Design of True Random Numbers Generators with Ternary Physical Unclonable Functions

Bertrand Francis Cambou

School of Informatics Computing and Cyber Systems, Northern Arizona University, Flagstaff, 86004, USA

A R T I C L E   I N F O

Article history:
Received: 06 April, 2018
Accepted: 02 May, 2018
Online: 07 May, 2018

Keywords:
Random numbers generators
Physical unclonable functions
Ternary states

A B S T R A C T

Memory based ternary physical unclonable functions contain cells with fuzzy states that are exploited to create multiple sources of physical randomness, and design true random numbers generators. A XOR compiler enhances the randomness of the binary data streams generated with such components, while a modulo-3 addition enhances the randomness of the native ternary data streams, also generated with the same method. Deviations from perfect randomness of these random numbers, in terms of probability to be non-random, was reported as low as $10^{-10}$ in the experimental section of this paper, which is considered as extremely random based on NIST criteria.

1. Introduction

This paper is an extension of the work presented at the annual computing SAI conference, London, July 2017 [1]. The strengthening of cryptographic protocols with random numbers [2-4] is widely accepted as mandatory to secure networks of cyber physical systems (CPS). Both pseudo random numbers generators (PRNG) that use mathematical methods [5-8] and true random numbers generators (TRNG) that exploit physical elements [9-11] are mainstream. The randomness based on mathematical algorithms for PRNGs could be weak when crypto-analysts armed with powerful computers know the algorithms. Such algorithms can also consume too much computing power, which may be a problem for small internet of things (IoT) peripherals. The need to quantify the randomness of PRNG, and TRNG is of prime importance [12-15]. Physical Unclonable functions (PUF) can be valuable sources of natural randomness [16-17], they have been adopted for the design of true random number generators (TRNG). However, the randomness of the physical elements is not always acceptable when subjected to temperature changes, aging, electromagnetic interferences, and other parametric drifts. The PUFs can be too predictable in some circumstances, which is not necessarily conducive to the design of quality TRNGs that rely on physical randomness to generate a fresh random number at every query.

We are presenting how ternary PUFs contain fuzzy elements that are excellent sources of randomness. The method is based on the identification of the cells of memory based PUFs that are naturally unstable under repetitive queries. When tested, the fuzzy cells can switch back and forth randomly between “1” and “0”, thereby generating random data streams. We are presenting three complementary elements:

i) how a XOR data compiler, which process the data available from multiple ternary cells, can create an extremely high level of randomness [18-21];

ii) how a probabilistic model allows the quantification of the level of randomness of the TRNG;

iii) how the method can be extended to the generation of native ternary random numbers with modulo-3 addition.

2. Designing a random number generator

2.1. Ternary physical unclonable functions

There is a growing interest in securing CPS’s with Physically Unclonable Functions (PUFs) to strengthen security when deployed in the cryptographic processes using a powerful set of physically derived cryptographic primitives [22-26]. PUFs act as virtual fingerprints for the hardware during the authentication processes to effectively block cyber thefts, Trojans, and malwares [27-34]. With error correcting methods, the PUFs can also generate cryptographic keys for symmetrical encryption schemes [35-36]. The inherent randomness, unclonability, secrecy, and physical nature of most PUFs makes it extremely hard to inspect during side channel attacks, or when lost to the enemy.

2.2. Quality considerations for PUFs

The PUFs, regardless of their design, must exhibit enough predictability overtime for reliable authentications, or encryption. The reference patterns of the PUFs that are
generated up front during the setup of protocol, called challenges, are compared over the life of the component with freshly generated patterns, called responses, during the authentication cycles. Quality PUFs need low challenge-response-pair (CRP) error rates, the intra-PUF challenge-response Hamming distance must be small enough to insure small level of false rejection rates (FRR). The PUF challenges should act as predictable “digital fingerprints” of the component, while the responses should be easily recognizable as a measurement of the same “digital fingerprints”. Error rates in the 3-7% range are usually acceptable when combined with error correcting techniques [35-36]. PUFs exploit the device-to-device randomness that is created during the manufacturing process of micro-components; it is desirable that the average inter-PUF hamming distance between different PUFs, divided by the length of the PUFs, should be in the 50% range to insure low level of false acceptance rates (FAR). This is achievable when the level of intra-PUF randomness, also called entropy, is high enough. PUFs with longer streams of bits have therefore higher entropy, and lower FAR. 128-bit CRPs, or higher, are usually required for this purpose.

Another important figure of merit for PUFs is the number of available CRP configurations for a unit. Strong PUFs, as opposed to weak PUFs, contains large quantities of possible CRPs that are addressable. For example, a ring oscillator PUF with 128 rings is a strong PUF. The number of possible pairing of two rings is \( N = \left(\frac{128}{2}\right) = 16,256 \). If the protocol use 128-bit long CRPs, the number of possible challenges of \( 2^{128} \), offers satisfactory entropy, and a low collision rate of the pairs. A memory PUF with random addressing capabilities is even stronger [36, 39-40]. For example, when the capacity of the memory is in the mega-byte range, millions of configurations are providing an entropy much higher than a 128-ring oscillator with “only” 16,256 possible configurations. Existing PUFs can have limitations, and lack of trustworthiness that could create a false sense of security. The signatures of PUFs are derived from intrinsic manufacturing variations, which could become predictable due fabrication excursions. Properties such as critical dimensions of printed structures, doping levels of semiconducting layers, and threshold voltages should make each device unique and identifiable from all other devices, abnormal operations during the manufacturing process could alter such randomness. When subject to changes related to temperature, voltage, EMI, aging, and other environmental factors these parameters can drift over time, the undesirable result, is weak PUFs with CRP error rates as high of 20%.

The main objective in designing ternary PUFs is to resolve some of these issues, and to reduce the CRP error rates by eliminating fuzzy CRPs during challenge generation. The figure of merit is to achieve trustworthy and robust intra-PUF CRP matching rates with low FRR during authentications, without increasing FAR during inter-PUF authentication of malicious challenges. The by-product of such design is the design of highly random TRNG with the fuzzy cells.

2.3. Memory based PUFs

The methods to design PUFs and TRNGs with SRAM memories have been published SRAM [36-38], SRAM based PUFs have been successfully commercialized. When powered up, each SRAM cells naturally flip to store either a 0, or a 1. In most of the cases, arrays of SRAM cells return to a similar pattern characteristic, i.e. a similar fingerprint. SRAM based PUFs designed with this feature can be reliable, however heavy error correcting methods are usually needed. The SRAM based PUFs are not particularly immune to side channel attacks. Significant research efforts have been published regarding the design of PUFs with Flash RAMs [39-40], DRAMs [41-44], magnetic RAMs [45-46], and resistive RAMs [47-49]. The cryptographic protocols leveraging memory PUFs are in general distinct from the ones developed with other mainstream PUFs such as ring oscillators, or gate delay arbiters. As shown in Fig. 1, the value of a parameter \( P \) is measured on each cell, and is compared with a threshold. The cells with parameter \( P \) below the threshold are “0”s, and are “1”s above the threshold. Examples of parameter P selected to design memory PUFs include: threshold voltages of Flash cells after fixed time programming; charges left on DRAM cells without refresh; high resistance value of MRAM cells after programming; and Vset of ReRAM cells.

The CRP matching is done after error correction. The cell-to-cell physical parameter variations due to manufacturing variations are too erratic for CRP generation. Fig.2 is a diagram showing how a drift of \( P \) toward the higher value is forcing the responses of the cells located close to the threshold to switch from 0 to 1, which increase CRP error rates. The cells located far from the transition are not impacted.

2.4. Ternary PUFs

The concept of memory based PUFs with ternary states having random number generation capabilities is described [49-
52]. The measurement of $\mathcal{P}$ of the cells of a memory PUF allows the segmentation of the cell population into three states. The cells with $\mathcal{P} < T_1$ (a low threshold) carry the state “-”, the cells with $\mathcal{P} > T_2$ (a high threshold) carry the state “+”, and the remaining cells carry the ternary state “0” cells, see Fig.3.

