# Deep Learning for XOR-Arbiter PUFs

Vlad Vasilescu

# What are XOR-Arbiter PUFs ?

- PUFs (Physically Unclonable Functions) - electronic circuits that manifest randomness in it's behavior due to it's technology of manufacturing
- It's considered that any electronic circuit is *unique (or unclonable)*, distinguishing it from any other circuit (even from the same family)
- A PUF instance is basically a function that maps an n-bit input to a m-bit output, which should be unique to any instance - i.e. there exists a unique function f that characterizes any circuit $f : \{0,1\}^n \rightarrow \{0, 1\}^m$
- Due to it's uniqueness various PUF combinations have been proposed for lightweight authentication protocols or remote hardware authentication
- Any pair consisting of a n-bit input and m-bit output is denoted as a "Challenge-Response-Pair" which can be used as a verification step for the regarding circuit

- A k-XOR Arbiter PUF consists of k Arbiter PUFs in parallel (meaning they get the same input) while their outputs are X-ORed, resulting in a much more difficult output to predict comparing to a single Arbiter
- Basically a PUF Arbiter determines how fast an arbitrary signal travels through n-Switch gates (a switch gate has 2 inputs and 2 outputs ; it also has 1 control bit which determines if the 2 inputs should continue their "own path" or switch places) ; the final circuit determines which of the 2 paths was faster and outputs 1 or 0;
- These binary outputs resulting from all k Arbiters (when the same input trigger is applied and the same control bits are set) are XORed, resulting the global output
- In our future discussion we'll refer to the control bits as our input and the final XOR output as our global output (since the trigger signal is the same for all arbiters)

# The Dataset

- The dataset used in this project is public, and can be found using the following link: https://github.com/Praneshss/Modeling_of_APUF_Compositions
- The features (parity bits) can be found in the file APUF_XOR_Challenge_Parity_64_1Million.csv
- The responses for each example can be found in the file 6resp_XOR_APUF_chal_64_1Million.csv
- The dataset contains 1 000 000 CRPs
- The features represent 64 parity bits, computed using the 64 control bits

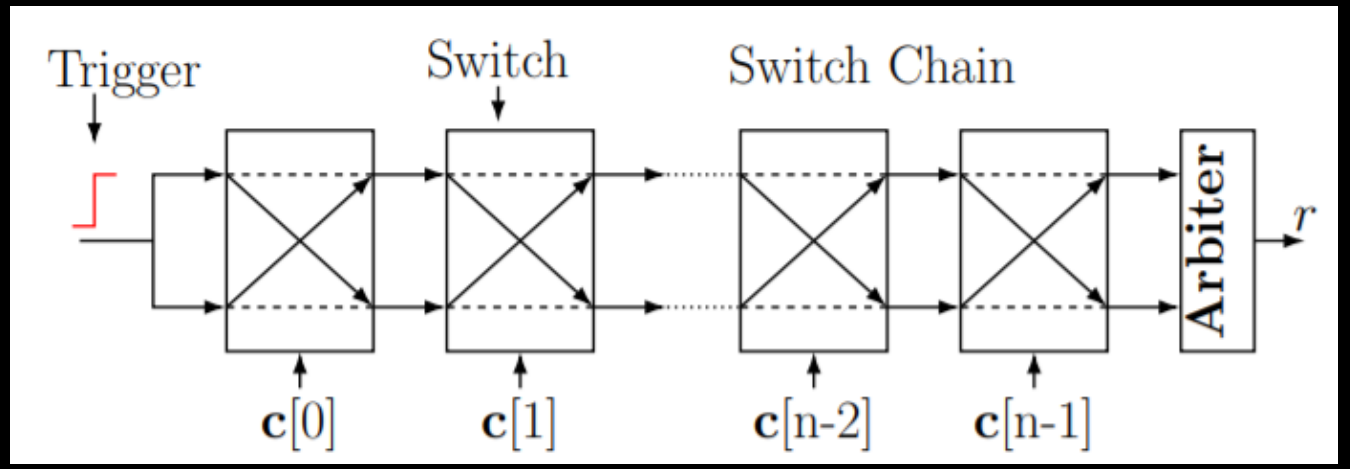
# The Dataset

- Given 64 control bits the parity bits can be calculated using the following formula

