Large-Scale Characterization and Optimization of Bistable Rings

by

Victor Simon Mercola

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Electrical and Computer Engineering

May 2023

APPROVED:

Prof. Fatemeh Ganji, Major Thesis Advisor

Prof. Shahin Tajik, Major Thesis Advisor

Acknowledgments

This thesis could not have been possible without the previous work done by my project partner Isabel Herrero Estrada during our MQP in the 2021-2022 academic year. She worked with me to build the tools I used to simulate bistable rings for our MQP; I later revamped those tools into the ones I used for this thesis. Her analysis method using the Chaos Decision Tree Algorithm inspired me to automate my own methods.

In addition, I would like to thank William L. Appleyard, the head of WPI's ECE shop, for providing us with the FPGAs used in my thesis.

Finally, I would also like to extend my sincere thanks to Rabin Yu Acharya from the University of Florida. His paper on chaogate parameter optimization helped me finalize my simulation code, and his advice helped me build my genetic algorithm strategies.

Table of Contents

List of Tables

List of Figures

Abstract

True random number generators (TRNGs) ideally produce unbiased, uncorrelated, and incompressible bits of information by extracting randomness from a stochastic process. These circuits help in secure communication, user authentication, and user identification protocols. Some TRNGs employ a bistable ring (BR), a digital logic circuit made up of an even number of inverters connected in a loop, as their core. When powered on, the BR oscillates and may settle into one of two states. We introduce concepts from nonlinear dynamic system analysis to determine whether the BR's trajectories are random enough to be considered a promising entropy source in TRNGs. Our example BR simulations and Monte Carlo process variation experiments in Ngspice show that the BR's trajectories are chaotic in the best-case scenario. We also study the FPGA realizations of BRs and observe periodic behavior in those implementations. Following these observations, we evolved instances of BRs using a genetic algorithm (GA) to determine whether one could surpass the chaotic and periodic characteristics of simulated and implemented BRs. According to our results for optimizing the BR trajectories' complexity (measured by permutation entropy), some instances created by the GA could exhibit stochastic behavior.

1 Introduction

Integrated circuits are useful in the field of cryptography since their physical characteristics can help verify and authenticate users, obfuscate keys for encryption, and generate random numbers. This paradigm of using physical devices for cryptographic purposes forms the basis for the field of hardware security. The bistable ring (BR) is a circuit that has found applications in this field; it uses digital logic gates to create analog signals without needing external sources or passive components. The BR is easy to build with digital logic gates or implement inside a field-programmable gate array (FPGA). We are interested in its use in random number generation as the core of a true random number generator (TRNG) like the one developed by Intel [1].

It is easy to treat a random number generator (RNG) as a black box, not caring whether its outputs are perfectly random. However, in cryptography, forgetting to check the quality of an RNG's outputs is a dangerous mistake. Kerckhoff's Principle states that one should assume all attackers know everything about how a cryptographic system in question works, except for its users' secrets (e.g., keys). Many of these secrets are indirect or direct results from an RNG; if the attacker figures out the secrets, then they can compromise the system's security [2].

The BR has also found use in authentication, identification, and key obfuscation as part of the fingerprint of a physical unclonable function (PUF) [3], [4]. PUFs and TRNGs built with the BR both consistently release it from the same unstable state [1]–[4]; how can it both predictably settle into a single result in the former yet unpredictably settle into one of two results in the latter? Resolving this contradiction is difficult since no generalized closed-form system of equations exists that can characterize any BR. When the mathematical models of engineering fail, one must turn to physics to get answers: the field of nonlinear dynamics can shed light on this issue by understanding the BR's voltage trajectories as a dynamical system.

Nonlinear dynamics, the branch of physics concerning nonlinear dynamical systems, contains tools that can determine whether a dynamical system is deterministic (i.e., follows an underlying ruleset) or stochastic (i.e., involves probability distribution parameters). If the BR is a deterministic system, then its underlying ruleset, initial conditions, and parameters define its corresponding trajectories; this would help certify its use in PUF design but harm its use as an entropy source for generating random numbers. Otherwise, if the BR is a stochastic system, then its trajectories are characterizable with probability distributions; this would help certify its use as an entropy source for generating random numbers but harm its use in PUF design.

The purpose of this thesis is to prove the efficacy of tools from nonlinear dynamics in the field of hardware security by characterizing the trajectories of different BR implementations to classify them as either deterministic or stochastic. Our Monte Carlo simulations in Ngspice and implementations in FPGA hardware show us how the BR propagates process noise, thermal noise, and measurement noise through its signals. We then combine nonlinear dynamics with genetic programming to evolve the most complex BRs to quantitatively increase the BR system's chaos.

2 Randomness and Random Number Generation

At its basic form, randomness is the unpredictability of outcomes. From a frequentist's perspective, an event's probability is its relative frequency of occurring in many identical trials [5]. Modern cryptographic methods could not exist without randomness; random number generators themselves are cryptographic primitives. Their applications include key generation algorithms for crafting public and private keys, user identification like password generators or unique user IDs, initialization vectors for block ciphers, and nonces for authentication protocols [2].

A random number generator (RNG) should output unbiased, uncorrelated, and incompressible numbers or bits upon request. When generating bits, one can model a single output of a random number generator as a Bernoulli random variable with a p-value of 0.5, outputting a "0" or a "1" with the same probability. These Bernoulli random variables can assemble to form a Bernoulli stochastic process, a fifty-fifty chance of outputting "0" or "1" in each trial over time [5]. Additionally, each output of the random number generator should have no correlation with other outputs: there should be no relationship between a new outcome and future or previous outcomes. The knowledge of previous outcomes should not allow one to predict future outcomes with a better accuracy than fifty percent. Finally, there should be no way to compress the string delivered by a random number generator [2].

As an example, algorithms running on a computer can generate numbers by using a pseudorandom number generator (PRNG). These RNGs create seemingly random numbers from an initial starting value, called a seed. The PRNG will always output the same string of random bits when given the same initial seed, making it deterministic. PRNGs can be adjusted to have equal probabilities of outputting "0" or "1", which eliminates any bias present in their bit string. However, the strong long-range correlations between different outputs of a PRNG can undermine its cryptographic strength; therefore, using a PRNG in cryptographic applications is a security risk [2]. Nonetheless, PRNGs are still useful in various applications including procedural generation because they are deterministic.

If one cannot rely on computers to generate random numbers, then one must get randomness from an external source: this is where True RNGs (TRNGs) come into play. TRNGs extract randomness by augmenting samples from a stochastic process to produce random bits; this makes them much better candidates for generating truly random numbers than PRNGs since they employ a non-deterministic system to generate unpredictable results. TRNGs are usually separate pieces of hardware that connect via a USB or PCI bus and are managed via custom drivers. TRNGs may produce incompressible bits of information, but their cryptographic strength may falter due to imperfections in the randomness extractor or the stochastic process source itself. One can measure these imperfections by determining a TRNG's bias b and serial autocorrelation coefficients $\{a_k\}$. The values that b and a_k can take are normalized to fit within the range $[-1, 1]$ and are set so an ideal RNG has $b = 0$ and $a_k = 0$. A TRNG's bias, shown in Equation 1, is the difference between the probability that it outputs "1" and the probability that it outputs "0" divided by two [2].

$$b = \frac{\mathbb{P}\left(RNG = 1\right) - \mathbb{P}\left(RNG = 0\right)}{2} \tag{1}$$

A TRNG's serial autocorrelation coefficients, shown in Equation 2, measure how much its output is correlated with itself. The serial autocorrelation coefficient of order k takes a TRNG bitstream $\mathbf{b}$ of length N and determines how each bit is correlated to the bit k values away; the first-order autocorrelation coefficient a_1 is usually referred to as just "autocorrelation" [2].

$$a_k = \frac{\sum_{i=1}^{N-k}\left(\mathbf{b}[i] - \bar{\mathbf{b}}\right)\left(\mathbf{b}[i+k] - \bar{\mathbf{b}}\right)}{\sum_{i=1}^{N-k}\left(\mathbf{b}[i] - \bar{\mathbf{b}}\right)^2} \tag{2}$$

If an attacker knows enough information about a TRNG, then it can become susceptible to attacks. An attacker can crack a TRNG if they can predict its future values via a passive attack or forcibly change its values to desired outcomes via an active attack; cracking a TRNG compromises the information it makes [6]. For example, if the process used as a TRNG's core is actually deterministic instead of stochastic, then an attacker could perform a passive attack by analyzing the TRNG as a deterministic dynamical system instead of treating it like a black box. Cracking an RNG could allow one to steal confidential information, determine account passwords, or force outcomes in video games.

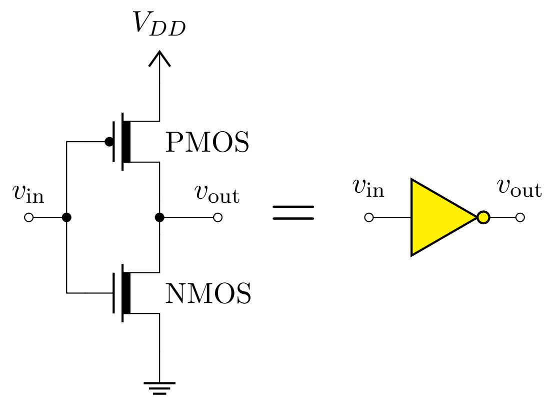
On using chaotic systems in RNGs. One would think that using chaotic systems as the core for generating random numbers is contradictory; they produce seemingly random signals driven by an internal underlying ruleset. Since a chaotic system only has a limited amount of information, chaotic RNGs will only produce a limited number of random bits and eventually run out of new states to generate random bits. Future bits produced by the chaotic RNG would be perfectly or near-perfectly correlated to the original set. Realistically, chaotic systems never truly adhere to their corresponding equation models thanks to random quantum or statistical effects. However, these added effects are insignificant to the observable chaos and are not considered as part of the chaotic RNG itself. Stipčević and Koç have suggested that researchers use chaotic systems in RNGs because they cannot tell the difference between chaos and noise, falsely believe that hard-to-describe systems are always random, or the chaotic system in use produces enough noise to overpower the underlying chaotic signals [2].

3 Understanding the Bistable Ring

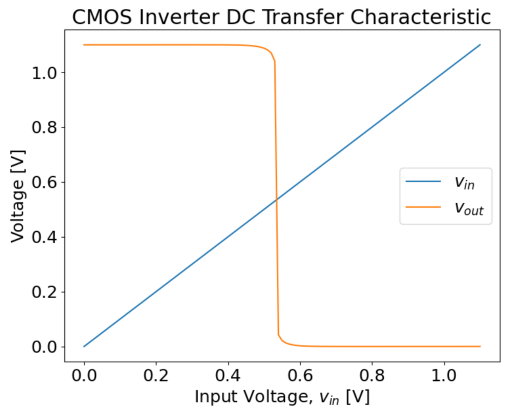
Recall that an inverter (a.k.a., NOT gate) is a digital logic building block that performs logical negation to output the opposite of its input. If the input to an inverter is "0" (logical low), it outputs "1" (logical high). A CMOS inverter contains a complementary pair of metal oxide semiconductor transistors that should ideally produce the logical opposite of its input, as shown in Figure 1a; however, it can also operate on analog voltages within and outside the range of its logical high and logical low. Inputting an intermediate value between logical low and logical high produces the voltage transfer characteristic shown in Figure 1b [7].

A bistable ring (BR) is a digital logic circuit made up of an even number of inverters connected to each other in a loop.[1] Figure 2 shows a BR with $N = 8$ stages. Each inverter Inv_i is associated

[1]An odd number of inverters connected to each other in a loop forms a ring oscillator (RO) [2].



(a) CMOS Inverter Internals



(b) CMOS Inverter DC Transfer

Figure 1: CMOS inverter's internals and DC transfer characteristic

with its input voltage signal $v_{\text{in}i}(t)$ and output voltage signal $v_{\text{out}i}(t)$. The output of each inverter Inv_i is equal to the input of the next inverter Inv_{i+1}, except for the last inverter Inv_{N-1} whose output is the input of the first inverter Inv_0. When the BR is powered on (or, more generally, released from an unstable state or metastable position), all inverters will simultaneously try to force their output voltages from logical low to logical high. However, this cannot happen because inverters should output the opposite logic value to their input value. Each inverter's input keeps changing since the previous inverter's output forces the change. As each inverter's input increases from logical low, the tendency for it to drive its corresponding output to logical high decreases. If an inverter's input goes beyond its metastable point (where the inverter's input has no preference on driving a logical low or logical high output), then the inverter will force its output to drop to logical low, with this tendency increasing as the inverter's input increases. Whether the BR will stabilize into one of its two states or continue oscillating further is dependent on process variation of the inverters' intrinsic properties and augmentative noise. For example, maybe the even-numbered outputs rise over their metastable points at a close-enough interval while the odd-numbered outputs stay above their metastable points; this would trigger a positive feedback loop where all inverters force the BR to settle onto a stable state. Alternatively, a voltage wave from previous stages in the ring could force the inverters to drop below the metastable point, forcing the inverters to oscillate further [3].

