3-10-2021

Hardware Trojan Detection Using Machine Learning

Daisy .
University of Windsor

Hardware Trojan Detection Using Machine Learning

By

Daisy

A Thesis
Submitted to the Faculty of Graduate Studies
through the Department of Electrical and Computer Engineering
in Partial Fulfillment of the Requirements for
the Degree of Master of Applied Science
at the University of Windsor

Windsor, Ontario, Canada

2021

Hardware Trojan Detection Using Machine Learning

By

Daisy

APPROVED BY:

A. Edrisy
Department of Mechanical, Automotive & Materials Engineering

H. Wu
Department of Electrical & Computer Engineering

M. Mirhassani, Advisor
Department of Electrical & Computer Engineering

January 22nd, 2021

Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

The cyber-physical system's security depends on the software and underlying hardware. In today's times, securing hardware is difficult because of the globalization of the Integrated circuit's manufacturing process. The main attack is to insert a "backdoor" that maliciously alters the original circuit's behaviour. Such a malicious insertion is called a hardware trojan.

In this thesis, Random Forest Model was proposed for hardware trojan detection and this research focuses on improving the detection accuracy of Random Forest model.. The detection technique used the random forest machine learning model, which was trained by using the power traces of the circuit behaviour. The data required for training was obtained from an extensive database by simulating the circuit behaviours with various input vectors. The machine learning model was then compared with the state-of-art models in terms of accuracy in detecting malicious hardware.

Our results show that the Random Forest classifier achieves an accuracy of 99.80 percent with a false positive rate (FPR)of 0.009 and a false negative rate (FNR) of 0.038 when the model is created to detect hardware trojans. Furthermore, our research shows that a trained model takes less training time and can be applied to large and complex datasets.

Acknowledgments

I wish to express my sincere gratitude to Dr.Mitra Mirhassani, for her patience, motivation and immense knowledge throughout my graduate study.

I would like to thank my family members, Mom, Dad, Grandfather, Yash and Kiran for their constant support and continuous encouragement during the time of completing my further study.

I would like to thank my committee members, Dr. Afsaneh Edrisy and Dr. Huapeng Wu.

I would also like to thank my colleagues at UWindsor's Faculty of Electrical and Computer Engineering, especially Andria Ballo, for their help and support.

Table of Contents

List of Figures

List of Abbreviations

HT	Hardware Trojan
IC	Integrated Circuit
CAD	Computer Aided Design
NSA	National Security Agency
SVM	Support Vector Machine
MLP	Multi-layer Perceptron
ASIC	Application Specific Integrated Circuit
SPICE	Simulation Program with Integrated Circuit Emphasis
PCB	Printed Circuit Board
GPS	Global Positioning System
FANCI	Functional Analysis for Nearly Unused Circuit Identification
RTL	Register Transfer Level
MERO	Multiple excitation of Rare Occurrence
EDA	Electronic Design Automation

Chapter 1

Introduction

Hardware security has become a serious concern with advancements in the IoT and autonomous and smart vehicles. The electronics parts and devices are now used in many household items and are affecting all aspects of life.

Securing the hardware and ensuring the integrity of electronic parts is a difficult task today due to the globalization of integrated circuits' manufacturing flow. Due to the increased demand for smart devices, hardware vendors outsource manufacturing portions of the design to un-registered and sometimes unknown third-party vendors [1].

By outsourcing the integrated circuit manufacturing, it becomes vulnerable to Hardware Trojans [2]. Hardware Trojans are small, simple, and stealthy; therefore, their detection is difficult. It has become one of the most critical issues in part production and manufacturing.

The Trojan hardware can disrupt the system operation, leak confidential information, and decrease its reliability. Detection of Hardware Trojan using Machine Learning (ML) is the major focus of this thesis.

1.1 Motivation

It is important to make sure that any device connected to IoT network is secure since it might greatly impact human lives. The security of a larger system depends on the integrity of internal software and underlying hardware.

In general, the system's security depends mainly on three components: the information processed, software, and underlying hardware. It should be noted that while a software patch or update might resolve issues with the software and application level of a system, hardware modifications cannot be performed this way. The modification which maliciously alters the behaviour of the original system or Hardware Trojan is done secretively to never be noticed [3] A remote system update cannot fix a fault in the system's hardware. Instead, the hardware has to be replaced or accepted as a source of the fault in the system [4]. The hardware trojan is a malicious modification of hardware, and this can take place at various stages of hardware's production or repair life cycle. The faulty hardware part in a system may require a system disassembly and part replacement [3]. It can have multiple effects, such as performance degradation, Denial of Service, leak of classified information [2].

The first real-world detection of hardware trojan happened in a military-grade field programmable array (FPGA) in 2012 that allowed an adversary to extract configuration data from the chip or even permanently damage the device.

The other example of Hardware Trojan insertion is the National Security Agency (NSA) intercepting shipments of network devices before their arrival at the destination. They inserted "beacon implants" into the network devices that allowed NSA to exploit the devices and survey the network [5]. Then one of the devices was used in the Syrian Telecommunication Establishment network. After that, NSA was able to track the call detail records having the billing information that exploited the users' identity with their geographic locations.

In this thesis, an advanced hardware trojan detection technique is proposed and implemented. This technique is based on a machine learning algorithm called Random Forest Classifier [6] to detect hardware trojan. The proposed machine learning model was more precisely trained based on the data collected from implemented circuits using Hspice circuit simulations.

Securing hardware is exceptionally difficult due to the complexity of the design and manufacturing variability, resulting in the integrated circuit's different physical characteristics, even among the integrated circuit coming from the same design.

The manufacturing flow of an IC follows these steps[7, 8]:

1. System Specification: System specifications are high-level requirements indicating the expected capabilities of the design.
2. Design: The design stage is to produce the physical layout, and it is composed of geometric representations of the circuit, which will guide the manufacturing process.
3. Manufacturing: In the manufacturing stage, the design is converted into wafers of semiconducting material. The manufacturing of IC's takes place at different locations.
4. Assembly and Market: The manufacturing process ends with packaging; the IC's are distributed to several different sectors, including transportation, military, technology, and financial organizations.

Any participant in the manufacturing process can be an adversary who potentially can insert a Hardware Trojan [9]. For example, a Hardware Trojan can be inserted into the design by a computer-aided design (CAD) designer at the design stage. CAD tools are used to perform synthesis through software, and hence Hardware Trojan can be inserted in the system. The Hardware Trojan insertion could happen by maliciously altered logic in the CAD tool or scripts running the tool [10].

However, in this thesis, we do not include the modifications at this stage of the design and assume that the CAD design is secure and trustworthy.

The common characteristics of a Hardware Trojan are reviewed here briefly [10].

1. Insertion phase: The manufacturing process consists of different phases, mainly the specification stage, design stage, fabrication stage, testing stage and packaging stage. The characteristic defining a hardware trojan is the phase at which it was inserted [10].
2. Trigger: Hardware Trojans can be categorized based on the type of trigger that activates them. There are three different types of triggers. The first type is the "always-on" where the Hardware Trojan is awake from the very beginning of insertion. The second is the "internal" where the Hardware Trojan is triggered by an internal condition or signal within the design [10]. The third one is when the Hardware Trojan is triggered by an external user.
3. Payload: The different impact hardware trojan can have on the design are the change of the circuit's functionality, degradation of the designs' performance, leak of sensitive information, and denial of service [10].
4. Insertion Location: Different HTs can be characterized by the location at which they were inserted within an integrated circuit. The possible areas are the processor, the memory, the input or output pins, the power supply lines within the IC, or the clock grid. Moreover, an HT can be distributed over several locations across the places mentioned above or only in one area [10].
5. Physical Characteristic: HTs can be differentiated based on their physical features such as their distribution (focused or dispersed) or size, measured in the number of gates or percentage of area occupied by HT compared to the entire circuit size [11].

1.1.1 Detection Methods

Detection of Hardware Trojan can be challenging because of the variety in its sizes and locations of inset. The detection approaches can be classified into these categories [3]: [7] the H

1. Run-Time Monitoring: It focuses on detecting Hardware Trojan after the deployment, and it can be used for the whole lifetime of the integrated circuit [11].
2. Design for Security: The first choice is to prevent the insertion of the Hardware Trojan by having restricted and precise design steps and processes for security in order to facilitate detection [11].
3. Activation Monitoring: After the fabrication process, the detection of the Hardware Trojan can be divided into the destructive and non-destructive approach. The destructive approach is to reverse engineering each layer of the integrated circuit and validate it. The non-destructive approach consists of three categories: the first one detects hardware trojan at the IP level. The second one depends on targeted test patterns applied to IC to activate the hidden hardware trojan and generate its effect to be detected [3]. The third category measures a side-channel such as a path delay or power supply current to expose an inserted hardware trojan.

Detection methods that are based on the side-channel analysis take place during the post-silicon testing stage. The side-channel information is the physical parameters of the design, such as delay or power consumption, and also acts as the unique signature of the hardware.

Most of the detection methods, however, rely on a Golden Circuit. A Golden Circuit is considered a Trojan free design Golden Circuit [7]. The side-channel parameters are compared with Golden Circuit's side channel parameters [12]. This helps in identifying whether the circuit is secured or compromised by the hardware trojan [3]. [13]

A modification to the original design in the manufacturing process will most probably present itself in side-channel information of the system when compared to trojan free circuit [3].

1.2 Objective

In this thesis, a machine learning-based hardware trojan detection approach is proposed. In this work, circuit simulations are used instead of the Golden Circuit. The simulations are the results of the statistical runs on the circuit parameters such as temperature, transistor corners, effects of mismatch and manufacturing inconsistencies. The simulations are based on the Hspice transistor modelling and are from the models developed by the TSMC manufacturing models.

Methods that are based on the Golden IC assume that they have a fabricated and trusted IC. This assumption is not valid in most cases and cannot be guaranteed at all. This process assumes that trojans are inserted into random ICs. It is more viable for an attacker to insert a stealthy trojan into every fabricated integrated circuit that passes manufacturing process and trust validations, avoiding the need for additional expensive masks. Therefore, all samples are infected, and it is not possible to have a trusted IC for detection uses. This raises the challenge of detecting the Trojans in integrated circuits without relying on golden ICs.

Machine learning helps to build models to help systems learn from data. Machine learning approaches can be divided into three categories [14] of Supervised learning [15], Unsupervised learning [15], and Reinforcement learning [13].

Supervised learning [14] is the one in which a model is trained with the input data called the training dataset. The model is able to make predictions, however, it needs to be corrected if these predictions are wrong [14].

Unsupervised learning [15] is the one in which a model makes predictions on its own, and its input data is not labelled. For example, K-means and clustering are the examples of unsupervised learning [14]. Since this method is not supervised, it might result in miscategorizing the infected IC and is generally avoided in this application.

In the reinforcement learning [13] the model learns to act through trial and error. This learning is used in video games [7], and adaptive learning process and is not suitable for classification problems such as separating the Infected ICs from the Trojan free Ics.

In this work, the Random Forest Algorithm [6] is used to train our model to detect hardware trojan. Random Forest is one of the supervised learning algorithms.

The Random Forest consists of several decision tree branches, which acts as an ensemble method [14]. Each individual tree gives out a prediction and the tree with the most votes becomes model's prediction.

Because of the low correlation function developed between the branches, this model outperforms other supervised models such as backpropagation. Decision trees are known for variance and even a small change in the data can result in a change in the final decision.

Moreover, it can overcome the problem of overfitting because of less correlation with the other decision trees. It can deal well with high dimensional data.

Random forest ensures that each tree is not too correlated with behavior of other trees. It uses these two methods:

– Bootstrap Aggregation (Bagging): Small changes to the training set can lead to different tree structures and decision trees are sensitive to the data. It allows each tree to randomly sample from the dataset with replacement. This process is known as Bagging [7].

– Feature Randomness: With the help of this method, there is more variation among the trees in the model and results in lower correlation among trees and more diversification

This supervised learning method is applied to detecting the Hardware Trojan on the power traces of a 256-bit AES. Additional improvements on the data labelling and training of the Random Forest has been carried out, which will be presented in the later chapters.

1.3 Organization of Thesis

The thesis is organized as follows.

- In chapter 2 different trojan detection approaches have been introduced and drawbacks of each detection techniques such as reverse engineering, side channel analysis, logic testing and SVM based detection approaches is provided.
- Chapter 3 introduced the Random Forest Classifier algorithm. It explains how this algorithm works and its advantages and disadvantages. It contains the different methods applied to the random forest model to improve its accuracy and reduce training time. We briefly introduce features and important hyperparameters of random forest which includes the predictive power and model speed. Finally, we discuss working of random forest algorithm and methods for missing values replacement for training and test set.

- Chapter 4 introduces the fundamental technology used to create random forest model, including data preparation, data cleaning which consists of fill-out missing values, fixing errors and removing rows with missing values, data normalization. Then we use train/test split and confusion matrix and cross validation method to train the model. At last of this chapter, we apply our proposed solution to make a comparison with published results achieved using machine learning techniques and achieve a considerable result.

- Chapter 5 is summarized the contribution of this thesis and suggestions for future works.

Chapter 2

Litrature Review and Background Information

In this chapter, various types of hardware trojans and different types of detection techniques will be reviewed. Hardware Trojans can be have been classified based on five major attributes [16] as following:

1. Insertion Phase
2. Abstraction Level
3. Activation Mechanism
4. Functionality
5. Location of Insertion

2.1 Insertion Phase

An Integrated circuit is manufactured in various steps from specification to fabrication. The insertion phase can be divided further into several manufacturing stages, from when the circuit specification is proposed to when the device and circuit are fabricated. Detecting a Trojan inserted at the design and specification stage is almost impossible and requires that trustworthy employees and designers [16]. This type of Trojan is not within the scope of this thesis. Instead, the Trojans that are inserted in the foundry and manufacturing stage will be considered.

A brief introduction and review of the main insertion phases are presented here. It should be noted that an actual manufacturing process is a complex function, and some of the stages can be divided into further categories.

2.1.1 Specification Phase

Hardware Trojans can be inserted by altering the specification of the integrated circuit. In many detection approaches, integrated circuits are being checked by comparing with the Golden ICs characteristics matching the specifications [1].

When the specification itself is altered, the detection mechanism will never work [10]. An adversary may deliberately identify weak system requirements, and as a potential consequence, the design's reliability could be compromised, rendering the system vulnerable to sensitive information leaks.

2.1.2 Design Phase

The design is mapped onto the technology, and transistor sizes and specifications will be determined during this stage. This phase is another source of Hardware Trojan threat as pre-silicon verification might not be available for such imported chips.

Even though the entire design is done in-house, the mere use of untrusted resources can result in a potential security breach. It can be influenced in a harmful way by libraries, third party IPs and regular cells [5].

For example, untrusted instruments may add additional circuitry to the system to insert backdoors in the system. If every design process step is outsourced, a Trojan could be directly added to the genuine circuit's hardware description data.

2.1.3 Fabrication Phase

In this phase, wafers are manufactured using the mask derived from the design phase. A minimal change in the mask can act as a threat to the IC. The design is vulnerable to be modified by the addition or removal of parts of it. The circuit's susceptibility to fault-based assault may also be significantly increased by the Trojans inserted at this stage [10]. Trojans inserted in the Fabrication phase are detectable by various tests and verifications, and comparisons with the Golden IC characteristics.

2.1.4 Assembly Phase

A printed circuit board is assembled in this step by putting various components together. For example, a Trojan can be implemented by introducing an I/O pin with high capacitance, leading to information leakage.

Therefore, a malicious and untrusted assembly can produce flaws in the system. To reduce the chance of Trojan insertion, the IC is encapsulated, and packaged [16]. This type of Trojans can be detected by proper test benches and matching the expected system behaviour with measurement values.

2.1.5 Test Phase

The test phase has always been used as a stage to ensure the trustworthiness of an IC. There is no scope for Trojan insertion in this phase, but unfaithful testing can lead to hardware trojan inserted ICs undetected [16].

However, any change in the circuit layout can alter the test set-up and its associated programs or reports to cover potential Trojan insertion. Moreover, since this is the last step in the flow of IC design and just before the manufacturing step, it is the last chance for original designers to detect Trojans before the stage of deployment [11].

2.2 Abstraction Level

Hardware Trojans can be categorized by the hardware definition level when they are inserted [5]. When an IC is designed and manufactured, it goes through different stages of hardware definition. The way transistors and gates are defined at the system, and abstract level is different when the design is defined at the transistor level. These levels all take place at the design stage and can be considered a sub-category of the design phase, as described above.

2.2.1 System Level

This is the highest level of abstract level when defining the circuit operation and design. At this level, the system is broken into components, modules, communication protocols and data. It is interesting to note that a Trojan inserted at this stage can be improved in the specifications of functions, protocols, and interfaces. Any obscure requirements may be introduced by an adversary involved at the machine level to give him control of secret data flowing through the produced unit [7].

For example, an opponent might modify the specifications of true random number generators (TRNG) at the specification stage to make it function predictably because only the HT owner is aware of [1]. This can reduce the reliability of secure systems based on these architectures and give away sensitive information.

2.2.2 Register Transfer level

At this level, an integrated circuit is defined in terms of its required registers and memory blocks, input/output signals, and combinational logic. If an attacker has access to design and changes it, it can cause serious damage because of greater hardware control. An HT can also be a simple modification of real RT-level codes or codes. An adversary may change the circuit's functions to provoke significant delays, or power consumption [11]—consequences, such if the unit is used for a cryptographic block; it can cause failures in the authentication applications. Attackers at the stage of design or possible HT insertion sources are untrusted code suppliers.

2.2.3 Gate Level

In most of the research work done for Trojan detection, the Trojan circuit is inserted by altering the gate-level netlist. In the initial netlist, the addition or removal of one or more gates is called an HT gate-level. It is also possible to use standard delay format files that contain device timing data [17]. Modifying, changing timing and power constraints can be used to mask HT insertion.

