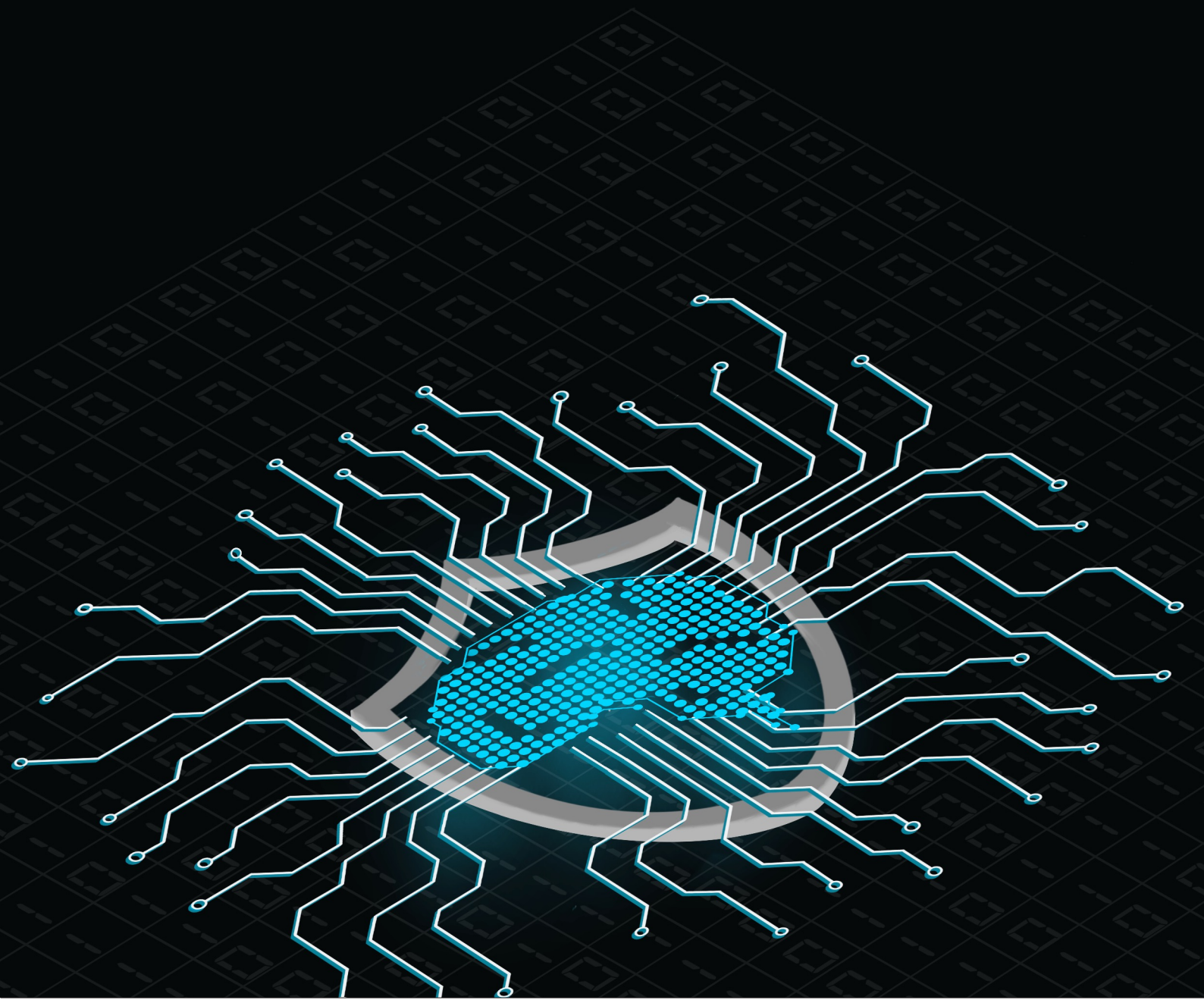
Pre-Silicon Power Leakage Assessment Framework using Generative Adversarial Networks

Mudit Saxena



TU Delft Delft University of Technology

Challenge the future

To my mother,
who sacrificed her career to build mine.

Pre-Silicon Power Leakage Assessment Framework using Generative Adversarial Networks

by

Mudit Saxena

in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Engineering

at the Delft University of Technology,
to be defended publicly on Monday August 23, 2021 at 01:00PM.

Student number:	5008409	
Thesis number:	Q&CE-CE-MS-2021-05	
Supervisor:	dr. ir. M. Taouil	
Thesis committee:	Prof. dr. ir. S. Hamdioui,	TU Delft
	dr. ir. M. Taouil,	TU Delft
	dr. ir. A. Bossche	TU Delft

TU Delft Delft University of Technology

ACKNOWLEDGEMENT

This thesis would not have been possible without the support of my research team at TU Delft. I would like to thank Ir. Abdullah Aljuffri for his constant support throughout the past year. I would also like to thank Prof. Dr. Ir. Said Hamdioui and Dr. Ir. Mottaqiallah Taouil for their invaluable inputs and for guiding me through these trying times, both academically and professionally. Thanks also to Dr. Cezar Rodolfo Wedig Reinbrecht for all the guidance, ideas and discussions.

A special thanks to TU Delft's admissions committee, for believing in my abilities and for granting me both Holland and TU Delft Excellence scholarships. I am truly indebted to TU Delft for the knowledge I have gained over the past two years.

Words can not express my gratitude to my parents, Reeta and Munesh, my sister Riya and brother-in-law Ankit for their never-ending love and support. I am forever thankful for my supportive, caring and forbearing family.

I would feel amiss if I were to omit acknowledging the emotional support of my dear friends, Kriti and Ishita. Thank you for being a wonderful part of this journey.

Mudit Saxena
Delft, August 2021

CONTENTS

List of Figures

List of Tables

ABBREVIATIONS

ACGAN	Auxiliary Classifier Generative Adversarial Network
AE	Autoencoder
AES	Advanced Encryption Standard
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
AR	Autoregressive
BCE	Binary Cross Entropy
CCGAN	Continuous Conditional Generative Adversarial Networks
CGAN	Conditional Generative Adversarial Networks
CNN	Convolutional Neural Network
CPA	Correlation Power Analysis
DCGAN	Deep Convolutional Generative Adversarial Network
DES	Data Encryption Standard
DL	Deep Learning
DNN	Dense Neural Network
DPA	Differential Power Analysis
DTW	Dynamic Time Warping
EBM	Energy Based Models
EM	Electromagnetic
GAN	Generative Adversarial Network
GRU	Gated Recurrent Unit
IC	Integrated Circuit
LSGAN	Least Squares Generative Adversarial Network
LSTM	Long short-term memory
ML	Machine Learning
MLP	Multi Layer Perceptron
MSE	Mean Squared Error
NF	Normalized Flow
NLP	Natural language processing
ONNX	Open Neural Network Exchange
PCA	Principle component analysis
PGGAN	Progressive Growing of Generative Adversarial Networks
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SBOX	Substitution Box
SPA	Simple Power Analysis
TBA	Template Based Attack
TSMC	Taiwan Semiconductor Manufacturing Company
tSNE	t-distributed Stochastic Neighbor Embedding
VAE	Variational Autoencoder
WGAN	Wasserstein Generative Adversarial Network
WGAN-GP	Wasserstein Generative Adversarial Network - Gradient Penalty
3DES	Triple DES

ABSTRACT

As of 2021, the world economic forum deems cyber-security failures as one of the most potent threats to the world. According to a McAfee report, the cost of cyber crimes in 2020 reached nearly 1 trillion US dollars, which was around 50 percent more than what it was in 2018. Exacerbating the already mammoth financial implication of such a failure is the ever-growing diversity in cyber attacks. Side-channel analysis is one such attack type wherein the information leaked via the implementation of a cryptographic algorithm is leveraged to obtain secret data, rather than any weaknesses in the cryptographic algorithm itself. This leaked information, amongst others, can be in terms of power, EM radiation, or the time taken to perform a cryptographic operation. Countermeasures against such side-channel attacks aim at reducing the amount of information leaked via the side channels or reducing the correlation between the secret operations and the information leaked.

Manufacturing the chip is often a prerequisite for evaluating the efficacy of such countermeasures, which is a costly and time-consuming process. Thus, the security evaluation of a design has a substantial impact on the design cost and the time to market. In case the design does not meet minimum security requirements, it has to be redesigned and manufactured, increasing not only costs but also the design time considerably. Hence, there is a need for pre-silicon leakage assessment tools that can provide designers a sense of certainty about the security aspects of their design. However, the existing pre-silicon leakage assessment tools are either deemed unreliable or too slow to be used to perform power leakage assessment, which is the problem this thesis aims to ameliorate.

This thesis explores the use of generative adversarial networks (GANs) for generating synthetic power traces. Generative deep learning has been used in various domains like computer vision, audio, and even for medical data like ECG. GANs have been introduced in the context of side-channel attacks to enlarge the size of the profiling dataset for carrying out profiled side-channel attacks. In this work, we propose a robust methodology to condition and train GANs to generate power traces that can be used to carry out leakage assessment. This methodology can even be extended to support the design space exploration of countermeasures by providing reliable leakage assessment at design time. The generated power traces are not only indistinguishable but also as attackable as the real traces. The conditioning technique helps the GAN to generalize to various scenarios and the proposed framework provides a speed-up of around 140 times over traditional CAD methods to simulate power traces while maintaining their structure and accuracy.

1

INTRODUCTION

This chapter will introduce the contents and the structure of this thesis. Section 1.1 describes the motivation behind leakage assessment and the financial implication of not safeguarding against side channel analysis. Section 1.2 briefly introduces the current available CAD tools to perform pre-silicon leakage assessment and its shortcomings. Section 1.3 enumerates the research questions of this thesis and Section 1.4 describes the main contributions. Finally, Section 1.5 outlines how this thesis is structured.

1.1. MOTIVATION

With the ever growing human reliance on cyber-physical systems, comes the fear of cyber attacks. Cyber-physical systems, as the name suggests, is an agglomeration of the cyber and the physical domain. Unlike the physical domain, the cyber domain is not tangible but has nevertheless proved to be an integral part of human existence. The Internet of things paradigm is boosting this technology inclusion even further due to recent advancements in networking [5] and AI [6]. This ubiquity of cyber physical systems is one of the major contributing factors to why the world economic forum deem cyber attacks and IT breakdowns as potent risks in their global risk perception report [7]. World economic forum specifies that the cyber-security infrastructure is often rendered obsolete due to ever increasing sophistication of cybercrimes which can not only have large financial implications but can also lead to geopolitical tensions.

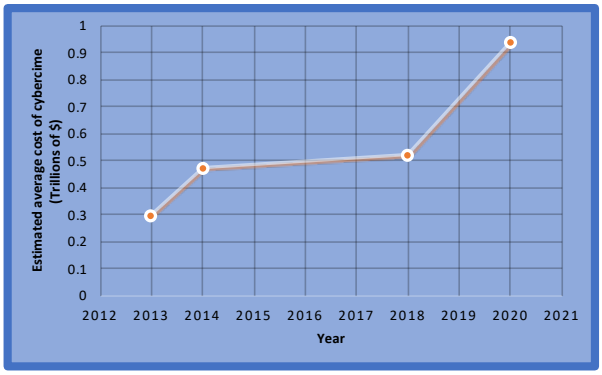


Figure 1.1: Cost of cybercrimes

According to a McAfee report [8], the cost of cyber-crimes in 2020 reached nearly 1 trillion US dollars, which is around 50 percent more than what it was in 2018. Figure 1.1 is inspired by [9] and highlights the massive increase in the cost of cybercrimes in the recent past. These figures almost sound unreal until we realize how intricately connected the entire digital ecosystem is. For example, in 2017 the WannaCry ransomware attack losses reached around 4 billion US dollars [10]. The reason the loss was so high is because it impacted not only individuals but also big organizations, since the attack leveraged vulnerabilities in the windows operating system affecting as many as 200,000 computers across the world [10]. This cyber-attack also impacted healthcare services [11] and even halted chip production at TSMC [12]. This chain of impact

on seemingly unrelated domains is what makes cyber-attacks such a potent risk. To make the situation even worse, these attacks can leverage vulnerabilities in the network, in the software and even in the hardware. A classification of some different Cyber-Physical attacks can be seen in Figure 1.2.

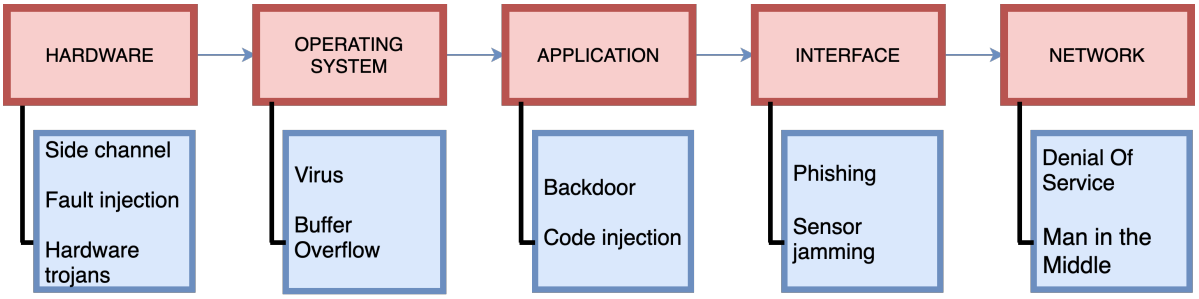


Figure 1.2: Flowchart of Cyber-Physical attacks

In this work, we are concerned with attacks in the hardware domain. A few popular hardware attacks are side channel analysis [13], fault injection [14] and hardware trojans [15]. Side channel analysis has gained a lot of interest in the recent past because unlike other hardware attacks, side channel analysis does not require modifications to the chip. Instead, they leverage the information leaked from the operation being executed on the chip. The side channel information source can be in the form of acoustics [16], EM radiation [17], power [18], time [19], photonic emission [20] and even through test infrastructure like scan chains [21].

Amongst the different kinds of side channels, power analysis has proven to be immensely popular due to three reasons. The first is that power attacks are simple to carry out since they just require simple tools and a low level of technical knowledge. Second is that they are much more robust to noise as compared to other side channels like acoustics. Finally, unlike other side channels like time, which is reliant on the response of the operation execution, power attacks allow the attacker to examine every stage of the secret operation. Hence the chance of identifying the weakest point of attack increases.

With the increase in the number and sophistication of power based side channel attacks, the demand for developing countermeasure against them has also increased. Developing countermeasure and evaluating their efficacy is a complex task. Designing for security from a hardware perspective is much more complex as compared to its software counterpart solely because in the event of discovering a vulnerability, just software patches will not alleviate the problem. Any security vulnerabilities in a design on a hardware level can only be addressed by changing the design and re-manufacturing, which can be immensely time consuming and expensive. This means that safeguarding hardware against such attacks requires holistic analysis. From a power-based side channel analysis point of view, this means that to test the security of a design a prototype must be manufactured which can then be used to collect power traces. The evaluation is done using security tools and equipment from companies like Rambus [22] and Riscure [23].

Manufacturing the prototype is a time consuming and expensive process. In a situation when the device fails to pass the security assessment, the design needs to be changed before it can be manufactured and tested again. This not only increases the R&D cost but also the design time. Hence, there is a requirement for a tool that can robustly perform pre-manufacturing (pre-silicon) leakage assessment.

1.2. STATE-OF-THE-ART LEAKAGE ASSESSMENT TOOLS

Leakage assessment tools can be classified as pre-silicon and post-silicon. Pre-silicon leakage assessment tools usually work under a white-box assumption, which means that there is full knowledge of the device under consideration. Post-silicon leakage assessment tools, on the other hand, usually work with a black-box/gray-box assumption. This means that there is very-minimal/some information available about the physical target [24]. Post-silicon tools are employed by software developers aiming to write side channel analysis resistent software. With a higher abstraction levels (as in black-box analysis), analyzing side channels are particularly difficult and require establishing high level leakage models. These models are limited to relatively simple architectures (most common tools work on ARM-Cortex M0) and work by correlating instructions with power consumption, with the objective of grouping instructions that consume power in a similar fashion. It is clear that there is an innate dependence on the micro-architecture details of a design and porting to different micro-architectures is a complex task. Two examples of such post-silicon tools are ELMO [25] and SAVRASCA [26].

For pre-silicon leakage assessment (which is the focus of this thesis) there are quite a few existing methods. A detailed comparison of such pre-silicon methods also exists [24]. The authors of this comparison call the state-of-the art pre-silicon tools as fragmented and non-reusable and highlight that they lack in execution speed. These techniques can be broadly divided into three categories, formal verification [27] tools, CAD tools [28] and functional simulation tools [29]. Formal verification mathematically analyzes the leakage, whereas the CAD tools try to simulate the power behaviour of the implementation. Two examples of formal verification are [27, 30]. Both [27] and [30], are targeted for performing leakage assessment for the masking countermeasure. Unfortunately, such solutions focus on analyzing randomness created by masks and is limited to just a particular form of countermeasure, which is masking. An example of a CAD-based solution is provided by Sadhukhan *et al.* [28] where they examine the leakage using both simulated and hypothetical power traces. Another example suggested by Nahiyan et al [31] is that they reduce the number of power traces needed to evaluate the leakage by improving the signal-to-noise ratio (SNR) algorithm. Unfortunately, creating simulated power traces is a time-consuming operation, and reducing the number of simulated traces cannot validate the protection against real attacks such as CPA, which typically require a large number of traces. Which is why in the industry it is common to see secure IPs being evaluated with a minimal of 10 million power traces [32]. Therefore, a CAD-based assessment solution can only be a viable solution if it can generate millions of power traces in a reasonable amount of time. Unfortunately, the CAD-based tools are known to be extremely slow and generating as many as 10 million power traces using them is not viable. For example, the solution proposed by Sadhukhan *et al.* [28] takes 5.47 seconds to generate a single trace. Generating as many as 10 million power traces using this technique will take around 633 days. This is, of course, not practical.

To obviate the need of using CAD-based tools for power trace generation, functional simulation tools like RTL-PSC [29] have been introduced. Wherein, the functional simulation of a design is used to carry out pre-silicon leakage assessment. This technique does not generate power trace as it is solely dependent on the switching activity file. This dependence on the switching activity makes this methodology not fit for modelling the design's technological behaviour(e.g., CMOS), which is useful for various countermeasures and is not visible using simply the switching activity. The switching activity also gives a rather ideal picture about the power consumption since it does not model any non-idealities like timing violation. Hence, there is need for a tool that can not only carry out pre-silicon leakage assessment in a reasonable amount of time, but can also accommodate the device's technological behaviour and non-idealities correctly so as to model its leakage robustly.

1.3. RESEARCH QUESTIONS

Introduced in 2014, Generative adversarial networks (GAN) are neural networks that aim at generating samples that resemble the data they have been trained on. This characteristic has made GAN a popular choice for a variety of data-based applications, such as images, audio, text, and videos. GANs may be used in a variety of ways to improve the quality of these applications. For example, GANs have been proposed as means to reduce the scarcity of training data in the field of bioinformatics, like in medical imaging. Here, GANs have been used to enhance the training data so as to train models that can accurately identify malignant tumors [33]. In the entertainment sector, GANs have seen a lot success, namely in enhancing the quality of images [34] and colorization [35] of monochrome images.

This research studies the ability to use GANs to generate reliable power traces using design simulation, that can be used for security analysis against side channel attacks. Our goal is to build a framework that simulates the switching activity of the target design and feeds this into a Generative Adversarial Network (GAN). The GAN then analyzes the switching activity and generates power traces. Finally, The generated power traces are then validated using well-known power attacks. This thesis tries to answer the following research questions:

1. Can GAN generated traces be used for security evaluation of an IC at design time?
2. Do different GAN architectures impact the quality of the generated traces?
3. What evaluation metrics can be used to evaluate the GAN generated traces?
4. How much speed-up can this GAN methodology provide over the standard CAD tools?
5. How well do GANs generalize to different implementations?

1.4. CONTRIBUTION

The objective of this thesis is to build a tool for pre-silicon power leakage assessment, which can quickly but robustly perform the evaluation. A Generative adversarial network (GAN) is used so as to synthetically generate power traces that can be used for this task. Different GAN architectures and label options are evaluated so as to determine the optimal architecture. The GAN is trained using 10000 power traces from FPGA evaluation of an AES-128 design. However, it can also be trained with power traces generated from an ASIC or even traces from CAD tools. To test that a trained GAN can generalize to different implementations of AES-128 (with and without countermeasures), validation and test labels using those designs are computed and are used to generate power traces. These generated power traces are then compared with the real power traces using various side channel analysis and signal processing metrics.

The main contributions of this thesis are:

- **A novel methodology for pre-silicon power leakage assessment:** this work, to the best of our knowledge, is the first application of generative deep learning for the task of pre-silicon power leakage assessment. GANs are trained on power traces corresponding to a specific implementation and their ability to generalize to other implementations is successfully demonstrated using various side channel analysis and signal processing metrics.
- **A detailed comparison of various GAN architectures and labels for power trace generation:** we analyzed and validated the impact of various GAN architectures, label options and hyperparameters on the task of power trace generation with the aim of finding the most optimal model for the task of pre-silicon power leakage assessment.
- **A detailed analysis of the impact of conditional GAN labels on power leakage:** we demonstrated that the choice of labels for conditional GANs have a grave impact on the amount of information leaked by the generated power traces. Various label options and conditioning techniques are explored so as to fine-tune GANs to generate power traces that are indistinguishable from the real traces, not only visually but also in terms of the information they contain.
- **A successful demonstration of the similarity between real and generated power traces:** using both side channel analysis and signal processing metrics, the GAN generated traces are holistically compared with the real traces to assess the ability of GAN to perform pre-silicon power leakage assessment. Inference time traces generated using finely tuned GAN architectures and label options are demonstrated to be indistinguishable from the real traces.

The contributions of this thesis have been condensed down into a research paper that has been attached as appendix E.

1.5. THESIS ORGANIZATION

The report is split into six chapters. Background information is covered in the first three chapters. The next two chapters provide a methodology for power trace generation using GANs as well as the associated results. Finally, the last chapter concludes this thesis along with some light on future research avenues. Each chapter contains the following information:

- Chapter 2: This chapter introduces the key concepts that form the foundation of modern day cryptographic algorithms and gives an overview of popular symmetric and asymmetric algorithms.
- Chapter 3 : This chapter gives an overview of different side channel attacks, with an emphasis on power based side channel attacks. Both profiled and non profiled attacks are explained in detail along with the countermeasures against them.
- Chapter 4: This chapter explains the deep learning concepts that are a prerequisite for this thesis. Beginning with the history of deep learning, the chapter slowly progresses towards modern-day generative deep learning models. It first covers fundamental neural networks and information on building, training and tuning the models using Keras and PyTorch. Then, generative deep learning is explained along with a comparison of state-of-the-art generative models.
- Chapter 5: This chapter describes the end to end methodology of generating power traces using GANs. Beginning with possible label options, the chapter covers different GAN architectures and the impact of

tuning the different hyperparameters associated with each architecture. The GANs are also compared amongst themselves to identify the most promising architecture and label option.

- Chapter 6: This chapter describes the results associated with the GANs introduced in Chapter 5 along with a detailed analysis of those results so as to propose a leakage assessment framework using VCD conditioning.
- Chapter 7: This chapter concludes this thesis with a brief overview of the entire text as well as future research avenues.

2

CRYPTOGRAPHIC ALGORITHMS

This chapter[1] *gives a brief introduction to cryptography. First, a few key concepts which form the foundation of modern day cryptographic algorithms are introduced in Section 2.1. Next, Section 2.2 explains the symmetric cryptographic algorithms along with a few examples of both block and stream ciphers. Similarly, Section 2.3 explains asymmetric cryptographic algorithms along with a few examples of the different kinds of one-way functions.*

2.1. HISTORY AND OVERVIEW

Cryptography is defined as the science of hiding a message's meaning. As a concept, it has been around for thousands of years. One famous historic example is Histiaeus of Miletus [36] who used to shave the heads of slaves and tattoo secret messages on them. He then waited for their hair to regrow before asking them to travel. Another example is Caesar's cipher [37] where each letter is shifted by a certain fixed offset. For this cipher, the offset value serves as a key. There is also the Scytale cipher [37] where a parchment is wrapped around a cylinder and then the secret message is written. For this cipher, the width of the cylinder serves a key.

Every encryption algorithm has two main operations, encryption and decryption. In encryption the sender encodes a message (*plaintext*) so as to obscure the meaning of that message (hence generating a *ciphertext*). The encoded message is then sent over a channel to a receiver who then decrypts this ciphertext back to the plaintext. The procedure of encryption and decryption is public knowledge, in accordance with Kerckhoffs' principle which states that *"A system must remain secure if everything except the key is known to the adversary"* [38]. All strong encryption schemes utilize *confusion* and *diffusion*. *Confusion* ensures that the relationship between the key and the plaintext is not obvious. *Diffusion* ensures that every plaintext symbol has an impact on multiple ciphertext symbols, for example changing on bit in the plaintext should change multiple bits in the ciphertext. The aim is to ensure the CIA criteria. Where the *'C'* comes from confidentiality, which means that the attacker should not be able to observe messages between the sender and the receiver, even in an un-secure environment. The *'I'* comes from integrity, which means that the message should not be altered during encryption or transmission. Finally, *'A'* comes from authenticity, which means that the message must come from the source it says it comes from.

Cryptographic algorithms can be further divided into 2 types, symmetric and asymmetric algorithms. These algorithms can in turn be classified as stream ciphers and block ciphers. Stream ciphers work on each bit of the plaintext individually whereas block ciphers work on blocks of plaintext bits. Crypto-algorithm classification can be seen in Figure 2.1.

2.2. SYMMETRIC ALGORITHMS

In symmetric algorithms, both the sender and receiver use the same key. If the plaintext bits are processed as blocks, the algorithm is known as block cipher. Some popular block and stream ciphers are as follows:

[1] heavily based on the book *Understanding Cryptography* by Christof Paar and Jan Pelzl [1]

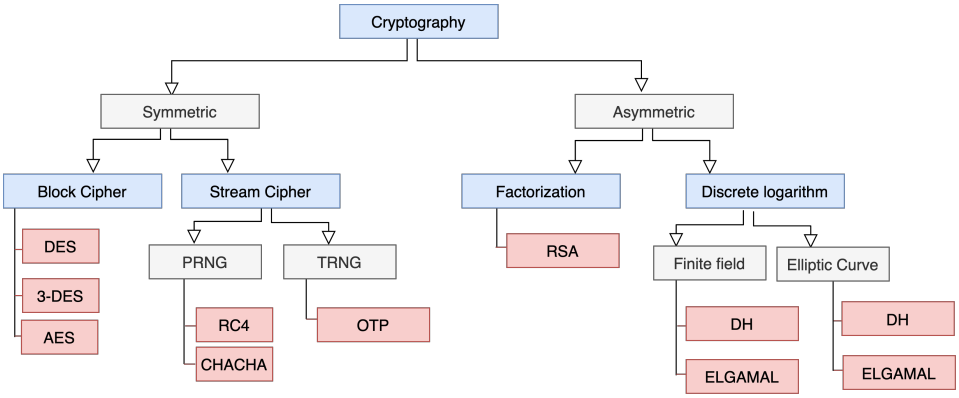


Figure 2.1: Cryptography Flowchart

2.2.1. DES

DES was introduced to the public by the National Security Agency in 1977 [39]. It encrypts 64 bits blocks with a key of size 56 bits. It has a total of 16 rounds and has a *Feistel network* structure, which means that for every input data block and subkey, the network outputs a block of the same size as that of the input data block. Note that only half of the input data block is encrypted in any particular round and the other half is simply xored. Also, the encryption and decryption in Feistel Networks are the same processes.. A flowchart of all DES operations can be seen in Figure 2.2.

Figure 2.2: DES Flowchart

DES does the following operations:

- Pass the 64 bit plaintext block through an 'initial permutation' function. This step has no importance on a cryptographic level and was solely made to assist the 8 bit hardware of mid 1970s.
- Next, this permuted input goes through 16 rounds of encryption, every round has its own subkey. All subkeys are derived from a main key, hence a key generator is used.
- The output of the encryption round then goes through a 'final permutation'. Since the initial and final permutation are inverse functions, this step reverses the effect of the initial permutation.

According to the intial permutation table, the 58^{th} input bit is mapped to 1^{st} output position, the 50^{th} input bit is mapped to 2^{nd} output position and so on. The final permutation would then inverse this effect. After the initial permutation, there are 16 rounds of encryption. All the 16 rounds have the same structure. The 64 input bits of each round are split equally into a left and a right component. The left 32 output bits are simply the right 32 input bits and the right 32 output bits are calculated using f function which takes a round subkey and the right 32 input bits.The operations can be mathematically represented as follows:

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \oplus f(R_{i-1}, k_i) \end{aligned} \tag{2.1}$$

The f function works on the right 32 bits of the previous round and current round key. The right 32 bits of the previous round are expanded to 48 bits using a so called E-box. Using this E-box, 16 of the 32 bits are duplicated and the resulting 48 bits are XORed with the round's key. The resulting 48 bit value is then split into eight 6-bit values that are fed into 8 different S-Boxes. These S-Boxes are an integral component to deter differential cryptoanalysis [40] since they introduce non linearity and increase the dependence of every plaintext bit and every key bit with every ciphertext bit. In short, S-Boxes ensures that the DES input and output can not be represented as a system of linear equations. If they were not part of the encryption scheme, it would be possible to solve for key bits using this system of linear equations.

The last thing to discuss about DES is the key schedule. The key schedule uses a 56-bit main key (out of the 64 bits, 8 bits are used for parity) which generates sixteen 48-bit round keys. The 64 bits first go through a permutation block labelled *PC-1*. Here, the 64 bits get converted to 56 bits, removing all the parity bits. The resulting 56 bits are split equally and go through a 1-bit shift left rotate in rounds 1,2,9 and 16 and a 2-bit shift left rotate for other rounds. The 56 bits after the shift left go through another permutation block labeled *PC-2* to make the output 48 bits, which can thereafter be used as the round key. The decryption process of DES is exactly the same as the encryption process and can be represented exactly as the encryption structure. Just the key schedule changes wherein for the first round there is no shift right rotates, for round 2, 9 ,16 there is a 1-bit shift right rotate and for the remaining there is a 2-bit shift right rotate. As is evident, the shifts in decryption are the absolute anti of those in the encryption.

2.2.2. TRIPLE-DES

The primary security issue with DES was its key length. 56 bits key space is considered too small for an algorithm to be deemed secure. This was the reason, 3-DES (pronounced *triple DES*) was introduced wherein 3 DES encryptions were carried out sequentially as can be seen in Figure 2.3. However, unlike the name suggests, 3-DES offers security equivalent to a key space of 112 bits (56*2) and not 168 bits (56*3). The reason for this is the *meet in the middle attack*. Simply put, since the encryption take place one after the other. Hence, the encryption output of the second DES block is the encryption input of the third DES block. Meet in the middle attack leverages this vulnerability by storing intermediate values of the encryption, with the aim of finding a key pair such that *encryption(plaintext,* key_1*) = decryption(ciphertext,* key_3*)*. where key_1 and key_3 are both 56 bits.

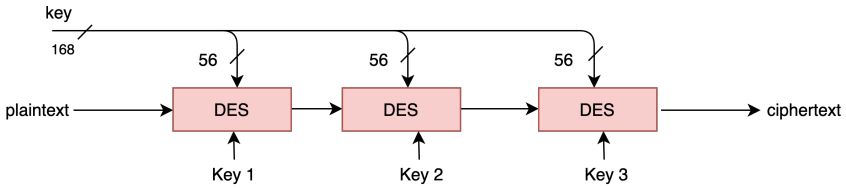


Figure 2.3: 3DES

Hence, instead of having to brute force 2^{168} possible keys, an attacker only need to brute force $2^{56} + 2^{112}$ possible keys.

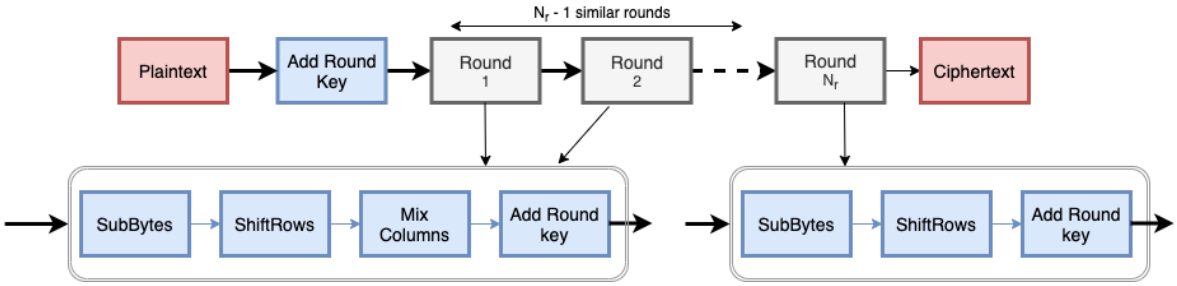
2.2.3. AES

3-DES is strong enough even for today's computationally standards. However, software implementation of DES is slow and 3-DES is thrice as slow and works on just 64-bit blocks. AES was introduced to handle these issues and became a United States federal standard in 2001. AES works on a block size of 128 and a key size of 128, 192 and 256 bits. The 128 bit key length version has a total of 10 rounds, 192 bits has 12 and 256 bits has 14 rounds. A block diagram of AES encryption can be seen in Figure 2.4. Note that every plaintext byte in AES is processed as an element of Galois field 2^8. Concisely, Galois Fields can be defined as a finite set of elements which can be added, multiplied, subtracted and inverted. The results of these mathematical operations result in a number that still belongs to the finite set owing to modulo arithmetic. For example, for Galois field(2) or GF(2), the operations are done modulo 2. If in GF(p) the value of p is prime, then the field is known as prime field. However, AES works with an extension field : GF(2^8). This field has a total of 256 elements and all the aforementioned mathematical operations are carried out using polynomial arithmetic.

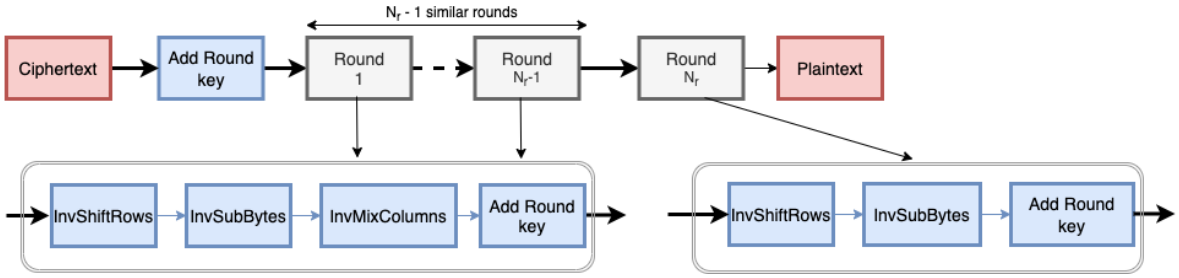
Every AES encryption has four operations which can be seen visually in Figure 2.5. Note that since AES works on a block size of 128, we can represent each block as a 4×4 matrix of bytes.

1. Substitute byte: This is a confusion layer and like DES, the substitute byte operation introduces a non linearity which makes AES resilient against differential cryptoanalysis [40]. Also, AES has a single SBOX as opposed to 16 SBoxes in DES. As represented in Figure 2.5a, each byte is substituted using the sbox. Sbox can either be implemented using a lookup table or using matrix multiplication and affine transform.

2. Shift Rows: As represented in Figure 2.5b, this is a Diffusion layer wherein bytes in every row are shifted left and rotated using 0,1,2,3 shifts for row 1,2,3,4 respectively.



(a) Encryption

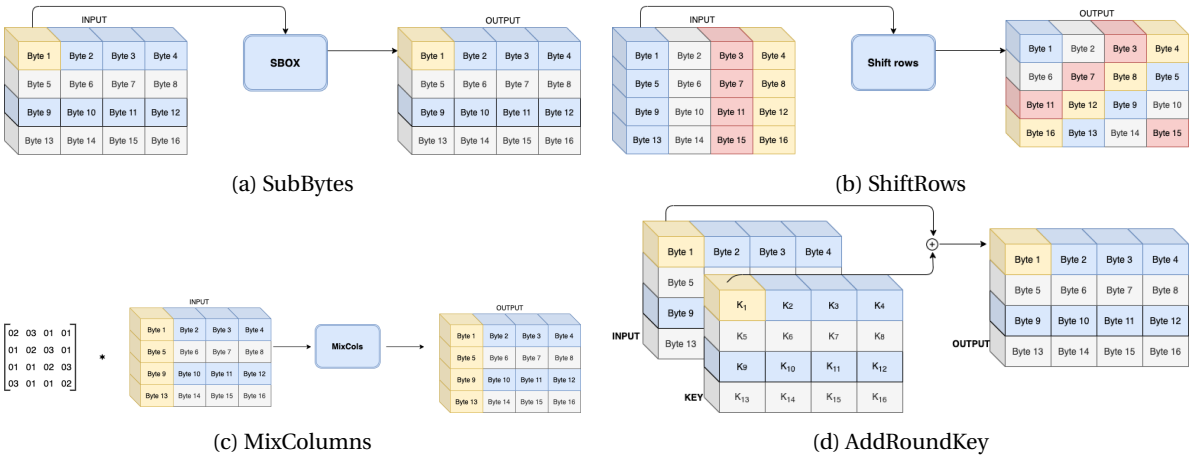


(b) Decryption

Figure 2.4: AES

3. Mix columns: Mix Columns is implemented using a matrix vector product as represented in Figure 2.5c. This is a Diffusion layer that combines all columns.

4. Add Round key: As can be seen in Figure 2.5d this is a confusion layer, where the input bytes are XORed with the round key.



(a) SubBytes (b) ShiftRows

(c) MixColumns (d) AddRoundKey

Figure 2.5: AES Operations

The last thing that needs to be discussed is the key schedule. The key schedule depends on the key length since that also decides the number of rounds. The 128 bits of the key are split into 16 parts, namely K_0, K_1, K_2 .... K_{15}. Our aim is to calculate a key expansion array which can be represented as W_0, W_1, W_2 .... W_{43}. K_0 is copied as is. For i in the range 1 to 10, the value of W is calculated as follows.

$$W[4i] = W[4(i-1)] + g(W[4i-1])$$

For next 3 words, keeping i in the range 1 to 10 and j as 1,2 and 3. The value of W can be calculated as follows

$$W[4i+j] = W[4i+j-1] + W[4(i-1)+j]$$

Note that g here rotates the input bytes, does a SBOX substitution and finally add a round coefficient. This function is necessary since it adds non linearity. The decryption process is not exactly the same as DES. Rather, the aim is to invert every operation carried out in the encryption phase. The two changes in decryption are that the key schedule is reversed and the order of operation is reverse of that of encryption. This can be seen in Figure 2.4.

2.2.4. CHACHA

ChaCha is a stream cipher, hence instead on working on blocks of plaintext bits like DES and AES it works on each bit individually. As a overview, it uses a 256 bit key and a 64 bit Nonce to encrypt 512 bits of the plaintext. In total there are 20 rounds, 10 column rounds and 10 diagonal rounds. All these rounds have just sum, XOR and shift operation and hence make ChaCha much faster than AES [41].

2.3. ASYMMETRIC ALGORITHMS

In Asymmetric algorithms, also known as public key cryptography, the sender and receiver use pair of keys known as public and private keys. They leverage one-way functions and do not depend on confusion and diffusion for encryption. We can divide asymmetric algorithms using the class of these one-way function it uses (as in Figure 2.1). Namely, factorization and discrete logarithm. If *f(x)* is a one-way function, calculating $f(x)$ is easy but calculating $f^{-1}(x)$ is computationally infeasible even using the state of the art algorithms and computation resources. In detail, the one-way functions can be defined as follows:

1. Factorization Problem : Say that function *f(x)* computes the multiplication of two prime numbers. Calculating $f(x)$ is easy but calculating $f^{-1}(x)$ is computationally infeasible. This means that finding prime multiplier and multiplicand corresponding to the output of $f(x)$ is an extremely hard problem.

2. Discrete Logarithm : Given g, b and B such that all $g, b, B \in Z_p{}^*$ where p is prime, calculating $g^b = B mod p$ is simple. However, using cyclic groups, calculating b when g and B are known is computationally infeasible.

Next, a few popular asymmetric algorithms will be discussed.

2.3.1. RSA

RSA leverages the factorization problem. Visually, the stepwise encryption and decryption process can be seen in Figure 2.6. As is evident from the Figure, RSA operations are based on exponentiation. To perform such large exponentiation operations, the square and multiply algorithm is used.

Square and multiply algorithm works bitwise on the binary representation of the exponent. Depending on the bits, the operation is decided. Bit '1' entails both square and multiply operations whereas bit '0' just requires the square operation. For example, if a value x is to be raised to the power of 5 (which is 101 in binary). The first '1' means that the value x is listed as is, the next bit '0' means that the value x will be squared to become x^2. Finally, the last bit '1' will mean that the value x^2 will be squared and then multiplied with the value x to become x^5.

2.3.2. DIFFIE HELMAN

In itself, Diffie helman is only used to generate a shared secret. Diffie Hellman ensures that the shared secret (key) is never communicated over the channel. Unlike RSA, Diffie Hellman leverages the Discrete logarithm problem.

Visually, the Diffie Hellman approach for generating a shared secret can be seen in Figure 2.7. This serves as an exchange scheme for symmetric encryption. For example, the shared secret can then be used for AES encryptions wherein the shared secret serves as the key.

2.3.3. ELGAMMAL

Another popular use of the Discrete logarithm problem is ElGammal crypto algorithm. This algorithms makes use of an ephemeral key and a masking key.

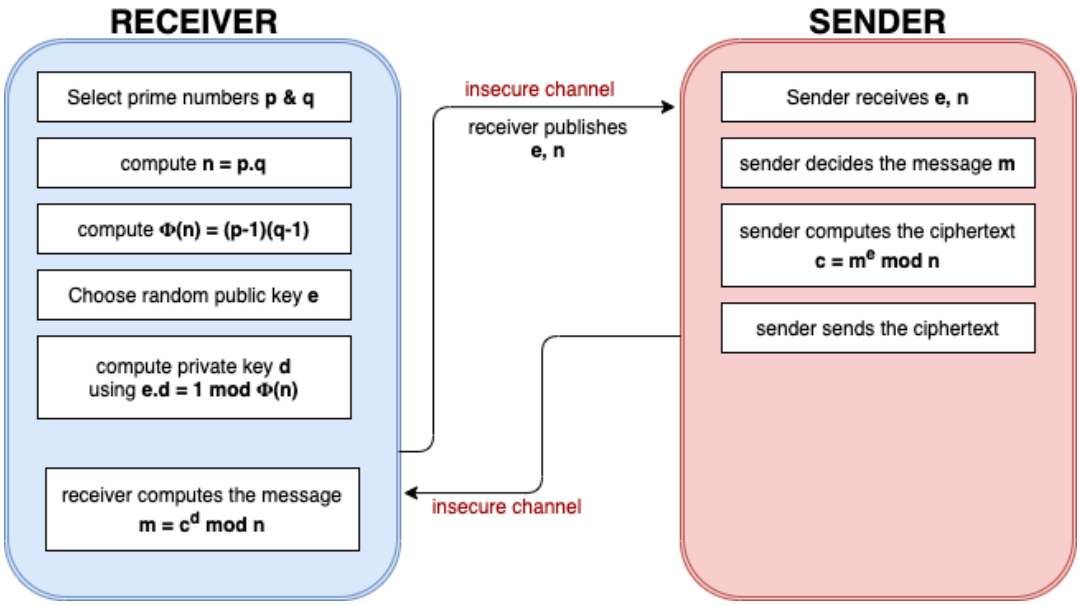


Figure 2.6: RSA

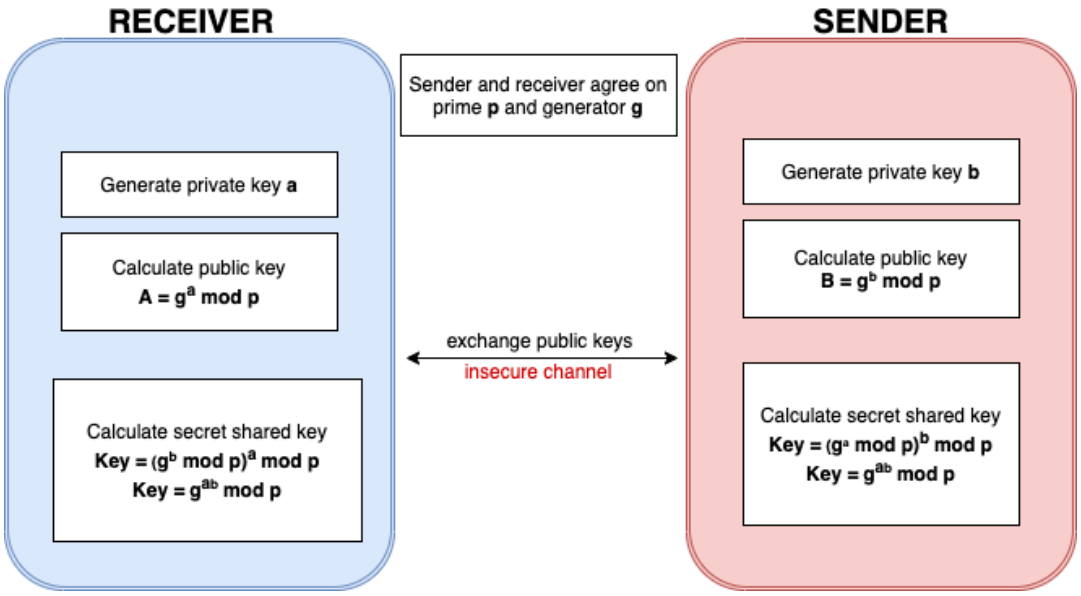


Figure 2.7: Diffie Hellman

Visually, the ElGammal crypto algorithm can be seen in Figure 2.8. The advantage of using an ephemeral key is that the same plaintext results in different ciphertexts since the ephemeral key is chosen by the sender themselves.

2.3.4. ECC

A very popular use of the discrete algorithm (especially for resource constrained devices) is Elliptic Curve Cryptography. Here, all the operations are done using points on an elliptic curve and an imaginary point infinity.

The prerequisites for an Elliptic curve here is that it must be symmetric about the x axis. Also, any straight line should not intersect the curve on more than three points. The addition of two points on the curve is done using the intersection of the line drawn through the two points and then mirroring it with respect to the x-axis. Adding a point to itself is done using tangents at that point to create a line.

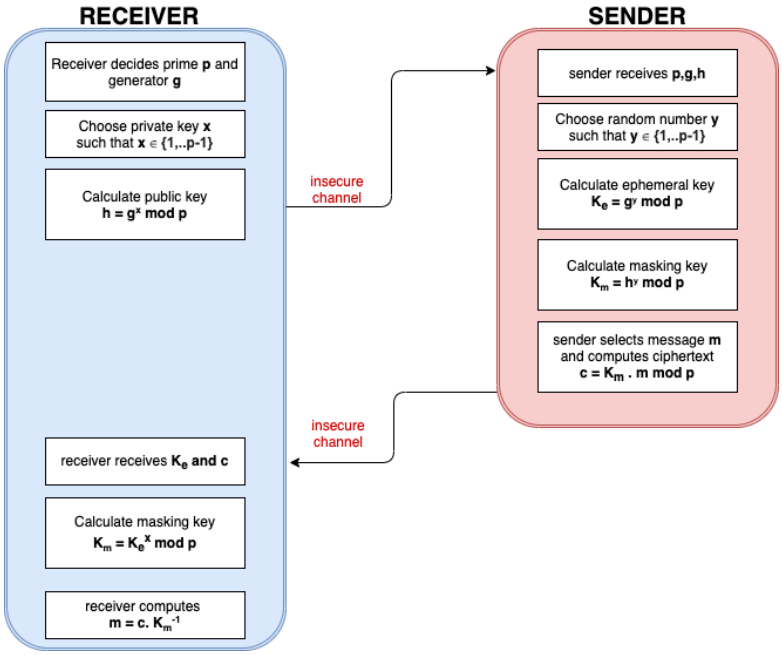


Figure 2.8: ElGammal

This addition process again and again such that the final point ends at the first point is a hard problem. Hence for a two arbitrary point P and Q, computing the value of k such that $k.P = Q$ is a hard problem. This problem can be used to replace the discrete logarithm problem in Diffie Helman and ElGammal.

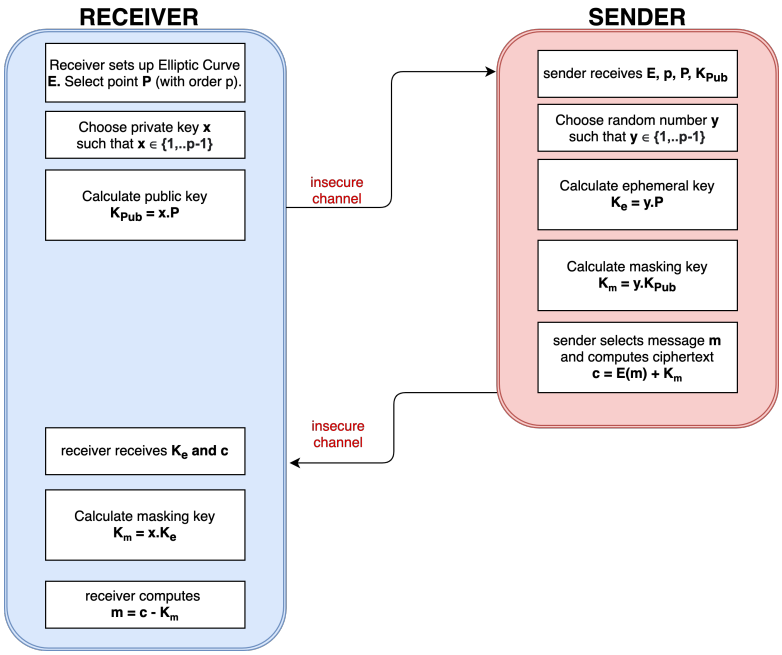


Figure 2.9: Elliptic curve ElGamal

The steps for ECC ElGammal can be seen in Figure 2.9. ECC is the preferred encryption scheme for embedded applications [42]. This is because 160-256 bit ECC provides the same level of security as 1024-3072 bit RSA, substantially reducing the computational requirements and the size of ciphertexts [43].

3

Side Channel Attacks and Countermeasures

Side channel attacks leverage the information leaked via the implementation of a cryptographic algorithm on hardware or software, rather than any weaknesses in that cryptographic algorithm itself. This leaked information, amongst others, can be in terms of power, EM radiation, or the time taken to perform a cryptographic operation. Section 3.1 describes the type of side channel attacks and Section 3.2 gives an overview of the two main kinds of power based side channel attacks, which are non-profiled and profiled. Section 3.3 explains non-profiled attacks and section 3.4 explains profiled attacks. Finally Section 3.5 describes possible countermeasure against such power based side channel attacks.

3.1. Types of Side Channel Attacks

Hardware attacks are classified in literature as in figure 3.1. Side channel attacks are a specific type of hardware attacks that aim at obtaining secret data using information leakage via the implementation of a cryptographic algorithm.

