PUF Scripts

March 2016

Version 0.5

Document Revisions

Date	Version Number	Document Changes
04/05/2015	v0.1	Initial Draft
06/07/2015	v0.2	Second Draft
10/4/15	V0.3	Third Draft
12/8/15	V0.4	Fourth Draft
3/15/16	V0.5	Fifth Draft
8/5/16	V0.6	Sixth Draft

Table of Contents

1 Introduction

1.1 Scope and Purpose

Physical Unclonable Function (PUF) is a hardware primitive used for secure authentication of Integrated Circuits (IC). Furthermore, PUF can be employed to generate secure keys for cryptographic functions. This user guide is about the **software** scripts that can be used to generate PUF response bits, PUF evaluation and analyzing PUF responses under different conditions.

To use these scripts, user does not need an in-depth knowledge of Information theory, Hardware circuit design or Cryptography. However, user must know the basics of statistics. In addition, user is expected to know how to run an application from the command line. In depth details of these schemes is available at [9].

1.2 Tools

These scripts are written in Python, therefore to run it successfully; user must have Python version 3.3 installed on his/her machine. Additionally, input files are in Comma Separated Values (CSV) format. To read CSV files, a Python package named csv has been used.

2 Converting Raw results into PUF IDs

Python based scripts are employed to convert the raw data collected from the FPGA devices into a binary PUF response. Raw data consists of either the counts of Ring Oscillators (ROs) or SR-Latches. Ring oscillators are used in RO-PUF. Similarly SR-Latches are used in the design of SR-Latch PUF. In this document RO and Latches are called components. As shown below in Fig. 1, the ID generation script has two inputs, Raw PUF data and Parameters. The Metrics are covered in PUF properties.

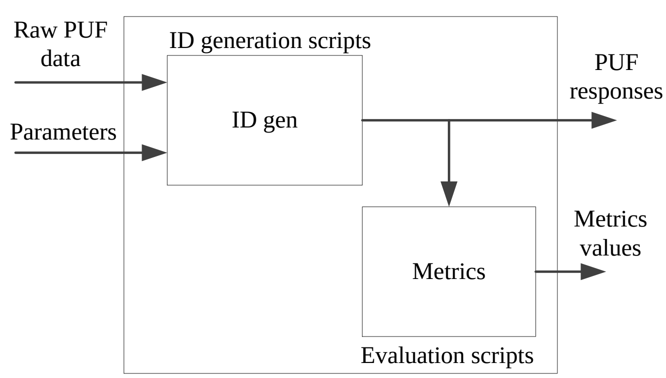


Fig. 1. PUF ID generation and evaluation

2.1 Configuration File

The script name *config.py* is provided. It lists all the parameters that are passed to PUF-ID generation schemes as shown in Fig. 2. In this file, the dataset consisting of Raw data is chosen. The input data is then passed to different schemes. We can enable multiple schemes to generate the PUF IDs. Another script file *all_schemes.py* is provided. This file imports the config parameters and reads raw data from csv files. Furthermore it also generates the PUF-IDs.

```
#Input file containing the Raw data
inputfile = 'VT_Enrolment_data.csv'

#Number of Devices
num_devices = 193

#Number of PUF response bits to generate
Response_bit_length = 128

#Directory Name of field input data required for Reliability
field_directory = 'VT_field'
```

Fig. 2. Config.py file format

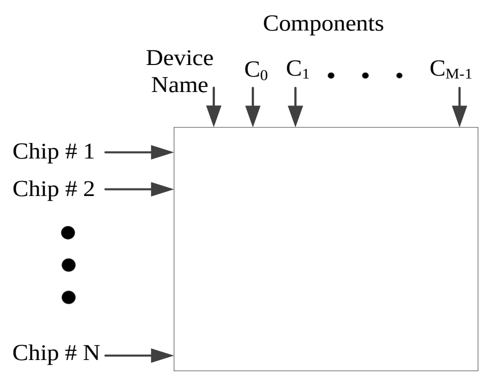


Fig. 3. Data input format for PUF-ID generating scripts

Raw data from the devices is stored in a csv file. As shown in Fig. 3, each device data is stored in a separate row. Similarly each row has M+1 columns, where first column contains the device name and the remaining M columns contains the raw data for M components. We provide the following five schemes to generate the PUF IDs. All the schemes can be called from *all_schemes.py*.