2.2.4 Transistor Level

Trojans inserted at the transistor level have the highest impact on the circuit's performance since these can affect and control power consumption, delay, channel length, parasitic capacitors, wire width and lengths, to name a few.

The addition of a small number of transistors will not significantly increase the circuit parameters such as power and delay. Furthermore, to increase critical path delays, transistors can be inserted, causing the circuit to malfunction. At this stage, the integrated circuit is at the manufacturing stage; hence the adversary is in the form of untrusted resources, libraries, and templates [11].

This type of Trojan detection is the most challenging type to detect due to small and minor changes that can be masked by the manufacturing effects. In this thesis, this type of Trojans has been considered.

2.3 Activation Mechanism

There are Hardware Trojans that are activated only under certain conditions. They can be classified into Time-based and physical conditions-based activation.

Some Trojans are designed to be always active and can affect the system at any time. If a Trojan is always triggered, some system properties may be upset by its impact on the circuit [2].

Suppose an HT stays dormant until it is active; in this case, its detection is challenging since its disruptions in the behaviour of the circuit become less visible, significantly obstructing its identification [18].

Trojans are likely to feature activation mechanisms for this reason. They are used only after verification and validation phases and are triggered under certain conditions. There are three main types of activation-based functions; always on, internal activation and external activation.

1. Always on: The conduct of the target circuit is always impacted by the hardware trojan [17]. Therefore, the Trojan is composed only of the payload, which is the added Trojan's destructive effect.

2. Internal activation: When a particular internal condition occurs in the circuit, a Trojan is triggered. An internal counter, for example, will activate the HT if the clock exceeds a certain value. Additionally, this type of Trojan can be triggered by internal signal patterns or unusual conditions within the integrated circuit [17].

3. External activation: This type of Trojans is triggered by an intruder who is conscious of HT's existence in the circuit and is applied from outside. For example, a certain input pattern can be used to activate the Trojan [17]. Sophisticated trigger processes rely on very rare trigger mechanisms. This type of Trojans can be detected by the side-channel probing o the circuit. Since these get activated only for a rare input pattern, it is almost impossible for users not to activate the Trojan.

2.4 Effects and Payloads

The attacker inserts Trojan with a goal to cause certain effects, which are as follows [19]:

1. Change of Functionality: Its effect can lead to a change in functionality that was not the purpose of the integrated circuit's specifications. For example, a Trojan inserted in the GPS can change the position data generated by the GPS.

2. Reduce Reliability: The Trojans is designed with the aim to downgrade the performance of the circuit. It causes the device to perform poorly after certain operations and increase system error and repeated system failures.
3. Leakage of Information: A Trojan can be inserted to only leak the confidential information and does not alter the system performance and functionality. This type of Trojans is mostly inserted on the PCB, leading to leakage of sensitive information [19].
4. Denial of Service: A trojan can restrict other users' access to the system by constantly requesting sta from the servers. This type of attack is simple and does not require complex circuits and operations [10].

2.5 Placement

Trojans are also categorized based on the computational units' placement within a system [16].

1. Processor: Trojan can modify, add, or remove processor instructions, causing it to run functions that are suspect and cause malfunctions.
2. Memory: Memory element controlled by attackers, provides them with access to memory components, releasing confidential information and secret keys.
3. I/O: The HT-controlled pins may cause the circuit to avoid certain specific conditions, input erroneous signals to the system, and track communications [16].
4. Power supply: Trojans in the circuit's power grid can control the device voltage or current, thus increasing leakages or causing failures.

2.6 Hardware Trojan Detection Approaches

Research work done on trojan detection techniques is categorized into several categories. These detection techniques can detect Trojans at pre-silicon or post-silicon stage.

With the logic testing, dedicated test patterns are generated and applied to det ect hardware trojans. Another approach is to rely on the side-channel analysis, where detection is done by comparing parameters such as total power, delay, or temperature of the circuit [2].

2.6.1 Logic Testing

A logic testing approach requires the Hardware Trojan trigger nodes to be activated with appropriate test patterns. This method requires exhaustive testing and does not have a high chance of Trojan detection. The Trojan can be hidden at a node that is not activated by any input pattern.

Using multiple excitations of rare occurrence (MERO) method, a set of test patterns is generated to reduce time and cost of detection and maximize the trojan detection coverage [2].

Another test pattern generation approach consists of guided test patterns which focus on small but vulnerable areas of chip where patterns show unusual activity [2]. A tool called FANCI detects vulnerable nets through Boolean Functional Analysis to some degree of success.

2.6.2 Side-Channel Analysis

Side-Channel Analysis is based on developing a testing scheme to capture the overall characteristics of the integrated circuit. These characteristics are then

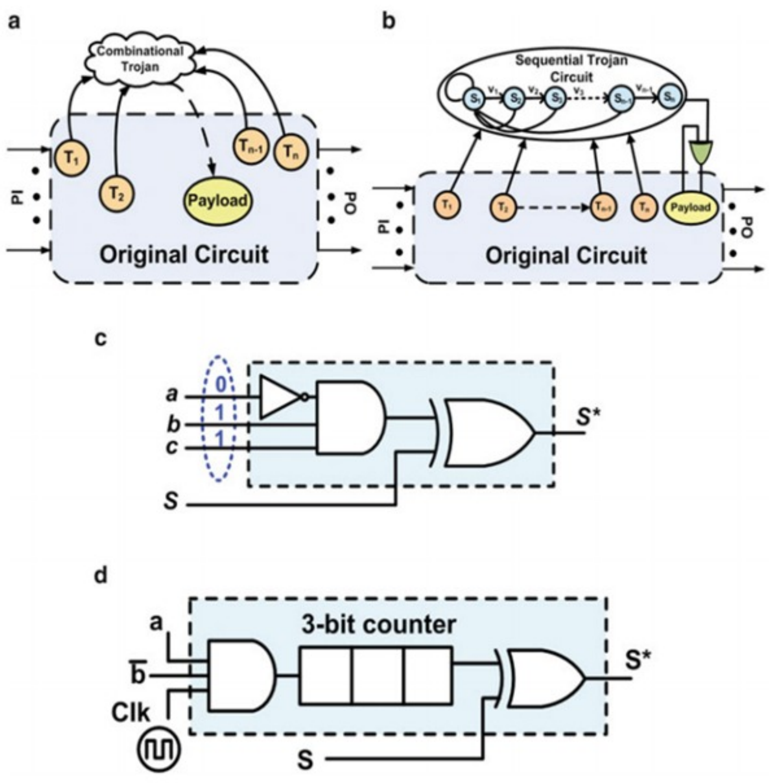

FIGURE 2.1: Generic model for combinational and sequential Trojan circuit [33]

compared with a baseline function and are used as a pass/fail criteria for the manufactured circuit [12].

An effective Trojan is only triggered under rare conditions. Thus, post-fabrication functional and structural testing conducted using a limited number of test patterns is usually not reliable to define the trustworthiness of a fabricated IC received from an external foundry.

Exhaustive testing, covering all possible input patterns, is also not a practical solution for most chips because of extensive time requirements [12].

Therefore, it is expected that Trojans' full activation followed by any observable change at the output should be an infrequent case under usual testing schemes.

Nonetheless, partial activation of Trojan is possible. During testing or normal operation, the Trojan circuit may receive input patterns that activate some of part of it for a very brief period.

The occurrence of signal transition at the Trojan gates' input is very likely to cause power or delay variation. Many of the Trojan detection techniques are based on observing the possible change in the IC's side channel behaviour due to Trojan's partial activation. This procedure provides a useful workaround, eliminating the need for exhaustive testing.

One of the critical issues regarding the side-channel analysis method is the process, environmental variation, and measurement noise [8]. These variations make it difficult to isolate the deviation of side-channel parameters caused by the Trojan, which is usually smaller than the process and environmental variations. Nevertheless, side-channel monitoring in order to detect the Trojan is one of the most effective methods of detecting Trojans.

2.6.3 Reverse Engineering

Reverse engineering can be performed at the chip, board, or system level. It includes verification of a design for quality control, fault analysis, trojan detection, and trust evaluation [13].

In terms of Trojan detection, the reverse Engineering of the circuit is the process of examining and analyzing the chip's internal configuration and layer by layer composition to detect added, and unwanted transistors or gates during the fabrication process [16].

Reverse engineering can provide convenient tools for the identification of malicious circuits. It is used to identify possible insertion points in circuits. However, this method is a destructive method to detect Trojans, and requires excessive manual effort and is very time-consuming.

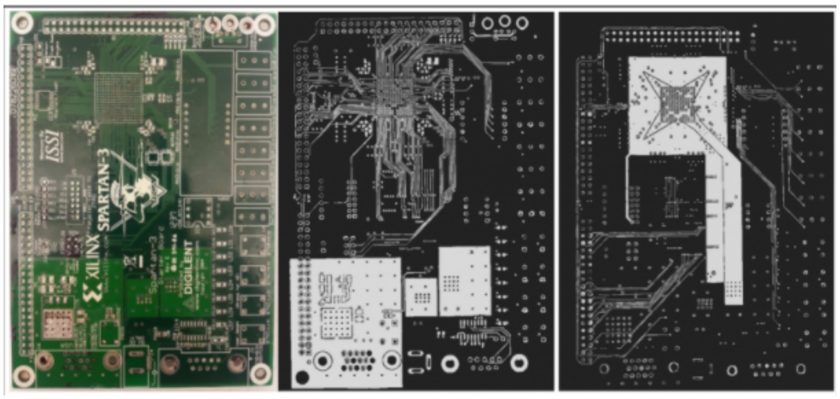

Figure 2.2: DIGITAL CIRCUIT [5]

In today's times, digital circuit design is realized at the RTL level that models signal flow among registers. Logical synthesis devices convert register transfer level descriptions to gate-level netlists. Then, place and route algorithms process the netlist and check where gates are placed and how the interconnections are connected. During this transfer, valuable information such as module binary information and hierarchy information is lost [20].

2.6.4 Trojan Scanner

Trojan Scanner is a newly introduced concept for the untrusted threat model. An advanced computer algorithm is combined with supervised learning models in order to differentiate features of the golden layout, and SEM images from the integrated circuit under authentication [7]. The outcomes of each process are then compared to detect any changes, which raises the flag for a potential Hardware Trojan. These can check changes due to fabrication, defects.

2.6.5 Design for Security (DFS) Approach

Design for Security Approach is to control the threat of trojans by introducing changes to the design. It is similar to the built-in self-test (BIST) technique used to detect faults in circuits [21].

Prevention is done using these two approaches:

1. Obfuscation based Approaches
2. Layout filler Approaches

Obfuscation-based approaches involve designing the system to make it difficult for the attacker to figure out the integrated circuit structure. Hence, it becomes hard for the adversary to insert a trojan and keep the circuit's behaviour intact.

2.6.6 Extreme Learning Machine (ELM):

Extreme Learning Machines are feedforward neural networks. [22] that are used for classification, regression and clustering. The ELM is a single layer neural network with randomly generated neurons and randomly chooses input weights and hidden units [23].

This approach analytically determines the output weights. It has the advantage of fast learning speed and good generalization performance, along with less overfitting problems.

ELM output is calculated by:

$$f_L(x) = \sum_{i=0}^{L} \beta_i g_i(x) = \sum_{i=0}^{L} \beta_i g(w_i * x_j + b_i), j = 1, ..., N \tag{2.1}$$

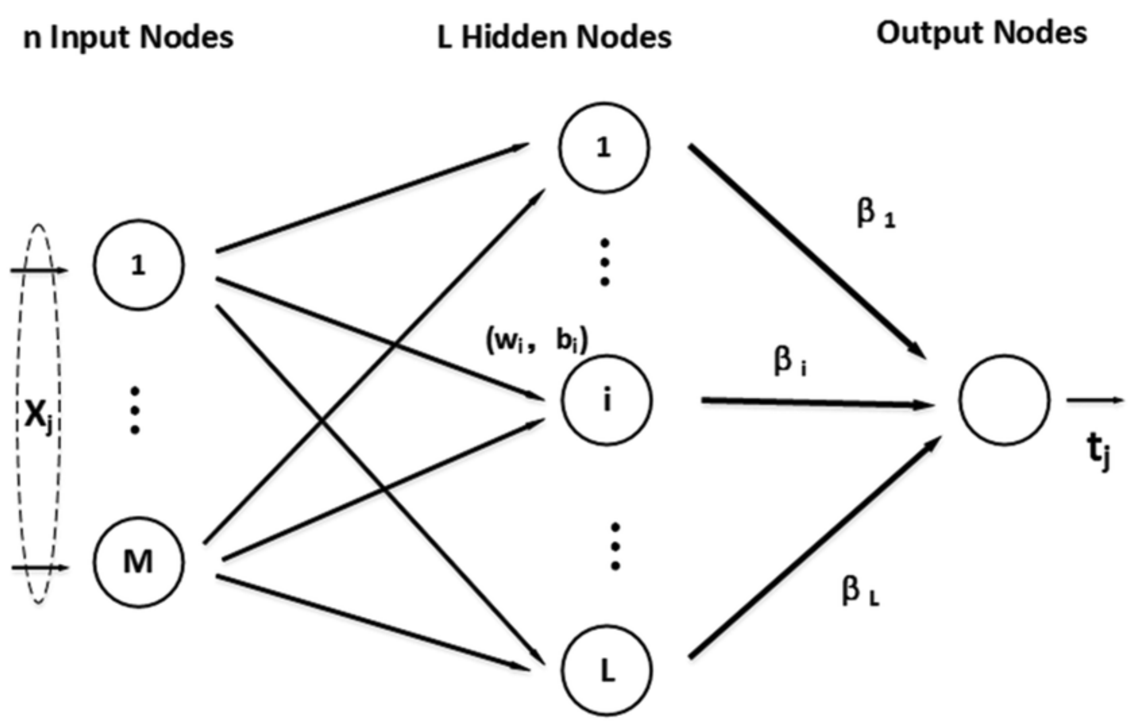

FIGURE 2.3: ELM [55]

where:

L is a number of hidden units

N is a number of training samples

Bi is weight vector between ith hidden layer and output

w is a weight vector between input and hidden layer

g is an activation function

b is a vias vector

x in an input vector

ELMs are not as precise as the conventional neural networks, but they can be used to deal with issues that involve real-time network retraining.

2.6.7 Random Forest Classifier

Random Forest Classifier is an ensemble algorithm and is a set of prediction trees where each tree depends on independently sampled random vectors with a distribution close to that of any other tree in the random forest [15].

Originally intended for machine learning, due to its high accuracy, the classifier has gained popularity in the remote-sensing community, where it is used in remote-sensed image classification [6].

Random Forest ensures that each tree's behaviour is not too correlated with any other trees' behaviour by using the Bootstrap Aggregation concept. The Bootstrap Aggregation (Bagging) decisions trees are susceptible to the information from which they are trained. Minor adjustments to the training set can lead to dramatically different structures of the tree.

Random forest takes advantage of this by enabling each tree to sample randomly with replacement from the dataset, resulting in numerous trees [24]. This mechanism is known as bagging [16]. This method is used in this thesis for the detection of the Trojans.

2.6.8 Why Random Forest

1. It is one of the most precise algorithms for learning. It produces a highly specific classifier for many data sets.
2. It runs very efficiently on large datasets.
3. It can handle hundreds of input variables without variables deletion.
4. It gives an estimation of which variables are important in classification.
5. It generates an unbiased estimate of generalization error as the model building progresses.

6. It has an efficient method of estimating missing data and preserves precision in the absence of a significant proportion of the data [24].

2.7 Summary

In this chapter, various characteristics of the Hardware Trojan is explained. Moreover, the reason for choosing the random forest approach is provided. Random forest works very well with high dimensional data, less overfitting due to using the bagging method. Forest trees are fully grown, unpruned, and the feature space is divided into smaller regions.

Chapter 3

Random Forest Classifier Algorithm

In this chapter, the main algorithm for detecting Trojan, called Random Forest Classification, will be reviewed.

In machine learning, a few classification algorithms learn from input data and use it to make new observations. The data can be bi-class or multi-class.

Examples of a bi-class data set are identifying an object to see if it is a pen or a scale, or for example, deciding when an email is a spam or not. Few other classification examples are speech recognition, handwriting recognition, and document classification [25].

3.1 Random Forest Algorithm

Machine learning algorithms may be divided into three different categories: supervised learning, unsupervised learning, and reinforcement learning [15].

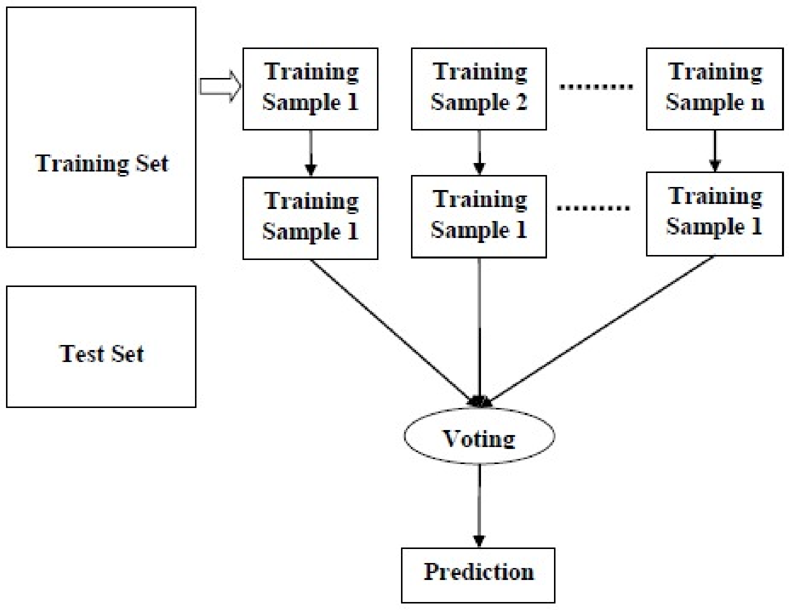

FIGURE 3.1: RANDOM FOREST [35]

Supervised learning is useful in cases where a particular dataset or a property (label) is available. The algorithm is required to predict or classify new data from a teacher/supervisor model. In cases where the task is to discover implicit associations in a given unlabeled dataset (items are not pre-assigned), unsupervised learning is useful[25].

