System Security (201700086)
Lab 1, Twente, Group 2

Æde Symen Hoekstra, Luigi Coniglio (Group 24)

May 17, 2018

1 PRNG

For this part of the assignment we first tried to implement a PRNG based around the digits of pi. However, this had a few downsides; firstly, the generator slowed down as it progressed, and secondly, the generator could not be properly seeded since this is not possible when calculating the digits of pi. We then switched to a Xorshift PRNG. Xorshift PRNGs are based upon LFSRs and are very fast and easy to implement. Our implementation is included in appendix A.

1.1 Results

Table 1: NIST statistical test results

test	P	result
monobit_test	0.678700221962	PASS
frequency_within_block_test	0.469369873972	PASS
runs_test	0.674495715008	PASS
longest_run_ones_in_a_block_test	0.9117452488814	PASS
binary_matrix_rank_test	0.482595351395	PASS
dft_test	0.959577289345	PASS
non_overlapping_template_matching_test	1.00000742287	PASS
overlapping_template_matching_test	0.486301369732	PASS
maurers_universal_test	0.999298778301	PASS
linear_complexity_test	0.0	FAIL
serial_test	0.822214445586	PASS
approximate_entropy_test	0.916508707114	PASS
cumulative_sums_test	0.7284271944	PASS
random_excursion_test	0.139100632064	PASS
random_excursion_variant_test	0.0792332113808	PASS

As you can see the Xorshift PRNG passes a lot of the NIST statistical tests, it fails however for the linear complexity test. The purpose of the linear complexity test is to determine whether the provided sequence s has an adequate degree of complexity to be considered random, this is done by calculating the length of the shortest LFSR to generate s (using the Berlekamp-Massey algorithm). If the length is too short then the sequence is not random.

The Xorshift PRNG can be improved by combining it with some non linear function such as addition or multiplication. Combining it with such a non linear function will help scramble the output of the PRNG in order the pass for example the linear_complexity_test.

2 Arbiter-PUF and XOR PUF

For this part of the assignment we implemented a simulation of an arbiter PUF and a simulation of a XOR PUF consisting of the combination of n arbiter PUFs (we used $n = 3$ for this assignment). For each of the PUFs we provided two implementations:

- A minimalistic implementation where the PUF is modeled as a simple weights vector, the challenges as features vectors and the output of the PUF is obtained with a simple multiplication of those two vectors. We like this implementation because it is elegant, fast and very concise. This model corresponds to the method proposed by [1].
- A more complete implementation where the PUF is modeled as a set of stages and each stage provides different delays according to the challenge bit. This implementation is bigger and slower, but it provides more insights regarding the inner working of an arbiter PUF.

Practically speaking the two implementations are equivalent.
The code can be found in the respective folders *minimalPUF* and *completePUF*.

After implementing the PUF we attacked it using a logistic regression classifier to try to predict the output of a PUF. We used the implementation of a logistic regression classifier provided by the python machine learning library *scikit-learn*.

2.1 Results

We conducted the attack on a simple arbiter PUF consisting of 32 stages and a XOR PUF consisting of 3 PUFs each with 8 stages. For our first attack we trained the classifiers using 100 randomly generated challenge response pairs (CRP) and test the result on 10000 CRPs. Table 2 shows the average score (on 100 runs) of the classifier for both PUFs (score is expressed in terms of probability).

Table 2: Average score of logistic regression classifier on arbiter and XOR PUFs

	Average score
Arbiter PUF	0.87343515625
XOR PUF	0.60555952381

While the classifier scores quite well on the arbiter-PUF the same cannot be said regarding the XOR PUF. In this last case the score of the classifier is closer to random guess, meaning that the layer of protection added by xoring the results of multiple PUFs (therefore introducing non-linearity) was an effective countermeasure against this type of classifier.

2.2 Number of CRPs and classification

Normally, the performances of a classifier are positively related to the size of the training set it was trained on. In other words, a classifier trained on a bigger training set performs better than one trained on a smaller set.

In the context of PUFs this means that the ability of an attacker in modeling the PUF is directly related to his ability in collecting challenge response pairs. An attacker which was able to collect more CRPs would also be able to create a better model of the target PUF.

For this part of the assignment we wanted to show this relation using our PUFs implementations and classifier. In our implementation the number of CRPs, as well as other parameters, is not fixed, this let us evaluate the performance of the classifier on different training set sizes.

Figure 1 shows the performances of the logistic regression classifier for the two PUFs trained on a different number of CRPs. Like before, we used a 32 stages arbiter PUF and a XOR PUF consisting of 3 arbiter PUF of 8 stages each. We tested the result of the classifiers on 10000 CRPs.