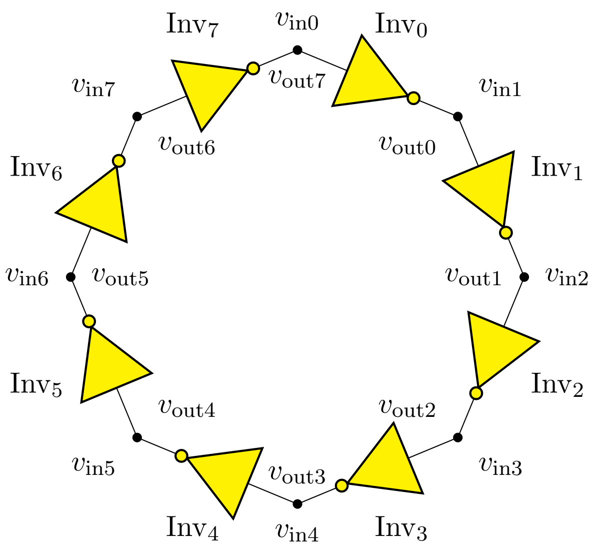


Figure 2: 8-Stage Bistable Ring (BR) with labeled inverters, input voltages, and output voltages

3.1 Applications of BRs

BRs are used in authentication, identification, and key obfuscation as part of the chip-specific electronic fingerprint of a physical unclonable function (PUF). A PUF is a device that uses its irreproducible physical characteristics to map a set of challenges (inputs) to a set of responses (outputs) [3]. A PUF's strength is a measure of its challenge-response pairs (CRPs), the number of different challenges that can map to the response set. The Bistable Ring PUF (BR-PUF), shown in Figure 3, is a strong PUF that maps the challenge set of all N-bit sequences to the response set $\{0, 1\}$ by returning the settled state of the challenge's corresponding N-stage BR in the BR-PUF as a single bit. Each bit in a challenge controls a single cell containing a multiplexer, a demultiplexer, and two NOR gates. While the reset input is low, the NOR gate acts as an inverter on its other input. The challenge bit's value selects which NOR gate in the cell is part of the challenge's corresponding BR. Since the BR-PUF has 2^N unique challenges, it can implement one of 2^N unique BRs [4].

Intel developed the TRNG in Figure 4 that incorporates a BR. This circuit has two inverters opposite each other (forming the BR) and two extra PMOS transistors; it has two stable states just like the BR [1]. Ideally, if everything is symmetric, its output should end up in either a low or high state when the transistors are driven high. Even though the output should be random, slight differences in the inverters' speeds or strengths would lead to a high imbalance between "1"

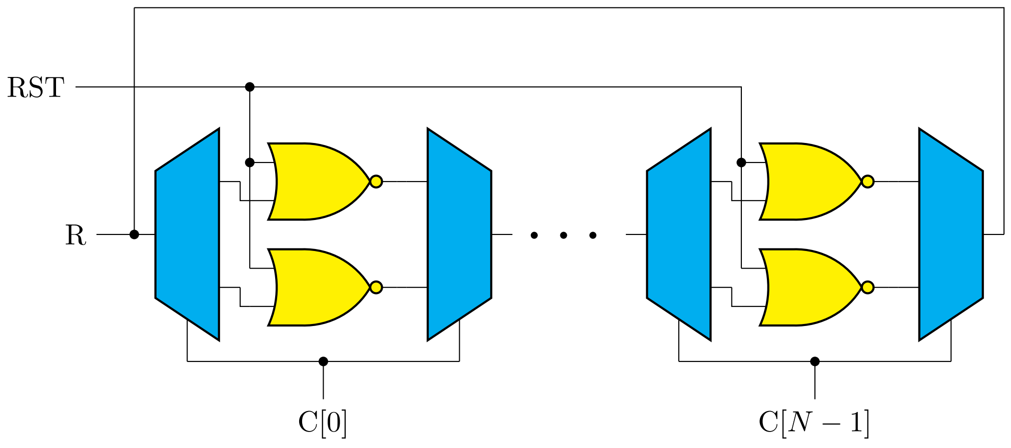


Figure 3: Bistable Ring PUF (BR-PUF) schematic, with an even number of stages

and "0" outputs, introducing bias into the system. To fix this, Intel added a current-injecting mechanism represented by the pulse voltage source V_{CLK} to help remove the bias. The resulting bitstream then seeds an external PRNG for post-processing; Intel's RNG may perform this because the individual generated bits from the circuit may be less random (e.g., high bias, high correlation, or both). Alternatively, Intel may have added the post-processing to comply with FIPS PUB-140, which explicitly does not endorse physical RNGs [2].

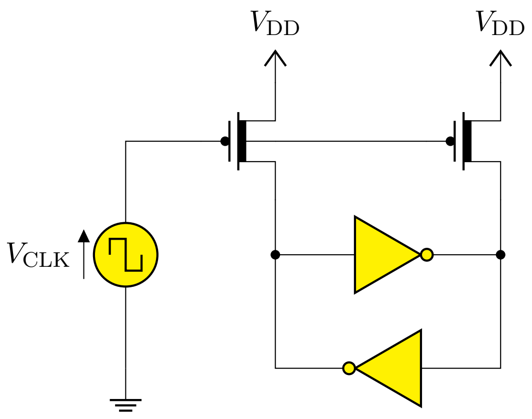


Figure 4: Schematic of Intel's RNG, excluding post-processing steps

3.2 Simulating BRs in Ngspice

We used the open-source data science platform Anaconda [8] and the community package repository `conda-forge` [9] to manage the required Python libraries and simulation tools for our results. The Predictive Technology Model (PTM) website from Arizona State University provides accurate transistor models compatible with most SPICE-based simulators. The provided library files are BSIM3 and BSIM4 (Berkeley Short-channel IGFET Model) transistors with process channel lengths ranging from 7 nm to 180 nm; we used the 45 nm low-power transistor models in our simulations [10]. We performed our circuit simulations using the cross-platform, open-source, SPICE-compatible circuit simulator Ngspice [11] since we found it the easiest to work with and automate compared to other SPICE flavors. We used the `PySpice` Python library to automate our Ngspice circuit simulations [12].

Each CMOS inverter in the BR consists of a PMOS transistor and an NMOS transistor (recall Figure 1), both of which have specifications contained within the ASU PTM 45 nm low-power transistor model card. We expected that the channel widths (w), channel lengths (l), and threshold voltages (V_{TH0}) of each transistor would be affected by process noise in semiconductor manufacturing; these properties can be modified directly when instantiating a MOS transistor with Ngspice syntax using `l`, `w`, and `delvto` respectively [11]. We set the default channel length and threshold voltages in accordance with the 45 nm PTM model card and used 120 nm from Jain et al. [13] as the default channel width. Since we can represent one inverter with six properties (PMOS length, PMOS width, PMOS V_{TH0}, NMOS length, NMOS width, and NMOS V_{TH0}), we can subsequently represent an N-stage BR as a $6 \times N$ array of parameters as shown in Table 2a. Each row i in the array corresponds to property i; each column j in the array corresponds to inverter j. Alternatively, we can equivalently represent an N-stage BR as a $6 \times N$ array of percent changes from the default values as shown in Table 2b. This alternative BR representation was useful for simulating transistor process variations in our Monte Carlo simulations. Our BR builder functions can produce a `PySpice` Circuit object containing the desired BR's SPICE netlist specified by either sub-table in Table 2. Both BR builder functions also include an isolated pulse voltage source V_{CLK} with a specified period to perform a virtual sample-and-hold for simulation.

Our simulation functions simulate a given BR circuit with the given initial voltage conditions and simulation options. By running the circuit simulation with a smaller maximum step time than the clock's period (i.e., oversampling), the clock voltage source V_{CLK} forces Ngspice to calculate voltage transients at time points that are multiples of V_{CLK}'s period. Our virtual sample-and-hold function used `pandas` [14], [15] and `numpy` [16] to filter the data from the circuit simulation to produce periodic time-series for analysis. This method circumvents Ngspice's inability to obtain

Table 1: Two ways to represent the same BR: parameters or percent changes

(a) Array of Parameters

	$\mathbf{Inv_0}$	$\mathbf{Inv_1}$	...	$\mathbf{Inv_{N-1}}$
$l_{P,i}$ [nm]	45	45	...	47.25
$w_{P,i}$ [nm]	120	120	...	126
$V_{TH0P,i}$ [V]	-0.587	-0.587	...	-0.616
$l_{N,i}$ [nm]	45	45	...	47.25
$w_{N,i}$ [nm]	120	120	...	126
$V_{TH0N,i}$ [V]	0.623	0.623	...	0.623

(b) Array of Percent Changes

	$\mathbf{Inv_0}$	$\mathbf{Inv_1}$	...	$\mathbf{Inv_{N-1}}$
$l_{P,i}$ [nm]	0%	0%	...	+5%
$w_{P,i}$ [nm]	0%	0%	...	+5%
$V_{TH0P,i}$ [V]	0%	0%	...	+5%
$l_{N,i}$ [nm]	0%	0%	...	+5%
$w_{N,i}$ [nm]	0%	0%	...	+5%
$V_{TH0N,i}$ [V]	0%	0%	...	+5%

periodic circuit simulation transients. In addition, we used `matplotlib` [17] to render simulation results in graphs like the one in Figure 5.

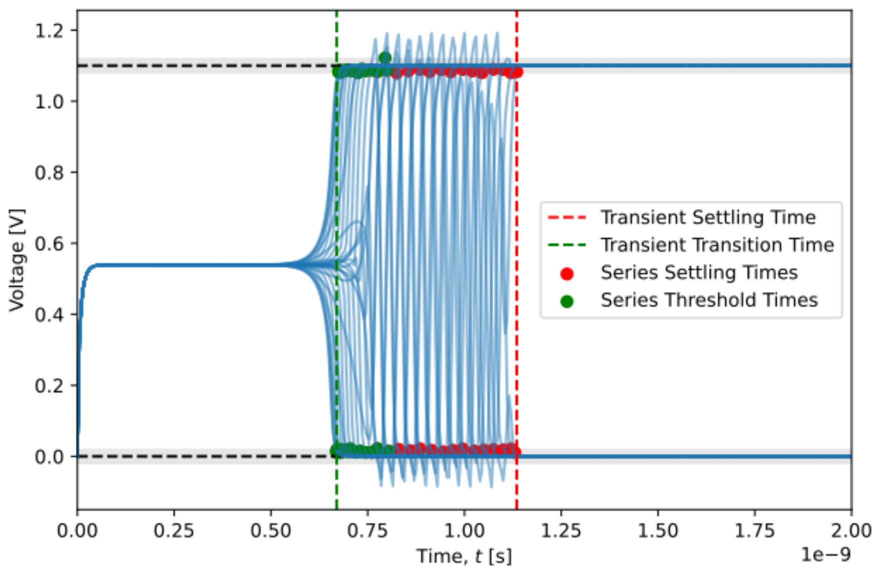


Figure 5: Voltage transients from a 16-stage BR simulation showing transition, oscillation, and settling regions

Initial test simulations of BRs showed that it behaved in at most three separate ways, like the 16-stage BR simulation result in Figure 5. Splitting the analysis of the BR's transient results into these three different regions helped us understand the BR's different convergence patterns. To separate the regions for analysis, we developed the logic-low settling thresholds $[lo_{\min}, lo_{\max}]^\top = V_{\text{DD}}\,[-ST, ST]^\top$ and logic-high settling thresholds $[hi_{\min}, hi_{\max}]^\top = V_{\text{DD}}\,[1-ST, 1+ST]^\top$ where $ST = 2\%$ like the MATLAB `lsiminfo` function used for computing linear response characteristics [18]. Using these thresholds, we split the BR's transient simulation into analyzable regions like

Vasyltsov et al.'s TRNG analysis [19]. Each inverter output $v_{\text{in}i}$ of an N-stage BR has the following regions and threshold times, listed below in chronological order:

1. Transition Region, $[0, t_{\text{transition}}]$: The inverter turns on at time $t = 0$, and the feedback loop in the BR begins.

2. Transition Time, $t_{\text{transition}}$: the time where the inverter's input voltage $v_{\text{in}i}(t)$ first exits the range $[lo_{\max}, hi_{\min}]$ after entering this range.

3. Oscillation Region, $(t_{\text{transition}}, t_{\text{settling}})$: the inverter's input voltage has completed transition; it *may* oscillate between V_{SS} and V_{DD} if the transition and settling times are far enough apart.

4. Settling Time, t_{settling}: the first time where both the current value and all future values of $v_{\text{in}i}(t)$ are within the range $[lo_{\min}, lo_{\max}] \cup [hi_{\min}, hi_{\max}]$.

5. Settling Region, $[t_{\text{settling}}, t_{\text{end}}]$: the inverter's input voltage *may* settle into either V_{SS} or V_{DD}.

Figure 5 shows the transition time $t_{\text{transition}}$ and settling time t_{settling} for each inverter in a 16-stage BR simulation as green points and red points, respectively. The first transition region marked with the dotted green line and the last settling region marked with the dotted red line act as separators between the BR's own transition, oscillation, and settling regions. We noticed that the transition region is present in every BR simulation, but not every BR shows oscillation or settling behavior within the set simulation time of 5 ns.

4 Methods

Prior work has already shown that the BR's convergence to stable states is nonlinear in nature [4]; this implies that deriving a single closed-form solution for the BR's signals from just its transistor properties would be difficult. Instead, we can use tools from nonlinear dynamics and time-series analysis to glean information about the BR from just observations.