2.2 Pair-wise Comparison (PC)

The function named *PC()* is provided. It is called from *all_schemes.py* to generate the PUF response bits by comparing the neighboring components. In this comparison each component is compared only once with its neighbor. The comparison of (C_0, C_1), (C_2, C_3), (C_4, C_5)…. (C_{M-2}, C_{M-1}) is carried out. Therefore for M components the total number of PUF response bits will be equal to $\lfloor M/2 \rfloor$. Fig. 4 shows this scheme.

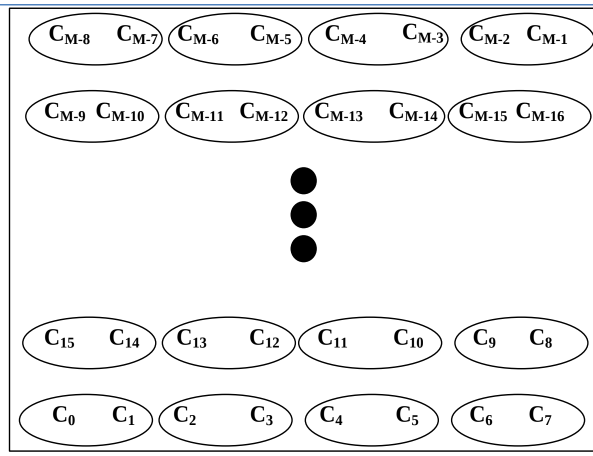


Fig. 4. Pairwise comparison with neighbors

Algorithm 1 Pairwise Comparison (PC)

```
PC(F[], M):  // F[] is the input array of M raw PUF responses
for(i=0; i <  ⌊M/2⌋ ; i++)
        if(F[2·i] > F[2·i+1])
                ID[i] = 1
        else
                ID[i] = 0
        end if
end for
return ID
```

Where `ID` array stores the PUF response of N components.

This scheme is called from *all_schemes.py* by providing the following parameters,

`PC(inputdata, outputfile, No. of devices, components, response)`

```
C:\CERG\PUF\scripts\PUF\docdb\McQ_3>all_schemes.py
All results stored at C:\CERG\PUF\scripts\PUF\docdb\McQ_3\VT_Results_3
Script used for Pairwise comparison, Each Component compared Once
Output file is 'VT_PC.txt'
Number of chips 2
Components per chip: 256
Total Components per chip:256
ID_size is 128

Chip identified: 'D059546'
Length of response 128
Response generated for chip#: 'D059546'

Chip identified: 'D061283'
Length of response 128
Response generated for chip#: 'D061283'

PUF IDs written to file: VT_Results_3\VT_PC.txt
```

2.3 Comparing the Neighboring Components (CNC)

The function named *CNC ()* is provided. It is called from *all_schemes.py* to generate the PUF response bits by comparing the neighboring components. In this scheme raw data from the neighboring components of FPGA are compared. Fig. 5 shows this scheme. In this figure C_0,

$C_1 \ldots C_{M-1}$ shows the physical location of components on the FPGA fabric. Raw data from the neighboring components are compared to mitigate the effect of systematic variation.

Algorithm 2 Comparing the Neighboring Components (CNC)

```
CNC(F[], M): // F[] is the input array of M raw PUF responses
for(i = 0; i< M-1; i++)
        if(F[i] > F[i+1])
                ID[i] = 1
        else
                ID[i] = 0
        end if
end for
return ID
```

Where ID array stores the PUF response of M components. In this scheme each component is compared with both neighbors except the first and last component. In case of first and last component, comparison is done only with a single neighbor. The comparison of (C_0,C_1), (C_1, C_2), (C_2, C_3)... (C_{M-2},C_{M-1}) is carried out. Therefore for M components the total number of PUF response bits will be equal to M-1.