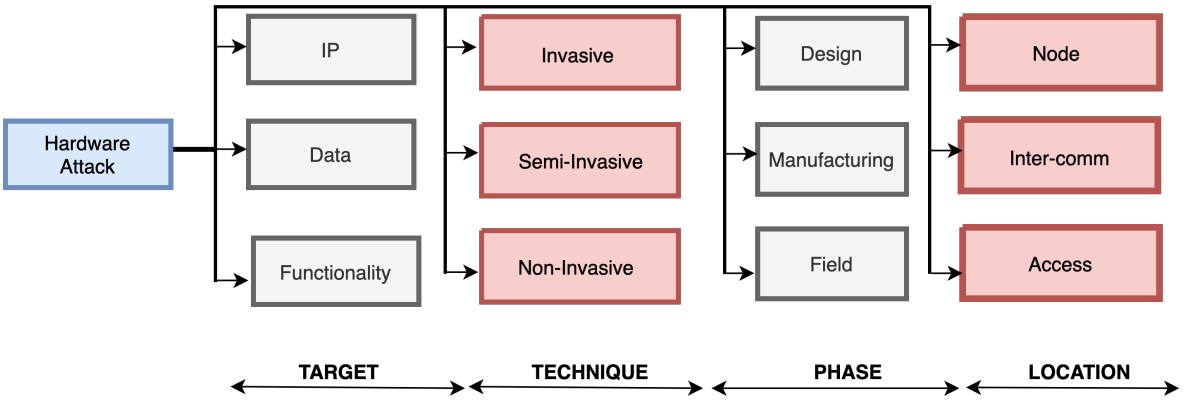


Figure 3.1: Classification of Hardware attacks

Every side channel attacks has four main components, as represented in Figure 3.2. First, the target of the attack should be decided. In this case, the target is the *data*. Other targets can be to steal the *IP* of design or to to hamper the *functionality*. Next, the technique is decided. Some of the attacks require removing the package of the IC. If the die is tampered when the attack is carried out, the attack is called *invasive*. If the die is not tampered but the package is removed, the attack is called *semi-invasive*. If the package is not required to be removed at all, the attack is called *non-invasive*. These attacks happen after manufacturing and once the device is in field hence the phase of the attack is *field*. Other attack phases can be during the *design* phase or during the *manufacturing* phase, when the design is being manufactured.

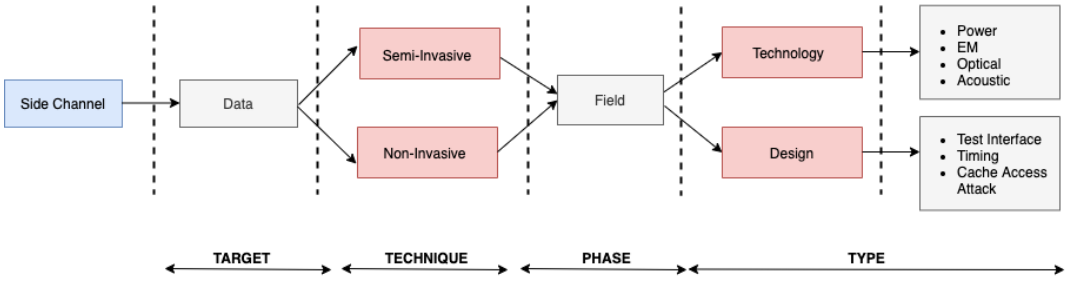


Figure 3.2: Side-channel attacks

Finally, the location where the attack is to be carried out is decided. This location then decides the *type* of the attack. If the attack targets the technology, the leakage can be in the form of Power[44], EM radiation[45], Optical[46] or Acoustic[47]. If the attack targets the design, the attack can leverage the disparity in the time taken to carry operations [48], memory access [49] and even the information in the scan chains [50]. In this work, our focus is power based side channel attacks. In the next section, they are discussed in detail.

3.2. POWER BASED SIDE CHANNEL ATTACKS

Power analysis attacks leverage the difference between power consumption during encryption. As demonstrated by Gu and Almasry [51], the power consumption depends on the switching activity of transistors. This switching activity of the transistor is proportional to the change of input values to the transistor. The ability to correlate the power with the secret inputs is the premise on which power side channel attacks are based. The amount of correlation between power consumption and the value of encryption keys can define the effectiveness of a side channel attack.

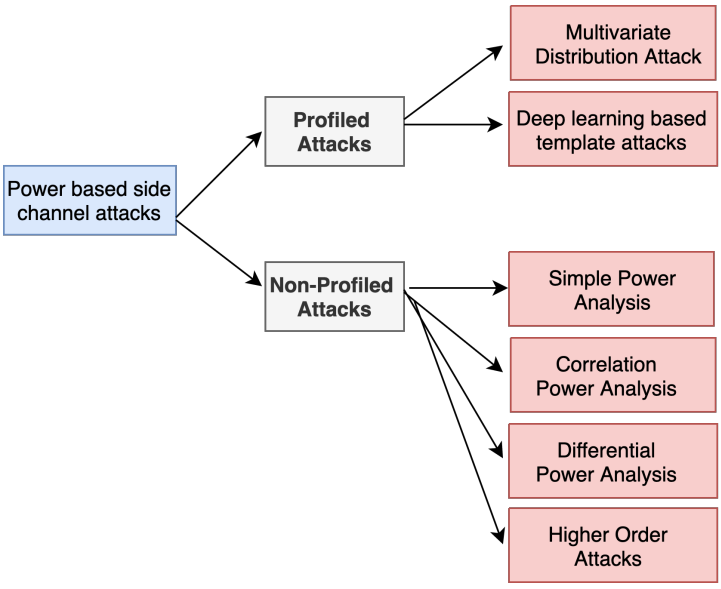


Figure 3.3: classification of side channel attacks

A taxonomy of the various kinds of power side channel attacks can be seen in Figure 3.3, this is inspired by [52]. For profiled attacks, the attacker is expected to have access to a clone of the device they plan to attack. Hence, the attacker has the liberty to perform all sorts of operations on the clone and make models (profiles) that can then be used at attack time. Of course, while attacking the target device they would not know the encryption key. However, the clone profiles obviate the need of taking a large amount of traces from the target device since the profile usually identify points of interests which then reduces the amount of noise in the target analysis by a substantial amount. As for the non-profiled attack, the attacker is expected to have limited access to the device and no liberty in terms of making profile models.

3.3. NON-PROFILED ATTACKS

In Non-Profiled attacks, the adversary can procure a limited amount of traces for a fixed unknown key using plaintexts of their choice. The adversary leverage public knowledge about the implementation of the encryption process to carry out the attack.

3.3.1. SPA : SIMPLE POWER ANALYSIS

As the name suggests, this is the simplest of all attacks that will be discussed in this chapter. Here, the power consumption during the execution of a cryptographic algorithm is constantly monitored. The aim is to map certain operations of the encryption process to their corresponding power consumption so as to extract the key.

Simple power analysis can be used against (unprotected) RSA's square and multiply algorithm, as can be seen in Figure 3.4. Taking this figure into account, it is easy to guess the exponent value. This is because, as explained in Section 2.3, RSA exponentiation is done using a square and multiply algorithm which internally uses the binary representation of the exponent to decide whether a square operation or a square and multiply operation should be carried out. It can also be used AES' key expansion phase and can be used to reduce the key space for brute forcing the key [53].

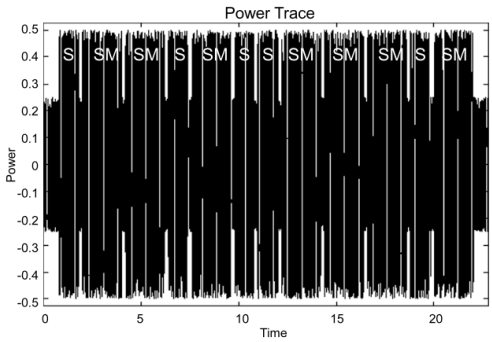


Figure 3.4: Simple power analysis against RSA [1])

3.3.2. DPA : DIFFERENTIAL POWER ANALYSIS

Differential power analysis is done by splitting the power trace into groups on the basis of each bit of the calculated intermediate value. In total there are 5 steps.

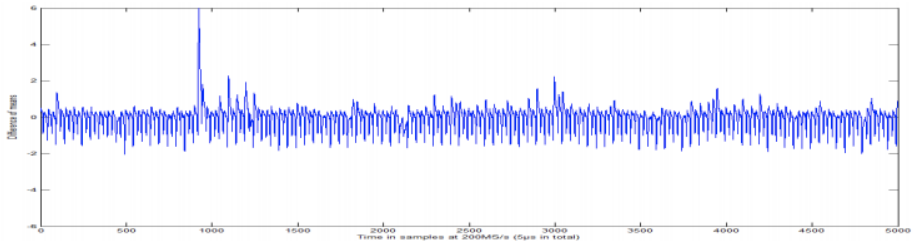


Figure 3.5: DPA against AES: Correct key guess [2])

Step 1: Select the intermediate value. SBOX output is chosen since SBOXs are non linear and hence are easily identifiable.

Step 2: Procure power traces for a set of plaintexts. If there are t plaintext and each trace has n sample points, the result of this step would be a vector of plaintexts and a $t \times n$ matrix with power traces.

Step 3: Formulate a hypothesis. Loop over all 256 keys and xor with the plaintext values. This xored value is then fed into our selected intermediate value (as in step 1), which is the sbox.

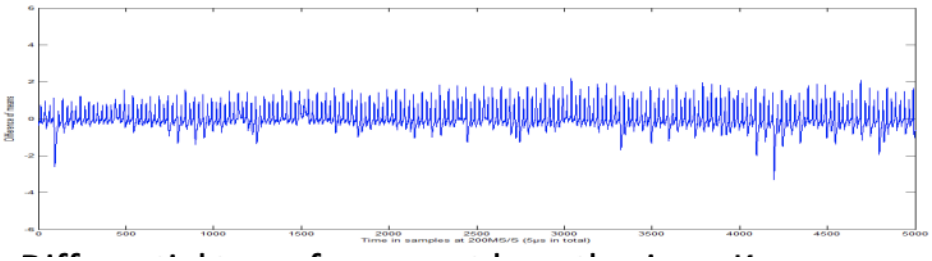


Figure 3.6: DPA against AES: Incorrect key guess [2])

Step 4: Partition traces on the basis of the hypothesis. This can be done in a number of ways. The LSB of the intermediate value can be chosen to partition the traces. Hence, the traces after this step will be partitioned into 2 sets. One where the LSB of the sbox output was 0 and one where the LSB of the sbox output was 1. They can be represented as matrices of size $t_1 \times n$ and $t_2 \times n$ such that $t_1 + t_2 = t$.

Step 5: Finally, a differential trace is calculated using difference of means of the partitioned traces. This step is repeated for all possible key hypothesis. A correct key guess results in large peaks (as can be seen in Figure 3.5) in the differential trace and incorrect key guess would result small/no peaks in the differential trace (as can be seen in Figure 3.6). These peaks can hence be used to predict the most likely key.

Differential power analysis, as an attack, is pretty intuitive. However, it is not extremely robust and the accuracy depends on the choice of partitioning. For example, in the explanation above the LSB of the intermediate value was chosen for partitioning. It is possible that some other form of partitioning works much better/worse. Also, working on just a single bit makes this attack lose out on information from the other bits.

3.3.3. CPA: CORRELATION POWER ANALYSIS

Correlation power analysis tries to address the drawbacks of Differential power analysis. It tries to take into account all the bits of the intermediate value using the concept of leakage models. Some popular leakage models are:

- Hamming weight : Number of set bits in the intermediate value is known as the hamming weight.
- Hamming distance : The difference between the number of set bits between the input and the output of intermediate function (SBOX, for example).

In total, there are 5 steps.

Step 1: First step is to select the intermediate value. SBOX output is chosen since SBOXs are non linear and hence are easily identifiable.

Step 2: Procure power traces for a set of plaintexts. If there are t plaintext and each trace has n sample points, the result of this step would be a $t \times n$ matrix with power traces and a vector of plaintexts.

Step 3: Formulate a hypothesis. Loop over all 256 keys and xor with the plaintext values. This xored value is then fed into our selected intermediate value (as in step 1), which is the sbox.

Step 4: Next, the leakage model is applied to the intermediate values. For every possible key, there will be $t \times 1$ vector signifying the leakage model output. Since there are 256 possible keys, this step's output can be represented as a $t \times 256$ matrix.

Step 5: Finally, Pearson correlation between the power traces and the leakage model output is calculated. Hence, the output of this step is a $256 \times n$ matrix. The row in this with the highest valued elements is the key guess.

3.3.4. HIGHER ORDER DIFFERENTIAL POWER ANALYSIS

Unlike standard DPA, this attack aims at analyzing multiple cryptographic sub-operations on every sample in a power trace. It is particularly effective against countermeasure like masking [54]. Simply put, with higher order attacks both key-dependent and key-independent operations are analyzed. For example, in case of

masking, the power trace is analyzed to not only obtain the key value but also the random mask value with the aim of cancelling the mask. The steps in higher order attack stay similar to the standard attack with the exception that signals collected from multiple sources with varied time offsets are considered [55]. Naturally, the higher order attacks entail a substantial computational overhead. This is a major contributing factor to why the higher order attacks are not as popular as other attacks introduced in this section.

3.4. PROFILED ATTACKS

In profiled attacks the attacker has access to a clone of the device that they plan to attack. Hence,they have the ability to perform a plethora of operations on that clone so as to make profiles to be used at attack time. These profiles obviate the need of procuring large amount of traces from the target device since the profile usually identify points of interests(POI) which reduces the amount of noise in the actual analysis by a substantial amount. These attacks are split across two phases: The first phase is for profiling and the second phase is extraction.

3.4.1. TEMPLATE ATTACK

Originally introduced by Chari *et al.* [56], template attacks leverage multivariate Gaussian distributions to model points of interest in the power traces from a cloned device. These multivariate Gaussians are then used on the power traces from the target device to carry out the attack. The attack is carried out on a byte-by-byte basis and the steps are as follows:

PROFILING

Step 1: The attacker needs to procure a clone device.

Step 2: Acquire power traces for a set of plaintexts. If there are t plaintext and each trace has n sample points, the result of this step would be a $t \times n$ matrix with power traces and a vector of plaintexts.

Step 3: Formulate a hypothesis. Since the key for the clone device is known, there is no need to loop over all 256 keys. Instead the known keys can be xored with the plaintext values. This xored value is then fed into our selected intermediate value (as in step 1), which is the sbox. Next, the leakage model can be applied to these intermediate values.

Step 4: The traces are separated on the basis on the leakage model values. For example, the traces can be separated into 9 group on the basis of hamming weight values. Note that these groups are not all of the same size. This class imbalance has a detrimental impact on the attack performance and over-sampling methods have be proven to enhance performance [57].

Step 5: A Mean vector is calculated for each of the groups. Difference between these mean vector are marked as POI or points of interest.

Step 6: For m points of interest, a mean vector ($m \times 1$ vector) and a covariance matrix ($m \times m$ matrix) are calculated. This is for each group, so if hamming weight is used, 9 such vectors/matrices will be computed.

EXTRACTION

Step 1: Procuring the target device

Step 2: Acquiring traces from the target device. The number of traces needs here are much less as compared to the profiling phase.

Step 3: Applying the multivariate normal probability function to the acquired traces. If there are k categories in the leakage model (for example, k=9 in case of hamming weight) and there are t traces: the output of this step will be a matrix of size $t \times k$.

Step 4: Similar to CPA, xor of all 256 keys with the plaintexts is calculated. This xored value is then fed into our selected intermediate value, which is the sbox. Next, the leakage model is applied to the intermediate values. For every possible key, there will be $t \times 1$ vector signifying the leakage model output. Since there are 256 possible keys, this step's output can be represented as a $t \times 256$ matrix.

Step 5: The matrix obtained in step 4 is correlated with the matrix obtained in step 3. The output of this step is a correlation matrix and logarithmic sum of every column in this matrix produces a 1×256 vector whose argmax is the key guess.

3.4.2. DEEP LEARNING BASED ATTACKS

This attack is almost the same as the template attack. The only difference here is that instead of profiling using a multivariate Gaussian, a deep neural network is used. The neural network can either be used to guess the leakage model output or the actual key. For guessing the leakage model output, the attack steps are as follows:

PROFILING

Step 1: The attacker needs to procure a clone device.

Step 2: Acquire power traces for a set of plaintexts. If there are t plaintext and each trace has n sample points, the result of this step would be a $t \times n$. matrix with power traces and a vector of plaintexts.

Step 3: A classifier is trained to classify the traces in to categories on the basis of leakage model's output. For example, if hamming weight is used as the leakage model, a classifier will be trained to classify power traces into 9 categories. Since the output of xoring the plaintext and the key is 16 bytes, 16 neural net classifiers need to be trained. It is also possible that the 16 bytes can be processed into a single value, and using that just a single classifier needs to be trained.

EXTRACTION

Step 1: Procuring the target device

Step 2: Acquiring traces from the target device

Step 3: Applying the acquired traces to the trained classifiers (or a single classifier depending on the label processing). If there are k categories in the leakage model (for example, k=9 in case of hamming weight) the output of this step is the guessed leakage models for the 16 bytes (or a single processed leakage model value).

Step 4: Finally, this guessed leakage model value can be used to make a key guess.

The same steps can be followed if the aim is to guess the key. For this, instead of the k categories of the leakage model, the classifier aims at predicting a key byte's value (which can be anything between 0 to 255).

3.5. COUNTERMEASURES

As the number of side channel attacks increased over the years, so have the countermeasures against them. The countermeasures aim at reducing the amount of information leaked via the side channels.

The countermeasures against side channel attacks can be subdivided into the following four categories [58] as can be seen in Figure 3.7.

1. Blinding : Blinding aims at reducing the information leaked by both the technology [59] and design [60]. On the technology front, using complementary logic is one of the blinding techniques. This technique unavoidably entails an area overhead. From the design aspect, using constant timing operations is a blinding technique. This technique entails a performance reduction since it often requires to avoid optimizations.

2. Randomization : Side channel analysis is heavily dependent on alignment. The leaked information is not of much use if it can not be correctly aligned with the cryptographic operations. The aim of randomization is to introduce misalignment. Randomization can be introduced in the form of random operations. From an AES perspective since the SBOX's non-linearity is mostly leveraged in side channel analysis, a technique called masking can be used to hide the leakage. Masking [61] works as follows: by selecting two random 8-bit values (mask m and mask n) the SubByte function can be calculated using Equation 3.1.

$$SBOX[P \oplus K] = SBOX'[P \oplus K \oplus n] \oplus m \tag{3.1}$$

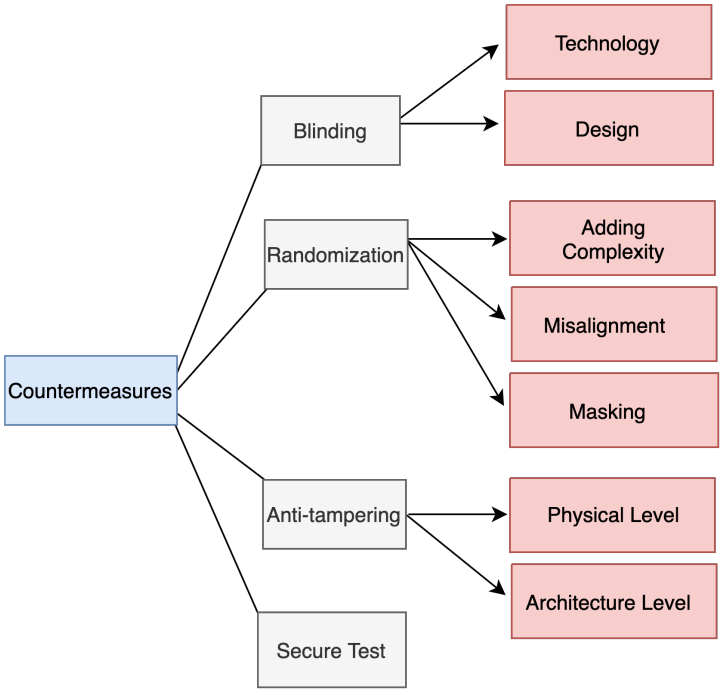


Figure 3.7: Countermeasures against side-channel attacks

In Equation 3.1, P denotes a byte of the plaintext, while K represents a byte of the secret key. To ensure that the encryption output is correct, the SBOX also needs to be modified. Note that the SBOX performs a non-linear transformation, hence for given plaintext P and mask n, the sbox output $SBOX(P \oplus n)$ is not equal to $SBOX(P) \oplus SBOX(n)$. This countermeasure aims to reduce the leakage by invalidating the Hamming Weight and Hamming Distance leakage models.

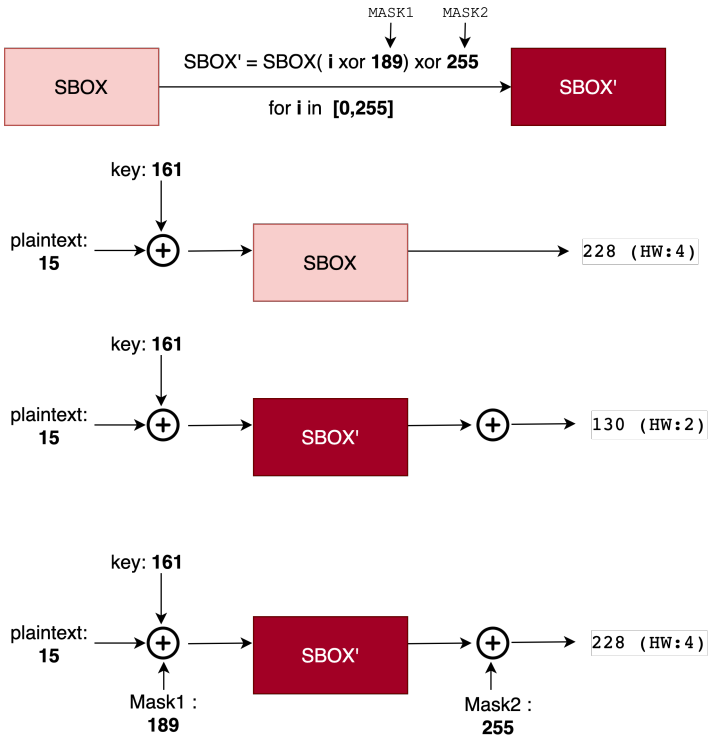


Figure 3.8: SBOX with and without masking

A simple example of the same can be seen in figure 3.8. In the example, mask pair (189,255) is used. First, the *SBOX* is modified using the mask pair so as to become *SBOX'*. The impact of that is that the same key and plaintext byte results in two different values, which are 228 and 130 for *SBOX* and *SBOX'* respectively. So as to correct the output of *SBOX'*, mask1 and mask2 are needed. Mask1 will make sure that the correct index is accessed in the SBOX look-up table and mask2 will undo the effect of masking on the value contained in the look-up table.

3. Anti-tampering : To prevent the optical side channel analysis and micro-probing, metal shields or probe sensors [62] can be used. On an architectural level, sensitive operations can be carried out in the so-called trust zones [63] that prevents non secured software to access data from secured resources.

4. Secure test : Many side channel attacks use the information in the design's test architecture like the scan chains [50]. To make this information hidden from the attackers, the scan chains can either be encrypted or the entire test infrastructure can be to destroyed after testing.

4

DEEP LEARNING

This chapter will give an overview about deep learning. Section 4.1 gives a brief account of the history of deep learning. In Section 4.2, a few fundamental neural networks are explained. Section 4.3 covers generative deep learning and gives a comparison of state-of-the-art generative models. Finally, Section 4.4 describes the process of building and training neural networks along with background information on using PyTorch [64] and Keras [65] as well as the process of hyperparameter tuning.

4.1. HISTORY OF DEEP LEARNING

Deep learning, as defined by GoodFellow *et al.* [3], is a kind of machine learning that learns nested hierarchical representation of the world wherein more abstract concepts are represented in forms of less abstract ones.

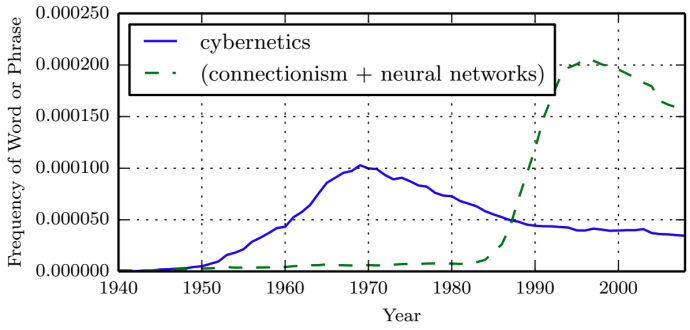


Figure 4.1: History of deep learning [3]

On its own it is not a new concept, the history of artificial neurons goes back to 1943 when McCulloch and Pitts made a mathematical model of an artificial neuron[66]. However, it must be stated that the term "Deep learning" is fairly new. It was known as Cybernetics between 1950 and 1960 and Connectionism between 1980 and 1990, as depicted in Figure 4.1. The current ubiquity of deep learning can be attributed to cheaper computation resources, cheaper memory storage and cheaper sensors all without compromising on the quality [4]. This combination of readily available high quality data, cheap yet fast storage and strong computation resources in form of GPUs is what makes Deep learning so popular.

Owing to the the artificial intelligence renaissance [67], machine learning is now ubiquitous. Deep learning is a sub domain of machine learning which depends on artificial neural networks to perform tasks with accuracy comparable to or exceeding that of humans. As the name suggests, these artificial neural networks take after the biological systems, with their constituent cells being neurons. The term "deep" in deep learning springs from the use of multiple layers of neurons. An example of a multi layer perceptron can be seen in Figure 4.2. Deep learning and classical machine learning models differ in the way they learn. Deep neural networks obviate the need of human intervention in feature extraction, hence rather than telling networks how to do a task, it is shown examples of what is to be done.

4.2. FUNDAMENTAL NEURAL NETWORKS

4.2.1. LINEAR NEURAL NETWORKS

Artificial neural networks (ANNs), loosely speaking, architecturally take after the human brain. The basic constituent of these networks is a neuron. Similar to how a biological neuron has dendrites, nucleus, axon and axon terminals ANN neurons have input terminals, processing node, output wire and output terminals respectively [4]. Biologically, information x_i arriving from other neurons is received in the dendrites and this information is scaled using the synaptic weights w_i. In the nucleus, these scaled values are combined as a weighted sum y = $\sum x_i * w_i + b_i$, where b_i is the bias added to the scaled information by the nucleus. The output of this stage is processed by the axon, wherein it is first passed through some non linearity and finally used. Analogously, for artificial neural networks the forward pass looks exactly the same.

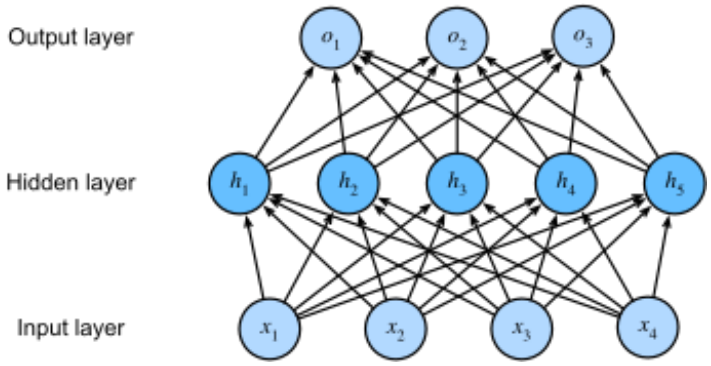


Figure 4.2: Linear neural network [4]

In linear neural networks, every neuron in one layer is connected to every neuron in the adjacent layers. The multi layer perceptron in Figure 4.2 is an example of linear neural networks. These networks are fairly easy to interpret and build, but the fully connected architecture make the number of trainable parameters extremely high for large networks.

4.2.2. CONVOLUTIONAL NEURAL NETWORKS

Unlike linear neural networks which flatten all inputs before a forward pass, convolution neural networks maintain the input's number of spatial dimensions as is. Instead of having connections between each pair of neurons in adjacent layers, CNNs are known for parameter sharing wherein the same set of weights and biases are used for the entire input. Also, using pooling, the CNNs are capable of focusing on the most important part of the input.

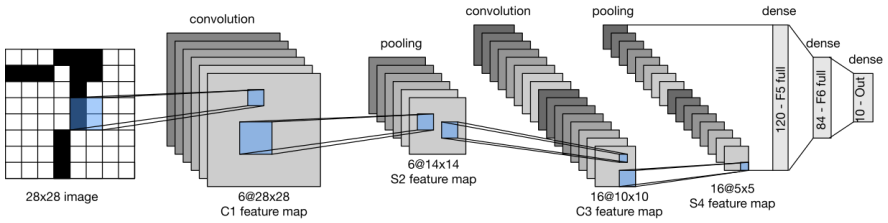


Figure 4.3: CNN: LeNet5 [4]

CNNs are architecturally quite different from linear neural networks, as can be seen in figure 4.3. The two main differences of the CNN as opposed to a linear neural network are: first, that it focuses on certain regions of the input at a time as opposed to all the input at once and second, that precise location of a particular feature in the input is not important. The first difference is a result of weight sharing and causes translational

equivariance which helps the CNN perform better in feature detection at different locations in the input . CNNs are equivariant, which implies that applying a transform and then convolving is the same as first convolving and then applying a transform. This is important since the same set of weights can detect a patterm irrespective of its position in the input. The second difference is a result of the pooling operation and is called translational invariance, which means that feature presence is more important then feature location. This is because the pooling operation is able to reduce the output at a certain position in the feature map to a summary statistic of the neighbouring outputs of that location in the feature map.

4.2.3. RECURRENT NEURAL NETWORKS

Recurrent neural networks are a class of artificial neural networks, that have the ability to understand sequential data. In essence, RNNs work as well with sequential information as the CNNs work with spatial information. RNNs are able to do so since they do not make the i.i.d assumption, which is not true for most data, especially for sequential data where there is strong dependency between subsequent samples. A good example can be seismic data for earthquakes, such data is not only spatially correlated (proximity to epicenter) but also temporally correlated (multiple small earthquakes succeed a major earthquake). Similarly, say that stock prices need to predicted at time t. For an autoregressive model, stock price at time t can be calculated as follows:

$$x_t \sim p(x_t \mid x_{t-1}, \ldots, x_1) \tag{4.1}$$

However, to keep this calculation tractable x_t can be calculated not using samples at all time steps but just a few previous samples. If only the previous K timesteps are considered, the model will be called a K^{th} order Markov model, like in equation 4.2.

$$x_t \sim p(x_t \mid x_{t-1}, \ldots, x_K) \tag{4.2}$$

Unlike multilayer perceptrons or convolutional neural networks that can be represented as directed acyclic graphs, RNN have self loops [4].

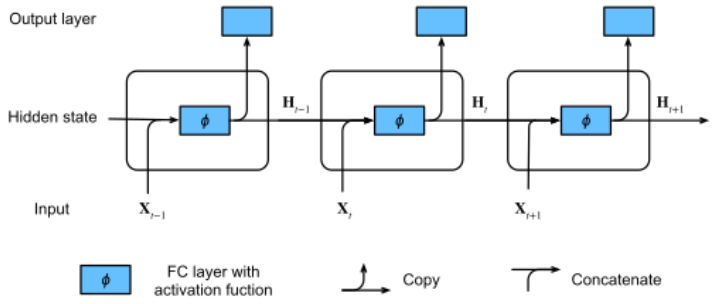
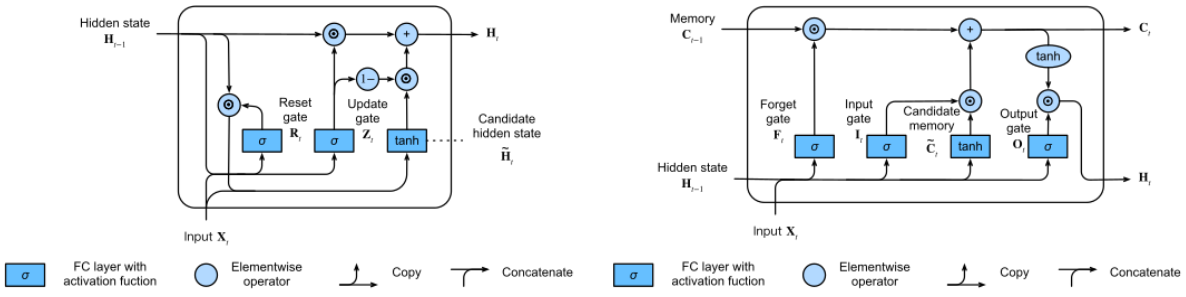


Figure 4.4: Simple RNN [4]

Hidden states in simple RNNs are computed as $H_t = \phi(X_t W_{xh} + H_{t-1} W_{hh} + b_h)$, as depicted in figure 4.4. The output for simple RNNs is calculated as $O_t = H_t W_{hq} + b_q$. Here, H is the hidden layer's output, W is the weight parameter, b is the bias and ϕ is the non-linear activation function. Since these hidden states are same for all timesteps, the number of trainable parameters do not increase with the number of timesteps. Unfortunately, standard RNNs suffer from the problem of vanishing and exploding gradients due to which they can not maintain long range dependencies.

4.2.4. MODERN RECURRENT NEURAL NETWORKS

Modern recurrent neural networks introduce the concept of gates, namely: forget, input, reset and update gates as a solution to the vanishing/exploding gradient problem in standard RNNs. These problems stem from the innate architectural deficiencies in the standard RNNs. Unlike the MLP introduced in Section 4.2.1, RNNs have self loops that signify a path for information to flow from one step to the next. Hence, every step not only includes information fed in at a particular time step but also hidden information from the previous time steps. This process, however, does not ensure that the RNN remembers all the information equally. This is because while backpropogating, the gradients for the hidden state for a particular timestep depends on the gradients of the hidden state after it. Hence, there is an exponential decrease/increase in the gradients while moving further into the memory of the RNN, causing the RNN to not learn long range dependencies.



(a) GRU [4]

(b) LSTM [68]

Figure 4.5: Modern RNNs [4]

GRUs try to solve this problem by using the reset and update gate, as follows:

$$\begin{array}{ll} \text{Reset gate} & R_t = \sigma\left(X_t W_{xr} + H_{t-1} W_{hr} + b_r\right) \\ \text{Update gate} & Z_t = \sigma\left(X_t W_{xz} + H_{t-1} W_{hz} + b_z\right) \end{array} \tag{4.3}$$

The values of the reset and the update gate are then used to calculate the candidate hidden state and the current state as follows:

$$\begin{aligned} \tilde{H}_t &= \tanh\left(X_t W_{xh} + \left(R_t \odot H_{t-1}\right) W_{hh} + b_h\right) \\ H_t &= H_{t-1} \odot Z_t + (1 - Z_t) \odot \tilde{H}_t \end{aligned} \tag{4.4}$$

This can also be seen visually in Figure 4.5a. Here, H is the hidden layer's output, W is the weight parameter and b is the bias. A LSTM, on the other hand makes use of the following gates:

$$\begin{array}{ll} \text{Input gate} & I_t = \sigma\left(X_t W_{xi} + H_{t-1} W_{hi} + b_i\right) \\ \text{Forget gate} & F_t = \sigma\left(X_t W_{xf} + H_{t-1} W_{hf} + b_f\right) \\ \text{Output gate} & O_t = \sigma\left(X_t W_{xo} + H_{t-1} W_{ho} + b_o\right) \\ \text{Candidate memory cell} & \tilde{C}_t = \tanh\left(X_t W_{xc} + H_{t-1} W_{hc} + b_c\right) \end{array} \tag{4.5}$$

Using the output of these gates, the LSTM calculates the memory cell state and the hidden state as follows:

$$\begin{aligned} C_t &= F_t \odot C_{t-1} + I_t \odot \tilde{C}_t \\ H_t &= O_t \odot \tanh\left(C_t\right) \end{aligned} \tag{4.6}$$

This can also be seen in Figure 4.5b. It must be noted that both GRU and LSTM are more computationally complex than a standard RNN. However, they provide a substantial performance improvement over standard RNN since they leverage the gate mechanisms to ensure that the vanishing gradient problem is solved. Also, the gates serve as a medium to focus on the important information in the input, making the modern recurrent networks much better than vanilla RNNs on tasks like language processing.

4.3. DEEP GENERATIVE MODELS

Deep generative models aim at approximating the underlying distribution of the training data. For example, say that there is a dataset of images of human faces such as the CelebA dataset [69]. All the samples of this dataset can be said to have been sampled from an underlying distribution p_{data}. If a generative model is trained on this dataset, the aim will be to approximate the distribution of human faces. To understand why traditional neural networks are not used to perform this task, lets consider the following example. Lets say that 255x255 colored pictures of human faces need to be generated. Since each colored picture has 3 channels: Red, Green and Blue, the total number of pixels in the images would be 195,075. Each of these pixels can in turn take a value between 0 and 255. Hence, out of $256^{195,075}$ possible combinations, a few possible combinations that closely mimic actual human faces need to be chosen. Mathematically, this can be represented as:

$$P(x \mid \theta) = \frac{f(x \mid \theta)}{\sum_z f(z \mid \theta)} \tag{4.7}$$

Here the denominator term is the summation of $256^{195,075}$ possible combinations. x is the image classified by a neural network f with parameters θ as real or fake. The problem doing so, other than the obvious computational infeasibility, is that procuring the amount of training data required for such a task is extremely tough.

This is why the field of deep generative learning was created and off late has seen a number of advancements. Since the inception of generative adversarial networks in 2014 [70], a lot of work has been done in the field of generative models for images [71] [72] [73]. Researchers have explored a lot of other avenues like generative models for audio [74] [75] [76] [77], ECG traces [78] [79], seismic data [80] and for time series data [81].

As a rule of thumb, generative models have three goals.

1. Density estimation : Unlike discriminative models that aim to find a class or a regressive value corresponding to the input, generative models aim at estimation the underlying distribution p_{data} of a particular dataset without overfitting on (memorizing) the training samples.

2. Generation: Once the generative model has been trained, it should be able to be used to generate samples that mimic the training data. Also, it should not output the same samples again and again.

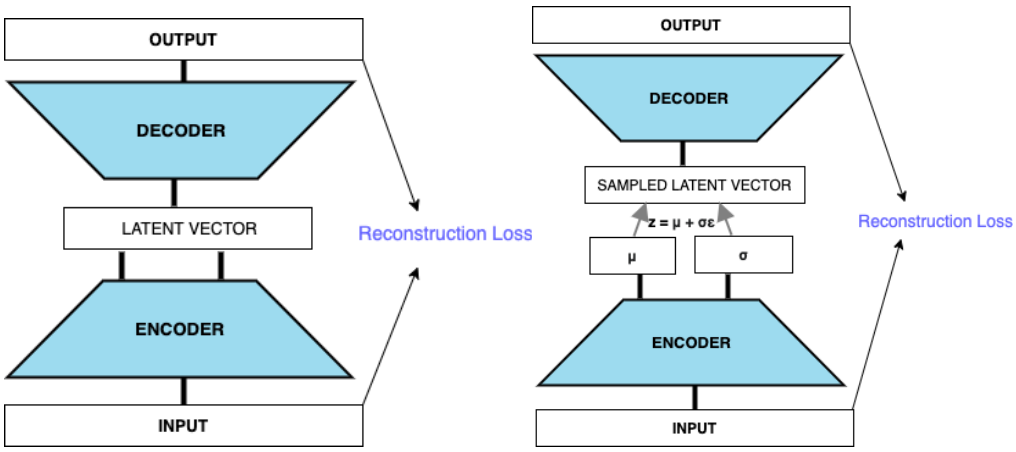
3. Having latent variables : Since some parameters of the training data (like the thickness of pen strokes while writing digits) can not be explicitly stated, its interesting to have the ability to tweak some of these parameters in the generative model.

Some popular deep genrative models are as follows:

4.3.1. VAE

Variational autoencoder are an extension of the traditional autoencoder. For the sake of completeness, lets first discuss what an autoencoder is. Autoencoders are two-part models, one part is the encoder that encodes the input into a lower dimensional space and second part is the decoder that uses the lower dimensional representation of the input to output something that is approximately equivalent to the input. Autoencoders use reconstruction loss as the loss metric. An example can be autoencoder for images where the objective is to learn a lower dimensional representation of the image for compression tasks. This idea can also be extended to remove noise from the input, say that the training data consists of pairs of noisy and clean images. The noisy image can be used as an input to the autoencoder and the output can be compared with the clean image. Using pairs of such images for training can make the autoencoder a denoising autoencoder.

An autoencoder can not be used as is for the generative task since all it knows is to copy the input it has been fed. Instead of mapping our input to a fixed latent space vector like in the case of autoencoder, if we map the input to a normal latent space distribution (parameterized by mean and variance vectors), our model will achieve the capability of generating different samples based on tweaking the values of the latent space. A visual comparison can be seen in Figure 4.6.



(a) Autoencoder (b) Variational Autoencoder

Figure 4.6: Autoencoders

It is also intriguing to just change the value of the autoencoder's latent space and see if it is able to generate different samples that mimic the training data (but does not copy it). Unfortunately, since autoencoders learn a fixed latent space representation, they do not perform well when these fixed latent space values are changed. Variational autoencoders on the other hand are much more flexible given that they learn normal

distributions instead of fixed values for the latent space. Mathematically, variational autoencoders aim at estimating the actually density of the latent space conditioned on the input; i.e, *p(z|x)*.

$$l_i(\theta,\phi) = -\mathbb{E}_{z\sim q_\theta(z|x_i)}\left[\log p_\phi\left(x_i \mid z\right)\right] + \mathbb{KL}\left(q_\theta\left(z \mid x_i\right) \| p(z)\right) \tag{4.8}$$

However, it is an intractable problem due to the absence of a closed form solution to *p(x)*. Hence, an approximation to it is found using variational inference where *p(z|x)* is compared to a simpler gaussian distribution *q(z|x)* and minimize the KL Divergence between them and both reconstruction and regularization losses are taken into account. While using variational inference, minimizing the KL Divergence (as in equation 4.8) is equivalent to maximing an evidence lower bound (ELBO).

4.3.2. GENERATIVE ADVERSARIAL NETWORKS

Unlike Variational autoencoders, that try to maximize the evidence lower bound (ELBO), GANs implicitly define a probability function of the data. GANs leverage game theory to be able to do so, namely a two player minimax game. For this, in addition to the generator network that accepts a noise vector as input and generates samples similar to the data that it was trained on, there is also a discriminator network that tries to distinguish between real and fake samples. Generator aims to fool that discriminator, which means that it wants the discriminator to think that the generated image is real. The discriminator on the other hand wants to correctly tell which sample is a fake and which sample is real. If we think about the output of the Discriminator as probabilities, a probability of 0.5 is what we want to achieve which means that the generated samples are indistinguishable from the samples that the GAN is trained on. Figure 4.3.2 shows a GAN similar to the one originally proposed by Goodfellow *et al.* [70]. In the original paper the authors used binary cross entropy loss, so the discriminator is trained exactly like a logistic classifier.

$$\min_G \max_D L(G,D) = \mathbb{E}_{x\sim \mathrm{p}_{\mathrm{data}}(x)}[\log D(x)] + \mathbb{E}_{z\sim p(z)}[\log(1-D(G(z)))] \tag{4.9}$$

GAN essentially computes the Jensen Shannon divergence between the generated data distribution and the real data distribution. The generator wants to minimize the loss since that would mean that the discriminator is confused between real and generated samples. The discriminator, on the other hand, wants to maximize the loss since that would mean that the discriminator can successfully distinguish between real and generated samples.

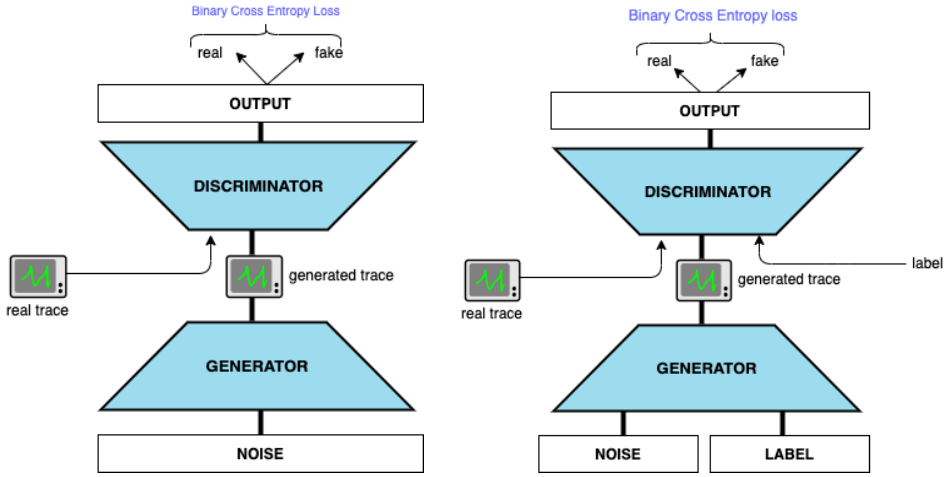
The original GAN architecture does not allow to input labels into the model and hence is unsupervised. This simpler architecture however has its limitations since the GAN can not be told about the type of samples it should generate. In order to do so, the label (y) must also a part of the loss function (as proposed by Mirza and Osindero [82]).

$$\min_G \max_D L(D,G) = \mathbb{E}_{x\sim \mathrm{p}_{\mathrm{data}}(x)}[\log D(x \mid y)] + \mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z \mid y)))] \tag{4.10}$$

The non cooperative game makes training GANs hard and unstable. The standard GAN loss function; i.e, the JS divergence is better than KL divergence since it is symmetric and is mathematically defined even if there is no overlap between the two probability density functions being compared. However, it has its own problems. Namely, the vanishing gradient problem wherein if the probability density functions don't have any overlap (when the distributions are disjoint), the loss value becomes a constant causing the gradients to vanish. Other than this, one of the biggest issues with GANs is when the generator gets stuck at outputting similar looking samples. This situation is known as mode collapse. Also, the discriminator gets good at discriminating very easily making the gradient vanish and preventing the generator from learning anything. To address these problems, a lot of work has been done to find better loss functions and better normalization techniques.

A number of different GAN loss functions have been proposed in order to address its well known issues. Some of the are:

1. Least Squares GAN (LSGAN) loss function [83] : LSGAN aims at increasing the quality of the generated samples by not limiting the use of discriminator's output. In equation 4.10, the discriminator's purpose is to distinguish between the real and fake samples, which is done using binary cross entropy loss. In LSGAN, however, we are not only concerned with the classification but also with how close or how far



(a) Generative adversarial network (b) Conditional Generative adversarial network

Figure 4.7: Generative adversarial networks

are the fake traces to the real ones. This can be seen in the loss function, as in equation 4.11.

$$\begin{aligned} \min_D L_{\text{LSGAN}}(D) &= \frac{1}{2}\mathrm{E}_{x\sim p_{\text{data}}(x)}\left[(D(x)-b)^2\right] + \frac{1}{2}\mathbb{E}_{\boldsymbol{z}\sim p_z(z)}\left[(D(G(z))-a)^2\right] \\ \min_G L_{\text{LSGAN}}(G) &= \frac{1}{2}\mathrm{E}_{z\sim p_z(z)}\left[(D(G(z))-c)^2\right] \end{aligned} \tag{4.11}$$

In equation 4.11, while training the discriminator b is set to 1 to signify real traces and a is set to 0 to signify fake traces . While training the generator c is set to 1 since the objective is to fool the discriminator .

2. WGAN loss function [84] : As mentioned before, the issue with JS Divergence is that when the distributions are disjoint the loss value becomes a constant causing the gradients to vanish. To address this issue Arjovsky *et al.* used the Wasserstein metric (also known as the earth movers distance) as a loss function for GANs. Wasserstein metric, is a concept borrowed from transport theory and can be interpreted as the minimum amount of work required to transform one probability distribution to another. However, in itself it is an intractable problem. Since Wasserstein distance is intractable, they use an approximation to it known as the KR duality. The WGAN's discriminator (known as the critic) tries to model a function that approximates the EM distance and not just distinguish the real samples from the generated ones. In its original implementation, the authors perform weight/bias clipping in a range [-c, c] so as to enforce a Lipschitz contraint on the critic. The loss function for WGAN is as follows:

$$L\left(p_r, p_g\right) = W\left(p_r, p_g\right) = \max_{w\in W}\mathbb{E}_{x\sim p_{\text{data}}x}\left[f_w(x)\right] - \mathbb{E}_{z\sim p_z(z)}\left[f_w\left(g_\theta(z)\right)\right] \tag{4.12}$$

The clipping method was criticized by the authors of the paper themselves and hence the gradient penalty version gained popularity. The main difference between basic GANs and Wasserstein GANs are:

- Loss function changes : As compared to the standard GAN loss, we remove the logarithmic function for WGANs.
- No sigmoid layer at the output: Since problem is now regression and not classification
- The Discriminator (critic) is trained more than the Generator. Usually, it is trained almost five times more.

3. WGAN-GP loss function [72]: The WGAN-GP comes from the family of Wasserstein GANs (WGAN). The objective of WGANs is to minimize the earth mover (EM) distance. The EM distance represents the level of dissimilarity between the distributions of the generated and real traces.Minimizing the EM distance leads to smoother gradients even when the generator outputs unsatisfactory traces. The WGAN's

Discriminator tries to model a function that approximates the EM distance and not just distinguish the real samples from the generated ones. The loss function for WGAN-GP is as follows:

$$L = \underbrace{\mathop{\mathbb{E}}_{\tilde{x}\sim\mathbb{P}_g}[D(\tilde{x})] - \mathop{\mathbb{E}}_{x\sim\mathbb{P}_r}[D(x)]}_{\text{Original critic loss}} + \underbrace{\lambda \mathop{\mathbb{E}}_{\hat{x}\sim\mathbb{P}_{\hat{x}}}\left[\left(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1\right)^2\right]}_{\text{The gradient penalty}} \tag{4.13}$$

Calculating the earth mover (EM) distance is an intractable problem and the Kantorovich-Rubinstein duality [85] can be used to make the problem simpler. The Kantorovich-Rubinstein duality is used to transform the EM distance minimization problem in order to find a least upper bound. The transformed loss function is required to satisfy K-Lipschitz continuity. The Lipschitz continuity limits how fast a function can change. In the original WGAN paper [84], the Lipschitz constraint is enforced by weight clipping. However, the weight clipping method is extremely sensitive to the clipping value hyperparameter and quite often reduces the network's ability to model complex functions. Instead, WGAN-GP adds a gradient penalty term to enforce the K-Lipschitz continuity as shown in Equation 4.13.

Despite the advancements in the GAN loss functions, training them is not always straight forward. Often, GANs suffer from *mode collapse* which means that the Generator outputs just a single (or a few) sample(s) that is good enough to fool the discriminator. Hence, losing its ability to generate a variety of samples. Another common problem with GAN is lack of convergence since with time, the discriminator's feedback deteriorates due to which the GAN's convergence is often a fleeting, rather than stable, state [86]. These two problems are also known as GAN's failure modes.It is important to use evaluation metrics that can actually identify such failures so that necessary changes can be made.

