method called *runAlgoDuplicate*. The modified algorithm is imported as algo_profile.

Snakeviz uses a sunburst kind of view to display profiles. Each differently coloured arc in the display represents distinct functions. A function may or may not posses child functions. In case there exists any child function, the calling function is addressed as parent and is marked with an arc for each child function it calls. The central-most circle in the figure 3.5 represents the root function-*main module*, surrounded by set of sub-functions being called by the root. In a similar fashion these second level child functions are again surrounded by functions that are called by them, and so on. The angular width of each arc around its inner arc defines the percent of time-taken by the particular called



```
Style: Sunburst
Depth: 5
Cutoff: 1 / 1000
Name:
insert
Cumulative Time:
1.04e+4 s (99.46 %)
File:
algoProfile.py
Line:
50
Directory:
```

FIGURE 3.5: Execution Time of Different Methods in the Algorithm

function. In other words, if an arc covers most of the area around its immediate inner circle, it implies that this particular function consumes most of the time of its calling function. Similarly a thinner arc would mean that the function is hardly taking any time. From the figure 3.5 it is clearly understood that the *main module* function spends most of its time in executing the function - *runAlgoDuplicate*, represented in light blue. The *runAlgoDuplicate* function in turn spends most of its time executing the *insert* function shown in pink. On the left side of the figure 3.5, we can see that the percent of execution time consumed by *insert* function is approximately 99.46%. This means that *insert* is the only expensive method consuming most of the time. Therefore, calculation of central sums(*insert*), is a good choice to execute in parallel.

# Chapter 4

# Implementation

Similar to the design chapter, implementation is divided into three sections and an additional section for benchmarking. Section 1 explains the implementation of the algorithm. Section 2 discusses the usage of algorithm on single node by using chunking in one dimension. The procedure of using the algorithm in parallel by chunking in two dimensions is mentioned in the section 3. Chapter ends with the discussion of the implementation of the benchmarks.

## 4.1 Algorithm

To detect vulnerability of the device under test (DUT) at order d, we need to perform t test at order d. In order to perform t test, we require two parameters i.e. means and variances at order d. Calculation of means up to order d requires central moments up to order d. Calculation of variance up to order d requires central moments up to order 2d. Central sums up to order 2d are required to calculate central moments up to order 2d. The actual implementation would go through calculation of central sums, central moments, standard moments and variances and then perform t test.

Implementation is divided into two classes as shown in the figure 4.1. *Parameters* class consists of the following methods.

- insert( ) - calculate central sums.
- combineState ( ) - update central sums.
- centralMoments ( ) - calculate central moments.
- standardMoments ( ) - calculate standard moments.

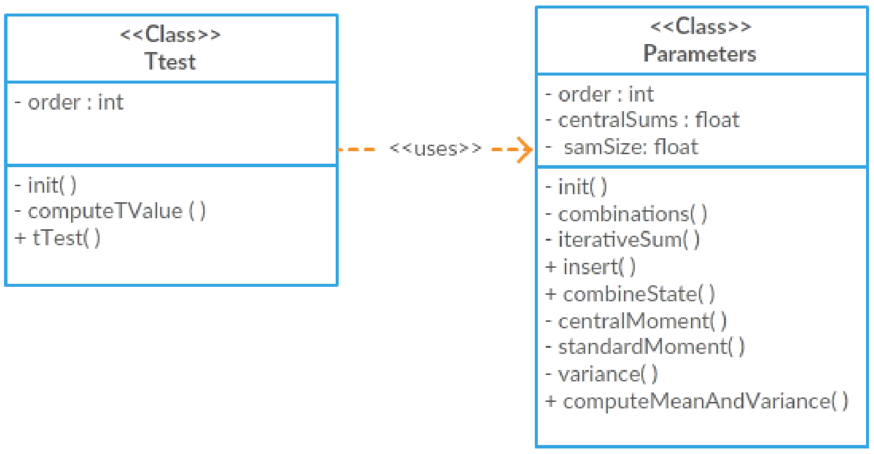


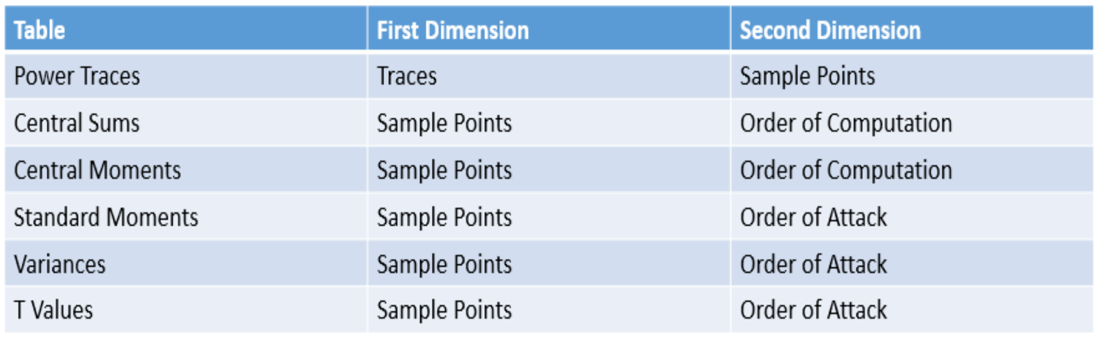
FIGURE 4.1: Class Diagram of the Algorithm

- variances ( ) - calculate variance.