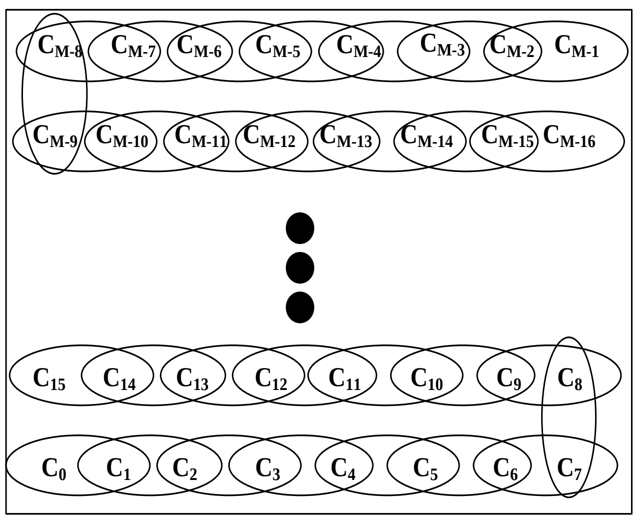


Fig. 5. Comparison of neighboring Components.

This scheme can be called from *all_schemes.py* as shown below,

```
pairwise_main( inputdata, outputfile, No. of devices, components, response)
```

```
C:\CERG\PUF\scripts\PUF\docdb\McQ_3>all_schemes.py
All results stored at C:\CERG\PUF\scripts\PUF\docdb\McQ_3\VT_Results_4
Script used for Pairwise comparison
Output file is 'VT_CNC.txt'
Number of devices 2
Components per chip: 129

Chip identified: 'D059546'
Length of response 128
Response generated for chip#:'D059546'

Chip identified: 'D061283'
Length of response 128
Response generated for chip#:'D061283'

PUF IDs written to file: VT_Results_4\VT_CNC.txt
```

2.4 Binary Lehmer-Gray (BLG) encoding

In LG encoding, all components are divided into sets of size S. Encoding the ordering of S component measurements $F^s = (F_0, \ldots, F_{s-1})$ results into an L-bit response.

A Lehmer code is a unique numerical representation of an ordering which is moreover efficient to obtain since it does not require explicit value sorting. It represents the sorted ordering of F components as a coefficient vector $L^{s-1} = (L_1, \ldots, L_{s-1})$ with $L_i \in \{0, 1, \ldots, i\}$. It is clear that L^{s-1} can take $2 \times 3 \times \ldots \times S = S!$ possible values which is exactly the number of possible orderings. The Lehmer coefficients are calculated from F as $L_j = \sum_{i=0}^{j-1} gt(F_j, F_i)$ with $gt(x, y) = 1$ if $x > y$ and 0 otherwise. The Lehmer encoding has the nice property that a minimal change in the sorted ordering caused by two neighboring values swapping places only changes a single Lehmer coefficient by ±1.

The total number of bits generated for each set is:

$$\text{Bits generated per set (L)} = \sum_{i=2}^{S} \lceil \log_2 i \rceil \qquad (1)$$

Below is a pseudo code for converting counts of S components into an L-bit response denoted by L_Response.

