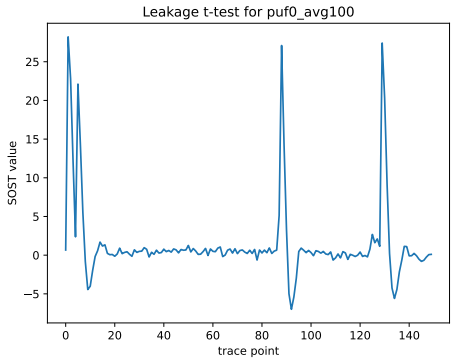
Hardware security project: SCA on Physically Unclonable Functions

Daniel Bücheler and Georgios Akkogiounoglou

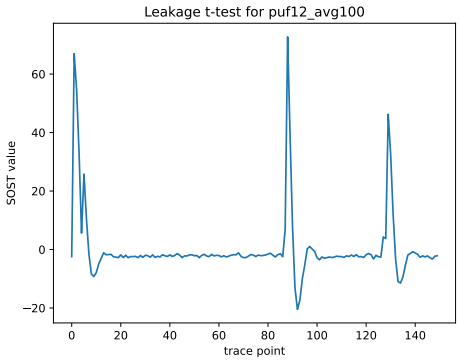
May 1, 2023

1 Introduction

A Physical Unclonable Function (PUF) is a hardware security primitive that generates a unique response based on physical properties of the device. PUFs are providing strong protection against cloning and tampering attacks. In this project, we aim to train and test a deep learning model for PUF classification using power traces captured from an Xilinx Artix-7 FPGA implementing a 128-bit arbiter PUF. Our goal is to accurately classify the PUF output response as either 0 or 1, given a set of power traces.



(a) Without extra flip-flops, plot generated using all 90 000 traces

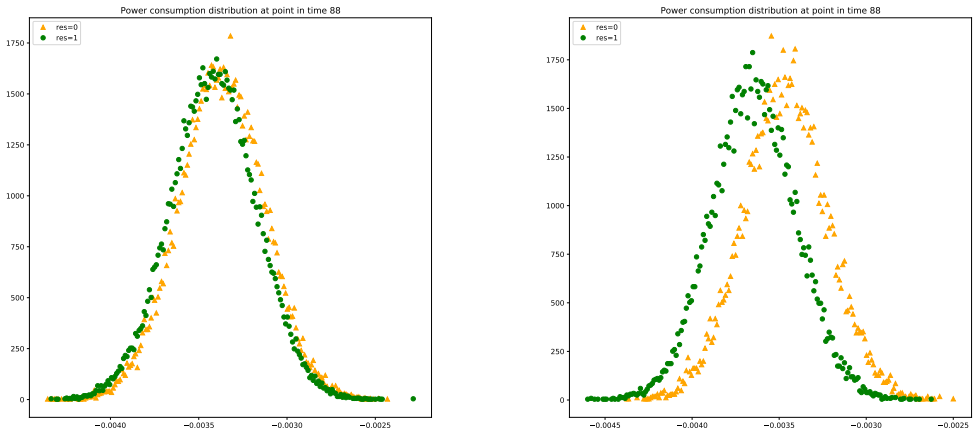


(b) With extra flip-flops, plot generated using all 40 000 traces

Figure 1: t-test results for the PUF with and without extra flip-flops

2 t-test results

Before training our models, we look at a t-test to determine if there is leakage. Figure 1 shows the results of the t-test run on the PUF's power consumption with and without the added extra flip-flops. It is immediately visible that the leakage appears at similar points in time, but the leakage is much stronger with the extra flip-flops embedded. Without extra flip-flops, the highest t-value obtained is about 28, whereas the highest t-value with the flip-flops is over 72. This is expected, as that was the purpose of adding the extra flip-flops.



