A Randomness Test Suite for Physical Unclonable Functions

Amit Deo, Charles Grover

Crypto Quantique Ltd

***Abstract*—The generation of high entropy random strings is crucial to modern cryptography. Without good quality randomness, one cannot be confident that any cryptosystem can adequately secure data. Abstractly, the problem of random number generation is well understood: most computers can provide sufficiently high quality randomness from a variety of well-known Random Number Generators (RNGs). Additionally, established test suites, such as those provided by NIST, allow practitioners to quantify the quality of random sources. However, in the context of the IoT, random number generation is less comprehensively understood, as most hardware RNGs are too large to fit within the footprint of IoT devices. A modern primitive known as the Physical Unclonable Function (PUF) alleviates this, providing devices with fixed quantities of random bits, whilst consuming only a small area. It is common to test PUF designs by applying statistical tests; however, subtle distinctions between PUFs and RNGs mean that most randomness test suites cannot be directly applied to PUFs. In this work, we solve this problem by designing a new randomness test suite specifically for PUFs. We construct our suite by making appropriate modifications to pre-existing suites, defining a set of tests that are comprehensive but well-understood.**

***Index Terms*—Randomness, Statistical Testing, Physical Unclonable Functions**

I. Introduction

Access to a source of random strings is a pre-requisite for almost all modern cryptography. In the design of secure systems, it is thus important to have access to a high quality source of random numbers. Typically, this is done using local access to a random number generator (RNG), which provides seeds or other random strings to support the necessary cryptographic functionality. Any well-designed RNG should be based on a mathematical or physical model which explains the source of random bits, and demonstrates that those bits are genuinely random. However, since by necessity RNGs tend to derive their entropy from physical phenomenon that can be challenging to model accurately, it is important that this model is not the end of the story. Instead, good practice dictates that the RNG should be thoroughly tested, typically in the form of running its outputs through a sequence of statistical tests. Without this testing, there may be some unforeseen error in the design which causes the outputs to have some undesirable patterns; seeds generated according to any unexpected pattern can have disastrous consequences for cryptography built on top of them.

Testing the output of RNGs is not an easy task. Although there are plenty of obvious properties to test for, such as the balance of 0s and 1s in a binary string, there may be more intricate patterns present in the output which are harder to detect, but still equally problematic. To handle this, it is common practice to apply a pre-existing suite of randomness tests designed to catch some typical modes of failure for RNGs. Popular test suites include the NIST 800-22 suite [1], the AIS31 tests [2], and the Diehard and Dieharder tests [3], [4]. Generally, RNG test suites contain a collection of statistical tests to be run on output from the RNG. Since these are statistical tests, best practice dictates that several blocks of input data from the RNG are input through the tests. Each block either passes or fails the test, and if an appropriate proportion of the blocks pass the test then the tests support the hypothesis that the RNG is outputting random data.

Existing test suites such as the NIST 800-22 suite are often considered to be appropriately thorough to test the performance of RNGs. However, it is not always possible for a cryptographic system to have access to a classical RNG. For example, in the world of embedded devices, it is often the case that pre-existing RNGs take up too much room, or are just too expensive to build into a lightweight device. A modern primitive for generating random seeds, particularly for cryptographic use on small or low powered devices, is the Physical Unclonable Function (PUF) [5], [6]. A PUF provides a fixed string of random bits, typically derived from process variations on the manufacturing line in microelectronic parts. In that sense, they can be viewed as an analogue of biometrics such as fingerprints, but for silicon parts rather than human users.

As with classical RNGs, it is crucial to both provide a thorough, process-based model for the randomness of a PUF and to supplement the model with tests on prototypes of the PUF design. However, pre-existing tests designed for classical RNGs are not naturally applicable to all PUF designs. Since a PUF generates its bits from a physical array, it is only capable of generating a fixed length output, which often rules out tests requiring large output strings. Furthermore, unlike a classical RNG, one cannot request more output bits from a PUF, so the randomness tests must be run on a per-PUF basis, as opposed to on many blocks of output from the same RNG. One could try to circumvent this issue by concatenating the output of multiple instances of the PUF and running randomness tests on these longer strings, but it is unlikely that a new PUF design will have sufficiently large data sets for this to be practical. Furthermore, it is important to catch defects on a per-PUF basis, as well as across the data set of all available PUFs. That is, another complication

Correspondence: Patrick Camilleri, Crypto Quantique Ltd
Email: pcamilleri@cryptoquantique.com

arising from the manufacture of PUFs is one of correlation; it is not sufficient to check that each PUF outputs a random string of bits. Rather, each PUF output must be random even when considered alongside the output of other PUFs of the same design. Otherwise, PUFs do not stand up well to their standard deployment scenario, where an embedded device has a built-in PUF which it uses to generate its randomness for required cryptographic functions such as device authentication [7]. Additionally, the physical nature of a PUF array can both educate and complicate the kind of testing required to validate the randomness; the locations in the output of a PUF bit string typically have more spatial interpretations than those of a generic RNG. Nonetheless, when experimentally validating new PUF designs, it is currently common practice to just apply a collection of tests from pre-existing suites, selecting tests according to those which the PUF has enough output bits to qualify for.

