Utah State University

ECE 5760 Hardware Security

# Physical Unclonable Functions

John Call, Braydn Clark, Josh Lynn

April 26, 2017

# Abstract

*This document describes a physical unclonable function (PUF) used to create a true random number. It will first provide background on the design of PUFS then the specific implementation used in this paper to produce a random number. This will be followed by the description of the interface allowing the output of the random stream. Finally a description of the common tests applied to check for true randomness will be given along with the results for the PUF described.*


# 1 Introduction

The manufacturing process of the structures exploits the random unpredictable physical factors to cause the structure to be difficult to duplicate. This can be exploited to create authentication systems, cryptographic keys, random numbers, or hundreds of other applications. This paper will discuss a PUF designed to produce a true random number.

# 2 Ring Oscillator

The PUF for random number generation is based on a ring oscillator structure. The structure of a ring oscillator is relatively simple. It consists of a series of not gates and a single feedback loop into an and gate along with an enable bit. This is pictured in Figure 1.

If the number of not gates is odd then upon assertion of the enable bit the value of the output bit will begin to oscillate. The frequency of this oscillation depends on the individual delay of the gates in the circuit. The increasing number of gates decreases the frequency.

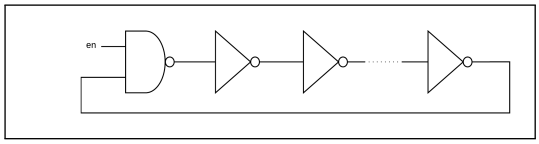



Figure 1: *Basic structure of a ring oscillator*

The ring oscillator has no reinforcement of the oscillation frequency and the frequency will slide and change over time. This will allow the oscillation to be used for random bit generation. Attaching a flip flop allows sampling of this randomized oscillation at set periods. The sampled output can then be entered into a xor tree along with an arbitrary number of other ring oscillator outputs. The combination of ring oscillators allows the final output stream to be as random as possible.

For the implementation discussed in this paper a series of 8 ring oscillators of varying sizes. This series is used as crappy clocks. Those clocks are sent into 2 bit generator modules containing 8 block generators. Each block generator uses one of the crappy clocks and eight ring oscillators with 4 NOT Gates. The eight ring oscillators are xor'd together to produce the output for a block generator. The 8 block generator outputs are xor'd together to produce a single output for the bit generator. The outputs of the 2 bit generator modules are then compared and any repeated sequences of 1's or 0's are removed. This allows 128 different ring oscillators to be used

to produce a single bit. This should allow the randomness of the PUF to be as great as possible.

This design was influenced by the description of a ring oscillator random number generator in the paper by Stewart Robson [2]. It was a design selected after several failed iterations. It was hoped that the increasing complexity surrounding the simple oscillators would subsequently increase the randomness of the PUF. It was not proven to be so however.

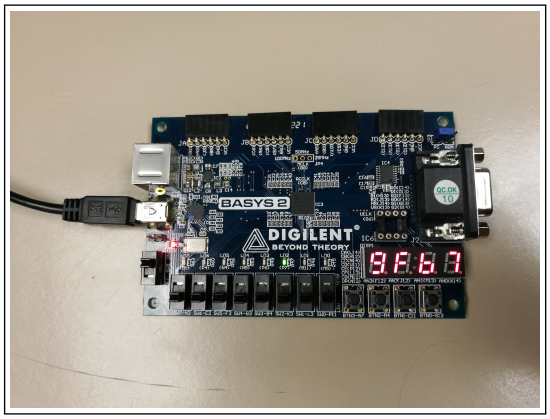



Figure 2: *Image of the output to seven segment display.*

# 3 Interface

The output of the random number generator was observed through multiple methods. Due to the continued synthesizing errors associated with the ring oscillator structure the first interface of a simple blinking LED light was developed. The development software was unable to correctly simulate the output of the oscillator as it failed to consider gate delay correctly. The output of the ring oscillator was instead divided to a sufficiently slow frequency for the human eye to observe the oscillation. This allowed the ring oscillator structure to be verified.

This type of verification was duplicated at each stage of the design verifying the oscillation of the output. While the blinking LED is not useful for verification of true randomness it proved invaluable for debugging the system. However, the random string was unable to be translated into a usable stream. Due to inconsistencies that we were unable to isolate, the stream produced large numbers of zero values. This was demonstrated using the seven segment display to provide immediate feedback as shown in Figure 2. This alone shows the output of the ring oscillator structure to be highly non-random which is the reason for the inclusion of the logic which removes those long sequences of zeros.

# 4 NIST tests

The National Institute of Standards and Technology supplies a series of statistical tests for random and pseudo-random number generators [1]. While these tests are sufficient for the purposes of this paper they are not enough to guarantee sufficient randomness for use in a cryptographic setting.

The stereotypical example of randomness is the coin flip. Assuming the coin is evenly balanced the chance of the coin landing on one side over the other is exactly 1/2. Each coin flip must also be independent of previous

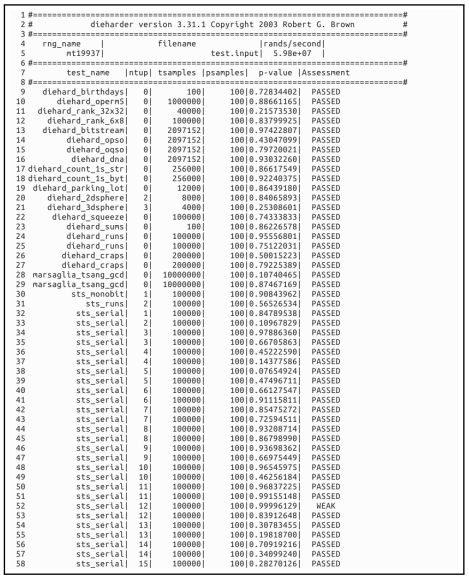
flips. This means the value of each flip cannot be predicted with any amount of reliability. This perfect unpredictability and probability of 1/2 is the baseline for all randomness test, the closer a binary sequence is to producing independence between two bits the more random the sequence is. In order to complete the NIST test suite three assumptions are made for the binary sequences. The first assumption is uniformity, that at any point in the sequence the probability of a 1 or 0 occurring is 1/2. Second is scalability, that any test applied to the full sequence can be applied to any subsequence. Third, and finally, consistency, that the behavior of the random number generator is consistent across starting values.