This scheme can be called from `all_schemes.py` as shown below,

```
BLG ( inputdata, outputfile, No. of devices, components, set_size, response)
```

```
C:\CERG\PUF\scripts\PUF\docdb\McQ_3>all_schemes.py
All results stored at C:\CERG\PUF\scripts\PUF\docdb\McQ_3\VT_Results_5
Script used for Lehmer-Gray Encoding
Output file is 'VT_BLG.txt'
Set size 16
Total Devices 2
Components per chip: 48

Chip identified: 'D059546'
ID size 147
Chip identified: 'D061283'
ID size 147

PUF IDs written to file: VT_Results_5\VT_BLG.txt
```

Algorithm 3 Binary Lehmer Gray Encoding

```
BLG (F[],M,S):  // S=Set size
for(t=0; t < ⌊M/S⌋; t++)
        ID= ID || Lehmer(F[t·S:(t+1)·S-1],S)
end for
return ID

Lehmer(array[],S):
for(j=1; j<S; j++)
  sum = 0
  for(i=0; i<j; i++)
        if (array[j] > array[i])
                sum++
        end if
  end for
  L_Response = L_Response || (Gray(bin(sum, ⌈log2 j⌉)))
end for
return L_Response

Gray(bin_bits)://array of binary bits is passed to Gray().
G[0] = bin_bits[0]
for(i=1; i < len(bin_bits); i++)
        G[i] = bin_bits[i] XOR bin_bits[i-1]
end for
```

Above in a pseudo code a function named bin(p,t), converts a decimal number p to t binary bits. For M components, the total response is equal to (M/S)*L bits. In [3, 4], set size S, is 16. Similarly, a function named Gray() is used for encoding the Lehmer co-efficients. Gray encoding make it sure that each subsequent number is only a single bit different than a previous one. Where bin_bits is an array of binary bits. G is an array of corresponding Gray encoded bits.

2.5 S-ArbRO-2

S-ArbRO-2 is described in [6]. In this design the number of CRPs have been improved. Components are divided into elements. Each element has a pair of components associated with it as shown below in Fig. 6,

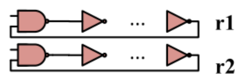


Fig. 6. Element contains a pair of components.

The difference between the counts of components in each element is the respective count associated with that element (r_1-r_2 or r_2-r_1). This difference in count value may be positive or negative. The next step is to select a group size for elements. This is done by selecting a value for parameter K. Inside this group, elements are added with each other. The range of K is $2 \leq K \leq N$. Here, N is the total number of elements and K is the number of elements selected in each group. Challenge is the selection of group of elements, while response (Rc) is the one bit result as shown in the figure below,

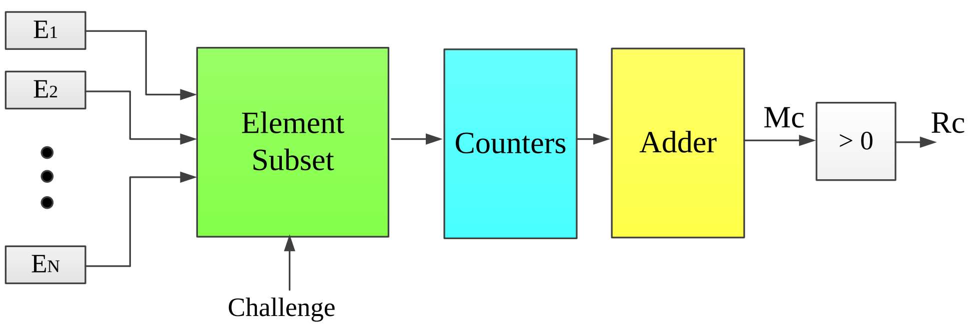


Fig. 7. S-ArbRO-2 showing the relationship between Challenge Response Pairs (CRPs)

Algorithm 4 S-ArbRO-2

```
S-ArbRO-2(E[],K, M/2):      //K is the size of an element group
i = 0
for combo in combinations(E,K,M/2):
//choose K out of M/2 elements each element E[s] consists of
//two raw responses E[s][0] and E[s][1]
  for(j=0; j < (2^K-1); j++)
     p = bin(j, K)
     sum = 0
     for (s=0; s<K; s++)
       if p[s] == 0
         sum += combo.E[s][0] - combo.E[s][1] //r1-r2
       else
         sum += combo.E[s][1] - combo.E[s][0] //r2-r1
       end if
     end for
     if sum > 0
      ID[i] = 1
     else
      ID[i] = 0
     end if
     i++
```

```
  end for
end for
return ID
```

As evident from the Fig. 7, the total number of elements is N.