4.3.3. AUTOREGRESSIVE

Unlike GANs, Autoregressive models can compute exact likelihoods and is much stable to train as compared to GANs. However, this ability comes with a cost. Autoregressive models are usually much slower than GANs. Also, similar to GANs, autoregressive models do not have mechanism for learning a latent space representation.Mathematically, autoregressive models perform maximum likelihood estimation by decomposing the likelihood into a product of conditional distributions as follows:

$$p(\mathbf{x}) = \prod_{i=1}^{n} p\left(x_i \mid x_1, x_2, \ldots, x_{i-1}\right) = \prod_{i=1}^{n} p\left(x_i \mid \mathbf{x}_{<i}\right) \tag{4.14}$$

If the product of conditional distibution makes no conditional independence assumption then the model is called autoregressive. This means that the prediction x_i depends on $x_1, x_2, x_3 \ldots x_{i-1}$, in the same order. This means that, unlike GANs, maximing the likelihood of data is possible in the case of autoregressive models by minimization of the following objective function:

$$-\ln p(\boldsymbol{x}) = -\sum_{i}^{n} \ln p\left(x_i \mid x_1, \ldots, x_{i-1}\right) \tag{4.15}$$

What is evident is that for every new timestep, the entire sequence need to be evaluated so as to compute the prediction. Hence, sampling from them is innately sequential and slow. To make this process faster, the concept of causal convolutions was introduced by van den Oord *et al.* [74]. First, causal convolutions ensures that the model does not look in the future while making predictions. Second, using dilated convolutions predictions for an entire sequence can be calculated in a single forward pass in a very efficient manner. A lot of research in this domain uses RNNs (those introduced in Section 4.2). However, a shift towards Self-Attention [87] is on the rise after their immense success in NLP. Autoregressive models will be compared with the other generative models in Section 4.3.6.

4.3.4. FLOW MODELS

Just like Autoregressive model flow models can compute exact likelihoods. However, they need a lot of layers and hence have a lot of trainable parameters. Flow models are also known for taking extremely long to train.

$$p_X(x) = p_Z\left(f^{-1}(x)\right) \left| \det\left(\frac{\partial f^{-1}(x)}{\partial x}\right)\right. \tag{4.16}$$

There are two main steps in a flow models: First, we start with a simple Gaussian distribution. Next, we transform the simple Gaussian distribution into a complex distribution using an invertible function. The process of transforming the simple distribution into a complex distribution using a chain of transformation is known as a normalizing flow. The prerequisite for this transformation is that the transformations must be (easily) invertible. This invertible requirement mandates that the input dimension of the transformation must be equal to the output dimension, making deep models very difficult to train. As an idea, flow models have also been coupled with autoregressive models [88]. Sadly, despite the advantage of exact likelihood computation as opposed to GANs, Flow models are very slow to train. Flow models lack in terms of parameter efficiency and take an order of magnitude more GPU days to train as compared to progressive GAN models [89].

4.3.5. ENERGY BASED MODELS

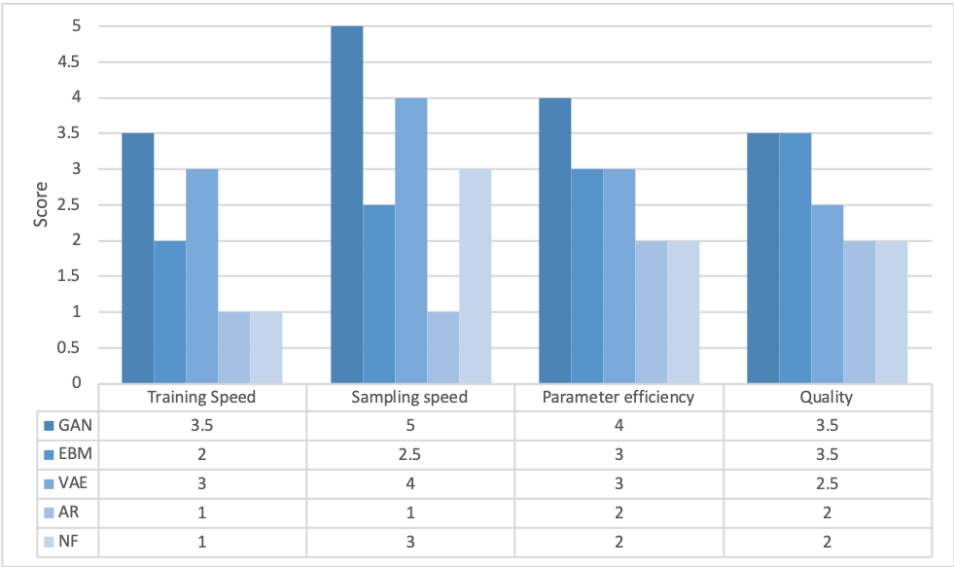
In energy based models the prerogative is to find an energy function which can equate realistic samples with low energy and unrealistic samples with high energy. The basis of these models is that any probability distribution function $p(\boldsymbol{x})$ for $\boldsymbol{x} \in \mathbb{R}^D$ can be written as

$$p(\boldsymbol{x}) = \frac{e^{-E(\boldsymbol{x})}}{\int_{\tilde{\boldsymbol{x}} \in \mathcal{X}} e^{-E(\tilde{\boldsymbol{x}})}} \tag{4.17}$$

However, computing the denominator of equation 4.17 is infeasible. Even with various computational tricks, energy based models are slow at training and sampling time. Some energy based models are capable of generating very high quality samples. Sampling from energy based models is usually time consuming since they employ Markov Chain Monte Carlo using Langevin Dynamics. This computational issue [90] is one of the reasons why such energy-based models have not gained a lot of popularity, as compared to its counterparts, due to

4.3.6. COMPARISON

Table 4.8 is based on an extensive comparative review of deep generative models by Bond-Taylor *et al.* [91]. Every parameter score is out of 5 and is a combination of all the architectures compared in the review. Hence, specific architecture might score higher (or lower) than the combined average. For example, in case of GANs DCGAN ranks 5 on the *'training speed'* since the training time is very less as compared to other deep generative models and StyleGAN ranks 3 since it takes longer than a few other deep generative models.



	Training Speed	Sampling speed	Parameter efficiency	Quality
GAN	3.5	5	4	3.5
EBM	2	2.5	3	3.5
VAE	3	4	3	2.5
AR	1	1	2	2
NF	1	3	2	2

Figure 4.8: Generative Model Comparison

This comparison was carried out for the computer vision domain and hence all the conclusion can not be transferred as is to any other domain. For example, they rate the AR models low on quality. However, it is well known that AR models have even been state-of-the-art models for TTS tasks [92]. Naturally, they have

seen a lot of application in audio and text generation. However, they are extremely slow and definitely not the preferred option for real-time tasks [77] and hence are rated very low on training and sampling speed.

For this work, the generative model has the following requirements:

1. Training speed: Most model training for this work was done on a modest MacBook Pro with an i5-8259U coffee lake processor and 8GB of LPDDR3 SDRAM, with the exception of a few large models that were trained on Google colab (with NVidia's Tesla P100 GPU). Many models were practically infeasible to train on the limited compute resources and it is reasonable to say that most end-users of this framework will also not have GPU clusters at their disposal. Hence, high training speed is important not just for practicality but also since it reduces the framework execution time, aiding in the potential speed-up over existing leakage evaluation frameworks.

2. Sampling speed: Similar to training speed, high sampling speed is important because of the aforementioned reasons. High quality results with a very slow sampling speed would obviate the need of the entire framework since it would negatively impact the potential speed up.

3. Parameter efficiency: Parameter efficiency plays a major role in the training speed. Models with millions of trainable parameters with a very minimal increase in the generated sample quality are not ideal for this framework.

4. Quality: Good quality generated samples are an absolute must. However, it must be noted that if better quality comes at the cost of extremely high training/sampling speed or very low parameter efficiency then it is not an ideal option.

Keeping all the metrics in mind, GANs and VAEs seem to be very promising generative models for our use case. In Chapter 5, GANs will be discussed in detail with the objective of power trace generation and a focus on possible label options. In Chapter 6, VAE results using those label options will be compared with the GAN results.

4.4. BUILDING AND TRAINING NETWORKS

Building and training machine learning models is a five step process. First, an architecture is defined. The initial model definition is usually based on the application domain and empirically proven techniques. For example, CNNs perform extremely well for computer vision related tasks and RNNs are the go-to for text data / NLP. Next, the loss function and optimizer is decided and the data is processed so as to make it ready for model training. The model is then trained using backpropogation and the training metrics on a validation dataset (like loss value, prediction accuracy etcetera) are analyzed. This training process can either go on for a predetermined set of epochs or can be stopped on the basis of the training metrics. Finally, the trained model can then be evaluated on a test dataset before it can be deployed. The final step can also be split into two steps with one step involving testing and cross validation and the other step involving deployment.

4.4.1. WEIGHT INITIALIZATION

The first step of the training process is to initialize the weights and the biases of the network. This initialization can be done using constant values or random values. However, this initialization step has been shown to have substantial implications on the training process and the network performance. There is also a detailed study on the importance of this step for deep neural networks for side channel analysis [93]. However, even the authors of this study could not decide on a particular initializer that works the best across all types of power traces. A lot of initialization methods have been proposed with the objective of stabilizing the gradients during backpropogation. Some popular methods are:

1. *random normal*: initializing weights with a normal distribution

2. *truncated normal* : initializing weights from a normal distribution but values more than two standard deviation from the mean are discarded and redrawn

3. *he normal* : truncated normal initialization with mean 0 and standard deviation= sqrt(2/fan_in + fan_out)

4. *glorot normal* : truncated normal initialization with mean 0 and standard deviation= sqrt(2/fan_in).

All these methods also have their corresponding uniform counterparts, for example in glorot uniform intialization the weights are initialized using a uniform initialization in the range [-limit,limit] where limit = sqrt(6/fan_in). The fan_in and fan_out, as might be evident, corresponds to the number of inputs and the number of outputs of the weight tensor.

4.4.2. LOSS FUNCTION

Much like other machine learning problems, the loss function or the cost function plays a pivotal role in model training. Chronologically: a model is defined, the cost function is chosen and then that cost function is minimized using gradient descent [94].

$$\mathscr{L}_{0-1}(y,t) = \begin{cases} 0 & \text{if } y = t \\ 1 & \text{otherwise} \end{cases} \tag{4.18}$$

Lets now discuss what loss function should be chosen and why. For the purpose of this discussion lets consider a binary classification problem. A simple loss function could be 0-1 loss, which means that only when the output of a classifier is equal to a specific value the output is one otherwise it is zero (like in equation 4.18). The problem with this approach will be that the gradient will stay zero until the boundary of zero and one is reached, making gradient descent ineffective.

$$\begin{aligned} y &= w^\top x + b \\ \mathscr{L}_{\text{SE}}(y,t) &= \frac{1}{2}(y-t)^2 \end{aligned} \tag{4.19}$$

Next, a squared error loss function can be tried like in equation 4.20. But in this case, if the model output's a value much larger than 1, the error will be huge. This is somehow not intuitive, since the model is very certain of the category being not zero but still gets a large loss value. Hence, we need to squish the model's output between zero and one so that the aforementioned scenario does not happen.

$$\begin{aligned} z &= w^\top x + b \\ y &= \sigma(z) \\ \mathscr{L}_{\text{SE}}(y,t) &= \frac{1}{2}(y-t)^2 \end{aligned} \tag{4.20}$$

To squish the output of the classifier between zero and one, we can use the logistic function on the model's output. This way even if we use the squared error loss function the value of loss does not increase when the model gets confident of a particular category. But the catch here is that since we are squishing the output of the model in the range of zero and one, even very wrongly classified instances will have minimal loss values making the gradient descent process tougher since the difference of loss value between correct and incorrect will be minuscule.

$$\mathscr{L}_{\text{CE}}(y,t) = -t\log y - (1-t)\log 1 - y \tag{4.21}$$

As a solution to this, cross entropy loss as in equation 4.21 can be introduced. Cross entropy solves the previous problem but it has one of its own. Namely, there can be numerical instability when the value of y reaches zero since $log(0)$ is undefined. To avoid this issue we can use the logistic cross entropy loss where we employ the logistic function.

$$\mathscr{L}_{\text{LCE}}(z,t) = \mathscr{L}_{\text{CE}}(\sigma(z),t) = t\log\left(1+e^{-z}\right) + (1-t)\log\left(1+e^{z}\right) \tag{4.22}$$

The logistic cross entropy loss can be seen in equation 4.22. Finally, to extend this understanding for binary classification to multi-class classification, a softmax function must be employed at the output of the model. This softmax function converts the output of our model to probabilities over the multiple classes.

$$y_k = \text{softmax}(z_1,\ldots,z_K)_k = \frac{e^{z_k}}{\sum_{k'} e^{z_{k'}}} \tag{4.23}$$

The binary cross entropy loss, as in equation 4.22, can then be extended to multiclass logistic regression as follows.

$$\begin{aligned} z &= Wx + b \\ y &= \text{softmax}(z) \\ \mathscr{L}_{\text{CE}} &= -t^\top(\log y) \end{aligned} \tag{4.24}$$

As the name suggests, the multi-class logistic regression is employed in multi-class classifiers. With this high level idea of how loss functions are chosen, a HW classifier (which is particularly common in side channel analysis) can be implemented. As will be explained in Chapter 5 and 6, a multi-class classifier will be useful for evaluating the quality of the generated traces. Depending on the dataset under consideration, different architectures perform differently. From a side channel perspective, CNN based classifiers have been very popular [95] [96] [97]. In this work we use them for the purpose of classifying real and generated traces. The accuracy of this classification then serves as an evaluation metric for the methodology.

An example classifier can be seen in Figure 4.9, here a trace is fed into one of the convolutional layers of the classifier. The classifier consists of multiple convolutional and dense layers. It outputs the probability of the trace belonging to one of the 9 HW classes. As can be seen in the final dense layer, a softmax activation function is used that squishes the output values of the dense layer in the 0 to 1 range (as explained in Section 4.4.2).

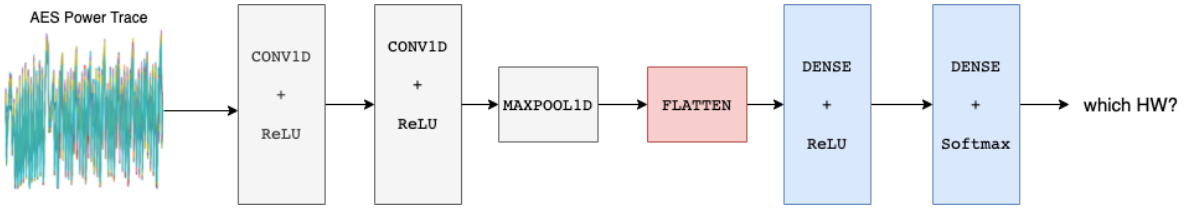


Figure 4.9: Hamming weight classifier

Often, as means of reducing the computational requirements of such classifiers, dimensionality reduction is employed. This aim is to transform data from a high dimensional space to a lower dimension. This transformation can be linear and non linear. Following is one example of each:

1. PCA : PCA is a linear dimensionality reduction method that involves mapping the high dimension data into a lower dimension. This is done by calculating eigenvalues and eigen vectors corresponding to the correlation matrix and then only using the first few (principle) components. This reduces the higher dimension data into few dimensions (as big as the number of chosen components). Depending on the amount of variance explained by these eigen vectors, the success of the dimensionality reduction is determined [98].

2. tSNE : t-distributed Stochastic Neighbor Embedding is a non linear technique for visualizing higher dimensional data [99]. It aims at reducing the Kullback-Leibler divergence between the higher dimensional data and the lower dimension embedding. Often, the data is pre-processed using PCA and then tSNE is applied.

Many other dimensionality methods exists. From a side channel analysis perspective, even autoencoders (as have been explained detail in Section 4.3.1) are fairly popular [95].

4.4.3. TYPES OF LEARNING

Now that we know different types of models and the methodology behind training them, let me now discuss the various ways a machine learning model learns to do its assigned task. This decides the data requirements for model training and depending on these requirements and learning type we can classify the models into different training schemes:

1. Supervised Learning: In supervised learning, we not only provide the model with a training sample but also with a corresponding label. This type of learning is used for the tasks like:

   - Regression : This is the kind of problem where our objective is to find *How much?* or *How Many?* [4]. An example can be to find How much will it snow in a particular month given a few Meteorological parameters. In this case the *Meteorological parameters* are the data and the *amount of snow* is the label.
   - Classification : This is the kind of problem where our objective is to find *Which category?*. Say that we have images of cats and dogs and our objective is to train a model that can differentiate between the two types. In this case our problem will be a classification problem.

- Tagging : Similar to classification, a more complex computer vision problem can be image tagging where given an image we need the ability to classify all types of animals in an image or all types of objects in an image. This problem can also be extended to finding tags (keywords) of certain articles/ research papers.

2. Unsupervised Learning: In unsupervised Learning, we do not have any label corresponding to the data samples in our dataset. In such a case, our models are solely dependent on the data to solve the task at hand. An example can be to group different types of similar looking training samples together, this problem is known as clustering. A real world example is using clustering on railway track data, where the objective is to group together risky zones on the basis of number of train slide accidents [100].

3. Reinforcement Learning: This learning technique is for machine learning models that take actions on the basis of the stimuli in their immediate environments. The model keeps on observing the environment and chooses what action should be performed for every stimulus. This action is then either reprimanded or rewarded and on the basis of these rewards (reprimand can be represented as a negative reward), the model decided how to update its weight and biases to perform better. An example for deep reinforcement learning is grasp for stacking. As the name suggests, the aim is to make a robot able to grasp and stack objects [101].

4. Transfer learning: This form of learning happens when a model trained for one task is used as a beginning model for another task. This idealogy is based on the premise that a trained model already contains high level information that can then be fine-tuned to cater to the needs of another application domain. Often, the training step in transfer learning involves freezing the learned parameters of the trained model and only training the newly added layers. As a example, lets suppose that our objective is to build a binary image classifier. One option is to build and train a model from scratch. If a limited dataset is available, the model is very likely to overfit on the training data. It is advised that transfer learning should be used in such cases. First, a pretrained classifier model (for example a CIFAR-10 [102] classifier) is taken. Next the 10 neuron output is fed into a Dense layer that outputs 2 values. Now, while training, the weight and biases of the pretrained classifier will be frozen but the remaining layers of the model will be trained. This process is called 'transfer' learning since it involves transferring the information learned by a particular model for a specific task to another model for a different (but related) task. This methodology works since the information learned for one task can still be useful for the other task, for example a classifier for trucks might be useful for classifying cars.

4.4.4. BACKPROPOGATION AND AUTOMATIC DIFFERENTIATION

The backpropogation technique is a version of chain rule where the aim is to share the computations that are repeated, wherever possible. Formally, the chain rule can be represented as follows:

$$\frac{\mathrm{d}}{\mathrm{d}t} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t} \tag{4.25}$$

In backpropogation the derivative is calculated in reverse order and then chain rule is used to calculate the remaining derivatives, hence going from the output layer to the hidden layers in a so-called computation graph. The computation graph for equation 4.26 can be seen in Figure 4.10.

$$\begin{aligned} z &= wx + b \\ y &= \sigma(z) \\ \mathscr{L} &= \frac{1}{2}(y-t)^2 \\ \mathscr{R} &= \frac{1}{2}w^2 \\ \mathscr{L}_{\text{reg}} &= \mathscr{L} + \lambda\mathscr{R} \end{aligned} \tag{4.26}$$

Hence, while backpropogating for this equation, the derivative of L_{reg} is first calculated with itself (which will be 1), then the derivative of L_{reg} with respect to $\mathscr{R}$ is calculated followed by derivative of L_{reg} with respect to L. This process will continue till the derivative of L_{reg} with respect to b and w (which are the weights and the biases) has been calculated. During this entire process since there is dependency between variables, chain rule will be employed to reuse the computation that has been already done.

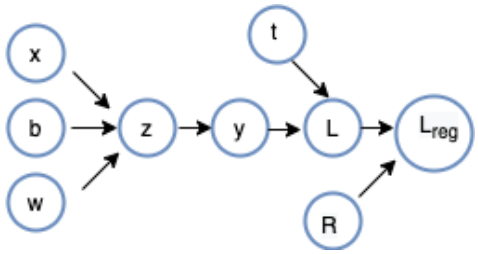


Figure 4.10: Backpropogation Example

On a software level, this is done using the automatic differentiation package built into libraries like TensorFlow and PyTorch. In Tensorflow-1.0, the graph is built node by node by the user before calling `session.run()` whereas in PyTorch this is done implicity (and is known as eager execution). Tensorflow-2.0 removes the concept of `session.run()` and takes after PyTorch by inclusion of eager execution. Also, the `tf.function()` decorator has the ability to generate python indepedent dataflow graphs which are much faster than eager execution. Let's now discuss the automatic differentiation package of PyTorch, called *autograd* in a little more detail.

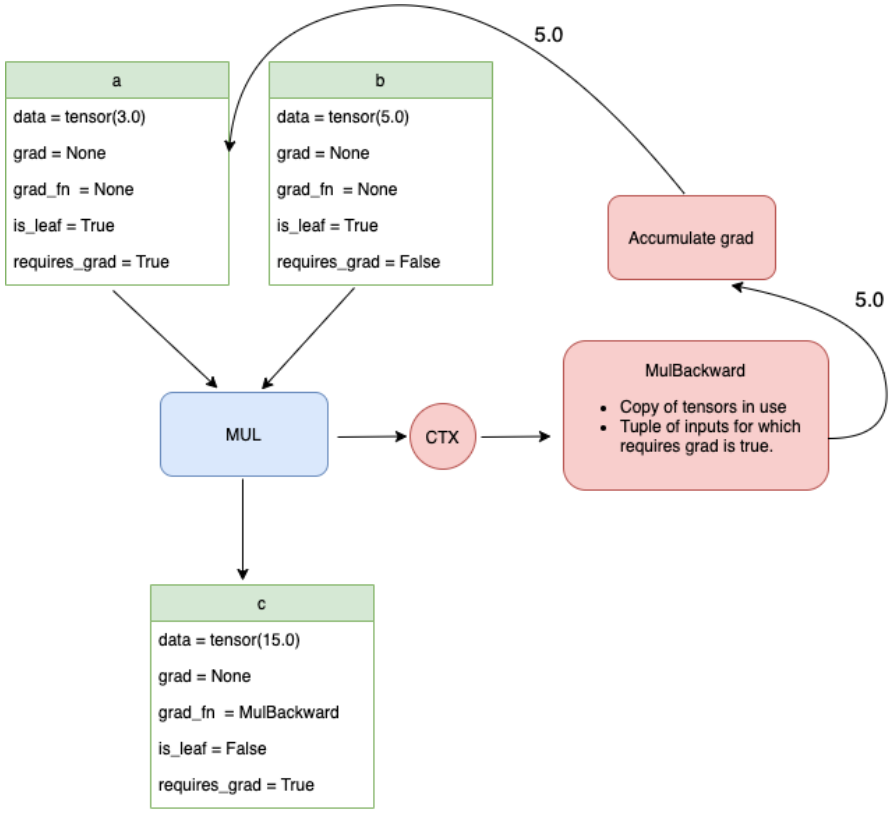


Figure 4.11: Autograd : PyTorch

As can be seen in Figure 4.11, the process of calculating gradients for a simple multiplication example. Tensor *a* and *b* is marked as a leaf node since no other tensor is used to calculate either of them. However, since tensor *c* is calculated using tensor *a* and *b*, it is not a leaf node. Now, depending on the *requires_grad* attribute the context manager maintains a list of tensors that, when processed, can yield the differentiation result.

4.4.5. OPTIMIZATION ALGORITHMS

Our objective while training deep learning models is to optimize the loss function. This is done using gradient based optimization. Here, we reduce the value of the loss function by calculating its gradient with respect to

the parameters of the network and then updating the parameters using backpropogation. A toy example of the this process can be seen in Figure 4.12. This technique was first introduced in [103]. Mathematically, this process can be explained using the Jacobian and the Hessian matrices. The Jacobian matrix will contain the partial derivatives of the loss function with respect to all network parameters and the Hessian matrix will contain their second derivatives, whose eigenvalues can be used to determine local minima, maxima or saddle points.

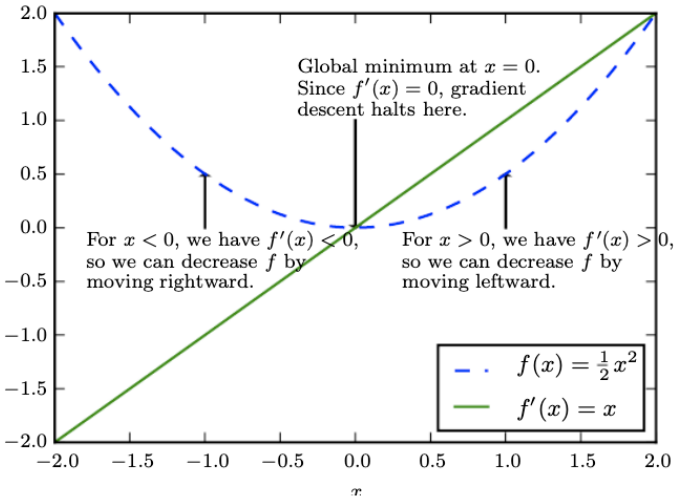


Figure 4.12: Gradients while optimizing [3]

In deep learning, the problems are not convex, which gives rise to issues like local minimas, saddle points and vanishing gradients [4]. Lets now discuss a few popular optimization algorithms. Gradient descent is a cornerstone of machine learning algorithms and is also a precursor to various advanced optimization algorithms that we see in production in present day deep learning models. The concept behind the optimization process in neural networks is pretty straight forward. Altogether there are four steps.

1. Do a forward pass through the model. If for example we have a binary classification problem for images, the input to the model would be an image and the output of the model, depending on the architecture, should be a value in the range 0 and 1.

2. Depending on the loss function we choose, a loss value will be calculated between the expected value of the image and the actual value that the model outputs. For example if we want to classify a particular image as cancerous or non-cancerous, we will set the label of cancerous images as 1 and that of non-cancerous images as 0. After a forward pass of the image, the model will output a value: say 0.4. If the image was non-cancerous and the loss value will be $0.5 * (0-0.4)^2$ which is 0.08. Note that if the model's output was 0.99 for the same non-cancerous image, the loss value will be $0.5 * (0-0.99)^2$ which is 0.49. This is expected behaviour since the model should output a high loss value for such strongly miss-classified images.

3. Now that we have the loss value, the next step is to carry out backpropogation as described in subsection 4.4.4.

4. Finally, we get to the point of updating weights and biases and here the optimization algorithms come into play.

Mathematically, gradient descent can be represented as follows. Say that $f_i(x)$ is the loss value for i^{th} sample in a dataset with a total of n samples. The loss function in this case will be:

$$f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{x})$$

and the gradient of the loss function will be:

$$\nabla f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\mathbf{x})$$

the parameters of the model (θ) can then be updated (θ') using this gradient as follows (η is the learning rate):

$$\theta' = \theta - \eta \nabla f(\mathbf{x})$$

It is evident that this process is computationally expensive and scales linearly with the size of the dataset. Since this is not desirable, we have another (more popular) variant of gradient descent called stochastic gradient descent wherein we randomly choose samples from the dataset and calculate the gradient just on the basis of those samples. This process is computationally less intensive but is known to be extremely noisy. Also it is still not the most data efficient since it does not make full use of vectorization and caches in CPUs and GPUs. As a solution to this, a middle ground between gradient descent and stochastic gradient descent was proposed and was named batch gradient descent. Here the dataset is divided into multiple batches. Each batch contains training samples is a random order. Each training sample only appears once every epoch.

Now lets discuss more advanced optimization techniques, which take after EWMA. EWMA stands for exponentially weighted moving average. Say for example that a particular value after a few time steps in a particular time series needs to be predicted. Remebering all the values before the present time step can be extremely memory intensive for large time series, here the concept of EWMA comes to the rescue. The idea is to calculate weighted sums from the second to the last timestep. This way, when we have to predict a value, it only depends on EWMA of the last time step. Mathematically, for value y_t at time t and $0 \leq \rho \leq 1$, the value of Exponentially weighted moving average (EWMA) : $S_t = \big(\rho S_{t-1}\big) + (1-\rho)y_t$. Optimization algorithms leverage EWMA to update the weights and biases of the model in the following way:

- SGD with Momentum: The aim is to smooth the noisy gradients using EWMA. Just using gradient descent can slowly ossicalte towards the minimum, preventing us from using a larger learning rate. When usimg SGD with momentum, we compute moving averages as in equation 4.27.

$$\begin{aligned} v_i &= \rho v_{i-1} + (1-\rho)\nabla_\theta \\ \theta' &= \theta - \epsilon v_i \end{aligned} \tag{4.27}$$

- RMSProp: The aim is to smooth the zero centered variance of noisy gradients using EWMA. This can be seen in equation 4.28.

$$\begin{aligned} r_i &= \rho r_{i-1} + (1-\rho)\nabla_\theta^2 \\ \theta' &= \theta - \epsilon \frac{\nabla_\theta}{\sqrt{r_i}} \end{aligned} \tag{4.28}$$

- ADAM:This is a combination of momentum and RMSProp and can be seen in equation 4.29.

$$\begin{aligned} v_i &= \rho_1 v_{i-1} + \big(1-\rho_1\big)\nabla_\theta, \quad & r_i &= \rho_2 r_{i-1} + \big(1-\rho_2\big)\nabla_\theta^2 \\ \hat{v}_i &= \frac{v_i}{\big(1-\rho_1^i\big)}, & \hat{r}_i &= \frac{r_i}{\big(1-\rho_2^i\big)} \\ \theta' &= \theta - \epsilon \frac{\hat{v}_i}{\sqrt{\hat{r}_i}} \end{aligned} \tag{4.29}$$

Even though exhaustive comparison studies on the efficacy of various optimizers exist [104], there is no ideal optimizer that can be used across all application domains mainly because of the hyperparameter tuning that it associated with them.

4.4.6. STANDARDIZATION AND NORMALIZATION

Standardizing and Normalizing features has a big impact on the optimization process and the training time. As defined by Huang *et al.* [105], normalization is the process of applying a transformation on a dataset to ensure that the dataset obeys certain statistical properties. Not normalizing features causes different features to have a very different range of values. This in turn may cause the gradients to oscillate back and forth, increasing the convergence time . After standardizing / normalizing the data, all features are in a similar range and the convergence time reduces.

A few such techniques are as follows:

1. MinMaxScaler [106]: This scaling technique, also known as min-max normalization, scales each feature x_i by subtracting the minimum value of that feature and dividing by the range of that feature before being scaled to the desired range. Equation 4.30 lists these operations.

$$\begin{aligned} x_i &= x_i - min(x_i) \\ x_{std} &= \frac{x_i}{max(x_i) - min(x_i)} \\ x_{scaled} &= x_{std} * (max - min) + min \end{aligned} \tag{4.30}$$

2. StandardScaler [107]: This method is also known as z-score normalization, each feature is standardized by removing the mean and then scaling to unit variance, as in equation 4.31.

$$\begin{aligned} x_{mean} &= \frac{1}{m}\sum_{i=1}^{m} x_i \\ x_{std} &= \sqrt{\frac{1}{m}\sum_{i=1}^{m} (x_i - x_{mean})^2} \\ x_i &= \frac{x_i - x_{mean}}{x_{std}} \end{aligned} \tag{4.31}$$

3. MaxAbsScaler [108]: The max absolute scaler simply scales each feature by its maximum value. This means that the maximum value for every feature ends up being 1.

4. RobustScaler [109]: This normalization technique is robust against outliers, as the name suggests. Each feature is scaled using the median and the inter quartile range. This means that the scaling process can be successful in case a particular feature has outlier. However, this technique is not completely resilient to outliers.

$$x_i = \frac{x_i - x_{median}}{x_{0.75} - x_{0.25}} \tag{4.32}$$

5. Batch normalization [110]: This normalization process is also essential for the hidden features of deep neural networks. This is done using batch normalization, wherein for every layer mean and variance are calculated and then used to normalize the layer's output. Batch normalization is required to check the problem of exploding gradients which usually occurs due to very large activation values of certain layers of the models. Intuitively, lets assume that all layers of a model are books and are stacked one on top of each other. Let us also assume that the task at hand is to carry the books from one place to the other (analogy for training the model). If while carrying the books we are struck by a gust of wind, the books at the upper level will shift a bit, making the pile unstable. This process will continue till the pile collapses. This process is termed as covariate shift. To make sure that the pile does not collapse, we must maintain the distribution of weight on the pile. Batch normalization does exactly this. For every batch, the input is scaled by a parameter γ and shifted by a parameter β [111]. Mathematically, it can be represented as follows.

$$\begin{aligned} \mu_{\mathscr{B}} &\leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \\ \sigma^2_{\mathscr{B}} &\leftarrow \frac{1}{m}\sum_{i=1}^{m} \left(x_i - \mu_{\mathscr{B}}\right)^2 \\ \widehat{x}_i &\leftarrow \frac{x_i - \mu_{\mathscr{B}}}{\sqrt{\sigma^2_{\mathscr{B}} + \epsilon}} \\ y_i &\leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \end{aligned} \tag{4.33}$$

   Here, for the mini batch $x_{1..m}$ we calculate the mean and the variance to normalize the values. However, since this process might be a bit too harsh for training, hence the trainable parameters γ and β are introduced, such that if the model wishes to, it can undo the normalization itself.

Batch normalization is a very popular normalization technique which is seen application in most state-of-the art architectures across domains. Scaling techniques, like the ones discussed in this Section, are equally popular. Many comparison reviews about the efficacy of these techniques exist [112]. As a rule of thumb, it is recommended to use *RobustScaler* when there are outliers in the data. Since that is not the case for power traces, it makes sense to experiment with different scaling techniques to empirically determine which technique works best.

4.4.7. REGULARIZATION

Deep neural networks can have parameters in the order of millions. Since training data, irrespective of the purpose of the network is limited, it is highly likely that the model learns to do well on the training data but performs very poorly on testing data. This situation is known as overfitting and it causes a model to not generalize well to unseen data. Popular solutions to overfitting are as follows:

1. Parameter norm: The first way to reduce the complexity of the model is to reduce the parametric weight of the model by scaling individual weights down using l1, l2 norm of the total weights.

2. Early stopping: Overfitting can also be caused by training the model for extended number of epochs. This means that the model has adjusted its weights way too many times on the basis of a particular dataset. Causing it to perform very well during training but poorly during validation and testing. To make sure this situation does not arise, we can introduce early stopping. In Keras, this option can be passed into the *model.fit()* API as a callback. In Pytorch, this can be done using a number of plugins like *Ignite* and *lightning*.

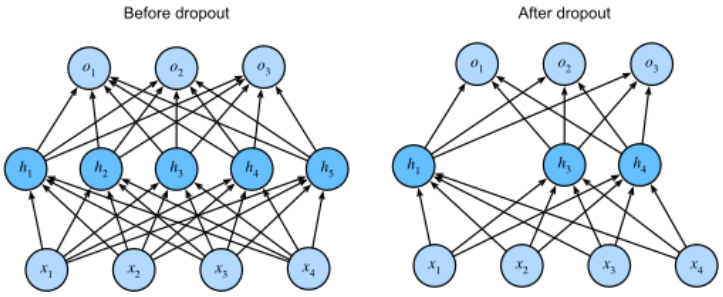


Figure 4.13: Dropout [4]

3. Dropout: Srivastava *et al.* [113] proposed this regularization technique of randomly pruning nodes, yielding many different subsets of a particular network. By training these sub-networks, a final network can be produced by using all nodes. However, this networks exhibits far lower error rates than an equivalent network trained as a single unit. This is because while training, the node availability is highly unpredictable and each node can no longer rely on other nodes correcting them. This makes the network much more robust and enhances its ability to generalize.

4.4.8. HYPERPARAMETER TUNING

Hyperparameters are very important components of the learning process and they indeed have a lot of influence over how well the trained model performs. Simply put, hyperparameters are the parameters of the model that are chosen before hand and are not updated during backpropogation in the training loop. These hyperparameters are tuned using a validation dataset.

Once the model has been defined, we need to decide the *learning rate*, *batch size* and the *number of epochs*. Finding optimal hyperparameters for every model is not a trivial process. However, there are a few methods for the task, like:

1. Manually Searching: The simplest method to find optimal hyperparameters is to manually change them for different training runs and choosing the ones that give the most validation accuracy.

2. Grid Searching: Manually choosing hyperparameters can be cumbersome and hence we can loop over a continuous range of hyperparameter values to find the one with best validation accuracy.

3. Random search: Instead of searching the hyperparameters over a continuous range of values, we can randomly pick hyperparameter values in a range to find the one with best validation accuracy.

Hyperparameter tuning gets much tricky for generative deep learning since formulating the validation accuracy is a complex task. For computer vision and audio tasks, researchers use services like Amazon Mechanical Turk to compare the generated samples with actual samples. As far as generative deep learning for hardware security is concerned, we choose to depend more on side channel analysis metrics like guessing entropy and success rate rather than other machine learning metrics, as suggested in this in-depth study [57].

4.4.9. PYTORCH AND KERAS

Now that all details about building and training networks have been explained, the mechanism to build them must be discussed. For different programming languages many machine learning frameworks exist. For C++, there is Caffe [114]. For Java there is DL4J [115]. However, in this work only python deep learning libraries are used; which are PyTorch [64] and Tensorflow [116] (with Keras [65] interface). Both of them are internally implemented in C/C++ instead of pure python due to obvious performance benefits.

Table 4.1: PyTorch and Keras APIs

Task	PyTorch API	TF2.0 / Keras API
Making the dataset	Subclassing the Dataset class and overriding magic methods to return samples. Finally, using DataLoader to split data is batches	Either using the TF dataset class or passing the data as arguments to the *model.fit* API.
Build Model	Subclassing from the Module class and overriding the forward method.	Either subclassing the Model class or using *model.compile* and specifying the input and output values (and shapes).
Using loss function and optimizer	Instantiating chosen loss function and optimizer.	Specifying as arguments in the *model.compile* API
Training	Manually looping over the dataloader, computing loss (*loss.backward*) and taking optimizer steps (*optimizer.step*)	Using model.fit API
Logging	Manual log statements in the train loop conditioned on the epoch or batch count.	Specifying metrics in *model.compile* and verbosity in *model.fit* API. Can also be done using callbacks.
Saving	Using the *torch.save* API	Using the *model.save* API or specifying *ModelCheckpoint* as a callback.

Table 4.1 list the main ML operations and their corresponding Pytorch and Keras APIs. As a refresher, every machine learning model has a few tasks associate with it . First, is to process the data and make a train, validation and test dataset. Second, is to build the model. Third, is to decide the loss function and optimizer, depending on the task that the model is supposed to carry out as well as the data that it is being trained on. Fourth, is to train the model and log its performance. Finally, the last task is to deploy/save the trained model.

5

METHODOLOGY

This chapter presents the methodology for training GANs to generate power traces that can be used for security analysis against power attacks. Firstly, Section 5.1 briefly explains the related work that has already been done in the context of GANs for side channel analysis. Next Section 5.2 provides a comparison between the task of generating images and generating power traces as most of the research for GANs has been dedicated to computer vision tasks. Followed by lists of the evaluation metrics used for assessing the GAN's performance in Section 5.3. Then Sections 5.4 and 5.5 highlight the label options and GANs different architectures, respectively. Finally, Section 5.6 tries to highlight the impact of the hyperparameters on GAN performance.

5.1. RELATED WORK

Most work in the field of generative deep learning has be been dedicated to images. Although very intuitive, those models can not be applied as is to the task of generating power traces. Having said that, recently quite a few researchers have ventured into audio waveform generation using generative deep learning. Notwithstanding a few caveats, these models are much inline with our objective.

The field of using generative models for hardware security tasks is not unventured. Wang *et al.* [117] recently demonstrated how generated traces can be used to enhance side channel attacks. However, in their own words, their study was aimed as a proof of concept and not a robust methodology. The four main issues with their design is as follows:

First, in their initial experiments they use a very limited training set of 500 traces. Details about the architecture are not completely disclosed, but for a dense GAN network with a 100 dimension latent vector, just 500 traces seem pretty small. Such a small training set is proven to generate substandard samples unless differential augmentation is used [118]. Second, they only use dense generator and discriminator networks. The problem with dense models is that the number of parameters quite easily go out of hand once the input/output size of the dense layers are increased. Third, they limited their study to only a few label options, namely least significant bit and Hamming weight of the SBOX output. In our study, we find how different label options can perform extremely well than by just using the LSB and Hamming weight. Finally, they use the basic conditional GAN loss (proposed by Mirza and Osindero [82]) as explained in Section 4.3.2. The issue with that loss formulation is that it is known to make generator stuck amongst a few categories, hence forgetting about the remaining categories. As explained by Odena *et al.* [119], it is quite a potent issue in GANs which can be resolved by making a few architectural changes.

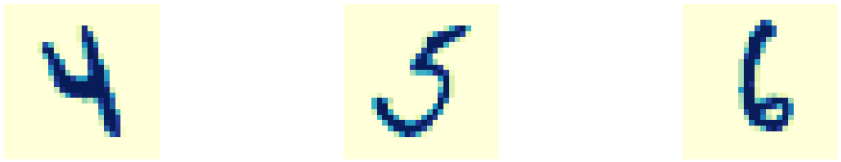
In [120], the author uses GANs as a dataset augmentation technique for side channel analysis, similar to [57]. The author proposes the use of nine convolutional unsupervised GANs (one for each hamming weight). In addition to the GANs, a supervised classifier model is trained that is expected to ascertain that the GAN generated traces actually belong the correct Hamming weight category. The impact of class distribution on the performance of the side channel attacks as well as model performance is assessed and it is concluded that class imbalance causes a serious detrimental impact on both.To make the per hamming weight training possible, meaning that to segregate the traces on the basis of the leakage model, the proposed methodology must make use of the known key and the plaintext in the data cleaning step. Hence, it was proposed as a data augmentation technique, in the profiled side channel attacks. The author concludes that it is not sensible to carry out dataset augmentation with GANs since they require more training data to perform well than is

required to make good quality templates.

5.2. COMPARISON WITH COMPUTER VISION

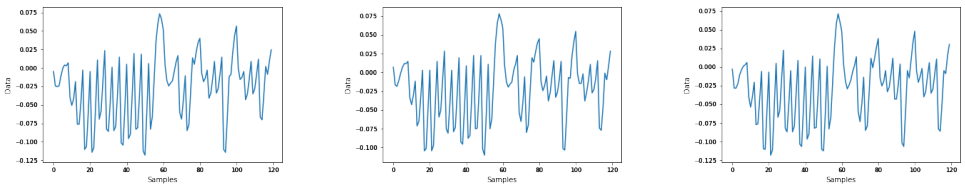
Since most reference GAN models belong to the computer vision domain. As a first step, it is important to compare the training data of such computer vision models and power traces. Figure 5.1 shows training images from the MNIST [121] dataset corresponding to a few digit classes. Figure 5.2 shows power traces corresponding to different hamming weight classes. What is particularly evident is that the MNIST can easily be distinguished and the power traces can not.

(a) Number 1 (b) Number 2 (c) Number 3

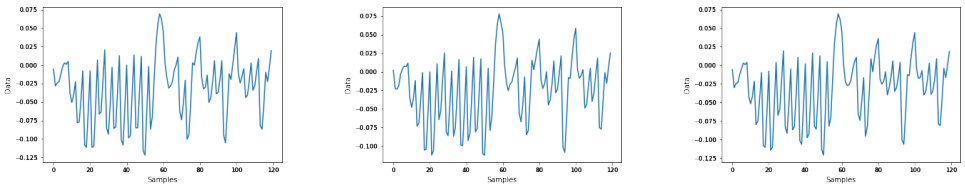


(d) Number 4 (e) Number 5 (f) Number 6

Figure 5.1: MNIST images corresponding to different classes



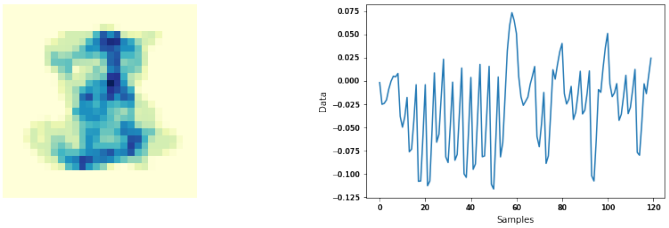
(a) HW 1 (b) HW 2 (c) HW 3



(d) HW 4 (e) HW 5 (f) HW 6

Figure 5.2: Power traces corresponding to different *mode(HW)* classes

Although quite evident, it is also important to mention that the average of training samples corresponding to these classes have very different visual properties. In Figure 5.3a, the average of MNIST classes produce an image that clearly does not belong to any of the classes. The same is not true for the power traces, the average of all HW classes produce a power trace that looks like a power trace corresponding to any other label (as in Figure 5.3b). The key takeaway from this is that the average of all power traces can be used to guide a generative model to generate power traces. In situations when the GAN outputs power traces that look particularly different from the set of real traces, correction can be done using a model of the different features of the power trace.



(a) Average image of MNIST classes (b) Average trace of HW classes

Figure 5.3: Average of all classes

To give an exact explanation of the same, consider that the training data is scaled using any scaling technique and the feature wise statistics are calculated. For example if *StandardScaler* [107] is used, the mean and the variance corresponding to all time samples in the power trace will be calculated. After GAN training, when the GAN is expected to generate power traces, these statistics can be used to correct the generated traces. In sklearn [122], these two steps are done using the *fit_transform()* and the *inverse_transform()* APIs.

5.3. EVALUATION METRICS

Before the methodology is discussed, it is important that the evaluation metrics are decided. Evaluating a GANs performance is a complex task. Evaluating GANs for side channel analysis is much more complex. This is because unlike other domains where the generated samples can be evaluated visually or using mean opinion scores [123], for side channel analysis the information leaked by the trace can not be determined by visual appearance. In this section, various evaluation metrics for judging the quality of the generated traces will be discussed.

To begin, we will discuss the traces evaluation using quantitative metrics such as dynamic time warping [124], power spectral density [125] and even Euclidean distance [126]. The simplest metric can be used to evaluate the traces is Euclidean distance. The drawback of this approach is that even if the generated traces are slightly misaligned (but look fairly similar), the Euclidean distance will be extremely high. Dynamic time warping provides a solution to this problem and aims at finding the best alignment between signals and while doing so, it calculates a distance matrix and outputs a distance measure. *Dynamic time warping (DTW)* [127] finds an optimal alignment between two unmatched temporal sequences. This optimal alignment or the 'warping path' maps the two sequences such that the distance between them is minimized. The minimum distance can be used as a measure for similarity between any such two sequences. Similarly, even the *Power spectral density (PSD)* of two signals can be used as a measure for similarity between them. *Power spectral density (PSD)* is the measure of power distributed across different frequency components that compose a signal [125]. For the real and the generated traces to be visually similar, the *DTW* distance should be low and the *PSD* should be very similar. In all these three metrics, the issue is that if the GAN goes through a collapse mode and outputs just a single trace again and again, the similarity in the traces is so high that the error values will stay low. Hence, other metrics must be explored.

Another approach is to use the popular computer vision metrics for GANs such as Frechet inception distance [128] and Inception score [129]. Unfortunately both of them make use of the Inception classifier [130] trained on Imagenet[131] and hence can not be applied as is to the side channel analysis domain, given the obvious differences in the data representation discussed in Section 5.2. However, their idea can be slightly tweaked so as to make a good evaluation metric. The idea is as follows. A HW classifier, like that introduced in Section 4.4.2, can be trained on a AES traces for a set of random plaintext and random key. This is similar to the methodology introduced in [95]. Next, the classifier can be tested on both real and generated traces. Comparing the accuracy of the classifier can give a good insight on how similar the power traces are. Using the classifier, classification similarity metrics can be defined which means that if a trained classifier classifies both real and generated traces similarly, it can be concluded that the traces are almost indistinguishable.

Other than the classifier methodology, the traces can also be compared using side channel analysis metrics like guessing entropy. If a CPA attack is carried out on the real and generated traces, the similarity in the number of attackable bytes (termed as *attackability* from here on) will suggest that the generated traces are very similar to the real traces.

Table 5.1: Evaluation metrics

Metric	Definition	Drawback
PSD	Comparing power in the frequency components of real and generated power traces	Similar PSD does not guarantee similar attackability.
DTW	Comparing the real and generated power traces temporally with the aim to find an optimal alignment between the two. The DTW distance gives an insight on how close or far the real and generated traces are.	Similar DTW does not guarantee similar attackability.
CPA attack ranks	Comparing the attack ranks on the real and the generated traces for the same set of plaintext and keys. The aim is to find if both sets of power traces leak information equally.	Similar attackability does not guarantee identical trace structure.
SNR	Comparing the signal to noise ratio of the real and the generated power traces using specific leakage models. Spikes in the SNR plot give an insight on how attackable the traces are.	Sensitive to the leakage model.
TVLA	Comparing the number of leakage points for real and generated power traces using Welch's t-test. Similar number of leakage points for both suggest that traces are leaking similar amount of information.	Not robust. Proven to not quantify side-channel vulnerabilities often.
Classifier similarity	Comparing the performance of a pre-trained HW classifier on the real and generated traces. Aim is to find if both real and generated traces have similar accuracy.	Class imbalance problem in the HW label can give a false sense of similarity.

Finally, some conformance style testing can also be used to evaluate the amount of information leaked by a particular design. For example, the signal to noise ratio can be used. SNR was introduced in the context of side channel analysis by Messerges *et al.* [132] and since then has been very popular in the hardware security domain. For example, it was used in [133] to evaluate the efficacy of countermeasure against side channel attacks. The aim behind finding the SNR is as follows, the leakage from the power traces is said to be constituted by three main components. First, is the exploitable component as a result of the cryptographic operation being executed. Second, is a noise component which is said to follow a Gaussian distribution and third, is a constant component which is not dependent on either the noise or the exploitable component. In all, the total leakage is as in equation 5.1, where $L(p,s)$ is the total leakage, $\epsilon \cdot V(p,s)$ is the exploitable component, L_N is the noise component and L_0 is the constant which is nothing more than an offset being caused by other unrelated parts of the cryptographic device [134].

$$L(p,s) = \epsilon \cdot V(p,s) + L_N + L_0 \tag{5.1}$$

The SNR is calculated as the ratio of the variance of the signal to variance of the noise. In the chipwhisperer [135] API, the segregation on the trace on the basis of the specified leakage model is done so as to segregate the traces into different hamming weight categories before the SNR can be calculated. Another popular form of conformance style testing like SNR is TVLA testing [136]. Based on Welch's t-test, it is often used in the side channel domain to determine if the power traces leak infomation about the key. The aim is to compare two sets of power traces. One set with a fixed plaintext and the other set with random plaintexts. The t-test leverages the mean and variance of these two sets, as in equation 5.2, to determine the t value which is then compared with a predetermined constant to assess leakage. Here $\bar{X}_1$, S_1^2, and N_1 represents the means, the variance, the total number of used fixed plaintext traces, respectively, while $\bar{X}_2$, S_2^2, and N_2 represents the means, the variance, the total number of used random plaintext traces, respectively. It must be noted that although popular, TVLA results have been proven to not quantify side-channel vulnerabilities

correctly in certain cases [137].

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{S_1^2}{N_1} + \frac{S_2^2}{N_2}}} \tag{5.2}$$

To sum it up, the GAN can be evaluated on the basis of the metrics presented in table 5.1. However, extensive studies on the shortcoming of using of machine learning metrics for side channel evaluations have been done [57] concluding that using side channel analysis metrics like guessing entropy is much better than using ML metrics. Hence the generated traces will first be evaluated using side channel analysis metrics, namely CPA attack ranks. For visual comparison of the traces, similar to some CAD simulator tools that aim at power trace generation [138] [24], DTW distance will be used. In the discussion section, the remaining evaluation metrics will be used.