During challenge generation, the cells are sorted into ternary states. During response generation, only the cells with “-” or “+” states are queried, while the cells with “0” state are ignored, see Fig.4. The PUF CRP error rates are significantly lowered, the distance $T_2 - T_1$ acts as a buffer between the states “-”, and “+”. When the distance $T_2 - T_1$ between thresholds increases, the CRP error rate can reach extremely low values, and is less sensitive to various drifts.

3. Random number generators

3.1. Pseudo Random number generators

There are numerous excellent PRNG available to the system developers, which are highly reliable [4-8]. For example, a PRNG $\{a_1, a_2, ..., a_n\}$ can be designed with congruential generators, where $a$ is the multiplier, $c$ the increment, $m$ the modulus, and $X_i$, $b$, $c$, $m$ are natural numbers, typically, $c$ and $m$ are chosen to be relatively prime:

$$a_{i+1} = (b \cdot a_i + c) \mod m$$  \hspace{1cm} (1)

Other example of PRNG can be constructed by using iterative encryption, as shown in Fig 5, $a_{i+1}$ is the cipher of $X_i$ which is encrypted by the code E, and the key $K_i$. Proving that a PRNG or a TRNG is “random” is a very complicated task that could take years to validate, and billions of data points. The National Institute of Standard and Technology (NIST) has developed an excellent suite of tools available on line that can test the randomness of any random numbers generators [12-15].

Examples of parameters that are tested include deviation from randomness, a frequency test (monobit test), Serial test (two-bit test), a Poker test (non-overlapping parts), run tests (gap and blocks), and autocorrelation tests [Menezes, van Oorschot, Vanstone - Handbook of Applied Cryptography].

In this paper, we are using statistical analysis to quantify randomness, and the parameter $\lambda$ defined below in this section. Each bit “$a_i$” of a data stream of $n$ bits should have a 50% probability to be either a “1” or a “0”. The average deviation from perfect randomness $\lambda$ is given by:

$$P|0| = P|1| = 0.5$$  \hspace{1cm} (2)

$$\lambda = \frac{P|0| - P|1|}{0.5}$$  \hspace{1cm} (3)

Assuming that the length of the data stream is $n = 128$, with $P|0| = 0.5$, the number of possible combinations, also called entropy, is $2^{128} = 3.4 \times 10^{38}$, which is large enough to protect cryptographic functions from existing or foreseeable computers. When the RNG is not totally random, in this case $\lambda \neq 0$, the entropy is lower than $2^{128}$, and is further reduced with larger $\lambda$. A position paper from (NIST) [12], suggested in 1999 that $\lambda$ greater than $10^3$ would not be acceptable, sophisticated crypto-analysis methods could be effective to break the PRNG. NIST in 2010 and others [33-34] revisited this. The value of $\lambda$ that is acceptable to get a safe TRNG is a moving target as modern computers get increasingly powerful. To the best of our knowledge, $\lambda < 10^5$ is currently considered an excellent target, while $\lambda < 10^{10}$ is considered outstanding.

3.2. Use of XOR to enhance PRNGs

Exclusive OR, XOR, is a Boolean logic gate widely adopted in cryptography [18-21]. Two input bits $a_i$ and $a_{i+1}$ are transformed into $c_i = a_i \oplus a_{i+1}$, with the following equations:

$$c_i = 0 \quad \text{if} \quad a_i = a_{i+1} \quad (0 \oplus 0 \text{ or } 1 \oplus 1)$$  \hspace{1cm} (4)

$$c_i = 1 \quad \text{if} \quad a_i \neq a_{i+1} \quad (0 \oplus 1 \text{ or } 1 \oplus 0)$$  \hspace{1cm} (5)

XOR logic is part of most encryption algorithms such as the Data Encryption System (DES), the Advanced Encryption System (AES), and hash functions such as SHA. XOR functions are adding confusion and randomization in the encryption process while been reversible in the decryption process. As part of the encryption, the data streams generated by plain texts are
often XORed with cryptographic keys, or sub-keys, then XORed again during decryption. XOR scramblers can enhance randomization in multicarrier communications [19]. XOR are also used to generate scrambling sequences to achieve data randomization in a memory circuit, as well as enhancing random number generators [20]. Some of the important reasons for the use of XOR functions in cryptography are:

- $c_i$ is not disclosing the value of $a_i$ and $a_{i+1}$:
  - $c_i = 0$ can be the result of the pair 00, or the pair 11;
  - $c_i = 1$ can be the result of the pair 01, or the pair 10;
- XOR is a symmetrical function when applied twice:
  \[
  a_i \oplus a_{i+1} \oplus a_{i+2} = a_i
  \]
- If two bits $a_i$ and $a_{i+1}$ are random, the bit $c_i$, defined by
  \[
  c_i = a_i \oplus a_{i+1},
  \]

These properties are exploited in the design of the XOR data compiler as presented in section 4.

3.3. Ternary PUFs as sources of randomness

The cells of a ternary PUF with “0” state, as described in section 2.3, are exploited as sources of randomness to design TRNG [1, 17], as explained in Figure 6. The cells located in the center of the distribution, the “0” states, can flip back and forward when the value of their parameter $P$ is compared to a threshold centered in the median point of the distribution.

When the variations within cells are much lower than the cell-to-cell variations, the “finger print” of the memory PUF is stable and predictable. On the opposite side, to design a TRNG, it is desirable to select only the cells extremely close to the transition of parameter $P$ between “0” and “1”, i.e. the one with ternary state “0”. If $T_M$ is the median of the distribution, the average value $Tx$ of $P$ of each -cells should be such that:

\[
|Tx - T_M| < \sigma_{cell}
\]

This maximizes the chance of a random number to flip between “0s” and “1s”. In order to enhance the level of randomness only a very small percentage of the memory arrays are selected as sources of randomness. Current secure microcontrollers have very large embedded memory density, typically in the 1 to 100 Mbits, the percentage of the array consumed for TRNG can be relatively small. In the following sections, we are developing a statistical model to study how to enhance the randomness of a data stream generated from the fuzzy cells. One of the tradeoffs to model is the compromise between tightly selecting the “0” cells around the median $T_M$, versus improving randomness; in the case of the generation of native ternary streams, we study the use of modulo3 adders.

4. Modeling a ternary PUF for TRNG

As shown in Fig 7, the cells that are sorted as unstable with a “0” state can be segmented into two subgroups:

- The cells that have a higher probability to be tested above the median are called A-cells, see Fig 8. They have an average probability $P_A$ to generate a “1” in the stream of random numbers, their average deviation to randomness is $\lambda_A$. The A cells have an average probability $P'_A$ to generate a “0” in the stream of random number:
  \[
P_A = 0.5 + \lambda_A
  \]
  \[
P'_A = 0.5 - \lambda_A; \quad 1 = P'_A + P_A
  \]

- The cells that have a higher probability to be tested below the median are called B-cells, see Fig 9. They have an average probability $P_B$ to generate a “0”, and an average deviation to randomness $\lambda_B$. The average probability $P_B$ to be generate a “1” is:
  \[
P_B = 0.5 + \lambda_B
  \]
  \[
P'_B = 0.5 - \lambda_B; \quad 1 = P'_B + P_B
  \]

The selection of the transition $T_M$ of parameter $P$ can be such that the number of A-cells equal the number of B-cells and:

\[
P_A = P'_B, \quad P'_A = P_B, \quad \lambda_A = \lambda_B.
\]

When the distance $T_2 - T_1$ between the two thresholds used to select the cells with “0” states is reduced, the probability to test these cells either as below the median, or above the median at each query is closer to 50%. For example, the selection of 1,000 cells located close to the median will represent a strong pool to design TRNG. These 1,000 cells can be queried many times to generate long random numbers. Each cell acts as a single source of independant randomness subject to noise, and measurement uncertainties. Within the cells of a particular memory array, the distribution of the physical parameter $P$, which determine if a cell is a “0” or a “1”, is following a distribution with a standard variation $\sigma_{array}$ due to cell-to-cell variations created during manufacturing, and other instabilities. Repetitive measurements of parameter $P$ on the same cell follow a distribution with the standard variation $\sigma_{cell}$ responding to various measurement instabilities, noise, and environmental variations. Low error rate PUFs, with predictable CRPs, should have these variations verifying:

\[
\sigma_{cell} < \sigma_{array}
\]
This stream is grouped in chunks of \( f \) bits, \( i \in \{1 \text{ to } f \} ; f < n \).