The tools from nonlinear dynamics introduced in this section form the components of the Chaos Decision Tree Algorithm (CDTA). This algorithm is a recent innovation by Toker et al. that can describe the behavior of an input time-series $\mathbf{y}$ from a dynamical system as nonstationary, stochastic, periodic, or chaotic. The algorithm distinguishes which behavior $\mathbf{y}$ exhibits via process

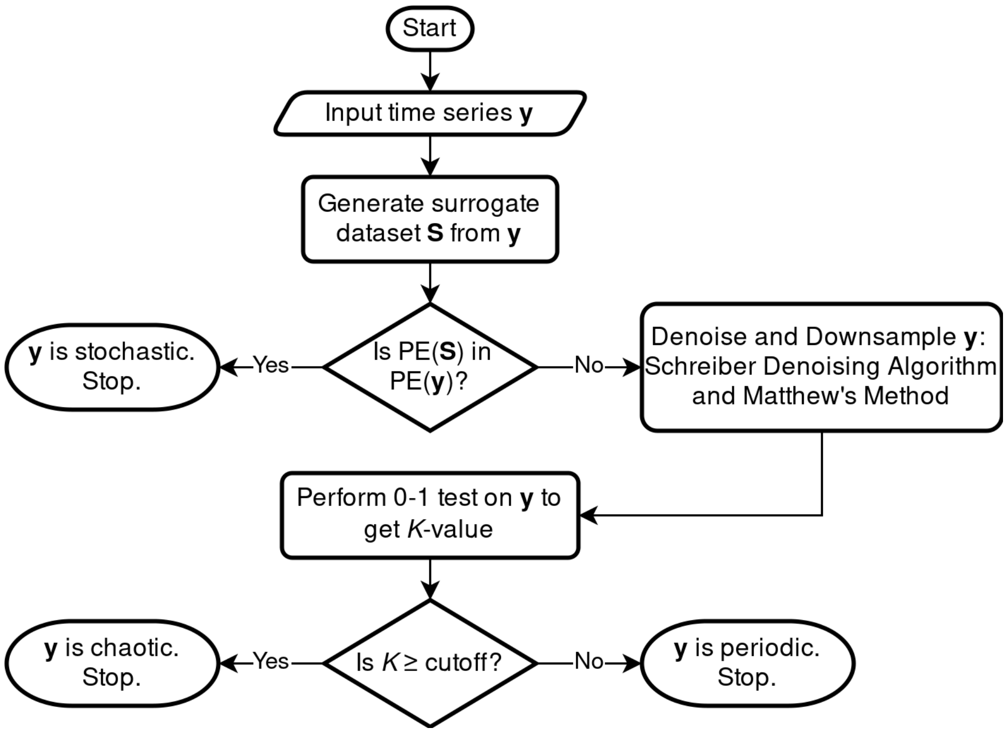

Start
Input time series **y**
Generate surrogate dataset **S** from **y**
Is PE(**S**) in PE(**y**)?
Yes
y is stochastic. Stop.
No
Denoise and Downsample **y**: Schreiber Denoising Algorithm and Matthew's Method
Perform 0-1 test on **y** to get *K*-value
Is *K* ≥ cutoff?
Yes
y is chaotic. Stop.
No
y is periodic. Stop.

Figure 6: Simplified flowchart of the Chaos Decision Tree Algorithm (CDTA)

of elimination using concepts from nonlinear dynamics and linear time-system analysis [20], [21]. As shown in Figure 6, The CDTA first relies on the surrogate data method to determine if the time-series $\mathbf{y}$ is not stochastic; after post-processing $\mathbf{y}$ using Schreiber's denoising algorithm and Matthews' Method, a modified version of the 0-1 test for chaos determines whether $\mathbf{y}$ is chaotic or periodic.

In addition to classifying the behavior of the input time-series $\mathbf{y}$, the CDTA optionally provides permutation entropy and 0-1 test results of $\mathbf{y}$ depending on its behavior. These optional metrics are indirect but quantitative measures of $\mathbf{y}$'s complexity and chaos, respectively. This leads to the following question: how can we make the most complex BR? We can answer this problem with genetic algorithms since they can maximize or minimize functions without closed-form representations.

4.1 Permutation Entropy

Permutation entropy is an indirect measure of the complexity of an input time-series $\mathbf{y}$, calculated by analyzing occurrences of the relative amplitude patterns in sets of its consecutive elements [22]–[24]. To calculate permutation entropy $PE(\mathbf{y}, n, \tau)$ of order $n \geq 2$ and lag $\tau \geq 1$ of an input time-series $\mathbf{y} = \{y_1, y_2, \ldots, y_T\}$, first partition it into sub-vectors of length n and lag τ. Next, assign one of $n!$ permutations $\boldsymbol{\pi}_i$ to each sub-vector, in order of their rank. If two elements in a sub-vector have the same value, then the element with the lower index has the lower rank. Third, calculate the pattern probability distribution $\{p(\boldsymbol{\pi}_i) \, \forall \, i = \{1, \ldots, n!\}\}$ by computing the relative frequencies of each possible permutation $\boldsymbol{\pi}_i$ in the list of sub-vectors as shown in Equation 3. Finally, calculate the Shannon entropy of the relative frequency distribution shown in Equation 4 [24].

$$p(\boldsymbol{\pi}_i) = \frac{\text{number of occurrences where } (y_j, \ldots, y_{j+(n-1)\tau}) \text{ has type } \boldsymbol{\pi}_i}{\text{Total number of sub-vectors in } \mathbf{y}} \tag{3}$$

$$PE(\mathbf{y}, n, \tau) = -\sum_{i=1}^{n!} p(\boldsymbol{\pi}_i) \log_2 p(\boldsymbol{\pi}_i) \tag{4}$$

For example, let us calculate the permutation entropy of order $n = 3$ and lag $\tau = 1$ of the time-series $\mathbf{y}$ shown in Equation 5. We can write the set of $n! = 6$ permutations $\boldsymbol{\pi}$ as shown in Equation 6. Next, we split $\mathbf{y}$ into a list of sub-vectors $\mathbf{D}$ as shown in Equation 7, assigning each sub-vector $\mathbf{d}_i \in \mathbf{D}$ to its corresponding permutation $\boldsymbol{\pi}_i \in \boldsymbol{\pi}$. For each permutation $\boldsymbol{\pi}_i \in \boldsymbol{\pi}$, we calculate its relative frequency as shown in Equation 8. Finally, we calculate that the permutation entropy of $\mathbf{y}$ of order 3 and lag 1 is equal to 2.5 as shown in Equation 9. The Shannon entropy of a distribution measures its unpredictability; given the relative frequency distribution shown in Equation 8, it would be hard to predict what permutation would come next in $\mathbf{y}$.

$$\mathbf{y} = \{2, 5, -3, 3, 0, -2, 1, 5, 3, -1\} \tag{5}$$

$$\boldsymbol{\pi} = \{(012), (021), (102), (120), (201), (210)\} \tag{6}$$

$$\mathbf{D} = \{\underbrace{\{2, 5, -3\}}_{(120)}, \underbrace{\{5, -3, 3\}}_{(201)}, \underbrace{\{-3, 3, 0\}}_{(021)}, \ldots, \underbrace{\{1, 5, 3\}}_{(021)}, \underbrace{\{5, 3, -1\}}_{(210)}\} \tag{7}$$

$$p((012)) = p((102)) = p((201)) + p((210)) = \frac{1}{8}; \quad p((021)) = p((210)) = \frac{2}{8} \tag{8}$$

$$PE(\mathbf{y}, 3, 1) = -\sum_{i=1}^{3!} p(\boldsymbol{\pi}_i) \log_2 p(\boldsymbol{\pi}_i) = 2.5 \tag{9}$$

Toker et al. performed experiments that empirically demonstrate that permutation entropy tracks the largest Lyapunov exponent of typical chaotic systems serving as benchmark functions: the logistic map, the tent map, and the Duffing Oscillator [21]. The permutation entropy calculation gives a quantitative rating of a time-series' complexity, shown graphically in Figure 7: a time-series with a PE of 0 has only one type of permutation, while a time-series with a PE of $\log_2(n!)$ has an equal relative distribution of every permutation. The CDTA comes prepackaged with the `petropy` function developed by Andreas Müller [25]. However, we also use the `permutation_entropy()` function from `ordpy` [26] in our genetic algorithm as an equivalent alternative to reduce the number of calls to MATLAB.

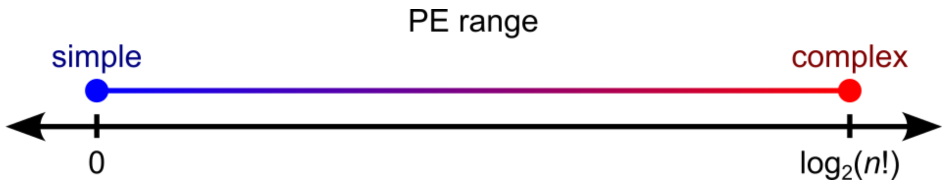


Figure 7: Range of values of permutation entropy with corresponding interpretations

4.2 The Surrogate Data Method

The surrogate data method tests whether a given time-series does or does not satisfy a specific hypothesis; it can also test whether an interaction exists between two time-series. It contains three components: the null hypothesis, a statement that could describe $\mathbf{y}$; the surrogate algorithm, the method used to generate new time-series that do not follow the null hypothesis; and the discriminating statistic $q(\cdot)$, a function that separates the time-series that obey the null hypothesis from time-series that do not [23]. The CDTA uses the surrogate data method to examine if the input time-series $\mathbf{y}$ is not stochastic. Toker et al. experimentally showed that the best-performing discriminating statistic was permutation entropy with order $n = 8$ and lag $\tau = 1$ [21].

The first step of the surrogate data method is to choose a null hypothesis to test against the time-series $\mathbf{y}$ (e.g., the statement "$\mathbf{y}$ follows a linear stochastic process"). The goal of the surrogate method is to reject the null hypothesis, proving that it does not describe $\mathbf{y}$ [23]. However, if the surrogate data method fails to reject the null hypothesis, it does not necessarily mean that $\mathbf{y}$ does follow the null hypothesis; it could mean that the surrogate data method was not properly set up enough to discriminate between the two types of time-series [27].

The second step is to choose a surrogate algorithm that corresponds to the chosen null hypothesis. The surrogate algorithm generates the surrogate time-series dataset $\mathbf{S}$; each surrogate time-series in $\mathbf{S}$ is an augmented copy of the original time-series $\mathbf{y}$ that obeys the defining properties of the chosen null hypothesis but removes other properties [23]. For example, the Amplitude-Adjusted Fourier Transform (AAFT) surrogate algorithm matches the null hypothesis regarding the linearity and destroys other relationships in the linear time-series. The AAFT surrogate algorithm first transforms the time-series $\mathbf{y}$ into the time-series $\mathbf{x}$ by conforming it to realizations of a Gaussian distribution in such a way that the rank of each value in $\mathbf{y}$ is preserved in $\mathbf{x}$. Next, the AAFT algorithm calculates the Fourier transform $\mathcal{F}\{\mathbf{x}\}$, turning the adjusted time-series $\mathbf{x}$ into a sequence of magnitudes and phases. A single surrogate time-series $\mathbf{s}_i \in \mathbf{S}$ shares the same sequence of magnitudes of $\mathcal{F}\{\mathbf{x}\}$, but the sequence of phases contains random values chosen from a uniform distribution with the range $[0, 2\pi]$ [27].

The third step of the surrogate data method is to apply a discriminating statistic function $q(\cdot)$, also dependent on the null hypothesis, to the original time-series $\mathbf{y}$ and the elements in the surrogate time-series dataset $\mathbf{S}$ [23]. When the discriminating statistic $q(\cdot)$ evaluates the elements in the surrogate dataset $\mathbf{S}$, it creates a distribution of discriminating statistic results that can be compared to the original time-series' result [27]. If $q(\mathbf{y})$ is not within the range of values in $q(\mathbf{S})$, then the null hypothesis can be rejected within some confidence interval. However, if $q(\mathbf{y})$ is within the range of values in $q(\mathbf{S})$, then the null hypothesis is *not* necessarily confirmed. The surrogate data method can only figure out how a time-series is *not* characterized [23].

4.2.1 Other Surrogate Algorithms

The surrogate data method is flexible enough to handle several null hypotheses, each with corresponding surrogate algorithms [27]. The CDTA harnesses this flexibility by offering different surrogate algorithms to test whether the input time-series $\mathbf{y}$ displays linearly stochastic and/or non-linearly stochastic behaviors, including but not limited to the Random Permutation (RP), Cyclic Phase Permutation (CPP), and a hybrid of the AAFT and CPP algorithms [20].

The Random Permutation (RP) surrogate algorithm can check whether an input time-series $\mathbf{y}$ possesses any temporal structure or just contains uncorrelated noise. Each surrogate time-series $\mathbf{s}_i \in \mathbf{S}$ generated by the RP surrogate algorithm is a randomly shuffled version of the original time-series. The mean, variance, and histogram distribution of each surrogate time-series $\mathbf{s}_i$ is identical to the original time-series $\mathbf{y}$. The corresponding null hypothesis of the RP surrogate algorithm is "$\mathbf{y}$ is fully described by a sequence of independent and identically distributed (IID) random variables" [27].