If the result of adding elements is positive, the response is ‘1’, otherwise it is ‘0’. The total Challenge Response (CR) space is,

$$Total\ CR\ space = \frac{N!}{K! \times (N-K)!} \times 2^{K-1} \quad (2)$$

For example, 64 components will result in N = 32 elements. Suppose the parameter K is equal to 2. Then the total number of possible combinations are 992. In the pseudo code, E[] is an input array of N elements.

Each Element has 2 components. K is the subset size. Index holds all the possible combinations of $\binom{N}{K}$ elements. Sums will hold all the sums for K elements. Combination(N,K) calculates the $\binom{N}{K}$, sum(array) adds all the elements of an array. The response of S-ArbRO-2 is returned by an array Response[]. The pseudo code will generate all the possible CRPs. Assume the list of components is [10,5,6,4,17,11]. Therefore the three elements formed are E_1=[10,5], E_2=[6,4] and E_3=[17,11]. The possible number of combinations for K = 2 is $\binom{3}{2} = 3$. Hence three groups of elements formed will be {E_1, E_2}, {E_1, E_3} and {E_2, E_3}. Index will contain [{E_1, E_2}, {E_1, E_3},{E_2, E_3}] or [{[10,5], [6,4]}, {[10,5], [17,11]},{[6,4], [17,11]}]. If the group challenge is (01), it will select group {E_1,E_3}. Similarly inside each group if the challenge is (00). It will result in 0=>E_1[r1-r2]= 10-5= 5 and 0=>E_3[r1-r2]= 17-11= 6. Sums will hold 5+6=11. Since 11>0, therefore the final PUF response will be ‘1’.

This scheme can be called from *all_schemes.py* as shown below,

```
S_ArbRO_2(inputdata, outputfile, No. of devices, components, K, response)
```

```
C:\CERG\PUF\scripts\PUF\docdb\McQ_3>all_schemes.py
All results stored at C:\CERG\PUF\scripts\PUF\docdb\McQ_3\VT_Results_6
Script used for S_ArbRO_2
Number of chips 2
Components per chip: 24
Elements: 12.0
K chosen: 2

Chip identified: 'D059546'
ID_size is 132

Chip identified: 'D061283'
ID_size is 132


PUF IDs written to file: VT_Results_6\VT_SArbRO.txt
```

2.6 Identity Mapping

This scheme is described in [5]. In identity mapping *m* components can generate $2^m - m - 1$ response bits. In this method, *t* component counts are selected from *m* component counts where $2 \leq t \leq m$. Initially all pairs of component counts are determined S_2,

$$|S_2| = \binom{m}{2}$$

Similarly, S_3 contains all possible triplets of component counts.

$$|S_3| = \binom{m}{3}$$

Likewise,

$$|S_t| = \binom{m}{t}$$

Then a random variable Q_t is defined that assigns a real number *X* to each outcome of S_t

$Q_t: S_t \rightarrow X \text{ such that}$

$$Q_t(f_{x1}, f_{x2}, f_{x3}, \ldots . f_{xt}) = \sum_{u=1}^{t-1} \sum_{v=u+1}^{t} w_{(xu)(xv)} \cdot \left\| f_{(xu)} - f_{(xu)} \right\|^e \quad (3)$$

Where $1 \leq x1, x2, x3, \ldots, xt \leq m$

And, $x1 \neq x2 \neq x3 \neq .. \neq xt$ and $2 \leq t \leq m$

The weight factor $w_{(xu)(xv)}$ can depend on a particular design. However, in our script it is equivalent to 1. We chose 1 because we believe that systematic variations come into the effect, when far away components are compared. Therefore we keep the weight factor constant for all Q values. Response R, from Q is generated by using the following equation,

$$R = mod(Q[i]/q, 2)) \;\; for \text{ i =0,1,2,..}$$

Where q is the bucket size. The size of array Q depends on the value of *t* selected. For instance if t =2, the total elements of Q are $\binom{m}{2}$. If t=3, then total length of Q will be $\binom{m}{2} + \binom{m}{3}$, and so on.