**Chunk of bits:** \[ \{a_{i1}, a_{i2}, ..., a_{in}, a_{if}\} \quad (15) \]

For example, 1,280 random bits are grouped in 128 chunks of 10 bits. With a XOR, the stream \( c_j \), with \( j \in \{1 \text{ to } m\} \) and \( n=m.f \), is generated from the stream \( a_i \), as shown in Fig. 10.

**Out coming stream:** \[ \{c_1, c_2, ..., c_p, ..., c_m\} \quad (16) \]
\[ c_j=a_{j1}a_{j2}a_{j3}...a_{jnf} \quad (17) \]

Such a XOR compiler can be implemented in hardware with only a few logic gates which can be inserted as part of the crypto-processor of the secure processor. The PRNGs presented section 3.1, are generated sequentially, the random number \( a_{i+1} \) of a stream of \( n \) bits is generated from the previous random numbers \( a_i \). Conversely, the TRNGs with XOR gates can be generated in parallel eq (17) in one cycle. XOR gate is also an addition modulo 2 without carry over. A quicker way to compute eq (17) is to count how many “1s” are present in the stream \( \{a_{ij}, a_{ij}, ..., a_{ij}\} \). If the number of “1s” in the stream is odd then \( c_j=1 \), when even \( c_j=0 \).

\[ c_j= a_{ij}a_{ij}a_{ij}...a_{ij} \; \text{mod} \; 2 \quad (18) \]

### 6. Modeling a 2-bit XOR compiler

We analyze a 2-bit XOR compiler, the incoming data stream of \( 2n \) bits is “XORed” two bits by two bits to generate a stream of \( n \) bits, \( f=2 \). There are three possible configurations for each “XORing”: both cells are A-cells, one cell is an A-cell and the second is a B-cell, and both cells are B-cells. Let us choose: \( P_A=P_B=0.52 \); \( P_A'=P_B'=0.48 \); \( \lambda_A=\lambda_B=2 \times 10^{-2} \).

\[ \rightarrow \text{Number of A-cells is even: two A-cells, or two B-cells} \]

The probability \( P'c \) to have \( c_j=a_{ij}a_{ij}a_{ij} \) at “0” is occurring when the two cells \( a_{ij}, a_{ij} \) are at (00) or (11):

\[ P'c = P_A^2 + P_B^2 = 0.5008 \; \rightarrow \; \lambda_c = 8 \times 10^{-4} \quad (19) \]

The probability \( P_c \) to have \( c_j=a_{ij}a_{ij}a_{ij} \) at “1” is occurring when the two cells \( a_{ij}, a_{ij} \) are at (01) or (10):

\[ P_c = 2 (P_A P'_A) = 0.4992 \quad (20) \]

\[ \rightarrow \text{Number of A-cells is odd: one A-cells, and one B-cell.} \]

The probability \( P_c \) to have \( c_j=a_{ij}a_{ij}a_{ij} \) at “1” is occurring when the two cells \( a_{ij}, a_{ij} \) are at (01) or (10):

\[ P'_c = P_A P_B^2 = 0.5008 \; \rightarrow \; \lambda_c = 8 \times 10^{-4} \quad (21) \]

The probability \( P'_c \) to have \( c_j=a_{ij}a_{ij}a_{ij} \) at “0” is occurring when the two cells \( a_{ij}, a_{ij} \) are at (00) or (11):

\[ P'_c = 2 (P_A' P_A') = 0.4992 \quad (22) \]

Let us assume that the incoming stream with \( 2n \) bits is generated from a memory PUF with 50% A-cells and 50% B-cells, and with \( \lambda_A=2 \times 10^{-2} \). The 2-bit XOR compiler can statistically generate an out coming stream of \( n \) bits having 50% C-cells, and 50% D-cells with \( \lambda_c = 8 \times 10^{-4} \), see Fig. 11. The C-cells are made of pairs of either AA cells or BB cells, while the D-cells are made of pairs of either AB cells or BA cells. In both cases, \( f=2 \) is even. The general equations developed below in section 8, eq. (29) to (39) are applicable. When the number of B-cell is even \( P_c < P'_c \), and are reversed when the number of B-
cell is odd $\mathbf{P}_c > \mathbf{P}'c$. The deviation from randomness $\lambda_c=8 \times 10^{-4}$ is 25 times smaller than the deviation before the 3-bit-XOR compilation, $\lambda_A=2 \times 10^{-2}$.

![Diagram showing a 2-bit XOR compiler.](image)

**Figure. 11: Diagram showing a 2-bit XOR compiler.**

7. **Modeling a 3-bit XOR compiler**

In this section we analyze a 3-bit XOR compiler, in which the incoming data stream of $3n$ bits is “XORed” three bits by three bits to generate a stream of $n$ bits, $f=3$. We are again choosing the same example:

$$\mathbf{P}_A = \mathbf{P}'_B = 0.52; \mathbf{P}'_A = \mathbf{P}_B = 0.48; \lambda_A = 2 \times 10^{-2}.$$  

There are four possible configurations for each “XORing”: three cells are A-cells, two cells are A-cells & one cell is B-cell, one cell is A-cell & two cells are B-cells, and finally three cells are B-cells.

➢ **Number of B-cells is even:** three A-cells, A-cell & two B-cells. The probability $\mathbf{P}_c$ to have $c_{ij}=a_{ij} \oplus a_{j1} \oplus a_{j2}$ at “1” occurring when the three cells $(a_{ij}, a_{j1}, a_{j2})$ at (111), (101), (011) or (001):

$$\mathbf{P}_c = \mathbf{P}_A^3 + 3 \mathbf{P}_A^2 \mathbf{P}_A^2 = 0.500032 \Rightarrow \lambda_c = 3.2 \times 10^{-5}$$  

The probability $\mathbf{P}'_c$ of having $c_{ij}=a_{ij} \oplus a_{j1} \oplus a_{j2}$ at “0” occurring when the three cells $(a_{ij}, a_{j1}, a_{j2})$ are at (000), (100), (110) or (011):

$$\mathbf{P}'_c = \mathbf{P}'_A^3 + 3 \mathbf{P}'_A^2 \mathbf{P}'_A^2 = 0.499968$$  

➢ **Number of B-cells is odd:** three B-cells, B-cell & two A-cells. The probability $\mathbf{P}_c$ to have $c_{ij}=a_{ij} \oplus a_{j1} \oplus a_{j2}$ at “0” occurring when the three cells $(a_{ij}, a_{j1}, a_{j2})$ are at (000), (100), (110) or (011):

$$\mathbf{P}_c = \mathbf{P}_A^3 + 3 \mathbf{P}_A^2 \mathbf{P}_A^2 = 0.500032 \Rightarrow \lambda_c = 3.2 \times 10^{-5}$$

The probability $\mathbf{P}'_c$ of having $c_{ij}=a_{ij} \oplus a_{j1} \oplus a_{j2}$ at “0” occurring when the three cells $(a_{ij}, a_{j1}, a_{j2})$ are at (000), (110), (011) or (101):

$$\mathbf{P}'_c = \mathbf{P}'_A^3 + 3 \mathbf{P}'_A^2 \mathbf{P}'_A^2 = 0.499968$$  

Let us assume that the incoming stream with $3n$ bits is generated by a memory PUF having 50% A-cells, and 50% B-cells, and with $\lambda_A=2 \times 10^{-2}$. The 3-bit XOR compiler can statistically generate an out coming stream of 128 bits having 50% C-cells, and 50% D-cells with $\lambda_c=3.2 \times 10^{-5}$, see Fig. 12. The C-cells are made of triplets of either AAA cells, ABB cells BAB cells, or BBA cells. The D-cells are made of triplets of either AAB cells, ABA cells, BAA cells, or BBB cells.

