Problem 1.1 (Uncovering the Mask of Xorro). Melbo was unhappy that the arbiter and XOR PUF devices were broken so easily using a simple linear model. In an effort to make an unbreakable PUF, Melbo came up with another idea devices called ring oscillators. Below, we first describe a traditional ring oscillator, then we describe a more interesting ring oscillator using XOR gates (called a XOR Ring Oscillator or XORRO) and finally we describe how to create a powerful PUF using multiple XORROs.

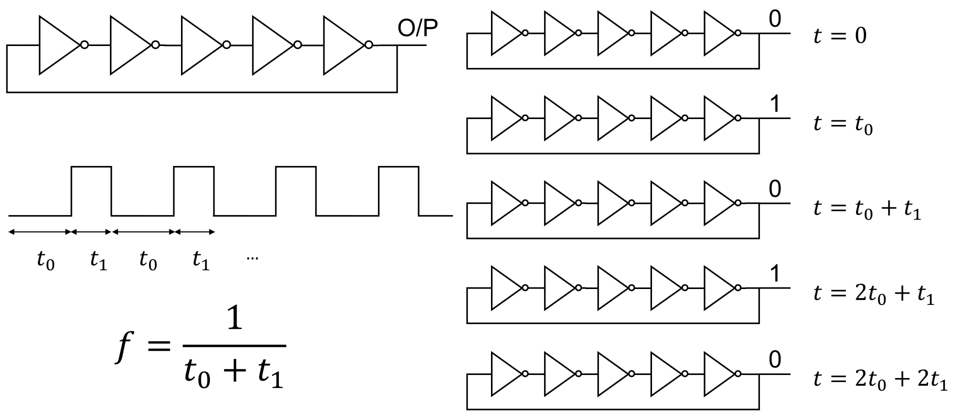


Figure 1: A simple ring oscillator using 5 inverters.

Ring Oscillator (RO). The figure above shows a simple ring oscillator created using 5 NOT gates. The NOT gates are inverters since they invert their input – a 0 input yields a 1 output and a 1 input gives a 0 output. The name of this device is very apt since the NOT gates are connected in a ring and the output of the last NOT gate (marked as O/P) oscillates between 0 and 1. To see why there is an oscillation, notice that if a bit is inverted an odd number of times, the final output will be inverted i.e. a 0 inverted an odd number of times times will yield a 1 and a 1 inverted an odd number of times will give us a 0.

Since the O/P is connected back as input to the first NOT gate, after 5 inversions, O/P will flip. Then that output is fed back again and thus, after 5 inversions, O/P will flip again. However, NOT gates do not give their output instantly – there is a delay. Suppose the 5 NOT gates are such that if 0 is input to them, they give their output (which will be 1) after δ_i^0 seconds $i = 0, 1, \ldots, 4$. Similarly, if the NOT gates are input 1, then they give the output 0 after δ_i^1 seconds $i = 0, 1, \ldots, 4$. We assume that delays caused due to the wires connecting the NOT gates is absorbed in the δ_j^i values themselves.

This means that if the value of O/P is 0 at $t = 0$, then it will flip to 1 after $t_0 = \delta_0^0 + \delta_1^1 + \delta_2^0 + \delta_3^1 + \delta_4^0$ seconds. Similarly, it will flip back to 0 after $t_1 = \delta_0^1 + \delta_1^0 + \delta_2^1 + \delta_3^0 + \delta_4^1$ seconds. Note that it may be the case that $t_0 \neq t_1$. However, it can be safely assumed that t_0 and t_1 remain constant over time and do not change. It is also very difficult to manufacture an RO with precise the same values of t_0, t_1 which is why the frequency of the RO

$$f \overset{\text{def}}{=} \frac{1}{t_0 + t_1}$$

can be treated as a unique fingerprint of the RO.