The Cyclic Phase Permutation (CPP) surrogate algorithm's null hypothesis is "two systems have independent phase dynamics". The CPP algorithm works directly with the phase of an input time-series $\mathbf{y}$, breaking the relationship between how the evolution of one system's phase dynamics would affect the evolution of another system's phase dynamics. The CPP surrogate algorithm first extracts the instantaneous phase $\boldsymbol{\phi}$ of the input time-series $\mathbf{y}$ by using either the Hilbert or wavelet transforms; the algorithm then wraps this phase into the range $[0, 2\pi)$. Discontinuities in the instantaneous phase $\boldsymbol{\phi}$ denote the boundaries of individual cycles within the time-series $\mathbf{y}$. The CPP algorithm does not assume that the time-series $\mathbf{y}$ begins or ends with a complete cycle; instead, the incomplete beginning and end cycles are preserved while the complete intermediate cycles are shuffled for each surrogate time-series $\mathbf{s}_i \in \mathbf{S}$ [27].

Toker et al. developed the AAFT+CPP surrogate algorithm and empirically demonstrated that it was the most effective surrogate algorithm in the CDTA. With this option, the CDTA will run the surrogate data method twice: once with the AAFT surrogate algorithm and once with the CPP surrogate algorithm. If one or both surrogate sub-methods fail, then the surrogate method fails overall; otherwise, the surrogate method passes overall. Toker et al. empirically demonstrated the AAFT+CPP algorithm's efficacy through tests with both simulated and realized systems [21].

4.3 Matthews' Method

The CDTA uses Matthews' method to improve accuracy in cases where the input time-series $\mathbf{y}$ is oversampled. The first step of Matthews' method calculates the measure η, equal to the range of $\mathbf{y}$ divided by the average difference between consecutive time points, as shown in Equation 10.

$$\eta = \frac{\max(\mathbf{y}) - \min(\mathbf{y})}{\text{avg(diff. between consecutive time points)}} \tag{10}$$

The time-series $\mathbf{y}$ is oversampled while $\eta > 10$. To fix this, downsample $\mathbf{y}$ by two and recalculate η until $\eta \leq 10$ or $\mathbf{y}$ has less than 100 time-points [21].

4.4 The Modified 0-1 Test for Chaos

The CDTA's last step uses a modified version of the "0-1 test for chaos" to figure out whether the time-series $\mathbf{y}$ is either chaotic or periodic [21]. This test was originally developed by Gottwald and Melbourne; it works directly with a dynamical system's measurements to determine whether it exhibits regular or chaotic dynamics using a one-dimensional time-series $\mathbf{y}(j)\,\forall j \in \{1, 2, \ldots, N\}$ as input for a Monte Carlo experiment. First, the data from $\mathbf{y}$ and a uniform-randomly selected value of $c \in (0, 2\pi)$ drive the two-dimensional system of difference equations in Equation 11 [28].

$$\begin{aligned} p_c(n+1) &= p_c(n) + \mathbf{y}(n)\cos(cn) \\ q_c(n+1) &= q_c(n) + \mathbf{y}(n)\sin(cn) \end{aligned} \tag{11}$$

Equation 12 is the solution to the system in Equation 11 for a given c-value in the range $(0, \pi)$. If the underlying dynamics of the input time-series $\mathbf{y}$ are regular (i.e., periodic or quasi-periodic), then the dynamics of $p_c(n)$ and $q_c(n)$ are bounded. However, if the underlying dynamics are irregular (i.e., chaotic), then the dynamics of $p_c(n)$ and $q_c(n)$ will display diffusive Brownian-like motion with zero drift [28]. The next steps of the 0-1 test figure out the dynamics of $p_c(n)$ and $q_c(n)$ to determine the corresponding underlying dynamics of $\mathbf{y}$ [21], [28].

$$\begin{aligned} p_c(n) &= \sum_{j=1}^{n} \mathbf{y}(j)\cos(jc) \\ q_c(n) &= \sum_{j=1}^{n} \mathbf{y}(j)\sin(jc) \end{aligned} \tag{12}$$

The original 0-1 test tracked how the time-averaged mean square displacement grew to analyze the dynamics of $p_c(n)$ and $q_c(n)$ as n increased [28]. The modified 0-1 test adds noise to this mean-square displacement measurement as shown in Equation 13: the value η_n is a sample of a uniformly distributed random variable with the range $\left[-\frac{1}{2}, \frac{1}{2}\right]$, and the parameter σ controls the noise level [21].

$$M_c(n) = \frac{1}{N}\sum_{j=1}^{N}\left([p_c(j+n) - p_c(j)]^2 + [q_c(j+n) - q_c(j)]^2\right) + \underbrace{\sigma\eta_n}_{\text{noise}} \tag{13}$$

If the noisy time-averaged mean square displacement $M_c(n)$ stays bounded as n increases, then

$p_c(n)$ and $q_c(n)$ are also bounded for the fixed value c. However, if the noisy time-averaged mean square displacement $M_c(n)$ grew linearly as n increased, then $p_c(n)$ and $q_c(n)$ display the Brownian-like motion. The growth rate K_c can be calculated as the Pearson correlation coefficient between the vector $\xi = [1, 2, \dots, N/10]^\top$ and $\Delta = [M_c(1), M_c(2), \dots, M_c(N/10)]^\top$, as shown in Equation 14 [28].

$$K_c = \text{corr}(\xi, \Delta) = \frac{\text{cov}[\xi, \Delta]}{\sqrt{\text{var}[\xi]\text{var}[\Delta]}} \in [-1, 1] \tag{14}$$

The K_c-values are computed for at least one hundred different values of c randomly sampled from a uniform distribution with bounds $(0, 2\pi)$. The final output of the test K is the median of the observed K_c-values [21].

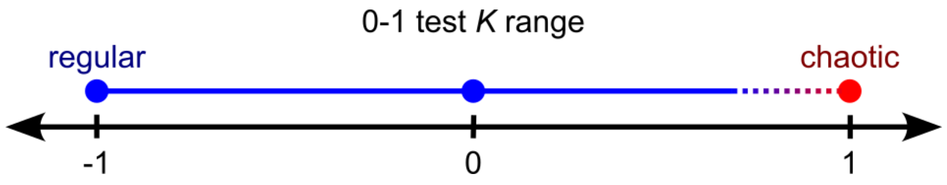


Figure 8: Range of the modified 0-1 test's output K and corresponding underlying dynamics of the input time-series $\mathbf{y}$

If K approaches 1, then $p_c(n)$ and $q_c(n)$ have Brownian-like motion for most sampled values of c; therefore, the 0-1 test states that the underlying dynamics of the time-series $\mathbf{y}$ are irregular (i.e., chaotic). Otherwise, if K approaches 0, then $p_c(n)$ and $q_c(n)$ have bounded dynamics for most samples of c; therefore, the 0-1 test states that the underlying dynamics of the time-series $\mathbf{y}$ are regular [28]. Realistically, the 0-1 test's output K will be in the range $(-1, 1)$ and unequal to neither 0 nor 1. Toker et al. demonstrated that showed that systems with K-values below 0.72 will show regular behavior, while K-values between 0.72 and 1.0 had more ambiguous behavior. In this scenario, the CDTA will automatically calculate the optimal K cutoff value to differentiate between periodic and chaotic systems as a function of the length of the time-series $\mathbf{y}$ [21]. We can therefore define the total range of the function as shown in Figure 8.

4.5 Modifications to the Chaos Decision Tree Algorithm

Some of the BR voltage signals were incompatible with the CPP surrogate algorithm because the signals did not complete a full phase cycle. To circumvent this issue, we propose the AAFT+RP surrogate algorithm. Like the AAFT+CPP surrogate method described in Section 4.2.1, this option makes the CDTA run two separate surrogate sub-methods: once with the AAFT surrogate algorithm and once with the RP surrogate algorithm. If one or both surrogate sub-methods fail, then the surrogate method fails overall; otherwise, the surrogate method passes overall.

4.6 Genetic Algorithms

Genetic algorithms (GAs) are methods of finding solutions to parameter optimization problems by mimicking biological evolution. This method is useful for solving problems that do not have closed-form solutions [29]. In our case, it was an effective method for optimizing the BR's complexity.

We have previously shown in Table 1b that we can represent an N-stage BR as a $6 \times N$ array of percent changes; unraveling this array for a 64-stage BR produces a list of 384 percent changes. We can call this list the genome of an individual BR. One genome contains 384 genes, the process variation percent changes in the BR. The population of individual BRs is the list of all individual BRs in the GA's current iteration, or generation. The GA starts with randomly generated individual BRs in the first generation. The fittest individuals in each generation will pass their genes to the next generation, creating a new population like in biological evolution [29].

How do we define what fittest means in this context? Our goal for the GA is to find the parameters that describe the most complex BR; we can rephrase this question into a cost function that we want to minimize. This gives us a quantitative fitness value of an individual BR in the population, equal to the negative value of our cost function. We decided that our cost function should be the negative permutation entropy of an individual BR's first inverter oscillation region if applicable, and 1.0 otherwise. This way, the most complex BRs will have the lowest cost function result. We can perform a selection operation to choose the best BRs in our current generation to populate the next generation. We chose tournament selection with size 3: we partition the individuals into subgroups of threes and preserve the individual with the lowest cost function result in each subgroup [29].

Finally, we use the preserved individuals from the current generation to populate the next generation. We begin making the next generation by breeding the preserved individual BRs from the current generation. First, we partition the fittest individuals of the current generation into pairs of parents to make children. The crossover operation combines and/or rearranges the genomes of the two parent BRs to form the child BRs' genomes. In addition to receiving the parents' genes, a mutation operator may augment some of the child BRs' resulting genes; this adds more diversity to the resulting population. The child BRs constitute the population of the GA's next generation. This process of evaluating, selecting, and repopulating continues until the GA meets a stopping condition [29].

5 Transient Analysis of BRs

We analyzed nine 16-stage BRs, nine 32-stage BRs, and nine 64-stage BRs to further our understanding of the BR's different behavior. We further subdivided each BR group into what regions were present in their transient simulation (transition-oscillation, transition-settling, transition-oscillation-settling).[2] We analyzed the transition, oscillation, and settling regions of each BR's N voltage outputs using the CDTA with the new AAFT+RP surrogate algorithm. Each row in Table 2 corresponds to a group of three unique N-stage BRs that we observed having the same behavior. We tallied the number of CDTA results (**S**tochastic, **P**eriodic, **C**haotic, **E**rror) per region for each group of BRs on the three rightmost columns.

Table 2: CDTA classification results tabulated from each example BR's output

		Trans. Result				**Osc. Result**				**Sett. Result**			
Stages, N	**Behavior**	**S**	**P**	**C**	**E**	**S**	**P**	**C**	**E**	**S**	**P**	**C**	**E**
16	Trans.-Osc.	0	48	0	0	0	39	9	0	-	-	-	-
	Trans.-Osc.-Sett.	0	33	0	15	0	15	33	0	0	48	0	0
	Trans.-Sett.	0	48	0	0	-	-	-	-	0	48	0	0
32	Trans.-Osc.	0	72	0	24	0	2	94	0	-	-	-	-
	Trans.-Osc.-Sett.	0	96	0	0	1	3	61	31	0	75	4	17
	Trans.-Sett.	0	82	0	14	-	-	-	-	0	96	0	0
64	Trans.-Osc.	0	192	0	0	0	11	181	0	-	-	-	-
	Trans.-Osc.-Sett.	0	181	0	11	0	0	192	0	1	171	2	18
	Trans.-Sett.	0	192	0	0	-	-	-	-	0	192	0	0

The AAFT+RP surrogate algorithm failed to run on some of the signals because of issues within the CDTA's AAFT surrogate algorithm implementation. In all the example BR simulations, most of the transition and settling regions were periodic. From a dynamical systems perspective, this makes sense since the BR is close to an unstable equilibrium in the transition region and

[2]We did not observe any BRs with eight stages or less that had an oscillation region.

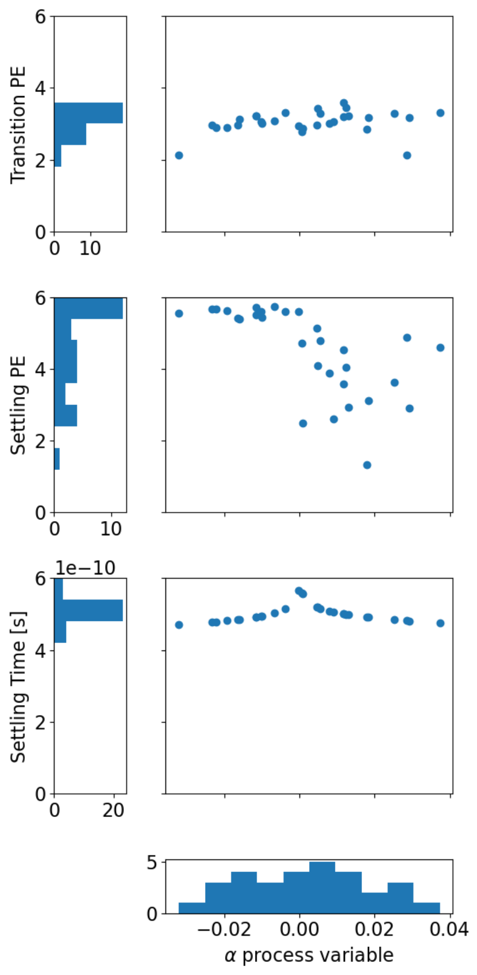
approaches a stable equilibrium in the settling region. Out of all groups, the 16-stage BRs had the most simulations where oscillation regions were classified as periodic. In comparison, most of the oscillation regions in the 32-stage and 64-stage BR simulations were classified as chaotic.