In both cases, $f=3$ is odd. The general equations developed in the next section, eq. (29) to (38) are applicable. When the number of B-cells is even, $\mathbf{P}_c < \mathbf{P}'_c$, and are reversed when the number of B-cells is odd, $\mathbf{P}_c > \mathbf{P}'_c$. The resulting deviation from randomness, $\lambda_c=3.2 \times 10^{-5}$, is 25x25=625 times smaller than the deviation before the 3-XOR compilation, $\lambda_A=2 \times 10^{-2}$. It is interesting to notice that a 3-bits XOR data compiler needs only 50% more starting cells than a 2-bits compiler, and has a level of non-randomness 25 times lower.

![Diagram showing a 3-bits XOR compiler.](image)

**Figure. 12: Diagram showing a 3-bits XOR compiler.**

8. **Modeling the XOR compiler in general terms**

The goal is to develop a model that quantifies the effect of a XOR compiler, which enhance the level of randomization of a data stream, as a function of the size $f$ of the chunk of incoming bits that are XORed together. The incoming stream $\{a_1, \ldots, a_n\}$ has a deviation from randomness $\lambda_A$, and the out coming stream $\{c_1, c_2, \ldots, c_n\}$ has a deviation $\lambda_c$. This variation is obtained by computing $\mathbf{P}_c$, the probability for $c_i$ to be a “1”:

$$| \lambda_c | = 1 - \mathbf{P}_c$$  

The incoming random bits $a_{ij}$ are generated from A-cells or B-cells of the ternary PUFs. As stated in section 4.1, the transition $T_m$ is selected in such a way that the probability to have an A-cell, and a B-cell is equal to 0.5. If each of the $f$ long chunks have $s$ A-cells and $t$ B-cells with $s+t=f$. The numbers of possible combinations $(f, s)$ is:

$$C_{ts} = \binom{f}{s} = f! / s! (f-s)!$$  

1.1 All cells of chunk $j$ are A-cells.

The probability of any of the A-cells of the stream to be a “1” is $\mathbf{P}_A$, and the probability to be a “0” is $\mathbf{P}'_A$. Both $\mathbf{P}_A$ and $\mathbf{P}'_A$ are following Bernoulli formula:

$$1 = \sum_{i=0}^{f} \binom{f}{i} \mathbf{P}_A^i \mathbf{P}'_A^{f-i}$$  

$$1 = \sum_{i=0}^{f} \binom{f}{i} \mathbf{P}_A^i \mathbf{P}'_A^{f-i} + \sum_{i=0}^{f} \binom{f}{i} \mathbf{P}_A^{i+1} \mathbf{P}'_A^{f-i} + \sum_{i=0}^{f} \binom{f}{i} \mathbf{P}_A^i \mathbf{P}'_A^{f-i+1}$$  

$$1 = \sum_{i=0}^{f} \binom{f}{i} \mathbf{P}_A^i \mathbf{P}'_A^{f-i}$$

Let us assume that the incoming stream with $3n$ bits is generated by a memory PUF having 50% A-cells, and 50% B-cells, and with $\lambda_A=2 \times 10^{-2}$. The 3-bit XOR compiler can statistically generate an out coming stream of 128 bits having 50% C-cells, and 50% D-cells with $\lambda_c=3.2 \times 10^{-5}$, see Fig. 12. The C-cells are made of triplets of either AAA cells, ABB cells BAB cells, or BBA cells. The D-cells are made of triplets of either AAB cells, ABA cells, BAA cells, or BBB cells.

The probability $\mathbf{P}_c$, is the sum of all terms having $i$ odd:

$$\mathbf{P}_c = \sum_{i=0}^{f} \binom{f}{i} \mathbf{P}_A^{i+1} \mathbf{P}'_A^{f-i}$$

$$\mathbf{P}_c = \sum_{i=0}^{f} \binom{f}{i} \mathbf{P}_A^i \mathbf{P}'_A^{f-i}$$
If \( f = 2k \) is even:
\[
P_c = \sum_{i=0}^{k-1} \binom{2k}{2i+1} \binom{2k}{2i}^2 \left( \frac{P'}{P} \right)^{2i} \left( 1 - \frac{P'}{P} \right)^{2k-2i-1} \tag{33}
\]

If \( f = 2k+1 \) is odd:
\[
P_c = \sum_{i=0}^{k} \binom{2k}{2i+1} \binom{2k}{2i}^2 \left( \frac{P'}{P} \right)^{2i} \left( 1 - \frac{P'}{P} \right)^{2k-2i-1} \tag{34}
\]

The probability \( P_c' \) is the sum of all terms having \( i \) even:
\[
P_c' = \sum_{i=0}^{\left\lfloor \frac{f+1}{2} \right\rfloor} \binom{2k}{2i} \binom{2k}{2i}^2 \left( \frac{P'}{P} \right)^{2i} \left( 1 - \frac{P'}{P} \right)^{2k-2i-1} \tag{35}
\]

If \( f = 2k \) is even:
\[
P_c' = \sum_{i=0}^{k} \binom{2k}{2i} \binom{2k}{2i}^2 \left( \frac{P'}{P} \right)^{2i} \left( 1 - \frac{P'}{P} \right)^{2k-2i-1} \tag{36}
\]

If \( f = 2k+1 \) is odd:
\[
P_c' = \sum_{i=0}^{k} \binom{2k+1}{2i+1} \binom{2k}{2i}^2 \left( \frac{P'}{P} \right)^{2i} \left( 1 - \frac{P'}{P} \right)^{2k+1-2i-1} \tag{37}
\]

When \( f \) is even, \( P_c < P_c' \) is written as \( P_c = 0.5 + \lambda c_j \) or \( P_c' = 0.5 + \lambda c_j \), with \( \lambda c_j \) the deviation from randomness of \( c_j \).

When \( f \) is odd, \( P_c > P_c' \) and is written as \( P_c = 0.5 + \lambda c_j \) or \( P_c' = 0.5 - \lambda c_j \).

### 1.2 Chunks \( j \) are a combination of A-cells & B-cells

The \( f \) cells randomly contain A-cells and B-cells. The symmetry between the A-cell and the B-cells (\( P_A = P_B \) and \( P_A = P_B \)) results in the following property:

- If the chunk of bits \( \{a_1, a_2, ..., a_f\} \) is generated by an even number of B-cells, the probabilities \( P_c \) and \( P_c' \) are the same as if the chunk was only generated by A-cells. If \( f \) is even, \( P_c \) and \( P_c' \) are respectively computed with eq. (33) and (37); if \( f \) is odd, \( P_c \) and \( P_c' \) are computed with eq. (34) and (38).

- If the chunk of bits \( \{a_1, a_2, ..., a_f\} \) is generated by an odd number of B-cells, the probabilities \( P_c \) and \( P_c' \) are the opposite of the ones generated by A-cells as described by eq. (37) (33) and eq (38) (34):

If \( f \) is even, \( P_c \) is (eq. (37)): \[
P_c = \sum_{i=0}^{k} \binom{2k+1}{2i+1} \binom{2k}{2i}^2 \left( \frac{P'}{P} \right)^{2i} \left( 1 - \frac{P'}{P} \right)^{2k+1-2i-1}, \text{ and}
\]
\[
P_c' = \sum_{i=0}^{k} \binom{2k+1}{2i+1} \binom{2k}{2i}^2 \left( \frac{P'}{P} \right)^{2i} \left( 1 - \frac{P'}{P} \right)^{2k-2i-1} \tag{38}
\]

When \( f \) is odd, \( P_c \) is (eq. (38)): \[
P_c = \sum_{i=0}^{k} \binom{2k+1}{2i+1} \binom{2k}{2i}^2 \left( \frac{P'}{P} \right)^{2i} \left( 1 - \frac{P'}{P} \right)^{2k+1-2i-1}, \text{ and}
\]
\[
P_c' = \sum_{i=0}^{k} \binom{2k+1}{2i+1} \binom{2k}{2i}^2 \left( \frac{P'}{P} \right)^{2i} \left( 1 - \frac{P'}{P} \right)^{2k-2i-1} \tag{39}
\]

### 1.3 Simplification of the model

The objective of this model is to calculate the absolute deviation from perfect randomness, it is not important to know if \( P_c > P_c' \), or if \( P_c' > P_c \). In all cases, \( 1/k \lambda j \) is the statistical deviation from pure randomness, regardless of \( P_c \) being greater or lower than \( P_c' \). Therefore, assuming that all cells are A-cells is simplifying the computation without reducing the accuracy of the model.