XOR Ring Oscillator (XORRO). The simple RO does not allow multiple challenge-response pairs (CRP) to be created. To remedy this, we notice that the XOR gate can also act as a *configurable* inverter. The XOR gate takes two inputs and the output is 1 only if exactly one of the inputs is 1. If both or neither of the inputs is 1 then the output is 0. Note that if the second input to a XOR gate is fixed to 1 then the XOR gate starts acting as an inverter

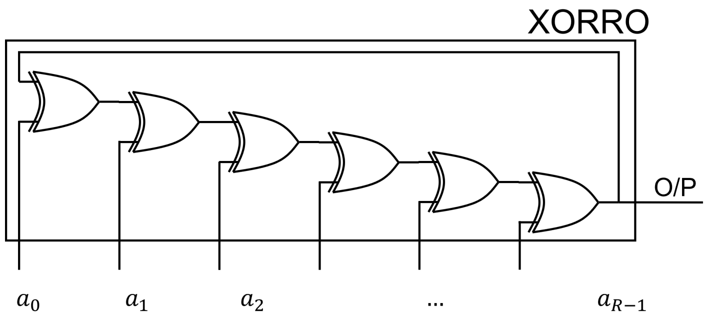


Figure 2: A configurable ring oscillator using 5 XOR gates.

with respect to the first input since XOR(0,1) = 1 and XOR(1,1) = 0. Similarly, if the second input to a XOR gate is fixed to 0 then the XOR gate starts acting as an identity with respect to the first input since XOR(0,0) = 0 and XOR(1,0) = 1. Using this, the XOR Ring Oscillator is created as shown in the figure above.

The XORRO has R XOR gates and for each gate, the second input is linked to a config bit that tells us whether that XOR gate will act like an inverter or an identity. The first inputs of all the XOR gates is connected in a ring to create a ring oscillator. Note that in order for the XORRO to indeed oscillate, an odd number of XOR gates must act as inverters. This means an odd number of config bits $a_0, \ldots a_{R-1}$ must be set to 1 otherwise the XORRO will not oscillate.

The XORRO offers us flexibility to create multiple challenge responses pairs since there are more delay parameters now. Let $\delta^i_{00}, \delta^i_{01}, \delta^i_{10}, \delta^i_1$ be the time that the i^{th} XOR gate takes before giving its output when the input to that gate is, respectively 00, 01, 10 and 11.

Note that the XORRO does not have a unique oscillation frequency as its oscillation frequency depends on the *config* bits $\mathbf{a} \stackrel{\text{def}}{=} [a_0, a_1, \ldots, a_{R-1}]$ send into the XORRO. This gives us potentially 2^R different oscillation frequencies and we can set a particular frequency simply by setting the corresponding config bits. Note that it is very difficult to manufacture a XOR gate with exactly these 2^R frequencies but we assume that the frequencies remain stable over time.

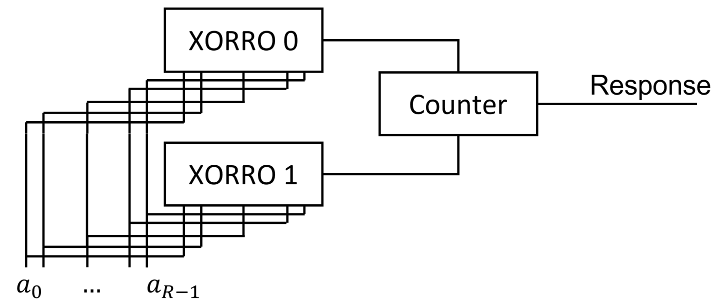
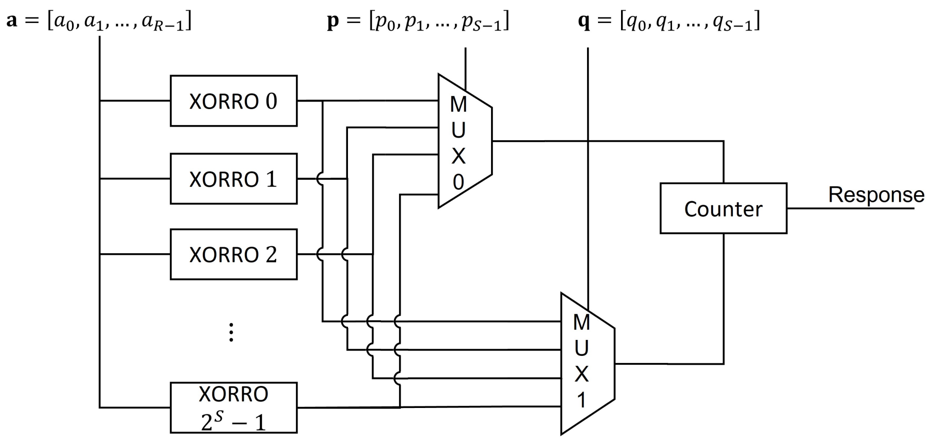