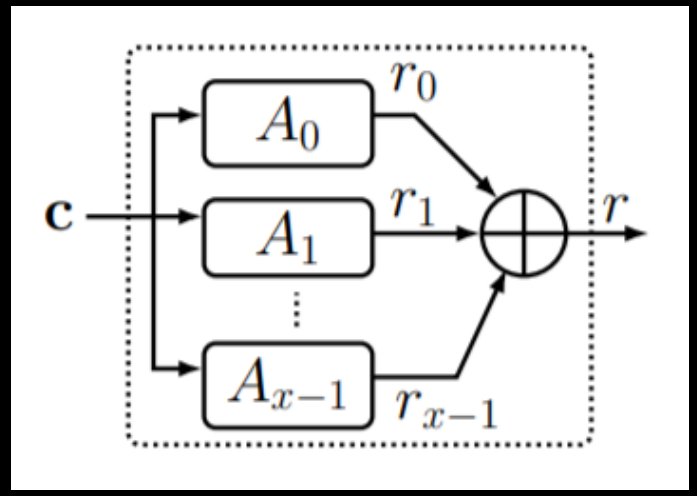
$$\Phi[n] = 1$$

$$\Phi[i] = \prod_{j=i}^{n-1} (1 - 2c[j]),\ i = 0, 1, \cdots n-1$$

- As you can see the last bit is always 1, that's why we'll exclude it, leaving us with 64 parity bits
- Control bits can be either 0 or 1 (like normal bits) while parity bits take values from {-1, 1}
- There are different rules for calculating parity bits, depending on the Arbiter type we're using




n-bit Arbiter PUF




x-XOR Arbiter PUF

# Deep Learning based Attacks on XOR-Arbiter PUFs

- Efficient attacks on these type of security systems are very important due to the fact that for a sufficient number of known CPRs (input-output pairs) we can manage to predict the output for almost all input combinations
- This leads to a pretty big flaw in this type of system, as we can manage to *copy* it's identity (using the appropriate mathematical model) based on a sufficient number of known CPRs
- Neural Netowrks are universal function approximatiors, so it is intuetivly to use them for modeling our system
- In this project I'll present 2 different approaches on building NNs for predicting the output of 6-XOR Arbiters
- I've chosen 6-XOR Arbiters because many articles show that for a number of arbiters $n < 6$ the input-output mapping can be learned by simple models such as Logistic Regression

# First approach

- Simple forward NN with ReLU activations in between layers and Softmax activation for output layer
- Input dimension : 64 ; Layer dimensions : 256, 256, 128, 32, 2
- Optimized using Adam algorithm, with learning rate = 0.001
- Loss : categorical crossentropy
- Number of classes : 2

# First approach – model summary

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input_1 (InputLayer) | (None, 64) | 0 |
| dense_1 (Dense) | (None, 256) | 16640 |
| dense_2 (Dense) | (None, 256) | 65792 |
| dense_3 (Dense) | (None, 128) | 32896 |
| dense_4 (Dense) | (None, 32) | 4128 |
| dense_5 (Dense) | (None, 2) | 66 |