6 Monte Carlo Analysis

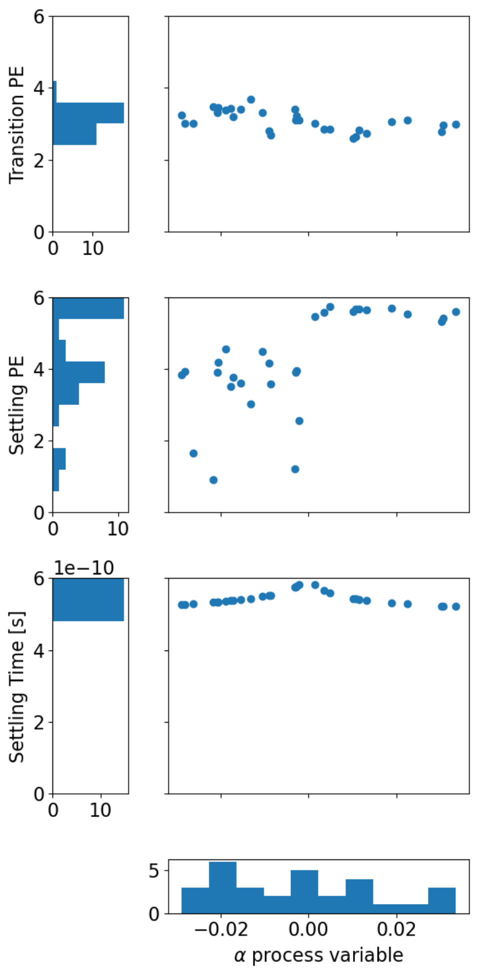
Using our BR builder functions for percent changes (as demonstrated in Table 1b), we set up the following Monte Carlo experiments for 64-stage BRs where each Monte Carlo experiment varied a different number of inverters' parameters by a sample of the random variable $\alpha \sim \mathcal{N}(0, 1.\bar{6}\%)$ corresponding to a zero-mean Gaussian distribution with 6σ region of 5%. We ran each Monte Carlo experiment below with thirty unique 64-stage BRs. We chose to vary the PMOS and NMOS V_{TH0} parameters since we believed that they had a nonlinear impact on the BR system.

1. Change the BR's first inverter's PMOS V_{TH0} by a sample of the random variable α (in Figure 9a).
2. Change the BR's first inverter's NMOS V_{TH0} by a sample of the random variable α (in Figure 9b).
3. Change the PMOS V_{TH0} of $k = \log_2(64) = 6$ randomly selected inverters in the BR by the same sample of the random variable α (in Figure 10a).
4. Change the NMOS V_{TH0} of $k = \log_2(64) = 6$ randomly selected inverters in the BR by the same sample of the random variable α (in Figure 10b).
5. Change all the PMOS V_{TH0} values in the BR by the same sample of the random variable α (in Figure 11a).
6. Change all the NMOS V_{TH0} values in the BR by the same sample of the random variable α (in Figure 11b).

The default control case for these Monte Carlo experiments is when the BR's inverters all have default parameters provided by the transistor SPICE model card. When the BR turns on, all inverters' voltages trend upward towards $V_{DD}/2$. We expected that the voltage trajectories would eventually settle with all inverter voltages equal to $V_{DD}/2$, finding a stable equilibrium point seated in an unstable equilibrium line in the BR's state space. However, we saw that the 64-stage control BR oscillated for more than 5 ns. We hypothesize that these results did not match our expectations because of imperfections within Ngspice when simulating unstable equilibria.



(a) First inverter's PMOS V_{TH0}



(b) First inverter's NMOS V_{TH0}

Figure 9: Monte Carlo simulations for varying the first inverter's property by the percent change α.

Figure 9 shows the results of the first two Monte Carlo experiments: each BR has the first inverter's PMOS V_{TH0} or NMOS V_{TH0} augmented by a sample of α. The resulting BRs did not oscillate, showing us that modifying a single parameter threw off the BR's instability altogether. The decrease in settling time as a function of deviation from $\alpha = 0$ follows an inversely proportional curve. All transition regions and all settling regions in Figure 9 were periodic according to the

CDTA through the modified 0-1 test; no K-value in this Monte Carlo experiment exceeded the cutoff threshold needed to be chaotic.

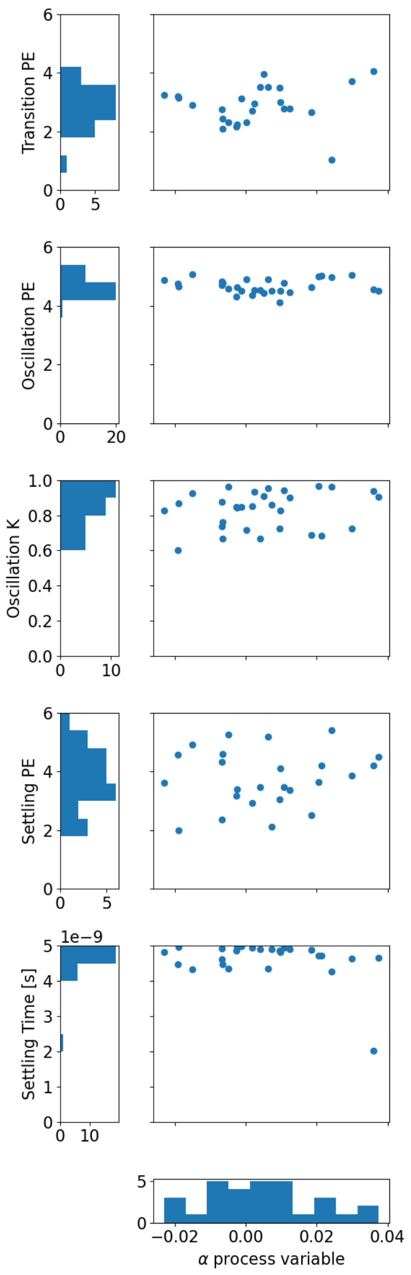
Figure 10 shows the results of the next two Monte Carlo experiments: a random selection of $k = \log_2(64) = 6$ inverters in each BR simulation were randomly chosen to have their PMOS V_{TH0} or NMOS V_{TH0} values augmented by the same sample of α. The CDTA classified all transition regions in Figure 10 as periodic. In Figure 10a, one settling region was stochastic and one settling region was chaotic; all remaining settling regions were periodic. Figure 10a shows two distinct groups of oscillation K-values: the group with K-values in $[0.6, 0.72]$ were periodic while the rest were chaotic. A similar trend occurred in Figure 10b: K-values in $[0.59, 0.72]$ were periodic.

Figure 11 shows the results of the last two Monte Carlo experiments: all inverters in each BR simulation had their PMOS V_{TH0} or NMOS V_{TH0} values augmented by the same sample of α. All transition regions in Figure 11 were periodic. In Figure 11a, one settling region was chaotic while the remaining settling regions in Figure 11 were periodic. Like the previous case, there was a wider split in the distribution of K-values of the oscillation region that separated the periodic and chaotic trajectories.

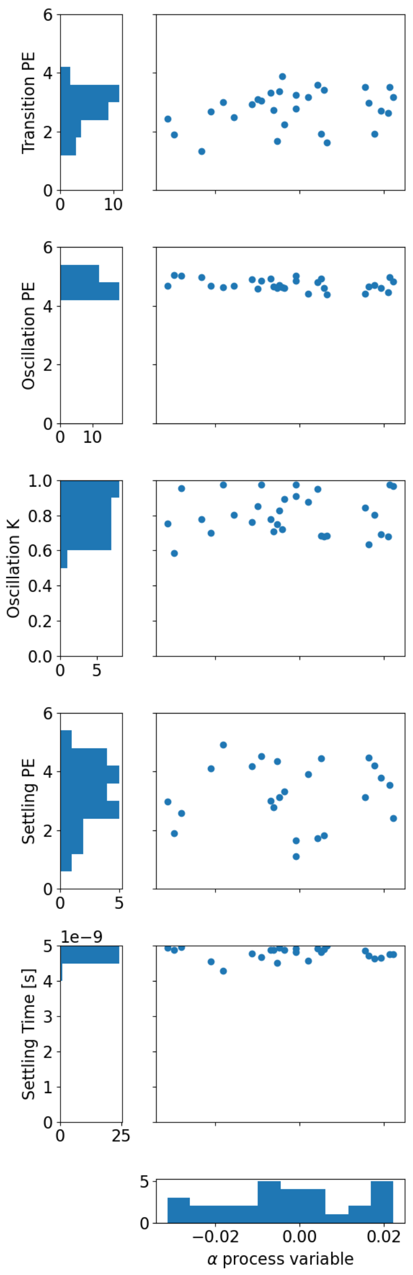
The settling time distributions for both Figure 10 and Figure 11 show two main clusters of points close to the end of the simulation time (at 5 ns). As mentioned in Section 3.2, the settling time is the first time when all future values of a single voltage transient are within the settling region; however, the circuit's other voltage trajectories are still oscillating at this point. This highlights that the process manufacturing variation influences the frequency of the BRs' oscillations during the oscillation region.

7 FPGA Implementation

As recommended by Toker et al., one should only make a verdict about a system in question once the CDTA analyzes the trajectories from both the system's simulation and the system's implementation [21]. We used a 64-stage BR-PUF (from Figure 3) to implement multiple 64-stage BRs on a Xilinx XC7S25-1CSGA324 FPGA housed on a Digilent Arty S7-25 development board to complement our results from the Monte Carlo simulation. We built two controller modules to sift through the BR-PUF's 2^{64} BR realizations and collect data from different BR trajectories: the metastable BR

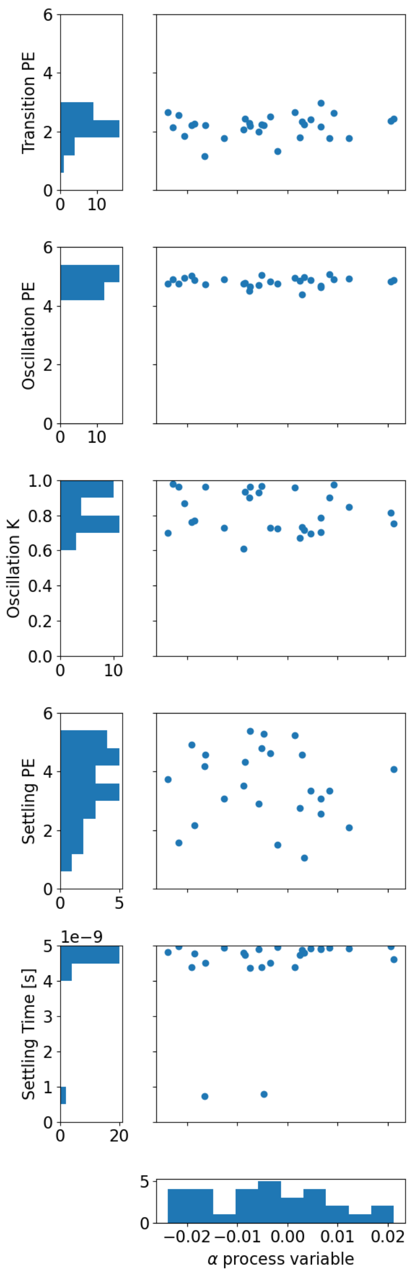


(a) k inverters' PMOS V_{TH0}

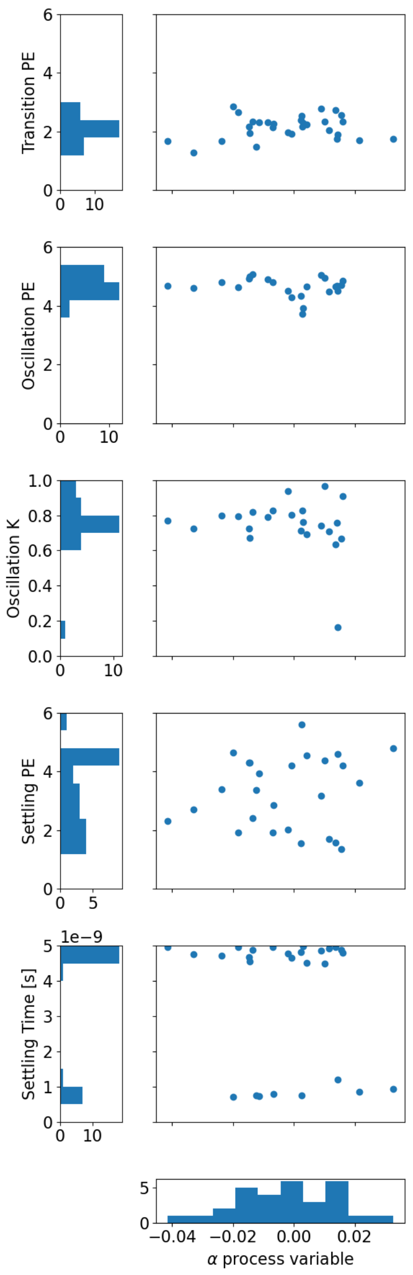


(b) k inverters' NMOS V_{TH0}

Figure 10: Monte Carlo simulations for varying a random selection of $k = \log_2(N) = 6$ inverters' properties by the percent change α.



