IL1333 Hardware Security: PUF Project Analysis

Axel Månson Lokrantz, Mohammed Louai Alayoubi

2024-03-28

1 Introduction

The goal of the project is to perform a profiling side-channel attack on a 128-bit Arbiter PUF implemented on a Xilinx Artix-7 FPGA. Using power traces from FPGA boards a CNN (Convolutional Neural Network) is trained to learn the APUF (Arbiter PUF) binary responses of "0" or "1". The model is then tested on unseen test data, which the model has not encountered during training. The performance is assessed based on its accuracy in correctly classifying the binary classification problem. The project includes two different pre trained models PUF0, an unmodified PUF, and PUF12 with 12 injected redundant Flip-Flops (FF). The process of adding the FFs is first finding the target in the bit stream and creating the redundant FFs and then routing them. The APUF instance remains the same but its response signal is significantly strengthened by the modification.

2 Statistical t-test

The t-test is used to determine if there are statistically significant differences in the power consumption between the two sets of APUF responses (either '0' or '1'). The t-test compares the means of the power trances to identify moments when the power consumption differs significantly between the two responses. This leakage about the APUF respons can then be used in a side-channel attack (SCA) to predict the output. The points with the highest t-statistic values indicate the points with greatest potential for use in the classification model.

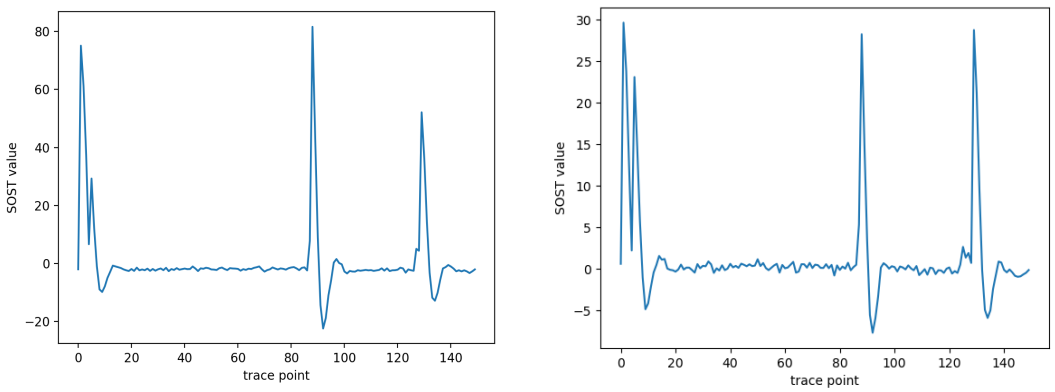

Figure 1: T-test for the two models. Figure left: PUF12. Figure right: PUF0

The t-test show similar results for the two models. PUF0 has the highest value for sum of squared pairwise t-differences (SOST) at trace points 1, 2, 5, 88 and 129 with a max SOST value of 29.7 while PUF12 has the highestr SOST values around trace points 1, 2, 88, 89 and 129 with a max SOST value of 81.6.

3 Sample distribution

The distribution of differences in mean power consumption between ’1’ and ’0’ can be used at a particular attack point as seen in figure 2.

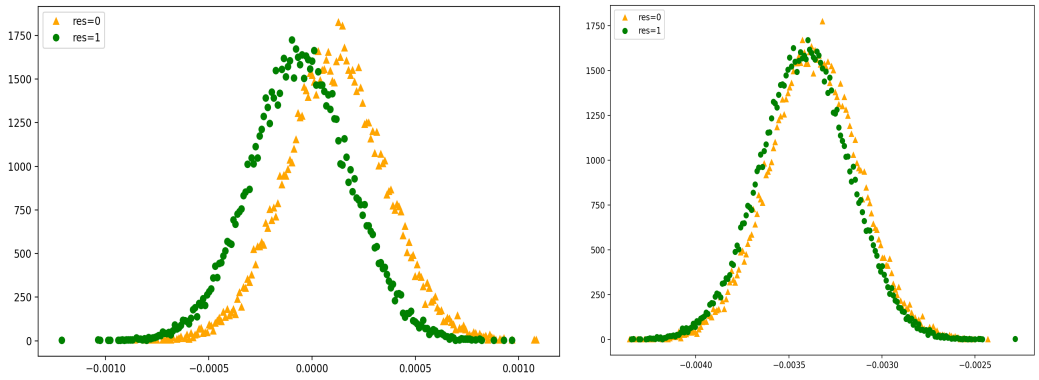

Figure 2: Sample distribution of difference in power consumption between '1' and '0'. Figure left: PUF12 with an attack point at trace 1. Figure right: PUF0 with an attack point at trace 88.

Obviously the more separated the distributions are the easier it is for a model distinguish and predict APUF responses. Through visual inspection we can conclude that the PUF12 traces, which uses redundant FFs, show more separation at a certain attack point in mean power output than the PUF10 traces.

4 Training

The models were trained with a Convolutional Neural Network (CNN), which is a type of deep learning algorithm often used in classification tasks that can overcome trace misalignment and jitter-based countermeasure. The model in our testing uses 1 Batch Normalization Layer, 2 Dense Layers (with 4 neurons) and 1 Output Layer. The baseline model uses a Rectified Linear Unit (ReLU) which is a type of activation function. It helps mitigate the vanishing gradient problem and accelerates convergence during the training process. To increase the accuracy of the models a few experiments where we conducted where we adjusted the number of neurons in the neural network and activation functions. The activation function that yielded the best results was the Exponential Linear Unit (ELU).

