Opleiding Informatica

Expanding Side-Channel Data Sets

Using Conditional Generative Adversarial Networks

Joren van den Berg

Supervisors:
Guilherme Perin

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

06/04/2023

Abstract

Side-channel analyses using deep learning usually require large data sets to perform well. Obtaining these data sets can prove to be quite difficult, as manufactures of chips implement certain safeguards against obtaining the required measures. The performance of side-channel analyses can be improved by generating synthetic traces from the original data set of traces instead of measuring directly from the target device. This research proposes using conditional generative adversarial networks (cGAN) to generate the synthetic traces, on which a deep-learning model can be trained to execute a Deep-Learning Side-Channel Analysis. We will compare the effectiveness of various cGAN models and investigate the effect of certain hyperparameters on the performance of our analysis. The experiments will be performed on an ATMega8515 chip running AES 128-bit encryption. To compare our models, we will calculate the guessing entropy of the predictions made by a MLP model, trained on the synthetic traces generated by our cGAN model. We will also compare models using the Signal-To-Noise ratio of the synthetic traces and the real traces (the SNR should be similar). Our results show much promise in using cGAN models to enhance existing data sets and using them perform Side-Channel Analyses.

Contents

1 Introduction

Cryptographic algorithms are mathematically secure. Knowing the input and output data (i.e., plaintexts and ciphertexts), as well as every detail about the cryptographic algorithm, is not enough to recover the key. However, when a cryptographic implementation executes on electronic devices (e.g., microprocessors, microcontrolers), there are several sources of unintended and unavoidable information leakages. Most common types are power consumption, electromagnetic emission, execution time, temperature and acoustics. An attacker may be able to measure these side-channel information during the execution of encryption or decryption operations and to perform statistical analysis to extract the key from the side-channel measurements.

Side-Channel Analyses (SCA) mainly consist of two categories: `profiling` attacks and `non-profiling` attacks. Non-profiling attacks mean that we recover the secret key by analyzing the leakages of a target device directly. Differential power analysis (DPA) [KJJ99] and Correlation Power Analysis (CPA) [BCO04] are examples of direct or non-profiling attacks. With profiling attacks, we are in the possession of an open identical copy of the target device, of which we know the secret key beforehand. We use this copy to construct a model (in our case, a deep learning model), which we later use to attack our target device.

1.1 The problem

Obtaining enough measurements for our attack to be successful, can be time-consuming and expensive, if not sometimes impossible. When the encryption algorithm we are attacking is part of an application, we may need to collect traces for the whole application, which can take more time. In other cases the secret key may only be temporarily valid, meaning we can only obtain traces during a small time window. The limited number of measurements can also be a consequence of implemented countermeasures against SCA.

1.2 Proposed solution

To solve this problem, we may be able to make use of Generative Adversarial Networks (GANs). Using these few insufficient profiling traces, we might be able train a Conditional Generative Adversarial Network which we can use to expand our data set, such that the expanded data set is sufficient to execute a successful attack. One solution to increase the dataset size is data augmentation, a technique usually applied as a regularization of the model. Data augmentation generates synthetic measurements by applying modifications to the original dataset. The original dataset represents an approximation distribution from a true and unknown distribution. By augmenting the training set, one expects the approximation distribution to be a better representation of the real distribution. In this research, we will investigate how effective this method is at reducing the amount of real measurement required to make our attack successful, thus reducing the impact of these issues.

1.3 Contributions

Source Code The source code used in our experiments will be published online and available. Anyone can use the source code to train a model and use it to improve the performance of SCA. View source on GitHub

Reproducability As the source code is publicly available, the experiments in this paper can easily be reproduced by anyone.

Effect of cGAN hyperparameters In our experiments we will research the effect of hyperparameters on the effectiveness of Conditional GANs for SCA. The goal is to improve understanding of why certain cGAN models perform better or worse than others in the context of SCA.

1.4 Thesis overview

This research was part of a bachelor thesis at the Leiden Institute of Advanced Computer Science (LIACS), supervised by Guilherme Perin.

This thesis is organized as follows: in Section 2, we state some definitions and explain the data set used for our experiments, Generative Adversarial Networks, Conditional Generative Adversarial Networks, the evaluation and the algorithm used. In Section 3, we discuss some related research and their effects on our research. In Section 4, we first explain the methodology of our experiments, followed by establishing a baseline for our experiments, and finally we discuss the experiments themselves.

2 Background

In this section, we will explain some background information that will be used throughout this paper.

2.1 AES

AES (Advanced Encryption Standard) is a symmetric encryption algorithm widely used to secure sensitive data. The side-channel analysis application of this work targets an AES 128-bit algorithm, which applies an 128-bit key. This key, which we will try to recover, consists of 16 bytes. AES 128-bit algorithm consists of ten rounds for encryption or decryption, and in each round the following operations may appear:

- AddRoundKey: this operation XORs an intermediate AES 128-bit state with the corresponding round key.
- SubBytes: each byte of the input block, given by an AES 128-bit state, is substituted with a corresponding byte from the AES S-box, which is a predefined lookup table.

- ShiftRows: the bytes in each row of the block are cyclically shifted to the left. The first row remains unchanged, the second row is shifted by one position, the third row by two positions, and the fourth row by three positions.
- MixColumns: the columns of the block are treated as polynomials and multiplied with a fixed matrix using a special multiplication called Galois field multiplication.

Figure 1 illustrates the sequence of execution rounds for the encryption operation. In each round, a round key is considered and all round keys are generated by a the key expansion algorithm.

Of the 16 AES key bytes, we will focus on recovering only the first one, as the first two key bytes are unprotected in the ASCAD dataset. We will be focusing on the *SubBytes* operation of the AES encryption algorithm. The *SubBytes* step transforms every byte to another byte using an *sbox*, which is simply a mapping from one byte to another, where both bytes are never the same and are never a compliment of each other. If we can predict the *SubBytes* output operation, we are able to predict the key, as the plaintext is assumed to be known.

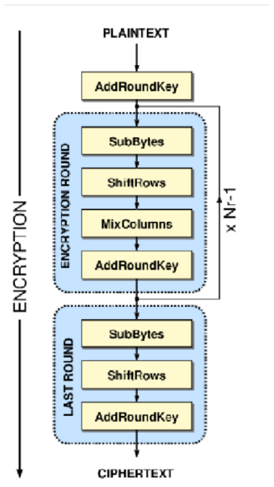

Figure 1: AES encryption overview. Available from: https://www.researchgate.net/figure/Architecture-of-AES-Algorithm_fig3_301644181. Accessed 18/07/2023

2.2 ASCAD Data set

The data set we will use is the ASCAD dataset. This dataset is intended to be used similarly to the MNIST, and is introduced first in [PSB+18]. The dataset has 200.000 profiling traces and 100.000 attack traces. These traces have been measured on an ATMega8515 chip while running a first-order masked AES 128-bit encryption. We will use the following amount of those traces:

- Profiling: 200.000 traces (from the profiling set)
- Validation: 20.000 traces (from the attack set)
- Attack: 20.000 (from the attack set)

Each side-channel measurement in this dataset contains 250 000 sample points. This interval represents the side-channel leakages from the entire first AES encryption round computation over the 128-bit key (i.e., AddRoundKey and SubBytes operation). In this work, we focus on a specific interval with 1,000 measurement points belonging to the first SubBytes operation (measurement points from 49,900 to 50,900), and corresponding metadata: a key and a plaintext, both 16 bytes.