*Ttest* class includes the following methods.

- computeTValue ( ) - calculate t values.
- tTest ( ) - perform t test

Dimensions of the data structures used in the algorithm are as shown in the table 4.2.



| Table | First Dimension | Second Dimension |
|---|---|---|
| Power Traces | Traces | Sample Points |
| Central Sums | Sample Points | Order of Computation |
| Central Moments | Sample Points | Order of Computation |
| Standard Moments | Sample Points | Order of Attack |
| Variances | Sample Points | Order of Attack |
| T Values | Sample Points | Order of Attack |

FIGURE 4.2: Dimensions of Tables Created in the Algorithm

## Central Sums (CS)

Formula for the calculation of central sums at different orders is available in design section 3.2. Every trace point in a particular sample updates the central sums of that sample at all orders less than or equal to the order of attack. If the order of attack is 5, then the central sums are calculated and updated at 2,3,4, and 5. Central sums at order 1 are not calculated, since central moment at order 1 is zero, instead mean is stored as central sum. Central sums are calculated from the existing central sums and the new trace point. There are no existing central sums, when the first trace is considered. For the first trace, the existing central sums are zero. Along with central sums, sample size (number of traces inserted) is also updated i.e., sample size is increased by one for every trace.

There are three components in the formula, old central sums, iterative part and constant part for particular order. The old central sums can be directly accessed, while constant part is straight forward calculation. One of the requirements was performing the test up-to order 5. If this is the case we can include hard code methods for the five orders, but it would make the algorithm inextensible. Before implementing the iterative part, we need to get the value of number of combinations for specific d (order of attack) and k (constant between 1 and d). Python package *scipy* has a module called *misc* (miscellaneous). Combinations method is available as *comb* in the misc module. To calculate the iterative part *comb* can be used.

### iterativeSum

```python
def iterativeSum(self, oldCentralSums, orderOfComp, delta, sampleSize):
    iterativeSums = 0.0
    for k in range(1, orderOfComp - 1):
        sumOfComb = self.combinations(orderOfComp, k)
        denom = sampleSize**k
        iterativeSums += (sumOfComb * oldCentralSums[orderOfComp - k - 1] \
            * (( - delta) ** k ))/denom
    return iterativeSums
```

Iterative part is iterated for d-2 times for a given d. It is the summation of the values obtained from these d-2 times. If delta ($\Delta$), sample size and the central sums for a particular order and sample are given, it is possible to compute the value of *iterativesum*. To make the code readable and easy to understand, it is decided to make it a separate method. *Iterativesum* requires order, sample size, old central sums and delta as parameters. It would return a single float value.

### insert

```python
def insert(self, traces, chunkSamNo):
    orderOfComp = 2 * self._order
    cntSums = np.zeros([traces.shape[1], orderOfComp])
    sampleSize = 0.0
    delta = np.zeros(traces.shape[1])
```

```
6     for i in xrange(0, traces.shape[0]):
7         for s in xrange(0, traces.shape[1]):
8             delta[s] = traces[i,s] - cntSums[s,0]
9             n = sampleSize + i + 1
10            if n > 1:
11                iterativeSums = np.zeros(traces.shape[1])
12                for j in range(orderOfComp, 0, -1):
13                    iterativeSums[s] = self.iterativeSum(cntSums[s], \
14                                                j, delta[s], n)
15                    temp = 1.0-((-1)**(j - 1)/float(((n)-1)**(j - 1)))
16                    constantPart = (((((n) - 1.0) * delta[s])/(n))**j) * temp
17                    cntSums[s][j - 1] = cntSums[s][j - 1] + \
18                                                iterativeSums[s] + constantPart
19            cntSums[s][0] = cntSums[s][0] + (delta[s] / (n))
20    return {"cntSums":cntSums,"chunk":(traces.shape[0],traces.shape[1],\
21                                                chunkSamNo)}
```

Input to the insert is power traces and output is central sums. The dimensions of Power traces and Central sums can be seen from the table 4.2. Order of Computation is double the order of attack(d). Unless explicitly, stated order means order of attack through out this paper. Please note that if we want to perform test up-to order d, we need mean and variances up-to order d. To calculate variances up-to d, we need central moments up to 2d, hence central sums up to 2d.