(a) Without extra flip-flops

(b) With 12 flip-flops

Figure 2: Power distribution of 0 and 1 groups at $t = 88$

3 Distribution of power consumption for ’0’ and ’1’

Figure 2 shows the power distribution of ’0’ and ’1’ PUF responses for the implementation with and without 12 extra flip-flops added. The power consumption is plotted at $t = 88$, where we have observed the strongest leakage in the t-test. The green dots are representing the runs with response 0, while the yellow triangles represent runs with response 1.

In both cases, the power consumption follows a normal distribution with very similar means. However, without the extra flip-flops (Figure 2a), the distributions are closely aligned (similar mean) and hard to distinguish. With the extra flip-flops (Figure 2b), they are more clearly distinguishable (more different mean). This could be due to the added flip-flops changing the way that the PUF responds to input challenges, leading to a shift in the distribution of the output responses and increasing side-channel leakage.

4 Train models for both PUFs

To train the model, we first load the traces and labels from the specified directory (either ‘../traces/puf0_avg100/‘ or ‘../traces/puf12_avg100/‘), which contain the measured signals and the corresponding true output (bit) of the PUF. Then, it defines a neural network model with batch normalization and software output layer. Next, it trains the model using the ‘train_model‘ function, which takes in the training data, the model, and the name of the file to save the trained model. During training, is also saves the best-performing model based on the validation accuracy. Additionally, if uncommented, it generates a plot of the model accuracy per epoch using ‘plt.plot‘. Finally, it executes by loading the traces, creating and training the model, and saving the trained model.

groupsize	puf0	puf0example	puf12	puf12example
1	0.52644	0.52605	0.47276	0.47075
10	0.58756	0.57188	0.47042	0.47042
100	0.72266	0.66186	0.47042	0.47042
1000	0.84958	0.59956	0.47042	0.47042

Table 1: Model accuracy results

5 Testing the Models

The ‘puf_testing.py‘ tests a machine learning model. With the function ‘load_sca_model‘ which loads a saved model with or without extra flip flops. The function ‘rank_func‘, is defined to rank the predictions made by the loaded model against the labels of the test data. It then makes predictions and compares the labels with the true labels using the bitwise ‘xor‘ operation. Finally, the ‘check_model‘ function is defined to check the model against the test data. It loads the test traces and labels, loads the saved model and then ranks the model's predictions against the true labels. The accuracy of the model is calculated as 1 minus the mean rank. The models are tested using the non-averaged data from the PUF without additional flip-flops (`puf0_avg0_blocks_of_1000`). Unfortunately, no non-averaged data was available for the PUF without additional flip-flops, so we couldn't test the model performance in the same way.

In order to improve the accuracy, the model testing script combines predictions for multiple traces. As described in the project description, the non-averaged traces are captured as blocks of 1000 traces with the same input. Therefore, we combine the model's predictions on 1, 10, 100 or 1000 traces by multiplying all the trace's score vectors before choosing the most likely label. In order to do this in a numerically stable way, we use the logarithm of our probabilities, i.e. take the logarithm before adding up the score values (as $\log(\Pi_i p_i) = \sum_i \log(p_i)$). Because the logarithm is strictly monotone, this preserves our maximum.

6 Model accuracy results

The resulting model accuracy is presented in Figure 3 and Table 1. Four models are drawn in the graph, two trained on data from a PUF without extra flip-flops (`puf0`) and two from a PUF with extra flip-flops (`puf1`). The models called `_example` are the ones given by the instructors, the other two were generated using the code in Appendix A. Other custom models are not included, as they didn't perform significantly better than the example model.

First, it is immediately visible that the `puf12` perform very poorly. This is of course due to the fact that they were trained using different kind of data than they are now being tested. Unfortunately, the non-averaged trace data was not available with the extra flip-flops.