5.4. LABELS FOR GAN

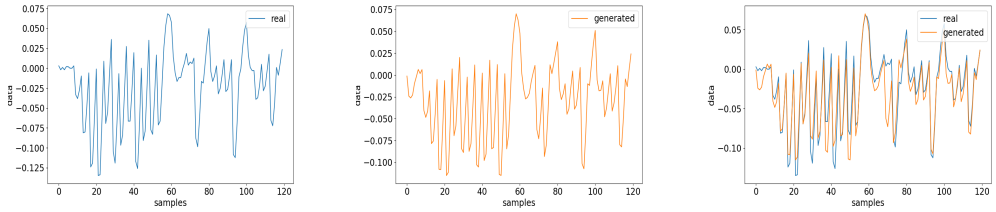
GANs were initially proposed [70] as unsupervised methods, which means that the training procedure did not require labels associated with each training sample. Hence, even though GANs were proven to generate good quality images, it was not possible to specify exactly what type/class of image it should produce since the generator of the GAN only took random noise as input. To address this issue a conditional GAN variant [82] was introduced soon after. Their proposed GAN took both random noise and a class label as input. The class label was passed through an embedding layer which was then concatenated with the feature map before being passed through the hidden layers. This technique made it possible to tell the GAN exactly what class of image it should generate.

It is important to address the importance of labels in the context of GANs. A Multilayer perceptron GAN (MLP-GAN), as can be seen in table 5.2, was trained using 10000 AES encryption traces and then 1000 traces were generated using the GAN. A comparison of the real and the generated trace can be seen in Figure 5.4.

Table 5.2: MLPGAN

	Generator	Discriminator
Input Layer	Dense*(noise dim, width//4), BatchNorm1d**, LeakyReLu(0.2), Dropout(0.3)	Dense(power trace length, width), BatchNorm1d, LeakyReLu(0.2), Dropout(0.3)
Hidden Layer 1	Dense(width//4, width//2), BatchNorm1d, LeakyReLu(0.2), Dropout(0.3)	Dense(width,width//2), BatchNorm1d, LeakyReLu(0.2), Dropout(0.3)
Hidden Layer 2	Dense(width//2, width), BatchNorm1d, LeakyReLu(0.2), Dropout(0.3)	Dense(width//2,width//4), BatchNorm1d, LeakyReLu(0.2), Dropout(0.3)
Output Layer	Dense(width, power trace length), Tanh	Dense(width//4,1), Sigmoid

*parameter order is (input-size, output-size) ** number of features = output-size of Dense layer



(a) Real AES Power trace (b) Generated AES Power trace (c) Comparison

Figure 5.4: AES Traces generated using a MLPGAN

Table 5.3: CPA attack ranks : GAN

Trace	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15	Byte 16
Real	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Generated	100	184	35	251	107	63	195	175	60	18	249	165	122	148	192	28

As can be seen in Figure 5.4, the real and the generated traces are almost identical. However, the CPA attack ranks in table 5.3 tell a different story. The generated traces are not attackable. This is expected since

even though the GAN was able to learn the structure of the power traces, the lack of any label causes the GAN to output power traces that can not be associated with the plaintexts and the keys. This lack of association makes the CPA attack unsuccessful. An extension to the unsupervised GAN presented in this section is the Information Maximizing GAN (InfoGAN) [139] wherein the GAN implicitly tries to learn a segregation between different classes by learning different categorical control variables for different label values. However, correctly classifying the traces to their respective leakage model categories is a non-trivial task, unlike visually classifying the MNIST [140] digits that are used by the authors of the paper [139]. When using computer vision datasets like MNIST it is very straightforward to vary the control variable and visualize the output of the GAN so as to determine which control variable belongs to which class. However, power traces can not be visually classified into their leakage model class and hence the categorical control variables learned by the GAN can not be easily deciphered. Therefore, even if the GAN learned to distinguish between different leakage models, just telling which categorical control variable corresponds to which leakage model label is tricky. Using a trained classifier doesn't make sense either since training the classifier is a profiling step and would mean that we have access to a clone device and know both the plaintexts and keys. In that case, its more straight forward to use the proposed supervised approach since training an InfoGAN implies training not only the generator and the discriminator but also an additional neural network which aims at modelling the distribution of the control variables (leakage model labels) given the power trace.

Since using GANs without any labels produce traces that incorrectly portray the leakage, in the following Sections different label options for the GAN will be discussed.

5.4.1. HW/HD LABEL OPTIONS FOR THE GAN

Conditioning the GAN on informative labels is by far the most important component of the entire methodology. As discussed in Section 5.4, uninformative labels can cause the GAN to generate traces that look extremely similar to the actual traces, however these traces will not be attackable. The visual similarity stems from the innate visual indistinguishability in the power traces. If a wrong label is used to generate the power trace, the correlation between the power trace and the intermediate value will be incorrect, causing the guessing entropy to stay high.

Let's first consider generating power traces using just the Hamming weight for just one byte of the SBOX output of plaintext and known key. The structure of this GAN will look like Figure 5.5. As indicated in the Figure, once we compute the sbox output of the plaintext byte and the key byte, we need to process the output somehow. This is where the leakage model (hamming weight) comes into place. We can also calculate a hamming distance value. However, the power trace that we aim to generate (and have in our training set) is not for a particular byte but for all 16 bytes. This puts us in a dilemma. If we just use one byte, conditioning the GAN on a that byte's hamming weight is pretty straightforward. However, we are wasting a lot of useful information in that case. This has a substantial impact on the performance of the GAN.

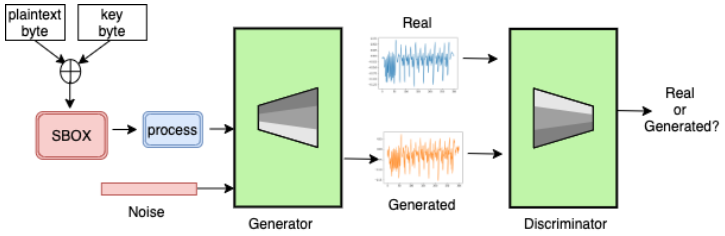
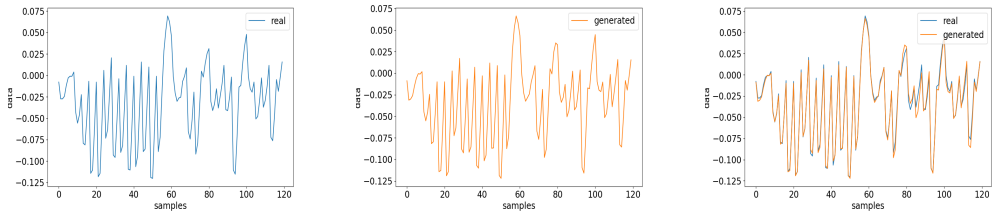


Figure 5.5: Using processed HW as label

As can be seen in Figure 5.6, the generated traces are very close to the real traces, just like the unsupervised GAN in Section 5.4. Where the change lies, is the cpa attack ranks. As can be seen in table 5.4, the first byte is attackable. It must be noted the first byte's HW was used as label in the training process and only that byte is attackable using the GAN generated traces. Since most of the bytes are not attackable, it is evident that just a single byte's HW is not very informative. As a solution to this issue, the 16 HW values corresponding to each power trace must be processed and associated with each power trace as a label during the GAN training. This process can be done in a variety of ways.

The aforementioned process aims at reducing the 16 HW values into one single value which can then be used as a label. The most viable option seems to take the mean of all the bytes' HW. However, there is an issue in that approach. Almost all GAN architectures use embedding layers for labels. These embedding layers are



(a) Real AES Power trace (b) Generated AES Power trace (c) Comparison

Figure 5.6: AES Traces generated using HW of single byte as label

like look-up tables and hence expect a whole number as input. Mean values of course are not guaranteed to be whole numbers, making the labelling process tricky. I will discuss a technique to condition the GAN on mean values in Section 5.5.5. Till then, lets consider methods that process the 16 bytes' HW and return a whole number value. One option is to either take *mode* of hamming weight of the 16 bytes. Also, the mean of HW of the 16 bytes can be processed using *floor*, *round* or *ceil* functions. A similar process can be carried out using the hamming distance.

Table 5.4: CPA attack ranks for HW labels

Trace	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15	Byte 16
Real	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Generated: Single-byte HW Label	1	191	209	199	236	172	66	166	168	7	220	164	6	107	191	57
Generated: Mode of HW Label	40	22	1	57	10	29	24	1	1	3	88	1	13	1	88	241
Generated: Round of mean of HW Label	1	1	1	1	1	1	1	1	1	1	1	9	1	1	1	2
Generated: Ceil of mean of HW Label	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Generated: Floor of mean of HW Label	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Generated: HW template Label	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

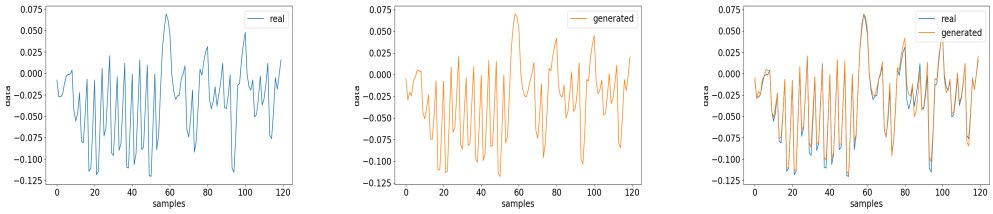
Each processing technique, of course, yields different label values and also performs quite differently as is evident in table 5.4. Using mode of HW label, the results are much different from the single HW byte label. Visually, the traces still look identical as in Figure 5.7. However, CPA attack on the generated traces performs better. As for taking the mean of the HW values: ceil, floor and round perform much better. The reason behind this, according to me, can be derived from label count-plots as in Figure 5.9 wherein it is evident that ceil, floor and round processing reduces the label into just two almost equal categories. This acts as a solution to the HW class imbalance issue (as can be seen in the *single-byte* and *mode* plots) and enhances the performance of the GAN. It must be mentioned that just like mode of hamming weight case, the generated traces visually look similar to the validation set in the remaining label options.

This HW idea can further be extended to make template based labels. This label takes after the pre-processing step of the template based attacks explained in Section 3.4. Point of interests can be calculated using the sum of difference method. Traces can then be segregated on the basis of HW. For these segregated traces, mean and covariance matrices can be made. These matrices can then be flattened out to be used as an input vector for the generator. Visually, a flattened (and concatenated) mean and covariance matrix for 9 different hamming weights looks like Figure 5.8. However, the embedding layer in this case can naturally not be used. Instead, a Dense layer is used and the output of this layer is then concatenated with the noise dimension. This means that the architecture presented in table 5.5 is only slightly modified such that the embedding layer is replaced with a dense layer followed by concatenation with the noise vector. This archi-

Table 5.5: MLPCGAN

	Generator	Discriminator
Embedding Layer	Embedding(9,10) Concatenate(noise, label embedding)	Embedding(9,10) Concatenate(noise, label embedding)
Input Layer	Dense(noise dim + 10, width//4)*, BatchNorm1d**, LeakyReLu(0.2), Dropout(0.3)	Dense(power trace length + 10, width), BatchNorm1d, LeakyReLu(0.2), Dropout(0.3)
Hidden Layer 1	Dense(width//4, width//2), BatchNorm1d, LeakyReLu(0.2), Dropout(0.3)	Dense(width,width//2), BatchNorm1d, LeakyReLu(0.2), Dropout(0.3)
Hidden Layer 2	Dense(width//2, width), BatchNorm1d, LeakyReLu(0.2), Dropout(0.3)	Dense(width//2,width//4), BatchNorm1d, LeakyReLu(0.2), Dropout(0.3)
Output Layer	Dense(width, power trace length), Tanh	Dense(width//4,1), Sigmoid

*parameter order is (input-size, output-size) ** number of features = output-size of Dense layer



(a) Real AES Power trace (b) Generated AES Power trace (c) Comparison

Figure 5.7: AES Traces generated using mode of HW as label

tecture can also be extended to use the entire 16 bytes HW as labels for the GAN. The results for both cases is impressive, as pointed out in table 5.4.

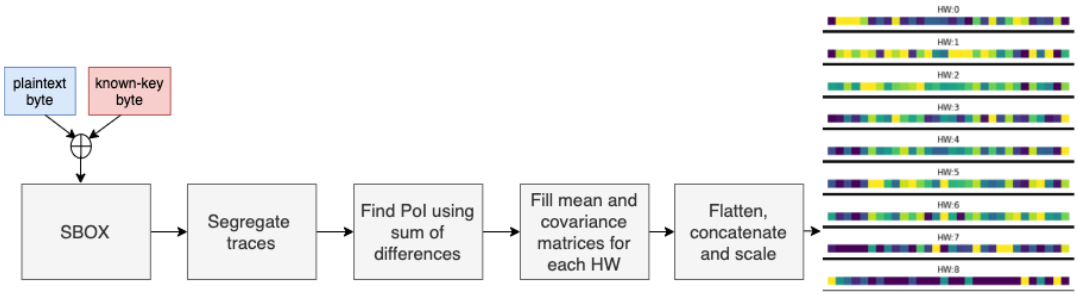
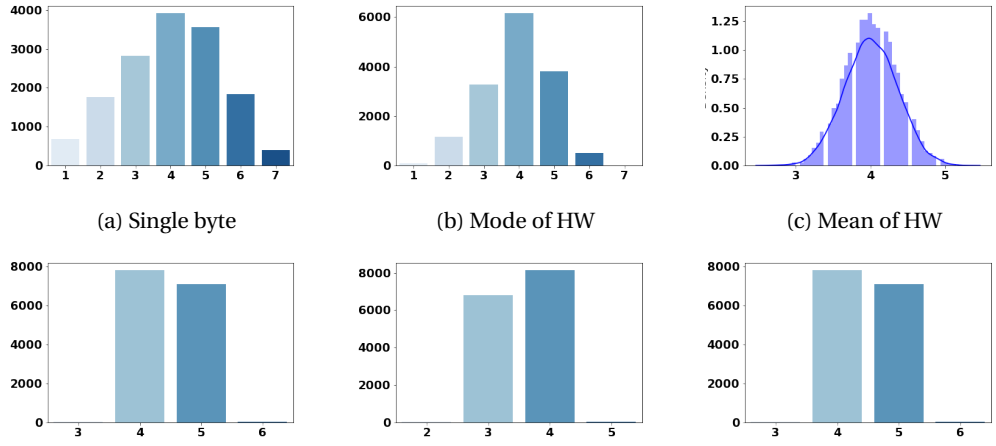


Figure 5.8: HW templates

All the results presented in this Section are for traces generated using the GAN architecture presented in table 5.5. Conditioning the GAN on Hamming weight values gave good results. However, there are issues with this methodology. First, we need the know the actual key. Generating the HW labels just using key guesses between 0 to 255 does not work. This is because if we loop over all possible key values, the mean/mode of hamming weight will become 4 for every plaintext that is encrypted. Second, the inherent class imbalance in hamming weight seems to have an impact on the GAN performance. A plot of labels, while using mode of HW of all 16 bytes can be seen in Figure 5.9. Ideally, there should be 9 categories (0 to 8), each with similar number of samples. However, as is apparent form the Figure, that is clearly not the case here. This issue becomes much more apparent once the number of generated traces is reduced. This will be highlighted in chapter 6.

Using the known key to calculate labels for the GAN and then judging the GAN generated traces on the basis of their attackability seems trivial. So as to make the GAN flexible, such that it can be used to task independently generate power traces, value change dump (VCD) files need to be used as labels. VCD or value change dump files are collections of the variables changes during simulation. Next, VCD conditioning will be discussed in detail.



(a) Single byte (b) Mode of HW (c) Mean of HW

(d) Ceil of Mean of HW (e) Floor of Mean of HW (f) Round of Mean of HW

Figure 5.9: Hamming Weight Label options for GAN

5.4.2. VCD LABEL OPTION FOR GANS

VCD or value change dump files are collections of the variables changes during simulation. These variable changes during simulation is a good measure for power consumption. As demonstrated by Gu and Almasry [51], the power consumption depends on the switching activity of transistors. This switching activity of the transistor is proportional to change of the input values to the transistor. Also, since the VCD files are not limited to any specific operation, they can innately make the GAN more flexible.

Using the VCD files as labels, however, is not as straight forward as using Hamming weight values. For HW, an embedding layer was used. The embedding layer's output is then concatenated with the input noise vector and finally fed into the generator. The VCD files hence are required to be processed, so that it can be concatenated with the noise vector. First, the VCD file must be converted into a transition lists. These lists combine the number of transitions into an array, such that for every power trace in the training set there is a corresponding transition list. The transition list must be processed so that it can be used as an input to the generator. This can be done in the following ways:

1. Spectrogram of the transition array: A Spectrogram is said to be *"a picture of sound"*. It represents all the constituent frequencies of a signal with respect to time. To compute a spectrogram of the transition array, we need to perform three steps. First, we need to perform short time fourier transform using certain length of the windowed signal and hop-length. Second, the output needs to be squared so as to remove the complex values. Finally, we need to convert the power spectrogram to the dB scale.

2. Mean and variance pairs of the transition array: The transition array can also be reduced to just their corresponding mean and variance values. These mean and variance pairs can then be feed into the network using continuous conditioning as explained in 5.5.5.

3. Using entire transition array: Finally, even the entire transition array can be used as an input vector to the generator.

As will be explained in the chapter 6, the size of the transition array after processing is reasonably short and hence just the transition array can be used as an input vector. In case of very long transition array length, the aforementioned processing techniques can be very helpful. The mean and covariance method loses out on a lot of information and due to the high inter-trace are almost indistinguishable. The spectrogram methodology, depending on the choice of the number of channels and window length, can reduce the size of the transition array. This spectrogram can then be flattened and used as an input to the generator to generate attackable traces.

The VCD file is structured in a way that segregates the transitions in the wires and registers on the basis of time. These VCD timestamps can then be correlated with the plaintexts that are encrypted at that instant. Visually, the process can be seen in Figure 5.10. Our aim is to procure training data for the GAN using the VCD

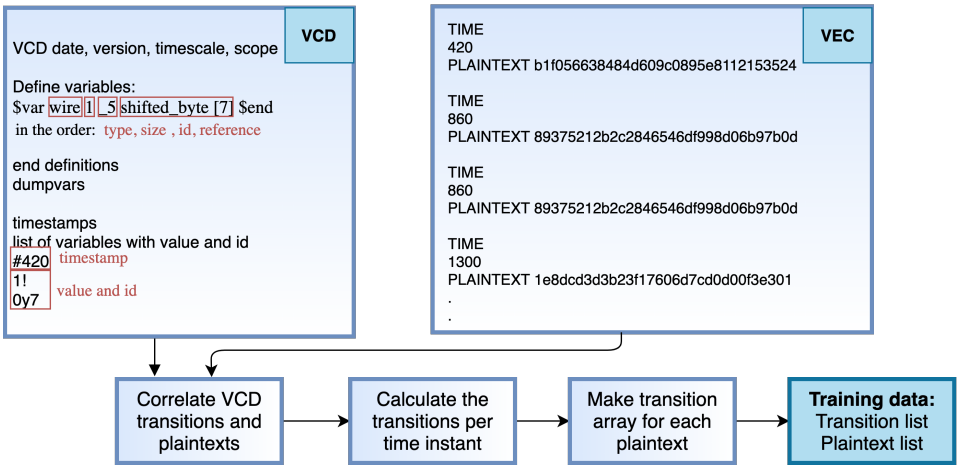


Figure 5.10: VCD parsing to get training data

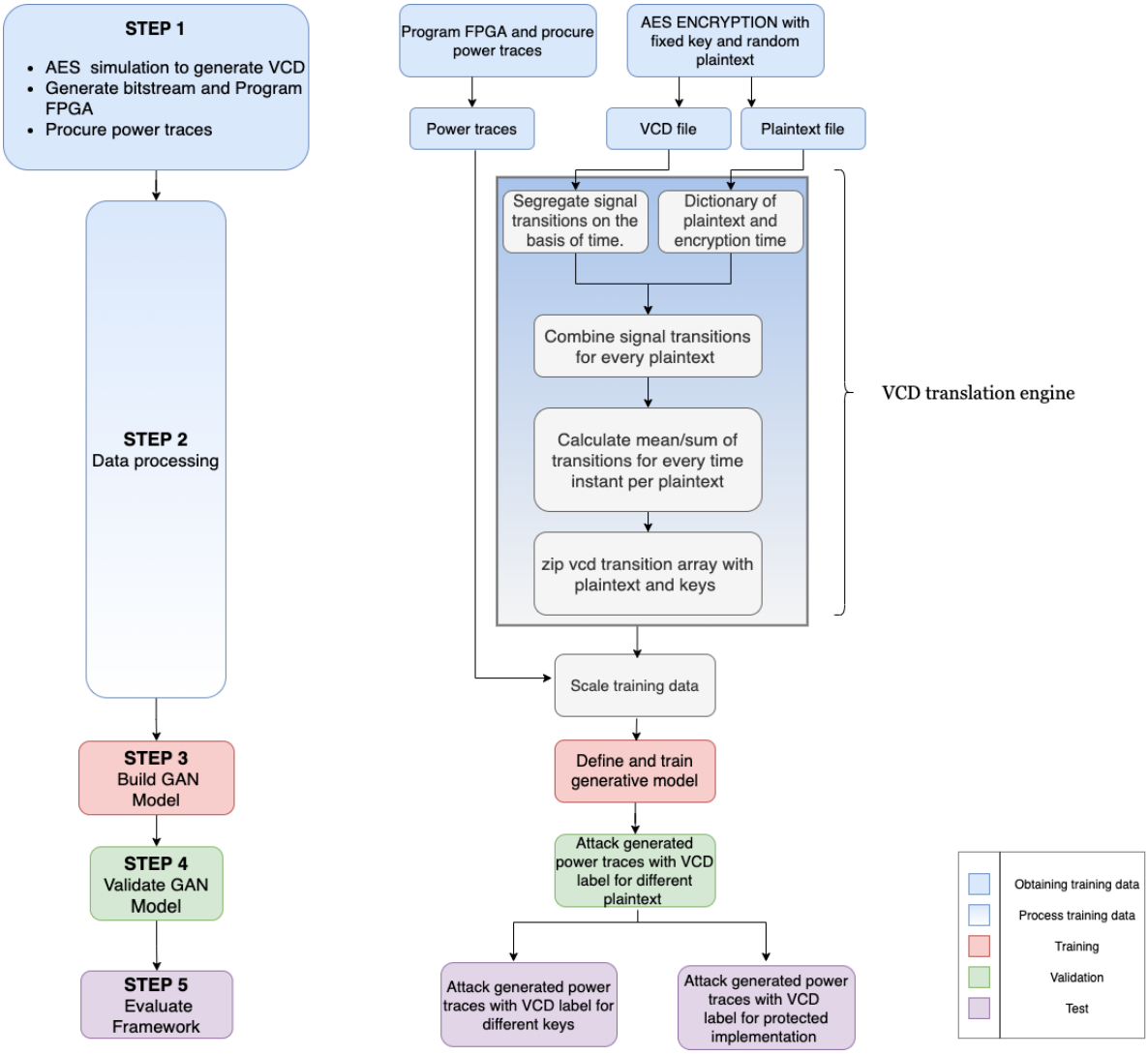


Figure 5.11: Using VCD to generated power traces

file when the vec file (list of plainexts) is available. These plaintext and keys are then run on a chipwhisperer

emulating the AES design to record power traces corresponding to those encryptions.

VCD files are human readable, since they are based on the ASCII standard, and contain the following information [141]:

1. Header : Every VCD file contains a header portion to highlight the tool used to make the VCD file, datetime of creation and the timescale used.

2. Scope : This corresponds to the scope to which the variables belong to. This term can be nested to refer to nested scopes. For example, for a signal belonging to module which is in turn located within a particular top level design will have nested scope entries in the VCD file.

3. Variable declaration: Every variable declaration has four components. First is the type of the variable, which can be a register, a wire or a parameter. Next, is the size of the variable in bits. Third, is an identification code corresponding to the variable. Finally, there is the user defined name (known as *reference*) corresponding to the variable.

4. Data Section: Finally, the data Section contains a series of timestamps and operations corresponding to variables on those timestamps.

Figure 5.11 explains the methodology to use VCD files as conditioning for GAN. The core idea behind this methodology is to use the wire transitions to obtain a transition array that can be used as input to the GAN. Since VCD transitions are application independent, this approach in theory can be extended to generated power traces corresponding to any implementation. VCD conditioning in GANs does not pose any added architectural requirements. If anything, it makes the architecture much more flexible since it removes the need of an embedding layer. The generator and the discriminator can be implemented using dense or convolutional layers, similar to the MLPGAN and DCGAN as can be seen in table 5.6 and 5.7. Detailed diagrams can be seen on page 95 and 96.

Once the VCD transitions are processed using the vcd translation engine, as in figure 5.11, the transition array can be used as an input to the generator. The length of the vcd transition list is termed as *vcd length* in table 5.6 and 5.7.

Table 5.6: MLPGAN : VCD Label

	Generator	Discriminator
Input Layer	Dense(vcd length, width//4)*, BatchNorm1d**, LeakyReLu(0.1), Dropout(0.3)	Concatenate(vcd, power trace), Dense(vcd length + power trace length, width), BatchNorm1d, LeakyReLu(0.1), Dropout(0.3)
Hidden Layer 1	Dense(width//4, width//2), BatchNorm1d, LeakyReLu(0.1), Dropout(0.3)	Dense(width,width//2), BatchNorm1d, LeakyReLu(0.1), Dropout(0.3)
Hidden Layer 2	Dense(width//2, width), BatchNorm1d, LeakyReLu(0.1), Dropout(0.3)	Dense(width//2,width//4), BatchNorm1d, LeakyReLu(0.1), Dropout(0.3)
Output Layer	Dense(width, power trace length), Tanh	Dense(width//4,1), Sigmoid

*parameter order is (input-size, output-size) ** number of features = output-size of Dense layer

Table 5.7: DCGAN : VCD Label

	Generator	Discriminator
Input Layer	Conv1d(1, width, 9, 4)*, Upsample(factor=2, mode='nearest') BatchNorm1d**, LeakyReLu(0.1)	Concatenate(vcd, power trace), Conv1d(1, width, 9, 4, 2), BatchNorm1d, LeakyReLu(0.1) MaxPool1d(2,2)
Hidden Layer 1	Conv1d(width, width//2, 9, 4), BatchNorm1d, LeakyReLu(0.1)	Conv1d(width, width//2, 9, 4, 2), BatchNorm1d, LeakyReLu(0.1) MaxPool1d(2,2)
Hidden Layer 2	Conv1d(width//2, width//4, 9, 4), BatchNorm1d, LeakyReLu(0.1)	Conv1d(width//2, 1, 9, 4, 2), BatchNorm1d, LeakyReLu(0.1)
Output Layer	Conv1d(width//4, 1, 9, 4), Tanh	MaxPool1d(2,2), Sigmoid

*parameters in order: in channels, out channels, kernel size, padding, stride **number of features = out-channels of Conv1d

However, as pointed out in Section 4.4.8, scaling is an integral part of the framework while working with a limited training set. This means that the generated traces, even for different VCD files will look very similar to the training set. This is not ideal since for different implementations, the power traces might have very different amplitude and frequency components which will not be modelled by the GAN. To address this issue, in some cases an additional labels can be used to signify the different implementations. Otherwise the GAN can either be completely trained again or transfer learning can be introduced. This will be explained in the discussion section of chapter 6.

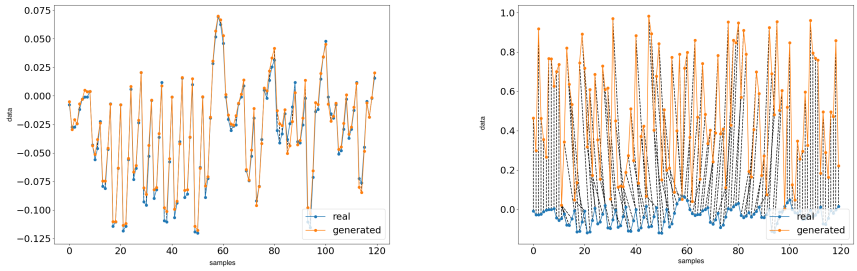
5.4.3. IMPORTANCE OF POWER TRACES IN GAN TRAINING

Now that we know the importance of labels in GAN training. It is important to explore how important power traces are in the same context. Of course, there is a visual aspect. If this tool is used in tandem with an existing simulation framework, it is important that the generated traces visually look very similar to what the end user would expect. However, since the assessing criteria for our framework till now also includes CPA attack ranks, it is possible that the dependence on the power trace structure is not paramount.

To analyze this in greater detail, CPA attack ranks of the generated power traces will be juxtaposed with an additional metrics like power spectrum [142] and dynamic time warping [124].

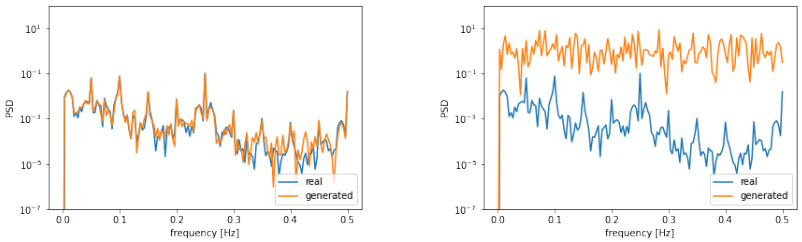
Table 5.8: Comparing real and generated traces

	trained using real traces	trained using random noise
DTW	0.525	63.539
Attackable bytes	16	16



(a) Trained using actual power trace (b) Trained using random noise

Figure 5.12: DTW of real vs generated



(a) Trained using actual power trace (b) Trained using random noise

Figure 5.13: PSD of real vs generated

Despite the visual disparity between the generated traces in the two cases (as in Figure 5.12) as well as the DTW and PSD inequality, all the bytes are attackable using a CPA attack (as in table 5.8). With this, we can conclude that the power traces are necessary for the visual appearance of the generated traces. However, they do not have any substantial impact on the attackability of the power traces. This is because the labels are

computed using the known key and implictly categorize the traces into their respective leakage model bin, making the GAN capable of generating traces that leak information which can be leveraged by a CPA attack. Of course, the noise has a detrimental impact on the generated traces which is very apparent when only a limited number of traces are generated. In table 5.8 the attackable bytes are a result of using 4000 generated traces. If the number of traces is reduced to 1000, the attackable bytes when using the real traces to train will become 11. However, the attackable bytes when using random noise to train will become 8. This is because using noise hinders the CPA attack, which is sensitive to amount of noise in the trace.

5.5. CHOOSING THE RIGHT GAN ARCHITECTURE

The previous Sections discussed possible label options as well as the importance of labels in GAN training. This Section will introduce and compare the architectures which can accommodate those different label values. These architectures vary substantially depending on the label value and power trace length.

5.5.1. POSSIBLE OPTIONS

Before deciding on the best architecture for the task of power trace generation, it is important to discuss a few possible GAN architectures. These architectures will mainly be classified on the basis of the layers and labels they use. As presented in Section 5.4.1, GAN architectures can simply be made using Dense layers. GANs can also be made using convolutional layers as well as RNN layers, in tandem with the Dense layers. For whole number labels, an embedding layer can be used.

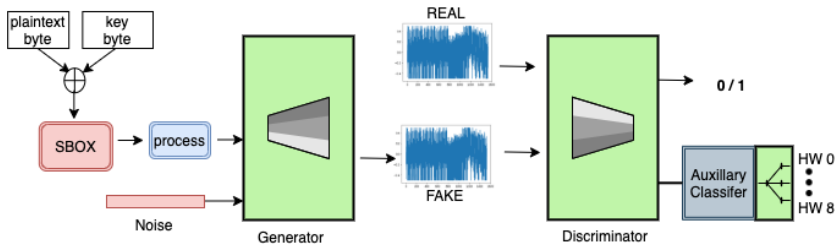


Figure 5.14: Auxiliary classifier GAN

On a high level, these GANs can have structure like that shown earlier in Figure 5.5 or have an auxiliary classifier attached to the discriminator's output [119] like Figure 5.14. The length of the power trace has a big impact on the choice of the GAN architecture. A basic conditional GAN architecture like the one proposed by [117] works fairly well with shorter traces (around 60-100 samples). To make the GAN more robust across all label categories, an auxiliary classifier can be attached to the discriminator's output. This auxiliary classifier adds a classification loss in addition to the traditional binary cross entropy loss of the GAN. The classification loss forces the GAN to learn the difference between different classes. In either case, different type of layers can be used. For example, if convolutional layers are used, the resulting GAN will have a structure like table 5.9. Such a GAN is known as deep convolutional GAN [143] (DCGAN).

The structure of DCGAN is similar to that of a dense GAN. However, the difference is that the number of trainable parameters decrease. This is because convolutional layers innately have lesser parameters than dense layers because of weight sharing. Using convolutional layers, also provides the GAN all the advantages stated in Section 4.2.2. Similar to the DCGAN, a LSTM based GAN can be made. LSTM layers can been used in either one of the generator or the discriminator or both the generator and discriminator like in [144] and [145]respectively. An example of a CNN-LSTM GAN is as in table 5.10.

Now that the most common GAN architectures have been introduced, other possible architectures like an architecture for generating really long power traces (greater than 1000 samples) and an architecture for using real number labels (also known as regression labels) will be explained in Section 5.5.2 and Section 5.5.5 respectively.

5.5.2. GANS FOR SHORT TRACES

With short traces, we have the liberty of using any type of GAN architecture discussed till now. This means that all conditional MLPGAN [117], ACGAN [119], DCGAN [143], CNN-LSTM GAN [144] perform well as will be highlighted later in chapter 6.

Table 5.9: DCGAN

	Generator	Discriminator
Embedding Layer	Embedding(9,10) Concatenate(noise, label embedding)	Embedding(9,10) Concatenate(noise, label embedding)
Input Layer	Conv1d(1, width, 9, 4)*, Upsample(factor=2, mode='nearest') BatchNorm1d**, LeakyReLu(0.1)	Conv1d(1, width, 9, 4, 2), BatchNorm1d, LeakyReLu(0.1), MaxPool1d(2, 2)***
Hidden Layer 1	Conv1d(width, width//2, 9, 4), Upsample(factor=2, mode='nearest') BatchNorm1d, LeakyReLu(0.1)	Conv1d(width, width//2, 9, 4, 2), BatchNorm1d, LeakyReLu(0.1), MaxPool1d(2, 2)
Hidden Layer 2	Conv1d(width//2, width//4, 9, 4), Upsample(factor=2, mode='nearest') BatchNorm1d, LeakyReLu(0.1)	Conv1d(width//2, width//4, 9, 4, 2), BatchNorm1d, LeakyReLu(0.1), MaxPool1d(2, 2)
Output Layer	Conv1d(width//4, 1, 9, 4), Tanh	Conv1d(width//4, 1, 9, 4, 2), MaxPool1d(2,2), Sigmoid

*parameters in order: in channels, out channels, kernel size, padding, stride
**number of features = out-channels of Conv1d ***parameters in order: kernel size, stride

Table 5.10: CNN-LSTM GAN

	Generator	Discriminator
Embedding Layer	Embedding(9,10) Concatenate(noise, label embedding)	Embedding(9,10) Concatenate(noise, label embedding)
Input Layer	Conv1d(1, width, 9, 4)*, Upsample(factor=2, mode='nearest') BatchNorm1d**, LeakyReLu(0.1)	Reshape(batch-size, seq-length, input-size) LSTM(input-size, hidden-size, num-layers) Where: trace length = seq-length x input-size hidden-size = width
Hidden Layer 1	Conv1d(width, width//2, 9, 4), Upsample(factor=2, mode='nearest') BatchNorm1d, LeakyReLu(0.1)	
Hidden Layer 2	Conv1d(width//2, width//4, 9, 4), Upsample(factor=2, mode='nearest') BatchNorm1d, LeakyReLu(0.1)	
Output Layer	Conv1d(width//4, 1, 9, 4), Tanh	Dense(width,1)***, Sigmoid

*parameters in order: in channels, out channels, kernel size, padding
**number of features = out-channels of Conv1d ***parameters in order: (input-size,output-size)

Using LSTM layers in both generator and discriminator does not work as well as the other GANs. It is possible that finetuning the implementation results in better results. However, it is also possible that just binary cross entropy loss might not be sufficient for LSTM GANs to model the entire temporal dynamics in the training data as stated by Yoon *et al.* [81]. This result is inline with [97] where the authors find that LSTM based classifiers are sub optimal as compared to other deep learning techniques when used for attacking the power traces for an AES hardware implementation. LSTM based classifiers have been shown to perform well on software traces, that are much longer than the hardware traces. LSTM GAN might perform better on such long traces.

A large variety of GANs work well for short trace length. From a leakage assessment point of view, when working with short traces the labels and associated conditioning techniques are much more important that the complexity of the GAN.

5.5.3. GANS FOR LONG TRACES

A progressive GAN approach works much better for long traces. Originally proposed by Karras *et al.* [71], these architectures grow during training and gently fade in the activations from previous layers while growing. This methodology is proven to work for generating large high quality images. Hartmann *et al.* [78] used a progressive architecture to generate EEG signal of length 768 samples. Harell *et al.* take after the EEG-GAN architecture to generate appliance power signatures. For making conditional variants of progressive GANs, the authors had access to very informative labels. For example, in PowerGAN, Harell *et al.* use the appliance type as the label. Appliance power traces for different appliances have a stark difference, hence making it

easier for the GAN to distinguish for different labels.

A conditional progressive GAN can not only incorporate the label options discussed in Section 5.4.1 but can also maintain the quality of the generated traces as the trace length increases. Visually, this can be seen in Figure 5.15.

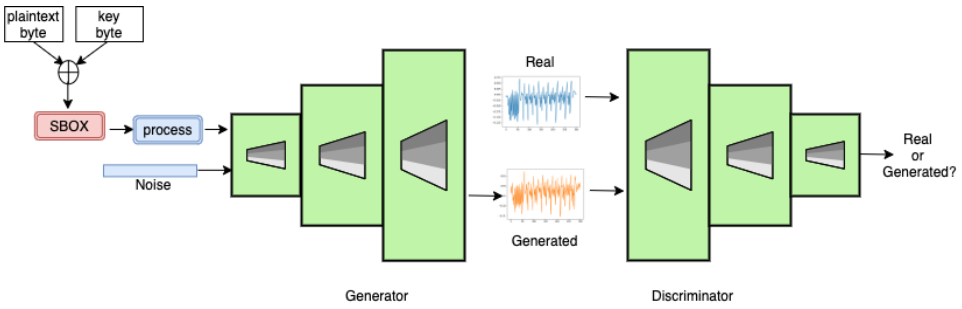


Figure 5.15: Progressive GAN for generating longer traces

Progressive GAN performs well due to the following reasons:

- Iterative increase in the size of generated traces : This architecture is termed as the *Multi-scale architecture*. First, the generator generates traces of 24 time samples and there after the output size doubles after a set number of epochs/iterations. The discriminator's input size is also adjusted accordingly. This process continues till we reach the desired output size. This process assists the training process since generating good quality smaller traces purely from a random noise vector is an easier than generating good quality larger traces. During the training process, when we double the output size of the generator the smaller output and the larger output are blended together in a weighted fashion using a parameter α. This process can be seen in Figure 5.16.

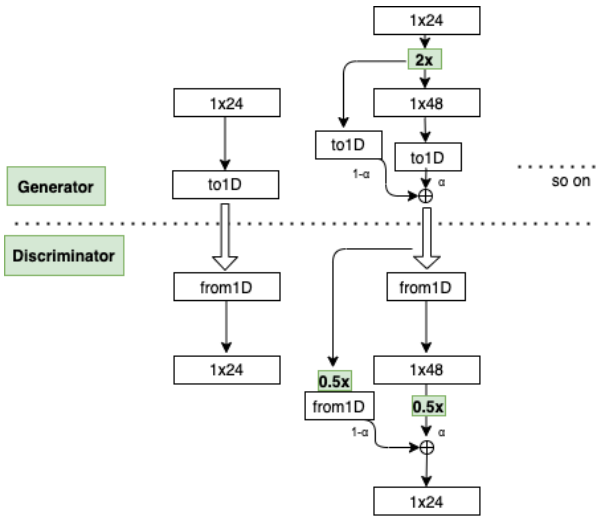
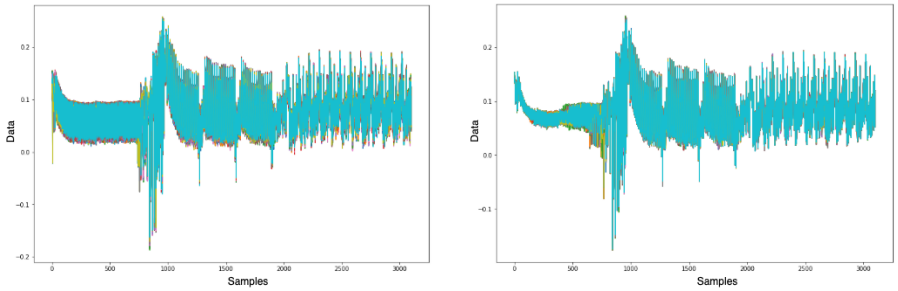


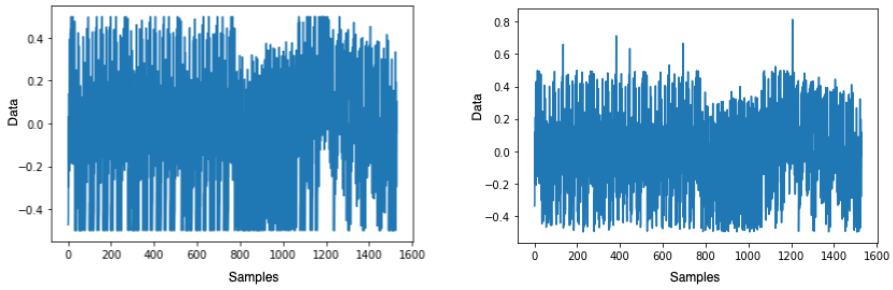
Figure 5.16: Progressive GAN combining outputs

- Equalized Learning rate: Progressive GANs use an equalized learning rate to scale the weights of the network by a constant value such that the weights stay in a similar scale.
- Mini-batch standard deviation: To increase the diversity of images, standard deviation over all features is calculated for every minibatch and is averaged and concatenated to all spatial locations in a feature map towards the end of the discriminator.
- WGAN-GP loss function: Progressive GAN uses the WGAN loss function along with gradient penalty. Refer Section 4.3.2 for a detailed explanation of the same.



(a) 100 Original RSA traces (b) 100 Generated RSA traces

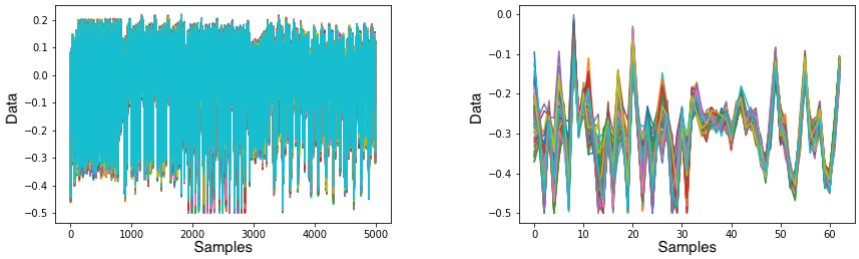
Figure 5.17: RSA Traces generated using progressive GAN



(a) Original AES traces (b) Generated AES traces

Figure 5.18: AES Traces generated using progressive GAN

The results can be seen in Figure 5.17. Each trace has 3072 samples. As for AES traces, the results are equally good, as can be seen in Figure 5.18. Each trace here had 1560 samples. However, it must be noted that there are noticeable spikes in the traces. These spikes should be a result of the fading in process in the progressive GAN. In this work, since we are mostly concerned with hardware AES implementation and its associated traces, the use of progressive GANs is an overkill. However, when using the same methodology with software AES traces or power traces for other encryption algorithms like RSA (like in Figure 5.17), it makes sense to use the progressive GAN methodology.



(a) 100 Orginal traces (b) 100 PoI Traces

Figure 5.19: Reducing long traces using Point of Interests

To bypass the idea of using progressive GANs, even if the power traces under consideration are quite long, the concept of calculating point of interests must be used. This idea is based on the point of interest (PoI) calculation step in the template based attacks. PoIs are calculated using a specific leakage model and make use of the known key. Once the PoIs are known, the trace array can be reduced to just those point of interests and this reduced length power trace can be used while training the GAN. Despite the obvious impact on the structure of the trace, both the training and generated traces are attackable. An example of the same can be seen in Figure 5.19.

5.5.4. GANS FOR A LIMITED TRAINING SET

Till now, all of the GANs were trained using thousands of traces. Acquiring these traces is not a complicated process. However, it may be possible that only a limited number of traces could be extracted given that the time spent on trace acquisition is noticeable. GANs can be trained using limited amount of training data if differentiable augmentation is used.

Zhao *et al.* [118] propose the importance of differentiable transforms in GAN training (specifically for computer vision applications) by using only a 100 images to train large GAN models like Style-GAN and Big-GAN. Without the differentiable augmentation the quality of the generated samples is significantly worse, empirically proving the importance of this technique. The transformation is applied on both the generated and the real images, as can be seen in Figure 5.20. These invertible transforms are essentially means for

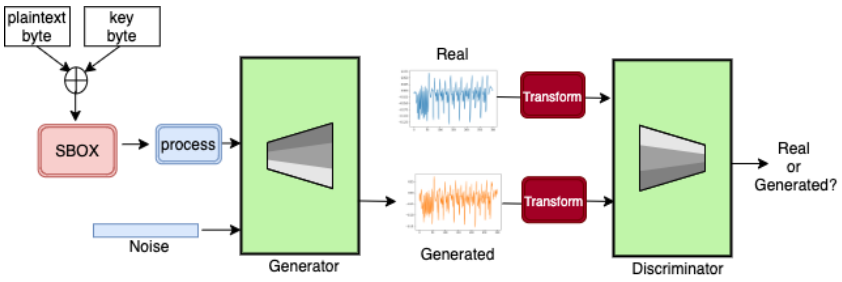


Figure 5.20: Data efficient GAN

data augmentation, which on its own is quite a common practice to prevent overfitting. However, applying data augmentation on GANs was not a common until this paper was presented. Zhao *et al.* propose a few differentiable transforms and provide a framework for applying them while training the GAN.

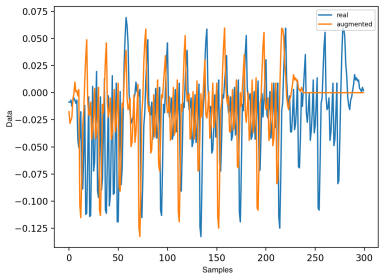


Figure 5.21: Translation differential augmentation

Since the technique was proposed for images, transforms like color,translation and cutout are not meant for power traces. Naively using the transforms does not help in the training process and in some cases even deteriorates the attackability of the generated traces. This is evident from Figure 5.21. Such zero padded shifts introduce misalignment which in turn acts as a countermeasure against the attack. Hence, the differential augmentation technique did not provide any enhancement in terms of attack ranks or training stability. Detailed experiments with other forms of differential augmentations might result in better performance even with much less training data. This, however, is an avenue for future work.

5.5.5. GANS FOR REGRESSION LABELS

All architecture discussed till now used embedding layers for the labels. One disadvantage of using embedding layers is that they expect whole numbers as input. This is because they are implemented as simple look-up tables. However, this simple look-up mechanism makes the embedding layers computationally efficient. Unfortunately, due to the whole number requirement, the label options for the GAN were slightly limited.

As a solution to this problem, continuous conditioning can be used [146]. Unlike traditional CGAN or ACGAN, this technique (known as CCGAN) assumes that each label is actually a float (the authors of [146] call such labels as regression labels). While using regression labels, the overarching problem is that the number

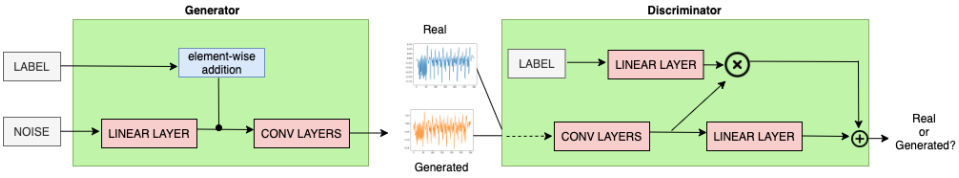


Figure 5.22: Continuous conditional GAN

of possible labels in infinite. In the continuous conditioning method, kernel density estimation is used to estimate the marginal label distributions. Hence, while training, the GAN is conditioned on traces that are in vicinity of the actual label value. This technique is really helpful in situations where the GAN does not have training samples corresponding to particular label values while training. In other words, this training technique helps the GAN generalize much better. Since this conditioning technique does not involve embedding layers, the authors propose a novel conditioning method as can be seen in Figure 5.22. Of course, just Dense/Linear layers can also be used in tandem with this conditioning method.

5.6. TUNING GAN HYPERPARAMETERS

Usually, there is no rule of thumb to find the best set of neural network hyperparameters in one go. Evaluation metrics play an important part in hyperparameter tuning. As discussed in Section 5.4.3, quantitative metrics do not have a direct impact on the attackability of the generated power traces. Since the attackability is of paramount importance while assessing leakage of any design, CPA attack ranks will be the main evaluation metric for the hyperparameter tuning. Having said that, it is essential to address the importance of the trace structure. If this framework is employed along with existing tools, the generated traces must not be too different visually from actual power traces. Hence, DTW will also be assessed.

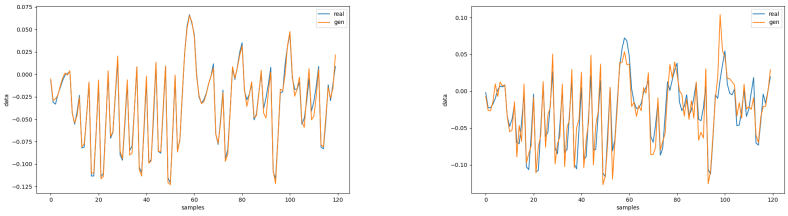
Unfortunately, traditional grid search / random search APIs from sklearn [122] do not fit the purpose as is. This is because GAN's two model training process doesn't implicitly fit into any popular hyperparameter searching APIs.

Hence, the hyperparmeter tuning has been implemented using a self made scorer function to assess GAN's performance on the CPA attack ranks while randomly looping over various hyperparameter values. Needless to say, given the random optimization process, better hyperparameter values might exist. Hence, instead of presenting some hyperparameter values as ideal, this Section discuss the implication of changing a few aspects of the design on the basis of the trace structure (DTW) and attackability.

The initial configuration was made as a combination of the "GAN Hacks" proposed by quite a few GAN researchers [147] and the tuning process was done for the following hyperparameters:

1. Normalization : One of the the most prominent GAN hacks revolve around normalization and scaling. Of course, on its own it is not a novel technique invented just for GANs. The rational behind using normalization in machine learning is to bring all features on a similar scale so as to stabilize and accelerate training as well as increase the generalization ability of the model [105]. With the use of scaling the attackability increases and the DTW distance decreases. As can be seen in Figure 5.23, the generated trace do have a stark visual difference from the validation set with / without scaling. It was observed that batch normalization [110] enhanced the performance of the models. Weight normalization [148] gave an almost comparable performance to batch normalization [149]. However, it must be mentioned that the choice of normalization technique is dependent on the GAN architecture, for example in the ACGAN [119] architecture the authors do not use any form normalization within the networks. Spectral normalization [150] is said to outperform both batch normalization and weight normalization as proposed by Kurach *et al.* [151] wherein they propose using spectral norm over other normalization techniques when training complex GANs. However, since in this thesis we are limited to relatively short traces, just using Batch Normalization in both generator and discriminator works very well.

2. Activation function: Compared to Softplus, Softsign, ReLU and Tanh, LeakyReLu with a slope coefficient of 0.2 performs much better on both attackability and DTW distance. In all cases, the Tanh activation is used at the generator's output and Sigmoid is used at the discriminator's output. With DCGAN, the authors propose using ReLU activation function in the generator and LeakyReLU in the discriminator.