Approach:

- Iterate over traces (i: row by row of power traces).
- Iterate over sample points (s: element by element in column of power traces).
- Iterate over order of attack (j: here order is double the order of attack used in test).
- For each order calculate the central sums from old central sums, delta, order and sample size.

To access power traces, iterate through the traces and for every trace iterate through sample points as shown in the line numbers 6 and 7. The delta value is calculated using i and s indexes. To access old central sums and update new central sums indexes j and s are used. From the formula of central sums 3.2, it can be seen that mean is required to calculate central sums. Central moment at order one is zero. Mean of a sample is stored as central sum at order one. In this way mean can be accessed and updated. So central sums at order one is not central sums but central moment. While calculating central moments from central sums, remember not to alter the first column of central sums or central sums at order one, since it is already central moment.

### Combine state

```python
def combineState(self, obj, lenOfSam):
    cntSums, chunk = obj["cntSums"], obj["chunk"]
    lenOfTrOfCh, lenOfsamOfCh, sampleNo = chunk
    oldSums = self._centralSums[sampleNo:sampleNo+lenOfsamOfCh,:]
    n1 = self._samSize[sampleNo/lenOfSam]
    newSums = cntSums
    n2 = float(lenOfTrOfCh)
    if n1==0.0:
        self._centralSums[sampleNo:sampleNo+lenOfsamOfCh,:] = cntSums
        self._samSize[sampleNo/lenOfSam] = n2
    else:
        orderOfComp = 2 * self._order
        combinedCS = np.zeros((lenOfsamOfCh, orderOfComp))
        delta = np.zeros(lenOfsamOfCh)
        totalSum = np.zeros(lenOfsamOfCh)
        delta = newSums[:,0] - oldSums[:,0]
        for s in xrange( 0, oldSums.shape[0]):
            totalSum[s] = ( n1 * oldSums[s][0] ) + (n2 * newSums[s][0])
            combinedCS[s][0] = totalSum[s] / ( n1 + n2 )
            for order in range(2, orderOfComp + 1):
                iterativeSum = 0.0
                for k in range(1, order-1):
                    comb = float( self.combinations(order, k)) * \
                                        delta[s] ** k
                    sumPart1 = ((-n2 / (n2 + n1)) ** k) * oldSums[s]\
                                        [order - k - 1]
                    sumPart2 = ((n1 / (n2 + n1)) ** k) * newSums[s]\
                                        [order - k - 1]
                    iterativeSum += comb * (sumPart1 + sumPart2)
                constant = (((n2 * n1 * (delta[s])) / float(n2 + n1)) \
                                        ** order) * ((1 / float(n2 ** (order - 1))) - \
                                        (( -1 / float(n1)) ** (order - 1)))
                combinedCS[s][order - 1] = oldSums[s][order - 1] + \
                    newSums[s][order - 1]+ iterativeSum+ constant
        self._centralSums[sampleNo:sampleNo+lenOfsamOfCh,:] = combinedCS
        self._samSize[sampleNo/lenOfSam] = n1 + n2
```

Equation 3.3 provided in the design is used to update central sums and sample size. The insert method calculates CS for a given chunk of traces. Either we can directly update the CS or return them. In design, after profiling, it was decided that the calculation of CS would be done in parallel. Hence it is required that the CS are not updated but returned by the engines, as we are remembering the state only on the master node. combineState method is designed to access part(depends on chunk details) of instance _centralSums and update it. The important task in this method is to access only those part/chunk of the central sums which it wants to update. To identify specific chunk, we are storing the details of the tip of the chunk(first sample point number $j_c$, and number of traces inserted before the chunk i.e. sample size n) and lenOfTr(number of traces used for chunking) and lenOfSam(number of samples points used for chunking). First sample point number is passed as argument to the insert method. Insert method creates chunk details by adding the size of both dimensions and given first sample point number. One remaining information is the number of traces inserted before the current chunk, this information can be obtained from the instance variable sample size.

### Central Moments

The shape of central sums and central moments is same. First dimension is sample points and second dimension is double the order of attack as shown in the table 4.2, due to reason mentioned in the beginning paragraph of this section.

```python
def _centralMoment(self, lenOfSam):
    orderOfComp = 2 * self._order
    samSize = self._samSize
    cntSums = self._centralSums
    noOfSam = cntSums.shape[0]
    cntMoments = np.zeros(cntSums.shape)
    cntMoments[:,0] = cntSums[:,0]
    for i in range(0, (samSize.shape[0])-1):
        tempcntSums = cntSums[lenOfSam*i:lenOfSam*(i+1),1:]
        cntMoments[lenOfSam*i:lenOfSam*(i+1),1:] =  tempcntSums / samSize[i]
    if (lenOfSam*(len(samSize)-1))!=noOfSam:
        temp = cntSums[lenOfSam*(len(samSize)-1):,1:]
        cntMoments[lenOfSam*(len(samSize)-1):,1:] =  temp / samSize[-1]
    return cntMoments
```

The approach would be

- Iterate over sample points
- Iterative over order (except first order)
- Access each element of central sums, divide it by sample size and store it in the central moments table.