Random Forest is a supervised learning algorithm. It can be used for both regression and classification [35]. It is also the algorithm that is the most versatile and straightforward to use.

Due to its high accuracy, the classifier has gained popularity in the remote-sensing community, where it is used in remote-sensed image classification [6].

The Random Forest classifier is a set of prediction trees where each tree depends on randomly sampled random vectors with a distribution close to any other tree in the set. A collection of trees comprises the forest; the more trees it has, the more resilient the forest is.

Random Forest generates decision trees on randomly selected data samples, gets predictions from each tree, and decides the best solution by voting [6]. It also offers a good indicator of the function's value and is used in function approximation applications.

In the process of training, the individuality of the trees is essential. Therefore, it is crucial to define a set of unique specifications for each tree. Moreover, random subsets of the initial training samples are used in the tree training [6]. In order to ensure that the tree is set properly, it is required to select an optimal split from the randomly chosen features of the unpruned tree nodes. Additionally, each tree grows without limits and should not be pruned.

The main features of the Random Forest algorithm are:

- High level of accuracy
- works effectively on large data sets
- Can handle thousands of input variables without eliminating any variable
- Provides estimates of the degree of importance of each variables in the classification
- As the forest building progresses, it produces an internal unbiased estimate of the generalization error.
- It has an efficient method of calculating missing data and preserves precision when there is a significant proportion of missing data.
- It has methods for balancing errors in unbalanced data sets of class population.
- Other data can save the created forests for future use.
- Prototypes that provide information on the relationship between the variables and the classification are computed.

- It calculates proximities between pairs of cases that can be used to cluster, identify outliers, or provide interesting views of the data.
- The above capabilities can be applied to unlabeled data, resulting in unmonitored clustering, data viewing and outlier detection.

3.1.1 Feature Importance

Another excellent quality of the Random Forest algorithm is that each function's relative significance on the forecast is straightforward to calculate.

Sklearn offers a great tool for this; it tests the value of a feature by examining how impurity is reduced across all trees in the forest by the tree nodes that use that feature [15]. After training, it calculates this score automatically for each feature and scales the results so that the sum of all significant features is equal to one.

3.1.2 Important Hyperparameters

The Random Forest hyperparameters are used to improve the model's predictive ability and to make the model is trained faster [26].

3.1.3 Increasing the Predictive Power

Random Forest benefits from the hyperparameter of n-estimators, which is just the number of trees that the algorithm creates before taking the full vote or taking prediction averages. A higher number of trees generally increases accuracy and makes the predictions more stable, but the estimation process is also slowed down.

Another relevant hyperparameter is the "Max" feature, which is the maximum number of Random Forest features considered to separate the nodes. For example, “leaf” is one of the meaningful hyperparameters, which specifies the minimum number of leaves required for an internal node to be broken.

3.1.4 Increasing the Training Speed

The “n” jobs hyperparameter tells the training engine how many processors it can use. If it has a value of one, only one processor can be used for it. A value of “-1” means that no limit exists.

The hyperparameter random state renders the performance of the model replicable. The model will always produce the same results when it has a definite random-state value, and if the same hyperparameters and training data have been given.

Finally, there is the “oob-score” (out-of-bag-score), which is a form of random forest cross-validation. Around one-third of the data is not used to train the model and assess its performance. Such samples are known as out-of-bag samples. It is quite similar to the approach of leave-one-out-cross-validation, but it goes along with almost no additional computational pressure [6].

3.2 Working of Random Forest Algorithm

Further knowledge about how the Random Forest operates is helpful to understand and use different choices. Two data objects created by random forests rely on most of the options.

Around one-third of the cases are left out of the sample when the current tree’s training is drawn by sampling with replacement. The oob data is used to achieve an unbiased running estimation of the classification error when adding trees to the forest. It is often used to achieve variable importance estimates. In replacing missing data, finding outliers, and generating illuminating low-dimensional data views, proximities are used [26].

All data is run down the tree after it is created, and proximities are computed for each pair of instances. If the same terminal node is occupied in two instances, their proximity is increased by one. At the end of the race, by dividing by the number of trees, the proximities are normalized.

3.2.1 The out-of-bag(oob) error estimate

In random forests, to achieve an unbiased estimation of the test set error, there is no need for cross-validation or a separate test set. Internally, during the race, it's calculated as follows:

By using a different bootstrap sample from the original data, each tree is built [14]. Conventionally one-third of the cases are left out of the bootstrap sample and not included in the kth tree construction. This value is a general rule and is not a set value.

3.2.2 Variable Importance

The oob cases are propagating down in every tree grown in the forest, the number of votes cast for the correct Class is counted. The values of variable m in the oob cases are now randomly permitted and placed down the tree.

Then the number of votes in variable-m-permuted oob data for the correct Class is subtracted from the number of votes in untouched oob data. The raw value score for variable m is the average of this number among all trees in the forest.

If the tree-to-tree score values are independent, a standard approach for calculating the error can be used. The correlations of the error scores between trees were measured for several points within the data sets. In this thesis, the errors are calculated classically; divide the raw score to get a z-score by its standard error, and allocate a significance level to the z-score assuming normality. If the number of variables is very high, forests with all variables can be run once [6].

3.2.3 Gini Impurity Criterion

The Gini impurity criterion for the two descendant nodes is less than the parent node any time a division of a node is made on variable m[25]. Adding the gini, decreases the overall trees in the forest for each individual variable and provides a simple variable significance that is also entirely compatible with the measure of permutation value.

3.2.4 Interactions

The concept of interaction is used if a split on one variable occurs. The implementation used is based on the gini, $g(m)$ values for each forest tree. These are ranked for each tree, and the absolute difference of their ranks overall trees is summed for every two variables [6].

The hypothesis that the two variables are independent of each other, and the latter is subtracted from the former also calculates this figure [31]. A big positive value for the gini means a split on one variable, and vice versa inhibits a split on the other [14]. This is an experimental technique whose findings must be treated with caution. On only a few data sets, it has been checked.

3.2.5 Proximities

In random woods, proximity is one o the most helpful variables for training. Initially, the proximities formed a matrix of $N \times N$. All of the results, both training and oob, are propagated down the tree after a tree is grown. If the k and n cases are in the same terminal node, their proximity will be increased by one. The vicinity is normalized at the end by dividing it by the number of trees [26].

For large data sets, it is possible that an $N \times N$ matrix exceeds the memory allocation of the system. In this case, a shift decreases the required memory size to $N \times T$, where T is the number of trees in the forest. The user is given the option of keeping only the largest proximities to each case to speed up the computational-intensive scaling and iterative missing value replacement.

When a test set is present, it is also possible to compute each case's proximity in the training set to ensure that the program can be run using the available computational resources.

3.2.6 Scaling

A matrix of $prox(n,k)$ is formed by the proximity between cases n and k. This matrix is symmetrical, positive, definite, and bounded by 1, with diagonal elements equal to 1.

It follows that $1 - prox(n,k)$ values are square distances of dimensions not greater than the number of cases in a Euclidean space [23].

Let $prox(-,k)$ over the 1-st coordinate be the average of $prox(n,k)$ and $prox(n,-)$ over the 2-nd coordinate to be the average of $prox(n,k)$, and $prox(-,-)$ over both coordinates. The matrix is then calculated as

$$cv(n,k) = 0.5 \times (prox(n,k) - prox(n,-) - prox(-,k) + prox(-,-)) \tag{3.1}$$

This matrix is the matrix of products of distances and is positive definite symmetric.

The eigenvalues of cv matrix is $\lambda(j)$ and its eigenvectors is $\nu_j(n)$. Then the vectors $x(n)$ are formed as follows:

$$x(n) = (\sqrt{\lambda(1)}\nu_1(n), \sqrt{\lambda(2)}\nu_2(n), ...,) \tag{3.2}$$

The idea is to approximate the vectors $x(n)$ by the first few scaling coordinates in metric scaling. This is achieved in random trees by extracting the largest eigenvalues and their corresponding eigenvectors of the matrix cv. The two-dimensional plot of the coordinate of i-th scaling vs. j-th also offers valuable data details. The graph of the 2-nd vs. the 1-st is generally the most useful.

Despite many of its advantages, the computational burden may be time-consuming because of its need to calculate the eigenfunctions of an $N \times N$ matrix.[23]. To make this approximation quicker, we suggest that $nrnn$ be considerably smaller than the sample size. This feature is used in the next chapter successfuly for the training.

There are more precise methods of projecting low-dimensional distances, such as the Roweis and Saul algorithms. But the strong performance of metric scaling, so far, has been sufficient for most applications. Velocity is another factor; for projecting down, metric scaling is the fastest existing algorithm. Generally, three or four coordinates for scaling are good to give a view of the data [23].

3.2.7 Prototypes

Prototypes are a way to get a picture of how the classification applies to the variables. For the j-th Class, we find the case that has among its k nearest neighbours the largest number of Class j instances, calculated using the proximities. For each vector, we find the median, 25-th percentile, and 75-th percentile among these k instances.

The medians are the $j$class prototype, and the quartiles have an approximation of stability. The process is repeated for the second prototype. However, only the cases are considered, which are not among the initial k, and so on [15].

Prototypes for continuous variables are standardized by subtracting the 5-th percentile and dividing by the gap between the 95-th and 5-th percentiles when the query for prototypes is to be executed. The prototype is the most prevalent value for categorical variables. All frequencies for categorical variables are provided when we ask for prototypes to be shown on the screen or saved to a computer.

3.3 Missing Value Replacement for the Training Set

There are two methods of replacing missing values in Random Forests. The first method is quick. If the m-th variable is not a categorical variable, the method calculates the median of all the values of the m-th variable in Class j, and then replaces all missing values of the m-th variable in Class j with this value [14].

If the m-th variable is a categorical variable, the most common non-missing value in Class j is the substitution variable. These values for replacement are called fills.

Computationally, the second way of replacing missing values is more costly but has provided better results than the first, even with large quantities of missing data [40]. Only in the training set does it substitute missing values. It starts by filling in the missing values approximately and inaccurately. It then does a forest run and measures proximity [14].

If $x(m, n)$ is a missing continuous value, its fill is estimated as an average over the m-th variables' non-missing values weighted by the proximities between the n-th case and the case of non-missing value.

3.4 Missing Value Replacement for the Test Set

There are two distinct types of substitution when there is a test set, depending on whether marks exist for the test set. If they do, then as substitutes, the fills extracted from the training set are used.

If labels do not exist, class n which is the number of classes will be repeated for each case in the test set. Class1 is considered to be the first duplicate of a case, and Class1 fills are used to replace missing values. The 2-nd replica is believed to be Class 2, and the fills used on it are $Class2$ [15].

The tree runs down this augmented test range. The one receiving the most votes decides the Class of the original case in each set of replicates.

3.5 Mislabeled Cases

By using human judgement to assign labels, the training sets are also created[15]. This contributes to a high level of mislabeling in certain regions. Using the outlier test, many of the mislabeled cases can be observed.

3.6 Outliers

In general, outliers are classified as cases that are excluded from the data's main body [40]. Outliers are cases whose proximity in the data is usually small compared to all other cases. An outlier in Class$j$ is, therefore, a case whose proximity to all other cases of Class$j$ is minimal [26].

The average proximity to the rest of training data Class$j$ from case n in Class$j$ is defined as follows:

$$P(n) = \sum_{d(k)=j} n, k$$

3.7 Advantages

- Because of the number of decision trees involved in the procedure, random forests are regarded as a highly accurate and robust system.
- The overfitting issue does not cause it to suffer. The primary explanation is that the sum of all the forecasts is taken, which cancels the biases.
- In both classification and regression problems, the algorithm can be used.
- Missing values can also be managed by random woods. There are two ways to deal with these: to replace continuous variables by using median values, [25] and to compute the proximity-weighted average of missing values.
- You can obtain the relative significance of the feature, which helps to get the most contributing features.

3.8 Disadvantages

- Random forests are slow to produce predictions because they have many trees of choice. Whenever a forecast is made, all trees in the forest have to make a forecast for the same feedback given and then vote on it. This whole approach is time-consuming. [15]

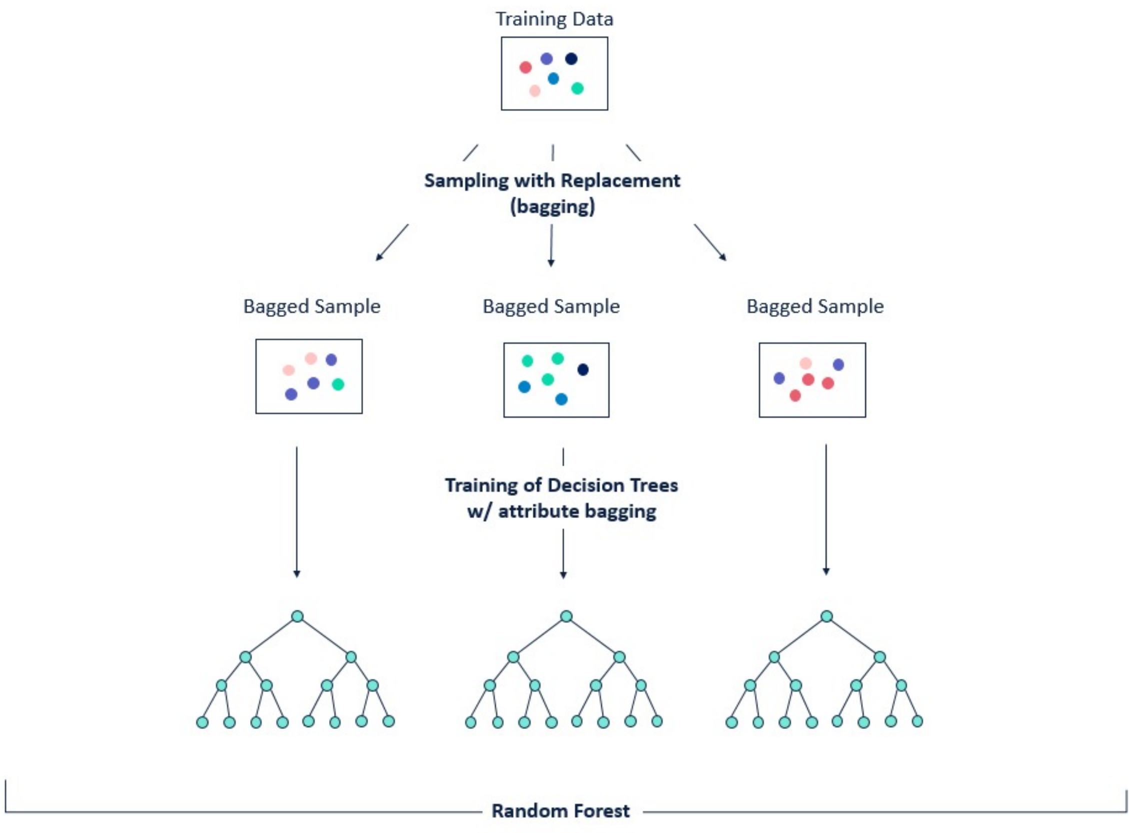
Training Data
Sampling with Replacement (bagging)
Bagged Sample
Bagged Sample
Bagged Sample
Training of Decision Trees w/ attribute bagging
Random Forest

FIGURE 3.2: RANDOM FOREST VISUALIZATION [21]

- Compared to a decision tree, the model is hard to read, where you can easily make a decision by following the route in the tree.

This research discussed Random Forest Classifier algorithm, how this algorithm works and its advantages and disadvantages. It contains the different methods applied to the random forest model to improve its accuracy and reduce training time. We briefly introduce features and important hyperparameters of random forest which includes the predictive power and model speed. Finally, we discuss working of random forest algorithm and methods for missing values replacement for training and test set.

Chapter 4

Designing Random Forest Model for Hardware Trojan Detection

This chapter presents the research methodology information, including data understanding, data preparation, and the Random Forest classifier model. Then the model is applied to detect hardware trojan. Then the research illustrates the result of the model corresponding to different test sets.

4.1 Fundamental Technology Background

In this section, the fundamental software that is required throughout software implementation is introduced. First, different technologies used to create the Random Forest model and its internal architecture are introduced.

FIGURE 4.1: Data Science Technologies

4.2 Creating the Random Forest Model

This research has used python programming for developing the code that runs the algorithm. Fig. **??** shows the logos of the used software packages.

1. Jupyter Notebook: Jupyter Notebook is a web-based open-source program for creating and sharing documents containing live code, equations, visualizations and storytelling texts. This software is used in this thesis to perform tasks such as data cleaning, data transformation, data visualization, and developing machine learning.

2. Scikit learn: It's a library that contains several machine learning algorithms. The Random Forest algorithm used in this work is developed using Scikit learn [40]. Moreover, this library contains the functions for splitting the dataset, confusion matrix, and cross-validation [15].

3. Pandas: This library is used for data manipulations and analysis. It handles all the operations related to the data frame.

4. NumPy: Using this library, the research had performed operations such as scaling the data and data transformation to convert the data frames into NumPy array for the machine learning model.