In addition to the response bits, a set of helper data is also generated. This helper data is used to reduce the effect of noise in the field. For example, with noise the count value for a component is different from the one calculated during enrolment. Therefore, a helper data is used to mitigate the effect of noise. Helper data W_t is calculated using the following equation,

$$W_t = \left(\frac{q}{2}\right) - (Q[i] - q * \left\lfloor \frac{Q[i]}{q} \right\rfloor) \;\; for \text{ i=0, 1, 2, 3…}$$

In the above equation, q is the bucket size. It must be appropriately chosen. If q is chosen very big, then too many bits will be encoded into the same bucket. Therefore, it will reduce the uniqueness significantly. Similarly, if q is chosen very small, then it will affect the reliability property. A small change by the noise in the field will move the response bit to another bucket. It must be noted that for each element of Q, only one value of W_t is calculated.

In the pseudo code, components[] is an input array that contains the counts of components. q is the bucket size , parameter e is any real number except 1 and t is the parameter , such that $2 \leq t \leq m$. PUF Response during enrolment is stored in an array ID_enrollment. While Wt is the array that contains the helper data. In the field, each response bit is recalculated using W_t and noisy Q'_t values. ID_field[] contains the PUF response generated at the field.

This scheme can be called from *all_schemes.py* as shown below,

```
identity_map (inputfile, enrol_outputfile, field_outputfile,field_response,
enrol_data_row, field_data_row , components, q, t, e, response)
```

```
C:\CERG\PUF\scripts\PUF\docdb\McQ_3>all_schemes.py
All results stored at C:\CERG\PUF\scripts\PUF\docdb\McQ_3\VT_Results_7
Script used for Identity Mapping
Input file is'VT_field/1.2V_25C.csv'
Total chips available: 5
Components per chip: 17

Row selected during enrolment: 2
Chip identified: 'D113938.csv'
Bucket size chosen: 30
Parameter , t =  2
Parameter , e =  0.5

PUF bits generated: 136
PUF IDs written to file :VT_Enrol_R_Id_Map.txt
PUF Qs written to file : Enrol_Q
Helper Data written to file : Enrol_Wt.txt

Row selected, field data: 2
Chip identified: 'D113938.csv'

Field response written to file : Field_R.txt
```

Algorithm 5 Identity Mapping

```
Id map enroll(F[], t, q, e)://Enrollment: q, t and e
i=0                      //are parameters
for combo in combinations(F[], t, M)
 // choose t out of M raw results F[j]
 Qt = 0
 for (Ru, Rv) in combinations(combo, 2, t)
    // choose 2 out of t raw results F[j] from combo
    Qt += |Ru - Rv|^e
 end for

 ID_enrollment[i] = ⌊(Qt/q)⌋ mod 2
 Wt[i] = 0.5·q - (Qt-(⌊Qt/q⌋·q))
 i++
end for
return ID_enrollment, Wt



Id map reproduce(F[], Wt[], t, q, e) //Reproduction
i=0
for combo in combinations(F[], t, M)
 // choose t out of M raw results F[j]
 Qt = 0
 for (Fu, Fv) in combinations(combo, 2, t)
    // choose 2 out of t raw results F[j] from combo
```

```
    Qt += |Fu - Fv|^e
  end for

  Qt' = Qt + Wt[i]
  ID_field[i] = ⌊(Qt'/q)⌋ mod 2
  i++
end for
```

return ID

3 PUF Properties

The following are the important properties of PUF. These properties evaluate the quality of PUF.