### Standard Moments

The dimensions of the standard moments are as shown in the table 4.2. Standard moments are normalized central moments by the variance. The general formula used to calculate the standard moments was given in the beginning of the design section.

```python
def _standardMoment(self, cntMoments):
    orderOfAttack = self._order
    stdMoments = np.zeros((cntMoments.shape[0], orderOfAttack))
    stdMoments[:,[0,1]] = cntMoments[:,[0,1]]
    for s in xrange(0, cntMoments.shape[0]):
        for i in range(2, orderOfAttack):
            stdMoments[s][i] = cntMoments[s][i] / \
                (math.sqrt(cntMoments[s][1]) ** (i + 1))
    return stdMoments
```

### Variances

The first dimension is sample points and second dimension is order of attack, similar to standard moments. The formula 3.6 specified in the design can be used to calculate variances at different orders except at first and second order. The variance at first order is central moment at second order, since the central moment at second is the variance of the data. The variance at second order is the difference of central moment at order four and square of central moment at order two.

```
def _variance(self, cntMoments):
    orderOfAttack = self._order
    variances = np.zeros((cntMoments.shape[0], orderOfAttack))
    variances[:,0] = cntMoments[:,1]
    variances[:,[1]] = cntMoments[:,[3]] - (cntMoments[:,[1]] ** 2)
    for s in xrange(0, len(cntMoments)):
        for i in range(2, orderOfAttack):
            denom = cntMoments[s][1] ** (i + 1)
            variances[s][i]=(cntMoments[s][2 * i + 1] - \
                (cntMoments[s][i] ** 2))/denom
    return variances
```

## 4.2 Single Node

Module h5py is utilized to store and retrieve traces. A direct view with one engine is created to perform the test. As explained in the design section, one dimensional(traces) chunking is used in the implementation. The size of power traces considered is two thousand traces and eight thousand four hundred sample points. The test is performed for every chunk (200 traces and 8400 sample points). A total of 10 tests are performed. The test took around 1.8 hours to complete.

## 4.3 Parallel

A load balanced view with 26 engines is created to perform the test. As explained in the design section, two dimensional chunking is used in the implementation. The size of power traces considered is same as that used in case of single node i.e. two thousand traces and eight thousand four hundred sample points. Chunking is done in both dimensions (25 traces and 840 sample points). The test took around 0.35 hours to complete. Though 26 engines are used, the performance gain obtained is 5 times that of the one engine. One reason for not achieving expected performance gain is the utilization of hyperthreded cores in the parallel calculation, since all the engines available on a node are being used in the creation of the load balanced view.

## 4.4 Benchmarking

This section discusses the basic approach of benchmarking followed by few scenarios to test the effect of different parameters on the execution time of the algorithm. The steps involved in the benchmarking can be generalized as follows

**Import libraries used on the local machine**
Some libraries may depend on the specific requirement of the experiment. The import of *developed algorithm*, *numpy*, *pickle*[**?** ] and *IPython parallel* module is compulsory. The algorithm is in a IPython notebook, which has .ipnb extension. Python imports files(modules) with .py extension, hence the IPython notebook cannot be directly imported. First download the IPython notebook as python(.py), then copy the file to the home directory of the user or the IPython. Instead of copying, the module can be uploaded from the upload option available on the homepage of IPython Notebook. Upload option is used in the thesis as it is convenient and there was no direct access to the directory structure.

**Create direct and/or load balanced views**
The creation of a direct view is mandatory as it is needed for remote imports on all the engines performing the task. Creation of load balanced view depends on the requirement of the experiment. Generally speaking load balanced view is not required for the experiments related execution time but it is needed for the latency related experiments.

**Import libraries required on the remote engines**
Only the import of the *developed algorithm* and *numpy* is mandatory. Though the *time* module is imported on the remote engines according to the experiment. If we create a direct synchronous object, the metadata about the execution is not directly available. We need to track it explicitly using *time module.*

**Write methods to run the required experiment and store the results**
These methods should take some parameters, including the number of times the experiment has to be repeated. The system tasks performed by the operating system may produce inappropriate results, hence it is required that the experiments are repeated to

minimize the error. The execution of experiments is a time consuming process and we may lose results once the experiments is done. Though it is possible to plot graphs but storing the results gives the possibility of analysing the data at a later point of time. One of the lessons learned from performing experiments is saving the results. Module pickle can be used to save and retrieve the results of the experiment.