2.3 Generative Adversarial Networks (GAN)

Generative Adversarial Networks (GANs) were first introduced by Goodfellow et al. in 2014 [GPAM+14] and have been largely adopted as a generative model for data generation such as images, video and audio. A generative adversarial network is a neural network architecture which consists of 2 sub-models, a generator model and a discriminator model. These models work as adversaries in a zero-sum game, with the generator trying to maximize the loss and the discriminator trying to minimize the loss. Equation 1 represents the loss function of a GAN model.

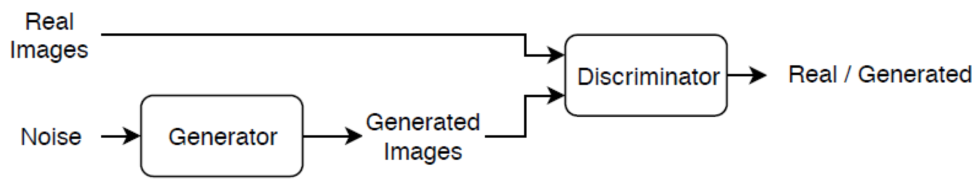

Figure 2: GAN architecture. Available from: https://nl.mathworks.com/help/deeplearning/ug/train-conditional-generative-adversarial-network.html. Accessed 18/07/2023.

The generator's task is to learn the real data distribution and generate fake data using random noise as z input resembling the real data as close as possible. The discriminator's task is to determine whether a given sample is real (labeled as '1') or fake (labeled as '0'). The GAN loss function is given by the following equation:

$$\min_{G} \max_{D} V(G, D) = E_x[log(D(x))] + E_z[log(1 - D(G(z)))] \tag{1}$$

where $D(x)$ is a value between 0 and 1 representing the probability of the trace being real, with x being a trace from the real data set. A value of 0.5 indicates the trace could just as likely be real as fake, meaning our discriminator is unsure. $G(z)$ represents a synthetic trace generated by the generator, and $D(G(z))$ in turn represents how sure the discriminator is of the trace's authenticity.

As can be seen from Equation 1, the discriminator's goal is to maximize $D(x)$ (real data, so classify as '1') and minimize $D(G(z))$ (fake data, so classify as '0'). The generator tries to maximize $D(G(z))$, and has no influence on $D(x)$ to prevent over-fitting.

2.4 Conditional Generative Adversarial Networks (cGAN)

In our case, using GANs is not sufficient as there is no way to specify the class (i.e., the label) of the data we want to generate. Thus, we use Conditonal Generative Adversarial Networks (cGAN) [MO14]. A cGAN is a GAN which can be conditioned using class labels. Other than this, a cGAN works much like a GAN. In this paper, the labels represent an intermediate byte (between 0 and 255) being processed by the AES encryption algorithm for which the trace is generated. More specifically, this byte is the first output byte of the SubBytes output in the first encryption round. Figure 3 shows the structure of the CGAN model that is considered in this work.

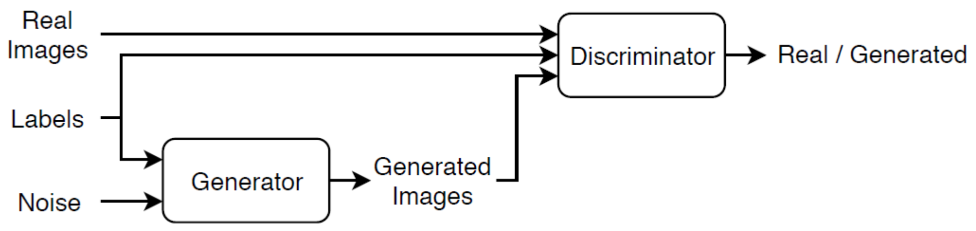

Figure 3: cGAN architecture. Available from: https://nl.mathworks.com/help/deeplearning/ug/train-conditional-generative-adversarial-network.html. Accessed 18/07/2023.

2.5 Multilayer Perceptron (MLP)

As mentioned in the previous sections, the CGAN architecture requires two separate neural networks models: a generator and a discriminator. In this work, we consider Multilayer Perceptrons (MLPs) as the neural network model architecture for both generator and discriminator. Moreover, in the profiling attack part, a deep neural network is also considered as the profiling model. Thus, we also adopt MLP models for this task.

A MLP is a fully-connected feed-forward neural network algorithm, which consists of an input layer, one or more hidden layers, and an output layer. Every layer consists of multiple neurons and may implement a non-linear function with an activation function. Every node in a layer i is connected to all nodes of the next layer j with a certain weight w_{ij}, hence the name *fully-connected*.

The attack model, which performs the final side-channel analysis using traces generated by the cGAN model, will be an MLP. However, we will not perform any experiments using variations of this MLP and will be using the same model as our attack model every time, since hyperparameter tuning for the attack model is not the focus of this research. This is explained further in Section 3.3.

The generator and discriminator will both also be MLPs. We've decided against also experimenting using CNNs. CNNs are in general more complex, difficult to train and have a larger amount of hyperparameters to tune, and as such there was not enough time available to take CNNs into consideration too. Moreover, CNNs are known to be shift-invariant and more robust to trace desynchronizations in SCA context [CDP17]. In this work, we only consider side-channel measurements that are synchronized in the time domain.

2.6 Side-Channel Attacks (SCA)

Side-channel attacks mainly consist of two categories: *profiling attacks* and *non-profiling attacks*. Non-profiling attacks mean that we recover the secret key by analyzing the leakages of a target device directly. With profiling attacks, we are in the possession of an open identical copy of the target device, of which we know the secret key beforehand. We use this copy to construct a model (in our case, a deep learning model), which we later use to attack our target device. In this paper we will only be investigating profiling side-channel attacks, as we want to investigate the impact using a cGAN to generate data to train a MLP has on the effectiveness of our attack, thus a non-profiling attack would make no sense.

2.7 Evaluation Metric

To compare different cGAN models, we need a way to measure the effectiveness of our attack. To do this, we will use *guessing entropy*, a proven SCA evaluation metric [SMY09]. Giving a list of $\mathcal{K}$ key candidates, the guessing entropy measures the position of the correct key among all $k \in \mathcal{K}$.

Let N_k be the number of possible keys/labels, and N_a be the size of the attack set. In our case, as we target only the first key byte, the number of possible keys N_k is 2^8. When a trained deep neural network is provided with a set of N_a attack traces, the output layer provides a two-dimensional matrix P containing the predicted probability for each class for each trace. The matrix P has dimensions $\{N_a \times N_k\}$. Each element $p_{i,y}$ in P is the probability that the trace i is represented by the class y. The class y, as explained in previous sections, is derived from the leakage model. In our case, traces are labeled according to the intermediate byte obtained from the first SubBytes output in the first AES encryption round. Thus, the label is defined as $y = S(k \oplus d_i)$, in which S indicates the substitution operation from SubBytes and d_i is the plaintext byte from the i-th attack trace. Note how the label associated to the trace i differs for different key candidates.

For each key byte candidate, we compute a cumulative sum of probabilities:

$$S(k) = \sum_{i=0}^{N_a-1} \log p_{i,y} \tag{2}$$

where the label y is defined according to the key byte candidate. After computing $S(k)$ for all key candidates, we sort their values by order of magnitude, which gives us a vector with sorted key probabilities. The key rank is the position of the correct key candidate inside the sorted vector. In other words, to obtain the rank r of the correct key c for given trace t with prediction vector $p = P_t$, with length N_k, define $p_{sorted} = p \mid p_i >= p_j$. The rank r of the correct key c is r where $p_{sorted,r} = p_c$.