(a) Trained on Scaled data (b) Trained on Unscaled data

Figure 5.23: Effect of Scaling

Table 5.11: Hyperparameters for GAN Training

Hyperparameter	Details
Input Noise Vector	Must not be disproportional to the width of the hidden layers.
Batch size	128 for smaller traces. 32 when using progressive GAN for long traces.
Scaling	Scaling power traces using Standard scaler [107].Other scaling techniques like MinMaxScaler [106], Robust Scaler [109] do not perform much worse. The effect of the transformation is reversed using inverse transform during trace generation.
Normalization	Using batch normalization in both Generator and Discriminator. Spectral normalization when using WGAN [84].
Activation function	Using Tanh at the generator's output and Sigmoid at the Discriminator's output. Using LeakyReLU everywhere else with a slope of 0.1.
Weight initialization	All model weights shall be randomly initialized from a Normal distribution with mean=0, stddev=0.02, like that proposed by the DCGAN [143] authors, performs better than other tested initialization methods.
Number of hidden layers	Depends on the trace length or available training data. Need to be conscious of overfitting. Using 2 to 3 hidden layers usually gives good results with different architectures.
Width of hidden layers	Same as before, it depends on the trace length or available training data. Need to be conscious of overfitting.
Optimizer	Adam with learning rate 0.0002 and β values (0.9, 0.999) works better than other optimizers in most cases.

3. Weight Initialization: Using random initialization from a Normal distribution with mean=0, stdev=0.02, like that proposed by the DCGAN [143] authors performs equal-to or better-than other tested initialization methods.

4. Epochs : The number of epochs have an impact on the DTW value and the attackability. However, it must be pointed out that GAN training is quite unstable and different runs do give different results. This, of course, is also dependent on the available training data. Very limited training data can have detrimental effects on GAN training as highlighted in [118].

5. Batch size : Both [152] and [153] suggest the use of larger mini-batches to reduce the variance in gradient calculations. Yazici *et al.* [152] demonstrate that GANs can overfit when using large batch sizes and GANs are more susceptible to mode collapse when trained with small batch sizes. Hence there is a trade-off that needs to be considered. Additionally, batch sizes also have a substantial impact on the training time. Therefore, as a trade-off a batch size of 128 was chosen. With different training data, this choice might need to be changed (when using progressive GANs for long power traces, the batch size must be reduced in accordance with the available computational resources).

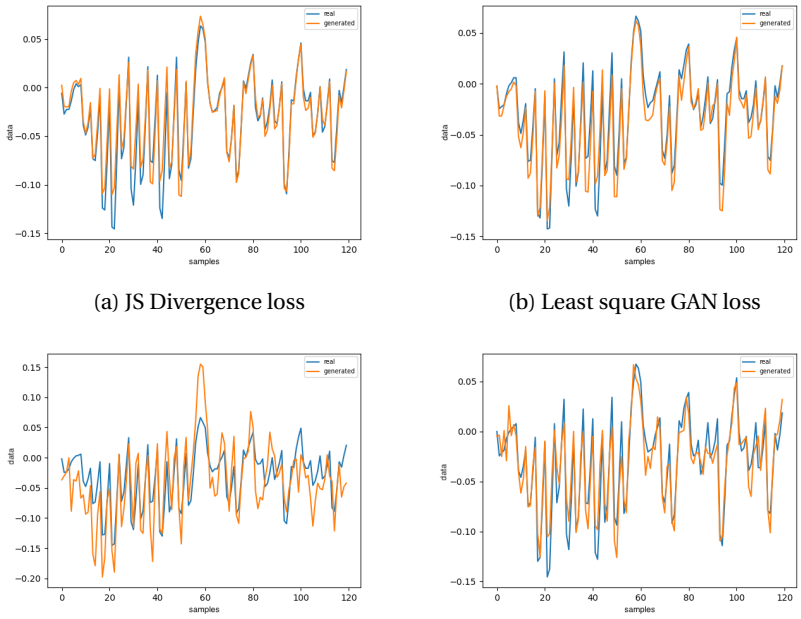
6. Number of hidden layers : The number of hidden layers has a direct impact on the evaluation metrics of the generated traces. If a model is too simple, the generated traces are noticeably different from the validation/training set. A complex model with too many layers performs noticeably worse on both DTW distance and attackability if limited training samples are used. Naturally, with more training data the model performance gets better.

7. Width of layers : The wider the layers, the more the number of trainable parameters. This is why the wider GANs perform well only when sufficient training data is available. Using very complex GANs with limited training data gave substantial worse results as compared to using simpler GANs on the same training set. Also complex GANs seemed to overfit on the training data, in the sense that the attackability on the training set was much higher than that on the validation set.

8. Size of input noise vector : The size of the input noise vector must not be disproportional to the size of the hidden layers. Extremely large noise vectors give sub-optimal results. For VCD conditioning, just modulating the amplitude of the VCD transition list so as to introduce stochasticity is sufficient. This is analogous to the MelGAN [77] architecture, where the authors randomly modulate the mel-spectrogram (which acts as an input to the GAN) for audio generation.

9. Optimizer : For most popular GAN models, the de facto optimizer is Adam [154]. When comparing the results with SGD with momentum [155] and RMSprop [156], Adam outperformed the other optimizers in both DTW distance and attackability.

To sum it up, GAN architectures for power trace generation *should* have hyperparameters as in table 5.11. In addition to the discussed hyperparameters, the impact of loss function on GAN training is also substantial. As introduced in Section 4.3.2, substantial research has been done in the context of loss functions for GANs. Wasserstein GAN with gradient penalty is perhaps the most popular GAN loss function currently. However, it has also been claimed that the quality of the GAN generated samples are not substantially dependent on the loss functions [157]. Although there was some visual difference in the traces generated by the GAN when different loss functions were used, the attackability of the traces was very much comparable.



(a) JS Divergence loss

(b) Least square GAN loss

(c) WGAN loss

(d) WGAN-GP loss

Figure 5.24: Generated traces for different loss functions

As can be seen in Figure 5.24, the LSGAN, WGAN-GP and JS Divergences loss functions perform very similarly. However, the WGAN loss function seems to give sub-optimal results. This might be as a result of the clipping hyperparameter chosen while training the WGAN. So as to highlight the visual dissimilarities in the

traces, the scaler *inverse_transform* was not carried out. Given the results in this subsection, it is clear that the impact of loss function, as far as such short traces are concerned is minimal. Hence, all the results presented in the next chapter will be for the original JS divergence GAN loss [70].

6

Results and Analysis

This chapter explains the process of procuring the training data and details about training and testing the GANs. Section 6.1 explains the experimental setup needed to obtain the power traces used for training. Section 6.2 analyzes the results when GANs are conditioned on HW based labels for power trace generation. Similarly, Section 6.3 analyzes results for VCD conditioning. Finally, Section 6.4 discusses the results in further detail while highlighting the advantages and shortcomings of this framework.

6.1. Experimental Platform

Despite the existence of deep learning libraries for almost all major programming languages, python has emerged as the de facto langauge for most deep learning research in the recent history. This trend is due to two main factors. First, is the obvious ease of use of python over other languages and second is the ever growing library support. Challenging tasks like GPU programming through CUDA [158] have been substantially eased using libraries like PyCuda [159] and Numba [160]. Integeration of C/C++ using Cython [161] has also accounted for a big boon to python's success. All these factors have made it possible for libraries like PyTorch and Tensorflow to provide amazing performance along with the high level of abstraction. In this work, all the models have been trained using PyTorch [64] and Tensorflow 2.0 (with Keras interface) [116]. The reason why both of the libraries were used and not just a single one was simply to stay as close to the original implementations as possible. Most of the models made as a part of this work are based on existing popular implementations from the computer vision domain. Since the authors of these papers did not have a consensus on using a particular framework, the reference implementation were either in PyTorch or Tensorflow. For the sake of reproducibility, all the models in this chapter will be available in the ONNX format [162] so that they can be imported to any framework without any hassle.

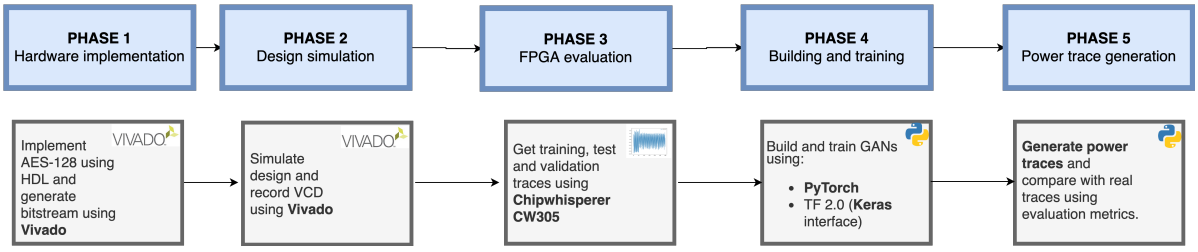


Figure 6.1: Implementation flowchart

A visual representation of the experimental platform can be seen in Figure 6.1. All models were written as python scripts and Jupyter notebooks were used in plenty. Whenever possible, both training and inference were performed on a MacBook Pro with 8259U coffee lake processor and 8GB of LPDDR3 SDRAM. For larger models, Google colab [163] was used. The implementation, as shown in Figure 6.1, is split into 5 phases. Each phase will be explained in detail next:

Phase 1: Hardware implementation. In this phase, a 128-bit AES is implemented using HDL. Since we also want

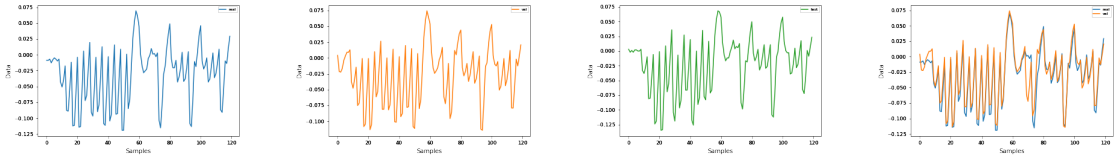
the GAN to generalize to different implementations, various different implementations are made with changes predominantly concerning the SBOX.

Phase 2: **Design simulation.** In this phase, the design is simulated using Vivado (this can also be done using ModelSim or QuestaSim) for a set of random plaintexts and a fixed key. These simulations are recorded in the form of a VCD file which is then processed and used as labels for the GAN.

Phase 3: **FPGA evaluation.** In this phase the implemented design is evaluated on a FPGA. Using Vivado, the bitstream of a selected design is generated. Next, the generated bitstream is used to program a chipwhisperer board and real power traces for training/validation/test datasets are recorded using the chipwhisperer python APIs. As the name suggests, the real traces for training are used for training the GAN. The real traces for validation and test dataset are only used for comparison with the GAN generated traces.

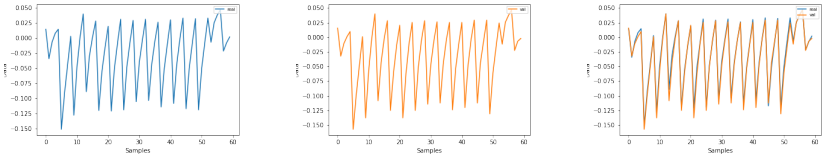
Phase 4: **Building and training GANs.** In this phase the GAN is made using PyTorch / Keras. The real power trace and labels (either leakage model labels or VCD labels) are used to train the GAN.

Phase 5: **Power trace generation.** In this phase, The trained GAN is used to generate power traces corresponding to validation / test labels. CPA analysis is done on the generated power traces. The evaluation metrics presented in Section 5.3 are also used to compare the generated power traces with the real ones.



(a) Training (b) Validation (c) Test (d) Comparison

Figure 6.2: Long Power traces



(a) Training (b) Validation (c) Comparison

Figure 6.3: Short power traces

The dataset used for our experiment will be as follows: The training set is of 10,000 samples. The validation set is 1,000 samples and the test set is 1,000 samples as well. The trace length is 120 time samples for the long traces (see Figure 6.2) and 60 time samples for the short traces (see Figure 6.3). Traces upto 300 time samples are also tested and some associated results will be discussed in the later sections. The training and validation set will consist of traces corresponding to the same implementation and the test set will have traces for a different implementation. As can be seen in figure 6.2 and figure 6.3, all three sets have visually indistinguishable traces. However, the amount of information leaked in the validation and test implementations is substantially different.

Next, each phase in figure 6.1 will be explained in detail.

Phase 1 **Hardware Implementation**

Step 1: Implement an unprotected AES-128 using any hardware description language so as to use for training the GAN.

Step 2: Implement AES-128 along with countermeasures against power side channel attacks, so as to use for testing and validation of the GAN.

Step 3: Generate a bitstream for every implementation using Vivado.

The output of this step are the designs and bitstreams corresponding to various AES implementations. Both protected and unprotected implementation are made so as to ascertain that the GAN can train on either of them and still at inference time generalize to the other implementation.

Phase 2 **Design Simulation**

Step 1: Decide a fixed key for the implementation and loop over a set of random plaintext values.

Step 2: Simulate the implemented AES-128 designs using Vivado for the fixed key and random plaintexts.

Step 3: Record a VCD file corresponding to the simulation of the AES-128 design.

Step 4: Process the recorded VCD file so as to obtain the transition list as explained in section 5.4.2.

Step 5: Repeat step 1 through to step 4 for all the implementations and segregate VCD transition lists on the basis of training, validation and test datasets on the basis of design type.

The output of this step are the VCD files corresponding to the AES implementations made in phase 1. This step is only required if the VCD transition list labels are used with the GAN. If the GAN is trained only using the leakage model labels, this step can be safely skipped.

Phase 3 **FPGA evaluation**

Step 1: Using the bitstream generated in phase 1, program a chipwhisperer CW305 board using chipwhisperer python APIs [135].

Step 2: Using the chipwhisperer python APIs again, record power traces for the same set of random plaintexts and fixed key as in phase 2.

Step 3: Repeat step 1 and step 2 for all the AES-128 designs and segregate them as training, validation and test datasets.

The output of this step are real power traces. Traces for one AES-128 design are used for training the GAN. Traces for the other designs are simply used for comparison with the generated traces.

Phase 4 **Building and Training**

Step 1: Scale the power traces using *StandardScaler* [107].

Step 2: Get labels associated with each power trace. Can either be leakage model labels. Can also be in VCD transition list labels.

Step 3: If the leakage model labels are used: for every batch, input random noise drawn from a normal distribution with zero mean and unit standard deviation into the generator, along with the label values. If the VCD transition list labels are used, modulate the transition list using a random value between 0.85 and 1 and feed the modulated VCD into the generator. This step yields the generated traces.

Step 4: Set validity labels as one for real traces and zero for generated traces. Pass generated traces through the discriminator and save the output. Do the same for real traces.

Step 5: Use the saved discriminator outputs and the validity labels to compute binary cross entropy loss for the discriminator. Backpropagate the loss and take an optimizer step for the discriminator parameters.

Step 6: Flip validity labels, such that the labels for generated traces are one. Pass the generated traces through the discriminator and use discriminator's output to compute binary cross entropy loss again. Backpropagate the loss and take an optimizer step for the generator's parameters.

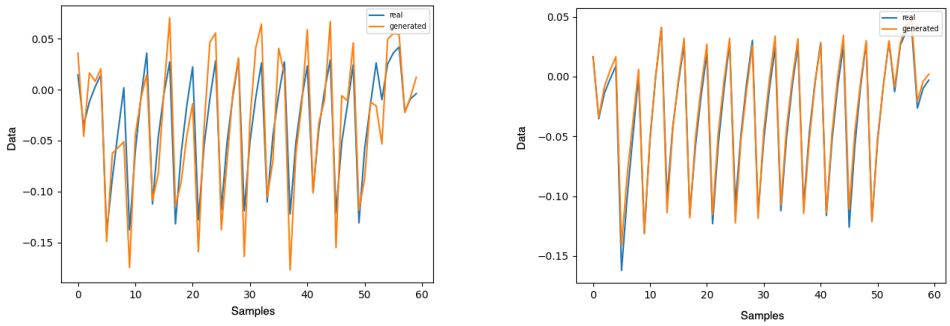
Step 7: Continue Step 3 to Step 6 till the epochs are completed, or the evaluation metrics get to a satisfactory level. Note that the results presented in this section are all for GANs trained for a maximum of 100 epochs.

A code-first introduction of these training steps can be found in appendix B. The output of this step is a trained GAN network. The generator part of the GAN can then be used to generated power traces.

Phase 5 **Power trace generation**

Step 1: Input a batch of random noise drawn from a normal distribution with zero mean and unit standard deviation into the generator, along with a batch of validation labels. Repeat this process until the desired number of traces are generated. Of course, the number of traces is limited by the number of labels available when using supervised GAN.

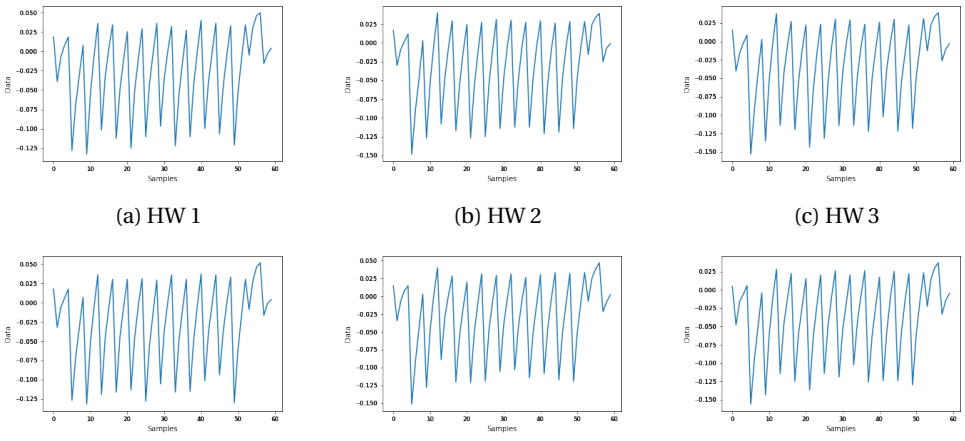
Step 2: Use *inverse_transform* to reverse the effects of *StandardScaler* on the generated traces and then compare them to the real traces.



(a) Without scaling (b) With scaling

Figure 6.4: Effect of Scaling on GANs

Scaling the power traces before training (and the corresponding inverse-transform during inference) serves as a correction step for the generated traces). For example, in figure 6.4a it can be seen that the generated traces without any scaling is definitely not as close to the real traces as compared to that in figure 6.4b.



(a) HW 1 (b) HW 2 (c) HW 3

(d) HW 4 (e) HW 5 (f) HW 6

Figure 6.5: Power traces corresponding to different *mode(HW)* classes

For our use-case this scaling procedure is ideal, given that the inter-trace difference is minimal for different leakage model classes (as can be seen in figure 6.5). This minimal difference is what is leveraged by the side channel attacks. Hence, unnecessary variations in the traces is not required. If anything, it can introduce more noise in the traces which can have a detrimental impact on the cpa attack ranks as highlighted in Section 5.4.3. Next, we discuss the metrics to compare the real and the generated traces as well as metrics to compare different GANs.

Compare Traces

Step 1: Attackability: It is the number of attackable bytes out of 16 using a CPA attack. The aim will be to be as close to the real power traces' attackability as possible. Calculating these attack ranks is very time consuming especially since most open source APIs are written natively in python [135]. Hence, to accelerate the attack process, Numba [160] acceleration is used. Appendix D explains this in detail.

Step 2: Visual similarity: To quantify visual appearance, mean dynamic time warping [124] distance will be used. Ideally this distance should be zero.The lesser the distance the closer the generated traces are to the real traces

Step 3: Conformance style testing: Both signal to noise ratio (SNR) and test vector leakage assessment (TVLA) will be used to compare the real and the generated traces. Ideally, the SNR peaks for both the real and the generated traces should follow the same trend. Similarly, the number of points failing the leakage assessment should follow a similar trend in both real and generated traces.

Step 4: Classifier similarity : As stated in Section 5.3, the aim of this evaluation metric is to compare the test accuracy of a HW classifier on the real traces and the generated traces. If the classification accuracy is similar for both real and generated traces and if the classifier correctly classifies the traces to their corresponding HW category, it can be said that both the set of traces are indeed very similar. This similarity is not just in terms of the visual appearance but also in terms of the information contained in them.

Compare GANs

Step 1: Compare GAN generated traces with the real traces using the aforementioned metrics.

Step 2: Compare the training time. Since one of the aims of this study is to speed up the process of evaluating countermeasures, it is critical that the model does not only perform well but can also be trained in a reasonable amount of time.

Step 3: Compare the Inference time. Similar to the training duration, the inference duration or the time taken to generate power traces should be as low as possible

Step 4: Compare the number of trainable parameters. The probability of a network to overfit on given training data increases with a growing number of trainable parameters. As a result more training data is required. In the context of GANs, Zhao *et al.* [118] demonstrated that without differential augmentations complex GANs perform very poorly if a limited training set is used. However, a very simple generative model might have a large bias and hence might not be able to model the characteristics of power traces well. Hence there is a trade off that must be considered when designing GAN architectures.

Of course, the value of each of these evaluation metrics will change as the amount of training data changes. For some use cases, there might not be any restriction on the amount of training data and in some cases, there might be. Hence, using 10,000 training samples is a good middle point to evaluate the performance. Some scenarios, where increased training set substantially improves the performance, will be highlighted in the text. ONNX diagrams of all the models are plotted in appendix A. These models have been plotted using Netron [164].

6.2. GANs WITH HW BASED LABELS

In this section, results for GANs with HW based labels will be discussed. Calculating labels for HW based conditioning involves using the known key value, which makes it somewhat trivial to evaluate the attackability of the generated traces. However, from the perspective of the template making step in template based attacks (TBA) (as in Section 3.4), it makes a lot of sense. While a template is populated for TBA, the attacker has access to not only a similar device as the one they are supposed to attack, but also have complete access to plaintexts and keys.

Tables 6.1 and 6.2 presents the results of long and short trace models, respectively. The first row of both tables lists the evaluation metrics on real traces. When comparing traces with themself, the Dynamic Time Warping (DTW) distance is zero. For the sake of comparison, when a power trace was compared with random

Table 6.1: GAN model comparison for HW label and long traces

Model	Label	DTW	Validation attackability	Test attackability	Train time (s)	Inference time (s)	Generator's trainable parameters	Discriminator's trainable parameters
Real traces	-	0.0	15	0	-	-	-	-
MLPGAN	-	1.04	0	0	107.5	0.028	26,584	25,857
INFOGAN	mode	0.92	4	3	141.5	0.033	26,560	d = 26,081 q = 31,819
	round	1.16	15	14	131.7	0.041	26,560	d = 26,081 q = 31,819
	ceil	1.18	16	16	135.6	0.039	26,560	d = 26,081 q = 31,819
	floor	0.91	16	16	138.5	0.037	26,560	d = 26,081 q = 31,819
	single	0.95	1	1	121.1	0.034	26,560	d = 26,081 q = 31,819
MLP-CGAN	mode	1.169	3	2	201.536	0.043	26,994	27,675
	round	1.099	13	12	208.422	0.047	26,994	27,675
	ceil	1.108	16	16	207.437	0.042	26,994	27,675
	floor	1.101	16	16	201.546	0.030	26,994	27,675
	single	1.104	1	1	200.047	0.030	26,994	27,675
ACGAN	mode	1.105	3	2	160.82	0.034	26,994	26,154
	round	1.043	11	10	157.25	0.029	26,994	26,154
	ceil	1.093	15	15	161.61	0.036	26,994	26,154
	floor	1.077	15	15	168.83	0.031	26,994	26,154
	single	1.109	1	1	163.65	0.028	26,994	26,154
DCGAN	mode	1.081	4	3	612.8	0.081	13,843	13,843
	round	1.095	15	14	615.6	0.078	13,843	13,843
	ceil	1.118	16	16	612.8	0.070	13,843	13,843
	floor	1.165	16	16	613.5	0.073	13,843	13,843
	single	1.177	1	1	602.2	0.075	13,843	13,843
CNN-LSTM GAN	mode	1.078	3	3	621.6	0.0806	13,843	14,203
	round	1.001	16	16	611.1	0.0855	13,843	14,203
	ceil	1.001	16	16	602.8	0.0855	13,843	14,203
	floor	1.002	16	16	609.2	0.0872	13,843	14,203
	single	1.012	1	1	603.5	0.109	13,843	14,203
CCGAN	mean	1.089	14	15	187.6	0.0379	25,048	25,889

noise in Section 5.4.3, the DTW distance was *63.539*. The attackability of the validation traces is 15 and that of the test traces is zero. All other rows list evaluation metrics on the generated traces of different Models (using different label processing options). The mean (DTW) distance of the generated traces from the real traces is in the range of *0.9* to *1.2*. A low DTW distance (like that in the range *0.9* to *1.2*) helps us conclude that all the architectures are successful in learning the structural characteristics of the power trace. However, just learning that does not guarantee that these traces carry information that can be leveraged by side channel attacks. For example, the *mode* and *single* label processing for all the Models have low DTW distance like other label processing options, however the validation and test attackability is much lower than the other label processing options. For *ceil*, *floor* and *round*, the validation and test attackability is much higher than that in the real traces. This means that the information being leaked by the generated traces is much more than that of the real traces. This is expected behaviour since we are feeding the generator with the same value that we are trying to obtain using the side channel attacks, albeit in a processed form. The number of trainable parameters for both generator and discriminator are fairly similar for all GANs except those that use convolutional layers. The trainable parameters are higher in the dense architectures since unlike convolutional layers, there is no weight sharing in dense layers. The convolutional GANs take longer to train than the others since convolutional layers are slower to train that dense layers. The inference time is almost comparable for all the architectures. The Information maximizing GAN (INFOGAN) takes much less time to train as opposed to the other GANs even though it clearly has more parameters. This is because it was trained using Tensorflow which was able to leverage some advanced vector extensions, like the *avx2 fma* acceleration and certain other mlir graph optimizations to speed up the training process. It must be mentioned that the timing information mentioned in this chapter is not very reliable, since it was not measured multiple times due to computational constraints. Also, as might be evident due to GANs stochasticity, both attackability and DTW can have minor variations when trying to reproduce results. Having said that, of the seven architectures compared in the

Table 6.2: GAN model comparison for HW label and short traces

Model	Label	DTW	Validation Attackability	Train time (s)	Inference time (s)	Generator's trainable parameters	Discriminator's trainable parameters
Real traces	-	0.0	0	-	-	-	-
MLPGAN	-	0.376	0	117.5	0.019	18,844	18,625
INFOGAN	mode	0.429	2	110.1	0.025	18,880	d = 18,401 q = 24,139
	round	0.431	14	112.1	0.021	18,880	d = 18,401 q = 24,139
	ceil	0.435	16	113.6	0.030	18,880	d = 18,401 q = 24,139
	floor	0.429	16	110.9	0.028	18,880	d = 18,401 q = 24,139
	single	0.431	1	119.1	0.031	18,880	d = 18,401 q = 24,139
MLP-CGAN	mode	0.452	1	136.8	0.015	14,806	17,755
	round	0.461	14	142.1	0.014	14,806	17,755
	ceil	0.441	16	139.7	0.016	14,806	17,755
	floor	0.472	16	133.2	0.017	14,806	17,755
	single	0.492	1	131.8	0.015	14,806	17,755
ACGAN	mode	0.453	1	140.2	0.022	17,622	16,714
	round	0.453	14	138.2	0.019	17,622	16,714
	ceil	0.453	15	135.3	0.022	17,622	16,714
	floor	0.450	16	130.4	0.017	17,622	16,714
	single	0.451	1	134.8	0.015	17,622	16,714
DCGAN	mode	0.471	4	453.8	0.088	11,323	13,734
	round	0.421	16	470.2	0.085	11,323	13,734
	ceil	0.418	16	490.2	0.088	11,323	13,734
	floor	0.465	16	481.1	0.089	11,323	13,734
	single	0.477	1	480.8	0.081	11,323	13,734
CNN-LSTM GAN	mode	0.463	3	483.8	0.086	11,323	12,601
	round	0.459	14	490.2	0.081	11,323	12,601
	ceil	0.459	14	484.6	0.087	11,323	12,601
	floor	0.460	15	489.3	0.089	11,323	12,601
	single	0.450	1	490.1	0.085	11,323	12,601
CCGAN	mean	0.312	16	183.1	0.021	17,308	18,209

table 6.1 and 6.2, the following conclusions can be drawn:

1. *Conditioning the GAN is important.* Completely unsupervised GANs fail to generate attackable traces. This is expected since the side channel attacks are heavily dependent on alignment and leverage correlation between the power traces and the leakage model to guess key values. If the power traces do not correspond to the encryption that it is supposed to represent, side channel attacks will fail. Hence even for the INFOGAN (even though it is an unsupervised GAN) at inference time the correct label values were required to make sure that the power trace corresponds to the correct plaintext and key. This is because, unlike computer vision, the power trace can not just be visually distinguished into their corresponding class. Even if that was possible, knowing the correct leakage model is an absolute necessity so that the correlation between the power traces and the leakage model can be leveraged to guess key values.

2. *Different architectures and label options have different training data requirements.* Just going by the results in the table, it is clear that some label options perform better than the others. The same can also be said about some architectures. However, it must be pointed out that as the amount of training data increases, the performance difference between many label options and architectures decreases. For example, in other experiments the mode of HW label gave very impressive results. However it was trained with 40,000 samples and around the same number were generated. It was also observed that the

gap between ceil, floor and mode label performance reduces as the amount of training data increases. The single byte HW label did not perform much better as the amount of training data increased. Hence, certain architectural modifications, data augmentation or increase in the amount of training data will effect the performance of each architecture / label. There is a noticeable increase in the stability of the GAN results once more training data is available. Also, the power trace length has implications on the GAN performance, this can be attributed to the change in trainable parameters when the power trace length is changed. Keeping the results of both table 6.1 and table 6.2 in mind, it can be concluded that given the amount of training data and the power trace length, the label option choice will play a critical role in the GAN performance.

3. *Visual similarity does not guarantee attackability*: Comparing the DTW distance of an unsupervised GAN to that of other supervised GANs, it is quite evident that the DTW score does not seem to have a stark effect on the attackability of the traces. This is also because the power traces corresponding to different label options are structurally very similar. However, attackable power traces must have certain minor variations that play a role in leaking the information required by the side channel attacks. That task is performed by the labels used during training / inference.

4. *HW labels fail to portray an accurate leakage assessment*: For the implementation used for longer traces: the generated traces (as in table 6.1), should have a validation attackability of around 12 out of 16 bytes and test attackability of 0 out of 16, when using 1000 traces. As for the shorter traces (as in table 6.2), the validation attackability should be 0 out of 16, meaning that the implementation is very hard to attack. Instead, in some scenarios all of the bytes are attackable meaning that the traces are leaking much more information than they otherwise should and in some cases the attackability is very low meaning that the traces are leaking less information that they should.

Now that a few GAN models that perform well on both the evaluation metrics are available, a detailed study on their leakage assessment can be done. The aim, as stated before, is to be as close to the actual leakage as possible. The methodology for the same is as follows.A DCGAN is trained on a unprotected AES-128 implementation. For inference, an AES-128 implementation with masking is considered. The value of the mask-pairs is changed, while the same set of random plaintexts and fixed key is used. For each of the mask pairs, traces are collected and a CPA attack is carried out. Inference labels for the masked implementation are also collected and using them and the trained GAN, power traces are generated. If the number of attackable bytes (attackability) for the real traces is similar to that of the generated traces, it can be concluded that the generated traces can be a replacement for the leakage assessment of the design.

The 10 masks used for this experiment as well as the attackability of the real and the generated traces can be seen in table 6.3. From the table, it is evident that for a GAN conditioned on processed HW labels, the attackability is inline with the actual traces. In this case, the label is computed and the effect of the mask is taken into account before deciding the modified label value. The disparity between the actual and the masked labels is what impacts the attackability of the traces.

Table 6.3: masking evaluation : unknown masks

Mask Pair	Mask 1	Mask 1 HW	Mask 2	Mask 2 HW	attackability : real	attackability: generated
1	163	4	39	4	0	0
2	228	4	105	4	0	0
3	172	4	142	4	0	0
4	240	4	184	4	0	0
5	114	4	154	4	0	0
6	133	3	2	1	0	0
7	110	5	233	5	0	0
8	208	3	17	2	0	0
9	213	5	118	5	0	0
10	189	6	255	8	0	0

As explained in Section 3.5, the masking countermeasure involves two mask values. The purpose of the first mask is to add an offset to the location in the SBOX table, whereas the purposed of the second mask is to change the contents of the SBOX table. . A GAN can only outputs power traces that are similar to the

ones it has been trained on. The label values govern the generator's output and the GAN fails to introduce misalignment that comes from the added operations when masking is used. Meaning that evaluating the efficacy of countermeasures only using the HW label is not a robust technique.

To explain the same in more detail, consider the results in table 6.3. Both the real and the generated traces are not attackable when masking is used. However, as stated before, this is only true for the generated traces since the HW labels get transformed with masking.

Table 6.4: masking evaluation : known masks

Mask Pair	Mask 1	Mask 1 HW	Mask 2	Mask 2 HW	attackability : real	attackability: generated
1	163	4	39	4	0	16
2	228	4	105	4	0	16
3	172	4	142	4	0	16
4	240	4	184	4	0	16
5	114	4	154	4	0	16
6	133	3	2	1	0	16
7	110	5	233	5	0	16
8	208	3	17	2	0	16
9	213	5	118	5	0	16
10	189	6	255	8	0	16

As in table 6.4, if the labels are corrected using the mask values, the attackability of the generated traces will be as much as the unprotected traces. This is because by using the values of mask1 and mask2, the effect of masking on the labels in undone. This is not ideal from a leakage assessment point of view due to two reasons. First, is the overarching dependence on the HW labels. This means that this technique is, at best, capable of evaluating the efficacy of countermeasure that impact the the SBOX (and hence the labels). Unfortunately, that makes this technique unfit for majority of the countermeasures. Second, is with regards to the inability to add misalignment that is caused by masking. The lack of attackability of the generated traces (as in table 6.3) is only because of the fact the the mapping between the leakage model and their corresponding power traces is impacted and not due to any change in the power traces themselves. Any added operation, irrespective of it being a countermeasure against side channel analysis or not, will have some implication on the structure of the power trace, which is sadly not the case here. This can be proved as follows, when the masks are used to correct the label (as in figure 3.8), the traces generated using the corrected labels are as attackable as the traces without any countermeasure. Table 6.5 lists the number of labels changed when the masks are unknown and when the masks are known. Of course once the known mask is used to correct the label values corresponding to every trace, the *labels changed* field become zero since the labels are exactly as they should be. However, without knowing the mask the labels can not be corrected. This means that for a power trace that should have a HW label 4, gets a HW label 2 (as in figure 3.8). Being a simple xor operation, there are also cases where the mask does not impact the value of the HW this is why in table 6.4 most of the labels change when masking is used but still some of the labels stay unchanged.

Hence, even though the HW based labels perform well as far as generating power trace close to the real traces are concerned, they fail to model the leakage behaviour of the power traces well. Depending on the labels used, the leakage is either way too much or way too less. The information presented in this Section indicates the importance of labels in the context of conditional GANs. Non informative labels cause the GAN to generate traces that although similar to the training set do not include information that can be leveraged by side channel attacks.

6.3. GANs WITH VCD LABELS

Now that the importance of the labels is discussed, it is clear that the labels used to condition the GAN must be correlated with the amount of the information leaked by the traces. Otherwise, the generated traces will fail to model the leakage behaviour of the power traces well. As a solution to this problem, value change dump files can be used. As explained in Section 5.4.2, these VCD files record all transitions in the design's wires, registers and parameters. As stated earlier in section 6.1, when using the VCD transition list labels, we do not make use of an additional noise vector. Instead, an architecture similar to the MelGAN [77] is used. Hence, the noise vector is removed and just the VCD transition is used as an input to the generator. To

Table 6.5: Impact of masking on HW labels

Mask Pair	Mask 1	Mask 1 HW	Mask 2	Mask 2 HW	unknown mask: labels changed*	known mask: labels changed*
1	163	4	39	4	3434	0
2	228	4	105	4	3369	0
3	172	4	142	4	3507	0
4	240	4	184	4	3380	0
5	114	4	154	4	3417	0
6	133	3	2	1	3573	0
7	110	5	233	5	3490	0
8	208	3	17	2	3652	0
9	213	5	118	5	3510	0
10	189	6	255	8	3703	0

*out of 5000

introduce stochasticity, the VCD transition list is multiplied by a random constant value between 0.85 and 1. For a trained GAN network, two main operations are carried out. Namely, GAN Validation and GAN testing. For GAN validation, it is tested using VCD transition list corresponding to the same design but with different plaintexts. For GAN testing, it is tested using VCD transition list corresponding to the same design but with different key and with a different AES-128 implementation (with masking countermeasure).

GANs discussed in section 5.4.2, namely MLPGAN and DCGAN, will be discussed in this section. The generated traces for both the cases can be seen in figure 6.6. The DCGAN generates traces that are almost indistinguishable from the validation data. The MLPGAN is not too far off, however the DCGAN seems to performs a bit better. Careful hyperparameter tuning might change these results in favor of either of the architectures.

Table 6.6: GAN Validation: CPA Key Rank for 1000 Traces

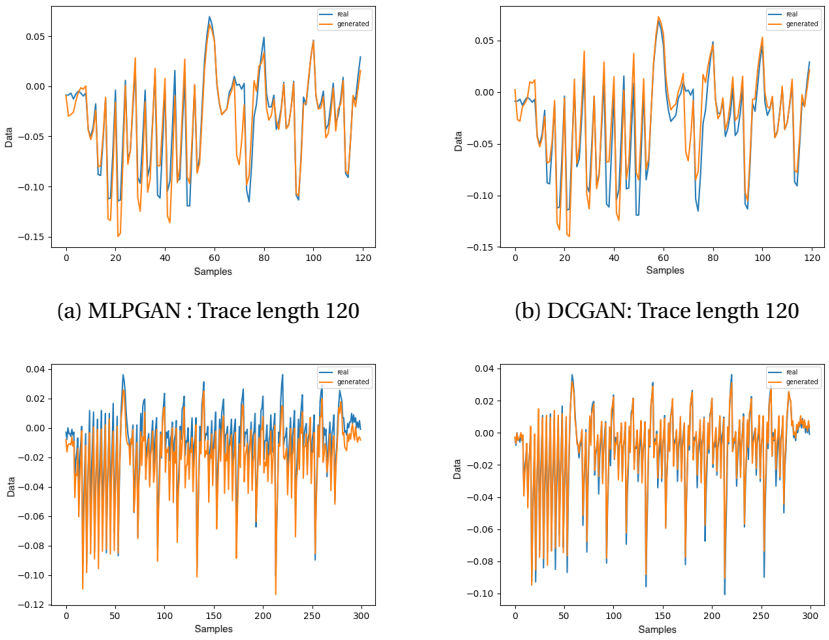
BYTE	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Real	1	1	1	1	1	1	1	9	1	1	1	1	1	2	1	1
MLPGAN	1	1	1	1	1	1	1	1	1	28	24	1	5	1	5	41
DCGAN	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	11

From the attackability point of view, again, the DCGAN produces traces where all but two bytes (byte 15 and 16) are not attackable. However, for the MLPGAN generated traces byte 10,11,13,15 and 16 are not attackable. This can be seen in table 6.6.

Table 6.7: GAN Testing

GAN type	Traces	Attackable bytes (out of 16)	Design resource utilization
MLPGAN	Generated: Masking	0	Uses 2511 LUTs and 3364 FFs
	Real: Masking	0	
	Generated: Unprotected	11	Uses 2430 LUTs and 3363 FFs
	Real: Unprotected	13	
DCGAN	Generated: Masking	0	Uses 2511 LUTs and 3364 FFs
	Real: Masking	0	
	Generated: Unprotected	14	Uses 2430 LUTs and 3363 FFs
	Real: Unprotected	13	

So as to test both the MLPGAN and DCGAN, VCD files corresponding to the same implementation (but different plaintext and key) as well as a different AES implementation (with the masking countermeasure) are used. As can be seen in table 6.7, for both of the scenarios the attackability of the real and the generated traces is not exactly similar for the MLPGAN. For the DCGAN, the attackability of the real and the generated traces



(a) MLPGAN : Trace length 120

(b) DCGAN: Trace length 120

(c) MLPGAN: Trace length 300

(d) DCGAN: Trace length 300

Figure 6.6: Real vs Generated traces : VCD Label

for both of the scenarios are very similar. Since the GAN generated traces follow the leakage trend of the real traces not just on the training set but also on the validation and test set, such a GAN based methodology using VCD labels can be extended to do design space exploration of different countermeasures.

Inline with the HW label methodology, DCGAN takes longer to train as compared to the MLPGAN. Training the DCGAN on 5000 samples takes 737 seconds for 100 epochs. The MLPGAN, on the other hand, takes 420 seconds for the same amount of training data and epochs. However, given that VCD processing step is much more time consuming than either of the training times , this is not the most important differentiating point between the architectures. There was also a noticeable difference in the training stability as the complexity of the GANs increased. Less complex GANs (on the basis of width for MLP GAN and number of filters for DCGAN) were also able to generate decent results. However, for the same GAN hyperparameters and training data, the attackability showed variations. This might be attributed to underfitting, which is a potent issue in GANs. Adlam *et al.* [165] prove that increasing the GAN complexity leads to reduction in the generalization gap.

6.4. DISCUSSION

In this section, the results presented in the previous two sections will be discussed with the aim of highlighting the advantages and disadvantages of using the GAN methodology of generating synthetic power traces as well as finding the similarity between the real and the generated traces.

6.4.1. DRAWBACKS OF AUTOENCODERS

Since the disparity among power traces is almost minimal, the autoencoder architecture seems very capable for the task of power trace generation. Hence, it is important to address why autoencoders (or just the decoder network) were not used to generate power traces. Since the latent space is not of paramount importance for our use case, just using decoder-like architecture taking embedding of labels as input and outputing power traces is also an interesting idea. It is true that both such architectures can also be used to generate power traces on the basis of either HW or VCD label values. However, the autoencoders failed to generalize well to different implementations when using the VCD methodology. When using the HW labels, the results were exactly inline with those of the conditional MLPGAN in table 6.1. This is expected behaviour since even at inference time, the possible labels can not be different from the ones that the model has been trained on.

Table 6.8: Comparison between GAN and modified AE

Model	Loss function	Test Attackability		
		Real	Generated : ceil-HW label	Generated : VCD label
Modified autoencoder*	MSE	13	16	5
Decoder-only*	MSE	13	16	4
VAE**	MSE + KL Div	13	16	7
DCGAN	BCE	13	16	13

*ONNX diagrams on page 98 **ONNX diagram on page 99

However, using the VCD labels the results were not comparable to the GANs. This is because at inference time there are variations in the VCD transition list as compared to to the training transition lists. The autoencoder model can not accommodate these changes well. The conditional VAE model, just like other deep generative models, can be applied to the task of generating power traces even with VCD label. Since the VAE was not rigorously tuned like the GAN, it gave suboptimal results. The tuned GAN showed good results for both HW and VCD labels. This can be seen in table 6.8. Thorough experiments with the VAE and its ensembles should yield equivalent results to GAN.

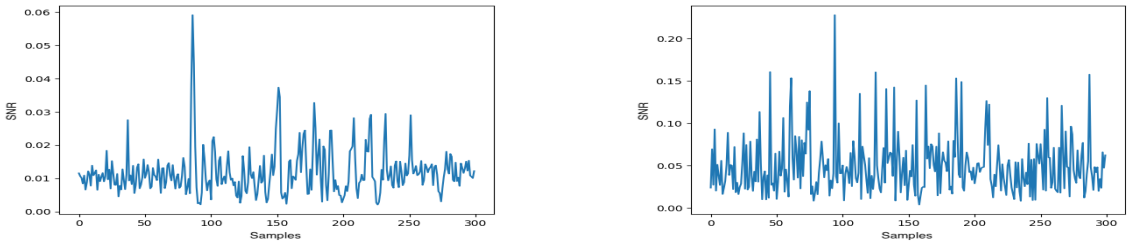
6.4.2. THOROUGH EVALUATION OF TRACES

To begin with analyzing the results, there is the discrepancy in GAN performance when using different processing functions for processing the HW labels. In other words, when using the processed HW labels some processing functions perform better than the others. The difference is even more pronounced when using lesser number of traces. As stated in section 5.9, the reason behind this is can attributed to the innate class imbalance problem of hamming weight/hamming distance leakage models. It is possible that since the GAN is being shown much more traces of a particular class label (HW 4), it might output traces that look more like that specific class label even if other labels are used. In table 6.2, the ceil, floor and round labels perform better than the others since they are able to split the traces into two equal size classes. A solution to the class imbalance problem, to some extent, is using SMOTE to process the training data for the GAN. The ACGAN actually performs much worse when SMOTE is not used. This is because the loss function also has a classification component, which perhaps hampers the ACGAN ability to produces traces corresponding to all classes well. For example, if there is binary classifier that is trained on an imbalanced dataset such that one class constitutes 90 percent of the data and the second class only constitutes 10 percent of the data, it is likely that the classifier just classifies everything to be of the first class. Since even in that case the classification accuracy will be 90 percent.

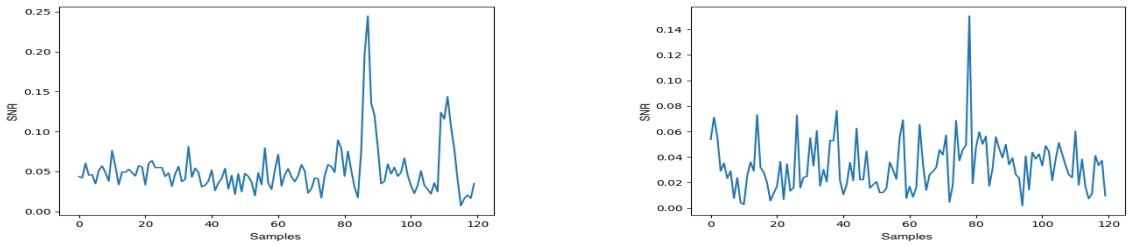
Next comes the question of the similarity in real and generated traces. Till now, we used evaluation-style comparison between the real and the generated traces. The similarity in the traces can also be quantified using conformance style testing like signal to noise ratio. The SNR of the real traces can be compared with the SNR of the generated traces. The peaks in the SNR signifies the amount of information that is leaked. From the SNR plots in figure 6.7 the noise (termed as *leakage noise* by Wang *et al.* [117]) is quite evident , but the peaks seem to be just like in the real traces. All three plots are for different trace lengths so as to highlight that the GAN performs quite similarly even for quite different trace lengths. These SNR plots of course change on the basis of the byte used and also in between runs because of GAN's stochastic nature. Hence, in order to compare the real and the generated traces for all bytes, the SNR plot in figure 6.7c was made. Here, the different colors represent different bytes. It is evident that the generated traces leak more information than the real traces causing the generated traces to be much more attackable than the real traces (as in table 6.2). When comparing the performance of the MLPGAN and DCGAN on the basis of the SNR plots, the MLPGAN generates traces wherein the 'leakage noise' is much more prominent as compared to the DCGAN (compare figure 6.7aand 6.7b to figure 6.7c)

Just like SNR, even TVLA can be used for conformance style testing. When using the VCD label and comparing the MLPGAN and the DCGAN architectures, the number of leakage points are as in table 6.9. In all the cases the number of leakage points are lesser in the masked implementation, than in the unprotected implementation. Exactly as it should be.

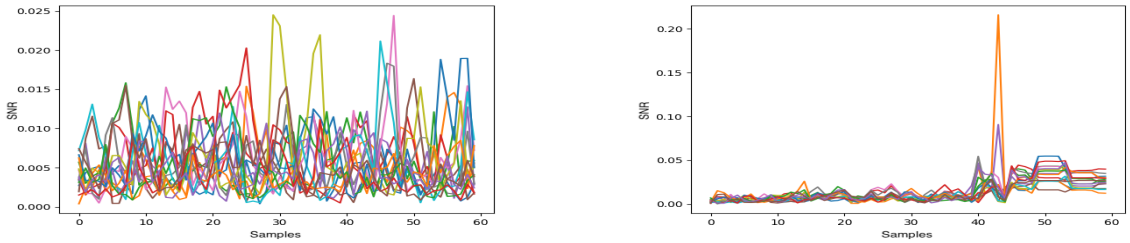
However, it is evident that for the MLPGAN the number of leakage points are more for both the unprotected and the masked implementation. I attribute this to the 'leakage noise' which was also prominent in



(a) HW label



(b) VCD Label



(c) HW Label : all 16 bytes

Figure 6.7: SNR plots of real (*left*) and generated (*right*) traces

Table 6.9: Comparing TVLA leakage points

Traces	Unprotected	Masked
Generated Traces : DCGAN	22	19
Generated Traces : MLPGAN	74	57
Real Traces	15	13

the SNR plots of the MLPGAN generated traces. Just like the SNR results, even these results change between runs however the leakage trend is almost always, correctly depicted.

Now that the real and generated traces have been compared using both evaluation and conformance metrics, as a final step, classifier similarity can be checked. If we compare the real and the generated traces using a trained HW classifier, it will be interesting to see how similarly does the classifier act on real and generated traces. A visualization of the classifier can be seen in figure 6.8.

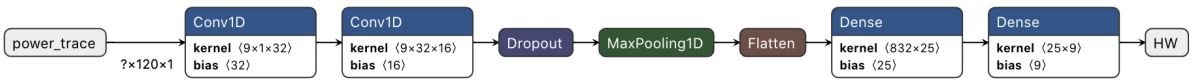


Figure 6.8: HW Classifier

Table 6.10 lists the results for classification similarity and accuracy. For the sake of clarity, consider that there are 100 traces in the test set. If the classifier correctly outputs 50 percent of the hamming weights, the classification accuracy is 50%. Classification similarity, on the other hand, aims to compare the percentage

Table 6.10: Classifier results

	Generated power trace	Real Power trace
Classification Accuracy	53.1%	53.5%
Classification similarity	87.1 %	

of real and generated samples that were similarly classified. Hence the classification results for both real and generated traces are compared. If 80 percent of the real and generated traces are similarly classified, the classification similarity is 80 percent. The idea is to input real and generated traces corresponding to different HW labels and comparing the classification results from the classifier. If the classifier outputs the same HW values for both the real and the generated traces, the real and the generated traces are indeed very similar. The classifier was trained using the *ceil(mean(HW))* label and SMOTE [166] upsampling was used so as to counter the class imbalance problem [57]. Different label options give different accuracy results. However from the point of view of similarity between the classifier's performance on real traces and on the generated traces, the difference is quite minimal. This is because even if the accuracy is low, similar values for both real and generated power traces on the classification accuracy and classification similarity metrics is sufficient for comparing the real and generated traces. Looking at the classifier's almost identical performance for both the real and the generated traces, we can be even more certain of the design leakage assessment ability of the generates traces.

6.4.3. VCD'S INABILITY TO OBVIATE GANS

Looking at the results presented in Section 6.3, specifically the information contained in the VCD transition list makes one feel why is the processed VCD not used on its own for leakage evaluation? The reason behind this is that the VCD transition array is innately much more cleaner than a power trace since it does not model any non-idealities as timing violations and hence is always much more attackable. This gives a false sense of weakness in the design. For example, if the generated traces have an attackability of 14 and the real traces have an attackability of 13, the VCD transition array has an attackability of 16. Just by depending on the VCD transition array attack ranks, it seems that the design is absolutely insecure. However, it is only using the GAN generated traces that a reasonable evaluation of the design can be made. Secondly, the GAN technique has the capacity to imitate the design's technological behaviour of the circuit, which is useful for various countermeasures and is not visible using simply VCD.