**Plot the results**
*Matplotlib* can be used plot the graphs. *Matplotlib* provides many different types of backends and hence it is important to check the default backend and change it according to the requirement. Instead of preparing environment for *matplotlib* manually, IPython magic command(*%matplotlib inline*) can be used in the notebook to prepare the environment automatically. The inline command doesn't actually import the *matplotlib*, it must be explicitly done.

The performance gain obtained by parallelization depends on the communication and coordination overhead. The required input data to be processed and results of the execution are transferred over network. There is extra overhead in the task assignment, data transfer and merging the results. Performance can be maximized by efficiently using the network and remote engine resources.

### 4.4.1 Execution time

The execution time may change depending on input data (chunk size used for distribution of tasks) and the parameters of the algorithm.

**Experiment 1: Relationship between execution time and chunk size**
By size we mean number of power traces, irrespective of traces and sample points. The approach is to increase the size of the chunk for each different insertion. Repeat the measurements to minimize the random behaviour created by the system tasks. Plot the graph to see the relationship. Information obtained from experiment can be exploited to find the relation between the execution time and chunk size. This relation can be used to predict the execution time for different chunk sizes.

**Experiment 2: Relationship between execution time and chunk Shape**
By chunk shape we mean number of traces and sample points used in the experiment. Suppose a chunk of consists of 10 traces and 5 sample points and another chunk consisting of 5 traces and 10 sample points, both are having same number of power traces but different shape. Design an experiment, to insert chunks of different shapes for a given number of power traces. As we increase the number of traces, the number of sample points are decreasing, since the total trace points are fixed.

**Experiment 3: Relationship between execution time and order of attack**
The computation of central sums at order d accesses the central sums at all orders less than d. The expected relationship is not linear. Increase the number of traces inserted for every order. Consider orders up to 5 only.

**Experiment 4: Performance gain obtained by using Hyperthreading**
Machines used in the cluster have two real and two hyperthreaded cores. It is known that the execution time increases for given task when the hyperthreaded cores are used, but the benefit is that these tasks are executed in parallel. Write test case to use one, two, three and four engines. Assign same amount to these four different scenarios. Suppose x is the total units of task, one engine is assigned x units, two engines are assigned x/2 and so on.

### 4.4.2 Overhead

Network overhead includes the time taken to transfer the actual data through the network and IPython overhead. The actual data transfer may depend on the speed and topology of the network. The jobs are assigned to the engines through the scheduler. The processing overhead at the client, scheduler and engines contributes to the IPython overhead. Only the insert method is executed on the remote engines. Data transfer time, the time taken to transfer data depends on the number of bytes of memory transferred through the network. The data sent through the network is the size of chunk of traces (traces, sample points) and data received depends on the size of central sums (sample points, order of computation) and RAM memory on the client, scheduler and engines. Total time taken to transfer data per chunk can be calculated as shown below. Suppose $l_t, l_s, d, d_t$ as the number of traces in a chunk, number of sample points in a chunk, order of attack and datatype used to store the power traces.

number of bytes of memory$(b_t) = l_t l_s d_t + 2 l_s d d_t = l_s d_t (l_t + 2d)$ bytes

where $b_t$ is bytes of memory transferred on the network, 2 is used as the second dimension of central sums is order of computation i.e. double the order of attack. If b is the bandwidth of the network in Gbits per second, then the time taken to transfer the chunk can be theoretically calculated as follows

bandwidth = b Gbits per second

bandwidth = b/8 Gbytes per second

bandwidth = $(b * 10^9)/8$ bytes per second

Transfer time $t_{tc} = b_t/bandwidth = (b_t * 8)/(b * 10^9)$

If it is possible to get the total network overhead i.e. the time between the job submission on client and result retrieved from the queue on the client, excluding the actual execution time, then overhead created by IPython and other factors can be determined, since actual transfer time can be theoretically calculated.

# Chapter 5

# Discussion

## 5.1 Execution Time

### Relationship between execution time and chunk size

As the interest is in finding execution time, a direct view with single engine is used for execution. Number of sample points considered for each chunk was fixed to 840. Number of traces considered are multiples of 2. Traces used are 10, 20, 40, 80, 160, 320, 640, 1280 and 2560. The task is repeated for 10 times.