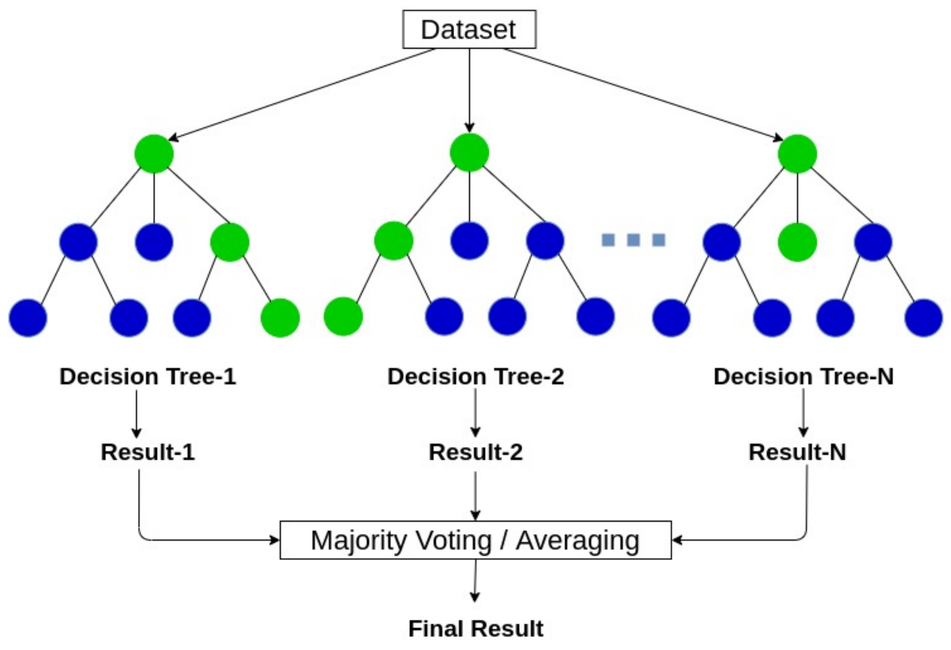

FIGURE 4.2: VISUALIZATION [40]

5. Matplotlib: Used for plotting the graphs and for data visualization. Fig. **??** shows a sample window from the visualization output window generated by this package.

4.3 Research Methodology

A research method is devised in a structured manner based on CRISP-DM, a data science methodology [8]. The research methodology and the process are described below. Fig. **??** presents various stages of the research.

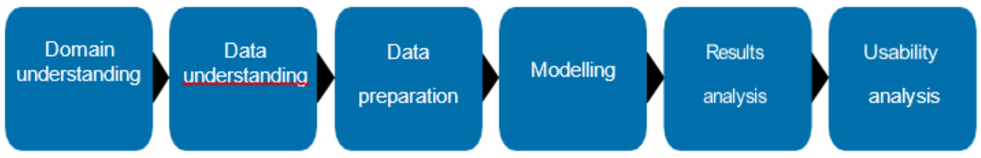

FIGURE 4.3: Research Methodology

4.3.1 Domain Understanding

This phase is needed to understand the domain of hardware trojan. Relevant literature on hardware trojan detection is found in this phase. Then, the impact of hardware trojan on cybersecurity and businesses is analyzed.

4.3.2 Data Understanding

The dataset used in this research is provided by the hardware cybersecurity team. This phase is required to understand the content of the dataset provided. The dataset content is explored with the use of multiple visualizations using matplotlib.

4.3.3 Data Preparation

The number of steps is needed to construct a dataset that can be used for the creation of detection models.

4.3.4 Modelling

This process consists of selecting machine learning techniques, setting up experiments, and training and testing machine learning techniques.

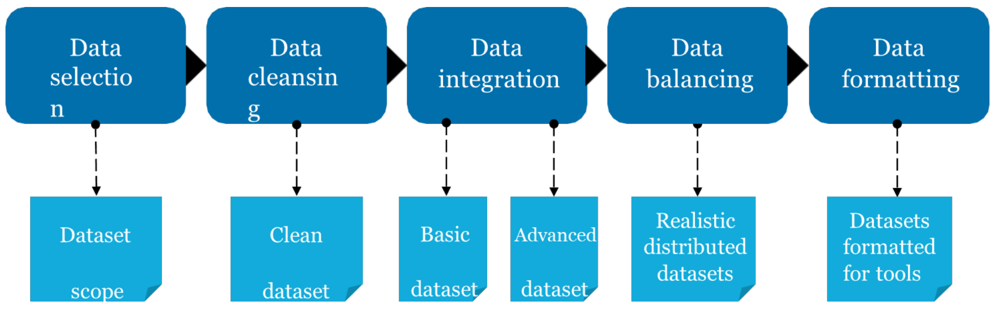

FIGURE 4.4: Data Preparation

4.3.5 Results Analysis

The results of the experiment are collected in this phase. The data outcome is shown in forms of true/phase, positive/negative categories.

4.4 Data Preparation

After cleaning up our dataset, this study had to make the data set ready for computer study, another step in data preparation. In this stage, the research has created three functions to get the final combined data as the research has 52 CSV files that need to be merged and then passed to Random Forest Classifier.

The function, def GetFilePath(number), was to fetch all the file paths and return it. Next function, def GetAllData( ), was used to iterate through all of the 52 CSV files and combined the data of al csv files into a single data frame which contains $2,599,948$ rows and 2 columns i.e $[2599948, 2]$ matrix, and returns the data frame which contains the data of all 52 CSV files. Moreover, during this process, the data is normalized. The data preparation mode was further devised in a structured manner based on CRISP-DM, a data science methodology [8]. The research methodology and the process are described below, and its structure is shown in Fig. 4.4.

4.4.1 Data Selection

The data selection is based on the relevance of attributes for the training and testing of different machine learning classifiers. Columns which are empty are dropped and some which columns should not be included in the experiment are needed for data integration process.

4.5 Data Cleaning

Machine learning requires training and feeding algorithms with data to accomplish various computational intensive tasks. However, organizations are typically challenged to have the right data for machine learning or to clean up irrelevant and error-prone data. In other words, most time is spent cleaning data sets or building an error-free data set while using ML data [27]. Establishing a quality plan, filling out missing values, eliminating rows, and reducing data size are some of the most commonly used techniques to create a useful dataset with quality data.

4.5.1 Fill-out missing values

One of the first moves to clean the data set errors is to search for and fill in missing values. It will categorize much of the data. It is easier to fill the missing values based on various categories or build new categories to include the missing values.

The research can use mean and medium to correct errors if the data is numerical or fill the mean with the missing data [8]. The average can be based on various factors as well.

There were several missing numerical values present in this work, and to fill those missing values, the mean value was used to fill it.

4.5.2 Removing rows with missing values

One of the easiest things in data cleansing is the removal or deletion of rows with missing values. This may not be the perfect move when the training data are subject to a large number of errors. If there are considerably lower missing values, then it is right to remove or remove missing values. You must be very confident that information present in the other rows of training data is not included in the data removed [25].

4.5.3 Fixing errors in the structure

Ensure there is no upper or lower case typographical defects and inconsistencies. Go through your data collection, find and correct these errors to ensure your training set is entirely errored free. This will allow you to achieve better results through the functions of your computer [14]. Delete the categorization duplication from your data list and simplify your data.

4.5.4 Removing Duplicate

Duplicates are repeated data points in your data collection. So this is a common problem in any dataset, and if there is such a problem, then one can remove those duplicate values from the dataset [23]. There were many redundant data in our dataset that the study found using NaN(), .isNull() functions available in python's numpy library.

4.6 Data Normalization

Normalization is a method often used in machine learning data preparation. Normalization aims to change numeric column values into a common dimension in the datasets without distorting values' rates.

All datasets do not need Normalization for machine learning. It is required only if features have numerous ranges [26]. Also, in this study, normalized data is used and converted to a useable form by removing the unnecessary columns from the datasets, which doesn't help a machine learning model find any pattern. The remaining columns are then converted into a dataframe that a machine learning model expects in the sklearn library. So the Normalize Data function, after removing the unnecessary data it returns a data frame.

4.7 Data Modeling

During this phase, the dataset is used for training and testing of hardware trojan detection. The figure shows the activity and its output in the modelling process.

4.7.1 Selection of machine learning techniques

In this section, the most promising machine learning techniques are selected. The Random Forest, Multi-layer perceptron and Neural Networks algorithm showed the highest performance as compared to the other machine learning algorithms in True Positive Rate and less amount of false values (False positive rate and False-negative rate). This research on the use of Random Forest for the detection of hardware trojan has shown better results.

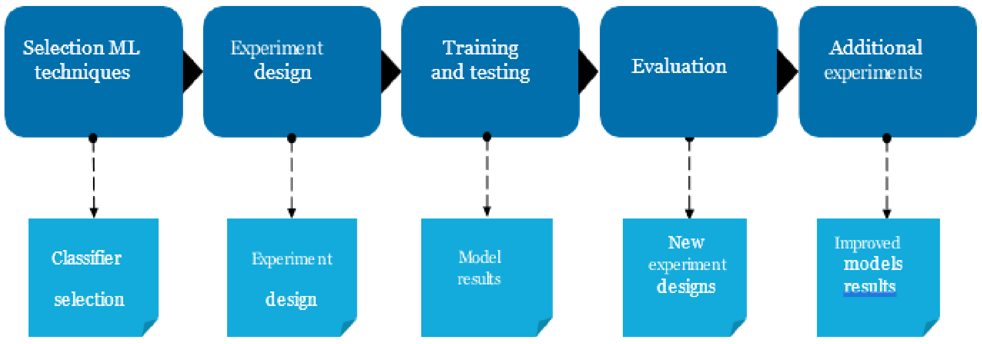

FIGURE 4.5: Modelling activities and output

4.7.2 Train/Test split

In order to make the final model, this study needs data on which it will feed a machine learning model with data, and some part is required to validate the data. Therefore, the final developed model is a machine learning model on which new data is passed to make a prediction.

In the developed model, the training dataset is a sample of data fed into the machine learning model to fit the model. The actual data that one uses to train a machine learning model and the model finds the train datasets' patterns and learns from this data. If the quality of data and if it is pre-processed properly, the model can learn properly [6]. The test dataset is a sample of data that provides an unbiased result for the final model, which is fit on the training data.

4.7.3 Validation Dataset

The data sample used to provide a model evaluation that is suitable for the training data set during tuning model hyperparameters. The assessment is made more inclusive by integrating ability in the validation dataset into the model configuration. This study uses results from validation and change hyperparameters of higher quality. The validation set, therefore, only affects a model indirectly [6].

4.7.4 Scaling the Dataset

StandardScaler assumes that the data is usually distributed in each function and scales it so, with a standard deviation of 1, the distribution is now centred around 0. StandardScaler eliminates the mean and values of each function/unit variance vector. The procedure is done separately from a function point of view. The StandardScaler can be affected by outliers (if they occur in a data set) since each function's empirical mean and standard deviation are calculated [6].

4.8 Confusion Matrix

There are several ways to evaluate the classificatory model's results, but none have been tested for a time such as the confusion matrix. It allows us to analyze how the model worked, how it went wrong. The model's performance can be viewed from a holistic perspective using confusion matrix [26].

A confusion matrix is an $N \times N$ matrix for the performance evaluation of a classification model in which N is the number of target groups. In this matrix, the real target values are compared to the machine learning model predicted. This offers a comprehensive view of our classification model's success and the types of errors it makes.

Two values are there in the target variable: Positive and Negative. Columns are the actual value of the target value. Rows are the predicted value of the target value.

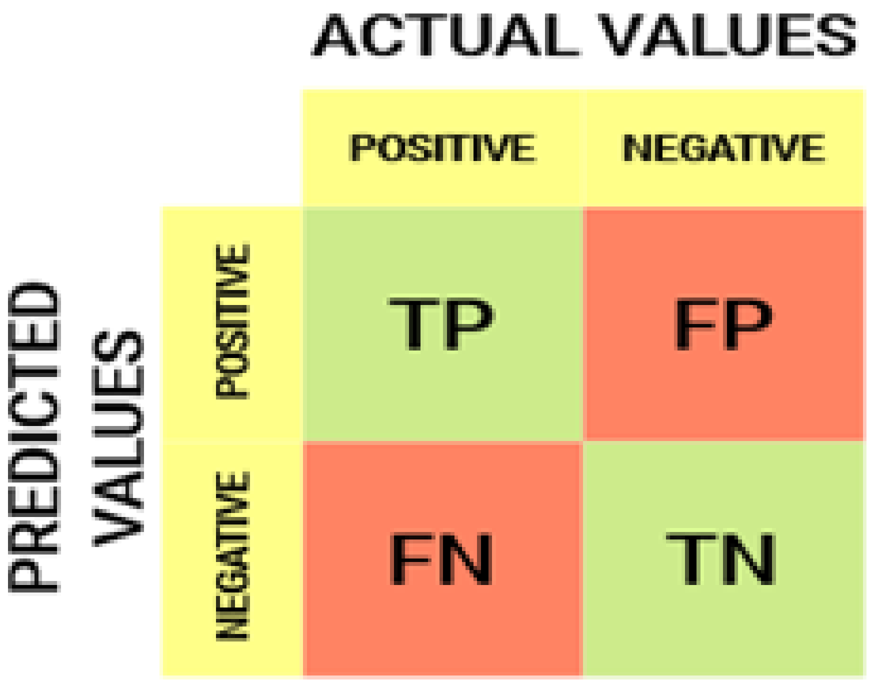

FIGURE 4.6: CONFUSION MATRIX [40]

4.9 Confusion Matrix for the Developed Model

The matric was developed for Trojan detection. A data set of $3,000,000$ samples from an encryption algorithm was provided to investigate the application of the trojan. The samples included the power consumption, a form of side-channel information, from the integrated circuit. Among the data, there were circuits with and without the Trojan circuit.

For this study, this information was provided and known to verify the effectiveness of the developed model. However, in real-life scenarios, this might not be possible. A trojan free or infected circuit is not easy to be operated from the rest.

The Confusion Matrix of the data set is presented Table **??**.

	Positive	Negative
Positive	25239	1655
Negative	1339	751752

TABLE 4.1: Confusion Matrix

- **True Positive (TP):** Predicted value matches the actual value. Actual value was positive and positive value was expected in the model. In this work, $TP = 25,239$; which means that, $25,239$ Positive class data points have been correctly classified by model.

- **True Negative (TN):** The value to be predicted matches the actual value. Actual value was negative and negative value was expected in the model. Here, $TN = 751,752$; which means that, 751752 negative class data points have been correctly classified by model.

- **False Positive (FP) - Also know ad Type 1 error:** The value which is predicted is falsely predicted. Actual value was negative and positive value was expected in the model. In our developed model, $FP = 1,655$; meaning that the model wrongly classifies $1,655$ negative class data points as being of the positive class.

- **False Negative (FN) - Type 2 error:** The value which is predicted is falsely predicted. Actual value was positive and negative value was expected in the model. For the developed model, $FN = 1,339$; meaning that the model wrongly classifies $1,339$ positive class data points as being of the negative class.

  The confusion matrix provides much detail of the accuracy and result of recognition. However, a more concise measure was also considered to evaluate the model's accuracy. These extra measures are presented next.

- **Accuracy** of a classifier on a given data points is the percentage of test setups properly identified by the classifier. Below is the equation used to calculate the accuracy from the confusion matrix.

$$Accuracy = (TP + TN)/(TP + FP + TN + FN) \tag{4.1}$$

In our model, $TP = 25,239$, $TN = 751,752$, $FP = 1,655$, $FN = 1,339$. The accuracy for our Random Forest Classifier turned out to be

$$Accuracy = (25239+751752)/(25239+751752+1655+1339) = 99.80 \tag{4.2}$$

- **Precision** is evaluated as $TP/(TP + FP)$, where TP is the true positive number and FP is the false positive number. In the developed model, precision is evaluated to be equal to:

$$Precision = 25239/(25239 + 1655) = 93.84 \tag{4.3}$$

- **Recall** provides information on how many of the positive cases with our model have been correctly predicted, and is calculated as follows:

$$Recall = TP/(TP + FN)25239/(25239 + 1339) = 94.75 \tag{4.4}$$

- **F1- Score** is a weighted average of the actual positive (recall) rate and precision, and is equal to $2(precision * recall/precision + recall)$.

$$F1 - score = 2(93.84 * 94.75/93.84 + 94.75) = 2(8891.34/188.59) = 94.29 \tag{4.5}$$

4.10 Cross validation

Data is divided into k sub-sets in K Fold cross validation. Now the method of holdout is repeatedly used k times to construct a training set with one of the k

subsets each time and the other $k-1$ subsets are assembled. In all k trials the error estimation is summed to achieve full efficacy of our model. Each data point is used exactly once in a validation set and in a training set $k-1$ times [14]. This decreases the bias significantly as this research uses most fitting data and decreases the variance also significantly as the bulk of data is also used in validation sets. The sharing of training and test sets also contributes to the efficiency of the process.

For the data set, the k value must be carefully selected. A poorly selected value for k could lead to a misrepresenting interpretation of the model's ability, like a score that has a high variance (which could differ a great deal depending on the data used for the model), or a high bias (e.g. overestimating the model's skills) [28]. The value for k is selected to make each train / test group of data samples big enough to represent the wider dataset statistically. So for our program it took $cv/k = 5$. The following command was used to indicate this value: *rfcValue=cross val score(randomForestClassifier,X train,y train,cv=5)*

The first argument passed in the *cross val score* is the random forest classifier model which this research have developed. The next two parameters are the training data in which X train contains the features and y train contains the labels. Now the main variable is *cv* which helps to define the value of k in k-fold cross-validation. Once the operation is completed this, *cross val score* returns an array of score of the estimator for each run of the cross validation [24]. Fig. 4.7 shows the k fold cross validation scheme.