To calculate the guessing entropy G, we perform this ranking process multiple times on a randomly selected subset of predictions and average the key rank for every trace. As such, guessing entropy is the average of multiple key rank results.

The metric that is considered in this work is the evolution of guessing entropy with respect to the number of attack traces, which is very useful for showing the efficiency of our attacks. This way, we compute guessing entropy for different number of attack traces, ranging from 1 to N_a. The faster our guessing entropy stabilizes at 1, meaning the correct key was predicted as most likely, the more efficient our attack is. An unsuccessful attack will stabilize at around $N_k/2$, indicating our model is not generalizing correctly.

3 Related Work

In this section, we review recent publication relating side-channel analysis and conditional generative adversarial networks.

3.1 GANs in SCA

Generative models have seen limited usage in side-channel analysis, primarily focused on specific applications. In [WCL+20], researchers explored the application of generative adversarial networks (GANs) for data augmentation purposes. Authors from [MBPK22], utilized conditional generative adversarial networks (cGANs) for data augmentation. Both analyses specifically targeted protected AES implementations.
Authors of [ZBC+23] employed Variational Auto Encoders (VAEs) to generate reconstructed and synthetic traces, effectively modeling the true conditional probability distribution of real side-channel traces. Another approach [CZG+22] proposed a GAN-based structure to address issues related to the transferability of profiling models.

3.2 ASCAD data set

The data set used in this research, as previously mentioned in Section 2.2, was introduced as a way to benchmark new Deep-Learning Side-Channel Analyses (DL-SCA) models against other state of the art models, by providing a standard data set [PSB+18]. This data set can be found on GitHub. Later, in several publications, the ASCAD dataset became the main dataset for benchmarking in research. An updated overview of deep learning-based side-channel analysis is provided in [PPM+23], in which most of the mentioned methods and papers considered ASCAD data set as the target device for proof-of-concepts.

3.3 Attack Model

The primary purpose of this research is to determine the effectiveness of using cGANs for side-channel analyses and to determine the effect of hyperparameters of these cGANs on the effectiveness of side-channel analyses. However, we cannot execute an attack using a cGAN model without implementing an attack model, as the cGAN will only generate fake data, which needs to be fed into another deep learning model, which will execute the attack.

As this is not the purpose of this research, we have copied a well-researched MLP which has proven to be effective at deep-learning based SCA [PSB+18]. In this paper, CNNs were also found to be very effective at deep-learning based SCA, even more so than standard MLPs. However, we have decided to implement an MLP model instead of a CNN model, as CNNs are in general more difficult to train and more complex. We deemed the performance difference not significant enough, and since all our experiments use the same attack model, this should not affect our results. We will use the model deemed as the best in this paper, which was named "MLP_{best}".

This model also provides us with a perfect benchmark, as our only addition to this model is the cGAN model used to generate samples. Thus, we can compare the performance of analyses with generated samples to the performance of analyses performed with only MLP_{best}.

4 Experiments

The algorithm for which we are trying to uncover secret information is AES 128-bit. In the ASCAD data set we use, the first two key bytes are unprotected, while the rest of the key bytes are protected with masking. We want to target an unprotected scenario, and as such we only target the first key byte.

The main purpose of our experiments, is researching the effects the various hyperparameters of cGANs have on the effectiveness of augmenting the profiling set to train deep-learning models to perform side-channel analysis. To do so, we try to minimize the variations in results that can possibly be caused by our deep-learning model, while also trying to maximize the performance of said deep-learning model, to avoid drawing incorrect conclusions about the effectiveness of the analyses (if we use an under powered deep-learning model, we might draw the conclusion that cGANs are not useful in the context of DL-SCA).

We will be executing several different experiments to determine the effect various hyperparameters have on the effectiveness of using Conditional GANs for side-channel analysis. We will also be investigating the effects of certain variations in the training of cGAN models, such as differing number of epochs, batch sizes and data set sizes. As mentioned in Section 3.3, the MLP_{best} model will provide us with a solid bench marking foundation, against which we can compare our models. We will not reuse the results from [PSB+18], but instead generate our own results, to prevent any small inconsistencies in methodology to interfere with our findings. We will then have a set of results of our attack model without the use of data augmentation using a CGAN model, against which we can compare our other findings.

4.1 Methodology

We will train a cGAN model on the data set defined in Section 2.2. We will first define Equation 3, which represents the parameters used in training our models, where b is the batch size, e is the number of epochs and t is the training set size. In all but one experiment, we will use a training process defined as $T(400, 10, 200.000)$. In the remaining experiment we will investigate the impact these parameters have on the effectiveness of our model.

$$T(b, e, t) \tag{3}$$

We will train the model for e epochs, with a batch size of b, meaning t/b batches per epoch. For every batch, we first train our discriminator on a batch of real samples, without affecting and updating our generator. Next, we generate b synthetic samples using our generator, on which we then train our discriminator, still without affecting and updating our generator. Finally, we generate b random latent points, on which we train the entire cGAN model, which only affects and updates our generator. After every epoch, we calculate the *Signal-To-Noise Ratio* with 10.000 randomly selected real traces and 10.000 generated fake traces, which will be shown in the results of some experiments.

When training is finished, we discard our discriminator and enter the attack phase. During the attack phase, we use our generator to generate synthetic traces. We generate 100 traces for a random possible label (1 byte $\rightarrow$ 8 bits $\rightarrow 2^8$, so between 0 and 255), find the mean of these 100 traces, and do this 10.000 times, resulting in 10.000 means, which we will use as our synthetic traces. Using these synthetic traces, we train our MLP model (see Section 4.2.1) for 200 epochs, with a batch size of 100. We use our MLP model to make predictions about our attack data set (see Section 2.2).

Finally, we use our predictions to calculate the *Guessing Entropy*, which indicates the rank of the correct key over a limited number of attack traces. If our *Guessing Entropy* stabilizes at GE = 1, we can conclude the attack has been successful and using a cGAN model has been useful. The earlier the *Guessing Entropy* stabilizes as GE = 1, the less traces were required to get there. For example, if we reach a *Guessing Entropy* of 1 after 500 traces, we would have only needed to collect 500 traces from our attack set, even though in the experiments we use 20.000 traces to perform the attack. Obviously, the less traces required the better, as this saves significant time and effort while performing side-channel analyses.

The predictions consist of a matrix P with dimensions 20.000 x 256 (N_a = 20.000 traces, 256 possible keys for every trace), where every key is given a probability of being the correct key. To calculate the *Guessing Entropy*, we randomly select 5.000 traces out of 20.000, sort the predictions of every trace in decreasing order and for every 10_{th} trace find the index of the correct key, which is the key rank. We do this 40 times, adding up all key ranks and finally dividing by 40, which results in the average key rank over time. This averaged key rank is essentially the guessing entropy. This random selection and repetition is done to improve the statistical significance of our findings.

The results of some executions within an experiment were pretty similar, due to the fact that we perform the calculation for every 10_{th} trace. As such, certain executions might both yield a result of 50 traces, while in reality there might still be a small difference. For the parameters with the most promising results, where such a situation applies, we might run another experiment where we calculate the *Guessing Entropy* for every single trace, while also doing the entire execution 500 times instead of 40, to further improve the significance of our findings.