The NIST suite contains 15 separate tests and they each operate on binary sequences of arbitrary length. Each test compares the probability distributions against either the standard normal distribution or chi-square distribution. If with the application of the tests results in a significant probability of randomness, than the sequence passes the test.

The NIST suite was unable to be applied to the PUF described in this paper due to the lack of time. The sequence produced was therefore unable to be proven to be significantly random. This means that the all that can be said for the PUF is that the randomness produced was sufficient to appear random to a human observer.

# 5 DIEHARDER test

The DIEHARDER suite of tests is another series of statistical tests for random and pseudo-random number generators. It was initially assumed that larger data sets than were easy to produce through the PUF described in this paper were required. When it was realized that only a small sequence was necessary the DIEHARDER was applied. The results were consistent, showing they passed most of the tests with a weak result on two of them. The weak result came from an sts-serial and an rgb-lagged-sum test. A portion of the results is shown in Figure /refdie



```
#=============================================================================#
#            diharder version 3.31.1 Copyright 2003 Robert G. Brown           #
#=============================================================================#
   rng_name    |           filename             |rands/second|
        mt19937|                        test.input|  5.98e+07  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.72834402|  PASSED
      diehard_operm5|   0|   1000000|     100|0.88661165|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.21573530|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.83799925|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.97422807|  PASSED
         diehard_opso|   0|   2097152|     100|0.43047099|  PASSED
         diehard_oqso|   0|   2097152|     100|0.79720021|  PASSED
          diehard_dna|   0|   2097152|     100|0.93032260|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.86617549|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.92240375|  PASSED
diehard_parking_lot|   0|     12000|     100|0.86439180|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.84065893|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.25308601|  PASSED
     diehard_squeeze|   0|    100000|     100|0.74333833|  PASSED
        diehard_sums|   0|       100|     100|0.86226578|  PASSED
        diehard_runs|   0|    100000|     100|0.95556801|  PASSED
        diehard_runs|   0|    100000|     100|0.75122031|  PASSED
       diehard_craps|   0|    200000|     100|0.50015223|  PASSED
       diehard_craps|   0|    200000|     100|0.79225389|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.10740465|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.87467169|  PASSED
         sts_monobit|   1|    100000|     100|0.90843962|  PASSED
            sts_runs|   2|    100000|     100|0.56526534|  PASSED
          sts_serial|   1|    100000|     100|0.84789538|  PASSED
          sts_serial|   2|    100000|     100|0.10967829|  PASSED
          sts_serial|   3|    100000|     100|0.97886360|  PASSED
          sts_serial|   3|    100000|     100|0.66705863|  PASSED
          sts_serial|   4|    100000|     100|0.45222590|  PASSED
          sts_serial|   4|    100000|     100|0.14377586|  PASSED
          sts_serial|   5|    100000|     100|0.07654924|  PASSED
          sts_serial|   5|    100000|     100|0.47496711|  PASSED
          sts_serial|   6|    100000|     100|0.66127547|  PASSED
          sts_serial|   6|    100000|     100|0.91115811|  PASSED
          sts_serial|   7|    100000|     100|0.85475272|  PASSED
          sts_serial|   7|    100000|     100|0.72594511|  PASSED
          sts_serial|   8|    100000|     100|0.93208714|  PASSED
          sts_serial|   8|    100000|     100|0.86798990|  PASSED
          sts_serial|   9|    100000|     100|0.93693362|  PASSED
          sts_serial|   9|    100000|     100|0.66975449|  PASSED
          sts_serial|  10|    100000|     100|0.96545975|  PASSED
          sts_serial|  10|    100000|     100|0.46256184|  PASSED
          sts_serial|  11|    100000|     100|0.96837225|  PASSED
          sts_serial|  11|    100000|     100|0.99155148|  PASSED
          sts_serial|  12|    100000|     100|0.99996129|   WEAK
          sts_serial|  12|    100000|     100|0.83912648|  PASSED
          sts_serial|  13|    100000|     100|0.30783455|  PASSED
          sts_serial|  13|    100000|     100|0.19818700|  PASSED
          sts_serial|  14|    100000|     100|0.70919216|  PASSED
          sts_serial|  14|    100000|     100|0.34099240|  PASSED
          sts_serial|  15|    100000|     100|0.28270126|  PASSED
```

Figure 3: *Partial results of the DIEHARDER suite of tests.*

# 6 Conclusion

The physically unclonable function described in the paper successfully demonstrated the logic necessary to produce a ring oscillator, and produce a seemingly random sequence. However, it was not able to be shown that this sequence was sufficiently random to pass the NIST sequence of tests.

# References


[1] Rukhin, Andrew et al. "A Statistical Test Suite For Random And Pseudorandom Number Generators For Cryptographic Applications". National Institute of Standards and Technology 800.22 (2010): n. pag. Print.

[2] Robson, Stewart. "A Ring Oscillator Based Truly Random Number Generator". MS. University of Waterloo, 2013. Print.