6.4.4. POTENTIAL SPEED-UP

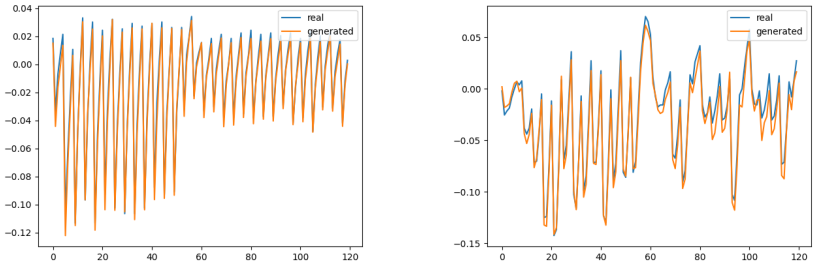
One of the most promising features of the GAN methodology for generating power traces is that it guarantees a very high speed up. To quantify the exact value of the speed-up, the time for generating traces using a GAN can be compared to the time required to procure power traces using the chipwhisperer platform. Not accounting the time taken to generate the AES bitstream and the time taken to program the chipwhisperer FPGA; the time taken to procure 10 Million power traces using the chipwhisperer APIs is around 9.6 days whereas the time taken to generated the same number of traces using the GAN (including the compute heavy VCD processing step) is around 4.6 days. This means that even with an un-optimized line by line processing python script for VCD processing, the GAN methodology is 2.1 times faster than traditional power trace procurement. As compared to the CAD based power trace generation, as in [167], the GAN methodology is around 150 times faster.

6.4.5. LIMITATIONS IN GENERALIZING

Talking about leakage assessment, the test implementation considered till now had power traces that although similar to the training set had very different attackability (using a masking countermeasure). However, if the implementation is drastically changed the structure of the power trace also changes and so does the the VCD transition list. If there is no change in the vcd / power trace length, the changed VCD transition list can be accommodated using three techniques. First, it is possible to categorize the multiple power traces using different label values. Here, in addition to the VCD conditioning, there is an extra embedding layer for labels (used for categorizing the power trace types). Second, transfer learning can be employed and third, the GAN can be retrained on the new set of training data. If there is a change in the vcd / power trace length as well, then the architecture must be changed so as to accommodate the new power trace length. In such a case

transfer learning can be or the GAN can be completely retrained.

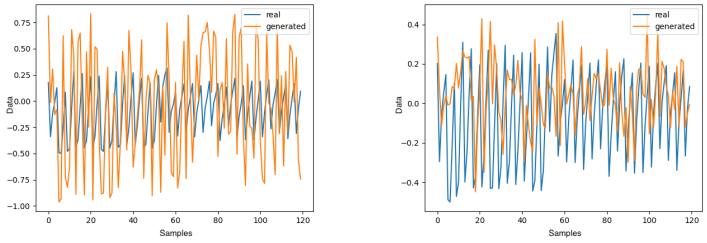
Using labels for denoting the type of VCD/Power trace work well as far as the visual appearance of the different traces are concerned. ONNX diagram for the architecture used for this experiment is available on page 97. In Figure 6.9, generated power traces for different implementations can be seen. Here, the aforementioned conditioning is used and the GAN is able to generate power traces corresponding to different implementation just using the power trace category label. However, the attackability seems to take a hit in this case if Standard scaler [107] is used for scaling the VCD. This is be because, even when scaled using standard scaler, different VCD transitiona list will have a different range that has an impact on the GAN training / performance. Hence, if multiple such VCD transition lists are to be used using MinMaxScaler [106] gives better results.



(a) Generated power trace: category 1 (b) Generated power trace: category 2

Figure 6.9: Using category labels for power traces

The second option in such scenarios is transfer learning. The idea of transfer learning in the context of GANs is not immensely researched, however fairly recently there have been some papers addressing this issue. Wang *et al.* [168] propose the use of transfer learning by using pretrained Generator and Discriminator networks for use cases with limited training data. Figure 6.10 presents the real vs generated traces with and without transfer learning. For both cases only 100 traces were used to train. In the transfer learning case, pretrained generator and discriminator networks (on 5000 training samples) were used. In both cases, *inverse_transform()* was not used so as to fairly highlight the impact of transfer learning.



(a) Without transfer learning (b) With transfer learning

Figure 6.10: Real vs Generated traces

Using this methodology the generated traces visually look much more similar, although the generated trace is definitely not as good as in the previous experiments. For both the cases, the attackability was zero, just like the real traces. This of course does not mean that the GAN learns the leakage behaviour correctly since the zero attackability can also be attributed to one of the GAN's failure modes. However, it must be pointed out that this technique is only worth considering when just a few hundred power traces can be taken from the new implementation. Exactly similar results, if not better, can be obtained if the GAN is completely retrained without any transfer learning, albeit with more training data as highlighted earlier in Section 6.3. Transfer learning might be useful when using very large GAN models for very long power traces like those presented in Section 5.5.3, however for the trace length and implementations considered in this work just retraining the GAN is probably much more intuitive.

7

CONCLUSION

This chapter summarizes the work presented in this thesis along with a few future research avenues. Section 7.1 gives a chapter wise summary and Section 7.2 gives answers to the research questions presented at the beginning of the thesis. Finally, section 7.3 presents possible future research that can be carried out using this work as a baseline.

7.1. SUMMARY

Chapter 1 introduced the impact of hardware attacks in the cyber-security domain with a focus on side channel analysis, specifically power-based side channel analysis. The chapter briefly introduced existing leakage assessment frameworks and highlighted their shortcomings. These shortcomings became the motivation behind this thesis, i.e, requirement for a tool that can robustly perform pre-silicon leakage assessment.

Chapter 2 of this thesis explained the prerequisite cryptography concepts. Starting with a brief history of the domain and progressing towards different kinds of cryptographic algorithms, namely symmetric and asymmetric. Many popular algorithms were discussed in detail.

In chapter 3, a classification of the side channel attacks against cryptographic algorithms was presented. The focus here was, again, power-based side channel attacks. Countermeasures against such side channel attacks were also explained.

Chapter 4 covered the prerequisite deep learning concepts. Starting with a note on the history of deep learning, the chapter explained different kinds of neural networks along with information on building, training, testing and tuning them. Generative deep learning was also explained and a comparison between popular deep generative models was made.

In chapter 5, the methodology for training GANs for power trace generation was proposed. Every component of the GAN architecture was discussed so as to posit a set of architectures that can accommodate different power trace lengths and different label options.

Chapter 6 presented the results associated with the two forms of GAN conditioning, namely HW-based and VCD-based. The shortcomings of HW-based conditioning were highlighted and a robust framework for leakage assessment was proposed using the VCD-based approach. In the chapter discussion, the methodology is assessed in detail. The impact of using different implementations / technologies is also discussed along with highlighting the required architectural and training changes in such scenarios.

7.2. REFLECTION ON RESEARCH QUESTIONS

Analyzing the results presented in this thesis, the following answers to this thesis' research questions can be made:

1. **Can GAN generated traces be used for security evaluation of an IC at design time?**
   Yes. Given that the used labels carry sufficient information about the implementation (like the VCD label) it is possible to use GAN generated traces for leakage analysis and hence to carry out security evaluation of an IC at design time.

2. **Do different GAN architectures impact the quality of the generated traces?**
   Yes. Carefully tuning hyperparameters presented in chapter 5, it is possible to use GANs for power trace

generation. However different GAN architectures and different labels have an impact on the "quality" of the generated traces. If we are solely concerned with generating visually indistinguishable traces, it can be safely said that most GAN architectures will be able to generate good quality power traces. Since the word 'quality' is also concerned with the amount of information contained in the power traces, the methodology is heavily dependent on the labels that are used in the training process. Informative labels help in generating attackable power traces which contain sufficient information so as to be called good quality.

3. **What evaluation metrics can be used to evaluate the GAN generated traces?**
   The generated power traces can be compared to the real power traces using side channel analysis metrics like CPA attack ranks, TVLA and SNR. However, to accurately compare the visual appearance of the two, it is important to employ signal comparison metrics like dynamic time warping.

4. **How much speed-up can this GAN methodology provide over the standard CAD tools?**
   The answer, of course, depends on the type of CAD tool used for generating the synthetic power traces. However, as a rule of thumb, the CAD tools take at least a few seconds to generate every power trace ([167] takes 5.47s). The GAN methodology, with the compute heavy VCD procurement and processing step, can generate the same number of traces in 4.62 days . This means, for generating 10 Million traces (as is required for robust leakage assessment [22]) the speed up is around 140 times.

5. **How well do GANs generalize to different implementations?**
   Passably. In the experiments presented in chapter 6, only when both the VCD transition list and the traces for different implementations are similar, does the GAN perform well. To accommodate extremely different VCD transition lists and power traces, architectural changes are required. If the length of the VCD and the power trace are similar and only certain characteristics (like amplitude) change, an extra embedding layer can be used to signify different power trace types. However, if their length also changes, the GAN must be structurally changed before either being completely retrained or being retrained with transfer learning.

7.3. FUTURE WORK

In this section, future research avenues for enhancing the work presented in this thesis will be discussed.

- Encoding power traces as images: Hettwer *et al.* [169] demonstrated that encoding traces as images using techniques like Gramian Angular Field transformation [170] can increase the efficiency of deep learning based side channel attacks, especially for desynchronized and noisy traces. The generative models presented in this work can be extended to work on the image representation of the traces so as to enhance the profiling set.

- Experimenting with other models and ensembles: Generative deep learning is a very active research space and a number of new models and ensembles are being proposed quite frequently. Many of these existing generative models might fit in very well with the task of power trace generation and might outperform the GAN on metrics like training stability, quality and ability to generalize to different implementations. For example, some energy based models [171] have recently been shown to outperform GANs on many computer vision metrics. Similarly, deep generative ensemble methods might also be able to outperform the GANs presented in this work.

- Quantization and Inference time acceleration: Despite the immense speed up that the GAN methodology guarantees, there is still room for improvement for large GAN models. Specifically models with a Resnet [172] like architecture (for example MelGAN [77]). Since CUDA enabled GPUs are not ubiquitous, accelerating the GAN inference procedure is an interesting research avenue. Appendix C discusses the FINN compiler by Xilinx for inference time acceleration from a GAN perspective.

- Differential augmentation techniques : Proposed by Zhao *et al.* [118], differential augmentations for GANs are augmentation techniques for images like color,translation and cutout that help even large GANs perform very well on limited training data. The proposed transforms, as is, are not meant for power traces. Naively using the transforms does not help in the training process and in some cases even deteriorates the attackability of the generated traces. This is because the zero padded shifts, as a

result of the transforms like translation, introduce misalignment which in turn acts as a countermeasure against the attack. Detailed experiments with other forms of differential augmentations might result in better performance even with much less training data.

A

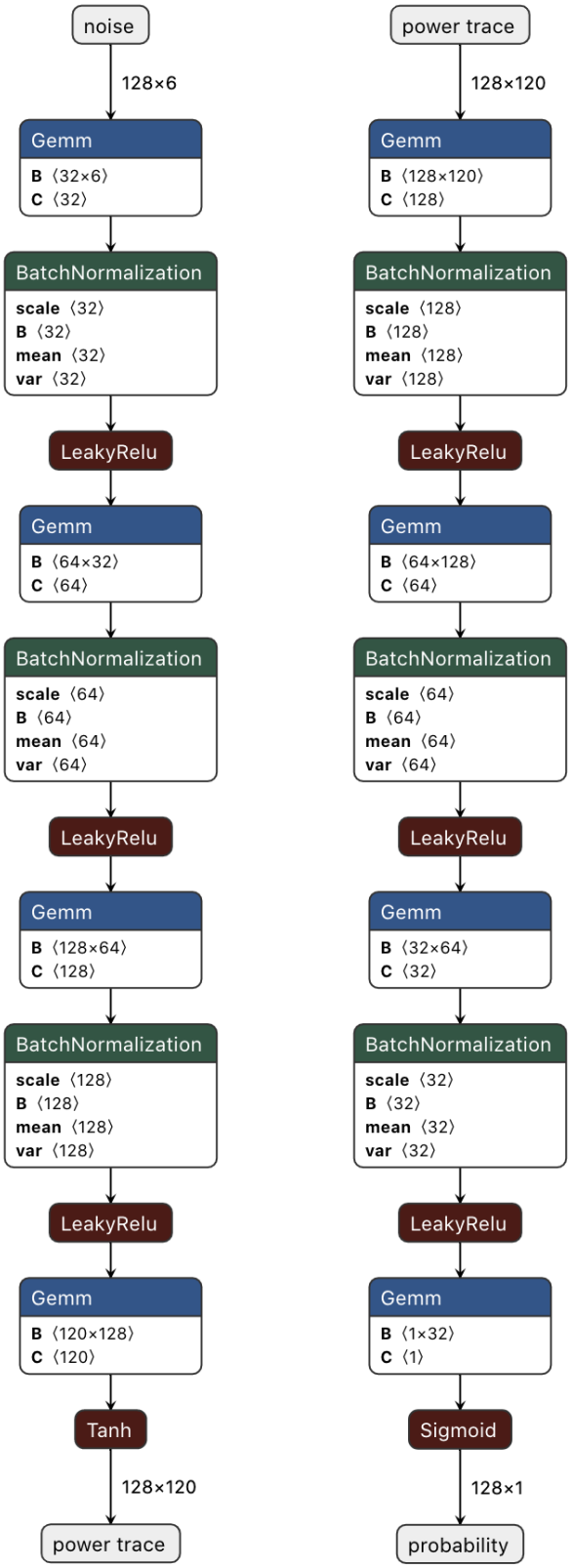
ONNX DIAGRAMS

The diagrams in this section have been generated using Netron [164]. Models were first saved into the ONNX format and then processed using Netron. ONNX or Open Neural Network Exchange aims to act as a framework independent medium for storing and exchanging neural network models. It does so by utilizing a common set of operators that act as building blocks for the neural networks. Different frameworks have different APIs to convert models to their corresponding models. For PyTorch, the API is *torch.onnx.export()* and it takes the model definitions, input names and dummy inputs as parameters. For Tensorflow, the API is *tf2onnx.convert.from_keras* and it also takes model definition and inputs as parameters.Once the ONNX models are available, they can be passed as arguments to the Netron app. Netron internally uses HTML and JavaScript to parse through the ONNX and generate a visual representation of the model. Before these visual representation are presented, a short introduction to some of the ONNX operators must me made. This is because all the models will be represented in terms of these operators. Although most of the operator names are very straightforward to understand, there are some operators that must be mentioned for the sake of clarity. A list of these can be found in table A.1.

Table A.1: ONNX operators

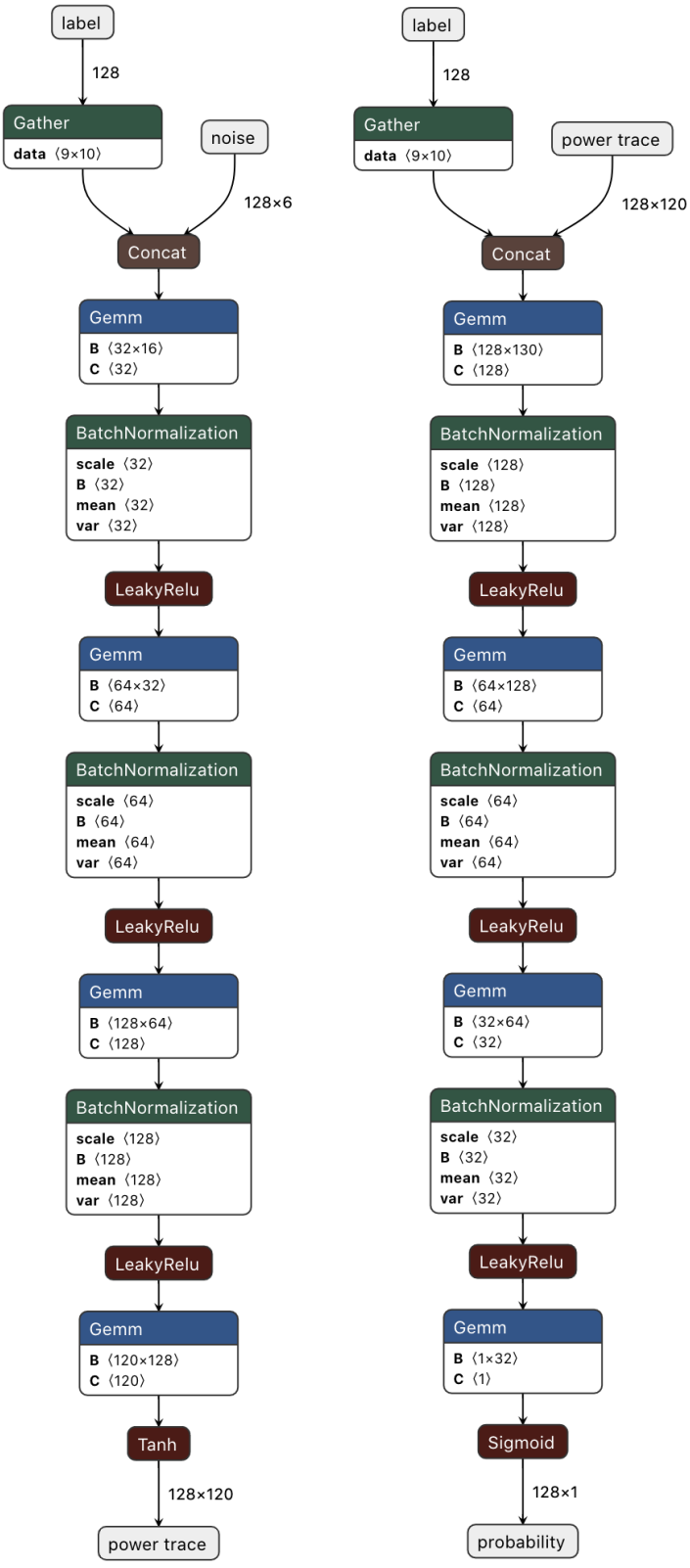
ONNX operator	Usage
Cast	Converting the datatype of a particular tensor. Used whenever there is any discrepancy in datatypes within the model.
Concat	Combining a list of tensors into one tensor. Will be used to combine the noise vector with the embedding layer output.
Expand	Broadcasting a tensor into a desired shape. Will be used only for the CCGAN architecture for accommodating the label repeat operation.
Gather	Combining the contents of a tensor for a given axis. Will be used to signify embedding layers.
Gemm	General Matrix multiplication. Will be used to signify dense layers.
ReduceSum	Calculating the sum of the tensor along a particular axis. Used in CCGAN to accommodate label inclusion in the discriminator.
Tile	Similar to numpy, used to create a tensor by repeating another tensor a number of times. Use in tandem with 'expand' operator for CCGAN.
Unsqueeze	Expanding the tensor dimension by inserting single dimension entires to the tensor's shape.

Next, ONNX models corresponding to all GAN architectures will be presented from page 88 to 99.



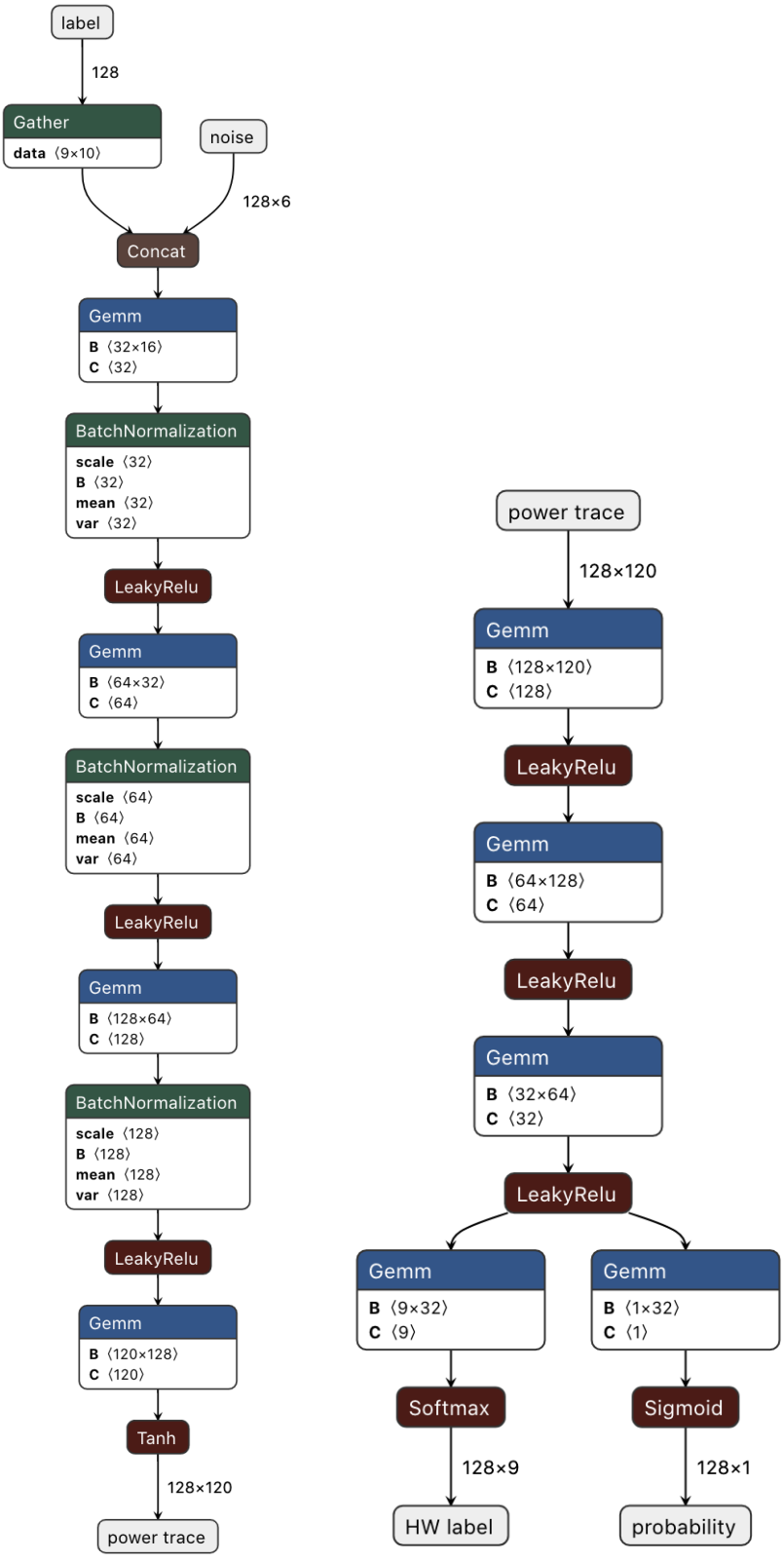
(a) Generator (b) Discriminator

Figure A.1: Multi Layer Perceptron GAN - Unsupervised



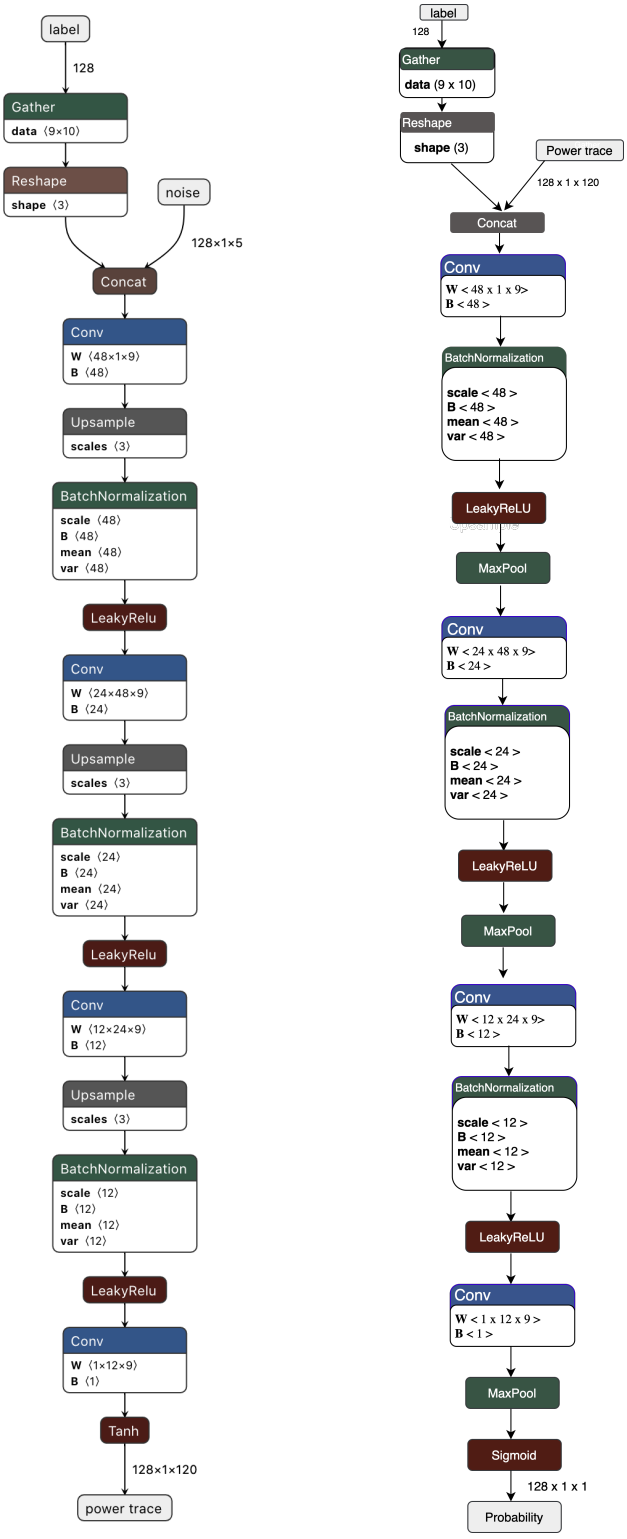
(a) MLP Generator (b) MLP Discriminator

Figure A.2: Multi Layer Perceptron GAN - HW Label



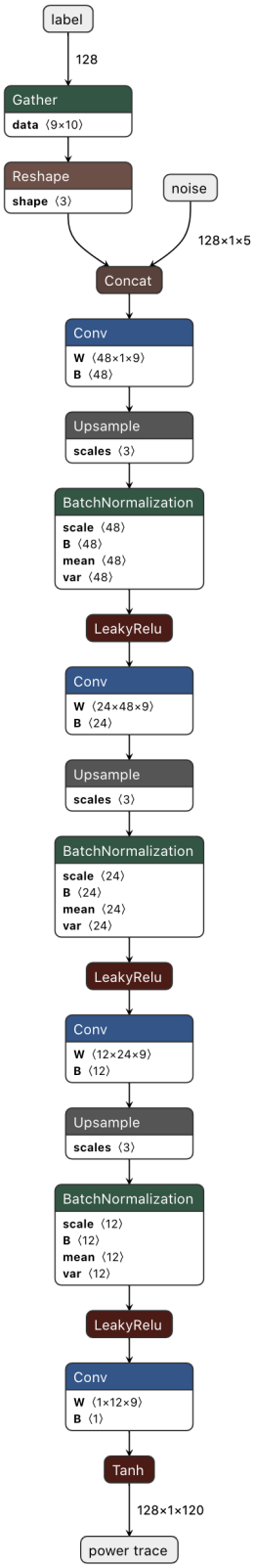
(a) AC Generator (b) AC Discriminator

Figure A.3: Auxiliary classifier GAN - HW Label

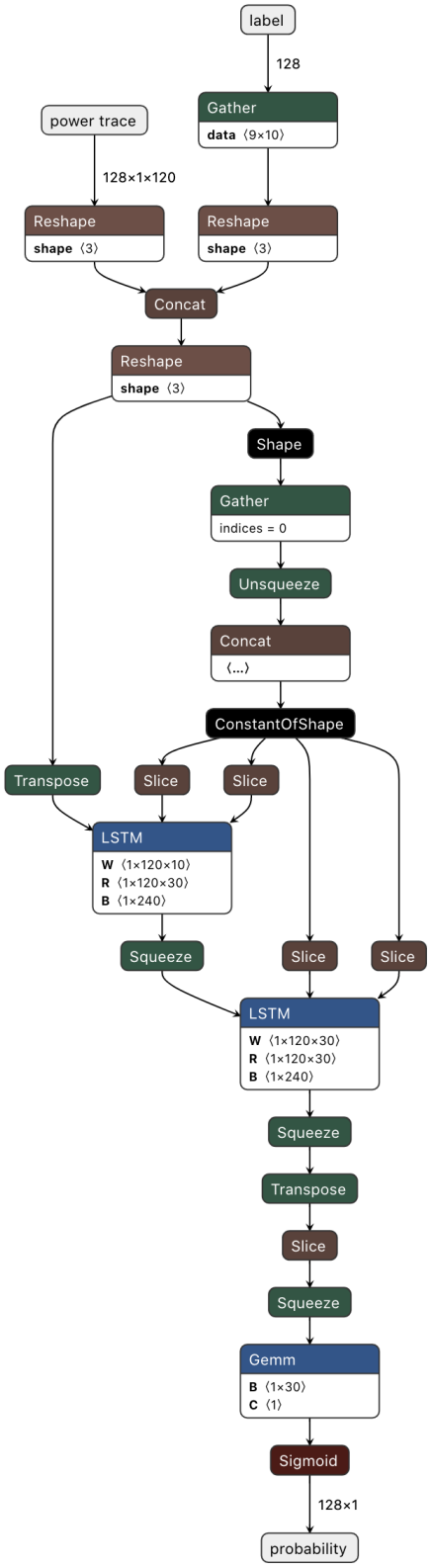


(a) DC Generator (b) DC Discriminator

Figure A.4: Deep convolutional GAN - HW Label

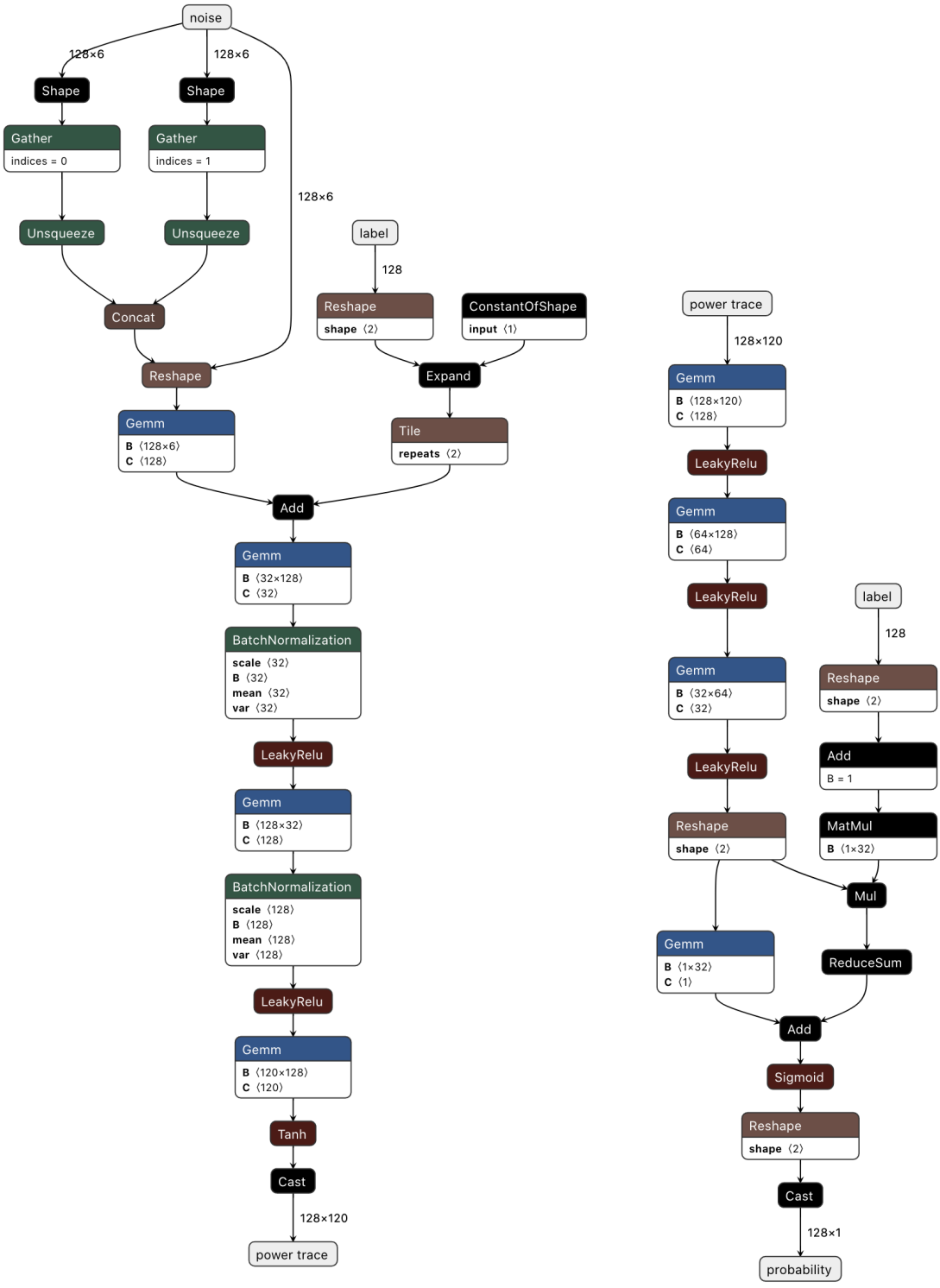


(a) DC Generator)



(b) LSTM Discriminator)

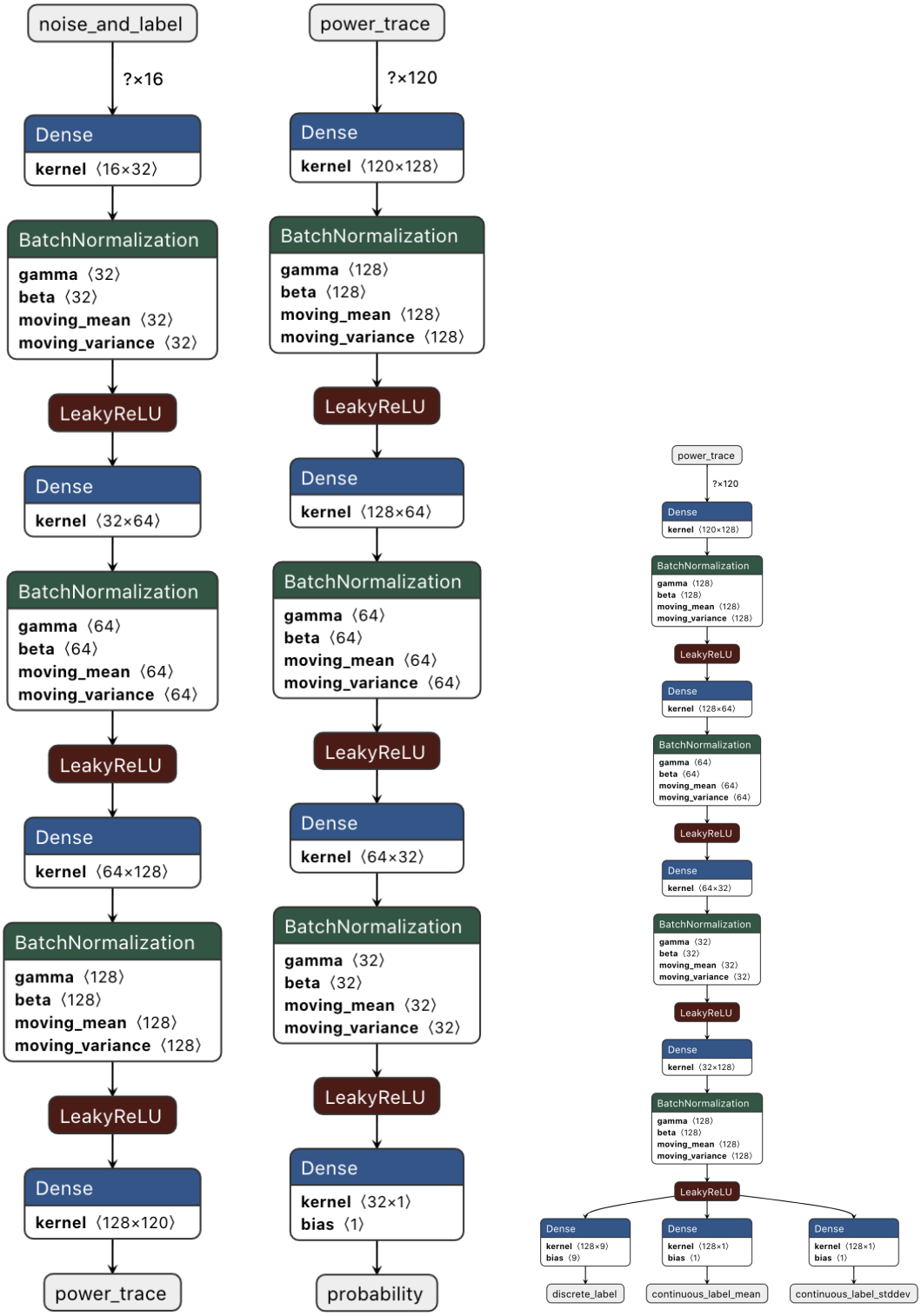
Figure A.5: LSTM GAN - HW Label



(a) CC Generator

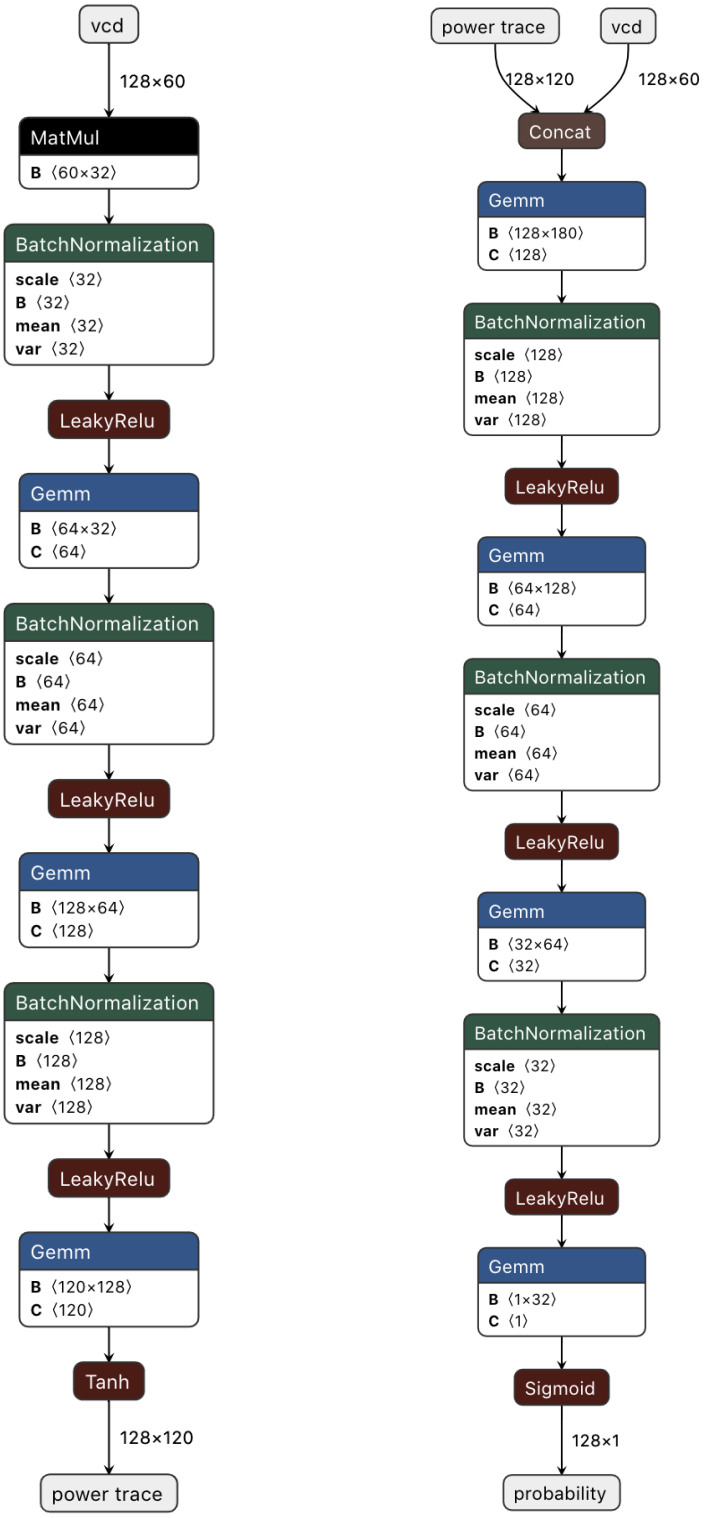
(b) CC Discriminator

Figure A.6: Continuous Conditional GAN - HW Label



(a) Generator (b) Discriminator (c) Recognition Network

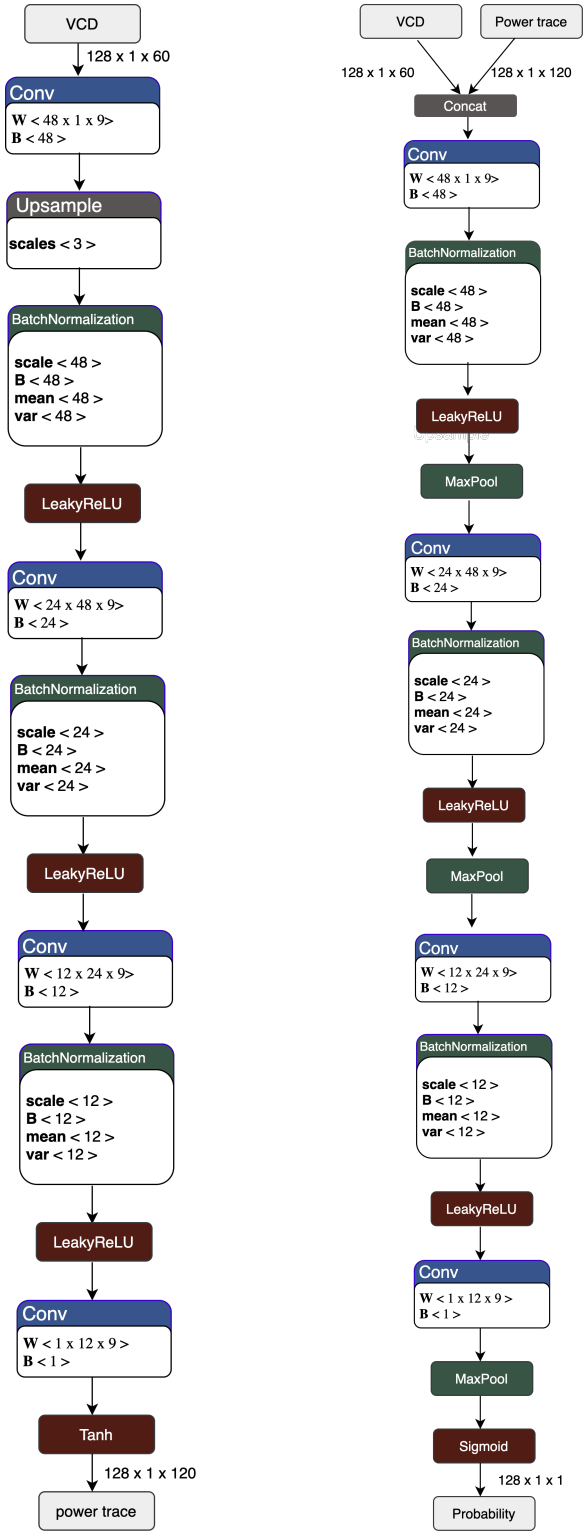
Figure A.7: INFOGAN - HW Label



(a) MLP Generator

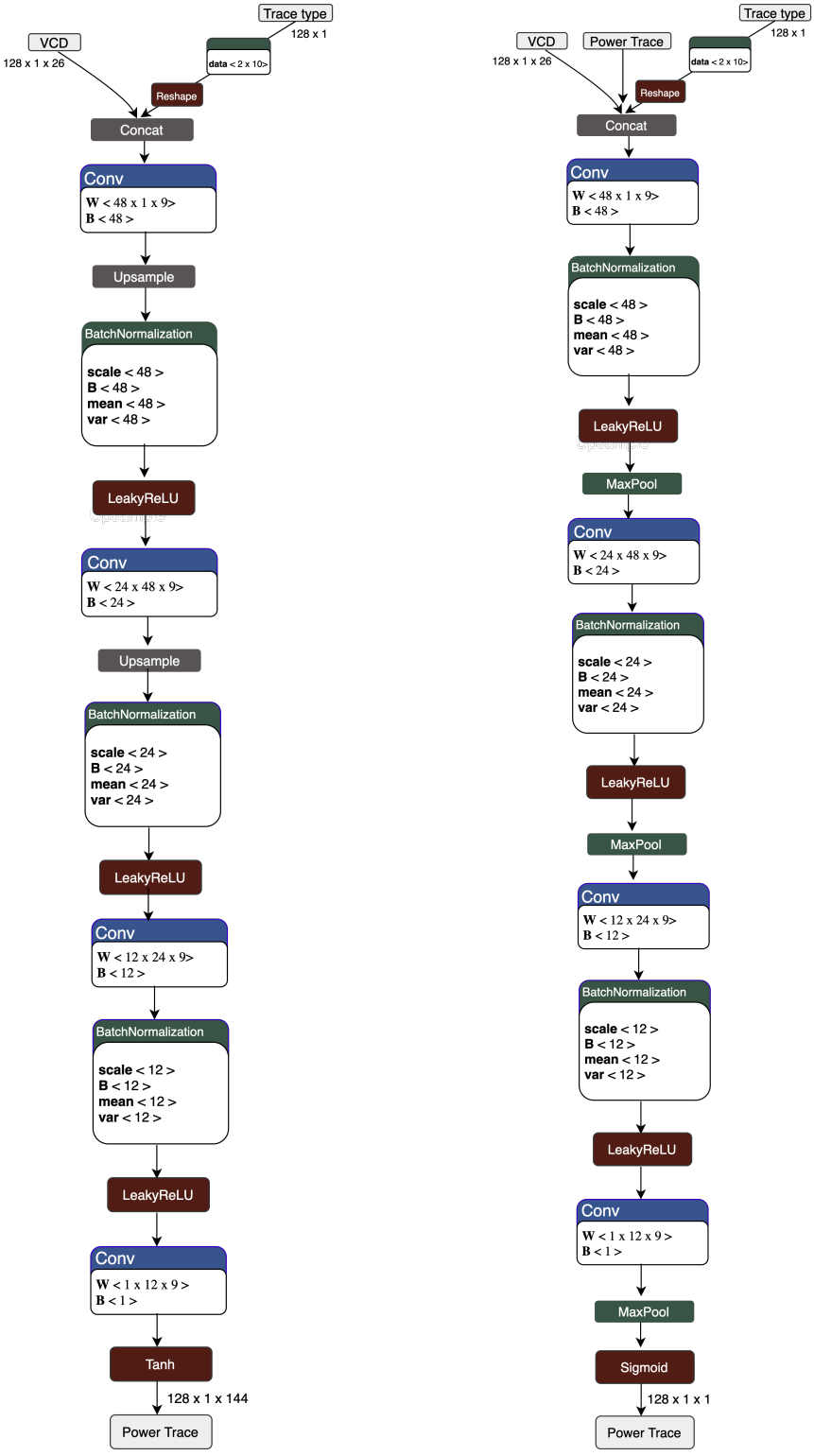
(b) MLP Discriminator

Figure A.8: MLPGAN for VCD Label



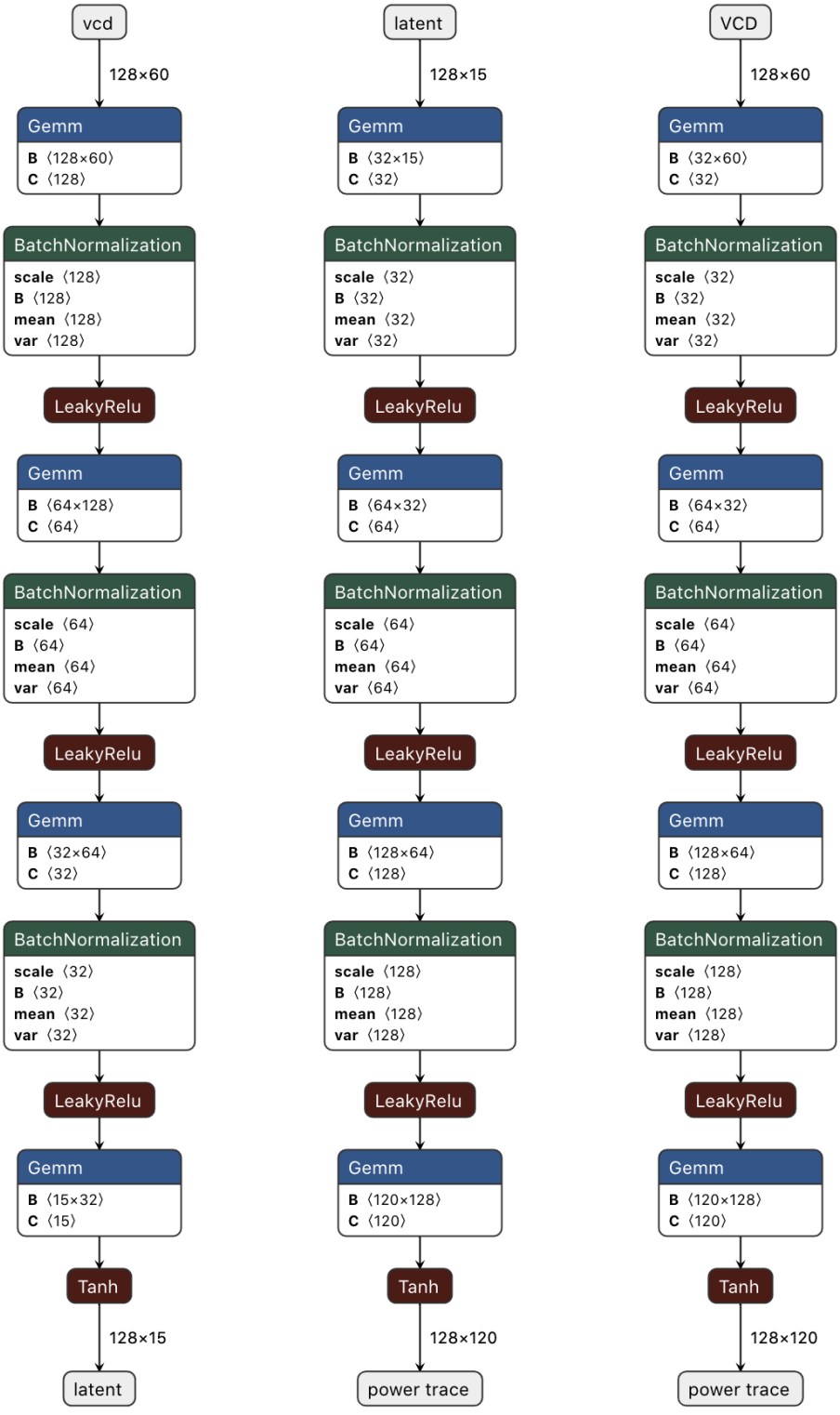
(a) DC Generator (b) DC Discriminator

Figure A.9: DCGAN for VCD Label



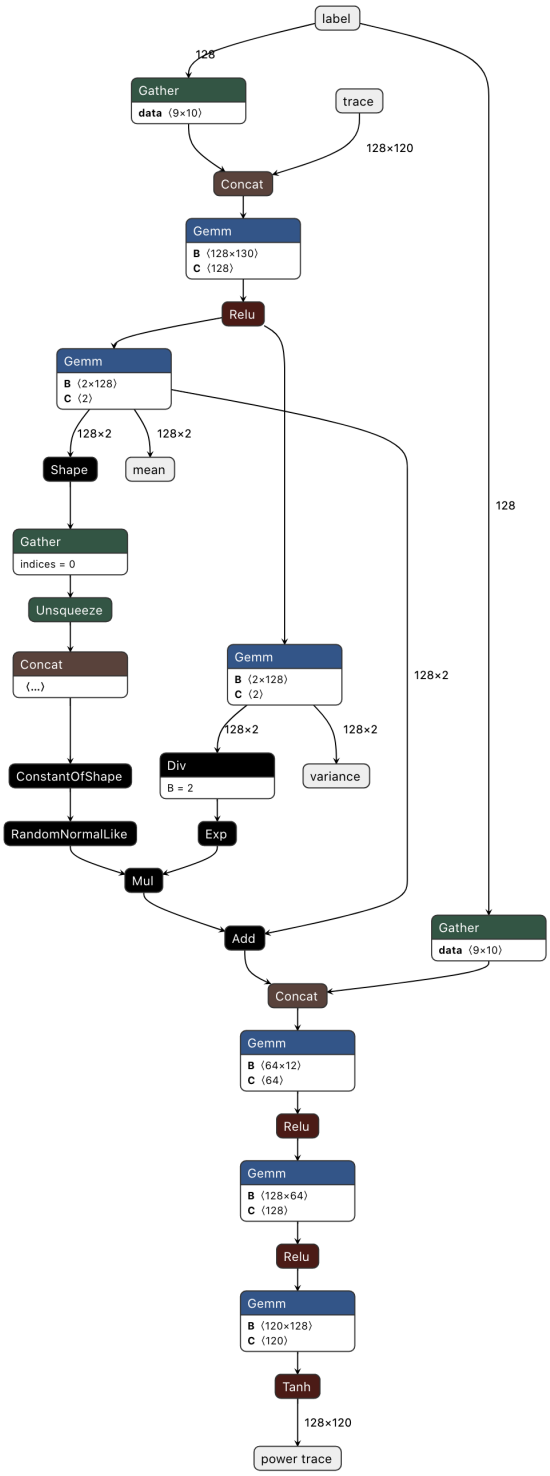
(a) DC Generator (b) DC Discriminator

Figure A.10: DCGAN for VCD Label with Trace Choice

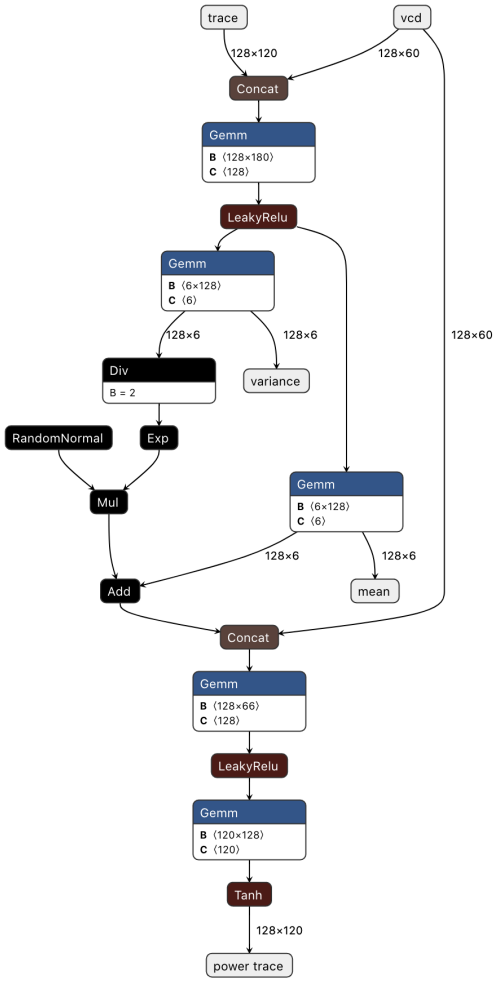


(a) MAE Encoder (b) MAE Decoder (c) Just Decoder

Figure A.11: Modified Autoencoders



(a) HW Label



(b) VCD Label

Figure A.12: Variation Autoencoder

B

End-to-End Flow

The aim of this appendix is to provide a code-first introduction to the end to end flow of power trace generation using GANs. It is easy to be overwhelmed by the numerous GANs introduced in this thesis. Hence, for the sake of clarity, an absolute basic end-to-end flow will now be discussed. The *endToend* python script can be found in the appendix section of the code repository. The entire process can be split into 5 steps as in figure B.1.