4.2 Model Architecture

To keep the experiments clear and organized, we will be defining our models as equations. We will do so for both the generator and discriminator, except for the MLP. The MLP will be kept the same throughout all experiments, as we want to investigate only the generator and discriminator. Let l be the number of layers, d be the percentage of dropout between every non-input/output dense layer (only for the discriminator), n the number of nodes for every layer, and a be the activation function. The discriminator and generator models can then defined using Equations 4 and 5. For the generator, there are no dropout layers.

$$D(l, d, n, a) \tag{4}$$

$$G(l, n, a) \tag{5}$$

In the models shown below (Sections 4.2.2 and 4.2.3), we have excluded the input layers from this definition. These input layers introduce some variation in the labels, which prevent over-fitting in our models.

Finally, we can define our cGAN. Let d be a discriminator, g be a generator, o be an optimizer, and l be a learning rate for the optimizer o. We then define our cGAN as seen in Equation 6. Loss values will be calculated using Binary Cross-Entropy.

$$N(d, g, o, l) \tag{6}$$

4.2.1 Attack Model

As mentioned in Section 3.3, this model was taken from [PSB+18], and was already tailored for ASCAD dataset. This model will stay the same throughout all experiments, as the main focus of our experiments is on the cGAN models. Thus, we want this model to influence our results as little as possible. This model is an MLP, trained for 200 epochs with a batch size of 100 and an *RMSprop* optimizer with a learning rate of 0.00001. The MLP will be trained on data generated by the generator. Essentially, this MLP is used as a supervised model, in which the traces produced by the generator are used for training and the attack and validation sets are taken from the original ASCAD data set. The MLP model is detailed below:

Layer Type	Output Shape	Parameter #
Input	(None, 1,000)	0
Dense	(None, 200)	200,200
Dense	(None, 200)	40,200
Dense	(None, 200)	40,200
Dense	(None, 200)	40,200
Dense	(None, 200)	40,200
Dense	(None, 200)	40,200
Dense (Output)	(None, 256)	51,456

Total Parameters: 452,656
Trainable Parameters: 452,656
Non-trainable Parameters: 0

To obtain results against which we can benchmark our experiments, we will perform a side-channel analysis using this model without performing any data augmentation using CGAN models, but with only our actual attack data set. In the other experiments, our *Attack Model* is trained using 10.000 synthetic traces (as explained in Section 4.1) generated by our cGAN model. To obtain our benchmark without data augmentation, we will instead use 10.000 traces directly from the profiling traces from the ASCAD data set. The guessing entropy results without data augmentation with our MLP model can be seen in Figure 4.

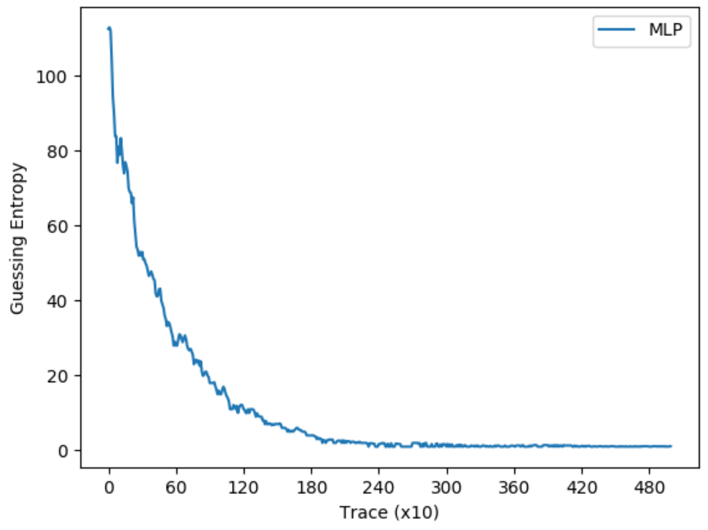

Figure 4: Guessing Entropy without data augmentation

4.2.2 Discriminator

The base discriminator model is defined as follows. It is a simple *Fully-Connected Neural Network*. The first 4 layers are not variable and will be the same for every single model. These 4 layers allow for a label as input, and perform so called *Label Smoothing* ($0 \rightarrow 0.1$ and $1 \rightarrow 0.9$, for example), which has been shown to improve the stability of GANs in [SGZ+16]. These layers are concatenated with our data input, which has 1,000 nodes, as our traces each have 1,000 measurement points (Section 2.2). Dropout layers are added between every dense layer to prevent over-fitting. The architecture of the discriminator is described below:

Layer Type	Output Shape	Parameter #
Input	(None, 1)	0
Embedding	(None, 1, 256)	65,536
Dense	(None, 1, 200)	51,400
Flatten (x)	(None, 200)	0
Input (y)	(None, 1,000)	0
Concatenate (x & y)	(None, 1,200)	0
Dense (ELU)	(None, 250)	300,250
Dropout (30%)	(None, 250)	0
Dense (ELU)	(None, 250)	62,750
Dropout (30%)	(None, 250)	0
Dense (ELU)	(None, 250)	62,750
Dropout (30%)	(None, 250)	0
Dense (ELU)	(None, 250)	62,750
Dense (Linear)	(None, 1)	401
Total Parameters: 1,078,937		
Trainable Parameters: 1,078,937		
Non-trainable Parameters: 0		

Using Equation 4, this model would be defined as $D(4, 30\%, 250, ELU)$.

4.2.3 Generator

The base generator model is defined as shown below. Once again, it's a *Fully Connected Neural Network*. The first 4 layers are not variable and will be the same for every single model. These 4 layers allow for a label as input, and perform so called *Label Smoothing* ($0 \rightarrow 0.1$ and $1 \rightarrow 0.9$, for example), which has been shown to improve the stability of GANs in [SGZ+16]. These layers are concatenated with our data input, which has 1,000 nodes, as our traces each have 1,000 measurement points (Section 2.2). Our generator does not have dropout layers, as the generator needs to learn the data distribution as close as possible to generate believable synthetic traces.

Layer Type	Output Shape	Parameter #
Input	(None, 1)	0
Embedding	(None, 1, 256)	65,536
Dense	(None, 1, 400)	102,800
Flatten (x)	(None, 400)	0
Input (y)	(None, 1,000)	0
Concatenate (x & y)	(None, 1,400)	0
Dense (ELU)	(None, 160)	224,160
Dense (ELU)	(None, 160)	25,760
Dense (ELU)	(None, 160)	25,760
Dense (ELU)	(None, 160)	25,760
Dense (ELU)	(None, 160)	25,760
Dense (ELU)	(None, 160)	25,760
Dense (Sigmoid)	(None, 1,000)	161,000

Total Parameters: 682,296
Trainable Parameters: 682,296
Non-trainable Parameters: 0

Using Equation 5, this model would be defined as $G(6, 160, ELU)$.

4.2.4 GAN

The models discussed in Section 4.2.2 (D) and 4.2.3 (G) can be combined into a cGAN model $N(D, G, Adam, 0.0002)$. The *Adam optimizer* will have a β_1 value of 0.5, a β_2 value of 0.999 and a ϵ value of 0.0000001; these will not be changed in the experiments, except for in the network optimizer experiment, where an SGD optimizer will be tested, which does not have these parameters.

4.3 Exploring the Impact of Hyperparameters

In this section, we explore the impact of different hyperparameters on generator and discriminator architectures.

4.3.1 Layers

In this experiment, we will try to understand the effect that the number of layers has on the performance of our model. A bigger model (meaning more layers) does not always mean a better model. Too many layers and nodes can also an increase in computational power required and potentially overfitting to the training set. In our case, we fill focus on the impact on the effectiveness of our model and leave training and attack duration out of consideration.