1. Uniqueness
2. Reliability
3. Entropy

3.1 Uniqueness

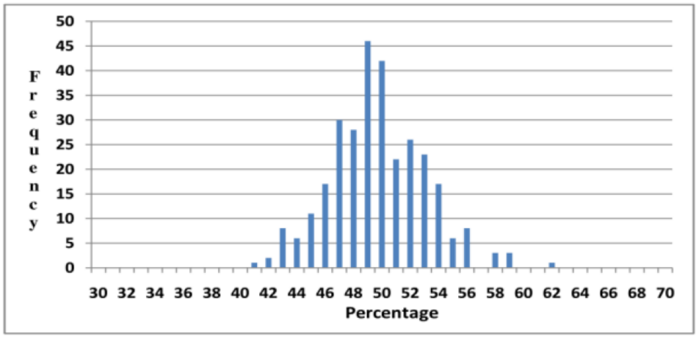
The function named *uniqueness()* determines the Uniqueness property of PUF. The input for this script is the PUF response of more than one chip. This function can be called from *all_schemes.py* by providing the following parameters,

```
uniqueness(inputdata, outputfile, No. of devices, response_bits)
```

Finally, the script generates the uniqueness and stores the result in an output file. This script also generates the Hamming Distances between all the devices. These distances can be used to determine the minimum, maximum or mean inter-chip Hamming distance.

```
Uniqueness starts
N = 2 devices
L = 128 bits
Uniqueness results are written to uniq_result_p.txt
--------------------------------------------------
```

The information of Hamming distance generated by the script can be used to draw the histogram like the one shown below,



Inter-chip Hamming distance

In the above histogram, the x-axis shows the normalized Hamming Distance. While the y-axis (denoted frequency) shows the total number of times a given normalized inter-chip Hamming distance was obtained. In the above figure 25 devices have been used, the total number of combinations (i.e., the total number of board pairs {i,j}) is $\binom{25}{2} = 300$. In ideal case, the normalized inter-chip HD should be 50% and will follow the binomial distribution. It means that 50% PUF output bits are different between PUF A and PUF B.

3.2 Reliability

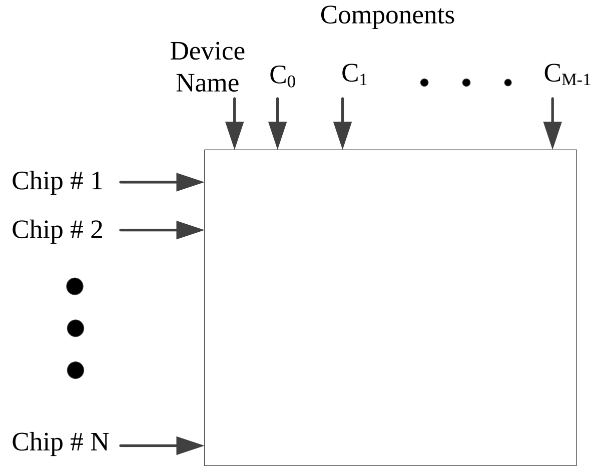
The function named *reliability()* determines the reliability of PUF output under different conditions. The input files consists of the PUF response at room temperature and nominal voltage. On the other hand Field files consist of PUF response at different temperature or voltage conditions in the field. The function can be called from *all_schemes.py* by providing the following parameters,

```
reliability(inputdata_nominal, inputdata_field, output_file, Num_devices_field,
Response_length, directory_name)
```

To determine the reliability of devices at field conditions, the user has to provide the name of directory where the field data is stored, in the *config.py* as shown below,

```
#Directory Name of field input data required for Reliability
field_directory = 'VT_field'
#File Name in the Field Directory at nominal conditions required for Reliability
field_nominal = '1.2V_25C.csv'
```