Figure 3: A Simple XORRO PUF using 2 XORROs.

Simple XORRO PUF. The above realization allows us to create a PUF as shown above. We take two XORROs. Our challenge is an R bit string that is fed into both XORROs as their config bits. Each of the XORROs will now start oscillating. The (oscillating) outputs of the two XORROs is fed into a counter which can find out which XORRO has a higher frequency. If the upper XORRO (in this case XORRO 0) has a higher frequency, the counter outputs 1 else if the lower XORRO (in this case XORRO 1) has a higher frequency, the counter outputs 0. The output of the counter is our response to the challenge. Assume that it will never be the case that both XORROs have the same frequency i.e. the counter will never be confused.



$\mathbf{a} = [a_0, a_1, \dots, a_{R-1}]$
$\mathbf{p} = [p_0, p_1, \dots, p_{S-1}]$
$\mathbf{q} = [q_0, q_1, \dots, q_{S-1}]$
XORRO 0
XORRO 1
XORRO 2
XORRO $2^S - 1$
MUX 0
MUX 1
Counter
Response

Figure 4: An Advanced XORRO PUF using 2^S XORROs.

Advanced XORRO PUF. Worried by the previous failures, Melbo does not want to take any chances this time and decides to make the PUF even stronger. Instead of having just 2 XORROs, Melbo now takes 2^S XORROs and extends the challenge vector to have $2S$ more bits which we call *select* bits. The first S of these select bits is interpreted as a number between 0 and $2^S - 1$ and used to select the corresponding XORRO as the first XORRO. This selection is done by using a multiplexer. The next S select bits are used to select another XORRO as the second XORRO. For example, if the $S = 3$ (i.e. there are $2^3 = 8$ XORROs) and the $2S$ bits are 110010 then the XORRO number 6 will get selected as the first XORRO (as $110 \equiv 6$) and XORRO number 2 will get selected as the second XORRO (as $010 \equiv 2$).

Note that for this selection scheme to work, the XORROs are numbered from 0 to 7, not 1 to 8. Also, notice that the counter will definitely get confused if the same XORRO is chosen twice e.g. if the select bits are 110110 or 010010. Thus, we are assured that the first S select bits are not the same as the second S select bits. The frequencies of the two selected XORROs are compared using a counter. If the XORRO selected by the upper MUX (in this case MUX 0) has a higher frequency, the counter outputs 1 else if the XORRO selected by the lower MUX (in this case MUX 1) has a higher frequency, the counter outputs 0. The output of the counter is our response to the challenge. Note that the challenge now has $R + 2S$ bits.

Melbo thinks that with 2^R possible frequencies in each XORRO and 2^S XORROS, there is no way any machine learning model, let alone a linear one, can predict the responses if given a few thousand challenge-response pairs. Your job is to prove Melbo wrong! You will do this by showing that even in this case, there does exist a model that can perfectly predict the responses of a XOR-PUF and this model can be learnt if given enough challenge-response pairs (CRPs).