### 9. Experimental analysis with XOR compiler

#### 9.1. Variations of ReRAM memory PUFs

The experimental data presented in this paper is based on the study of resistive random-access memory (ReRAM). The cells of ReRAMs, see references [53-60], are constructed with stacks of two electrodes separated by solid electrolytes, the first one is active to REDOX cycles, and the second one is inert. As shown in Figure 13, differential voltages applied on these stacks can move positively, or negatively, elements such as positive oxygen vacancies or positive metallic cations, which result in varying the resistance of the stacks. The basic physical effect described in Fig. 13, can be achieved with several manufacturing technologies:

- Conductive bridge random access memories (CBRAM) that are based on the conduction of cations such as Ag+, or Cu+ through solid chalcogenide electrolytes, or porous silicon [53-58]. The active electrodes could be made of copper, or silver, while the inert electrode can be fabricated with tungsten;

- Memristors devices can operate as ReRAM, or act as active Boolean gates [59-60]. The conductive filaments usually contain oxygen vacancies. The solid electrolyte can be fabricated with HfO, or TaOx.

In this work, we had access to Cu/TaOx/Pt resistive crossbar arrays fabricated on thermally oxidized Si wafers, Reference [38]. The Cu/TaOx/Pt switches from “0” to “1” based on the formation and the rupture of filaments, made of oxygen vacancies, bridging the dielectric between both electrodes. The initial conditioning of the ReRAM cells, in which conductive filaments are formed, typically requires a positive voltage of approximately 2 to 5 Volt. After forming, the cells can respond to programming and erasing cycles. It exists a minimum negative Vset voltage applied across the cells, in the -0.5 to -3.0 Volt range, that force the positive ions or oxygen vacancies to migrate back, breaking the conductive filament. The resulting high resistance state (HRS) is then in the 20 Mohm range. In the positive direction, a minimum Vset voltage applied across the switch, reposition the positive ions or oxygen vacancies, forming again the conductive filament. As shown in Fig. 14, when the voltage is ramping, the current remains low until Vset is reached, then the current quickly increases. This effect is reversible, and the filaments can partially be dissolved with opposite voltages.

The parameter \( P \), that is analyzed for the purpose of designing TRNGs, is the distribution of the Vset across the cells of ReRAM arrays. The entire population of all cells of the array has a Vset distribution that is well represented by a normal distribution having a standard deviation \( \sigma_{\text{array}} = 0.5 \text{V} \) and a median value of 2.1V. The repetitive measurement of the Vset of each cells is also well represented by a normal distribution having a standard deviation \( \sigma_{\text{cell}} = 0.1 \text{V} \).

For the purpose of random number generation, the Vset of each cell is measured; a cell is considered as a “0” state when
Vset<2.1V, and a “1” state when Vset>2.1V. The cells having average Vset measurements at or close to 2.1 Volt, are good candidates for TRNG. The five populations described below are subsets of the total distribution of cells of the array:

- **Case-1:** Only 2% of the cells are the ternary 0-states. They are used to generate the random numbers. For these cells parameter $\mathcal{P}$ is close to the transition of 2.1 Volt. Half of the cells, the A-cells, have $P_A=0.52$ probabilities to be “1”, $P'_A=0.48$ probabilities to be “0”, with $\lambda_A=2 \times 10^{-2}$. The second group, the B-cells, have $P_B=0.48$ probabilities to be “1”, $P'_B=0.52$ probabilities to be “0”, with $\lambda_B=\lambda_A=2 \times 10^{-2}$.

- **Case-2:** 4% of the cells are ternary 0-states. The probabilities as defined above are:

  $P_A=P'_B=0.54$; $P'_A=P_B=0.46$; $\lambda_B=\lambda_A=4 \times 10^{-2}$

- **Case-3:** 7% of the cells are ternary states. The probabilities as defined above are:

  $P_A=P'_B=0.56$; $P'_A=P_B=0.44$; $\lambda_B=\lambda_A=6 \times 10^{-2}$

- **Case-4:** 11% of the cells are ternary states. The probabilities as defined above are:

  $P_A=P'_B=0.60$; $P'_A=P_B=0.40$; $\lambda_B=\lambda_A=1 \times 10^{-1}$

- **Case-5:** 100% of the cells are included. The probabilities as defined above are:

  $P_A=P'_B=0.90$; $P'_A=P_B=0.10$; $\lambda_B=\lambda_A=4 \times 10^{-1}$

In this last case, there are no ternary states, the entire memory array is used to generate random numbers. The reason we are considering this range of options is to quantify the effectiveness of the XOR data compiler to generate a random number as a function of how tight the ternary state distribution is. Case-1 is the one with the highest initial randomness, while Case-5 is the lowest one.

### 9.2. Effect of the XOR compiler on the TRNG

The probabilistic model presented in this section is used to analyze the five experimental cases presented above. Fig 15, and Fig. 16 summarize the impact of the XOR data compiler when $f$ varies from 2 to 5. We are observing a lack of efficiency of the XOR compiler in case-5, the one without ternary states. The lack of initial randomness of this case is such that the XORing cannot “clean up” the stream. In other cases, the XOR data compiler when combined with the ternary 0-states is very efficient. Case-1 with the highest level of initial randomness is benefiting the most from the XOR compiler: with 5-cell XOR, $\lambda=5.12 \times 10^8$, which is a very small deviation from absolute randomness.

| Case | % of 0-cells | $\lambda_A$ (Initial) | $\lambda_A$ (2-bit XOR) | $\lambda_A$ (3-bit XOR) | $\lambda_A$ (4-bit XOR) | $\lambda_A$ (5-bit XOR) |
|------|-------------|----------------------|------------------------|------------------------|------------------------|------------------------|
| 1    | 52% - 48%   | 2 $\times 10^{-2}$   | 8 $\times 10^{-4}$     | 1.28 $\times 10^{-6}$  | 5.12 $\times 10^{-8}$  |
| 2    | 54% - 46%   | 4 $\times 10^{-2}$   | 3.2 $\times 10^{-3}$   | 2.15 $\times 10^{-5}$  | 1.64 $\times 10^{-6}$  |
| 3    | 56% - 44%   | 6 $\times 10^{-2}$   | 7.2 $\times 10^{-3}$   | 1.04 $\times 10^{-5}$  | 1.2 $\times 10^{-5}$   |
| 4    | 60% - 40%   | 11%                  | 1 $\times 10^{-1}$     | 4 $\times 10^{-3}$     | 8 $\times 10^{-4}$     | 1.6 $\times 10^{-4}$   |
| 5    | 90% -10%    | 100%                 | 4 $\times 10^{-1}$     | 3.2 $\times 10^{-1}$   | 2.38 $\times 10^{-1}$  | 2.05 $\times 10^{-1}$  |

### 10. Example of algorithms for TRNG

#### 10.1. Minimization of the impact of parameter drifts

The randomness of the TRNG originates from the physical parameters of multiple cells that provide independent sources of physical randomness. This is a fundamental strength compared with mathematically generated pseudo RNG (PRNG) because mathematical algorithms cannot describe unclonable physical elements. The cell-to-cell randomness is due to micro-variations during manufacturing and natural noise effect during measurements. However, physical elements can vary often in a predictable way when subject to effects such as temperature change, biasing conditions, and induced attacks. For example, the value of the Vset of a resistive RAM goes down when subject to higher temperature. A hacker could submit the physical element to a hot air blower to increase temperature,
reduce Vset, thereby making both A-cells and B-cells appear similar, creating a high probability to be tested as “0”. Such a drift or malicious attack could result in a collapse of the level of randomness with lower entropy. The remedy of such an attack is to make the size of the population of A-cells and B-cells equivalent, by adjusting the threshold (T2) between the “0” states, and “1” states, in the median point of the 0-cell distribution, see Fig 17:

![Algorithm to reduce the effect of a parameter drift.](image)

**Figure. 17**: Algorithm to reduce the effect of a parameter drift.

1. Identify ni cells of the memory PUF that are part of the fuzzy 0-cells; ni=m ∗ f, in preparation of the f-bit XOR compiler;
2. Measure parameter P of all these ni cells;
3. Identification of the threshold Tm placed at the median value of all measurements of parameter P of the population. By design, half of the cells should have a value below Tm, and half above Tm.
4. Generate ni bits, “0”’s below Tm, and “1”’s above Tm.
5. Use the XOR compiler to combine chunks of f bits together.
6. The resulting stream of m bits is the stream of the TRNG.