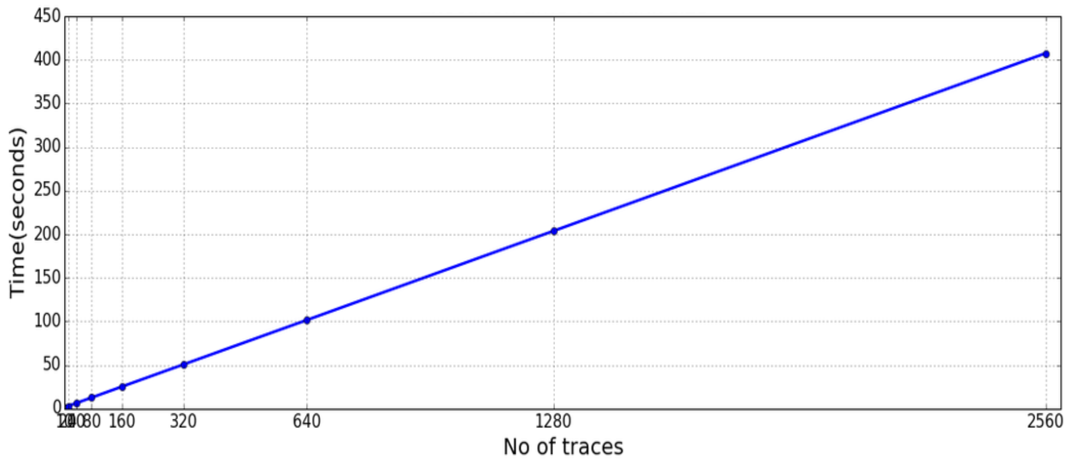


FIGURE 5.1: Execution Time for Increasing Size of Chunks

As it can be seen from the figure, the execution times of the tasks are linearly increasing with the chunk size.

### Relationship between execution time and chunk shape

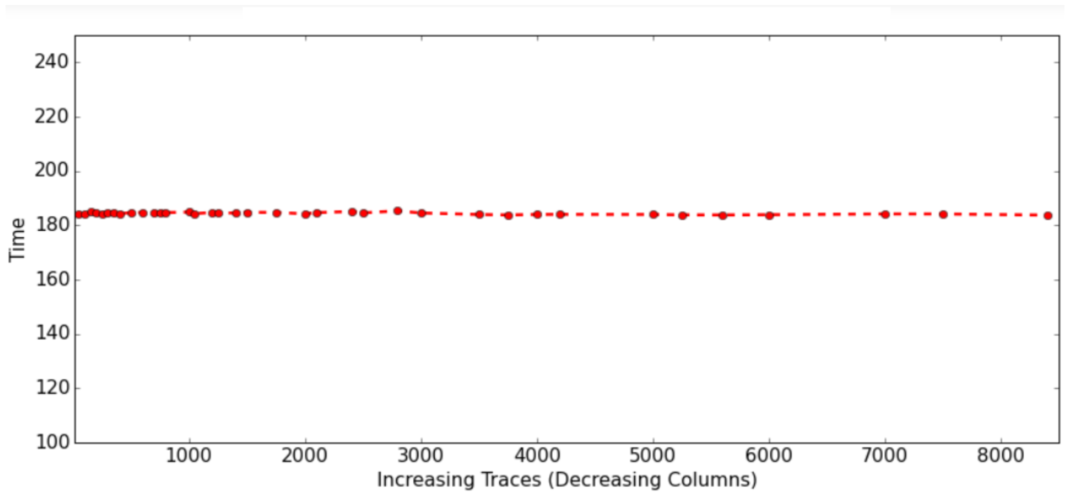


FIGURE 5.2: Execution Time of Chunks With Different Shapes

The total trace points (each cell or element in the power traces) used in the experiment were 420,000. The total trace points are fixed but the number of traces and sample points changed for each task. For example if the traces are 50, then the sample points are 8400 and if the traces are 100, sample points are 4200, making a total of 420,000 trace points. Total such 37 different task are designed for the experiment.

It can clearly seen from the graph that the shape of the chunk doesn't effect the execution time.

### Relationship between execution time and order of attack

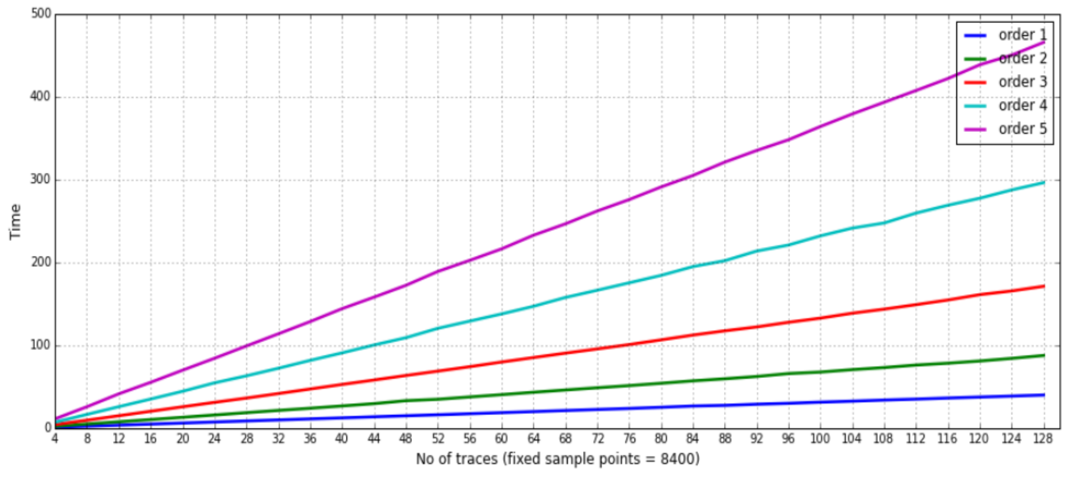


FIGURE 5.3: Execution Time of Chunks at Different Orders

A single machine with four engines is used for the execution. Number of sample points are fixed to 8400 for every task. The number of traces are taken in increasing steps of 4. The traces used are 4, 8, 12 and so on upto 128. The experiment is repeated for five different orders of attack i.e. 1,2,3,4 and 5.