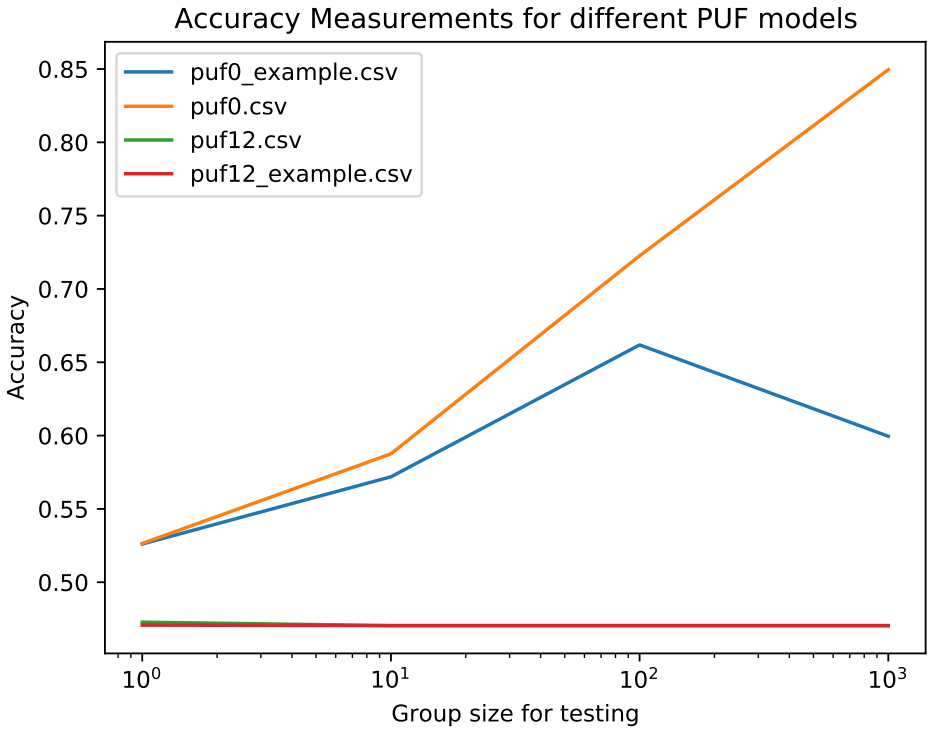


Figure 3: Resulting accuracy for different PUF models

Secondly, we can see that the accuracy improves significantly if we combine the prediction results from multiple traces. For example, when using a single trace, the accuracy for both `puf0` models is barely above 50%. When combining 100 or even 1000 traces, it can go as high as 85%, which is a very significant increase!

Interestingly, the example model performs best when combining 100 traces, whereas our custom model performs even better with block size 1000. It is not really clear why this is the case, as also the parameters use for training the example model are not given.

A Code excerpts

The training code was mostly unchanged from the given script. Only the parameters were changed in the `create_model` function as given in Listing 1.

Listing 1: Model creation function with parameters

```
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(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'])

    # print(model.summary())
    return model
```

The code for testing models was also largely unchanged from the given script. Only the `rank_func` was adapted as depicted in Listing 2. In order to summarize over the respective block size, an additional dimension is added to the numpy array. Then, the logarithm of the probabilities is taken and the results are summarized along the new dimension. Lastly, the predictions are propagated again to the entire block of traces and a full list of predictions is obtained.

Listing 2: Ranking function with parameters

```
def rank_func(model, dataset, labels, blocksize):

        # Get the input layer shape
        input_layer_shape = model.get_layer(index=0).
            input_shape

        if len(input_layer_shape) == 2:
                # This is a MLP
                input_data = dataset
        elif len(input_layer_shape) == 3:
                # This is a CNN: reshape the data
                input_data = dataset
                input_data = input_data.reshape((input_data.
                    shape[0], input_data.shape[1], 1))
        else:
                print("Error: model input shape length %d is
                     not expected ..." % len(
                    input_layer_shape))
                sys.exit(-1)

        # Combine probabilities for blocksize amount of
            traces
        predictions = model.predict(input_data)
        predictions_blocks = predictions.reshape(-1,
            blocksize, predictions.shape[-1])
        predictions_blocks = np.argmax(np.sum(np.log(
            predictions_blocks), axis=1), axis=1)
        predictions = np.repeat(predictions_blocks,
            blocksize)

        # xor = 0 if the same
        ranks_prd = np.bitwise_xor(predictions, labels)

        return (ranks_prd)
```