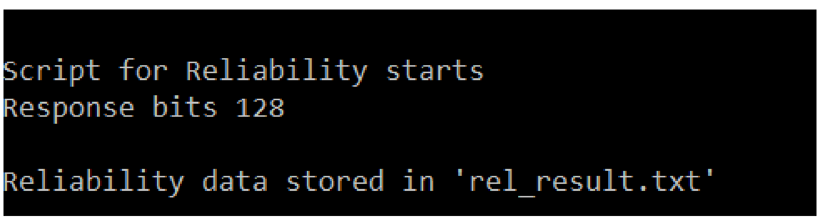
Additionally, the `field_nominal` is the name of file in the directory that contains the data for the same devices under nominal conditions (room temperature…). The format of input data is shown below,



Input Data Format

The list of device names in the first column of all input files in the field directory must be similar. The number of files in the field depends on the number of field conditions. For each condition of voltage and temperature a single csv file is required.

The output of the script shows the number of erroneous bits at each condition. It also shows the average reliability for all conditions. These results are stored in a separate file.



3.3 Entropy

The function named *entropy()* determines the Pairwise Joint Entropy (Worst case Entropy), Average Joint Entropy, Worst case Binary Entropy and Average Binary Entropy of PUF response of all the devices in a data set. The input for this script is the PUF response bits. This script can be called from *all_schemes.py* by providing the parameters like,

```
entropy(inputdata, outputfile, No. of devices, response_bits, directory_name)
```

The outputfile contains the result of entropy.

```
Entropy starts
N =5 devices
Bits per chip = 128
```

3.4 Files and Data Provided:

Below is a list of directories and file that are provided,

- Spartan_field_temperature
- Spartan_field_voltage
- VT_field
- Zynq_field_temperature
- Zynq_field_voltage
- Spartan_Enrolment_Data.csv
- VT_Enrolment_data.csv
- Zynq_Enrolment_data.csv
- User-Guide-Scripts.pdf
- all_schemes.py
- config.py

4 References

1. A. Maiti and P. Schaumont. “Improved Ring oscillator PUF: An FPGA Friendly Secure Primitive,” Journal of Cryptology, volume 24, number 2, pp. 375-397, Oct. 2010.
2. Hori Y, Yoshida T, Katashita T, Satoh A . “Quantitative and statistical performance evaluation of arbiter physical unclonable functions on FPGAs”. *In Proc*. International conference on Reconfigurable computing and FPGAs (ReConFig) 2010, pp 298–303, Dec 2010.
3. R. Maes, A. Herrewege and I. Verbauwhede, “PUFKY: A Fully Functional PUF-based Cryptographic Key Generator,” *In Proc*. Workshop on Cryptographic Hardware and Embedded Systems (CHES), 2012, LNCS vol. 7428, pp. 302-319.
4. R. Maes “Physically Unclonable Functions: Constructions, Properties and Applications”. PhD Thesis, katholiek universiteit Leuven (2012).
5. A. Maiti, I. Kim, and P. Schaumont "A Robust Physical Unclonable Function With Enhanced Challenge-Response Set". IEEE Transactions on Information Forensics and Security, Vol. 7, No. 1, February 2012.
6. D. Ganta and L.Nazhandali. “Easy-to-Build Arbiter Physical Unclonable Function with Enhanced Challenge/Response Set”. *In Proc*. Quality Electronic Design (ISQED), 2013 14th International Symposium on Quality Electronic Design.
7. Maiti, V. Gunreddy, and P. Schaumont. “A Systematic Method to Evaluate and Compare the Performance of Physical Unclonable Functions,” in *Embedded Systems Design with FPGAs*, Springer, 2013, pp. 245-267.
8. B. Habib, J. Kaps and K. Gaj . “Efficient SR-Latch PUF”, *In Proc*. 11th International Symposium on Applied Reconfigurable Computing, 2015, Bochum, Germany, April 15-17. 2015.
9. B. Habib and K. Gaj . "A Comprehensive Set of Schemes for PUF Response Generation," Proc. 12th International Symposium on Applied Reconfigurable Computing, Rio de Janeiro, Brazil, 22-24 March, 2016.