Your Data. We have provided you with data from an Advanced XORRO PUF with $R = 64, S = 4$ i.e. it has $64 + 4 + 4 = 72$ bit challenges. The training set consists of 60000 CRPs and the test set consists of 40000 CRPs. If you wish, you may create (held out/k-fold) validation sets out of this data in any way you like. Your job is to learn a model that can accurately predict the responses on the test set. However, a twist in the tale is that unlike in the Arbiter/XOR PUF case where a single linear model was sufficient to break the PUF, here you will need an *ensemble* model that contains $M > 1$ linear models (more details below).

Your Task. The following enumerates 4 parts to the question. Parts 1,2,4 need to be answered in the PDF file containing your report. Part 3 needs to be answered in the Python file.

1. By giving a detailed mathematical derivation (as given in the lecture slides), show how a simple XORRO PUF can be broken by a single linear model. Recall that the simple XORRO PUF has just two XORROs and has no select bits and no multiplexers (see above figure and discussion on Simple XORRO PUF). Thus, the challenge to a simple XORRO PUF has just R bits. More specifically, give derivations for a map $\phi : \{0,1\}^R \to \mathbb{R}^D$ mapping R-bit 0/1-valued challenge vectors to D-dimensional feature vectors (for some $D > 0$) and show that for any simple XORRO PUF, there exists a linear model i.e. $\mathbf{w} \in \mathbb{R}^D, b \in \mathbb{R}$ such that for all challenges $\mathbf{c} \in \{0,1\}^R$, the following expression

$$\frac{1 + \text{sign}(\mathbf{w}^\top \phi(\mathbf{c}) + b)}{2}$$

gives the correct response. (10 marks)

2. Show how to extend the above linear model to crack an Advanced XORRO PUF. Do this by treating an advanced XORRO PUF as a collection of multiple simple XORRO PUFs. For example, you may use $M = 2^{S-1}(2^S - 1)$ linear models, one for each pair of XORROs, to crack the advanced XORRO PUF. (10 marks)

3. Write code to solve this problem by learning the M linear models $\mathbf{w}^1, \ldots, \mathbf{w}^M$. You may use any linear classifier formulation e.g. LinearSVC, LogisticRegression, RidgeClassifier etc., to learn the linear model. However, the use of non-linear models is not allowed. You are allowed to use scikit-learn routines to learn the linear model you have chosen (i.e. you do not need to solve the SVM/LR/RC optimization problems yourself). Submit code for your chosen method in `submit.py`. Note that your code will need to implement at least 2 methods namely

   (a) `my_fit()` that should take CRPs for the advanced XORRO PUF as training data and learn the M linear models.

   (b) `my_predict()` that should take test challenges and predict responses on them.

   We will evaluate your method on a different dataset than the one we have given you and check how good is the method you submitted (see below for details). **Note that your learnt model must be a collection of one or more linear models**. The use of non-linear models such as decision trees, random forests, nearest neighbors, neural networks, kernel SVMs etc is not allowed. (35 marks)

4. Report outcomes of experiments with both the sklearn.svm.LinearSVC and sklearn.linear_model.LogisticRegression methods when used to learn the ensemble linear model. In particular, report how various hyperparameters affected training time and test accuracy using tables, charts. Report these experiments with both LinearSVC and LogisticRegression methods even if your own submission uses just one of these methods or some totally different linear model learning method e.g. RidgeClassifier) In particular, you must report the affect of training time and test accuracy of at least 2 of the following:

   (a) changing the `loss` hyperparameter in LinearSVC (hinge vs squared hinge)

   (b) setting `C` hyperparameter in LinearSVC and LogisticRegression to high/low/medium values

   (c) changing the `tol` hyperparameter in LinearSVC and LogisticRegression to high/low/medium values

   (d) changing the `penalty` (regularization) hyperparameter in LinearSVC and LogisticRegression (l2 vs l1)