(a) All inverters' PMOS V_{TH0}



(b) All inverters' NMOS V_{TH0}

Figure 11: Monte Carlo simulations for varying all $N = 64$ inverters' properties by the percent change α.

finder and the random BR runner. Each controller module connects to the BR and external I/O as shown in Figure 12. We analyzed the BR-PUF implementation's voltage trajectories with a PicoScope 3206D MSO oscilloscope and exported them as CSV files for analysis for the CDTA.

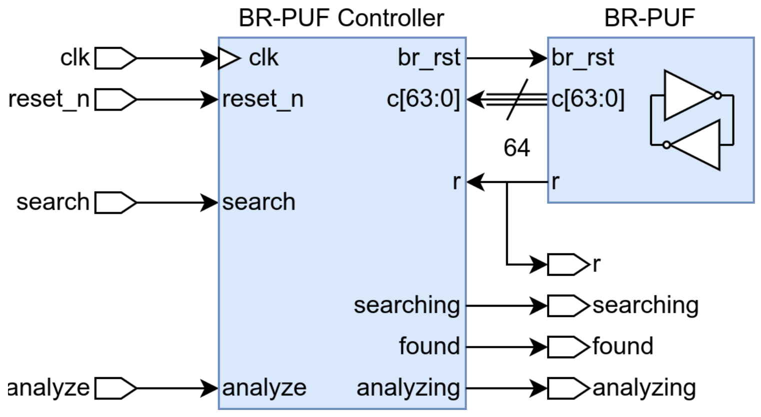


Figure 12: FPGA Block Diagram for connecting the BR-PUF controller to the BR-PUF

As shown in Figure 12, each controller module has a clocked input `clk` with an active-low synchronous reset `reset_n`. We used Xilinx Vivado's Clocking Wizard to convert the Arty S7's onboard 100 MHz clock to a 10 MHz clock. The user-controlled `search` button input prompts the controller module's internal FSM to start or resume the process of continuously sending challenges to the BR-PUF; the BR-PUF instantiates the corresponding BR and produces the response `r` in turn. The user-controlled `analyze` switch input prompts the controller module's internal FSM to start or stop sending the same challenge to the BR-PUF; we only used it in the metastable BR finder for analyzing sequential transients from the same metastable BR. We routed the outputs `searching`, `found`, and `analyzing` to the Arty S7's onboard LEDs to provide a visual aid for the controller module's status.

7.1 Metastable BR Finder

The metastable BR finder searched for, tested, and analyzed the BR configurations in the BR-PUF via brute force to find a bistable ring that showed instability for more than 0.1 ms. The metastable BR finder contains metastability check logic that captures the BR-PUF's `r` output and propagates it through a register of two clocked D flip-flops. The logic block stores the once-delayed and twice-delayed `r`-values as `r_d` and `r_dd`, respectively. The internal `metastable` wire is only equal to "1" if the FSM's time counter is greater than a threshold value and when there is a mismatch between sequentially captured r-values (i.e., r_d ≠ r_dd).

We implemented the metastable BR finder with three finite state machines running in parallel: the primary state machine selected the metastable BR finder's mode of operation, while two secondary counters kept track of elapsed time and the current testing challenge. The metastable BR finder's primary finite state machine (FSM) has the state diagram shown in Figure 13.

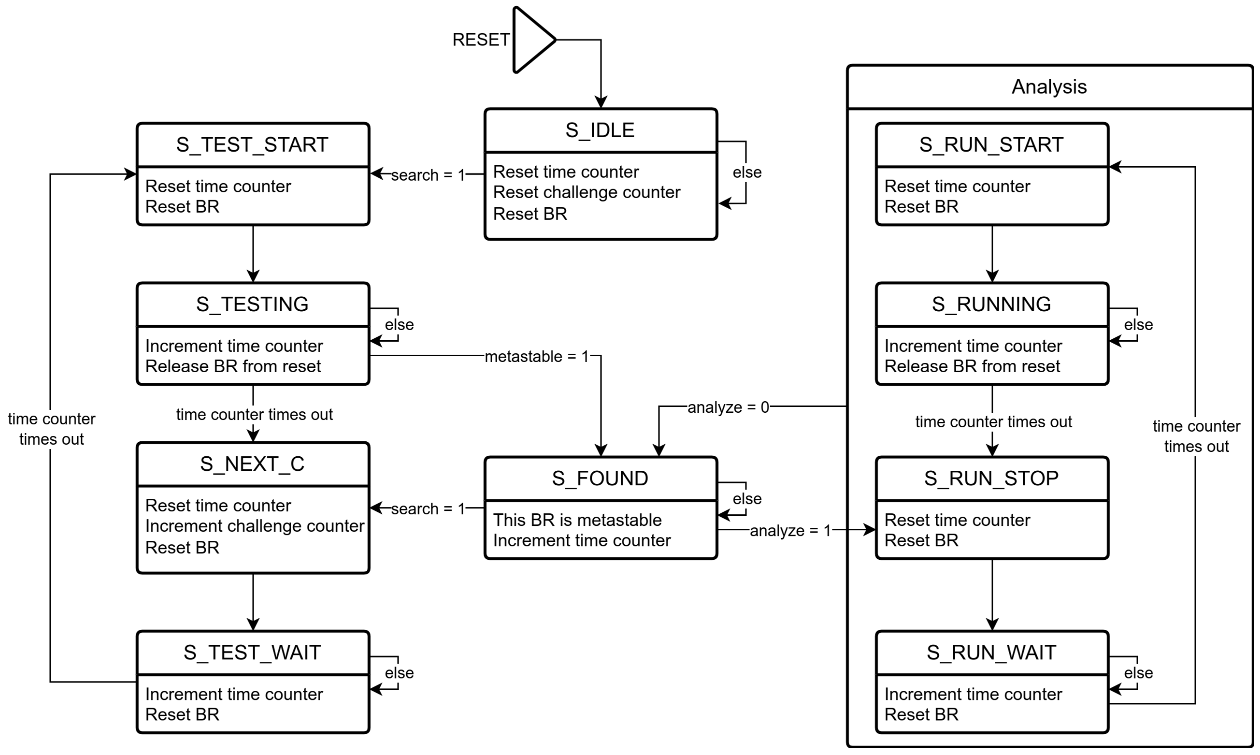


Figure 13: Metastable BR finder primary FSM state diagram

When the metastable BR finder turns on or resets, it starts in the idle state S_IDLE where it waits for the search input to be logical high. When this occurs, the metastable BR finder begins the brute-force search to find a metastable BR starting from the first challenge 64'd0.

The searching loop contains four states: S_TEST_START, S_TESTING, S_NEXT_C, and S_TEST_WAIT. The first of these states, S_TEST_START, resets the BR-PUF and the time counter to prepare for testing. It immediately changes into the S_TESTING state, where it releases the BR-PUF from the reset position and the time counter begins to count upward from zero. If the current challenge shows metastable behavior, then the metastable input will be high; in this case, the FSM exits the loop into the S_FOUND state. If the BR does not show metastable behavior, then the timer will eventually time out and the FSM will move to the S_NEXT_C state. In this state, the controller resets the BR-PUF, the time counter resets, and the challenge counter increments by one. The FSM immediately moves to the `S_TEST_WAIT` state, letting the BR-PUF stabilize while reset before returning to `S_TEST_START` state once the timer times out again.

The FSM only exits the searching loop when the current challenge in the BR-PUF corresponds to a metastable BR. The metastable BR finder is inactive while its FSM is in the `S_FOUND` state; if the `search` input is high, then it will return to the searching loop; if the `analyze` input is high, then it will move to the analysis loop.

The analysis loop acts like the searching loop, also containing four states: `S_RUN_START`, `S_RUNNING`, `S_RUN_STOP`, and `S_RUN_WAIT`. Each state acts like their corresponding searching loop state except for two key differences. Since the current challenge already corresponds to a metastable BR configuration, `S_RUNNING` is unresponsive to the `metastable` input and the challenge counter does not increment in `S_RUN_STOP`. The metastable BR finder enters this loop at the `S_RUN_STOP` state. This loop only runs while the `analyze` input is high; otherwise, the FSM will return to `S_FOUND`. Figure 15b shows sixty-four transients of the same metastable BR when the metastable BR finder is running in the analysis loop.

7.2 Random BR Runner

The random BR runner sent (pseudo-)randomly selected challenges to the BR-PUF for analysis using a 64-bit linear feedback shift register (LFSR) with the feedback polynomial $f(x) = x^{64} + x^{63} + x^{61} + x^{60} + 1$ starting from `64'd1`. Like the metastable BR finder, we implemented the random BR runner with three finite state machines running in parallel: the primary state machine selected the random BR finder's mode of operation, a counter kept track of elapsed time, and the LFSR kept track of the current testing challenge. The random BR runner's primary finite state machine (FSM) has the state diagram shown in Figure 14.

When the random BR runner turns on or resets, it starts in the idle state `S_IDLE` where it waits for the `search` input to be logical high. When this occurs, the random BR runner begins randomly sending challenges to the BR-PUF. Like the metastable BR finder's searching loop, the random BR runner's searching loop contains four states: `S_TEST_START`, `S_TESTING`, `S_NEXT_C`, and `S_TEST_WAIT`. These states act like the metastable BR finder's searching loop, except for two key differences: the `S_TESTING` state is unresponsive to the `metastable` input and `S_NEXT_C` state performs one LFSR

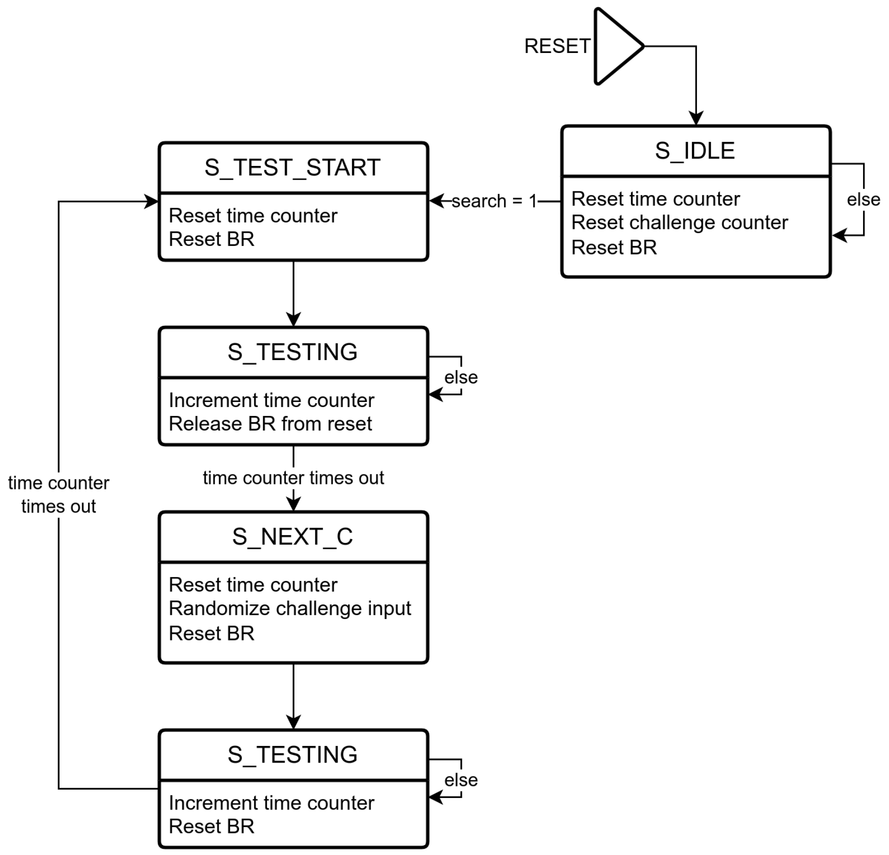


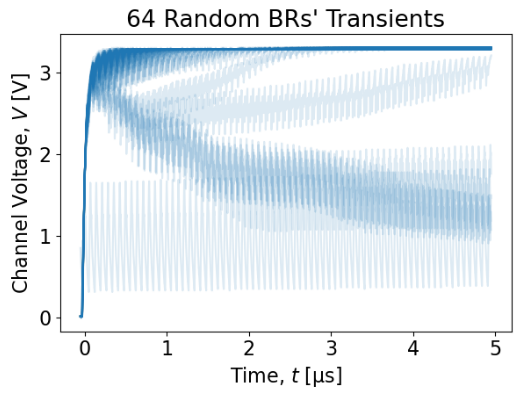
Figure 14: Random BR runner primary FSM state diagram

shift to generate the next challenge. Figure 15a shows sixty-four transients of sixty-four randomly selected BRs while the random BR runner is in the searching loop.

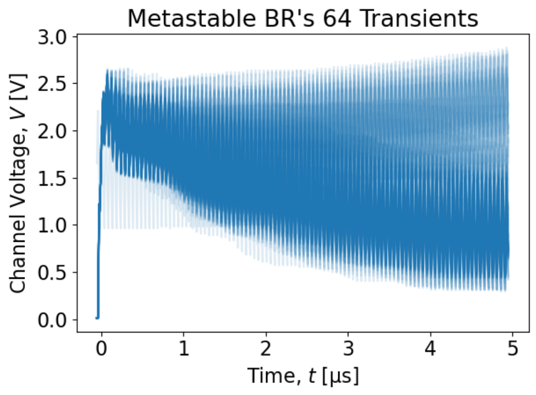
7.3 FPGA BR Analysis

We programmed both of our controller modules with Xilinx Vivado 2022.2 and implemented them on the same Digilent Arty S7-25 development board. We connected the oscilloscope to the FPGA's ground and digital output pins to record the voltage transients from the BR-PUF using the Pico-Scope 7 oscilloscope software. This software's ability to record up to sixty-four triggered waveforms back-to-back helped us obtain both the sixty-four transients of sixty-four randomly selected BRs in Figure 15a and sixty-four transients of one metastable BR in Figure 15b. While most of the randomly selected BRs settled quickly towards logic high within 5 µs, none of the metastable BR's transients showed any settling behavior.