Execution time of each chunk is different for different orders and also the increase in execution time with order is non-linear.

### Performance gain obtained by using Hyperthreading

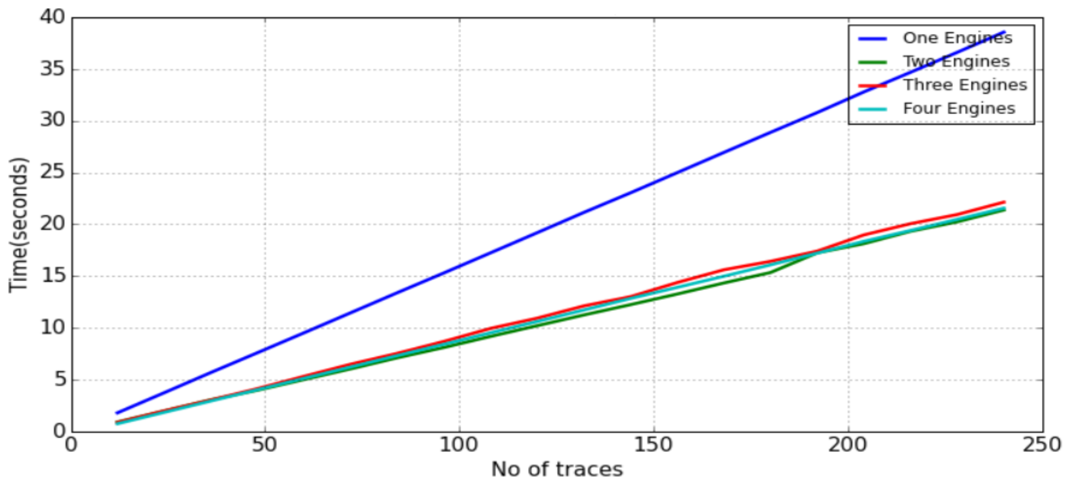


Figure 5.4: Execution time for different views

To test the execution time, four different direct views have been created. Each view uses one, two, three and four engines on the same machine. Same task is divided to all engines in the view. Suppose 128 traces are being inserted, if the view has one engine then it is assigned 128 traces and if the view has 2 engines then each engine is assigned 64 traces to insert. Number of traces considered for each chunk, are multiples of 12, as they are divisible by 1,2,3 and 4.

From the figure 5.4, it can interpreted that there is no performance gain achieved by using four engines compared to three but use of four engines has the benefit of fault tolerance. Suppose an engine on a machine fails, there are still three engines to maintain a good amount of parallel computation and performance.

# Chapter 6

# Conclusion

Amdahls states that the sequential part of the program is a limiting factor, for the speed-up that can be gained by adding additional processors. As explained in the section 3.4, the developed algorithm spends most of the time(99.46in the calculation of central sums. From this observation, one might expect the execution time of the algorithm to get reduced by the number of processors used for parallel computations. We have run the algorithm with the same input size on a single processor and on the cluster. The execution time taken on the single system was around 110 minutes. Execution time taken on cluster was around 21 minutes. Though 26 processor(cores) were used on cluster, the speed-up achieved is only five times that of the single system. This is due to the overhead of network, programming and time spent on coordinating the results. If we minimise the the time spent in coordinating the results, speed-up can be improved. Since the parallel component of the algorithm is high, the execution time can be reduced by increasing the number of processors.

The work presented in this paper has utilized state-of-the-art theoretical algorithms, and tools and technologies to perform the t test to check univariate leakages, by calculating required parameters by distributed computing. This work can be extended to check the vulnerability of DUT for multivariate leakages. The cluster used in this work, is a collection of Central Processing Units. The same work can also be implemented on the cluster of Graphical Processing Unit processors.

# Bibliography

[1] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks Revealing the Secrets of Smart Cards*, chapter Statistical Characteristics of power traces, pages 61–62, 101–114. Springer, 2007.

[2] Joshua Jaffe Paul Kocher and Benjamin Jun. Differential power analysis. In *Advances in Cryptology*, volume 1666, pages 388–397. Springer, August 1999.