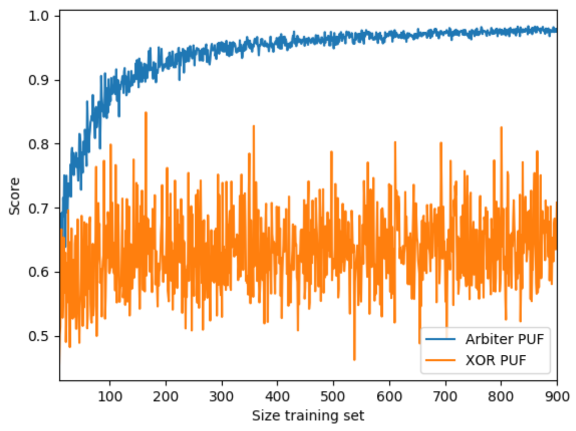

Figure 1: Score of the classifier per number of CRPs

As it was expected the classifier for the simple arbiter PUF increase with the size of the training suite. On the other hand, the results for the XOR PUF are very irregular, index of the fact that the classifier is not able to create a model of the XOR PUF despite the number of CRPs provided.

2.3 Number of stages and classification

So far we always used an arbiterPUF with 32 stages and a XOR PUF with 8 stages, but how much does the number of stages influences the effectiveness of our attack? In order to answer this question we trained a classifier for both PUFs using a variable number of stages. Figure 2 shows our result. As in section 2.1 we trained the logistic regression classifiers on 100 CRPs and evaluate their performances on a test suite of 10000 CRPs.

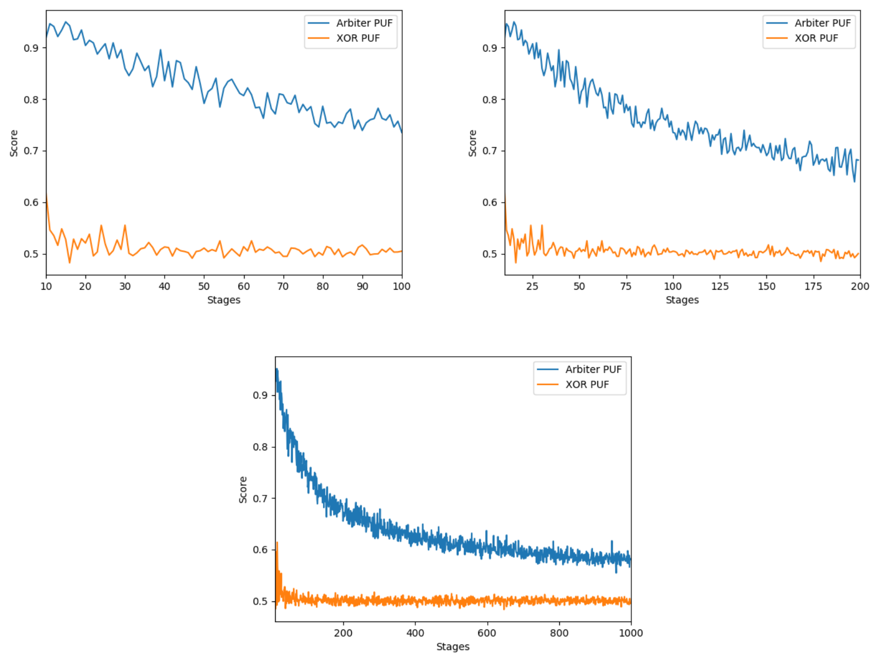

Figure 2: Performances of the classifier per number of stages.

As we can see the effectiveness of the classifier on the arbiter PUF drops quickly as the number of stages increases. Furthermore the figure confirms once again that our attack is totally ineffective against XOR PUFs.

References

[1] Johannes Tobisch and Georg T. Becker, *On the Scaling of Machine Learning Attacks on PUFs with Application to Noise Bifurcation.* NHorst Görtz Institute for IT Security & Ruhr-University Bochum, Germany

A PRNG Code

```
class PRNG(object):
    def __init__(self):
        self.seed([1, 2, 3, 4])

    def seed(self, state):
        self.state = state

    def iter_bits(self):
        while True:
            self.state = self.xorshift128(self.state)
            yield self.state[0] & 1

    def xorshift128(self, state):
        # https://en.wikipedia.org/wiki/Xorshift
        state = state[:]
        s = t = state[3]
        t ^= (t << 11) & 0xffffffff
        t ^= (t >> 8) & 0xffffffff
        state[3] = state[2]
        state[2] = state[1]
        state[1] = s = state[0]
        t ^= s
        t ^= (s >> 19) & 0xffffffff
        state[0] = t
        return state
```