To investigate the effect of layers on attack performance, we will investigate combinations of various generators and discriminators. We will define various generator and discriminator models, differing only in the number of layers. As such, we will define our discriminators as $D(l_d, 30\%, 250, ELU)$ with $l_d \in [2, 4, 6, 8]$. Our generators will be defined as $G(l_g, 160, ELU)$ with $l_g \in [2, 4, 6, 8, 10]$. This results in $4 \cdot 5 = 20$ unique cGAN models. Plotting all these results in one graph would affect readability, so plots contain one graph per l_d, where each line represents a different value for l_g. Figure 5 shows the results of this experiment.
A general trend can be seen where more layers equals a better performing model, while the generator should have more layers than the discriminator to ensure best performance (with some exceptions). The results of models with discriminators $l_d \in [4, 6, 8]$ are all very similar and show this trend well, with more powerful generators performing about the same. A generator with 6 layers seems to perform relatively similar to a generator with 10 layers. To reduce training cost and duration, 6 layers would be the better choice. Same goes for the discriminator, where there is little difference between choosing a discriminator with 4, 6 or 8 layers. As such, the best combination would be $l_g = 6$ and $l_d = 4$.

The results for the models with a discriminator with 2 layers do not follow the previously mentioned trend, but rather the opposite. When excluding $l_g = 10$, more generators causes worse performance. A discriminator is responsible for providing a learning gradient to the generator. With only 2 layers, the discriminator may not be powerful enough to provide a useful gradient and keep up with the more powerful generators. The model with a generator with 10 layers seems to be an anomaly. A big difference in layers between the generator and discriminator might also be a viable way to train the models. However, we do not have enough data to make such a conclusion at this moment.

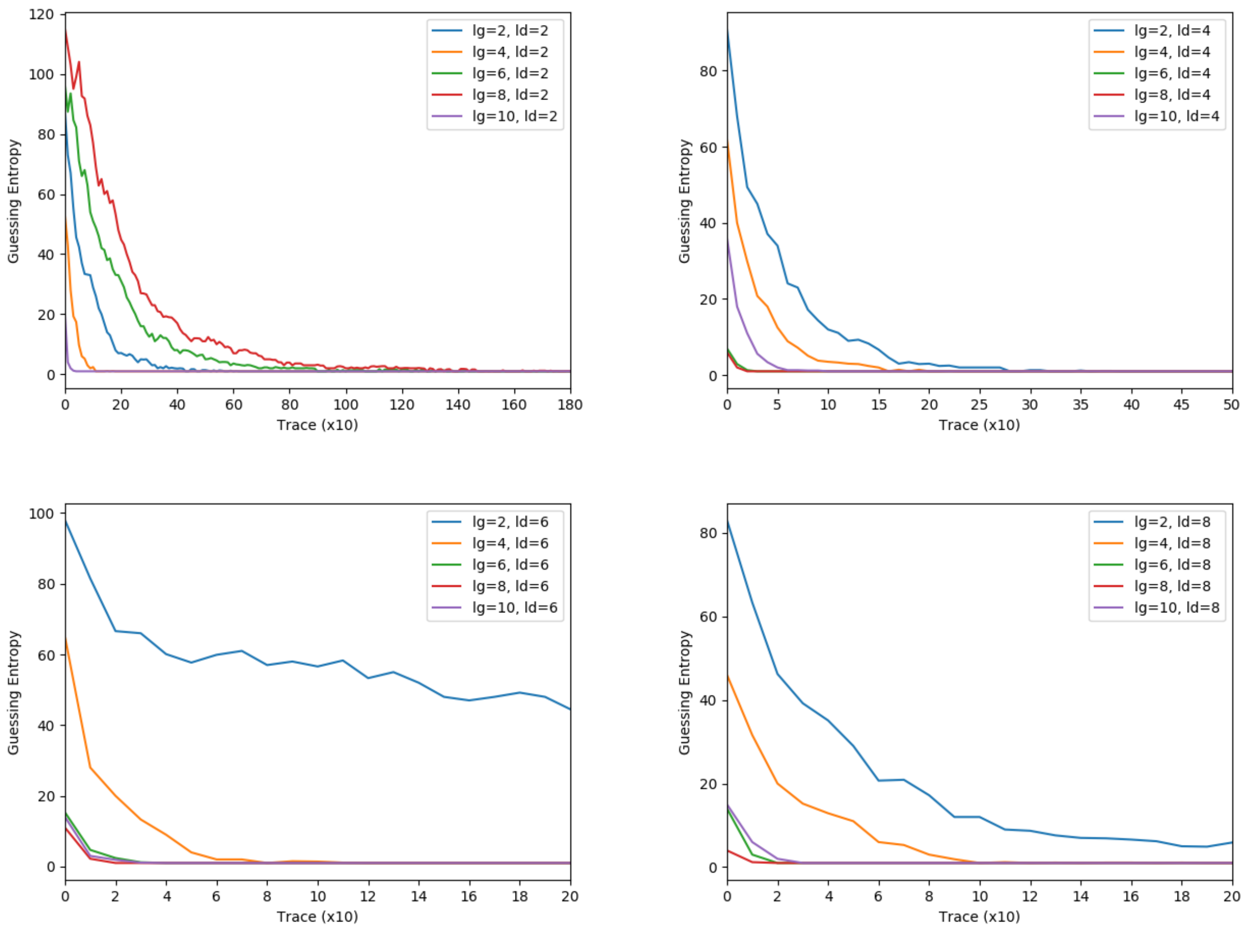

Figure 5: Guessing Entropy for varying number of layers

4.3.2 Nodes

In the case of the number of nodes, the idea is the same as with the number of layers. Too many can cause over fitting, too few can cause our model not to converge. For this experiment, our discriminators will be defined as $D(4, 30\%, n_d, ELU)$ with $n_d \in [100, 250, 400, 800]$. Our generators will be defined as $G(6, n_g, ELU)$ with $n_g \in [100, 160, 200, 400, 800]$. This results in $4 \cdot 5 = 20$ unique cGAN models. Once again we have split up the graphs, where we plotted one graph per value of n_d with the various lines representing a value of n_g.