Activation Function	Accuracy		
	PUF0	PUF12	Blocks
ReLU (Baseline)	0.73257	0.9773	0.52605
ELU	0.73445	0.97742	0.51550

Table 1: Prediction of accuracy with different acitivation functions.

The PUF0 and PUF12 models get a slightly better accuracy score with the ELU activation function while the block implementation gets slightly worse. The ELU activation function is more robust to noise in the input data since it activates for negative values which could, potentially, result in better representation power and enabling the network to learn more subtle patterns in the data. However, the differences between the two are minor. Increasing the number of neurons in the dense layer did not have a significant impact on the accuracy of the models.

5 Testing

The performance of the neural network models was evaluated by testing them with repeated traces of the same challenge. The aim was to assess whether the use of these repeated traces could improve the model's classification accuracy. Various strategies were employed to achieve this goal. The first strategy involved trying a different number of repetitions and combining their predictions using the sum of logs to see if the classification accuracy improved. The second strategy was to take the average of 1,000 and 10,000 repetitions of the trace for the same challenge to produce one trace with an improved SNR (Signal Noise Ratio). The results in Table 2 show that all three models, PUF0, PUF12, and blocks, achieved an accuracy of 1, because the model's correct prediction confidence for the single trace was consistently between 0.5 and 1.0, leading to an accuracy of 1. Obtaining more traces might lead to lower accuracy because different traces have noise in different places, and the model might be confident in some areas while not in others. Taking the average of a high number of trace repetitions might increase the SNR by combining the raw prediction values to yield better results. The blocks model has been greatly improved using the average of 1000, whereas the PUF12 achieved lower accuracy because the flip-flop on the PUF12 underwent modifications to make traces leak more, but taking the average had the opposite effect and led to a lower signal-to-noise ratio. PUF0 achieved the highest accuracy with a trace of 100 repetitions.

Traces	Classification Accuracy		
	PUF0	PUF12	Blocks
1	1	1	1
10	0.5	1	0,3
100	0.83	0.97	0.48
1000	0.753	0.981	0.532
Average of 1000	0.608	0.605	1
10000	0.7386	0.981	0.5217
Average of 10000	0.5929	0.605	0.4

Table 2: Using different numbers of repetitions to increase the Classification Accuracy

6 Appendix

ReLU implementation. Only the function "create_model" was altered in the training file. The rest of the code remained the same.

```
    def create_model(classes=2, input_size=150):
        input_shape = (input_size,)

        model = Sequential()
        model.add(BatchNormalization(input_shape=input_shape))

        model.add(Dense(4, kernel_initializer='he_uniform',
            input_shape=input_shape))
        model.add(BatchNormalization())
        model.add(ReLU())

        model.add(Dense(classes, kernel_initializer='he_uniform', input_dim=32))
        model.add(Softmax())

        optimizer = Nadam(lr=0.001, epsilon=1e-08)
        model.compile(loss='categorical_crossentropy',
            optimizer=optimizer, metrics=['accuracy'])

        return model
```

Load testing data function "load_traces" improved to load traces and labels for a specific number of repetitions used in the testing stage.

```
# Load testing data
def load_traces(repetitions):
        # Load traces and labels for the specified number of repetitions
        traces = np.load(data_folder+'trace.npy')[:repetitions,:]#[0:1000,:]
        labels = np.load(data_folder+'label.npy')[:repetitions]#[0:1000]

        print('traces shape:', traces.shape)
        print('labels shape:', labels.shape)

        # Scale (standardize) traces
        delimitedTraces = np.zeros(traces.shape)
        for x_index in range(traces.shape[0]):
                delimitedTraces[x_index,:] = -1+(traces[x_index,:]-
                np.min(traces[x_index,:]))*2/
                (np.max(traces[x_index,:])
                -np.min(traces[x_index,:]))
```

```
        return (delimitedTraces, labels)
```

Improving the check_model function to run different strategies, that help improving the classification accuracy of neural networks.

```
# Check a saved model against test traces
def check_model(model_folder,model,count):
        # Load model
        model_load = load_sca_model(model_folder,model)
        all_predictions = []
        all_true_labels = []
    # Load traces and labels for the current repetition
        traces, labels = load_traces(10000)
    # Create a new trace with shape (0, 150) for the average trace
        new_trace = np.zeros((0, 150))

        # Compute the average trace
        average_trace = np.mean(traces, axis=0)
        new_trace = np.vstack((average_trace, new_trace))

        # coorect if 0
        num_attack = int(labels.shape[0])
        ranks_prd = np.zeros((num_attack, 1))
        # change the trace depend on the stratge
        ranks_prd = rank_func(model_load, new_trace, labels)
        # Calculate classification accuracy for the current repetition
        classification_accuracy = 1 - np.mean(ranks_prd)
        # print the calssification accuracy for the average trace
        print(classification_accuracy)
        # Append classification accuracy and true labels
        # used to get the sum of log
        all_predictions.append(classification_accuracy)
        all_true_labels.append(np.mean(labels))
        # Combine predictions using sum of logs
        predictions = np.sum(np.log(all_predictions), axis=0)
        # Calcultate probability of correct prediction
        a = np.exp(predictions)
        print('test accuracy:', a)
        print('mean of lables:', all_true_labels)

        return
```