As previously explained in Section 3.2, we already developed a function that could split the



(a) Transients from randomly selected BRs



(b) Transients from one metastable BR

Figure 15: FPGA BR implementation transient plots

simulated BRs' transients into transition, oscillation, and settling regions. We modified this function to split the FPGA-implemented BRs' transients by increasing the settling threshold percentage from 2% to 25% since the FPGA-implemented BRs peaked at a relatively lower voltage value than the simulated BRs.

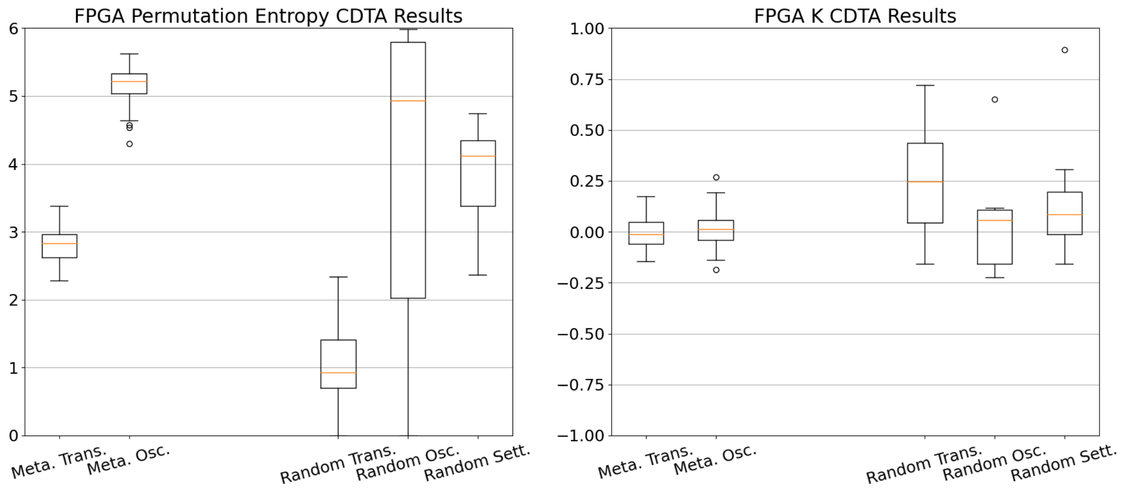


Figure 16: Permutation Entropy (left) and 0-1 Test K (right) box plots of FPGA BR transients

Figure 16 shows the metastable BR's results and random BRs' results from the CDTA. The metastable BR's signals had no settling region, so it is omitted from this figure. The metastable BR's transients were more complex per region than the random BRs' transients. However, the 0-1 test results show us that most BR regions were classified as periodic since the average results skew closer to 0 than 1. The outlier signals represented by the circles on the right box plot had 0-1 test results much closer to 1 than 0.

8 Optimization with Genetic Algorithms

We used the `deap` Python library [30] to run two different GA strategies, differentiated by whether they allowed the individual BRs' percent change values to be outside the range of $[-5\%, 5\%]$. Table 3 shows the key differences between the two strategies. Our unrestricted α strategy sampled values from the previously-used zero-mean Gaussian distribution with $6\sigma = 5\%$ as the initial percent change distribution. Individual genes could escape the $[-5\%, 5\%]$ range using the Blend Crossover and Gaussian Mutation functions. The restricted α strategy instead sampled values from a zero-mean truncated Gaussian distribution with $6\sigma = 5\%$ bounded to the range $[-5\%, 5\%]$ (from the Python library `scipy` [31]). We swapped out the crossover and mutation functions from the unrestricted α strategy with the Bounded Simulated Binary Crossover and Bounded Polynomial Mutation functions respectively to keep the genes within $[-5\%, 5\%]$. For extra redundancy, the restricted α cost function included an extra clause that would punish any individual BR with outlying percent changes in its genome.

Table 3: Genetic Algorithm (GA) procedure comparison between unrestricted α and restricted α strategies

Strategy	**Unrestricted α**	**Restricted α**
α Sample	$\alpha \sim \mathcal{N}(0, 1.\bar{6}\%)$	$\alpha \sim \text{TruncNorm}(0, 1.\bar{6}\%, -5\%, 5\%)$
Cost Function	Negative PE of oscillation region, if applicable; 1.0 otherwise.	Negative PE of oscillation region, if applicable; 1.0 otherwise. Punish BRs with outlier genes.
Crossover Op.	Blend Crossover	Bounded Simulated Binary Crossover
Mutation Op.	Gaussian Mutation	Bounded Polynomial Mutation

Both GA strategies started with an initial population size of 128. Each strategy had a maximum of one hundred generations to find the most complex BRs but would halt if it made no improvements within ten generations. Figure 17 shows our cost function statistics for both GA strategies across all generations. The restricted α strategy failed to make any improvements after forty-eight generations; the unrestricted α strategy continued for all one hundred generations. The large standard deviation spikes in the earlier generation are the result of the cost function penalizing individual BRs that either failed to oscillate or did not have a large enough oscillation region to calculate permutation entropy.

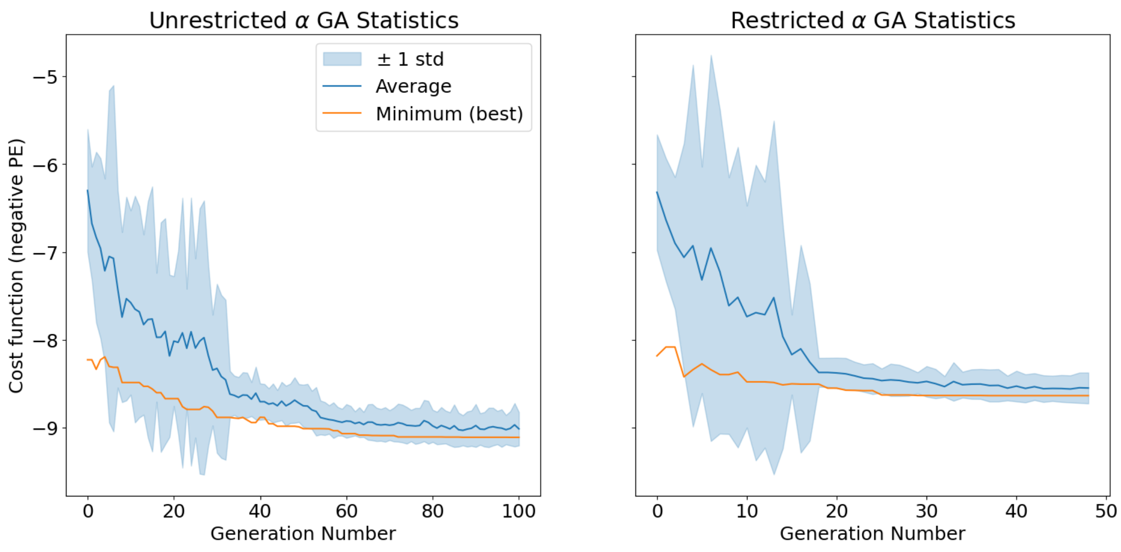


Figure 17: Standard deviation, average, and best results from both GA strategies

Figure 18 compares both GA strategies' initial process percentage α distributions to the spread of their last generation's genes. The spread of the unrestricted α strategy's gene distribution on the left shows how the unbounded mutation and crossover operations caused outliers to appear in the last generation's percent change distribution. These anomalies are not present in the restricted α strategy's gene distribution on the right. We observed in Figure 17 that the unrestricted α strategy produced a BR with a more complex oscillation region than the restricted α strategy; therefore, we believe that the outlier genes had a beneficial factor in making the unrestricted α strategy's population more fit than the restricted α strategy's population.

Figure 19 shows our analysis of the first inverter's transition, oscillation, and settling regions of each BR in both GA strategies' last generation. We omitted the transition regions from this figure because they were too short for analysis using the CDTA. The oscillation permutation entropy on the left graph of Figure 19 does not match the results shown in Figure 17 because the CDTA's output PE value uses order 5 after denoising and downsampling. However, this post-processing does not change that the unrestricted α strategy showed more complex oscillation signals than the restricted α strategy. As shown by the 0-1 test results on the right of Figure 19, optimizing permutation entropy resulted in higher 0-1 test results in both strategies. From the 0-1 test, we can declare that most of the oscillation regions are still chaotic. The CDTA could not analyze

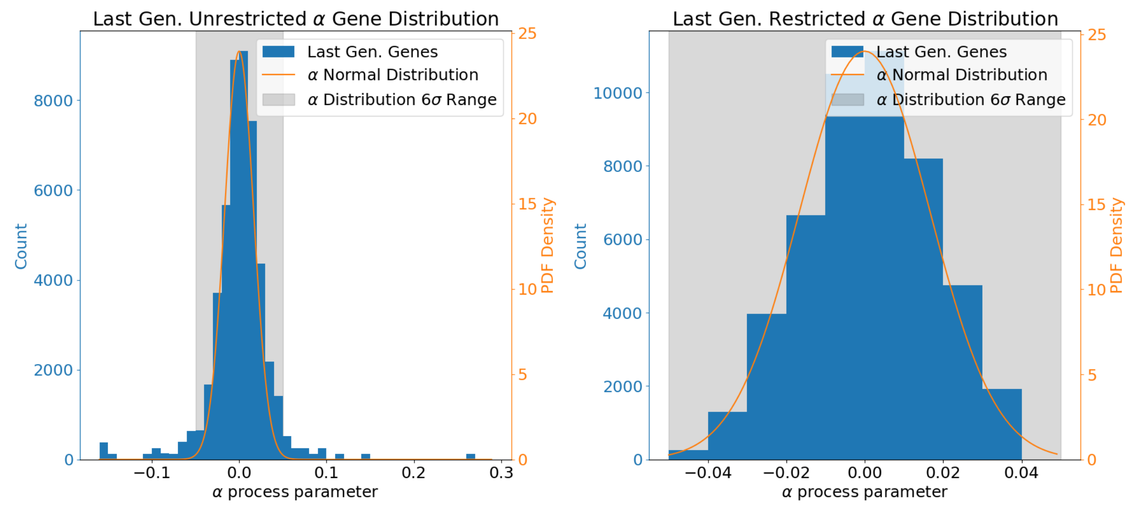


Figure 18: Last generation gene distribution from both GA strategies

the settling region of the restricted α population because there were not enough values in the time-series.

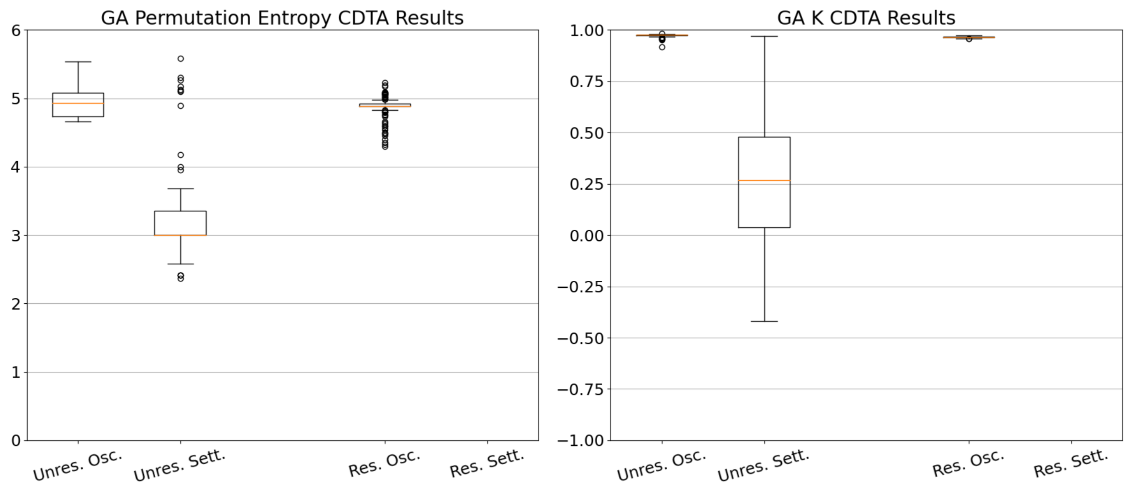


Figure 19: Last generation CDTA results from both GA strategies

9 Conclusion

In our Monte Carlo experiments, we saw that varying the NMOS and PMOS threshold voltage parameters provided only chaotic results in the best-case scenario; the transition and settling regions were periodic. Overall, the oscillation results did show the most chaotic dynamics, but we did not find any way to produce results that the CDTA would classify as stochastic with the AAFT+RP surrogate algorithm. While one settling region and one oscillation region were stochastic, these may be false classifications because of the introduced randomness from the AAFT+RP surrogate algorithm. From a big-picture perspective, we saw that only modifying a portion of the BR's inverters (from Figure 10) mixed inverters with matched transistors and inverters with unmatched transistors in the same BR and produced the highest amount oscillation regions in our sample. Modifying all inverters by the same value (from Figure 11) caused each inverter's transistor pair in the BR to become unmatched, which may be why some BRs settled in our sample. Modifying just one inverter (from Figure 9) completely threw off the BRs' metastability and caused all BRs in the sample to settle quickly. Overall, our simulation results show that the bistable ring alone is unsuitable for generating random numbers since its behavior is not primarily stochastic.