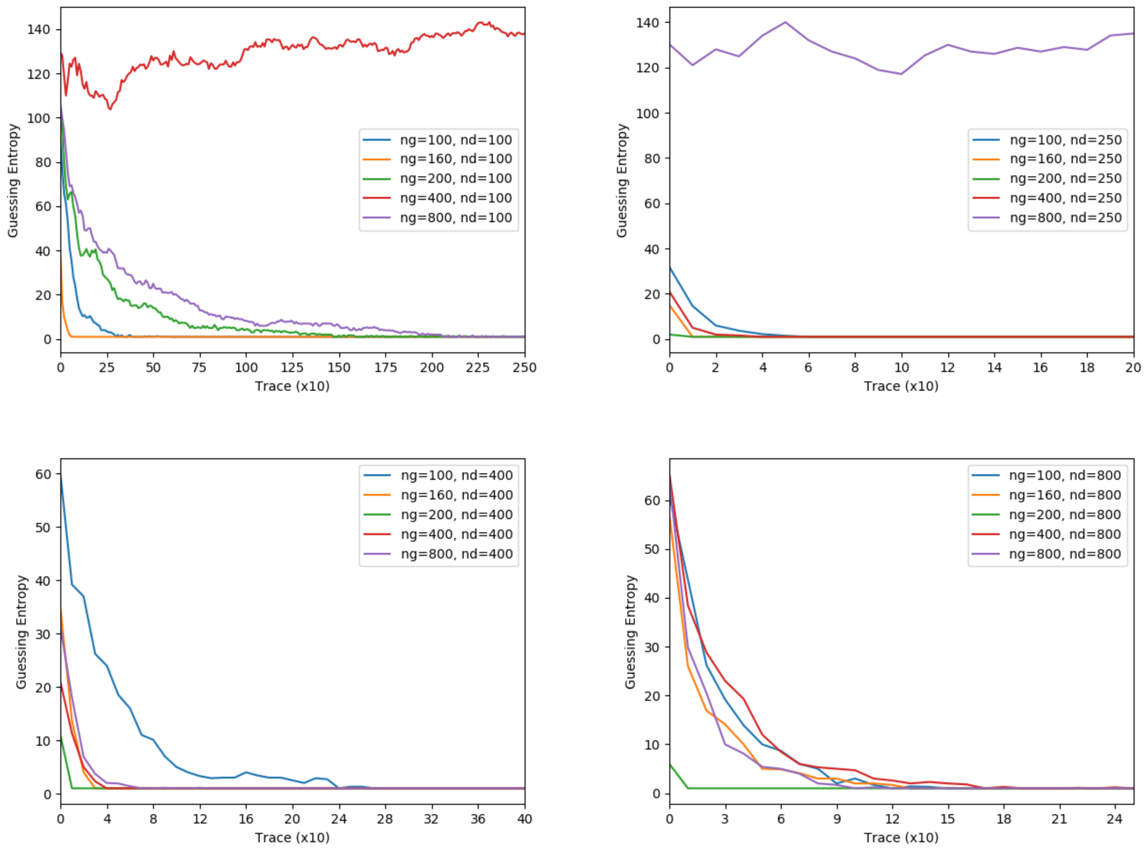

Figure 6: Guessing Entropy for varying number of nodes per layer

Figure 6 shows the results of this experiment. The 4 most efficient combinations ($n_g = 160$ and $n_g = 200$ with $n_d = 250$, $n_g = 200$ with $n_d = 400$ and $n_g = 200$ with $n_d = 800$) all converge to a *Guessing Entropy* of 1 in 10 traces. In general, a generator with $n_g = 200$ seems to have the best performance. This shows that more nodes does not necessarily equal a better performing model. However, as was the case in Section 4.3.2 with a discriminator with a small number of layers (2), here $n_g = 200$ does not have the best performance when the discriminator has a small number of nodes ($n_d = 100$). The best performing combination is $n_g = 200$ with $n_d = 250$, and increasing either n_g or n_d does not seem to improve attack performance in any way. To definitively say this, further experiments should be ran to investigate whether this is the case with $n_g > 800$ and $n_d > 800$.

4.3.3 Dropout

The dropout layer was first introduced by Srivastava et al. in 2014 [SHK+14]. The key idea of the dropout technique is to randomly disable certain nodes of the network during training. The idea is that this forces surrounding nodes to pick up the leftover workload, meaning the resulting predictions are never dependent on specific nodes. As explained in Section 4.2.3, we do not use dropout layers for our generator as our generator needs to learn the data distribution as close as possible to generate synthetic data, and as such over fitting is by definition not possible. However, the cGAN itself can over fit, and to solve this we use dropout layers in our discriminator. In this experiment, we will investigate the effect this hyperparameter has on our attack performance. We will once again do so using modified versions of our base models, D_{base} and G_{base}. In this experiment, the discriminators we will be using are $D(4, d, 250, ELU)$, with $d \in [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 0.9]$. For our generator, we will be using $G(6, 160, ELU)$.

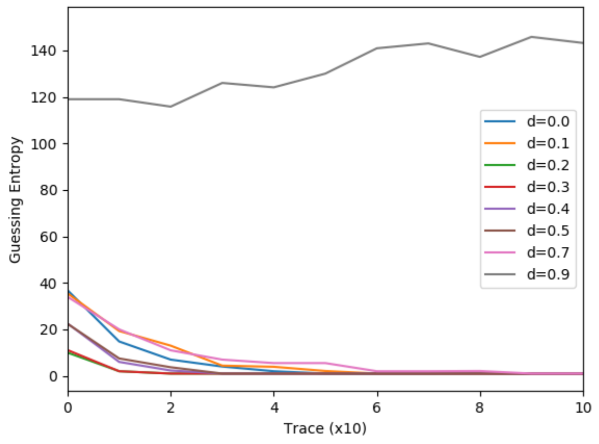

Figure 7: Guessing Entropy for varying dropout percentages

A dropout percentage of 20% or 30% seems to be the sweet spot, both converging after 20 traces, although there is little difference between all but the 90% parameter. A dropout that is too large (90%) seems to hinder the discriminator too much. This can cause the discriminator to fail in providing a learning gradient to the generator, causing the generator not to learn anything useful, which in turn causes our model to fail.

However, as can be seen by the relatively high performance of the model without any dropout applied, over-fitting does not seem to be too much of an issue. To test this, we will run another experiment where we also use varying numbers of layers and nodes. We will not use the combinations found as "best" in Section 4.3.1 and Section 4.3.2, but rather combinations which performed well but are as far opposite from each other as possible. This should give us some insight into the effect of dropout, and should show us whether over fitting is really not a problem, or only not a problem for the specific number of layers and nodes previously used. Our models will be defined as follows, with $d \in [0.0, 0.3]$, resulting in 8 models:

- $CGAN_1(d) = N(D(10, d, 100, ELU), G(2, 160, ELU), Adam, 0.0002)$

- $CGAN_2(d) = N(D(10, d, 800, ELU), G(2, 200, ELU), Adam, 0.0002)$
- $CGAN_3(d) = N(D(8, d, 100, ELU), G(8, 160, ELU), Adam, 0.0002)$
- $CGAN_4(d) = N(D(8, d, 800, ELU), G(8, 200, ELU), Adam, 0.0002)$

The results of these experiments can be seen in Figure 8, where both images show the same graph with a different limit on the x-axis. In this graph, we can clearly see that pairs are formed where both versions of every model sit pretty close to each other, one with dropout and one without. 3 pairs ($CGAN_1$, $CGAN_3$, and $CGAN_4$) converge to a *Guessing Entropy* of 1, while $CGAN_2$ does not converge within 5,000 traces. For all models but $CGAN_1$, using $d = 0.3$ improves performance over using $d = 0.0$. $CGAN_1$ has the smallest discriminator of all models, which can explain why this specific model can not take advantage of using dropout.

Using these results we can show that our previous conclusion — over-fitting is not a problem, but dropout can still be used for marginal performance improvements — is only partially correct. Model $CGAN_2$ shows this nicely, where $d = 0.0$ shows *Guessing Entropy* increasing, a clear sign of an unstable network, while *Guessing Entropy* decreases after applying $d = 0.3$.