With this method the raw data stream generated by the memory array and the 0-cell has a population with equal numbers of “0”’s and “1”, regardless of a potential drift in temperature caused by a natural variation, or caused by the hot air blower of the hacker. The method is applicable to compensate for any drifts, noise, or aging; the integrity of the TRNG is thereby protected.

10.2. Generalization to other TRNG designs

When the physical component generates a data stream with a deviation from absolute randomness λ allowing it is possible to model the size f of the chunks that are XORed together, as described in Fig.8, to meet a particular λ objective. The model can be used as a predictive tool. For example, as shown Fig.18, the number f necessary to compile a data stream of various initial randomness can be anticipated to be λc < 5 10^-8 or λc < 10^-10. This could be valuable to adjust the compilation as a function of the monitoring of the randomness of the incoming data stream.

![Predictive model – f-bit needed for a given objective λ.](image)

**Figure 18**: Predictive model – f-bit needed for a given objective λ.

| Initial Randomness | % of 0-cells | For λc < 5 10^-8 | For λc < 1 10^-10 |
|--------------------|--------------|------------------|------------------|
| 52% - 48%          | 2%           | 5                | 7                |
| 54% - 46%          | 4%           | 7                | 9                |
| 56% - 44%          | 7%           | 8                | 12               |
| 60% - 40%          | 11%          | 10               | 14               |

The proposed method to design TRNG is not limited to ReRAM arrays, and Vset as parameter P. The method is applicable to any memory device as long as it is possible to identify a parameter P that can be reliably tested to sort out the cells and identify enough unstable 0-cells. The algorithm presented Fig. 17 is generic:

- Flash or EEPROM memory: parameter P could be the trans-conductance of the cells after fixed time programming. The threshold voltage of each cell, after fixed time injection of electrons in the floating gate, vary cell-to-cell due to variations in fabrication parameters such as tunnel oxide thickness and doping levels. Very small changes of threshold voltage can create major changes in the trans-conductance, which are desirable sources of randomness.
- DRAM memory: parameter P could be the measurement of the residual charge left in a cell after constant discharging time. One effective method is to program all cells, and put the refresh cycle on hold. The fuzzy cells can flip above or below the threshold value of residual charge.
- ReRAM memory: in addition of the Vset as presented in this paper, parameter P could be the Vset (threshold voltage to erase the cells), Roff (resistivity on the high resistance state), or Ron (resistivity on the low resistivity state). Some parameters like Ron can be flaky, and jump in a non-erratic way from a set of several discrete values, which is not a desirable source of randomness for a TRNG.
- SRAM memory: the PUFs are based on the determinations of the cells flipping to either a 0 state, or a 1 state after power-off- power-on cycles. However, 3 to 5% of the SRAM cells are fuzzy, they can switch on either states at each cycle. The recommended methodology is to test the SRAM array, and keep track of the 0-states for TRNG.

The use of the XOR compiler enhances the randomness of any data streams regardless of their origin. The XORing by chunk of f-bits is therefore applicable to a stream of m incoming bits, as shown in Fig 19:

1. If the length of the incoming stream of m bits is not an integer number multiple of f, the two are related by eq(44), ri is the remainder of m congruent f.
   \[ m = m_i * f + r_i \]  (44)
2. Several 0’s can be added to the stream of ri bits to form a chunk with a length f. The total number of chunks will be equal to m_i + 1.
3. The XORing is done by chunks of f-bits.
4. The resulting stream of m_i+1 has a deviation to non-randomness that is lower than the incoming stream m.
11. Native ternary random numbers generators

In this section, we are presenting a method to directly generate native random trits from PUF memory arrays, as well as ways to enhance randomness with modulo 3 sum adders. As presented in the first section, the fuzzy cells of the PUFs are used as multiple sources of randomness, and the XOR compilers replace mod3 adders. Ternary computing uses trits, for example (0, 1, 2) or balanced (−, 0, +), instead of the bits (0, 1) used in binary computing [61-67]. Can ternary computing improve cybersecurity and Information Assurance [68-70]? Ternary computing is not a new concept, and is more complex to implement than binary computing. One suggested architecture uses heterogeneous computing elements, and combine binary units to run legacy codes, and native ternary computing units for security [71]:

- Better handling of the natural fuzziness, with lower reliance on error correction codes;
- Can take advantage of ternary hardware, and advances in microelectronics, such as the ternary PUFs described in the first section of this paper [72-79];
- The cryptography based on ternary states has more computing units for ternary computing, and softw.

For example, let us assume that the length of a data stream is N=128. The number of possible combinations for binary streams is $2^{128}=3.4 \times 10^{38}$, and becomes $3^{128}=1.2 \times 10^{61}$ for ternary streams, which is considerably larger. Native random numbers are valuable for cryptographic protocols based on ternary computing. One way to create ternary random numbers is to convert binary random numbers into decimal numbers, then to convert the decimal data stream back into ternary random numbers. Such a method add complexity, and can potentially expose the random numbers to hackers. A direct generation of native ternary random numbers is therefore desirable. The definition of deviation from perfect randomness for ternary TRNG, is similar to the one developed for binary data streams. As presented in the first section, each bit $a_i$ of the perfectly binary random stream $\{a_1, \ldots, a_n\}$ should have precisely the same probability to be either a “1” or a “0”. The average deviation from randomness, $\lambda$ is given by:

$$P_{a_i=1} = P_{a_i=0} = 0.5 \quad (50)$$

$$\lambda = \frac{1}{2} \left( |P_{a_i=1} - 0.5| + |P_{a_i=0} - 0.5| \right) = |P_{a_i=1} - 0.5| = |P_{a_i=0} - 0.5| \quad (51)$$

In the case of ternary data streams of trits with “−”, “0”, and “+” states, the term $\lambda$ is given by:

$$\lambda = 1/3(|P_{a_i=-1} - 1/3| + |P_{a_i=0} - 1/3| + |P_{a_i=+1} - 1/3|) \quad (52)$$

$$0 = P_{a_i=-1} + P_{a_i=0} + P_{a_i=+1} \quad (53)$$

12. Description of the method

12.1. Segmentation of the fuzzy cells of the memory PUFs

The fuzzy cells, the 0-cells, can be segmented into three subgroups, see Fig. 20:

- The cells that have a higher probability to be tested as “−” are called A-cells. They have an average probability $P_{a_i=0}$ to be tested as “−”, $P_{a_i=0}$ to be tested as “0”, and $P_{a_i=0}$ to be tested as “+”.
- The cells that have a higher probability to be tested as “0” are called B-cells. They have an average probability $P_{a_i=0}$ to be tested as “−”, $P_{a_i=0}$ to be tested as “0”, and $P_{a_i=0}$ to be tested as “+”.
- The cells that have a higher probability to be tested as “+” are called C-cells. They have an average probability $P_{a_i=0}$ to be tested as “−”, $P_{a_i=0}$ to be tested as “0”, and $P_{a_i=0}$ to be tested as “+”.