Total params: 119,522
Trainable params: 119,522
Non-trainable params: 0

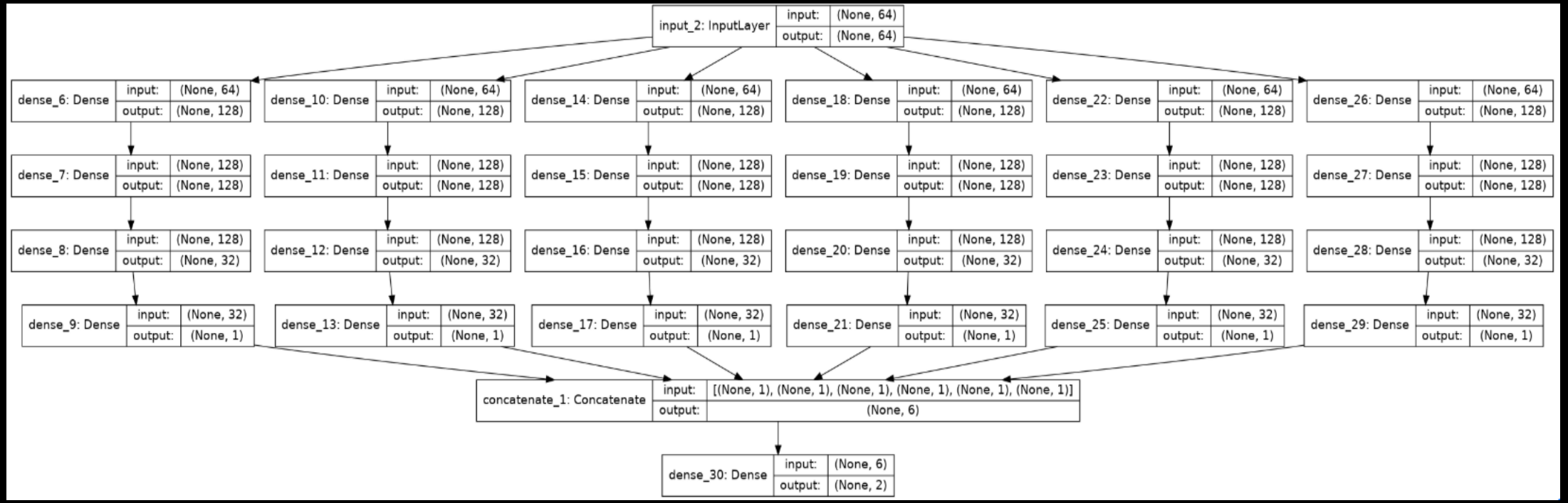
# First approach - results

- Training was done using 750 000 examples
- Testing was done using 250 000 examples
- Batch size was set to 1000
- Number of epochs for training : 2
- Accuracy on train set : 99.03 %
- Accuracy on test set : 98.94 %
- Time takne for training : 2:10 minutes

# Second approach

- While the first approach was not something uncommon, the second one aims to replicate the entire physical circuit, using 6 parallel NNs (1 for each Arbiter)
- Input is fed to 6 NNs which operate in parallel
- Their results are concatenated and passed to the output layer
- Each of the 6 NNs contains 4 Dense layers (128, 128, 32, 1)
- First 3 layers of each NN use ReLU activation, and the last one uses Sigmoid
- The reason for using Sigmoid is that the last neuron should model the output of each Arbiter, that is either 0 or 1

# Second approach – model graph



# Second approach - results

- Sam training and testing parameters
- Imroved train and test acc
- Train accuracy : 99.23 %
- Test accuracy : 99.25 %
- More time taken for training : 6:08 minutes

# Conclusion

ACCURACY IMPROVED WHEN MODEL IS BUILD IN A MANNER THAT REPRESENTS THE ACTUAL PHYSICAL SYSTEM

HOWEVER, THE COMPLEXITY OF THE SECOND MODEL IS REFLECTED IN THE SUBSTANTIAL INCREASE OF TRAINING TIME

WE MAY TRY TO FIND A TRADE-OFF BETWEEN COMPLEXITY AND PERFORMANCE

PARALLEL COMPUTING CAN ALSO BE USED IN ORDER TO DECREASE TIME TAKEN OR TRAINING, WITHOUT AFFECTING TOO MUCH THE ACCURACY

IN THE FUTURE I INTEND PUT TO THE TEST THESE IDEAS, AND ALSO WORK ON OTHER DATASETS