[3] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual information analysis. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems  CHES 2008*, volume 5154 of *Lecture Notes in Computer Science*, pages pp 426–442. Springer Berlin Heidelberg, August 2008.

[4] Gilbert Goodwill, Benjamin Jun, and Pankaj Rohatgi Josh Jaffe. A testing methodology for sidechannel resistance validation. Technical report, Cryptography Research Inc., September 2011.

[5] Philippe Pebay. Formulas for robust, one-pass parallel computation of covariances and arbitrary-order statistical moments. Technical report, Sandia National Laboratories, M.S. 9159, P.O. Box 969, Livermore, CA 94551, U.S.A., September 2008.

[6] Tobias Schneider and Amir Moradi. Leakage assessment methodology. In Tim Gneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, volume 9293 of *Lecture Notes in Computer Science*, pages 495–513. Springer Berlin Heidelberg, September 2015.

[7] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology  CRYPTO 96*, volume 1109 of *Lecture Notes in Computer Science*, pages pp 104–113. Springer Berlin Heidelberg, August 1996.

[8] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, Taylor and Francis Group, Boca Raton, London New York, 2007.

[9] Michael Hutter, Mario Kirschbaum, Thomas Plos, Jorn-Marc Schmidt, and Stefan Mangard. Exploiting the difference of side-channel leakages. In *Constructive Side-Channel Analysis and Secure Design*. Springer, May 2012.

[10] Md Intekhab Shaukat. Revisiting remote timing attacks on openssl. Master's thesis, Frankfurt University of Applied Sciences, April 2015.

[11] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks Revealing the Secrets of Smart Cards*, chapter Simple Power Analysis, pages pp 101–118. Springer US, 2007.

[12] Tom Caddy. *Encyclopedia of Cryptography and Security*, chapter Differential Power Analysis, pages pp 336–338. Springer US, 2011.

[13] Jean-Jacques Quisquater and David Samyde. *Smart Card Programming and Security*, volume 2140 of *Lecture Notes in Computer Science*, chapter ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards, pages pp 200–210. Springer Berlin Heidelberg, 2001.

[14] Deborah Rumsey. *Intermediate Statistics For Dummies*, chapter Going One-Way with Analysis of Variance, pages 161–163. Wiley Publications, Inc., 2007.

[15] Gideon Keren and Charles Lewis, editors. *A Handbook for Data Analysis in the Behaviorial Sciences*, chapter Relative Power, pages 507–510. Pschycology Press, 2009.

[16] Begl Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Higher-order threshold implementations. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology ASIACRYPT 2014*, volume 8874 of *Lecture Notes in Computer Science*, pages pp 326–343. Springer Berlin Heidelberg, 2014.

[17] Tobias Schneider, Amir Moradi, and Tim Gneysu. Arithmetic addition over boolean masking. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *Applied Cryptography and Network Security*, 2015.

[18] Pascal Sasdrich, Oliver Mischke, Amir Moradi, and Tim Gneysu. Side-channel protection by randomizing look-up tables on reconfigurable hardware. In Stefan Mangard and Axel Y. Poschmann, editors, *Constructive Side-Channel Analysis and Secure Design*, volume 9064 of *LNCS*, pages pp 95–107. Springer International Publishing, July 2015.

[19] Robert Szerwinski. Steam side-channel analysis cluster setup. Technical Report.

[20] Tranquil PC. Ubuntu orange box. http://cluster.engineering/ubuntu-orange-box/. Last Accessed: 28.03.16.

[21] D. M. Beazley. Scientific computing with python. http://www.adass.org/adass/proceedings/adass99/O3-02/, 2000. Last Accessed: 29.03.16.

[22] Travis Oliphant and NumPy Developers. Numpy. http://www.numpy.org. Last Accessed: 31.03.16.

[23] Eric Jones, Travis Oliphant, Pearu Peterson, et al. Scipy. https://www.scipy.org. Last Accessed Date: 31.03.2016.

[24] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[25] Python Software Foundations. doctest test interactive python examples. https://docs.python.org/2/library/doctest.html, . Last Accessed:29.03.16.

[26] Python Software Foundations. The python profilers. https://docs.python.org/2/library/profile.html, . Last Accessed Date: 28.03.2016.

[27] Karl Fogel. lsprof. https://launchpad.net/lsprof. Last Accessed Date: 31.03.16.

[28] IPython Development Team. Architecture overview. https://ipython.org/ipython-doc/2/parallel/parallel_intro.html, . Last Accessed Date: 31.03.16.

[29] The IPython Development Team. Using ipython for parallel computing. https://ipython.org/ipython-doc/3/parallel/index.html, . Last Accessed:30.03.2016.

[30] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks Revealing the Secrets of Smart Cards*, chapter Statistical Characteristics of Power Traces, page 61. Springer US, 2007. ISBN:978-0-387-30857-9.