Figure B.1: Coding steps

The first step is to load the power traces, plaintexts and key. It is critical that the power traces are aligned with the plaintexts since any misalignment can have a grave impact on the working of the entire methodology.

```
f = np.load('./fixed_key_traces.npz')
traces = f['trace_array']
textin_array = f['textin_array']
known_keys = f['key_array']
assert traces.shape[0]==textin_array.shape[0]==known_keys.shape[0], "unequal number of
    traces, pt, keys"
print(f'* loaded {traces.shape[0]} traces, pt, keys!')
```

Once the training data is loaded, the next step is to calculate the labels for the GAN using the known key and the plaintexts. If the VCD transition list is used as a label, this step can be omitted. Instead the VCD transition list can be standardized/normalized and used as an input vector to the GAN (along with some random modulation for stochasticity). The SBOX look-up table is omitted in the code for conciseness. When calculating the labels, the plaintext and the known-key are first used to calculate an intermediate value, which in this case is the hamming weight of the SBOX output. Since there are 16 plaintext bytes, a DataFrame with 16 columns is made. The number of rows in this DataFrame is, of course, same as the number of number of traces in the training set. Now, every row in the DataFrame is processed to yield a single value that can be used as a label for the conditional GAN.

```
# choose a leakage model wisely
def intermediate(pt, keyguess):
    return sbox[pt ^ keyguess]

hw = [bin(x).count("1") for x in range(256)]

intermediateValue = []
for i in range(traces.shape[0]):
    temp = []
    for j in range(16):
        temp.append(hw[intermediate(textin_array[i][j],known_keys[i][j])])
    intermediateValue.append(temp)
df_hw = pd.DataFrame(intermediateValue,columns=[f'{i}' for i in range(16)])
```

```
14 print(f'* made a hw dataframe with {df_hw.shape[0]} rows and 16 columns')
15
16 #choosing mode of hw. can be changed to floor, round or ceil of mean of hw
17 labels = []
18 for i in range(traces.shape[0]):
19   labels.append(df_hw.loc[i].mode()[0])
20 labels = np.array(labels).astype(np.int64)
21 print(f'* made a label array using the mode of 16 HW values in each dataframe row.')
```

Now that the traces and labels are available, these traces and labels can be used to create a Dataloader. However, for the Dataloader to be made it is important to make a Dataset class. The Dataset class takes the traces and associate labels as input and the Dataloader splits them into random batches depending on the batch-size that is provided to it. This is done as follows:

```
1 class PowerTracesDataset(Dataset):
2   def __init__(self,traces,labels):
3     super().__init__()
4     self.traces = traces
5     self.labels = labels
6     self.n_samples = traces.shape[0]
7   def __len__(self):
8     return self.n_samples
9   def __getitem__(self,index):
10    return torch.tensor(self.traces[index],dtype=torch.float32), torch.tensor(self.
      labels[index],dtype=torch.int64)
11
12 dataloader = DataLoader(PowerTracesDataset(train_traces,train_labels),batch_size,
      shuffle=True)
```

Now that the dataset is ready, we can build a GAN model. So as to keep this example at a bare minimum, a simple MLP GAN can be made as follows. An embedding layer is used for the labels (it is like a simple look-up table for the 9 possible hamming weight values). The output of this embedding layer is concatenated to the noise input to the generator (as in line 9) as well as to the input trace to the discriminator (as in line 22). The device variable in this case is set to 'cpu'. If a cuda enable GPU is available, the variable can be set to 'cuda' to leverage the performance of the GPU.

```
1 class Generator(nn.Module):
2   def __init__(self):
3     super(Generator, self).__init__()
4     self.embedding_layer = nn.Embedding(9,10)
5     self.layers = nn.Sequential(nn.Linear(22+10,128),nn.BatchNorm1d(128),nn.LeakyReLU
      (0.1),nn.Dropout(0.3)nn.Linear(128,64),nn.BatchNorm1d(64),nn.LeakyReLU(0.1),nn.
      Dropout(0.3),nn.Linear(64,60),nn.Tanh())
6
7   def forward(self,x,labels):
8     embedVal = self.embedding_layer(labels)
9     value_to_feed = torch.cat([x,embedVal],1)
10    output = self.layers(value_to_feed)
11    return output.to(device)
12
13
14 class Discriminator(nn.Module):
15   def __init__(self):
16     super(Discriminator, self).__init__()
17     self.embedding_layer = nn.Embedding(9,10)
18     self.layers = nn.Sequential(nn.Linear(60+10,128),nn.LeakyReLU(0.1),nn.Linear
      (128,64),nn.LeakyReLU(0.1),nn.Linear(64,1),nn.Sigmoid())
19
20   def forward(self,x,labels):
21     embedVal = self.embedding_layer(labels)
22     value_to_feed = torch.cat([x,embedVal],1)
23     output = self.layers(value_to_feed)
24     return output.to(device)
25
26 netG = Generator().to(device)
27 netD = Discriminator().to(device)
```

If the VCD transition list is used, the embedding layer is not required since we feed the scaled transition list as is to the generator. For the discriminator, the scaled transition list is concatenated with the power trace and then fed through the remaining layers. After building the GAN, it can be trained using the aforementioned

Dataloader for a defined number of epochs (100 in this case). The training process is carried out alternatively for the discriminator and the generator. In both the cases, validity labels are used to signify real and generated traces. Ideally, the real traces should have a validity label of 1 and the generated (or fake) traces should have a validity label of 0. Therefore, when training the discriminator, the discriminator's output on generated traces are coupled with the validity label 0 and the discriminator's output on real traces are coupled with the validity label 1 (as on line 11/12). Since GANs play the so-called minimax game, when training the generator the discriminator's output on generated traces are coupled with the validity label 1 (as on line 17). This no-cooperative game has been theoretically explained in section 4.3.2.

```
1  loss = nn.BCELoss()
2  print('*  Training!')
3  for i in tqdm(range(100)):
4    for trace,label in dataloader:
5      batch_size = trace.shape[0]
6      noise = torch.normal(mean=0, std=1, size=(batch_size, 22)).to(device)
7      #netD
8      netD_optim.zero_grad()
9      d_r = netD(trace.to(device),label.to(device))
10     d_f = netD(netG(noise.to(device),label.to(device)),label.to(device))
11     d_total_loss = (loss(d_r,torch.ones(batch_size,1)) + loss(d_f,torch.zeros(
       batch_size,1))) / 2
12     d_total_loss.backward()
13     netD_optim.step()
14     #netG
15     netG_optim.zero_grad()
16     d_f = netD(netG(noise.to(device),label.to(device)),label.to(device))
17     g_loss = loss(d_f,torch.ones(batch_size,1))
18     g_loss.backward()
19     netG_optim.step()
```

The output of this step is a trained GAN model. At inference time, we are only concerned with the generator model. The weights and biases of the generator model can be saved as a *.pt* file. The trained generator must be set to evaluation mode before it can be used to generate power traces, as follows:

```
1  netG = netG.eval()
2  generator_traces_list = []
3  start = 0
4  end = batch_size
5  for i in range(10):
6    z = torch.normal(mean=0, std=1, size=(batch_size, 22)).to(device)
7    val_label_list = torch.from_numpy(val_labels[start:end]).to(torch.int64)
8    generatedTraces = netG(z,val_label_list)
9    for trace in generatedTraces.detach().numpy():
10     generator_traces_list.append(trace)
11   start = start + batch_size
12   end = end + batch_size
13 generator_traces_list = np.array(generator_traces_list)
```

The process of generating power traces, as explained here, clearly depends on the known key since the labels (either training/validation) can not be calculated without them. However, that is not the case with the VCD Label GAN, wherein the embedding layer is replaced by a simple Linear layer that takes the VCD transition list as an input.

C

BREVITAS AND FINN

All the GANs presented in this thesis were trained using either Pytorch or Tensorflow. Internally these libraries provide options for specifying the datatype of tensors. However, most functionalities work best with 32 bit float datatype for tensors. All MLP-GAN architectures are fast at inference time and can generate traces almost instantaneously. The same can not be said about other complex GAN architectures since for larger trace length, the time taken to generate trace increases substantially.

As a solution to this issue, the FINN compiler can be used. The FINN compiler works on an ONNX representation of neural networks. This ONNX representation is generated using quantized brevitas layers. Brevitas is a PyTorch library for quantization aware training. The FINN compiler tranforms the quantized layers into a dataflow style architecture which can be deployed on a PYNQ board. The FINN framework was originally made for inference time acceleration for binary neural networks [173]. Till date, it is presented more like a research project from Xilinx rather than a fully functional concrete framework.

Naturally, it lacks a tight integration with PyTorch and hence most models can not be ported into their corresponding dataflow architecture through it. Most notably, architectures requiring multiple inputs are still not implicitly supported by brevitas' ONNX API. Since a conditional GAN requires two inputs, one for the noise vector and one for labels, using FINN is complicated. Add to that, the scarcity of synthesizable descriptions of model layers for certain architectures makes generating IP blocks arduous.

The process of synthesizing a dataflow architecture for brevitas models is fairly straightforward, given that FINN supported layers are used. The FINN supported layers are those brevitas layers that have a corresponding FINN HLS implementation. Since multiple inputs are not supported at the time of writing this thesis, I will be providing an end to end example of accelerating the hidden layers of the generator. This idea can then be extended for other GAN architectures.

The FINN compiler involves a total of 5 steps.

1. Exporting brevitas model into ONNX : The GAN is trained using brevitas layers, just like any other pytorch model. The model is then converted into its corresponding ONNX representation. ONNX is a format built specifically for standardized representation of machine learning models.Most popular ML libraries provide APIs for converting models into the ONNX representation.

2. Preparing the network : Next, the ONNX network representation is tweaked in a way such that it is a better fit for the finn-hls layers. This preparation includes the following transformations:

   - Tidy up : Ensures unique tensor names and calculates tensor shapes after every layer output. Any unused tensors are also removed.
   - Streamlining: This step involves collapsing multiple nodes into a custom nodes that can then be converted into their corresponding HLS implementation.

3. Exporting to HLS: After preparing the ONNX network, the network is transformed into its corresponding streaming dataflow architecture using FINN-HLS layers.

4. Hardware Build : The HLS IP layers are stitched together to create a Vivado HLS project. This project is then used to generate a PYNQ driver.

5. Deployment: The deployment files generated using FINN are copied onto the PYNQ board and used for inference time acceleration.

As a proof of concept, the brevitas implementation for a MLP conditional GAN was made. The brevitas implemetation of the same has been made available in the code repository. Once FINN compiler starts supporting an extended number of layers and model architectures, most designs can be ported into their corresponding data-flow architectures and deployed on PYNQ boards for inference time acceleration.

D

NUMBA ACCELERATED CPA

Chipwhisperer [135] side channel analysis APIs are extensively used in side channel analysis research. Most attack scripts are implemented natively in python along with some cython modules. Although very intuitive and easy to use, the native chipwhisperer APIs are slow. So as to be able to do hundreds of CPA attacks for the purpose of evaluating GAN performance in this thesis, it was imperative that the CPA attack script was accelerated.

The simplest way to do so was using NUMBA [160]. The accelerated CPA script is available in the code repository. The main changes were as follows:

- Adding the *njit* decorator for all the methods used in the traditional script. Numba is then able to translate the method to optimized machine code at runtime using the LLVM compiler library [174].
- The njit decorator only works when all the numpy/pandas functions used inside of the methods have corresponding ufuncs implemented. Not all numpy/pandas functions have their corresponding ufuncs, however from the purpose of the CPA attack: *np.mean() np.where(), np.sort(), np.sqrt()* and *np.argmax()* are needed.These ufuncs are just functions that work on numpy ndarrays and are implemented in compiled C code. Functions can be converted into its corresponding ufunc using the *np.frompyfunc()* function.
- The accelerated script can be made with minor changes in the existing chipwhisperer CPA implementation, such that only the aforementioned numpy functions are used.

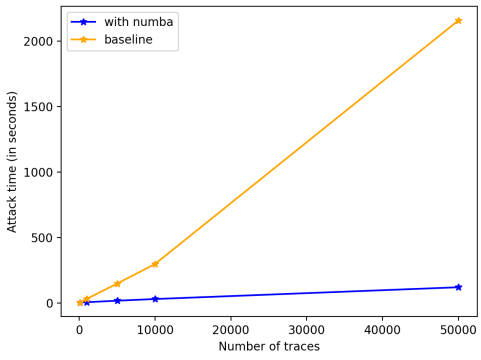


Figure D.1: Numba: comparison of CPA attack time

As can be seen in figure D.1, the Numba acceleration provides an immense speed up. Using Numba gives a better speed-up when a large number of traces are used, since numba adds an computation overhead while translating the decorated methods. This overhead is much more pronounced as compared to the baseline (unoptimized) attack time when only a few traces are used.

E

Research Paper

In this thesis, I explored the use of generative adversarial networks (GAN) for power trace generation. The objective of the GAN framework was to speed up the power trace generation as compared to traditional CAD tools. The requirement for a high speed-up was that the power traces generated using the CAD tools, although accurate can not be used for the task of security evaluation. This was because security evaluation takes place on a minimum of 10 million power traces and the CAD tools take at least a few seconds to generate every trace. Hence, the CAD tools would take years to generate the required number of traces. Using the VCD transition list label and the DCGAN architecture presented in section 5.4.2 a pre-silicon power leakage assessment framework was proposed, which provides **140x** speed-up over traditional CAD tools. The proposed framework can be used for

1. Security evaluation of designs at design time.
2. Evaluating the efficacy of countermeasures at design time.
3. Design space exploration of the countermeasures.

Since the training power traces in this work have been procured using an FPGA evaluation of a design, specifically using the chipwhisperer board [135], the framework can not factor in the technology behaviour of the design and hence can not be used to evaluate circuit level countermeasures [175] or technology level countermeasures [176]. So as to factor in the technology behaviour of the design, GANs can be trained on a few thousand power traces generated by the CAD tools. Then, the GAN framework can be used to generate the remaining amount of power traces so as to successfully carry out the security evaluation of the design. Ofcourse the speed up in this case will be less than 140x. However, there will still be an extremely large speed up since generating a few thousand training power traces will only take a couple of hours using the CAD tools. The contributions of this thesis have been condensed down into a research paper that has been attached in this appendix.

Pre-silicon Power Leakage Assessment using Generative Adversarial Networks

blind review
department
affiliation
City, Country
email address

***Abstract*—Today, security is an important feature that every system must provide. Depending on the application and environment of the target device, different threats should be considered at design time. However, the attack space is vast and hence, it is difficult to decide what components to protect, what level of protection they require and how efficient they are in the field. Only experienced designers are able to make good decisions and the level of protection the countermeasures provide can only be validated after the chip is manufactured. This paper tries to close this validation gap for power based side-channel attacks by providing a fast and reliable leakage assessment at design time that can be used to perform design space exploration for security. To accomplish our goal, we use Generative Adversarial Networks (GANs) to generate reliable power traces for hardware implementations at design time that are subsequently used to assess the leakage of the design. We tested our framework using masked and unmasked AES implementations. The generated traces were validated against real traces using the correlation power attack (CPA) and Test vector leakage assessment (TVLA). Our results show that both generated and real traces have very similar behavior in terms of leakage. Using Generative adversarial networks for the task of power trace generation yields a very high speed-up of around 140 times as compared to popular CAD tools.**

***Index Terms*—side-channel analysis, countermeasures, generative adversarial networks, symmetric cryptography, design exploration**

I. Introduction

There is no doubt that security has become a critical component of microelectronic devices. Nowadays, software protection solutions are being complemented by hardware security solutions (e.g., ARM TrustZone [1] and Intel Boot Guard [2]), which are more resilient and perform better in terms of speed and power consumption [3]. However, designing a hardware protection scheme is quite challenging as new vulnerabilities arise and, differently from software, they cannot be updated after deployment. One set of popular hardware attacks exploit the power consumption. For example, a malicious adversary can take advantage of power behavior to deduce secret information through side-channel attacks [4]. Currently, in the industry power-based vulnerabilities are only verified after the chip is manufactured, using off-the-shelf security tools and equipment (e.g., equipment of Rambus [5] and Riscure [6]). Consequently, this verification adds extra steps in the design process which increases the production cost. Even worse, when the minimal security is not met the chip has to be redesigned and manufactured, affecting not only costs but also the design time considerably. Therefore, a pre-manufacturing power leakage assessment solution is needed.

There are currently a few options to evaluate countermeasure designs prior to manufacturing. These options can be divided into three categories: formal verification [7], CAD tools [8] and functional simulation tools [9]. The aim of formal verification-based solutions is to mathematically analyze the leakage of an implementation. Formal verification examples can be found in [7, 10]. In [7] the authors use formal verification to verify hardware masking countermeasures. In [10], the authors present an SMT-solver (SMT stands for satisfiability modulo theories) for software masking countermeasures. Unfortunately, such solutions focus on analyzing randomness created by masks. As a result, they only operate on one type of countermeasure, i.e., masking countermeasures. CAD-based solutions, on the other hand, tend to produce the power behavior of the targeted implementation. An example of a CAD-based solution is provided by Sadhukhan et al. [11] where they examine the leakage using both simulated and hypothetical power traces. Another example suggested by Nahiyan et al [12] is that they reduce the number of power traces needed to evaluate the leakage by improving the signal-to-noise ratio (SNR) algorithm. Unfortunately, creating simulated power traces is a time-consuming operation, and reducing the number of simulated traces cannot validate the protection against real attacks such as CPA, which typically require a large number of traces. Additionally, we see in industry that secure IPs are evaluated with a minimal of 10 million power traces [13]. Therefore, a CAD-based assessment solution would be an interesting solution only if it can generate millions of power traces in a timely manner. Unfortunately, the CAD-based tools are known to be extremely slow and generating as many as 10 million power traces using them is not viable. For example, the solution proposed by Sadhukhan *et al.* [11] takes 5.47 seconds to generate a single trace. Generating as many as 10 million power traces using this technique will take around 633 days. This, of course, is not practical.

To obviate the need of using CAD-based tools for power trace generation, functional simulation tools like RTL-PSC [9] have been introduced. Wherein, the functional simulation of a design is used to carry out pre-silicon leakage assessment. This technique does not generate power trace as it is solely

dependent on the switching activity file. This dependence on the switching activity makes this methodology not fit for modelling the design's technological behaviour(e.g., CMOS), which is useful for various countermeasures and is not visible using simply the switching activity. The switching activity also gives a rather ideal picture about the power consumption since it does not model any non-idealities like timing violation. Hence, there is need for a tool that can not only carry out pre-silicon leakage assessment in a reasonable amount of time, but can also accommodate the device's technological behaviour and non-idealities correctly so as to model its leakage robustly.

This paper presents a novel methodology to obtain reliable power traces at design time in a much faster way than using standard CAD-tools. Our methodology uses the simulated switching activity of the target design and feeds this into a Generative Adversarial Network (GAN). The GAN analyzes the switching activity and generates power traces. The generated power traces are then validated using well-known power attacks. We use our methodology to build a framework which cannot only verify the security of a component at design time but also perform a complete design space exploration to find the most secure solution. Therefore, we present a case study, where several hardware implementations of AES algorithm are explored. In summary, the contributions of this paper are:

- Proposal of a novel methodology to generate power traces and a leakage assessment framework that can be used at design time.
- Proposal of evaluation metrics to measure the quality of the generated traces.
- Evaluation of the framework by performing security evaluation of different implementations using power-based attacks on various AES hardware implementations.
- Validation of the generated traces by comparing them to actual collected power traces.

The remainder of the paper is organized as follows. Section II describes the related work. Section III provides a background on AES, side-channel attacks, and generative adversarial networks. Section IV explains the proposed leakage assessment framework. Section V validates the proposed framework against different AES implementations. Section VI discusses the advantages and limitations of proposed design. Finally, Section VI concludes this paper.

II. Related Work

Generative deep models have already been applied to numerous applications like images [14], audio [15], video [16] and medical data like ECG [17]. In addition to the Generative Adversarial Networks, there are also other methods that could be used to generate fake traces such as *Variational autoencoders* [18], *Flow* [19] and *Auto-regressive* [20] based models. Ofcourse a number of ensemble models also exist, however for the sake of brevity we only experimented with the vanilla models. *Variational autoencoders*, are an extension of traditional autoencoders. An autoencoder can not be used as is for the generative task since all it knows is to copy the input it has been fed. Instead of mapping the input to a fixed latent space vector like in the case of an autoencoder, the input can be mapped to a normal latent space distribution (parameterized by mean and variance vectors). In such a case, the model is able to generate different samples based on tweaking the values of the latent space. Such a model is known as a *Variational autoencoder* or *VAE*. Although much more stable to train than GANs, VAEs are known for generating lower quality samples. On the other hand, *Flow* models use a sequence of invertible transformations to learn the exact data distribution. Despite their impressive ability to compute exact likelihoods, *Flow* models usually have manifold times more trainable parameters and require an order of magnitude more processing (in GPU-based platforms) for the training as compared to progressive GAN models [21]. Finally, *Auto-regressive* models decompose the likelihood into a product of conditional distributions. However, since the prediction at every time-step is dependent on all previous predictions, these models are implicitly slow. Hence, for this work we prefer GANs over other deep generative models as GANs can be trained relatively faster, have a reasonable number of trainable parameters, and have been empirically proven to produce really good quality results. From the perspective of modelling power leakage behavior, a GAN based model is ideal since not only is it known for generating good quality samples but is also known for being very quick at inference time and thus can produce power traces swiftly. The existing CAD based power trace simulation tools are extremely slow and hence employing GANs for this task can potentially lead to a large speed-up.

Generative adversarial networks have recently been introduced in the side-channel analysis domain. Wang et al. [22] proposed the usage of conditional GANs to enlarge the size of the profiling dataset for carrying out profiled side-channel attacks. They use a traditional CGAN architecture with dense layers and train it using the Jensen-Shannon Divergence approach as proposed by Mirza and Osindero [23]. In the author's own words, their study was aimed as a proof of concept and not a robust methodology. Hence their proposed methodology has a few limitations. First, they only use dense layers in their architecture. Such multi-layer perceptron GANs are best suited for shorter trace lengths since the number of trainable parameters grows substantially as the input/output size of the GAN's layers increases. Second, they condition the GAN on only a few labels, namely the least significant bit and Hamming weight of the SBOX output. Finally, they only use the Jenson-Shannon Divergence loss and do not experiment with other loss functions. Since the inception of GANs, many architectural changes and loss functions have been proposed that help in alleviating various problems associated with GANs as well as enhancing the training stability.

III. Background

This section provides the background needed for this paper. It first describes Advanced Encryption Standard. Thereafter, it presents the masking and hiding countermeasures. Finally, it discusses generative adversarial networks.

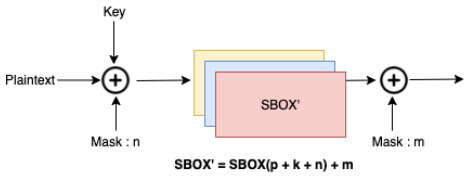


Fig. 1: Random Masking

A. Advanced Encryption Standard (AES)

AES, also known as Rijndael Cipher, has become the current standard for symmetric cryptographic algorithms. The AES algorithm consists of four main functions: *AddRoundKey*, *SubByte*, *MixCoulmns*, and *ShiftRows*, and it supports three different keys length variants 128, 196, and 256 bits. For more details regarding AES we refer the reader to [24]. Among all AES functions, the *SubByte* leaks the most information, and hence is often targeted by side-channel attackers. The *SubByte* function (or so-called SBOX) is an essential nonlinear substitution function that attempts to obfuscate the correlation between the key and the plaintext/ciphertext. The *SubByte* function works as follows: the multiplicative inverse of an 8-bit input representing a polynomial is calculated using the finite Galois Field GF(2^8) and the irreducible polynomial $p(x) = x^8 + x^4 + x^3 + x + 1$ followed by an affine transformation. To optimize the *SubByte* in terms of performance, a pre-calculated look-up table (LUT) containing all the possible values (256 values) of the 8-bit input is used, can be used which replaces the computation of the multiplicative inverse and affine transformation. In addition to the two naive implementations already described (i.e., Non-LUT and LUT-based), the SubByte can be also implemented in a variety of different ways mainly to prevent information leakage [25].

B. Masking Countermeasure

For the past two decades, several countermeasures have been proposed to thwart power attacks. These countermeasures vary based on their implementation (i.e., software, hardware design, circuit implementation) and technique (i.e., obfuscating or balancing the power consumption). One of the famous examples of obfuscating the power consumption is masking. Masking can be implemented in software [26], by modifying the hardware design [27] or using masked logic cells (i.e., masked dual-rail pre-charged logic) at circuit level [28].

In this paper, a simple form of masking implementation is used in addition to the naive implementations. The countermeasure, as shown in Figure 1, works as follows: by selecting two random 8-bit values (mask m and mask n) the SubByte function can be calculated using Equation1. In Equation 1, P denotes a byte of the plaintext, while K represents a byte of the secret key. To ensure that the encryption output is correct, the SBOX also needs to be modified [29] (the modified SBOX is depicted as SBOX' in equation 1). Note that the SBOX performs a non-linear transformation, hence for given plaintext P and mask n the sbox output $SBOX(P \oplus n)$ is not equal

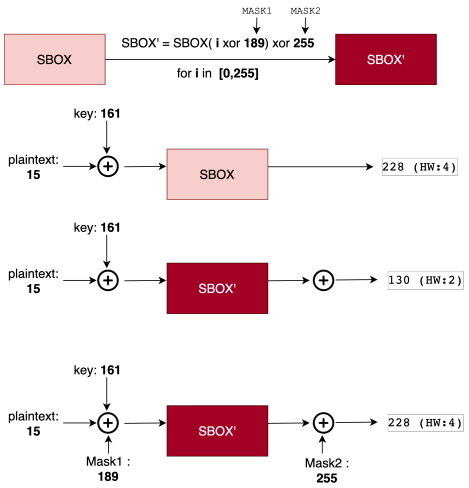


Fig. 2: Impact of masking on SBOX

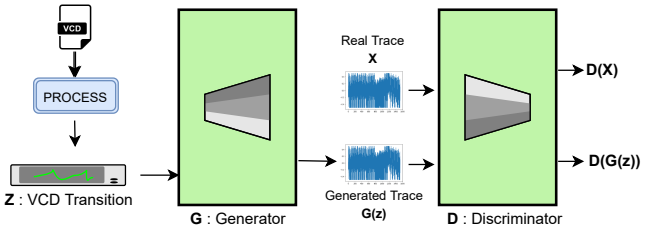


Fig. 3: Generative Adversarial Network

to $SBOX(P) \oplus SBOX(n)$. This countermeasure aims to reduce the leakage by invalidating the Hamming Weight and Hamming Distance leakage models.

$$SBOX[P \oplus K] = SBOX'[P \oplus K \oplus n] \oplus m \qquad (1)$$

A simple example of the same can be seen in Figure 2. In the example, mask pair (189,255) is used. First, the *SBOX* is modified using the mask pair so as to become *SBOX'*. The impact of that is that the same key and plaintext byte results in two different values, which are 228 and 130 for *SBOX* and *SBOX'* respectively. So as to correct the output of *SBOX'*, mask1 and mask2 are needed. Mask1 will make sure that the correct index is accessed in the SBOX look-up table and mask2 will undo the effect of masking on the value contained in the look-up table.

C. Generative Adversarial Networks (GANs)

GANs are used to generate new data with similar characteristics as the training set. GANs try to approximate the dataset's distribution and consist of two main components, i.e., the generator G and discriminator D. The generator's objective is to generate samples with a similar distribution to the actual dataset distribution *p(x)*. The discriminator's objective is to differentiate between the real and fake traces, namely x and *G(z)*. Hence, the training process of a GAN takes place in an adversarial setting wherein the discriminator and the generator

play a minimax game. The Loss function L can be expressed as follows:

$$\min_G \max_D L(G,D) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p(z)}[\log(1-D(G(z)))] \quad (2)$$

This loss function corresponds to the original GAN architecture as proposed by Goodfellow et al. [30]. However, since the training is completely unsupervised, there is no control over what the GAN generates at inference time. To have control over what the GAN generates, it should be conditioned on a categorical label corresponding to each encryption. The label y (which for example can correspond to the hamming weight/distance of the SBOX output) must also be a part of the loss function. Such conditional GAN loss function was originally proposed by Mirza and Osindero [23] and can be expressed as:

$$\min_G \max_D L(D,G) = \mathbb{E}_{x\sim \mathrm{p_{data}}(x)}[\log D(x \mid y)] + \mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z \mid y)))] \quad (3)$$

The non-cooperative game is what makes training GANs hard and unstable. To address this problem, a lot of research has been performed to find better loss functions and normalization techniques.

In this work, in addition to the loss function presented in Equation 3, we experimented with the Least Squares GAN (LSGAN) [31], the Wasserstein GAN (WGAN) loss and the Wasserstein GAN with Gradient penalty (WGAN-GP) loss function [32]. The LSGAN aims at increasing the quality of the generated samples by improving the discriminator's output by not only looking at the binary output decision but also quality of the generated traces. Note that in the GAN proposed by Mirza and Osindero, the discriminator's purpose is to distinguish between the real and fake samples as shown in Equation 3, which is realized using binary cross entropy loss. In LSGAN, however, in addition to binary classification, the concern is also how close or how far the fake traces are from the real ones. This can be seen in the loss function presented in Equation 4.

$$\begin{aligned} \min_D L_{\mathrm{LSGAN}}(D) &= \frac{1}{2}\mathrm{E}_{x\sim p_{\mathrm{data}}(x)}\left[(D(x)-b)^2\right] \\ &\quad + \frac{1}{2}\mathbb{E}_{\boldsymbol{z}\sim p_x(z)}\left[(D(G(z))-a)^2\right] \\ \min_G L_{\mathrm{LSGAN}}(G) &= \frac{1}{2}\mathrm{E}_{z\sim p_x(z)}\left[(D(G(z))-c)^2\right] \end{aligned} \quad (4)$$

During the training of the discriminator, *b* is set to 1 to signify real traces and *a* is set to 0 to signify generated traces. As the objective of the Generator is to create fake traces that look real, while training the generator c is set to 1.

An issue with JS Divergence is that when the distributions are disjoint, the loss value becomes a constant causing the gradients to vanish. To address this issue Arjovsky *et al.* [33] used the Wasserstein metric (also known as the earth mover's distance) as a loss function for GANs. Minimizing the earth mover's distance leads to smoother gradients even when the generator outputs unsatisfactory traces. The WGAN's Discriminator tries to model a function that approximates the EM distance and not just distinguish the real samples from the generated ones. Wasserstein metric, is a concept borrowed from transport theory and can be interpreted as the minimum amount of work required to transform one probability distribution to another. However, computing the earth mover's distance is an intractable problem and the Kantorovich-Rubinstein duality [34] is used to make the problem simple by transforming the EM distance minimization problem in order to find a least upper bound. The transformed loss function is required to satisfy K-Lipschitz continuity. The Lipschitz continuity limits how fast a function can change. In its original implementation, the authors perform weight/bias clipping in a range [-c, c] so as to enforce a Lipschitz contraint on the critic. The loss function for WGAN is as follows:

$$\mathrm{L}(p_r,p_g) = \max_{w\in W} \mathbb{E}_{x\sim p_{\mathrm{data}}x}\left[f_w(x)\right] - \mathbb{E}_{z\sim p_z(z)}\left[f_w\left(g_\theta(z)\right)\right] \quad (5)$$

The clipping method was criticized by the authors of the paper themselves and hence the gradient penalty version gained popularity. The gradient penalty version, known as WGAN-GP also aims at minimizing the earth mover (EM) distance. However, the Lipschitz constraint is enforced not enforced by weight clipping.

$$\mathrm{L}= \underbrace{\mathop{\mathbb{E}}_{\tilde{\boldsymbol{x}}\sim\mathbb{P}_g}[D(\tilde{\boldsymbol{x}})] - \mathop{\mathbb{E}}_{\boldsymbol{x}\sim\mathbb{P}_r}[D(\boldsymbol{x})]}_{\text{Original critic loss}} + \underbrace{\lambda \mathop{\mathbb{E}}_{\hat{\boldsymbol{x}}\sim\mathbb{P}_{\hat{\boldsymbol{x}}}}\left[\left(\|\nabla_{\hat{\boldsymbol{x}}}D(\hat{\boldsymbol{x}})\|_2 - 1\right)^2\right]}_{\text{gradient penalty}} \quad (6)$$

The weight clipping method is extremely sensitive to the clipping value hyperparameter and quite often reduces the network's ability to model complex functions. Instead, WGAN-GP adds a gradient penalty term to enforce the K-Lipschitz continuity as shown in Equation 6.

IV. Proposed Framework

This section presents our proposed framework. We first introduce the methodology at high level and then each step is explained in detail.

A. Methodology

Our proposed framework can be used to evaluate the efficacy and efficiency of countermeasures against side-channel attacks without the need of procuring actual power traces for an ASIC design. The methodology to create this framework consist of five major steps as shown in Figure 4. In the first step, *Generate Inputs*, the targeted circuit (e.g., AES implementation) is simulated and used to generate the required inputs. Next, in the *Process training data* step the VCD are transformed into labels for the GAN and real power traces are collected by evaluating the design on an FPGA. In the third step, called *Build GAN Model*, the GAN model is constructed and trained. Thereafter, in the *Validate GAN model* step, the GAN model is evaluated under different inputs for the same design and the quality of the generated traces is analyzed.

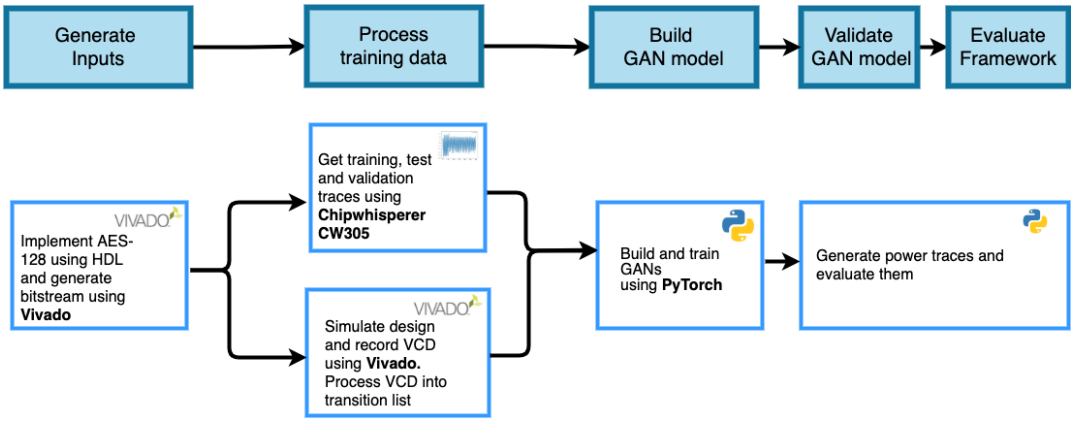


Fig. 4: Framework Methodology

Lastly, in the *Evaluate Framework* step the GAN model is evaluated for different inputs potentially for different hardware designs to check if it can generate reliable data independently of the input configuration (e.g., different plaintext or different key during an AES encryption). In the following subsections, we describe each step in more detail.

B. Generate Inputs

To successfully generate reliable power traces (i.e, traces that are similar to the actual power traces), we use the switching activity of a target circuit as the GAN's input. One of the most common representation of the switching activity is the value change dump (VCD) file. VCD files can be generated during RTL or netlist simulations. In this step AES encryptions are simulated with randomly selected plaintext and a fixed key. The result is a large VCD file. Additionally, to train the GAN we also need to collect real power traces with inputs matching to the simulated ones. This is accomplished by emulating the design in a FPGA and collecting power traces. Note however that the VCD file cannot be used directly to train the GAN, which bring us to the second step (i.e., Process training data).

C. Process training data

Our framework includes a VCD translation engine, which contains a set of scripts that process the VCD file, organize its content in a standard format (i.e., pre-defined timestamps, size of switching array, etc.), and create labels from them. For each encryption, all wire / register transitions are recorded in the VCD file. Therefore, the VCD can be used to extract a signal transition list for each encryption. The VCD translation engine ensures that the switching information is correctly processed independent of the target design. Extremely important for this step is that the label conveys a holistic picture about the power trace which is solely provided by the VCD transitions. In theory, as VCDs are application independent, the generative model has the ability to produce accurate power traces for a wide range of applications at inference time. Through our initial experiments, we observed that the usage of non-informative labels (such as Hamming Weight and Hamming Distance of the SBOX output) resulted in generated traces that were visually similar to the actual traces but did not convey the leakage characteristics correctly. A visual representation of the VCD translation engine can be seen in Figure 5.

D. Build GAN model

In this step, we build the GAN model. Unfortunately, using the switching activity as labels prevent us from using embedding layers in the GAN's architecture, as the embedding layers expect integer numbers as input. The reason for this is that they are implemented as simple look-up tables. In our framework, we remove the embedding layer as well as the noise vector and instead just use the VCD transition as an input to the GAN. A similar architectural choice was also made by Kumar et al. in their MelGAN architecture [35], wherein they observe little perceptual difference in the generated waveforms when additional noise is fed to the generator. Even Mathieu et al. [36] and Isola et al. [37] demonstrate the noise vector's redundancy when using highly informative conditioning.

E. Validate GAN model

In this step, the GAN model is tested on a similar VCD file that it was trained on, which means that the key and the implementation of the AES algorithm are still the same and only the plaintexts are modified. The aim here is to observe if the generated traces are able to portray the information from the VCD transition labels correctly.

F. Evaluate Framework

The evaluation of the framework is performed through a generalization test. Hence, we evaluate if our model can provide reliable power traces independently of the input configuration of the target circuit. In this step, not only the plaintext varies, but also the key value. In each scenario, the quality of the generated power traces is verified through two leakage assessment techniques, evaluation-test (i.e., Correlation Power Attack) and conformance-test (i.e., Test Vector Leakage Assessment (TVLA) and Signal to Noise Ratio (SNR)). Finally, both collected and generated power traces are compared. In

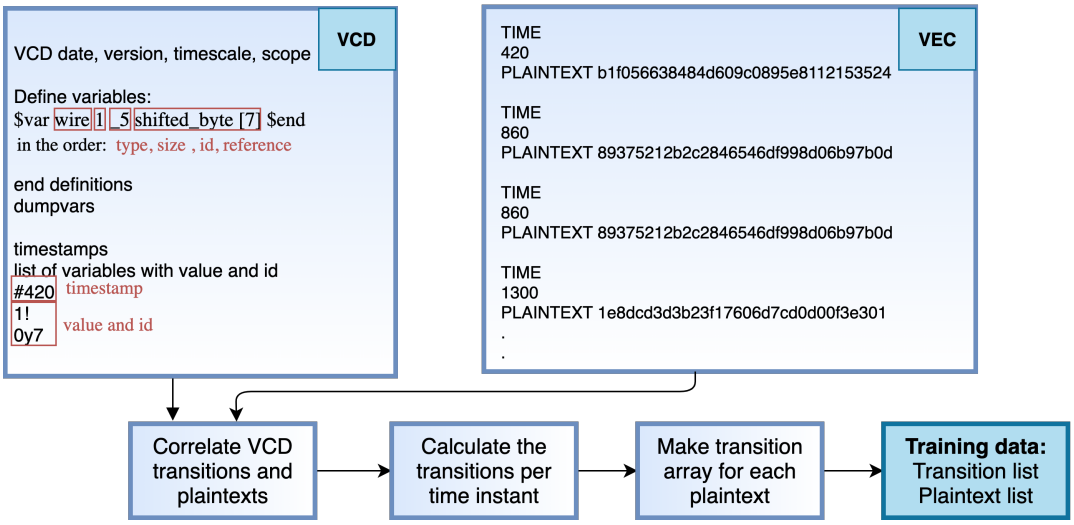


Fig. 5: VCD translation engine

addition, for various different implementations traces can also be generated and evaluated . In this context, the framework can be used to perform a design space exploration to find the most secure solution for a certain algorithm by quickly generating and evaluating traces for VCD files belonging to different design. We limit our tests to three implementations only, which are: unprotected AES implementation, AES implementation with masking and AES implementation with blinding.

When the framework is completed, designers can generate as many traces as they need. For example, according to security standards, security components (i.e., Intellectual Property blocks) used in the industry require for their vulnerability assessment 10 million, 100 million or 1 billion power traces [13].

V. Experimental Results

This section presents the experiments setup, performed experiments and evaluates the obtained results.

A. Setup

This subsection describe the set of tools and equipment used in our five-step methodology (see Subsection IV).

Generate Inputs and Process training data In the first two steps, generated VCD files and real power traces of the target design are collected. Thereafter, we simulate the design using Questasim simulator [38], which is used to generate the required VCD files. We emulate the AES in an FPGA and collect actual power traces from it. To realize this we use Chipwhisperer CW305 equipment [39], which contains a Xilinx FPGA and an power ADC to sample power traces.

Build GAN model: In the third step, a GAN model is built and trained. Our target computer was a MacBook Pro with 8259U coffee lake processor and 8GB of LPDDR3 SDRAM. Athough most popular GAN models are extremely compute heavy and make excessive use of GPUs, we wanted to limit the computation resources since we believe that most end-users of this framework will also not have GPU clusters at their disposal. All computations were performed using the python programming language, where the GAN model was made using the PyTorch [40] deep learning library.

Validate GAN model: In this step the generated traces of the GAN model are compared and validated against real power traces. We evaluate the similarity between them by performing a popular evaluation-style leakage assessment. The used metric is detailed later in subsection V-B

Evaluate Framework: To verify the accuracy of the framework we test our findings using various implementations with different keys to ensure that our approach works regardless of the input or target design. There are two unprotected LUT-based SBOXes, one for training and the other for validation of the GAN model; each use different keys and plaintext values. In addition, the GAN is used to generate traces based on a VCD belonging to a protected masked SBOX implementation as well as a protected blinding implementation which are validated against corresponding measured traces. Note that the GAN is only trained using a non-protected implementation.

B. Evaluation Metrics and Performed Experiments

In this subsection we first present the three metrics used to evaluate our results, namely evaluation-style, conformance-style and signal-processing based metrics. Thereafter we describe to which experiments these metrics have been applied.

Evaluation-style Metric: In evaluation-style testing, power traces are tested using actual side-channel attack scenarios. They show whether the implementations are resistant to these attacks or not. The attacks can be performed in a profiled or unprofiled manner. Examples of profiled side-channel attacks are template-based [41] and deep learning attacks [42]. Differential power analysis (DPA) [43] and correlation power analysis are two examples of unprofiled side-channel attacks (CPA) [44]. In this paper we limit our analysis to CPA as it is one of the most popular unprofiled techniques.

The correlation side-channel attacks work as follows for AES: given a number of recorded traces N of an AES implementation, the used key can be retrieved using the Pearson coefficient correlation which is calculated in Equation 7.

$$r_{i,j} = \frac{\sum^{N}(h_{k,i} - \mu_{h_i})\dot{(}t_{k,j} - \mu_{t_j})}{\sqrt{\sum^{N}(h_{k,i} - \mu_{h_i})^2 \dot{\sum}^{k}(t_{k,j} - \mu_{t_j})^2}} \tag{7}$$

Note that the key is retrieved one byte (i.e., subkey) at time. In Equation7, $h_{k,i}$ represents the hamming weight of the i^{th} intermediate operation (e.g., Hamming Weight of SBOX output of the first round), k the hypothetical assumed subkey of the encryption/decryption execution and $t_{k,j}$ the sample point j within the sub-trace k. The subkey with highest correlation is most likely the correct key value.

Conformance-style Metric: On the other hand, conformance-style testing examines traces for compliance with specific leakage criteria without taking actual attacks into account. Test Vector Leakage Assessment (TVLA) [45] and signal-to-noise ratio (SNR) analysis [46] are two examples of this form of analysis.

As the purpose of this study is to show the visibility of applying this style of testing and not to validate developed countermeasures, we decided to select one of the two methods without any preference, i.e., TVLA analysis. TVLA is based on Welch's t-test, which examines whether two populations have similar means. Welch's t-test is used in the side-channel domain to see whether the power traces of an encryption/decryption algorithm execution leak information about the secret key. The leakage is measured using two sets of power traces, one with fixed plaintext/ciphertext and the other with random plaintext/ciphertext. Note that the key value is the same in both sets. Equation 8 shows the equation used to perform this test.

$$t = \frac{\bar{X_1} - \bar{X_2}}{\sqrt{\frac{S_1^2}{N_1} + \frac{S_2^2}{N_2}}} \tag{8}$$

In the equation, $\bar{X_1}$, S_1^2, and N_1 represents the means, the variance, the total number of used *fixed plaintext/ciphertext* traces, respectively, while $\bar{X_2}$, S_2^2, and N_2 represents the means, the variance, the total number of used *random plaintext/ciphertext* traces, respectively.

Signal processing metrics: So as to compare the similarity between the real and the generated traces, certain signal processing metrics like *dynamic time warping* and *power spectral density* can be used. *Dynamic time warping (DTW)* [47] finds an optimal alignment between two unmatched temporal sequences. This optimal alignment or the 'warping path' maps the two sequences such that the distance between them is minimized. The minimum distance can be used as a measure for similarity between any such two sequences. Similarly, even the *Power spectral density (PSD)* of two signals can be used as a measure for similarity between them. *Power spectral density (PSD)* is the measure of power distributed across different frequency components that compose a signal [48]. For the real and the generated traces to be visually similar, the *DTW* distance should be low (the lowest it can be is zero) and the *PSD* should be very similar.

Performed Experiments: In this work, we performed two different experiments:

1) GAN Model Validation: This experiment shows the results for the fourth step of our methodology. It evaluates the GAN under different plaintext using evaluation-style and signal processing metrics.
2) Framework Evaluation: The last experiment investigates the generalization of our resulted GAN model, by applying different plaintext, keys and even using different AES implementations. The evaluation use all evaluation-style, conformance-style and signal processing testing.

Note that both the experiments were performed using the GAN architecture shown in Figure 6. Hyperparameter tuning for GANs is much more complex than hyperparameter tuning for other machine learning models since the two-model architecture of GANs does not easily fit into the popular hyperparameter search APIs. Our initial architecture was inspired by a popular GAN implementation [49] and then we randomly searched for optimal hyperparameters. For this work: the batch-size is 128, the VCD length is 60 samples, the power-trace length is 120 samples and the kernel size for the convolutional layers is 9. The idea of using a slightly larger kernel size was inspired from [50], where the author demonstrates that deep learning based power side-channel attacks using a convolutional neural network (CNN) with larger kernel sizes perform much better than CNNs with small kernel sizes. As for the loss function, the visual appearance of the traces as well as the leakage behaviour has minor variations for different loss functions as in Figure 7. The sub optimal performance on the WGAN loss function might be attributed to lack of tuning of the clipping hyperparameter. For the most part, the minor impact of loss functions on the generated traces is inline with [51] where the authors state that the quality of the GAN generated samples are not substantially dependent on the loss functions.

C. GAN Model Validation

To validate the trained GANs, we use both evaluation-style and signal processing metrics.