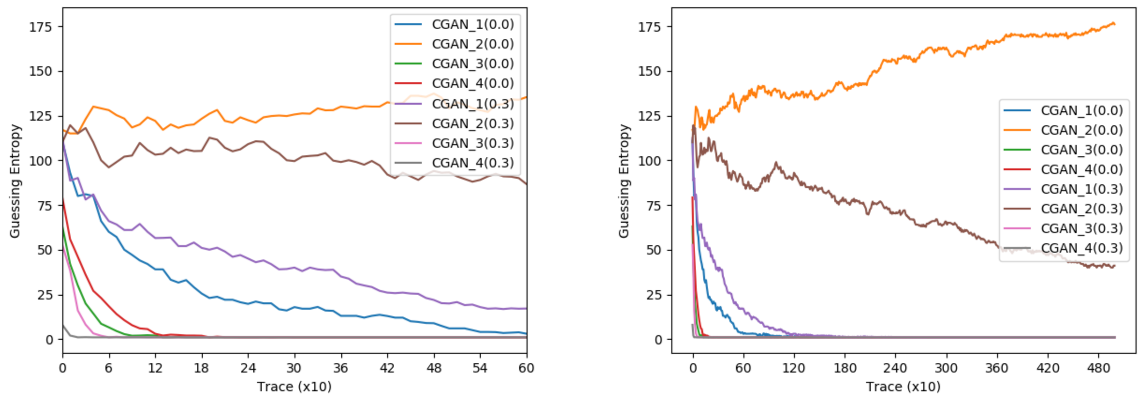

Figure 8: Guessing Entropy for varying numbers of layers, nodes, and dropout

4.3.4 Activation Functions

In this experiment, we will investigate the effects of activation functions on the generator and discriminator. We will define our discriminator as $D(4, 30\%, 250, a_d)$ with $a_d \in [ELU, ReLU, Leaky-ReLU]$, and our generator as $G(6, 160, a_g)$ with $a_g \in [ELU, ReLU, Leaky - ReLU]$.

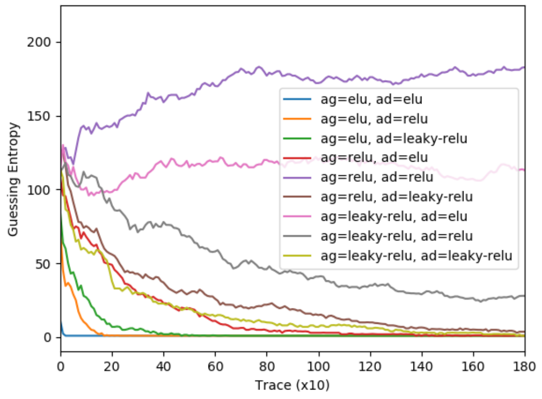

Figure 9: Guessing Entropy for various activation functions

We can see that the ELU activation function for both the generator and discriminator, converging after only 20 traces, is much more effective than any other combination, converging to a *Guessing Entropy* entropy of 1 after 170, 470 or 1000+ traces.

4.3.5 Network Optimizer

In this experiment, we investigate the effects of the network optimizer with various learning rates. We define our network as $N(D_{base}, G_{base}, o, l)$, with $o \in [Adam, SGD]$ and $l \in [0.00005, 0.0001, 0.0002, 0.0005]$, resulting in 8 GAN models differing only in the network optimizer used. As mentioned in Section 4.2.4, for the Adam optimizer a β_1 value of 0.5, a β_2 value of 0.999 and a ϵ value of 0.0000001 will be used. The SGD optimizer does not use these parameters. The results of this experiment can be seen in Figure 10.

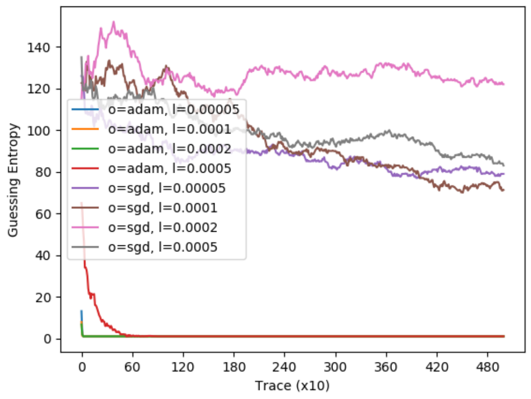

Figure 10: Guessing Entropy of Adam and SGD optimizers with various learning rates

We can immediately see that the SGD network optimizer does not work well for GANs, regardless of the learning rate used. This can be explained by the frequency of updates done by the SGD optimizer. It updates the models parameters quite frequently (once for every example). This can add to a models instability. GANs are notoriously unstable by themselves, and the SGD optimizer seems to magnify this effect. This can clearly be seen in Figure 11, where the generator loss with the Adam optimizer is much more stable than with the SGD optimizer.

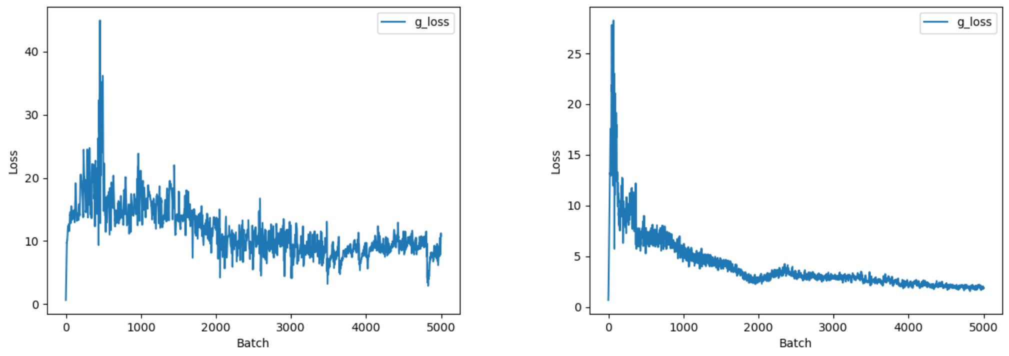

Figure 11: Generator Loss of SGD (left) and Adam (right) optimizers with a learning rate of 0.0002

The Adam optimizer however does work well. To better show the effect of the learning rate with the Adam optimizer, we have plotted the results again in Figure 12, but ignored the results from the SGD optimizer. The learning rate does not seem to affect our results too much, except for a learning rate of 0.00005, which does converge, but quite slow, after 510 traces.

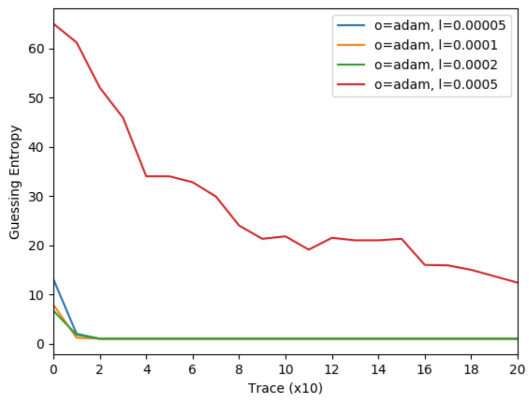

Figure 12: Guessing Entropy for Adam optimizer with various learning rates

4.3.6 Training

We also investigate the effect the various training parameters have on the effectiveness of our model. We use a CGAN model defined as $N(D(4, 250, 0.3, ELU), G(6, 160, ELU), Adam, 0.0002)$. The training process is defined using Equation 3, as $T(btc, ep, tr)$ with $btc \in [100, 200, 400, 600, 800]$, $ep \in [5, 10, 25]$ and $tr \in [50,000; 100,000; 200,000]$, resulting in 45 unique training processes. The results can be seen in Figure 13.