![Figure 20: Segmentation of the fuzzy cells in trits](Image)

The selection of the transition of parameter $P$ between “−” and “0”, $T_2$, and the transition of parameter $P$ between “0” and “+”, $T_3$, can be such that the total number of A-cells selected within the 0-cells equal the number of B-cells, and the number of C-cells. The deviation from perfect randomness $\lambda$ of the stream of native ternary random numbers generated from the fuzzy cells A, B, C is given by:

$$\lambda = \left[ \frac{1}{9}P_{a_i=\text{−}} + \frac{1}{9}P_{a_i=0} + \frac{1}{9}P_{a_i=+} \right] + \left[ \frac{1}{9}P_{a_i=\text{−}} + \frac{1}{9}P_{a_i=0} + \frac{1}{9}P_{a_i=+} \right] + \left[ \frac{1}{9}P_{a_i=\text{−}} + \frac{1}{9}P_{a_i=0} + \frac{1}{9}P_{a_i=+} \right] \quad (54)$$

12.2. Enhancement of the randomness with mod3 adders

The algorithm using a mod3 adder, see Fig.21 and 22, is similar to the one presented section A.

1) The number of cells needed to generate a stream of mi trits is $n_i = m_i \times f$, they are selected as part of the fuzzy 0-cells of the ternary memory PUF;
2) Parameter $P$ is measured on all $n_i$ cells;
3) The population of $n_i$ cells is segmented into three third based on the value of $P$. The threshold separating the bottom third and the central third is $T_2$; the threshold separating the central third and to top third is $T_3$;
4) With the segmentation in three done step 3, the $n_i$ cells at the bottom third are carrying “−” state, the cells in the middle are “0”'s, and the cells at the top third are “+”s;
5) The stream of trits is added by chunks of f trits; Instead of a XOR compiler, the mod 3 adder of chunks of trits enhance randomness. With mod 3 sum adders, two input trits $a_i$ and $a_{i+1}$ are transformed into $c_i = a_i \oplus a_{i+1}$, with the following truth table:

- $(a_i = 0; a_{i+1} = +)$, or $(a_i = +; a_{i+1} = 0)$, or $(a_i = a_{i+1} = +) \rightarrow c_i = - (a_i = +; a_{i+1} = -)$, or $(a_i = -; a_{i+1} = +)$, or $(a_i = a_{i+1} = 0) \rightarrow c_i = 0$
Mod 3 addition increases randomness, the knowledge of \( ci \) is not disclosing the value of \( ai \) and \( ai+1 \). \( ci = 0, 0, + \) or \( +, +, + \) can be the result of three possible pairs \( \{ ai, ai+1 \} \), with equal probability. If two trits \( ai \) and \( ai+1 \) are somewhat random, the trit \( ci \) is even more random than either \( ai \) or \( ai+1 \). Let us assume that the stream of random trits generated by the 0-cells of the memory PUF array is \( \{ a_1, a_2, \ldots, a_n \} \).

13. Modeling of the randomness after mod3 additions

13.1. Model with mod3 addition by chunks of two trits

In this section we are proposing a simplified model that quantifies the level of randomness of mod 3 adders when two adjacent trits are added mod 3. Fig.23 shows such a scheme. We are assuming that the 0-cells are distributed into three type of cells (A, B, and C), each of them with a probability of occurrence of 1/3. Statistically the stream of trits \( \{ ai, ai+1, \ldots, a_n \} \) contain trits with equal probability to be \( \ldots, 0, \) or \( \ldots, + \), also with a value of 1/3. However A-cells have a higher probability to have \( \ldots, + \)s, B-cells have a higher probability to have \( \ldots, 0 \)s, and C-cells have a higher probability to have \( \ldots, - \)s.

In Fig.24, we are showing an arbitrary set of probabilities verifying that the probability to have either A, B, or C cells is 1/3, and the probability to have either \( \ldots, + \), \( \ldots, 0 \), or \( \ldots, - \) states is also 1/3. In this table:

- \( P_{A0} = 1/3 + \Delta A_0 \cdot \lambda \); with \( \Delta A_0 = 1.8 \) (55)
- \( P_{B0} = 1/9 + \Delta B_0 \cdot \lambda \); with \( \Delta B_0 = 0.9 \) (56)
- \( P_{C0} = 1/9 + \Delta C_0 \cdot \lambda \); with \( \Delta C_0 = 1.8 \) (57)

Other tables and more complicated model can replace this arbitrary representation; however, the suggested simplified model describes quite well the experimental observations. When the cells are combined by pairs, and the trits added mod 3, 9 combinations of cells are possible with an equal probability of 1/9: AA, AB, AC, BA, BB, BC, CA, CB, CC. The average probability to have trits with \( \ldots, - \), \( \ldots, 0 \), and \( \ldots, + \) is 1/3.

Two cells AA have three possible combinations which can result in a trit at \( \ldots, - \):

- Both cells at \( \ldots, + \), the probability is: \( P_{A00} = P_{A00} \);
- The first cell is at \( \ldots, 0 \), the second at \( \ldots, - \), the probability is: \( P_{A00} = P_{A00} \);
- The first cell is at \( \ldots, - \), the second at \( \ldots, 0 \), the probability is: \( P_{A00} = P_{A00} \).

The resulting probability \( P_{AA} \) that two trits AA can result in a trit at \( \ldots, - \) is given by:

\[
\Delta \text{ per type of cell} \quad \text{Prob. to have} \quad A(-) \quad B(0) \quad C(+) \\
\begin{array}{ccc}
\text{Prob.} & - & + \\
\text{be:} & +1.8 & -0.45 & -1.35 \\
\text{be:} & + & -0.45 & +0.9 \\
\text{be:} & + & -1.35 & +1.8 \\
\end{array}
\]

The initial randomness is:

\[
\delta i = (1/9)\left( 1 \sum_{i=1}^{n} \Delta A_{Ai} \mid 1 \sum_{i=1}^{n} \Delta B_{Bi} \mid 1 \sum_{i=1}^{n} \Delta C_{Ci} \right) = 1/9 \left( 3.6 + 1.8 + 3.6 \right) \cdot \lambda = \lambda (58)
\]
\[ P_{AA \leftarrow} = (P_{A \leftarrow A} \cdot P_{A \leftarrow A} + (P_{A \leftarrow A} \cdot P_{A \leftarrow A} + (P_{A \leftarrow A} \cdot P_{A \leftarrow A}) = (1/1-1.35 \cdot \lambda)^2 + 2/(1/9 + 1.8 \cdot \lambda) (1/9 - 0.45 \cdot \lambda) = 1/27 + 0.2 \cdot \lambda^2 \]  

(59)

In Fig. 25 is showing the result of the computations of 27 configurations: probability to get “-“-, “0”-, or “1”- after addition mod 3 for each pair \( P_{AA \leftarrow} , P_{A \leftarrow B} , P_{A \leftarrow A} , P_{AB \leftarrow} , P_{AB \leftarrow} \).

14. Experimental analysis with mod3 adders

The analysis is based on the data presented in the first section related to the measurement of the Vset of ReRAM devices. We are again considering the same five cases to sort out the fuzzy “0-cells”:

- Case 1: Only 2% of the cells are 0-cells. For these cells parameter \( \mathcal{P} \) is very close to the transition of 2.1 Volt. \( \lambda = 2 \cdot 10^{-2} \)
- Case 2: 4% of the cells are 0-cells. \( \lambda = 4 \cdot 10^{-2} \)
- Case 3: 7% of the cells are 0-cells. \( \lambda = 6 \cdot 10^{-2} \)
- Case 4: 11% of the cells are 0-cells. \( \lambda = 1 \cdot 10^{-1} \)

This range of options allows the quantification of the effectiveness of the addition mod 3 to scramble the trits, as a function of the initial randomness coming from the ternary PUF memory. Case 1 is the one with the higher initial randomness, while Case 4 is the lowest one.