Evaluation-Style: Let' now analyze the GAN generated traces in detail. The generated power traces are extremely similar to the original power traces as can be seen in Figure 7. To ascertain that the generated power traces are also as informative as the original power traces, their attackability can be analyzed. A similar ratio for attackable and non-attackable bytes for real and generated traces can prove that the generated traces can be used as a substitute for the actual traces for evaluating the efficacy of a countermeasure. In this step we compare the attackability on the same AES implementation but on different plaintexts.

Table I compares the CPA attack ranking analysis of generated traces and actual power traces. From the table it can be clearly seen that the key ranks for the GAN generated traces are exactly the same as the real traces, except for byte 8 and

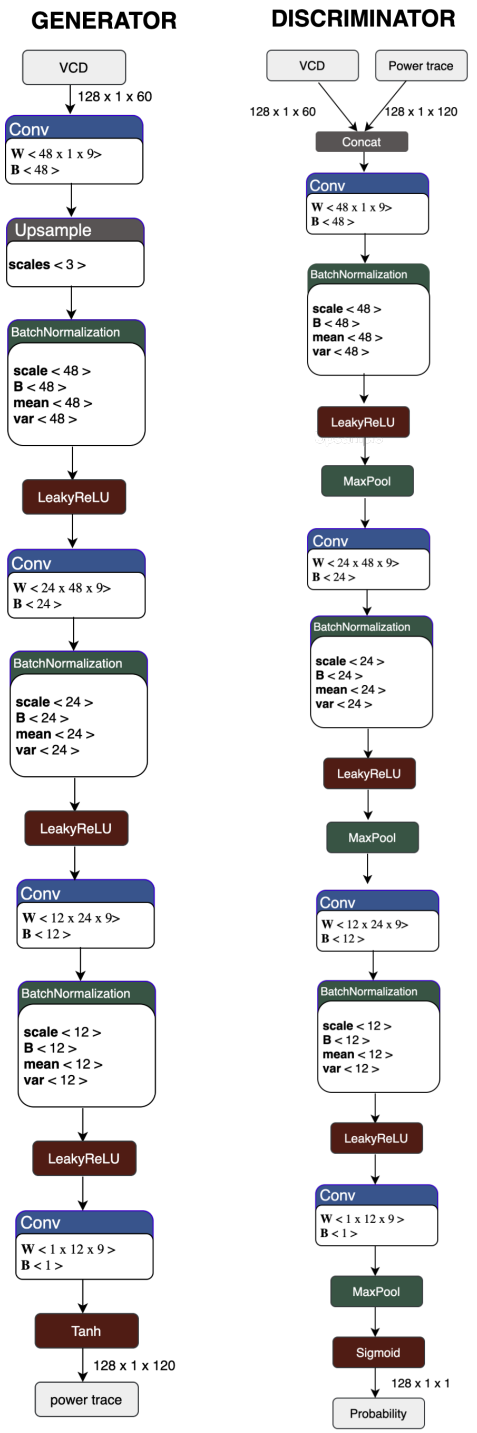


Fig. 6: GAN architecture

TABLE I: Comparison - CPA Key Rank for 1000 Traces

BYTE	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Real	1	1	1	1	1	1	1	9	1	1	1	1	1	2	1	1
GAN	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	11

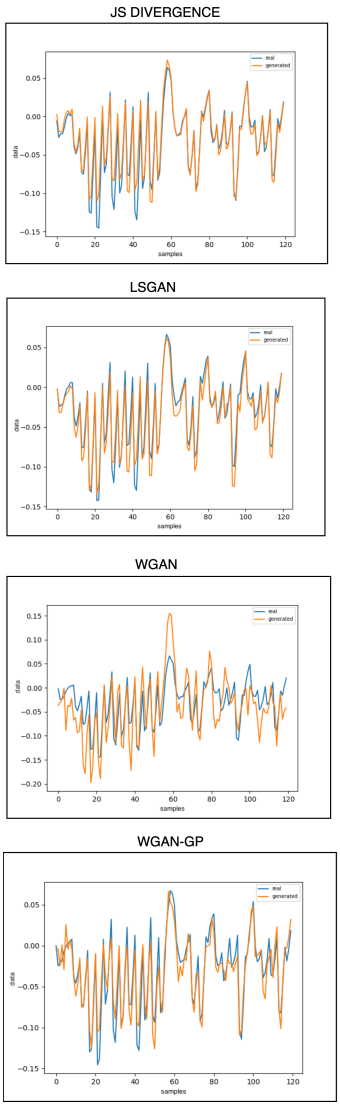


Fig. 7: Impact of Loss functions

TABLE II: GAN Framework Evaluation - CPA Rank

Traces	Att.	Non-att.	Resource utilization
Generated: Masking	0	16	Uses 2511 LUTs and 3364 FFs
Real: Masking	0	16	
Generated: Different pt/key	14	2	Uses 2430 LUTs and 3363 FFs
Real: Different pt/key	13	3	

16. Note that for byte 8 the GAN performed better than the actual traces. This means that this approach can be used as well to advance side-channel attacks. An attack would only be able to apply such an attack if the hardware implementation is publicly available as VCD's are required. On the other hand, for byte 16 the GAN performs this. We believe this to be a statistical discrepancy.

Signal processing metrics: The mean DTW distance between

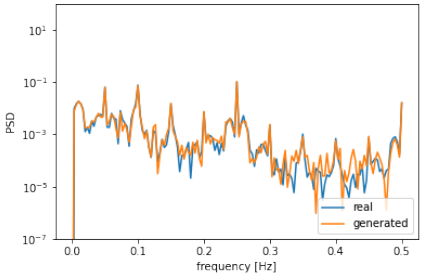


Fig. 8: Comparison of Power Spectral Density

TABLE III: Number of failing t-test points

	Unprotected	Masked	Ratio
Generated traces	22	19	1.15
Real traces	15	13	1.15

the real and the generated traces was **1.02**. This distance was calculated using FastDTW [52] which is an approximate DTW calculation algorithm and we used Euclidean distance as the distance measure for DTW. In Figure 8 Power spectral density (PSD) for the real and generated traces can be seen. Note that both the PSD plots are almost indistinguishable.

D. Framework Evaluation

The framework is evaluated using all evaluation-style, conformance-style and signal processing metrics.

Evaluation-Style: As described in the Evaluate Framework step (see Figure 4), we again compare the attackability of the generated traces with respect to the actual power traces. However, here the generative model's generalization ability is tested on VCD files corresponding to the same AES implementation but with a different key. Also, a different AES implementation with masking countermeasure is tested. As can be seen in Table II, the GAN generated traces behave similar to the real traces when we look at CPA ranking analysis. For example, in the case of the unprotected implementation, the number of attackable bytes was 13, which very close to the 14 attackable bytes when real traces were used. Similarly, in the mask-protected implementation, the number of the attackable was zero for the GAN generated traces, which is the same as the results obtained from real traces. This testifies the generative model's ability to generalize to different VCD inputs.

Conformance-Style: For conformance style testing, we compare the number of t-test points that fail for both generated and real. Given that GANs are innately stochastic, the number of points in the generated traces will be different from the real power traces. However, the generated traces follow a similar trend when compared to the actual traces. Table III shows that the real and generated traces follow a similar pattern for the number of t-test fail points. The table shows that for the unprotected implementation (with different key) and masked implementation how many test points failed for both designs with their ratio (e.g., 22/19 = 1.15). Although the protected implementation could not be attacked with CPA, TVLA shows that leakage is present. This means that more advanced attack could still lead to a successful attack. Note that we only applied a basic form of TVLA to study the accuracy of our proposed framework. Although popular, TVLA results have been proven to not quantify side-channel vulnerabilities correctly in certain cases [53]. Advanced TVLA techniques might provide a more accurate leakage assessment.

Signal processing metrics: The mean DTW distance between the real and the generated traces for the unprotected implementation was **1.08** and for the masked implementation was **1.21**. As before, the distance was calculated using FastDTW [52] and we used Euclidean distance as the distance measure for DTW. The Power spectral density (PSD) for the real and generated traces were almost identical, exactly like in Figure 8.

VI. Discussion & Conclusion

Our generative models were able to not only generate visually indistinguishable power traces from the training set but were also able to learn the characteristics of the VCD transition array which made the traces attackable using correlation power analysis. Through our experiments, we conclude that even using a few hundred VCD transition traces our GAN models were able to generate good quality power traces. However, we did not observe any substantial increase in performance when experimenting with different loss functions for either of the two GANs.

Since we conditioned the GAN on VCD transitions corresponding to the AES encryptions, we saw that the generated traces at times were more attackable than the actual power traces (like byte 8 in Table I). This is because the VCD is processed in a way that makes it much cleaner as opposed to the actual power traces. Also, due to the GAN's stochasticity, the generated power traces are not exactly identical to the actual power traces. This means that even though the synthetic power traces provide a good insight on the countermeasures efficacy and are a good substitute for the actual power traces, they are not a complete replacement for them.

Our framework has the ability to perform an early evaluation of secure IPs. This evaluation procedure usually requires as many as 10 million, 100 million or 1 Billion traces [13]. Using GANs, we were able to generate 1 million traces in 10.7s without any GPU support. Hence with enough RAM, the trace generation step can be performed on a standard desktop as quickly as 1.35 minutes, 13.5 minutes and 2.25 hours for 10 million, 100 million and 1 billion traces, respectively. In addition to the trace generation time, VCD generating and processing is a compute heavy part of this framework which will take 40 seconds for 1000 traces on a single CPU. The existing implementation makes use of line by line VCD processing and can be very easily be parallelized. Our current non-optimized VCD generation and translation engine is running on a common computer that requires about 4.62 days to process 10 million traces. These results have been obtained without any optimizations or usage of high-performance servers. Note that the collection of 10 million

real power traces would take approximately 9 days when the ChipWhisperer platform is used. Hence the GAN based power trace generation technique gives a **2x** speed-up as compared to even real time trace acquisition. The speed-up as compared to the CAD tools of course depends on the type of CAD tool used. As a rule of thumb such CAD tools usually take a few seconds to generate every power trace (the solution proposed by Sadhukhan *et al.* [11] takes 5.47 seconds to generate a single trace). Generating as many as 10 million power traces using this technique will take around 633 days. Hence the speed-up offered by the GAN methodology is **140x**.

After manufacturing, evaluation of real power traces will still be necessary to meet certain standards or certification. However, designers can be highly confident about the design at an early stage when a framework like this is used. It gives the designer quick ways to check if certain key values are more leaky and it additionally allows them to evaluate multiple designs simultaneously without the need to emulate or manufacture them.

Our experiments with differential augmentations, like those proposed by Zhao et al. [54] did not provide any enhancement in terms of attack ranks or training stability. Detailed experiments with other forms of differential augmentations might result in better performance even with much less training data. This, however, is an avenue for future work.

Limitations: Evaluation of GANs: Lack of well known evaluation metrics for generative models for non-image data [55] made us dependent on hardware security metrics like the CPA attack rank to evaluate the efficacy of our models. Being an active research avenue, we believe better evaluation metrics will soon be introduced that can be more proficient in evaluating the quality of the power traces.

Other crypto algorithms: Our framework is designed such that it can be extended to other crypto algorithms. However, since the trace length increases significantly for asymmetric algorithms like RSA or even software implementation of AES, a progressively growing convolutional architecture [56] like that proposed by Harrold et al. [57] must be used.

Hardware implementation: Different hardware implementations could end up with VCD files that have a different size. In our training process, we sliced the transition array for all implementation into the same size (60 samples). In some cases, when the VCD input was of shorter length, we can use padding to increase the vector size. To make the framework accommodate different size transition lists and power traces, the GAN architecture needs to be changed. The changed architecture can either be retrained completely or transfer learning can be used.

REFERENCES

[1] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," 2019.

[2] Intel, "A Security Model to Protect Intelligent Edge Devices." Available at: https://intel.ly/3CqyWS8. Accessed: 2021-05-08.

[3] G. Guez, "Why Hardware-Based Design Security is Essential for Every Application." Available at: https://bit.ly/2VAt9Zg. Accessed: 2021-05-08.

[4] M. Randolph and W. Diehl, "Power side-channel attack analysis: A review of 20 years of study for the layman," *Cryptography*, vol. 4, p. 15, May 2020.

[5] Rambus, "DPA Workstation Platform." Available at: https://www.rambus.com/security/dpa-countermeasures/dpa-workstation-platform/. Accessed: 2021-05-08.

[6] Riscure, "Inspector Side Channel Analysis." Available at: https://www.riscure.com/security-tools/inspector-sca. Accessed: 2021-05-08.

[7] R. Bloem, H. Gross, R. Iusupov, B. Könighofer, S. Mangard, and J. Winter, "Formal verification of masked hardware implementations in the presence of glitches," in *Advances in Cryptology – EUROCRYPT 2018*, 2018.

[8] A. Nahiyan, M. (Tony) He, J. Park, and M. Tehranipoor, *CAD for Side-Channel Leakage Assessment*. 2021.

[9] Miao, He, J. Park, A. Nahiyan, A. Vassilev, Y. Jin, and M. Tehranipoor, "Rtl-psc: Automated power side-channel leakage assessment at register-transfer level," 2019.

[10] H. Eldib, C. Wang, and P. Schaumont, "Formal verification of software countermeasures against side-channel attacks," 2014.

[11] R. Sadhukhan, P. Mathew, D. B. Roy, and D. Mukhopadhyay, "Count your toggles: a new leakage model for pre-silicon power analysis of crypto designs," *J. Electron. Test.*, 2019.

[12] A. Nahiyan, J. Park, M. He, Y. Iskander, F. Farahmandi, D. Forte, and M. Tehranipoor, "Script: A cad framework for power side-channel vulnerability assessment using information flow tracking and pattern generation," *ACM Transactions on Design Automation of Electronic Systems*, 2020.

[13] RAMBUS, "Dpa resistant core - rambus." https://www.rambus.com/security/dpa-countermeasures/dpa-resistant-core/. (Accessed on 05/10/2021).

[14] A. Brock, J. Donahue, and K. Simonyan, "Large scale gan training for high fidelity natural image synthesis," 2019.

[15] J. Engel, K. K. Agrawal, S. Chen, I. Gulrajani, C. Donahue, and A. Roberts, "Gansynth: Adversarial neural audio synthesis," 2019.

[16] A. Sagar, "Hrvgan: High resolution video generation using spatio-temporal gan," 2020.

[17] K. Antczak, "A generative adversarial approach to ecg synthesis and denoising," 2020.

[18] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2014.

[19] R. Prenger, R. Valle, and B. Catanzaro, "Waveglow: A flow-based generative network for speech synthesis," 2018.

[20] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," 2016.

[21] A. Odena, "Open questions about generative adversarial networks," *Distill*, 2019. https://distill.pub/2019/gan-open-problems.

[22] P. Wang, P. Chen, Z. Luo, G. Dong, M. Zheng, N. Yu, and H. Hu, "Enhancing the performance of practical profiling side-channel attacks using conditional generative adversarial networks," 2020.

[23] M. Mirza and S. Osindero, "Conditional generative adversarial nets," 2014.

[24] NIST, *Announcing the Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication 197, 2001.

[25] E. Oswald, S. Mangard, N. Pramstaller, and V. Rijmen, "A side-channel analysis resistant description of the aes s-box," in *Fast Software Encryption* (H. Gilbert and H. Handschuh, eds.), (Berlin, Heidelberg), pp. 413–423, Springer Berlin Heidelberg, 2005.

[26] M. Masoumi, P. Habibi, and M. Jadidi, "Efficient implementation of masked aes on side-channel attack standard evaluation board," in *2015 International Conference on Information Society (i-Society)*, pp. 151–156, 2015.

[27] H. Maghrebi, E. Prouff, S. Guilley, and J.-L. Danger, "A first-order leak-free masking countermeasure," in *Topics in Cryptology – CT-RSA 2012*, Springer Berlin Heidelberg, 2012.

[28] T. Popp and S. Mangard, "Masked dual-rail pre-charge logic: Dpa-resistance without routing constraints," in *Cryptographic Hardware and Embedded Systems – CHES 2005*, Springer Berlin Heidelberg, 2005.

[29] H. S. Kim and S. Hong, "New type of collision attack on first-order masked aess," *ETRI Journal*, vol. 38, no. 2, pp. 387–396, 2016.

[30] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014.

[31] X. Mao, Q. Li, H. Xie, R. Y. K. Lau, Z. Wang, and S. P. Smolley, "Least squares generative adversarial networks," 2017.

[32] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville, "Improved training of wasserstein gans," 2017.

[33] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein gan," 2017.

[34] G. Basso, "A hitchhikers guide to wasserstein distances." https://bit.ly/3lFixDe, June 2015. (Accessed on 05/12/2021).

[35] K. Kumar, R. Kumar, T. de Boissiere, L. Gestin, W. Z. Teoh, J. Sotelo, A. de Brebisson, Y. Bengio, and A. Courville, "Melgan: Generative adversarial networks for conditional waveform synthesis," 2019.

[36] M. Mathieu, C. Couprie, and Y. LeCun, "Deep multi-scale video prediction beyond mean square error," 2016.

[37] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," 2018.

[38] Siemens, "Questa Advanced Simulato." Available at: https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/. Accessed: 2021-05-08.

[39] N. Technology, "CW305 Artix FPGA Target Board." Available at: http://store.newae.com/cw305-artix-fpga-target-board/. Accessed: 2021-05-08.

[40] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[41] S. Chari, J. R. Rao, and P. Rohatgi, "Template attacks," in *Cryptographic Hardware and Embedded Systems*, 2002.

[42] H. Maghrebi, T. Portigliatti, and E. Prouff, "Breaking cryptographic implementations using deep learning techniques," in *IACR Cryptology ePrint Archive*, 2016.

[43] P. C. Kocher, J. Jae, and B. Jun, "Differential Power Analysis," in *CRYPTO*, 1999.

[44] E. Brier *et al.*, "Correlation power analysis with a leakage model," in *CHES*, 2004.

[45] G. Becker *et al.*, "Test Vector Leakage Assessment (TVLA) methodology in practice." Available at: https://pdfs.semanticscholar.org/, 2011. Accessed: 2019-09-23.

[46] S. Mangard, "Hardware Countermeasures against DPA – A Statistical Analysis of Their Effectiveness," in *CT-RSA*, 2004.

[47] M. Müller, *Information Retrieval for Music and Motion*. 01 2007.

[48] P. Stoica and R. Moses, "Spectral analysis of signals," *Prentice Hall*, 01 2005.

[49] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2016.

[50] R. Tubbing, "An Analysis of Deep Learning Based Profiled Side-channel Attacks," Master's thesis, Delft University of Technology, the Netherlands, 2019.

[51] M. Lucic, K. Kurach, M. Michalski, S. Gelly, and O. Bousquet, "Are gans created equal? a large-scale study," 2018.

[52] S. Salvador and P. Chan, "Fastdtw: Toward accurate dynamic time warping in linear time and space," 2004.

[53] "Leak me if you can : Does tvla reveal success rate ?," 2017.

[54] S. Zhao, Z. Liu, J. Lin, J.-Y. Zhu, and S. Han, "Differentiable augmentation for data-efficient gan training," 2020.

[55] A. Borji, "Pros and cons of gan evaluation measures," 2018.

[56] T. Karras, T. Aila, S. Laine, and J. Lehtinen, "Progressive growing of gans for improved quality, stability, and variation," 2018.

[57] A. Harell, R. Jones, S. Makonin, and I. V. Bajic, "Powergan: Synthesizing appliance power signatures using generative adversarial networks," 2020.

BIBLIOGRAPHY

[1] C. Paar and J. Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*, 1st ed. (Springer Publishing Company, Incorporated, 2009).

[2] B. Gierlichs, *Introduction to power analysis,* https://www.cosic.esat.kuleuven.be/ecrypt/courses/albena11/slides/benedikt_gierlichs_power_analysis.pdf (2011), (Accessed on 01/25/2021).

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016) http://www.deeplearningbook.org.

[4] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning* (2020) https://d2l.ai.

[5] B. Soret, I. L. Mayorga, S. Cioni, and P. Popovski, *5g satellite networks for iot: offloading and backhauling,* CoRR **abs/2011.05202** (2020), arXiv:2011.05202 .

[6] A. Balaji, T. Marty, A. Das, and F. Catthoor, *Run-time mapping of spiking neural networks to neuromorphic hardware,* CoRR **abs/2006.06777** (2020), arXiv:2006.06777 .

[7] *The global risks report 2021 | world economic forum,* https://www.weforum.org/reports/the-global-risks-report-2021 (2021), (Accessed on 06/09/2021).

[8] McAfee, *New mcafee report estimates global cybercrime losses to exceed $1 trillion| mcafee press release,* https://www.mcafee.com/enterprise/en-us/about/newsroom/press-releases/press-release.html?news_id=6859bd8c-9304-4147-bdab-32b35457e629 (2020), (Accessed on 06/09/2021).

[9] McAfee, *The hidden costs of cybercrime,* https://www.mcafee.com/enterprise/en-us/assets/reports/rp-hidden-costs-of-cybercrime.pdf (2020), (Accessed on 29/07/2021).

[10] kaspersky, *Ransomware wannacry: All you need to know | kaspersky,* https://www.kaspersky.com/resource-center/threats/ransomware-wannacry, (Accessed on 06/09/2021).

[11] S. Ghafur, S. Kristensen, K. Honeyford, G. Martin, A. Darzi, and P. Aylin, *A retrospective impact analysis of the wannacry cyberattack on the nhs,* npj Digital Medicine **2** (2019), 10.1038/s41746-019-0161-6.

[12] Avast, *Tsmc shut down by wannacry variant | avast,* https://blog.avast.com/tsmc-wannacry-variant (2018), (Accessed on 06/09/2021).

[13] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard, *Systematic classification of side-channel attacks: A case study for mobile devices,* IEEE Communications Surveys Tutorials **PP** (2017), 10.1109/COMST.2017.2779824.

[14] I. Verbauwhede, D. Karaklajic, and J.-M. Schmidt, *The fault attack jungle - a classification model to guide you,* (2011) pp. 3–8.

[15] J. Rajendran, E. Gavas, J. Jimenez, V. Padman, and R. Karri, *Towards a comprehensive and systematic classification of hardware trojans,* in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems* (2010) pp. 1871–1874.

[16] D. Genkin, A. Shamir, and E. Tromer, *Rsa key extraction via low-bandwidth acoustic cryptanalysis,* in *Advances in Cryptology – CRYPTO 2014*, edited by J. A. Garay and R. Gennaro (Springer Berlin Heidelberg, Berlin, Heidelberg, 2014) pp. 444–461.

[17] K. Gandolfi, C. Mourtel, and F. Olivier, *Electromagnetic analysis: Concrete results,* in *Cryptographic Hardware and Embedded Systems — CHES 2001*, edited by Ç. K. Koç, D. Naccache, and C. Paar (Springer Berlin Heidelberg, Berlin, Heidelberg, 2001) pp. 251–261.

[18] E. Brier, C. Clavier, and F. Olivier, *Correlation power analysis with a leakage model,* in *Cryptographic Hardware and Embedded Systems - CHES 2004*, edited by M. Joye and J.-J. Quisquater (Springer Berlin Heidelberg, Berlin, Heidelberg, 2004) pp. 16–29.

[19] P. C. Kocher, *Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems,* in *Advances in Cryptology — CRYPTO '96*, edited by N. Koblitz (Springer Berlin Heidelberg, Berlin, Heidelberg, 1996) pp. 104–113.

[20] A. Schlösser, D. Nedospasov, J. Krämer, S. Orlic, and J.-P. Seifert, *Simple photonic emission analysis of aes,* in *Cryptographic Hardware and Embedded Systems – CHES 2012*, edited by E. Prouff and P. Schaumont (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012) pp. 41–57.

[21] J. Da Rolt, A. Das, G. Di Natale, M.-L. Flottes, B. Rouzeyre, and I. Verbauwhede, *A new scan attack on rsa in presence of industrial countermeasures,* in *Constructive Side-Channel Analysis and Secure Design*, edited by W. Schindler and S. A. Huss (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012) pp. 89–104.

[22] Rambus, *DPA Workstation Platform,* Available at: https://www.rambus.com/security/dpa-countermeasures/dpa-workstation-platform/, accessed: 2021-05-08.

[23] Riscure, *Inspector Side Channel Analysis,* Available at: https://www.riscure.com/security-tools/inspector-sca, accessed: 2021-05-08.

[24] I. Buhan, L. Batina, Y. Yarom, and P. Schaumont, *Sok: Design tools for side-channel-aware implementations,* (2021), arXiv:2104.08593 [cs.CR] .

[25] D. McCann, E. Oswald, and C. Whitnall, *Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages,* in *26th USENIX Security Symposium (USENIX Security 17)* (USENIX Association, Vancouver, BC, 2017) pp. 199–216.

[26] N. Veshchikov and S. Guilley, *Use of simulators for side-channel analysis,* in *2017 IEEE European Symposium on Security and Privacy Workshops, EuroSP Workshops 2017, Paris, France, April 26-28, 2017* (IEEE, 2017) pp. 104–112.

[27] R. Bloem, H. Gross, R. Iusupov, B. Könighofer, S. Mangard, and J. Winter, *Formal verification of masked hardware implementations in the presence of glitches,* in *Advances in Cryptology – EUROCRYPT 2018* (2018).

[28] R. Sadhukhan, P. Mathew, D. B. Roy, and D. Mukhopadhyay, *Count your toggles: a new leakage model for pre-silicon power analysis of crypto designs,* J. Electron. Test. (2019).

[29] Miao, He, J. Park, A. Nahiyan, A. Vassilev, Y. Jin, and M. Tehranipoor, *Rtl-psc: Automated power side-channel leakage assessment at register-transfer level,* (2019), arXiv:1901.05909 [cs.CR] .

[30] H. Eldib, C. Wang, and P. Schaumont, *Formal verification of software countermeasures against side-channel attacks,* (2014).

[31] A. Nahiyan, J. Park, M. He, Y. Iskander, F. Farahmandi, D. Forte, and M. Tehranipoor, *Script: A cad framework for power side-channel vulnerability assessment using information flow tracking and pattern generation,* ACM Transactions on Design Automation of Electronic Systems (2020).

[32] RAMBUS, *Dpa resistant core - rambus,* https://www.rambus.com/security/dpa-countermeasures/dpa-resistant-core/, (Accessed on 05/10/2021).

[33] S. D. Desai, S. Giraddi, N. Verma, P. Gupta, and S. Ramya, *Breast cancer detection using gan for limited labeled dataset,* in *2020 12th International Conference on Computational Intelligence and Communication Networks (CICN)* (2020) pp. 34–39.

[34] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. P. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi, *Photo-realistic single image super-resolution using a generative adversarial network,* CoRR **abs/1609.04802** (2016), arXiv:1609.04802 .

[35] K. Nazeri and E. Ng, *Image colorization with generative adversarial networks,* CoRR **abs/1803.05400** (2018), arXiv:1803.05400 .

[36] T. E. of Encyclopaedia Britannica, *Histiaeus,* (2015).

[37] C. Paar and J. Pelzl, *Introduction to cryptography and data security,* in *Understanding Cryptography: A Textbook for Students and Practitioners* (Springer Publishing Company, Incorporated, 2009) 1st ed., Chap. 1, p. 2.

[38] J. J. G. Savard, *The ideal cipher,* http://www.quadibloc.com/crypto/mi0611.htm (1999), (Accessed on 01/15/2021).

[39] J. Siegel, *History of des,* (2013), (Accessed on 01/15/2021).

[40] H. M. Heys, *A tutorial on linear and differential cryptanalysis,* http://www.engr.mun.ca/~howard/PAPERS/ldc_tutorial.pdf, (Accessed on 01/15/2021).

[41] D. J. Bernstein, *The chacha family of stream ciphers,* https://cr.yp.to/chacha.html (2008), (Accessed on 01/18/2021).

[42] T. Unterluggauer and E. Wenger, *Efficient pairings and ecc for embedded systems,* in *CHES* (Springer, 2014) pp. 298–315.

[43] C. Paar and J. Pelzl, *Existence of finite fields,* in *Understanding Cryptography: A Textbook for Students and Practitioners* (Springer Publishing Company, Incorporated, 2009) 1st ed., Chap. 9, p. 255.

[44] M. Randolph and W. Diehl, *Power side-channel attack analysis: A review of 20 years of study for the layman,* Cryptography **4**, 15 (2020).

[45] J.-J. Quisquater and S. David, *Electromagnetic attack,* in *Encyclopedia of Cryptography and Security*, edited by H. C. A. van Tilborg (Springer US, Boston, MA, 2005) pp. 170–174.

[46] H. Wang, D. Ji, Y. Zhang, K. Chen, J. Chen, and Y. Wang, *Optical side channel attacks on singlechip,* in *Proceedings of the 2015 International Conference on Industrial Technology and Management Science* (Atlantis Press, 2015/11) pp. 364–369.

[47] P. Cheng, I. E. Bagci, U. Roedig, and J. Yan, *Sonarsnoop: Active acoustic side-channel attacks,* (2018), arXiv:1808.10250 [cs.CR] .

[48] H. Aly and M. ElGayyar, *Attacking aes using bernstein's attack on modern processors,* in *Progress in Cryptology – AFRICACRYPT 2013*, edited by A. Youssef, A. Nitaj, and A. E. Hassanien (Springer Berlin Heidelberg, Berlin, Heidelberg, 2013) pp. 127–139.

[49] D. A. Osvik, A. Shamir, and E. Tromer, *Cache attacks and countermeasures: The case of aes,* in *Topics in Cryptology – CT-RSA 2006*, edited by D. Pointcheval (Springer Berlin Heidelberg, Berlin, Heidelberg, 2006) pp. 1–20.

[50] X. Li, W. Li, J. Ye, H. Li, and Y. Hu, *Scan chain based attacks and countermeasures: A survey,* IEEE Access **7**, 85055 (2019).

[51] R. X. Gu and M. I. Elmasry, *Power dissipation analysis and optimization of deep submicron cmos digital circuits,* IEEE Journal of Solid-State Circuits **31**, 707 (1996).

[52] P. Venkatachalam, *S-net, a neural network based countermeasure for aes,* https://repository.tudelft.nl/islandora/object/uuid:59001df2-9b47-47c2-bfbe-d1c016902795 (2019), (Accessed on 01/21/2021).

[53] S. Mangard, *A simple power-analysis (spa) attack on implementations of the aes key expansion,* in *Information Security and Cryptology — ICISC 2002*, edited by P. J. Lee and C. H. Lim (Springer Berlin Heidelberg, Berlin, Heidelberg, 2003) pp. 343–358.

[54] M. Joye, P. Paillier, and B. Schoenmakers, *On second-order differential power analysis,* in *Cryptographic Hardware and Embedded Systems – CHES 2005*, edited by J. R. Rao and B. Sunar (Springer Berlin Heidelberg, Berlin, Heidelberg, 2005) pp. 293–308.

[55] A. A. Ding, L. Zhang, Y. Fei, and P. Luo, *A statistical model for higher order dpa on masked devices,* in *Cryptographic Hardware and Embedded Systems – CHES 2014*, edited by L. Batina and M. Robshaw (Springer Berlin Heidelberg, Berlin, Heidelberg, 2014) pp. 147–169.

[56] S. Chari, J. R. Rao, and P. Rohatgi, *Template attacks,* in *Cryptographic Hardware and Embedded Systems - CHES 2002*, edited by B. S. Kaliski, ç. K. Koç, and C. Paar (Springer Berlin Heidelberg, Berlin, Heidelberg, 2003) pp. 13–28.

[57] S. Picek, A. Heuser, A. Jovic, S. Bhasin, and F. Regazzoni, *The curse of class imbalance and conflicting metrics with machine learning for side-channel evaluations,* IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**, 209 (2018).

[58] W. Hu, C.-H. Chang, A. Sengupta, S. Bhunia, R. Kastner, and H. Li, *An overview of hardware security and trust: Threats, countermeasures, and design tools,* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **40**, 1010 (2021).

[59] J.-L. Danger, S. Guilley, S. Bhasin, and M. Nassar, *Overview of dual rail with precharge logic styles to thwart implementation-level attacks on hardware cryptoprocessors,* (2009) pp. 1 – 8.

[60] M. Doulcier-Verdier, J.-M. Dutertre, J. Fournier, J.-B. Rigaud, B. Robisson, and A. Tria, *A side-channel and fault-attack resistant aes circuit working on duplicated complemented values,* (2011) pp. 274–276.

[61] M. Masoumi, P. Habibi, A. Dehghan, M. Jadidi, and L. Yousefi, *Efficient implementation of power analysis attack resistant advanced encryption standard algorithm on side-channel attack standard evaluation board,* International Journal of Internet Technology and Secured Transactions **6**, 203 (2016).

[62] V. Immler, J. Obermaier, K. K. Ng, F. X. Ke, J. Lee, Y. P. Lim, W. K. Oh, K. H. Wee, and G. Sigl, *Secure physical enclosures from covers with tamper-resistance,* IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**, 51 (2018).

[63] ARM, *Trustzone for cortex-a,* `https://www.arm.com/why-arm/technologies/trustzone-for-cortex-a`, (Accessed on 05/27/2021).

[64] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *Pytorch: An imperative style, high-performance deep learning library,* in *Advances in Neural Information Processing Systems 32*, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Curran Associates, Inc., 2019) pp. 8024–8035.

[65] F. Chollet *et al.*, *Keras,* `https://github.com/fchollet/keras` (2015).

[66] W. S. McCulloch and W. Pitts, *A logical calculus of the ideas immanent in nervous activity,* The bulletin of mathematical biophysics **5**, 115 (1943).

[67] K.-H. Tan and B. P. Lim, *The artificial intelligence renaissance: deep learning and the road to human-level machine intelligence,* APSIPA Transactions on Signal and Information Processing **7**, e6 (2018).

[68] S. Hochreiter and J. Schmidhuber, *Long short-term memory,* Neural Comput. **9**, 1735–1780 (1997).

[69] Z. Liu, P. Luo, X. Wang, and X. Tang, *Deep learning face attributes in the wild,* in *Proceedings of International Conference on Computer Vision (ICCV)* (2015).

[70] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, *Generative adversarial networks,* (2014), arXiv:1406.2661 [stat.ML] .

[71] T. Karras, T. Aila, S. Laine, and J. Lehtinen, *Progressive growing of gans for improved quality, stability, and variation,* (2018), arXiv:1710.10196 [cs.NE] .

[72] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville, *Improved training of wasserstein gans,* (2017), arXiv:1704.00028 [cs.LG] .

[73] A. Brock, J. Donahue, and K. Simonyan, *Large scale gan training for high fidelity natural image synthesis,* (2019), arXiv:1809.11096 [cs.LG] .

[74] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, *Wavenet: A generative model for raw audio,* (2016), arXiv:1609.03499 [cs.SD] .

[75] R. Yamamoto, E. Song, and J.-M. Kim, *Parallel wavegan: A fast waveform generation model based on generative adversarial networks with multi-resolution spectrogram,* (2020), arXiv:1910.11480 [eess.AS] .

[76] C. Donahue, J. McAuley, and M. Puckette, *Adversarial audio synthesis,* (2019), arXiv:1802.04208 [cs.SD] .

[77] K. Kumar, R. Kumar, T. de Boissiere, L. Gestin, W. Z. Teoh, J. Sotelo, A. de Brebisson, Y. Bengio, and A. Courville, *Melgan: Generative adversarial networks for conditional waveform synthesis,* (2019), arXiv:1910.06711 [eess.AS] .

[78] K. G. Hartmann, R. T. Schirrmeister, and T. Ball, *Eeg-gan: Generative adversarial networks for electroencephalograhic (eeg) brain signals,* (2018), arXiv:1806.01875 [eess.SP] .

[79] A. M. Delaney, E. Brophy, and T. E. Ward, *Synthesis of realistic ecg using generative adversarial networks,* (2019), arXiv:1909.09150 [eess.SP] .

[80] S. Alwon, *Generative adversarial networks in seismic data processing,* (2018) pp. 1991–1995.

[81] J. Yoon, D. Jarrett, and M. van der Schaar, *Time-series generative adversarial networks,* in *Advances in Neural Information Processing Systems*, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Curran Associates, Inc., 2019) pp. 5508–5518.

[82] M. Mirza and S. Osindero, *Conditional generative adversarial nets,* (2014), arXiv:1411.1784 [cs.LG] .

[83] X. Mao, Q. Li, H. Xie, R. Y. K. Lau, Z. Wang, and S. P. Smolley, *Least squares generative adversarial networks,* (2017), arXiv:1611.04076 [cs.CV] .

[84] M. Arjovsky, S. Chintala, and L. Bottou, *Wasserstein gan,* (2017), arXiv:1701.07875 [stat.ML] .

[85] G. Basso, *A hitchhikers guide to wasserstein distances,* https://bit.ly/3fEYJeL (2015), (Accessed on 05/12/2021).

[86] G. Developers, *Gan training problems,* https://developers.google.com/machine-learning/gan/training (2019), (Accessed on 06/05/2021).

[87] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, *Attention is all you need,* (2017), arXiv:1706.03762 [cs.CL] .

[88] G. Papamakarios, T. Pavlakou, and I. Murray, *Masked autoregressive flow for density estimation,* (2018), arXiv:1705.07057 [stat.ML] .

[89] A. Odena, *Open questions about generative adversarial networks,* Distill (2019), 10.23915/distill.00018, https://distill.pub/2019/gan-open-problems.

[90] A. Grover, *Notes on deep generative models,* https://deepgenerativemodels.github.io/ (2018).

[91] S. Bond-Taylor, A. Leach, Y. Long, and C. G. Willcocks, *Deep generative modelling: A comparative review of vaes, gans, normalizing flows, energy-based and autoregressive models,* (2021), arXiv:2103.04922 [cs.LG] .

[92] A. Graves, *Generating sequences with recurrent neural networks,* (2014), arXiv:1308.0850 [cs.NE] .

[93] H. Li, M. Krček, and G. Perin, *A comparison of weight initializers in deep learning-based side-channel analysis,* Cryptology ePrint Archive, Report 2020/904 (2020), https://eprint.iacr.org/2020/904.

[94] R. Grosse, *Lecture notes in neural networks and deep learning,* (2019).

[95] A. Aljuffri, *Exploring deep learning for hardware attacks,* http://resolver.tudelft.nl/uuid:c0dddd21-bdc1-4641-bd5d-4abdbd7fe35f (2018), (Accessed on 01/21/2021).

[96] H. Wang, *Side-channel analysis of aes based on deep learning,* https://www.diva-portal.org/smash/get/diva2:1325691/FULLTEXT01.pdf (2019), (Accessed on 01/21/2021).

[97] H. Maghrebi, T. Portigliatti, and E. Prouff, *Breaking cryptographic implementations using deep learning techniques,* Cryptology ePrint Archive, Report 2016/921 (2016), https://eprint.iacr.org/2016/921.

[98] J. Shlens, *A tutorial on principal component analysis,* (2014), arXiv:1404.1100 [cs.LG] .

[99] L. van der Maaten and G. Hinton, *Viualizing data using t-sne,* Journal of Machine Learning Research **9**, 2579 (2008).

[100] W. Xie, A. Nunez, R. Maas, and Z. Li, *Decision support tool based on multi-source data analysis for the tram wheel-rail interface,* IFAC-PapersOnLine **51**, 118 (2018), 15th IFAC Symposium on Control in Transportation Systems CTS 2018.

[101] J. Zhang, W. Zhang, R. Song, L. Ma, and Y. Li, *Grasp for stacking via deep reinforcement learning,* in *2020 IEEE International Conference on Robotics and Automation (ICRA)* (2020) pp. 2543–2549.

[102] A. Krizhevsky, *Learning multiple layers of features from tiny images,* University of Toronto (2012).

[103] A. CAUCHY, *Methode generale pour la resolution des systemes d'equations simultanees,* C.R. Acad. Sci. Paris **25**, 536 (1847).

[104] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl, *On empirical comparisons of optimizers for deep learning,* CoRR **abs/1910.05446** (2019), arXiv:1910.05446 .

[105] L. Huang, J. Qin, Y. Zhou, F. Zhu, L. Liu, and L. Shao, *Normalization techniques in training dnns: Methodology, analysis and application,* (2020), arXiv:2009.12836 [cs.LG] .

[106] scikit-learn developers, *sklearn.preprocessing.minmaxscaler — scikit-learn 0.24.2 documentation,* https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html (), (Accessed on 05/30/2021).

[107] scikit-learn developers, *sklearn.preprocessing.standardscaler — scikit-learn 0.24.2 documentation,* https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html (), (Accessed on 06/01/2021).

[108] scikit-learn developers, *sklearn.preprocessing.maxabsscaler — scikit-learn 0.24.2 documentation,* https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MaxAbsScaler.html#sklearn.preprocessing.MaxAbsScaler (), (Accessed on 05/30/2021).

[109] scikit-learn developers, *sklearn.preprocessing.robustscaler — scikit-learn 0.24.2 documentation,* https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html (), (Accessed on 05/30/2021).

[110] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift,* (2015), arXiv:1502.03167 [cs.LG] .

[111] D. Foster, *Generative Deep Learning: Teaching Machines to Paint, Write, Compose, and Play,* 1st ed. (O'Reilly Media, 2019).

[112] scikit-learn developers, *Compare the effect of different scalers on data with outliers — scikit-learn 0.24.2 documentation,* https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html (), (Accessed on 06/01/2021).

[113] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, *Dropout: A simple way to prevent neural networks from overfitting,* Journal of Machine Learning Research (2014).

[114] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, *Caffe: Convolutional architecture for fast feature embedding,* (2014), arXiv:1408.5093 [cs.CV] .

[115] E. D. D. Team, *DL4J: Deep Learning for Java,* (2016).

[116] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems,* (2015), software available from tensorflow.org.

[117] P. Wang, P. Chen, Z. Luo, G. Dong, M. Zheng, N. Yu, and H. Hu, *Enhancing the performance of practical profiling side-channel attacks using conditional generative adversarial networks,* (2020), arXiv:2007.05285 [cs.CR] .

[118] S. Zhao, Z. Liu, J. Lin, J.-Y. Zhu, and S. Han, *Differentiable augmentation for data-efficient gan training,* (2020), arXiv:2006.10738 [cs.CV] .

[119] A. Odena, C. Olah, and J. Shlens, *Conditional image synthesis with auxiliary classifier gans,* (2017), arXiv:1610.09585 [stat.ML] .

[120] A. Vlogiaris, *Data augmentation techniques using generative adversarial neural networks on side channel analysis,* http://resolver.tudelft.nl/uuid:d2d00b11-cea1-466e-9b17-2b244e33be25 (2021), (Accessed on 07/08/2021).

[121] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-based learning applied to document recognition,* Proceedings of the IEEE **86**, 2278 (1998).

[122] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, *Scikit-learn: Machine learning in Python,* Journal of Machine Learning Research **12**, 2825 (2011).

[123] C.-C. Lo, S.-W. Fu, W.-C. Huang, X. Wang, J. Yamagishi, Y. Tsao, and H.-M. Wang, *Mosnet: Deep learning based objective assessment for voice conversion,* (2019), arXiv:1904.08352 [cs.SD] .

[124] S. Salvador and P. Chan, *Fastdtw: Toward accurate dynamic time warping in linear time and space,* (2004).

[125] P. Stoica and R. Moses, *Spectral analysis of signals,* Prentice Hall (2005).

[126] I. Dokmanic, R. Parhizkar, J. Ranieri, and M. Vetterli, *Euclidean distance matrices: Essential theory, algorithms, and applications,* IEEE Signal Processing Magazine **32**, 12–30 (2015).

[127] M. Müller, *Information Retrieval for Music and Motion* (2007).

[128] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, *Gans trained by a two time-scale update rule converge to a local nash equilibrium,* (2018), arXiv:1706.08500 [cs.LG] .

[129] S. Barratt and R. Sharma, *A note on the inception score,* (2018), arXiv:1801.01973 [stat.ML] .

[130] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, *Going deeper with convolutions,* (2014), arXiv:1409.4842 [cs.CV] .

[131] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, *Imagenet: A large-scale hierarchical image database,* in *2009 IEEE conference on computer vision and pattern recognition* (Ieee, 2009) pp. 248–255.

[132] T. Messerges, E. Dabbish, and R. Sloan, *Power analysis attacks of modular exponentiation in smartcards,* (1999) pp. 144–157.

[133] S. Mangard, *Hardware countermeasures against dpa – a statistical analysis of their effectiveness,* in *Topics in Cryptology – CT-RSA 2004*, edited by T. Okamoto (Springer Berlin Heidelberg, Berlin, Heidelberg, 2004) pp. 222–235.

[134] A. Cai, *Comparison of side channel analysis measurement setups,* https://research.tue.nl/en/studentTheses/comparison-of-side-channel-analysis-measurement-setups (2015), (Accessed on 06/10/2021).

[135] C. O'Flynn and Z. Chen, *Chipwhisperer: An open-source platform for hardware embedded security research,* (2014).

[136] G. Becker *et al.*, *Test Vector Leakage Assessment (TVLA) methodology in practice,* Available at: https://pdfs.semanticscholar.org/ (2011), accessed: 2019-09-23.

[137] D. B. Roy, S. Bhasin, S. Guilley, A. Heuser, S. Patranabis, and D. Mukhopadhyay, *Leak me if you can: Does tvla reveal success rate?* Cryptology ePrint Archive, Report 2016/1152 (2016), https://eprint.iacr.org/2016/1152.

[138] N. Veshchikov, *Silk: High level of abstraction leakage simulator for side channel analysis,* in *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, PPREW-4 (Association for Computing Machinery, New York, NY, USA, 2014).

[139] X. Chen, Y. Duan, R. Houthooft, J. Schulman, I. Sutskever, and P. Abbeel, *Infogan: Interpretable representation learning by information maximizing generative adversarial nets,* CoRR **abs/1606.03657** (2016), arXiv:1606.03657 .

[140] Y. LeCun, C. Cortes, and C. Burges, *Mnist handwritten digit database,* ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist **2** (2010).

[141] ZipCPU, *Writing your own vcd file,* https://zipcpu.com/blog/2017/07/31/vcd.html, (Accessed on 05/30/2021).

[142] Mathworks, *Measuring signal similarities,* .

[143] A. Radford, L. Metz, and S. Chintala, *Unsupervised representation learning with deep convolutional generative adversarial networks,* (2016), arXiv:1511.06434 [cs.LG] .

[144] L. Yu, W. Zhang, J. Wang, and Y. Yu, *Seqgan: Sequence generative adversarial nets with policy gradient,* (2017), arXiv:1609.05473 [cs.LG] .

[145] O. Mogren, *C-rnn-gan: Continuous recurrent neural networks with adversarial training,* (2016), arXiv:1611.09904 [cs.AI] .

[146] X. Ding, Y. Wang, Z. Xu, W. J. Welch, and Z. J. Wang, *Continuous conditional generative adversarial networks for image generation: Novel losses and label input mechanisms,* (2021), arXiv:2011.07466 [cs.CV] .

[147] J. Brownlee, *Tips for training stable generative adversarial networks,* https://machinelearningmastery.com/how-to-train-stable-generative-adversarial-networks/ (2019), (Accessed on 06/13/2021).

[148] T. Salimans and D. P. Kingma, *Weight normalization: A simple reparameterization to accelerate training of deep neural networks,* (2016), arXiv:1602.07868 [cs.LG] .

[149] S. Xiang and H. Li, *On the effects of batch and weight normalization in generative adversarial networks,* (2017), arXiv:1704.03971 [stat.ML] .

[150] T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida, *Spectral normalization for generative adversarial networks,* (2018), arXiv:1802.05957 [cs.LG] .

[151] K. Kurach, M. Lucic, X. Zhai, M. Michalski, and S. Gelly, *A large-scale study on regularization and normalization in gans,* (2019), arXiv:1807.04720 [cs.LG] .

[152] Y. Yazici, C.-S. Foo, S. Winkler, K.-H. Yap, and V. Chandrasekhar, *Empirical analysis of overfitting and mode drop in gan training,* (2020), arXiv:2006.14265 [cs.LG] .

[153] H. Thanh-Tung and T. Tran, *Catastrophic forgetting and mode collapse in gans,* (2020) pp. 1–10.

[154] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization,* (2017), arXiv:1412.6980 [cs.LG] .

[155] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, *On the importance of initialization and momentum in deep learning,* in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13 (JMLR.org, 2013) p. III–1139–III–1147.

[156] G. Hinton, *lec6.pdf,* https://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf, (Accessed on 05/29/2021).

[157] M. Lucic, K. Kurach, M. Michalski, S. Gelly, and O. Bousquet, *Are gans created equal? a large-scale study,* (2018), arXiv:1711.10337 [stat.ML] .

[158] NVIDIA, P. Vingelmann, and F. H. Fitzek, *Cuda, release: 10.2.89,* (2020).

[159] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, *PyCUDA and PyOpenCL: a scripting-based approach to GPU run-time code generation,* Parallel Computing **38**, 157 (2012).

[160] S. K. Lam, A. Pitrou, and S. Seibert, *Numba: A llvm-based python jit compiler,* in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15 (Association for Computing Machinery, New York, NY, USA, 2015).

[161] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, *Cython: The best of both worlds,* Computing in Science & Engineering **13**, 31 (2011).

[162] J. Bai, F. Lu, K. Zhang, *et al.*, *Onnx: Open neural network exchange,* https://github.com/onnx/onnx (2019).

[163] E. Bisong, *Google colaboratory,* in *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners* (Apress, Berkeley, CA, 2019) pp. 59–64.

[164] L. Roeder, *Netron: Visualizer for neural networks,* https://github.com/lutzroeder/netron (2021).

[165] B. Adlam, C. Weill, and A. Kapoor, *Investigating under and overfitting in wasserstein generative adversarial networks,* (2019), arXiv:1910.14137 [stat.ML] .

[166] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, *Smote: Synthetic minority over-sampling technique,* Journal of Artificial Intelligence Research **16**, 321–357 (2002).

[167] R. Sadhukhan, P. Mathew, D. B. Roy, and D. Mukhopadhyay, *Count your toggles: a new leakage model for pre-silicon power analysis of crypto designs,* Journal of Electronic Testing **35**, 605 (2019).

[168] Y. Wang, C. Wu, L. Herranz, J. van de Weijer, A. Gonzalez-Garcia, and B. Raducanu, *Transferring gans: generating images from limited data,* (2018), arXiv:1805.01677 [cs.CV] .

[169] B. Hettwer, T. Horn, S. Gehrer, and T. Güneysu, *Encoding power traces as images for efficient side-channel analysis,* (2020), arXiv:2004.11015 [cs.CR] .

[170] Z. Wang and T. Oates, *Imaging time-series to improve classification and imputation,* (2015), arXiv:1506.00327 [cs.LG] .

[171] M. Arbel, L. Zhou, and A. Gretton, *Generalized energy based models,* in *International Conference on Learning Representations* (2021).

[172] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition,* (2015), arXiv:1512.03385 [cs.CV] .

[173] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, *Finn,* Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (2017), 10.1145/3020078.3021744.

[174] C. Lattner and V. Adve, *LLVM: A compilation framework for lifelong program analysis and transformation,* (San Jose, CA, USA, 2004) pp. 75–88.

[175] J.-L. Danger, S. Guilley, S. Bhasin, and M. Nassar, *Overview of dual rail with precharge logic styles to thwart implementation-level attacks on hardware cryptoprocessors,* (2009) pp. 1 – 8.

[176] M. Masoumi, *Novel hybrid cmos/memristor implementation of the aes algorithm robust against differential power analysis attack,* IEEE Transactions on Circuits and Systems II: Express Briefs **67**, 1314 (2020).