Out-of-bag (OOB) error, also called out-of-bag estimate, is a method of measuring the prediction error of random forests, boosted decision trees, and other machine learning models utilizing bootstrap aggregating (bagging) to sub-sample data samples used for training. In order to see how many trees are necessary in my forest, the OOB error is plotted as the number of trees used in the forest is increased. A new random forest on each iteration is created with increasing numbers of trees

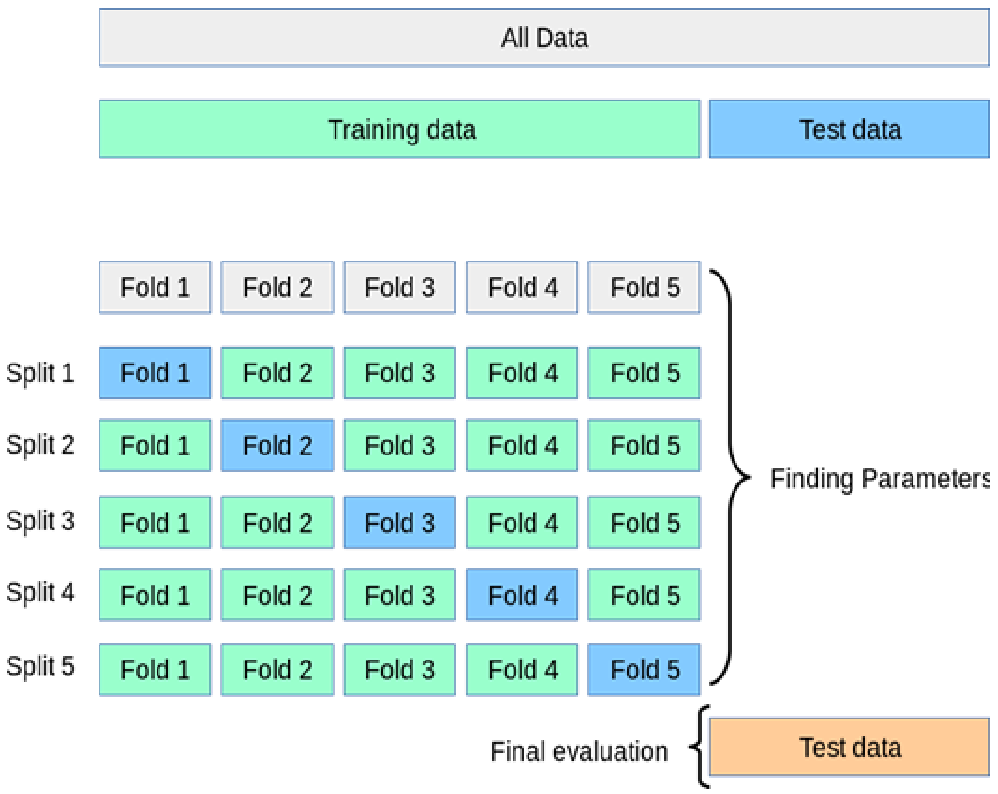

FIGURE 4.7: k-fold cross validation

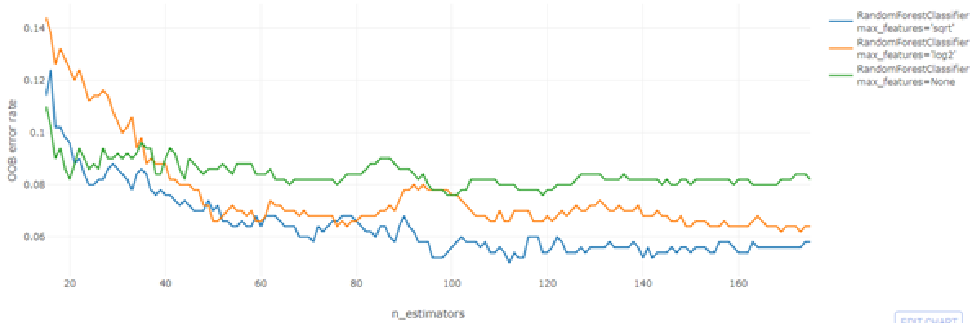

FIGURE 4.8: OOB error rate vs. N esmitator

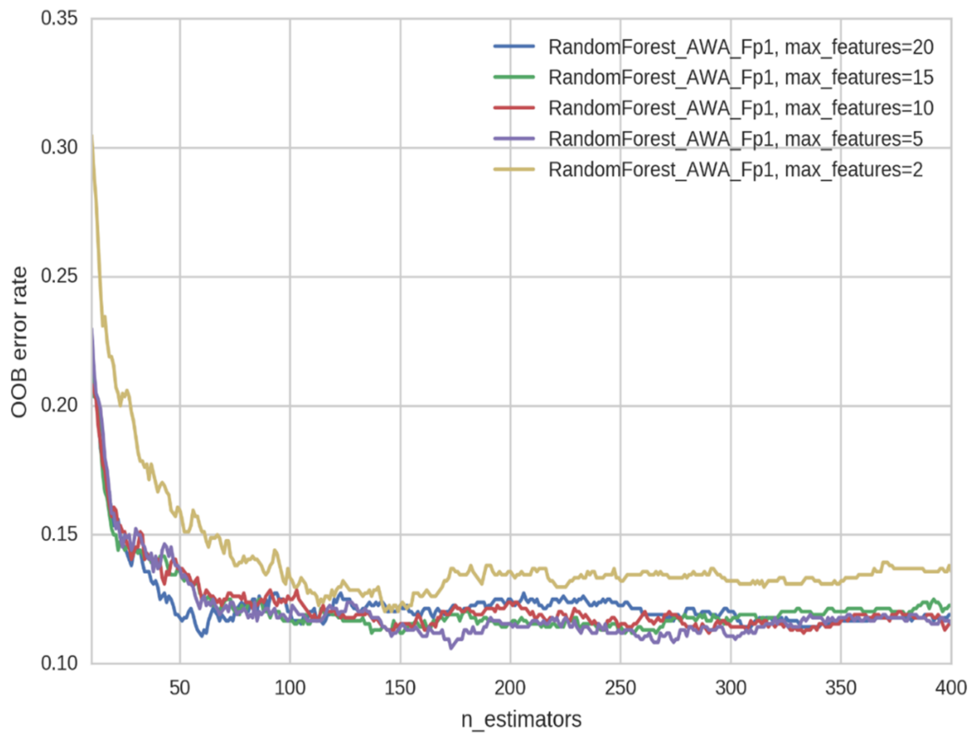

FIGURE 4.9: OOB ERROR RATE VS N ESTIMATOR

but this is too expensive [25]. Fig. 4.8 shows the OOb error for the developed model.

In random forests, there is no need for cross-validation or a separate test set to get an unbiased estimate of the test set error. It is estimated internally, during the run, as follows [21]: Each tree is constructed using a different bootstrap sample from the original data. About one-third of the cases are left out of the bootstrap sample and not used in the construction of the k-th tree. One out of two cases is left out in the construction of the k-th tree down to get a classification. In this way, a test set classification is obtained for each case in about one-third of the trees. At the end of the run, j is the class that got most of the votes every time case n was oob. The proportion of times that j is not equal to the true class of n averaged over all cases is the oob error estimate. This has proven to be unbiased in many tests [8].

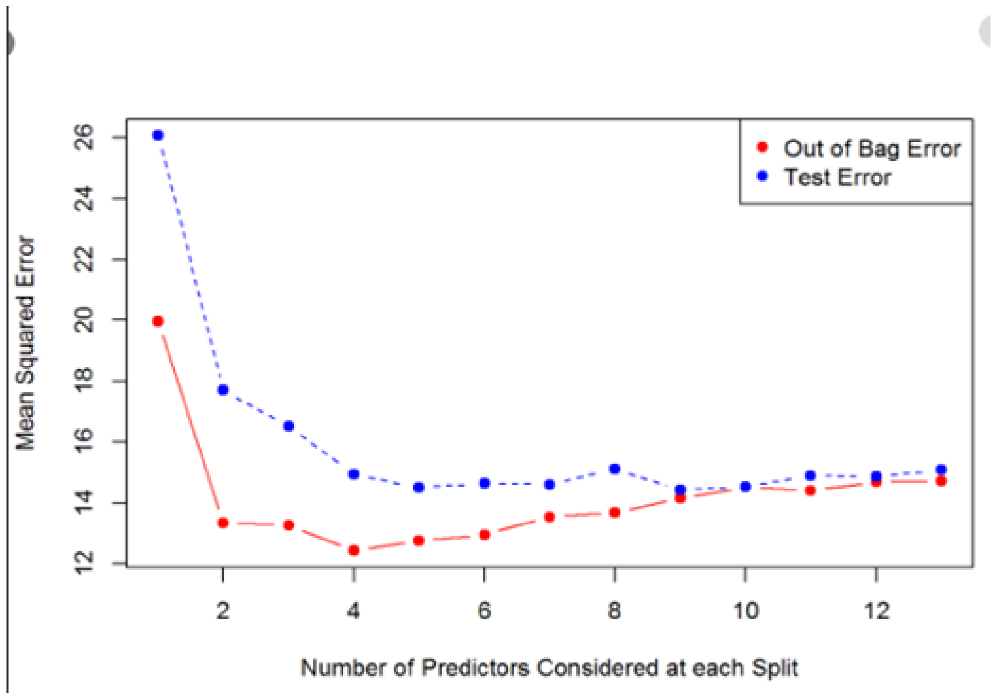

FIGURE 4.10: MSE vs. the number of predictor used at each split

In Fig. 4.10 the Red line is the Out of Bag Error Estimates and the Blue Line is the Error calculated on Test Set. Both curves are quiet smooth and the error estimates are somewhat correlated too. The Error Tends to be minimized at around $mtry = 4$. On the Extreme Right Hand Side of the above Plot all possible 13 predictors at each Split are shown which is Bagging.

There are two other parameters in Random Forest that require attention: number of trees and minimum node size. In order to find out the value of the trees of (ntree)in Random Forest, this research used *TrainSet* and *ValidationSet* for training and testing, respectively. Then the OOB error rate and train error rate with the independent test set (Va-lidationSet) error rate were compared as shown in Fig. ??. The plot shows that the OOB error rate follows the test set error rate fairly and closely, when the number tress are more than 100. Therefore, the sufficient number of tree is found around 100 [6].

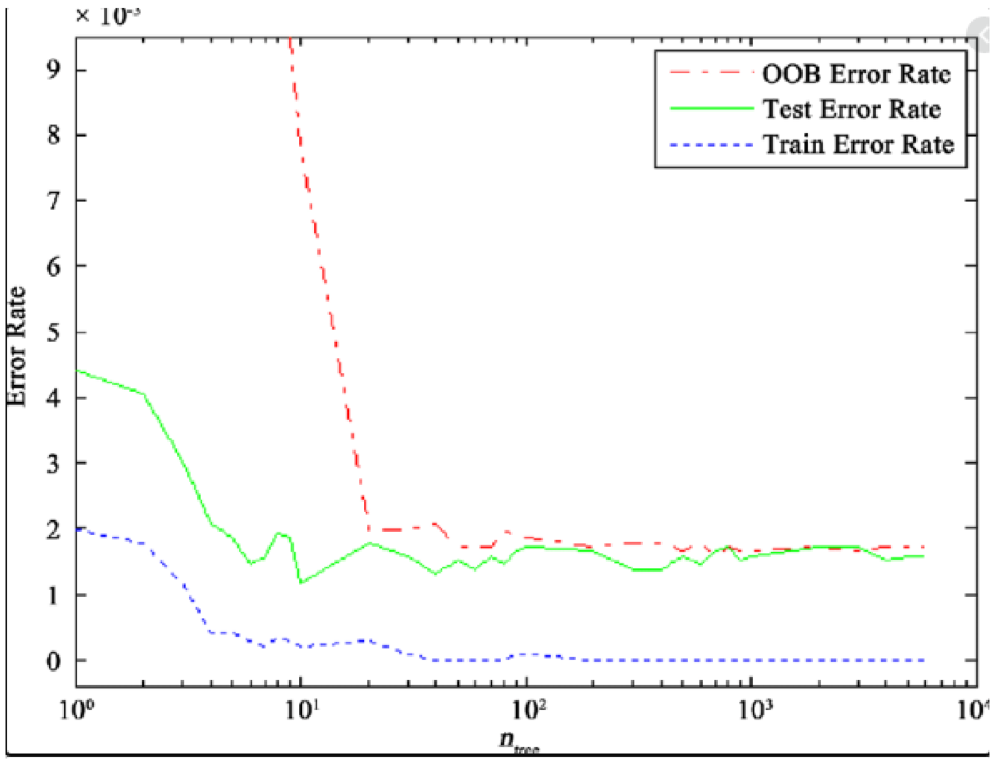

FIGURE 4.11: Error rate vs. time

In addition to it, Fig. ?? also shows an interesting phenomenon which is the characteristic of Random Forest; the test and OOB error rates do not increase after the training error reaches zero. Instead they converge to their "asymptotic" values, which is close to their minimum.

So far, we have discussed the fundamental technology used to create random forest model, including data preparation, data cleaning which consists of fill-out missing values, fixing errors and removing rows with missing values, data normalization. Then this research use train/test split and confusion matrix and cross validation method to train the model. Next, the developed solution is used to make a comparison with published results achieved using machine learning techniques and achieve a considerable result.

4.11 Grid Search in Random forest

Tuning of parameters is an important element in achieving ideal parameter values in the machine learning algorithms. Several studies and methods for the tuning of the parameters were successfully proposed to ensure higher accuracy in classification models.

Adjusting *hyperparameters* carefully and methodically can be helpful. It will improve the precision of the classification model, resulting in more precise predictions in general.

The tuning of *hyperparameters* is based more on experimental findings than on theory, and therefore the best way to decide the optimal setting is to test the output of each model using several different combinations.

Grid search enables one to do so simultaneously with many parameters to find the best parameters for the given data. In here, several different parameters were picked in order to have a set of values to choose from. The grid search will then fit models into any combination of those parameter values, using cross to evaluate each case's output.

Output from the grid search is presented below for the top 4 models in term of accuracy:

Model with rank: 1
Accuracy: 99.82%
Parameters: 'n_estimators': 500, 'min_samples_leaf': 5,
'min_weight_fraction_leaf': 0.1,
'max_depth': 7, 'max_leaf_nodes': 30, 'criterion':'gini',
'min_samples_split': 15,'max_feature':'sqrt'

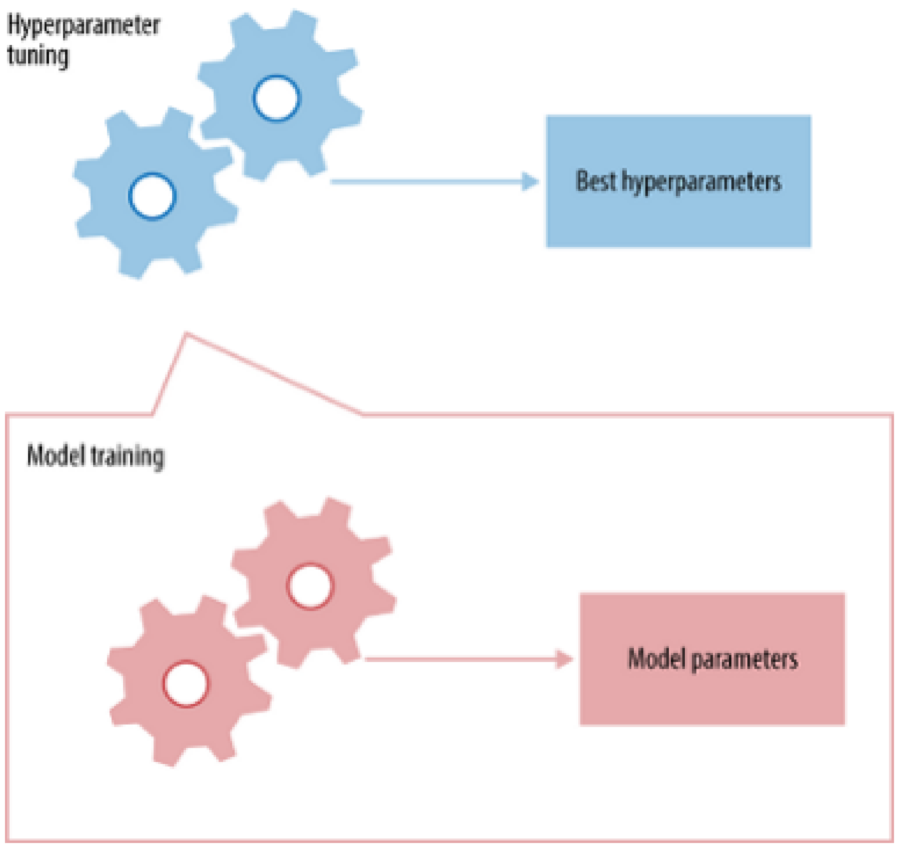

FIGURE 4.12: Parameter tuning

Model with rank: 2

Accuracy: 99.80%

Parameters: 'n_estimators': 400, 'min_samples_leaf': 10,

'min_weight_fraction_leaf': 0.1,

'max_depth': 9, 'max_leaf_nodes': 20, ‘criterion':'gini',

'min_samples_split': 20, 'max_feature':'sqrt'

Model with rank: 3

Accuracy: 98.45%

Parameters: 'n_estimators': 200, 'min_samples_leaf': 20, 'min_weight_fraction_leaf': 0.1,

'max_depth': 7, 'max_leaf_nodes': 20, 'criterion':'gini',

'min_samples_split': 15, 'max_feature':'log2'

Model with rank: 4

Accuracy: 97.79%

Parameters: 'n_estimators': 700, 'min_samples_leaf': 10,

'min_weight_fraction_leaf': 0.1,'max_depth': 5,

'max_leaf_nodes': 30, 'criterion':'gini', 'min_samples_split': 15, 'max_feature':'sqrt'

So from the above 4 models, that the current approach has achieved a good accuracy, and is a reliable method for detecting the trojan.

From the above table, it is shown the accuracy for model 1 is the best i.e 99.82 but the training time is quite more as compared to the model 2. The accuracy of model 2 is just 0.02 less than model 1, so the difference in accuracy is not significant. But considering the training time in this study, it can be concluded that model is best as it takes less time for training.

Approach	TPR	TNR	Accuracy
SVM	83%	49%	51%
NN	81%	69%	69%
Multi-NN	85%	70%	73%
Random Forest	68%	99.7%	99%

Table 4.2: Accuracy Comparison [8]

Comparison results are shown in table above, which shows the classification of the same data set among four different machine learning algorithms used to detect hardware trojan. The term NN refers to the neural network based approach and Multi-NN refers to the multi-middle-layer networks.