The probabilistic model developed above, is the base of the analysis of the four experimental cases. The results of the computations are shown in Fig. 26 and Fig. 27.

The impact of the addition modulo 3 addition on the level of randomness on the resulting data streams of trits is increasing when the number of cells \( f \) involved in the addition increases from \( f = 2 \) to \( f = 8 \).

![Figure 25: probability per configuration after mod 3 addition of two trits.](image)

![Figure 26: modeling of the effect of mod 3 addition as a function of the experimental cases.](image)

![Figure 27: deviation from perfect randomness plotted as a function of the size of the chunk of trits involved in mod3 addition for the four cases.](image)
Discussion and conclusion

The use of ternary PUFs to design TRNG, combined with a XOR compiler, or a modulo 3 addition has the following benefits:

➢ The cells with fuzzy behavior of a ternary PUF, the 0-cells, can provide multiple sources of independent randomness. A memory array in the megabyte range can have a large quantity of such cells;

➢ The randomness of the binary data streams extracted from the ternary 0-cells are enhanced by a XOR compiler. Based on a normal distribution of parameter \( \mathcal{P} \), the proposed statistical model can quantify the deviation from pure randomness of the TRNGs. It is possible to calculate \( f \), the length of the XOR, to reach a desired level of non-randomness \( \lambda_f \), as a function of the level of non-randomness \( \lambda_i \) of the incoming data stream extracted from the physical element.

➢ The randomness of the ternary data streams extracted from the 0-cells can be enhanced by a modulo 3 adder. It is possible to directly generate a stream of random trits without having to convert binary data streams into ternary data streams;

➢ It is possible to anticipate, with the suggested probabilistic model, the minimum size of chunks of data \( f \) that need to be processed to reach the level of randomness \( \lambda_f \). This is the case for the XOR compiler, and the mod 3 adder;

➢ The proposed methodology minimizes sensitivity to parameter drifts such as temperature, aging, or biasing conditions. It is anticipated that the drifts should not materially degrade the quality of the TRNGs.

➢ The hardware implementation of both the XOR compiler, and the mod 3 adder can use known commercial CMOS circuitry.

➢ The method can reach NIST expectations in terms of deviation from pure randomness of the TRNG, even if the randomness created by the PUF is weak.

The experimental section of this work, which is based on the measurements of the Vset of ReRAM cells, produced a distribution that is able to show enough randomness to generate random numbers. We noticed that the XOR data compiler is not effective when the initial data stream is not random. The model developed assumes that the initial random numbers generated from the 0-cells are symmetrically distributed between A-cells and B-cells.

Other statistical distributions beside the normal one are under consideration in our research effort. We are not anticipating that these improved statistical models will significantly change the outcome when only cells close to the median distribution are selected. This is not the case for wider distribution of the 0-cells away from the median.

Future work: The objective of this work was to develop TRNG for cryptographic protocols that can be embedded in the Internet of Things (IoT). The implementation of affordable sources of randomness to secure IoTs can benefit from the ease of use of ternary PUFs, which are tamper resistant. TRNGs are essentials elements to encryption protocols involving PUF CRPs, and other cryptographic keys. We are studying the design of a prototype that incorporates the proposed TRNG scheme with various ReRAM arrays. The prototype is intended to automatically extract large quantities of PUF CRPs and random streams of bits, and trits. We intend to use the prototype to further validate our statistical models, and to leverage the tools developed by NIST that are available online to quantify the entropy, and the level of randomness of the TRNGs. The prototype should have the built-in flexibility to allow us to analyze multiple types of memory arrays, with different methods of fabrication.

We are interested in optimizing the randomness of the TRNG while reducing CRP error rates of the PUFs, and developing cryptographic protocols that leverage the combined capabilities. The method described in this paper can be used to the swarm dynamics generating true random noises [8], and other similar applications requiring TRNG. To accelerate the process to generate fresh random numbers on demand, the 0-cells can be tested in advance [80], and the data can be stored in the memory. The read time of a ReRAM is typically 10ns/bit, so we believe that the generation of the TRNG has the potential to be done at a rate of 100Mbit/s. The method presented in this paper can also be extended to n-value logic, for example quaternary logic (4 bits), pentagonal logic (5 bits), or hexagonal logic (6bits). In such cases the 0-cells are divided in n different type of cells, and the addition of chunk of n-bits is done modulo n.

Acknowledgment

Dr. Marius Orłowski from Virginia Tech produced, and characterized the metal oxide ReRAM samples [49]. These measurements were used to prepare the experimental analysis.

Dr. Derek Sondergelder from the Department of Mathematics and Statistic of Northern Arizona University provided input for the development of the statistical model. The author is thanking both Dr. Donald Telesca from the US Air Force Research Labs for his support in this work.

References

[1] B. Cambou; A XOR data compiler combined with PUF for TRNG; SAI/IEEE computing conference, July 2017;
[2] C. Paar, and J. Pezl; Understanding Cryptography- A text book for students and practitioners; Springer editions, 2011;
[3] C. P. Pfleeger, et al; Security in Computing; Fifth edition; Prentice Hall editions, 2015;
[4] R. Soorat; Hardware random number generation for cryptography; https://arxiv.org/pdf/1510.01234 , 2015;
[5] D. Glosemeyer, and B Knapp; Random Number Generation; Wolfram Mathematical Tutorial collection, 2008;
[6] M. Stipievic, et al; true random number generator; Open problems in Mathematics and computational science: pp.275-315, Springer, 2014;
[7] H. Katzgraber; Random Numbers in scientific computing: an introduction; Int. school comp. science, 2010, Oldenburg, Germany
[8] Y. Shang, and R. Bouffanais; “Influence of the number of topologically interacting neighbors on swarm dynamics”; Scientific Reports DOI: 10.1038/srep04184, 2015.
[9] Berndt Gammel, et al; Jul 2012; Random Generator configured to combine states of memory cells; US patent No 7,979,482 B2;
[10] Daniel E Holcomb, et al; Power up SRAM state as an identifying Fingerprint and Source of True Random Numbers; IEEE Trans. On Computers, 2009 Vol 58, issue No09 Sept.
[69] D.M. Miller, and M.A. Thornton; Multiple Valued Logic: Concepts and Representations; Synthesis Lectures on Digital Circuits and Systems; Morgan & Claypool Publishers, 2007;
[70] S. Ahmad, and M. Alam; Balanced Ternary Logic For improving Computing; IJCSIT, 2014;
[71] P.G. Flikkema and B. Cambou; Adapting Processor Architectures for the Periphery of the IoT Nervous System; IEEE 3rd World Forum on Internet of Things (WF-IoT), December 2016;
[72] H. Gundersen; Aspect of balanced ternary arithmetic implemented using CMOS recharged semi-floating gate device; thesis Oslo Univ. 2008;
[73] P.C. Balla, and A. Antoniou; Low Power Dissipation MOS Ternary Logic Family; IEEE J of Solid State Circuits, Oct 1984;
[74] P.C. Balla, A. Antoniou; Low Power Dissipation MOS Ternary Logic Family; IEEE J of Solid State Circuits, Oct 1984;
[75] X.W. Wu; CMOS Ternary Logic Circuits; IEE Proceedings, Feb 1990
[76] P. Nagaraju, et al; Ternary Logic Gates and Ternary SRAM Implementation in VLSI; IJSR, Nov 2014;
[77] N.P. Wanjari, S.P. Hajare; VLSI Design and Implementation of Ternary Logic Gates and Ternary SRAM Cell; IJECSE, April 2013;
[78] A. Srivastava, and K. Venkatapathy; Design and Implementation of a Low Power Ternary Full Adder; VLSI Design, vol 4, No.1, 1996;
[79] N.P. Wanjari, and S.P. Hajare; VLSI Design and Implementation of Ternary Logic Gates and Ternary SRAM Cell; IJECSE, April 2013;
[80] Anuj Gupta, May 2005, Implementing Generic BIST for testing Kilo-Bit Memories; Master Thesis No-6030402 Deemed University Patiala India