We saw an evident mismatch between the FPGA-implemented BR's signal classification in Figure 16 and the simulated GA BR classifications in Figures 9, 10, and 11. The FPGA-implemented BRs were quantitatively less chaotic than the simulated BRs according to the CDTA's modified 0-1 test. In addition, the transients from simulated BRs like Figure 5 do not perfectly resemble the FPGA-implemented BRs shown in Figure 15. One probable reason this may occur is because the simulated BRs had a higher oscillation frequency than the FPGA-implemented BRs; we did not check whether the results that we observed on the PicoScope had undergone aliasing. Another probable reason is that the analog signals output by the BR-PUF were sent through digital logic paths and recorded from a digital logic output. This could have unintentionally processed our output results by limiting the slew rate. These two reasons may be why the modified 0-1 test results consistently showed that the simulated BRs' oscillation regions are quantitatively more chaotic than the FPGA BRs' oscillation regions.

Our genetic algorithm strategies were successful in optimizing the simulated BR's parameters to find BRs with complex oscillation region. Our observations in Figure 17 and Figure 19 showed us that restricting the percent change range to $[-5\%, 5\%]$ produced BRs with less complex oscillation results than not restricting the percent change range. This may imply that ASIC manufacturers can be less strict with process variations or errors when constructing ICs that employ the BR for cryptographic hardware applications. However, a single unbalanced process parameter should not be too overpowering or else the BR will not oscillate like those in the first two Monte Carlo experiments in Figure 9. One issue with genetic algorithms is that they have no concept of sacrificing short-term rewards for long-term gain; this results in GA strategies becoming stuck in local minima without finding their cost function's true maximum. The mutation operator does help prevent this in some sense by introducing new values into the system, but the GA would discard an individual that is at the top of a lower well in the cost function's landscape if it is not fit enough. Since we cannot visualize the 384-dimensional problem outlined in our cost function, we cannot confirm that our results are the most complex BRs in these ranges.

Toker et al. have recommended that each time-series input into the CDTA has at least five thousand samples, with the minimum tested sample length of one thousand. They also observed that the CDTA's result could incorrectly classify systems if the input time-series is not long enough (e.g., weakly chaotic systems misclassified as periodic) [21]. The BRs' metastable nature and limitations of Ngspice made it difficult to provide at least one thousand samples for each region in each simulation; this may be why the CDTA classified some of the oscillation behaviors in the BR as periodic. The CDTA also failed to run in some cases where the input time-series was incompatible with the AAFT surrogate algorithm; this may be another result from not having enough input data.

It should not be surprising that the simulated BR does not show stochastic behavior; Ngspice simulations themselves are deterministic unless otherwise specified in the netlist configuration [11]. Our simulation results come from a hyper-ideal scenario where all noise (e.g., Johnson-Nyquist noise, temperature fluctuations) is nonexistent and all components behave exactly as described in their mathematical model. In other words, we are observing the BR as mapping parameters and initial conditions into transients and settled results. The FPGA implementation of BRs provides a more realistic scenario where removing noise from the system is impossible and mathematical models may not necessarily reflect what occurs in hardware. The results from Figure 15b ideally should behave similar since we released the same BR from the same unstable state; however, the sixty-four different transients show us that the BR still produces many different signals even though each waveform was supposed to be produced from identical initial conditions and parameters. While we can *model* the BR as a chaotic system, this does not necessarily translate well into *predicting* how the BR's trajectory will evolve over time. The transient results from Figure 15b show us that there is only a small window of time in the transition region where all BR transients line up before diverging. In this scenario, we are observing how the BR propagates both process and thermal noise through its output signals.

The results of this thesis show several directions for future projects and experiments in the fields of nonlinear dynamics, genetic programming, and hardware security. One future project would be formalizing the CDTA as a fully-fledged library in a popular programming language repository. While the CDTA was integral for this thesis, it still needs some improvements: it is unoptimized for speed, contains typos in both code and documentation, and produces errors when analyzing systems with settling behavior using the default configuration. More rigorous testing on the CDTA and distribution via a programming language package service may help with finding more ways to improve the algorithm's efficacy. A second continuation would continue with the GA optimization by running one directly on the FPGA to produce a BR with the most complex results. For example, one could include a soft-core processor that evaluates the permutation entropy of different individual BR-PUF configuration bitstreams and subsequently runs the remaining steps of the GA procedure. Individuals in this GA would have bitstreams of size N, corresponding to a single BR-PUF challenge. Since GA procedure strategies already exist for optimization with binary genomes [29], [30], the obtained results could complement our simulated GA results and further help understand the BR. A third project could address the mismatch between FPGA and simulation results by analyzing the transients from BRs implemented without an FPGA. Implementing a BR-PUF on 4000-series logic ICs would be cheaper than creating a custom BR-PUF ASIC but may cause routing issues and introduce parasitic capacitance.

References

[1] G. Taylor and G. Cox, “Digital randomness,” *IEEE Spectrum*, vol. 48, pp. 32–58, 9 Sep. 2011, ISSN: 0018-9235. DOI: `10.1109/MSPEC.2011.5995897`. [Online]. Available: `http://ieeexplore.ieee.org/document/5995897/`.

[2] M. Stipčević and Ç. K. Koç, “True random number generators,” in Ç. K. Koç, Ed. Springer International Publishing, 2014, pp. 275–315. DOI: `10.1007/978-3-319-10683-0_12`. [Online]. Available: `https://doi.org/10.1007/978-3-319-10683-0_12`.

[3] Q. Chen, G. Csaba, P. Lugli, U. Schlichtmann, and U. Rührmair, “The bistable ring puf: A new architecture for strong physical unclonable functions,” IEEE, Jun. 2011, pp. 134–141, ISBN: 978-1-4577-1059-9. DOI: `10.1109/HST.2011.5955011`. [Online]. Available: `http://ieeexplore.ieee.org/document/5955011/`.

[4] Q. Chen, G. Csaba, P. Lugli, U. Schlichtmann, and U. Rührmair, “Characterization of the bistable ring puf,” IEEE, Mar. 2012, pp. 1459–1462, ISBN: 978-1-4577-2145-8. DOI: `10.1109/DATE.2012.6176596`. [Online]. Available: `http://ieeexplore.ieee.org/document/6176596/`.

[5] J. Liebow-Feeser. “Randomness 101: Lavarand in production.” (Nov. 2017), [Online]. Available: `https://blog.cloudflare.com/randomness-101-lavarand-in-production/`.

[6] P. Bayon, L. Bossuet, A. Aubert, *et al.*, “Contactless electromagnetic active attack on ring oscillator based true random number generator,” vol. 7275 LNCS, Springer Berlin Heidelberg, 2012, pp. 151–166, ISBN: 978-3-642-29912-4. DOI: `10.1007/978-3-642-29912-4_12`. [Online]. Available: `http://link.springer.com/10.1007/978-3-642-29912-4_12`.

[7] J. P. Uyemura, *CMOS Logic Circuit Design*, 1st ed. Springer New York, NY, 2001, ISBN: 978-0-306-47529-0. DOI: `10.1007/b117409`. [Online]. Available: `https://link.springer.com/book/10.1007/b117409`.

[8] Anaconda, *Anaconda software distribution*, version 2022.10, Oct. 2022. [Online]. Available: `https://anaconda.com`.

[9] conda-forge community, *The conda-forge project: Community-based software distribution built on the conda package format and ecosystem*, Jul. 2015. DOI: `10.5281/zenodo.4774216`. [Online]. Available: `https://doi.org/10.5281/zenodo.4774216`.

[10] Y. Cao, W. Zhao, E. Wang, *et al.* “Predictive technology model.” (2007), [Online]. Available: `https://ptm.asu.edu/`.

[11] D. Warning, H. Vogt, F. Lannutti, and P. Nenzi, *Ngspice*, version 37, May 2022. [Online]. Available: `https://ngspice.sourceforge.io`.

[12] F. Salvaire, *Pyspice*, version 1.5, May 2021. [Online]. Available: `https://pyspice.fabrice-salvaire.fr/`.

[13] P. Jain and A. M. Joshi, “Analyzing the impact of augmented transistor nmos configuration on parameters of 4x1 multiplexer,” *Radioelectronics and Communications Systems*, vol. 61, pp. 121–127, 3 Mar. 2018, ISSN: 07352727. DOI: `10.3103/S0735272718030044`. [Online]. Available: `https://link.springer.com/article/10.3103/S0735272718030044`.

[14] The pandas development team, *Pandas-dev/pandas: Pandas*, version 1.4.4, Feb. 2020. DOI: `10.5281/zenodo.3509134`. [Online]. Available: `https://doi.org/10.5281/zenodo.3509134`.

[15] W. McKinney, "Data structures for statistical computing in python," S. van der Walt and J. Millman, Eds., 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[16] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, "Array programming with numpy," *Nature*, vol. 585, pp. 357–362, 7825 Sep. 2020, ISSN: 0028-0836. DOI: 10.1038/s41586-020-2649-2.

[17] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, pp. 90–95, 3 2007, ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.55.

[18] The MathWorks Inc., *Control system toolbox*, version 2022a, 2022. [Online]. Available: https://www.mathworks.com/help/control/index.html?s_tid=CRUX_lftnav.

[19] I. Vasyltsov, E. Hambardzumyan, Y.-S. Kim, and B. Karpinskyy, "Fast digital trng based on metastable ring oscillator," E. Oswald and P. Rohatgi, Eds., Springer Berlin Heidelberg, 2008, pp. 164–180, ISBN: 978-3-540-85053-3. DOI: 10.1007/978-3-540-85053-3_11. [Online]. Available: http://link.springer.com/10.1007/978-3-540-85053-3_11.

[20] D. Toker, *The chaos decision tree algorithm*, Oct. 2019. DOI: 10.6084/m9.figshare.7476362.v7. [Online]. Available: https://figshare.com/articles/software/The_Chaos_Decision_Tree_Algorithm/7476362.

[21] D. Toker, F. T. Sommer, and M. D'Esposito, "A simple method for detecting chaos in nature," *Communications Biology*, vol. 3, p. 11, 1 2020, ISSN: 2399-3642. DOI: 10.1038/s42003-019-0715-9. [Online]. Available: https://doi.org/10.1038/s42003-019-0715-9.

[22] C. Bandt and B. Pompe, "Permutation entropy: A natural complexity measure for time series," *Physical Review Letters*, vol. 88, p. 174102, 17 Apr. 2002, ISSN: 0031-9007. DOI: 10.1103/PhysRevLett.88.174102. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.88.174102.

[23] G. Datseris and U. Parlitz, *Nonlinear Dynamics*, 1st ed. Springer Cham, Mar. 2022, ISBN: 978-3-030-91031-0. DOI: 10.1007/978-3-030-91032-7. [Online]. Available: https://link.springer.com/10.1007/978-3-030-91032-7.

[24] L. Zunino and C. W. Kulp, "Detecting nonlinearity in short and noisy time series using the permutation entropy," *Physics Letters A*, vol. 381, pp. 3627–3635, 42 Nov. 2017, ISSN: 03759601. DOI: 10.1016/j.physleta.2017.09.032. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0375960117308976.

[25] M. Riedl, A. Müller, and N. Wessel, "Practical considerations of permutation entropy," *The European Physical Journal Special Topics*, vol. 222, pp. 249–262, Jun. 2013, ISSN: 1951-6401. DOI: 10.1140/EPJST/E2013-01862-7. [Online]. Available: https://link.springer.com/article/10.1140/epjst/e2013-01862-7.

[26] A. A. B. Pessa and H. V. Ribeiro, "Ordpy: A python package for data analysis with permutation entropy and ordinal network methods," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 31, p. 063110, Feb. 2021. DOI: 10.1063/5.0049901.

[27] G. Lancaster, D. Iatsenko, A. Pidde, V. Ticcinelli, and A. Stefanovska, "Surrogate data for hypothesis testing of physical systems," *Physics Reports*, vol. 748, pp. 1–60, Jul. 2018, ISSN: 03701573. DOI: 10.1016/j.physrep.2018.06.001. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0370157318301340.

[28] G. A. Gottwald and I. Melbourne, "The 0-1 test for chaos: A review," in C. (Skokos, G. A. Gottwald, and J. Laskar, Eds. Springer Berlin Heidelberg, Mar. 2016, vol. 915, pp. 221–247, ISBN: 978-3-662-48410-4. DOI: 10.1007/978-3-662-48410-4_7.

[29] E. Wirsansky, *Hands-On Genetic Algorithms with Python.* Packt Publishing, Jan. 2020, ISBN: 9781838557744.

[30] F. A. Fortin, F.-M. D. Rainville, M. A. Gardner, M. Parizeau, and C. Gagné, "Deap: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, Jul. 2012.

[31] P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, "Scipy 1.0: Fundamental algorithms for scientific computing in python," *Nature Methods*, vol. 17, pp. 261–272, 2020. DOI: `10.1038/s41592-019-0686-2`.