Trojans identified to be hardware trojans correctly are called true positives. TP shows the number of true positives. Trojans which are mistakenly identified normal are called False Negatives and FN shows the number of false negatives. The True Positive rate is defined by $TPR = TP/TP + FN$ and True Negative Rate is defined by $TNR = TN/TN + FP$. The accuracy is defined by $(TP + TN)/(TP + FN + FP + TN)$.

This study has applied the grid search approach that is implemented in *Grid-SearchCV* to find optimal parameters of Random Forest algorithm. Experimental results on the Trojan dataset shows that Random Forest provides the best classifier with the accuracy of 0.9980 compared to those of other classification algorithms namely SVM, MLP and Neural Network. Tuning eight parameters of Random Forest results optimal values. The results show that tuning parameter has successfully generated the best classifier to classify a new data.

Chapter 5

Conclusion

In this chapter, it summarize and conclude the research of this thesis. Firstly, the achievement of our algorithm is presented and then propose the future work in related implementation.

5.1 Summary of Contribution

This research discussed the problem of hardware trojan detection. This research worked on the goal how can one detect hardware trojan using Random Forest with maximum accuracy so that a person can be sure that they are not compromising security of the system.

This study analysed the effectiveness of this method and compared with various existing trojan detection techniques and found out that most of these detection techniques uses Golden Circuit which is considered as trojan free design. It is more viable for an attacker to insert a stealthy trojan into every fabricated integrated circuit that passes manufacturing process and trust validations. This process assumes that trojans are inserted into random ICs and raises the challenge of detecting trojan in ICs without relying on golden ICs. So, this study replaced Golden Circuit with Simulations. It allows the user to draw a circuit design and then have the computer evaluate the circuit. It intercepts text listing describing the circuit and outputs the results of its detailed mathematical analysis tool. After extraction of data from simulation, data was prepared and processed. Then, Data is fed to our machine leaning pipeline to be trained and validated. ML pipeline consists of 3 algorithms: SVM, Random Forest, Neural Network. In order to fit the best one into the particular case, the output of each of them is studied according to several classification metrics. Random forest was set as the most accurate model as a comparison of performance between the different ML algorithms through several metrixs, namely Confusion Matrix, and F1-Measuremethod. This study applied the grid search that is implemented in GridSearchCV to find the optimal paramteres of Random Forest algorithm. Using this approach, this study tested all possible parameter values combinations and preserved the combination through which this research got the highest accuracy.

After applying GridSearchCV this study got four models with highest accuracy out of which model1 gave accuracy of 99.82 and model2 gave accuracy of 99.80. Furthermore, when this study focussed on the training time of model1 and model2, model1 took more than one hour and model2 took 48 minutes of training time. In terms of accuracy this study can say that model1 has more accuracy than model2, but training time is less for model2 as compared to model1. So, this study can say that model2 will be more reliable for real-world applications. It confirm that Random Forest is better in terms of performance and good prediction of positive and negative samples. Results of this review show that the classifier of the Random Forest is most appropriate for the detection of hardware trojans With an FPR of 0.0022 and a FNR of 0.0504, the f1 score is 0.9440. Additionally, this research provided: 1. Knowledge on which hyperparameters are important to detect hardware trojans. 2. Extensive analysis of influence of hyperparameters on the performance of multiple classifiers. 3. A global cost-benefit analysis of IC security options for businesses.

5.2 Future Work

Proposed machine learning model using Random forest algorithm effects the most research efforts on hardware trojan detection. In this thesis, it talks about training random forest model using simulations data of power traces collected to detect hardware trojan. In this thesis, this research finishes the first step, and define that proposed Random forest model modified is the better in critical path among the other methods. In the optimization work, to make this result more persuasive, this research will try to ensemble more algorithms to reduce error estimation using bagging techniques and with the advancement of new technology and methods. This will focus on improving False Negative Rates and False Positive Rates. While random forest showed high performance, other ensemble methods can be examined such as Gradient Boosting. So the potential work will discuss how this algorithm can be used in a parallel distributed environment so that the training data can be distributed over multiple instances of distributed system which can reduce the training time implementation of the machine learning models with improved accuracy to detect trojans.

Bibliography

[1] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware trojans in third-party digital ip cores," 2011.

[2] M. B. S. Bhunia and S. Narasimhan, "Hardware trojan attacks: Threat analysis and countermeasures," 2014.

[3] M. M. Potkonjak and T.Massey, "Hardware trojan horse detection using gate-level characterization," 2009.

[4] R. S.Narasimhan, D.Du, "Hardware trojan detection by multiple-parameter side-channel analysis," *IEEE Transactions on Computers*, vol. 62, no. 11, pp. 2183–2195, 2012.

[5] S. S.Narasimhan, D.Du, "Multiple-parameter side-channel analysis: A non-invasive hardware trojan detection approach," *IEEE International Symposium on Hardware-Oriented Security and Trust*, 2010.

[6] L. Y.Xiang and W.Zhou, "Random forest classifier for hardware trojan detection," *International Symposium on Computational Intelligence and Design*, vol. 2, no. 5, pp. 134–137, 2019.

[7] M. S.Bhunia, M.S.Hsiao and S.Narasimhan, "Hardware trojan attacks: threat analysis and countermeasures," vol. 102, pp. 1229–1247, IEEE, 2014.

[8] M. K.Hasegawa and N.Togawa, "A hardware-trojan classification method using machine learning at gate-level netlists based on trojan features,," *IEEE International Symposium on Hardware-Oriented Security and Trust*, vol. E100-A, no. 7, pp. 1427–1438, 2017.

[9] A. B. Ziad, Alanwar, "Homomorphic data isolation for hardware trojan protection," *IEEE Computer Society Annual Symposium on VLSI*, vol. 36, no. 2, pp. 216–222, 2015.

[10] S. Wei and M. Potkonjak, *Scalable Hardware Trojan Diagnosis.* 2011.

[11] B.Shakya, "Benchmarking of hardware trojans and maliciously affected circuits," *Hardware and Systems Security*, vol. 1, no. 1, pp. 85–102, 2017.

[12] S. S.Narasimhan, "Hardware trojan detection by multiple-parameter side-channel analysis," *IEEE Transactions on Computers*, vol. 62, no. 11, pp. 2183–2195, 2012.

[13] C.Bishop, *Pattern Recognition and Machine Learning.* Springer, 2006.

[14] A.Ng, *Machine Learning Course, Stanford University.* Coursera, 2017.

[15] s.-l. Blondel et al., "Machine learning in python," 2017.

[16] T.Reece and W.H.Robinson, "Detection of hardware trojans in third-party intellectual property using untrusted modules," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 3, pp. 357–366, 2016.

[17] E. A. Lee, "Cyber physical systems: Design challenges," in *11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, IEEE, 2008.

[18] S.Skorobogatov and C.Woods, "Breakthrough silicon scanning discovers backdoor in military chip," 2012.

[19] C. D.Forte and A.Srivastava, “Temperature tracking: An innovative run-time approach for hardware trojan detection,” 2013.

[20] D. C.Bao and A.Srivastava, “On reverse engineering-based hardware trojan detection,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2014.

[21] Y. A.Kulkarni and T.Mohsenin, “Adaptive real-time trojan detection framework through machine learning,” in *IEEE International Symposium on Hardware Oriented Security and Trust*, 2016.

[22] B. S.Bhasin, “Hardware trojan horses in cryptographic ip cores,” 2018.

[23] W. B.Deng, X.Zhang and D.Shang, “An overview of extreme learning machine,” *International Conference on Control, Robotics and Cybernetics*, vol. 29, no. 2, pp. 189–195, 2019.

[24] L.Breiman, *Random forests,Machine learning.* Springer, 2001.

[25] M. K.Hasegawa and N.Togawa, “Designing hardware trojans and their detection based on a svm-based approach,” *IEEE 12th International Conference on ASIC (ASICON)*, 2017.

[26] J.Rajendran, “Towards a comprehensive and systematic classification of hardware trojans,” *IEEE Intl Symp. Circuits and Systems*, vol. 16, no. 1, pp. 1871–1874, 2010.

[27] Y. A.Kulkarni and T.Mohsenin, “Svm-based real-time hardware trojan detection for many-core platform,” *IEEE signal processing magazine*, vol. 3, no. 4, pp. 362–367, 2016.

[28] F. F.Martinelli and F.Mercaldo, “Evaluating convolutional neural network for effective malware detection,” *IEEE Proceeding of Computer Science*, vol. 112, no. 1, pp. 2372–2381, 2017.