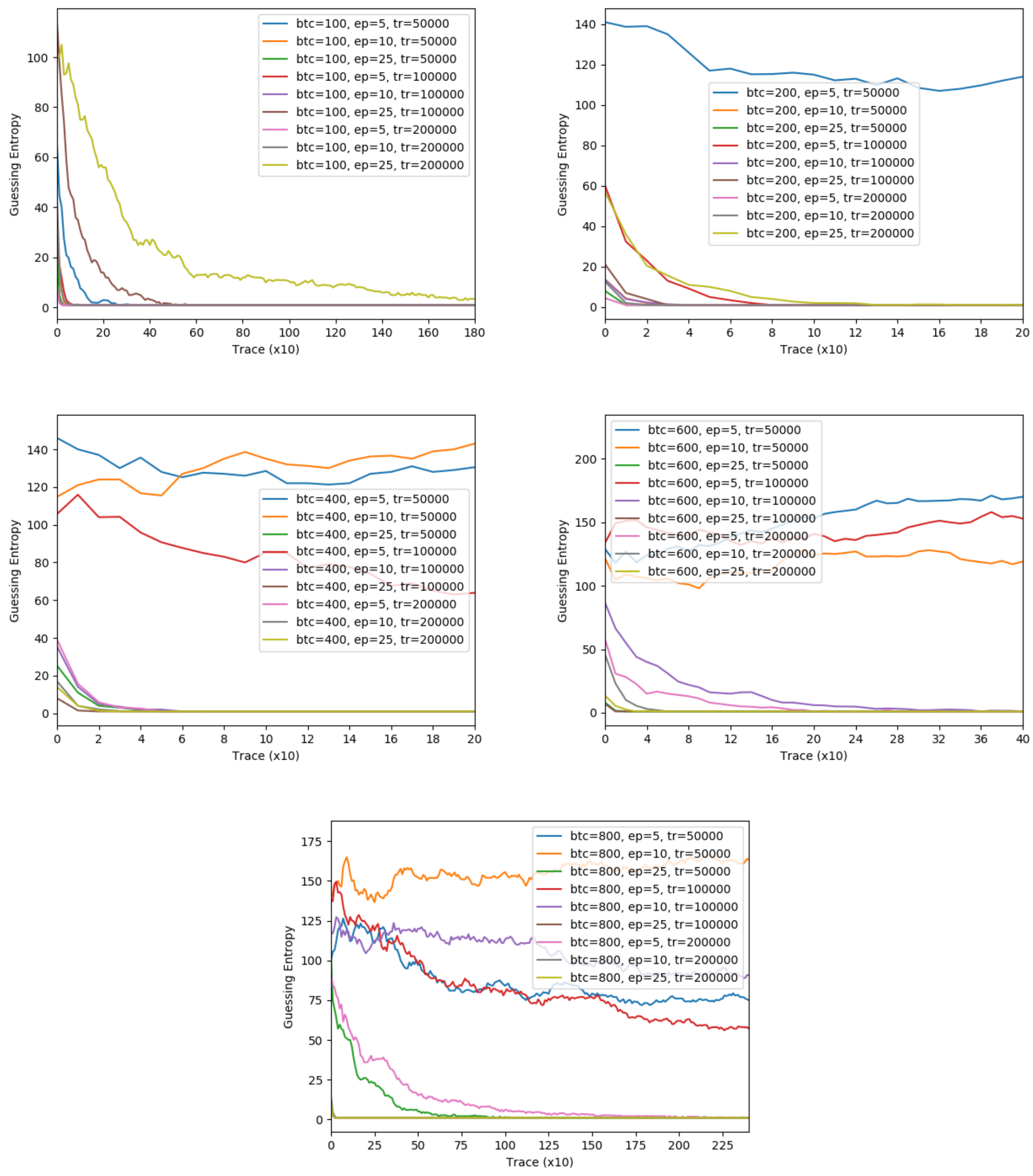

Figure 13: Guessing Entropy for various training settings

One clear conclusion that can be made, is that too high of a batch size is in general not effective. This seems to be counteracted somewhat by increasing the number of epochs and the training set size, as can be seen in the results for $b = 800$, where $tr = 200,000$ and $ep = 10$ or $ep = 25$ performs well. This would however be a waste of time and compute resources, as a higher number of epochs and a bigger training set cause training time to increase, while lowering the batch size does not.

Another trend we see, is that a small training set (50,000 or 100,000) coupled with a low number of epochs (5) does not perform well across any batch size. Simply increasing the number of epochs and training set also does not always work however. The model will simply not have learned enough to generate any meaningful samples.

5 Conclusions and Further Research

In all the experiments we have done, we have tested the effect various hyperparameters have on the effectiveness of using data augmentation with *Conditional Generative Adversarial Networks* to perform *Side-Channel Analysis*. It is hard to make any definitive conclusions about what works with certain proof. However, we have shown that in general there is some consistency in what works and what does not. Most experiments in this paper have used a set of hyperparameters, where only one or two hyperparameters were variable. To further improve our understanding of these hyperparameters, further research should be done combining all separate conclusions, and testing whether applying all the high-level ideas can take a model from unstable to stable and efficient.

In Section 4.2.1, we have shown a baseline for how well our attack model performs, without using data augmentation. Given the results from all our experiments, it is very clear that using a *Conditional Generative Adversarial Network* to perform data augmentation can improve the effectiveness of Deep-Learning based SCA. In Figure 4, the MLP does not achieve a *Guessing Entropy* of 1 after 5,000 traces, but gets very close at 1.125. Several models used in our experiments have shown convergence to a *Guessing Entropy* of 1 in less than 20 traces, which is a very significant improvement.

References

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004.

[CDP17] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Convolutional neural networks with data augmentation against jitter-based countermeasures - profiling attacks without pre-processing. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 45–68. Springer, 2017.

[CZG$^+$22] Pei Cao, Hongyi Zhang, Dawu Gu, Yan Lu, and Yidong Yuan. AL-PA: cross-device profiled side-channel attack using adversarial learning. In Rob Oshana, editor, *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*, pages 691–696. ACM, 2022.

[GPAM$^+$14] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pages 388–397, 1999.

[MBPK22] Naila Mukhtar, Lejla Batina, Stjepan Picek, and Yinan Kong. Fake it till you make it: Data augmentation using generative adversarial networks for all the crypto you need on small devices. In Steven D. Galbraith, editor, *Topics in Cryptology - CT-RSA 2022 - Cryptographers' Track at the RSA Conference 2022, Virtual Event, March 1-2, 2022, Proceedings*, volume 13161 of *Lecture Notes in Computer Science*, pages 297–321. Springer, 2022.

[MO14] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets, 2014.

[PPM$^+$23] Stjepan Picek, Guilherme Perin, Luca Mariot, Lichao Wu, and Lejla Batina. Sok: Deep learning-based physical side-channel analysis. *ACM Comput. Surv.*, 55(11):227:1–227:35, 2023.

[PSB$^+$18] Emmanuel Prouff, Rémi Strullu, Ryad Benadjila, Eleonora Cagli, and Cécile Canovas. Study of deep learning techniques for side-channel analysis and introduction to ascad database. *IACR Cryptol. ePrint Arch.*, 2018:53, 2018.

[SGZ$^+$16] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans, 2016.

[SHK$^+$14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[SMY09] François-Xavier Standaert, Tal G. Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, pages 443–461, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[WCL$^+$20] Ping Wang, Ping Chen, Zhimin Luo, Gaofeng Dong, Mengce Zheng, Nenghai Yu, and Honggang Hu. Enhancing the performance of practical profiling side-channel attacks using conditional generative adversarial networks. *CoRR*, abs/2007.05285, 2020.

[ZBC$^+$23] Gabriel Zaid, Lilian Bossuet, Mathieu Carbone, Amaury Habrard, and Alexandre Venelli. Conditional variational autoencoder based on stochastic attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(2):310–357, 2023.