Appendix

Python Codes

```
import pandas as pd
import seaborn as sb
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn import svm
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import confusion_matrix

from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score

#Normalizes the data and converts it in the useable form
def NormalizeData(data):
data=data.drop('Record Length',axis=1)
data=data.drop('50000',axis=1)
data=data.drop('Points',axis=1)
#Creating and setting up new data frome
newDataFrame=pd.DataFrame()
```

```
newDataFrame['Points']=data.iloc[:,0]
newDataFrame['Samples']=data.iloc[:,1]
return newDataFrame

def GetFilePath(number):
return "C:\\Users\\Admin\\trace"+str(number)+"Wfm.csv"



#Combines two dataframes
def CombineData(data1,data2):
data2=NormalizeData(data2)
combinedData=pd.DataFrame()
combinedData['Points']=data1['Points'].append(data2['Points'])
combinedData['Samples']=data1['Samples'].append(data2['Samples'])
return combinedData

def GetAllData():
#The main data frame object
data=pd.DataFrame()
data=pd.read_csv(GetFilePath(1))
data=NormalizeData(data)
for i in range(2,53):
newData=pd.read_csv(GetFilePath(i))
data=CombineData(data,newData)
return data

#trace=pd.read_csv("C:\\Users\\Admim\\trace1Wfm.csv")

trace=GetAllData()
```

```
trace=NormalizeData(trace)


bins=(0.8,1,1.1)
group_names=['bad','good']
trace['Samples']=pd.cut(trace['Samples'],bins=bins)
#Should be separate from the above preprocessing code
label_quality=LabelEncoder()
trace['Samples']=label_quality.fit_transform(trace
['Samples'])



#Separating data
X=trace.drop('Samples',axis=1)
y=trace['Samples']

#Spliting training and test data
X_train,X_test,y_train,y_test=train_test_split
(X,y,test_size=0.3)

sc=StandardScaler()
X_train=sc.fit_transform(X_train)
X_test=sc.transform(X_test)

#Creating a random forest classifier
rfc=RandomForestClassifier(n_estimators=200)
rfc.fit(X_train,y_train)
pred_rfc=rfc.predict(X_test)

#Lets see how our modl performed
```

```
print(classification_report(y_test,pred_rfc))
print(confusion_matrix(y_test,pred_rfc))

#SVM Classifier
clf=svm.SVC()
clf.fit(X_train,y_train)
pred_clf=clf.predict(X_test)

#Lets see how our modl performed
print(classification_report(y_test,pred_clf))
print(confusion_matrix(y_test,pred_clf))



#Neural Network
mlpc=MLPClassifier(hidden_layer_sizes=
(11,11,11),max_iter=500)
mlpc=mlpc.fit(X_train,y_train)
pred_mlpc=mlpc.predict(X_test)

#Lets see how our model performed
print(classification_report(y_test,pred_mlpc))
print(confusion_matrix(y_test,pred_mlpc))

#Model Selection
randomForestClassifier=RandomForestClassifier
(n_estimators=200)
clf = svm.SVC(kernel='linear', C=1)
mlp=MLPClassifier(hidden_layer_sizes=(11,11,11)
,max_iter=500)
```

```
rfcValue=cross_val_score(randomForestClas,
X_train,y_train,cv=5)
clfValue=cross_val_score(clf,X_train,y_train,cv=5)
mlpValue=cross_val_score(mlp,X_train,y_train,cv=5)

#Checking which model is how much accurate
print("Accuracy: ",rfcValue.mean()*100)
print("Accuracy: ",clfValue.mean()*100)
print("Accuracy: ",mlpValue.mean()*100)
```

```
#elm.py

from abc import ABCMeta, abstractmethod

import numpy as np
from scipy.linalg import pinv2

from sklearn.utils import as_float_array
from sklearn.utils.extmath import safe_sparse_dot
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.base import RegressorMixin
from sklearn.preprocessing import LabelBinarizer

from random_layer import RandomLayer, MLPRandomLayer

__all__ = ["ELMRegressor",
"ELMClassifier",
"GenELMRegressor",
"GenELMClassifier"]


# BaseELM class, regressor and hidden_layer attributes
# and provides defaults for docstrings
class BaseELM(BaseEstimator):
__metaclass__ = ABCMeta

def __init__(self, hidden_layer, regressor):
self.regressor = regressor
self.hidden_layer = hidden_layer

@abstractmethod
def fit(self, X, y):

def predict(self, X):


class GenELMRegressor(BaseELM, RegressorMixin):

def __init__(self,
hidden_layer=MLPRandomLayer(random_state=0),
regressor=None):

super(GenELMRegressor, self).__init__
(hidden_layer, regressor)

self.coefs_ = None
self.fitted_ = False
self.hidden_activations_ = None

def _fit_regression(self, y):
"""c
fit regression using pseudo-inverse
or supplied regressor
"""
if (self.regressor is None):
self.coefs_ = safe_sparse_dot(pinv2
(self.hidden_activations_), y)
else:
self.regressor.fit(self.hidden_activations_, y)

self.fitted_ = True

def fit(self, X, y):
# fit random hidden layer and compute the activations
self.hidden_activations_ =
self.hidden_layer.fit_transform(X)

# solve the regression from hidden activations to outputs
self._fit_regression(as_float_array(y, copy=True))

return self

def _get_predictions(self):
"""get predictions using internal least squares"""
if (self.regressor is None):
preds = safe_sparse_dot
```

```
(self.hidden_activations_, self.coefs_)
else:
preds = self.regressor.predict
(self.hidden_activations_)

return preds

def predict(self, X):
"""
Predict values using the model

Parameters
----------
X : {array-like, sparse matrix} of shape
 [n_samples, n_features]

Returns
-------
C : numpy array of shape [n_samples, n_outputs]
Predicted values.
"""
if (not self.fitted_):
raise ValueError("ELMRegressor not fitted")

# compute hidden layer activations
self.hidden_activations_ = self.hidden_layer.transform(X)

# compute output predictions for new hidden activations
predictions = self._get_predictions()
```

```
return predictions


class GenELMClassifier(BaseELM, ClassifierMixin):
def __init__(self,
hidden_layer=MLPRandomLayer(random_state=0),
binarizer=LabelBinarizer(-1, 1),
regressor=None):

super(GenELMClassifier, self).__init__
(hidden_layer, regressor)

self.binarizer = binarizer

self.classes_ = None
self.genelm_regressor_ = GenELMRegressor
(hidden_layer, regressor)

def decision_function(self, X):
"""
This function return the decision function
values related class on an array of test vectors X.

Parameters
----------
X : array-like of shape [n_samples, n_features]

Returns
```

```
-------
C : array of shape [n_samples, n_classes] or [n_samples,]
Decision function values related to each class, per sample.
In the two-class case, the shape is [n_samples,]
"""
return self.genelm_regressor_.predict(X)

def fit(self, X, y):
"""
Fit the model using X, y as training data.

Parameters
----------
X : {array-like, sparse matrix} of shape
 [n_samples, n_features]
Training vectors, where n_samples is the
number of samples and n_features is the number of features.

y : array-like of shape [n_samples, n_outputs]
Target values (class labels in classification, real numbers
regression)

Returns
-------
self : object

Returns an instance of self.
"""
self.classes_ = np.unique(y)
```

```
y_bin = self.binarizer.fit_transform(y)

self.genelm_regressor_.fit(X, y_bin)
return self

def predict(self, X):
"""Predict values using the model

Parameters
----------
X : {array-like, sparse matrix} of shape
[n_samples, n_features]

Returns
-------
C : numpy array of shape [n_samples, n_outputs]
Predicted values.
"""
raw_predictions=self.decision_function(X)
class_pred=self.binarizer.inverse_transform(raw)

return class_predictions

# ELMRegressor with default RandomLayer
class ELMRegressor(BaseEstimator, RegressorMixin):

def __init__(self, n_hidden=20, alpha=0.5, rbf_width=1.0,
activation_func='tanh', activation_args=None,
user_components=None, regressor=None, random_state=None):

self.n_hidden = n_hidden
self.alpha = alpha
self.random_state = random_state
self.activation_func = activation_func
self.activation_args = activation_args
self.user_components = user_components
self.rbf_width = rbf_width
self.regressor = regressor

self._genelm_regressor = None

def _create_random_layer(self):
"""Pass init params to RandomLayer"""

return RandomLayer(n_hidden=self.n_hidden,
alpha=self.alpha, random_state=self.random_state,
activation_func=self.activation_func,
activation_args=self.activation_args,
user_components=self.user_components,
rbf_width=self.rbf_width)

def fit(self, X, y):
"""
Fit the model using X, y as training data.

Parameters
----------
```

```
X : {array-like, sparse matrix} of shape
[n_samples, n_features]
Training vectors, where n_samples is the number
of samples and n_features is the number of features.

y : array-like of shape [n_samples, n_outputs]
Target values (class labels in classification, real
numbers in regression)

Returns
-------
self : object

Returns an instance of self.
"""
rhl = self._create_random_layer()
self._genelm_regressor = GenELMRegressor
(hidden_layer=rhl,
regressor=self.regressor)
self._genelm_regressor.fit(X, y)
return self

def predict(self, X):
"""
Predict values using the model

Parameters
----------
X : {array-like, sparse matrix} of shape
```

```
[n_samples, n_features]

Returns
-------
C : numpy array of shape [n_samples, n_outputs]
Predicted values.
"""
if (self._genelm_regressor is None):
raise ValueError("SimpleELMRegressor not fitted")

return self._genelm_regressor.predict(X)



class ELMClassifier(ELMRegressor):

def __init__(self, n_hidden=20, alpha=0.5, rbf_width=1.0,
activation_func='tanh', activation_args=None,
user_components=None, regressor=None,
binarizer=LabelBinarizer(-1, 1),
random_state=None):

super(ELMClassifier, self).__init__(n_hidden=n_hidden,
alpha=alpha,
random_state=random_state,
activation_func=activation_func,
activation_args=activation_args,
user_components=user_components,
rbf_width=rbf_width,
regressor=regressor)
```

```
self.classes_ = None
self.binarizer = binarizer

def decision_function(self, X):
"""
This function return the decision function values
related to each class on an array of test vectors X.

Parameters
----------
X : array-like of shape [n_samples, n_features]

Returns
-------
C : array of shape [n_samples, n_classes] or [n_samples,]
Decision function values related to each class, per sample.
In the two-class case, the shape is [n_samples,]
"""
return super(ELMClassifier, self).predict(X)

def fit(self, X, y):
"""
Fit the model using X, y as training data.

Parameters
----------
X : {array-like, sparse matrix} of shape
[n_samples, n_features]
```

```
Training vectors, where n_samples is the number
samples and n_features is the number of features.

y : array-like of shape [n_samples, n_outputs]
Target values (class labels in classification, real
 numbers in regression)

Returns
-------
self : object

Returns an instance of self.
"""
self.classes_ = np.unique(y)

y_bin = self.binarizer.fit_transform(y)

super(ELMClassifier, self).fit(X, y_bin)

return self

def predict(self, X):
"""
Predict values using the model

Parameters
----------
X : {array-like, sparse matrix} of shape
[n_samples, n_features]
```

```
Returns
-------
C : numpy array of shape [n_samples, n_outputs]
Predicted values.
"""
raw_pred = self.decision_function(X)
class_pred = self.binarizer.inverse_transform(raw_pred)

return class_predictions

def score(self, X, y):
"""Force use of accuracy score since we don't inherit
from ClassifierMixin"""

from sklearn.metrics import accuracy_score
return accuracy_score(y, self.predict(X))
```

```
#elm_notebook.py

from time import time
from sklearn.cluster import k_means

from elm import ELMRegressor, GenELMClassifier,
GenELMRegressor
from elm import ELMClassifier
from random_layer import RBFRandomLayer, GRBFRandomLayer
from random_layer import RandomLayer, MLPRandomLayer
import numpy as np
import matplotlib.pyplot as plt
```

```
from numpy import mean,std


def make_toy():
x = np.arange(0.25,20,0.1)
y = x*np.cos(x)+0.5*np.sqrt(x)*np.random.randn(x.shape[0])
x = x.reshape(-1,1)
y = y.reshape(-1,1)
return x, y


def res_dist(x, y, e, n_runs=100, random_state=None):
x_train,x_test,y_train= train_test_split
(x, y, test_size=0.4)

test_res = []
train_res = []
start_time = time()

for i in range(n_runs):
e.fit(x_train, y_train)
train_res.append(e.score(x_train, y_train))
test_res.append(e.score(x_test, y_test))
if (i%(n_runs/10) == 0): print ("%d"%i),

print ("\nTime: %.3f secs" % (time() - start_time))

print ("Test Min: %.3f Mean: %.3f Max: %.3f SD: %.3f"
% (min(test_res), mean(test_res),
max(test_res), std(test_res)))
print ("Train Min: %.3f Mean: %.3f Max: %.3f SD: %.3f"
 % (min(train_res), mean(train_res), max(train_res
 ), std(train_res)))
print()
return (train_res, test_res)

# <codecell>

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_diabetes,
make_regression
from sklearn.datasets import load_iris, load_digits

stdsc = StandardScaler()

iris = load_iris()
irx, iry = stdsc.fit_transform(iris.data), iris.target
irx_train, irx_test, iry_train, iry_test =
train_test_split(irx, iry, test_size=0.2)

digits = load_digits()
dgx, dgy = stdsc.fit_transform(digits.data/16.0),
 digits.target
dgx_train, dgx_test, dgy_train, dgy_test =
train_test_split(dgx, dgy, test_size=0.2)

diabetes = load_diabetes()
```

```
dbx, dby = stdsc.fit_transform(diabetes.data),
 diabetes.target
dbx_train, dbx_test, dby_train, dby_test =
train_test_split(dbx, dby, test_size=0.2)

mrx, mry = make_regression(n_samples=2000,
n_targets=4)
mrx_train, mrx_test, mry_train, mry_test =
train_test_split(mrx, mry, test_size=0.2)

xtoy, ytoy = make_toy()
xtoy, ytoy = stdsc.fit_transform(xtoy),
stdsc.fit_transform(ytoy)
xtoy_train, xtoy_test, ytoy_train, ytoy_test =
 train_test_split(xtoy, ytoy, test_size=0.2)
plt.plot(xtoy, ytoy)

# RBFRandomLayer tests
for af in RandomLayer.activation_func_names():
print (af),
elmc = ELMClassifier(activation_func=af)
tr,ts = res_dist(irx, iry, elmc, n_runs=200,
random_state=0)

for af in RandomLayer.activation_func_names():
print (af)
elmc = ELMClassifier(activation_func=af,
random_state=0)
tr,ts = res_dist(dgx, dgy, elmc, n_runs=100,
random_state=0)

elmc = ELMClassifier(n_hidden=500,
activation='multiquadric')
tr,ts = res_dist(dgx, dgy, elmc, n_runs=100,
 random_state=0)
plt.scatter(tr, ts, alpha=0.1, marker='D', c='r')

elmr = ELMRegressor(activation_func='gaussian',
alpha=0.0)
elmr.fit(xtoy_train, ytoy_train)
print( elmr.score(xtoy_train, ytoy_train),
elmr.score(xtoy_test, ytoy_test))
plt.plot(xtoy, ytoy, xtoy, elmr.predict(xtoy))

from sklearn import pipeline
from sklearn.linear_model import LinearRegression
elmr = pipeline.Pipeline([('rhl',
RandomLayer(activation_func='multiquadric')),
('lr', LinearRegression(fit_intercept=False))])
elmr.fit(xtoy_train, ytoy_train)
print (elmr.score(xtoy_train, ytoy_train),
elmr.score(xtoy_test, ytoy_test))
plt.plot(xtoy, ytoy, xtoy, elmr.predict(xtoy))

rhl = RandomLayer(n_hidden=200, alpha=1.0)
elmr = GenELMRegressor(hidden_layer=rhl)
tr, ts = res_dist(mrx, mry, elmr, n_runs=200, )
plt.scatter(tr, ts, alpha=0.1, marker='D', c='r')
```

```
rhl = RBFRandomLayer(n_hidden=15, rbf_width=0.8)
elmr = GenELMRegressor(hidden_layer=rhl)
elmr.fit(xtoy_train, ytoy_train)
print( elmr.score(xtoy_train, ytoy_train),
elmr.score(xtoy_test, ytoy_test))
plt.plot(xtoy, ytoy, xtoy, elmr.predict(xtoy))

nh = 15
(ctrs, _, _) = k_means(xtoy_train, nh)
unit_rs = np.ones(nh)

#rhl = RBFRandomLayer(n_hidden=nh,
activation_func='inv_multiquadric')
#rhl = RBFRandomLayer(n_hidden=nh,
 centers=ctrs, radii=unit_rs)
rhl = GRBFRandomLayer(n_hidden=nh,
grbf_lambda=.0001, centers=ctrs)
elmr = GenELMRegressor(hidden_layer=rhl)
elmr.fit(xtoy_train, ytoy_train)
print (elmr.score(xtoy_train, ytoy_train),
 elmr.score(xtoy_test, ytoy_test))
plt.plot(xtoy, ytoy, xtoy, elmr.predict(xtoy))


rbf_rhl = RBFRandomLayer(n_hidden=100,
 random_state=0, rbf_width=0.01)
elmc_rbf = GenELMClassifier(hidden_layer=rbf_rhl)
elmc_rbf.fit(dgx_train, dgy_train)
```

```
print (elmc_rbf.score(dgx_train, dgy_train),
 elmc_rbf.score(dgx_test, dgy_test))

def powtanh_xfer(activations, power=1.0):
return pow(np.tanh(activations), power)

tanh_rhl = MLPRandomLayer(n_hidden=100,
activation_func=powtanh_xfer,
activation_args={'power':3.0})
elmc_tanh = GenELMClassifier(hidden_layer=tanh_rhl)
elmc_tanh.fit(dgx_train, dgy_train)
print(elmc_tanh.score(dgx_train, dgy_train),
 elmc_tanh.score(dgx_test, dgy_test))

rbf_rhl = RBFRandomLayer(n_hidden=100, rbf_width=0.01)
tr, ts = res_dist(dgx, dgy,
GenELMClassifier(hidden_layer=rbf_rhl),
 n_runs=100, random_state=0)

plt.hist(ts), plt.hist(tr)
print()

from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor
tr, ts = res_dist(dbx, dby,
RandomForestRegressor(n_estimators=15),
 n_runs=100, random_state=0)
plt.hist(tr), plt.hist(ts)
print()
```

```
rhl = RBFRandomLayer(n_hidden=15, rbf_width=0.1)
tr,ts = res_dist(dbx, dby,
GenELMRegressor(rhl), n_runs=100,
 random_state=0)
plt.hist(tr), plt.hist(ts)
print()



elmc = ELMClassifier(n_hidden=1000,
activation_func='gaussian', alpha=0.0, random_state=0)
elmc.fit(dgx_train, dgy_train)
print (elmc.score(dgx_train, dgy_train),
elmc.score(dgx_test, dgy_test))

elmc = ELMClassifier(n_hidden=500,
activation_func='hardlim', alpha=1.0, random_state=0)
elmc.fit(dgx_train, dgy_train)
print (elmc.score(dgx_train, dgy_train),
elmc.score(dgx_test, dgy_test))

elmr = ELMRegressor(random_state=0)
elmr.fit(xtoy_train, ytoy_train)
print (elmr.score(xtoy_train, ytoy_train),
elmr.score(xtoy_test, ytoy_test))
plt.plot(xtoy, ytoy, xtoy, elmr.predict(xtoy))

elmr = ELMRegressor(activation_func='inv_tribas')
elmr.fit(xtoy_train, ytoy_train)
```

```
print (elmr.score(xtoy_train, ytoy_train),
elmr.score(xtoy_test, ytoy_test))
plt.plot(xtoy, ytoy, xtoy, elmr.predict(xtoy))
```

```
#random_layer.py

from abc import ABCMeta, abstractmethod

from math import sqrt

import numpy as np
import scipy.sparse as sp
from scipy.spatial.distance import cdist
from scipy.spatial import pdist, squareform

from sklearn.metrics import pairwise_distances
from sklearn.utils import check_random_state,
check_array
from sklearn.utils.extmath import safe_sparse_dot
from sklearn.base import BaseEstimator, TransformerMixin

__all__ = ['RandomLayer',
'MLPRandomLayer',
'RBFRandomLayer',
'GRBFRandomLayer',
]


class BaseRandomLayer(BaseEstimator, TransformerMixin):
"""Abstract Base Class for random  layers"""
```

```
__metaclass__ = ABCMeta

_internal_activation_funcs = dict()

@classmethod
def activation_func_names(cls):
"""Get list of internal activation function names"""
return cls._internal_activation_funcs.keys()

# take n_hidden and random_state, init components_ and
# input_activations_
def __init__(self, n_hidden=20, random_state=0,
activation_func=None,activation_args=None):

self.n_hidden = n_hidden
self.random_state = random_state
self.activation_func = activation_func
self.activation_args = activation_args

self.components_ = dict()
self.input_activations_ = None

# keyword args for internally defined funcs
self._extra_args = dict()

@abstractmethod
def _generate_components(self, X):
"""Generate components of hidden layer given X"""
```

```
@abstractmethod
def _compute_input_activations(self, X):
"""Compute input activations given X"""

# compute input activations and pass them
# through the hidden layer transfer functions
# to compute the transform
def _compute_hidden_activations(self, X):
"""Compute hidden activations given X"""

self._compute_input_activations(X)

acts = self.input_activations_

if (callable(self.activation_func)):
args_dict = self.activation_args if
(self.activation_args)
else {}
X_new = self.activation_func(acts, **args_dict)
else:
func_name = self.activation_func
func = self._internal_activation_funcs[func_name]

X_new = func(acts, **self._extra_args)

return X_new

# perform fit by generating random components based
# on the input array
```

```
def fit(self, X, y=None):
X = check_array(X)

self._generate_components(X)

return self


# perform transformation by compute_hidden_activations
# (which will normally call compute_input_activations)
def transform(self, X, y=None):
X = check_array(X)

if (self.components_ is None):
raise ValueError('No components initialized')

return self._compute_hidden_activations(X)



class RandomLayer(BaseRandomLayer):
# triangular activation function
_tribas = (lambda x: np.clip(1.0 - np.fabs(x), 0.0, 1.0))

# inverse triangular activation function
_inv_tribas = (lambda x: np.clip(np.fabs(x), 0.0, 1.0))

# sigmoid activation function
_sigmoid = (lambda x: 1.0/(1.0 + np.exp(-x)))

# hard limit activation function
```

```
_hardlim = (lambda x: np.array(x > 0.0, dtype=float))

_softlim = (lambda x: np.clip(x, 0.0, 1.0))

# gaussian RBF
_gaussian = (lambda x: np.exp(-pow(x, 2.0)))

# multiquadric RBF
_multiquadric = (lambda x:
np.sqrt(1.0 + pow(x, 2.0)))

# inverse multiquadric RBF
_inv_multiquadric = (lambda x:
1.0/(np.sqrt(1.0 + pow(x, 2.0))))

# internal activation function table
_internal_activation_funcs = {'sine': np.sin,
'tanh': np.tanh,
'tribas': _tribas,
'inv_tribas': _inv_tribas,
'sigmoid': _sigmoid,
'softlim': _softlim,
'hardlim': _hardlim,
'gaussian': _gaussian,
'multiquadric': _multiquadric,
'inv_multiquadric': _inv_multiquadric,
}

def __init__(self, n_hidden=20, alpha=0.5,
activation_func='tanh', activation_args=None,
user_components=None, rbf_width=1.0):

super(RandomLayer, self).__init__(n_hidden=n_hidden,
random_state=random_state,
activation_func=activation_func,
activation_args=activation_args)

if (isinstance(self.activation_func, str)):
func_names = self._internal_activation_funcs.keys()
if (self.activation_func not in func_names):
msg = "unknown activation function '%s'" % self.act
raise ValueError(msg)

self.alpha = alpha
self.rbf_width = rbf_width
self.user_components = user_components

self._use_mlp_input = (self.alpha != 0.0)
self._use_rbf_input = (self.alpha != 1.0)

def _get_user_components(self, key):
"""Look for given user component"""
try:
return self.user_components[key]
except (TypeError, KeyError):
return None

def _compute_radii(self):
```

```
"""Generate RBF radii"""

# use supplied radii if present
radii = self._get_user_components('radii')

# compute radii
if (radii is None):
centers = self.components_['centers']

n_centers = centers.shape[0]
max_dist = np.max(pairwise_distances(centers))
radii = np.ones(n_centers) *
max_dist/sqrt(2.0 * n_centers)

self.components_['radii'] = radii

def _compute_centers(self, X, sparse, rs):
"""Generate RBF centers"""

# use supplied centers if present
centers = self._get_user_components('centers')

# use points taken uniformly from the bounding
# hyperrectangle
if (centers is None):
n_features = X.shape[1]

if (sparse):
fxr = xrange(n_features)
```

```
cols = [X.getcol(i) for i in fxr]

min_dtype = X.dtype.type(1.0e10)
sp_min = lambda col: np.minimum(min_dtype,
np.min(col.data))
min_Xs = np.array(map(sp_min, cols))

max_dtype = X.dtype.type(-1.0e10)
sp_max = lambda col: np.maximum(max_dtype,
 np.max(col.data))
max_Xs = np.array(map(sp_max, cols))
else:
min_Xs = X.min(axis=0)
max_Xs = X.max(axis=0)

spans = max_Xs - min_Xs
ctrs_size = (self.n_hidden, n_features)
centers = min_Xs + spans * rs.uniform(0.0,
1.0, ctrs_size)

self.components_['centers'] = centers

def _compute_biases(self, rs):
"""Generate MLP biases"""

# use supplied biases if present
biases = self._get_user_components('biases')
if (biases is None):
b_size = self.n_hidden
```

```
biases = rs.normal(size=b_size)

self.components_['biases'] = biases

def _compute_weights(self, X, rs):
"""Generate MLP weights"""

# use supplied weights if present
weights = self._get_user_components('weights')
if (weights is None):
n_features = X.shape[1]
hw_size = (n_features, self.n_hidden)
weights = rs.normal(size=hw_size)

self.components_['weights'] = weights

def _generate_components(self, X):
"""Generate components of hidden layer given X"""

rs = check_random_state(self.random_state)
if (self._use_mlp_input):
self._compute_biases(rs)
self._compute_weights(X, rs)

if (self._use_rbf_input):
self._compute_centers(X, sp.issparse(X), rs)
self._compute_radii()

def _compute_input_activations(self, X):
```

```
"""Compute input activations given X"""

n_samples = X.shape[0]

mlp_acts = np.zeros((n_samples, self.n_hidden))
if (self._use_mlp_input):
b = self.components_['biases']
w = self.components_['weights']
mlp_acts = self.alpha * (safe_sparse_dot(X, w) + b)

rbf_acts = np.zeros((n_samples, self.n_hidden))
if (self._use_rbf_input):
radii = self.components_['radii']
centers = self.components_['centers']
scale = self.rbf_width * (1.0 - self.alpha)
rbf_acts = scale * cdist(X, centers)/radii

self.input_activations_ = mlp_acts + rbf_acts



class MLPRandomLayer(RandomLayer):
"""Wrapper for RandomLayer with alpha
to 1.0 for MLP activations only"""

def __init__(self, n_hidden=20, random_state=None,
activation_func='tanh', activation_args=None,
weights=None, biases=None):

user_components = {'weights': weights, 'biases': biases}
```

```
super(MLPRandomLayer, self).__init__(n_hidden=n_hidden,
random_state=random_state,
activation_func=activation_func,
activation_args=activation_args,
user_components=user_components,
alpha=1.0)



class RBFRandomLayer(RandomLayer):
"""Wrapper for RandomLayer with alpha
to 0.0 for RBF activations only"""

def __init__(self, n_hidden=20, random_state=None,
activation_func='gaussian', activation_args=None,
centers=None, radii=None, rbf_width=1.0):

user_components = {'centers': centers,
 'radii': radii}
super(RBFRandomLayer, self).__init__
(n_hidden=n_hidden,
random_state=random_state,
activation_func=activation_func,
activation_args=activation_args,
user_components=user_components,
rbf_width=rbf_width,
alpha=0.0)



class GRBFRandomLayer(RBFRandomLayer):
```

```
_grbf = (lambda acts, taus:
np.exp(np.exp(-pow(acts, taus))))

_internal_activation_funcs = {'grbf': _grbf}

def __init__(self, n_hidden=20, grbf_lambda=0.001,
centers=None, radii=None, random_state=None):

super(GRBFRandomLayer, self).__init__
(n_hidden=n_hidden,
activation_func='grbf',
centers=centers, radii=radii,
random_state=random_state)

self.grbf_lambda = grbf_lambda
self.dN_vals = None
self.dF_vals = None
self.tau_vals = None

def _compute_centers(self, X, sparse, rs):
"""Generate centers, then compute tau, dF
and dN vals"""

super(GRBFRandomLayer, self)._compute_centers
(X, sparse, rs)

centers = self.components_['centers']
sorted_distances = np.sort(squareform(pdist(centers)))
```

```
self.dF_vals = sorted_distances[:, -1]
self.dN_vals = sorted_distances[:, 1]/100.0
#self.dN_vals = 0.0002 * np.ones(self.dF_vals.shape)

tauNum = np.log(np.log(self.grbf_lambda) /
np.log(1.0 - self.grbf_lambda))

tauDenom = np.log(self.dF_vals/self.dN_vals)

self.tau_vals = tauNum/tauDenom

self._extra_args['taus'] = self.tau_vals

# get radii according to ref [1]
def _compute_radii(self):
"""Generate radii"""

denom = pow(-np.log(self.grbf_lambda),
 1.0/self.tau_vals)
self.components_['radii'] = self.dF_vals/denom
```

```
#random_layer.py
from abc import ABCMeta, abstractmethod

from math import sqrt

import numpy as np
import scipy.sparse as sp
from scipy.spatial.distance import cdist,
pdist, squareform
```

```
from sklearn.metrics import pairwise_distances
from sklearn.utils import check_random_state,
 check_array
from sklearn.utils.extmath import safe_sparse_dot
from sklearn.base import BaseEstimator,
TransformerMixin

__all__ = ['RandomLayer',
'MLPRandomLayer',
'RBFRandomLayer',
'GRBFRandomLayer',
]


class BaseRandomLayer(BaseEstimator,
 TransformerMixin):
"""Abstract Base Class for random  layers"""
__metaclass__ = ABCMeta

_internal_activation_funcs = dict()

@classmethod
def activation_func_names(cls):
"""Get list of internal activation function names"""
return cls._internal_activation_funcs.keys()

# take n_hidden and random_state, init components_ and
# input_activations_
```

```
def __init__(self, n_hidden=20, random_state=0,
activation_func=None,
activation_args=None):

self.n_hidden = n_hidden
self.random_state = random_state
self.activation_func = activation_func
self.activation_args = activation_args

self.components_ = dict()
self.input_activations_ = None

# keyword args for internally defined funcs
self._extra_args = dict()

@abstractmethod
def _generate_components(self, X):
"""Generate components of hidden layer given X"""

@abstractmethod
def _compute_input_activations(self, X):
"""Compute input activations given X"""

# compute input activations and pass them
# through the hidden layer transfer functions
# to compute the transform
def _compute_hidden_activations(self, X):
"""Compute hidden activations given X"""
```

```
self._compute_input_activations(X)

acts = self.input_activations_

if (callable(self.activation_func)):
args_dict = self.activation_args if
(self.activation_args) else {}
X_new = self.activation_func(acts, **args_dict)
else:
func_name = self.activation_func
func = self._internal_activation_funcs[func_name]

X_new = func(acts, **self._extra_args)

return X_new

# perform fit by generating random components based
# on the input array
def fit(self, X, y=None):
X = check_array(X)

self._generate_components(X)

return self

def transform(self, X, y=None):
X = check_array(X)

if (self.components_ is None):
```

```
raise ValueError('No components initialized')

return self._compute_hidden_activations(X)



class RandomLayer(BaseRandomLayer):
# triangular activation function
_tribas = (lambda x: np.clip(1.0 -
np.fabs(x), 0.0, 1.0))

# inverse triangular activation function
_inv_tribas = (lambda x: np.clip
(np.fabs(x), 0.0, 1.0))

# sigmoid activation function
_sigmoid = (lambda x: 1.0/(1.0 + np.exp(-x)))

# hard limit activation function
_hardlim = (lambda x: np.array(x > 0.0, dtype=float))

_softlim = (lambda x: np.clip(x, 0.0, 1.0))

# gaussian RBF
_gaussian = (lambda x: np.exp(-pow(x, 2.0)))

# multiquadric RBF
_multiquadric = (lambda x:
np.sqrt(1.0 + pow(x, 2.0)))
```

```
# inverse multiquadric RBF
_inv_multiquadric = (lambda x:
1.0/(np.sqrt(1.0 + pow(x, 2.0))))

# internal activation function table
_internal_activation_funcs = {'sine': np.sin,
'tanh': np.tanh,
'tribas': _tribas,
'inv_tribas': _inv_tribas,
'sigmoid': _sigmoid,
'softlim': _softlim,
'hardlim': _hardlim,
'gaussian': _gaussian,
'multiquadric': _multiquadric,
'inv_multiquadric': _inv_multiquadric,
}

def __init__(self, n_hidden=20, alpha=0.5,
activation_func='tanh', activation_args=None,
user_components=None, rbf_width=1.0):

super(RandomLayer, self).__init__(n_hidden=n_hidden,
random_state=random_state,
activation_func=activation_func,
activation_args=activation_args)

if (isinstance(self.activation_func, str)):
func_names = self._internal_activation_funcs.keys()
if (self.activation_func not in func_names):
```

```
msg = "unknown activation function '%s'"
raise ValueError(msg)

self.alpha = alpha
self.rbf_width = rbf_width
self.user_components = user_components

self._use_mlp_input = (self.alpha != 0.0)
self._use_rbf_input = (self.alpha != 1.0)

def _get_user_components(self, key):
"""Look for given user component"""
try:
return self.user_components[key]
except (TypeError, KeyError):
return None

def _compute_radii(self):
"""Generate RBF radii"""

# use supplied radii if present
radii = self._get_user_components('radii')

# compute radii
if (radii is None):
centers = self.components_['centers']

n_centers = centers.shape[0]
max_dist = np.max(pairwise_distances(centers))
```

```
radii = np.ones(n_centers) *
max_dist/sqrt(2.0 * n_centers)

self.components_['radii'] = radii

def _compute_centers(self, X, sparse, rs):
"""Generate RBF centers"""

# use supplied centers if present
centers = self._get_user_components('centers')

# use points taken uniformly from the bounding
# hyperrectangle
if (centers is None):
n_features = X.shape[1]

if (sparse):
fxr = xrange(n_features)
cols = [X.getcol(i) for i in fxr]

min_dtype = X.dtype.type(1.0e10)
sp_min = lambda col: np.minimum(min_dtype,
np.min(col.data))
min_Xs = np.array(map(sp_min, cols))

max_dtype = X.dtype.type(-1.0e10)
sp_max = lambda col: np.maximum(max_dtype,
 np.max(col.data))
max_Xs = np.array(map(sp_max, cols))
```

```
else:
min_Xs = X.min(axis=0)
max_Xs = X.max(axis=0)

spans = max_Xs - min_Xs
ctrs_size = (self.n_hidden, n_features)
centers = min_Xs + spans * rs.uniform(0.0, 1.0,
ctrs_size)

self.components_['centers'] = centers

def _compute_biases(self, rs):
"""Generate MLP biases"""

# use supplied biases if present
biases = self._get_user_components('biases')
if (biases is None):
b_size = self.n_hidden
biases = rs.normal(size=b_size)

self.components_['biases'] = biases

def _compute_weights(self, X, rs):
"""Generate MLP weights"""

# use supplied weights if present
weights = self._get_user_components('weights')
if (weights is None):
n_features = X.shape[1]
```

```
hw_size = (n_features, self.n_hidden)
weights = rs.normal(size=hw_size)

self.components_['weights'] = weights

def _generate_components(self, X):
"""Generate components of hidden layer given X"""

rs = check_random_state(self.random_state)
if (self._use_mlp_input):
self._compute_biases(rs)
self._compute_weights(X, rs)

if (self._use_rbf_input):
self._compute_centers(X, sp.issparse(X), rs)
self._compute_radii()

def _compute_input_activations(self, X):
"""Compute input activations given X"""

n_samples = X.shape[0]

mlp_acts = np.zeros((n_samples, self.n_hidden))
if (self._use_mlp_input):
b = self.components_['biases']
w = self.components_['weights']
mlp_acts = self.alpha * (safe_sparse_dot(X, w) + b)

rbf_acts = np.zeros((n_samples, self.n_hidden))
```

```
if (self._use_rbf_input):
radii = self.components_['radii']
centers = self.components_['centers']
scale = self.rbf_width * (1.0 - self.alpha)
rbf_acts = scale * cdist(X, centers)/radii

self.input_activations_ = mlp_acts + rbf_acts



class MLPRandomLayer(RandomLayer):
"""Wrapper for RandomLayer with alpha
to 1.0 for MLP activations only"""

def __init__(self, n_hidden=20,
random_state=None,
activation_func='tanh', activation_args=None,
weights=None, biases=None):

user_components = {'weights': weights,
'biases': biases}
super(MLPRandomLayer, self).__init__
(n_hidden=n_hidden,
random_state=random_state,
activation_func=activation_func,
activation_args=activation_args,
user_components=user_components,
alpha=1.0)
```

```
class RBFRandomLayer(RandomLayer):
"""Wrapper for RandomLayer with alpha
to 0.0 for RBF activations only"""

def __init__(self, n_hidden=20, random_state=None,
activation_func='gaussian', activation_args=None,
centers=None, radii=None, rbf_width=1.0):

user_components = {'centers': centers,
 'radii': radii}
super(RBFRandomLayer, self).__init__
(n_hidden=n_hidden,
random_state=random_state,
activation_func=activation_func,
activation_args=activation_args,
user_components=user_components,
rbf_width=rbf_width,
alpha=0.0)


class GRBFRandomLayer(RBFRandomLayer):

_grbf = (lambda acts, taus:
 np.exp(np.exp(-pow(acts, taus))))

_internal_activation_funcs = {'grbf': _grbf}

def __init__(self, n_hidden=20,
grbf_lambda=0.001,
centers=None, radii=None, random_state=None):

super(GRBFRandomLayer, self).__init__
(n_hidden=n_hidden,
activation_func='grbf',
centers=centers, radii=radii,
random_state=random_state)

self.grbf_lambda = grbf_lambda
self.dN_vals = None
self.dF_vals = None
self.tau_vals = None

def _compute_centers(self, X, sparse, rs):
"""Generate centers,compute tau, dF and
dN vals"""

super(GRBFRandomLayer,self)._compute_centers
(X, sparse, rs)

centers = self.components_['centers']
sorted_distances = np.sort(squareform
(pdist(centers)))
self.dF_vals = sorted_distances[:, -1]
self.dN_vals = sorted_distances[:, 1]/100.0
#self.dN_vals = 0.0002 * np.ones(self.dF_vals.shape)

tauNum = np.log(np.log(self.grbf_lambda) /
np.log(1.0 - self.grbf_lambda))
```

```
tauDenom = np.log(self.dF_vals/self.dN_vals)

self.tau_vals = tauNum/tauDenom

self._extra_args['taus'] = self.tau_vals

# get radii according to ref [1]
def _compute_radii(self):
"""Generate radii"""

denom = pow(-np.log(self.grbf_lambda),
 1.0/self.tau_vals)
self.components_['radii'] = self.dF_vals/denom
```

```
#plot_elm_comparison.py
print __doc__

import numpy as np
import pylab as pl

from matplotlib.colors import ListedColormap
from sklearn.datasets import make_classification
from sklearn.datasets import make_moons, make_circles
from sklearn.preprocessing import StandardScaler
from sklearn.cross_validation import train_test_split
from sklearn.linear_model import LogisticRegression

from elm import GenELMClassifier
from random_layer import RBFRandomLayer,
MLPRandomLayer


def get_data_bounds(X):
h = .02  # step size in the mesh

x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5

xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
np.arange(y_min, y_max, h))

return (x_min, x_max, y_min, y_max, xx, yy)


def plot_data(ax, X_train, y_train, X_test, y_test
, xx, yy):
cm = ListedColormap(['#FF0000', '#0000FF'])
# Plot the training points
ax.scatter(X_train[:, 0], X_train[:, 1],
 c=y_train, cmap=cm)
# and testing points
ax.scatter(X_test[:, 0], X_test[:, 1],
c=y_test, alpha=0.6)
ax.set_xlim(xx.min(), xx.max())
ax.set_ylim(yy.min(), yy.max())
ax.set_xticks(())
ax.set_yticks(())
```

```
def plot_contour(ax, X_train, y_train, X_test, y_test
, xx, yy, Z):
cm = pl.cm.RdBu
cm_bright = ListedColormap(['#FF0000', '#0000FF'])

ax.contourf(xx, yy, Z, cmap=cm, alpha=.8)

# Plot also the training points
ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train,
cmap=cm_bright)
# and testing points
ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test,
 cmap=cm_bright, alpha=0.6)

ax.set_xlim(xx.min(), xx.max())
ax.set_ylim(yy.min(), yy.max())
ax.set_xticks(())
ax.set_yticks(())

ax.set_title(name)
ax.text(xx.max() - 0.3, yy.min() + 0.3,
('%.2f' % score)
.lstrip('0'),
size=13, horizontalalignment='right')


def make_datasets():
```

```
return [make_moons(n_samples=200, noise=0.3),
make_circles(n_samples=200, noise=0.2, factor=0.5),
make_linearly_separable()]



def make_classifiers():


names = ["ELM(10,tanh)", "ELM(10,tanh,LR)",
 "ELM(10,sinsq)",
"ELM(10,tribas)", "ELM(hardlim)",
"ELM(20,rbf(0.1))"]

nh = 10

# pass user defined transfer func
sinsq = (lambda x: np.power(np.sin(x), 2.0))
srhl_sinsq = MLPRandomLayer(n_hidden=nh,
activation_func=sinsq)

# use internal transfer funcs
srhl_tanh = MLPRandomLayer(n_hidden=nh,
activation_func='tanh')

srhl_tribas = MLPRandomLayer(n_hidden=nh,
activation_func='tribas')

srhl_hardlim = MLPRandomLayer(n_hidden=nh,
 activation_func='hardlim')
```

```
# use gaussian RBF
srhl_rbf = RBFRandomLayer(n_hidden=nh*2,
rbf_width=0.1)

log_reg = LogisticRegression()

classifiers = [GenELMClassifier(hidden_layer=srhl_tanh),
GenELMClassifier(hidden_layer=srhl_tanh,
 regressor=log_reg),
GenELMClassifier(hidden_layer=srhl_sinsq),
GenELMClassifier(hidden_layer=srhl_tribas),
GenELMClassifier(hidden_layer=srhl_hardlim),
GenELMClassifier(hidden_layer=srhl_rbf)]

return names, classifiers



def make_linearly_separable():
X, y = make_classification(n_samples=200,
n_features=2, n_redundant=0,
n_informative=2, random_state=1,
n_clusters_per_class=1)
rng = np.random.RandomState(2)
X += 2 * rng.uniform(size=X.shape)
return (X, y)

###############################################################

datasets = make_datasets()
```

```
names, classifiers = make_classifiers()

i = 1
figure = pl.figure(figsize=(18, 9))

# iterate over datasets
for ds in datasets:
# preprocess dataset, split into training and test part
X, y = ds
X = StandardScaler().fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=.4,
random_state=0)

x_min, x_max, y_min, y_max, xx, yy = get_data_bounds(X)

# plot dataset first
ax = pl.subplot(len(datasets), len(classifiers) + 1, i)
plot_data(ax, X_train, y_train, X_test, y_test, xx, yy)

i += 1

# iterate over classifiers
for name, clf in zip(names, classifiers):
ax = pl.subplot(len(datasets), len(classifiers) + 1, i)
clf.fit(X_train, y_train)
score = clf.score(X_test, y_test)

# Plot the decision boundary. For that, we will asign
# point in the mesh [x_min, m_max]x[y_min, y_max].
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)

plot_contour(ax, X_train, y_train, X_test, y_test, xx,
yy, Z)

i += 1

figure.subplots_adjust(left=.02, right=.98)
pl.show()
```

Vita Auctoris

NAME	Daisy
PLACE OF BIRTH	Punjab, India
YEAR OF BIRTH	1995
EDUCATION	2012-2016
	Punjabi University, Patiala, Punjab, India ,
	Bachelor of Electronics and Communication Engineering.
	2019-2020,
	University of Windsor, Windsor, Ontario, Canada ,
	Master of Applied Science, Electrical and Computer Engineering