In this work, we show that PUF testing can be improved upon by understanding the key differences between a PUF and an RNG. The most notable differences are spatial dependence arising from the geometry of PUF designs and restrictions on PUF output lengths. Once the differences are properly understood, existing randomness tests from the NIST 800-22 suite can be re-purposed with little difficulty to test for PUF randomness. In order to define our suite, we carefully modify and reparametrize some tests that were originally intended for very long RNG bit-strings. The most important modifications that we describe are for the Binary Matrix Rank and Template Matching Tests. When modifying the latter, we significantly reduce the number of required bits (from a million to a few thousand) by replacing an m-dependent Central Limit Theorem with an efficient method for computing an *exact* distribution. We believe that it is valuable to describe a formal suite of tests to apply to a PUF. Our proposed suite significantly reduces that chance that designers accidentally or maliciously misrepresent the quality of their PUF by reducing room for interpretation left when applying RNG tests to PUFs. For example, [8] observes that the order in which PUF data is put into tests may dramatically effect the test results, a mistake which is easy to make and may be hard to detect. PUF-specific issues like these are automatically handled by our purpose built test suite. Additionally, we contribute a novel test for independence of PUF outputs, to help detect for PUF designs which have correlated outputs and thus are not suitable for providing cryptographic seeds.

Paper Outline: In Section II we provide the necessary background material to understand randomness testing and PUFs. In Section III, we present our recommendations for per-device randomness testing, including novel variations of two existing tests. In Section IV, we introduce a new test for independence of PUF outputs and explain its application. In Section V, we give a step-by-step summary of our recommended test suite. Finally, Section VI concludes the paper.

II. Background

In this section we will introduce some well-known pre-existing randomness tests that we use as the basis of our new test suite. These tests are designed for classical RNGs and we do not consider them universally suited to the case of testing PUFs. Thus, we will also explain some key differences between PUFs and RNGs that require a modification in testing methodology.

A. PUFs

A PUF, or *Physical Unclonable Function*, is a hardware object capable of returning a collection of random bits. The output of a PUF should be unpredictable and PUF instances should be physically unclonable and tamper-proof, so that an individual PUF is suitable for providing seeds for cryptographic use. There are two main output properties of interest for an ideal PUF: randomness, so that the distribution of outputs of a collection of PUF instances follows the uniform distribution, and reliability, meaning each PUF instance returns the same, or nearly the same, string each time it is called. We note that when we refer to a PUF in this way we are, strictly speaking, talking about weak PUFs, or "no-challenge" PUFs, where each device is only capable of outputting a single string. On the other hand, a strong PUF takes as input a binary challenge string and outputs a fixed, random, response. However most known strong PUF designs are vulnerable to machine learning attacks [9], [10] that model their output behaviour, and thus are not yet fit for purpose. Hence, we consider only weak PUFs in this work and for brevity refer to them merely as PUFs.

In practice, providing ideal behaviour is hard; PUFs are typically produced in integrated circuit form, and their behaviour is subject to small fluctuations due to the presence of electronic noise and similar real-world issues. To handle this, PUFs are typically supported by a small amount of post-processing, such as in the form of a fuzzy extractor [11], to smooth the output distribution and fix small inconsistencies in the output. Recalling the analogy between PUFs and biometrics, we see that they have similar post-processing requirements.

1) Comparing PUFs and RNGs: In this work, we are primarily interested in how the physical nature of a PUF distinguishes them from RNGs, which are typically capable of outputting a continuous string of random bits by extracting entropy from effects such as electronic noise. There are three main complications arising from the physical aspect of a PUF design when applying randomness tests designed for classical RNGs:

1) Since the output bits of the PUF are derived from fixed physical properties, each PUF contains only a finite number of output bits. Additionally, the number of output bits may be too small for complex randomness tests.
2) PUF designs often derive the bits from individual cells which are laid out in a matrix format, rather than deriving the bits in one long continuous string.
3) The output of each PUF should be independent of all other PUFs in its class. That is, not only should each PUF output a random string, but also it should be random even given access to arbitrarily many PUFs built according to the same design.

The first two of these properties primarily require taking care when applying tests designed for classical RNGs to the case of PUFs: only selecting those tests whose input requirements are satisfied by the PUF output and carefully selecting the order in which one inputs the PUF bits. However, the third is a novel aspect of PUFs not considered in previous tests. Of course, one could consider concatenating the outputs of many PUF instances and applying randomness tests to concatenated strings, but that assumes that the tests are sensitive to periodic behaviour of depth corresponding to the PUF output length, which may not be true in practice. Additionally, the number of available PUF chips may not be large enough to gather a sufficiently sized data set for meaningful statistical testing on their concatenated outputs (see for example Table 4 of [12], where many PUFs are introduced with less than a million total bits being tested).

2) Separating Randomness and Reliability: Another issue that arises when testing the randomness of PUFs is how to handle the unreliability of PUF bits. Although more complicated models are available [13], for intuition one can imagine each PUF bit having a preference towards outputting a 0 or a 1. Then, on each read, the PUF cell outputs its preferred bit except with some small error probability, in which case the bit flips e.g. outputs a 0 instead of the usual 1 (or vice versa). Depending on the use case of the PUF, the reliability of the output may or may not need to be considered, as we will explain in more detail in Section V. However, for simplicity we intend to decouple randomness from reliability. Thus, our randomness tests will take as input a single value for each cell, and it is important that how imperfect reliability is handled is chosen before the randomness tests are applied, so that a single value is determined for each cell in a manner which correctly reflects the PUF use case.

3) Mathematical Model: For clarity's sake, we describe a mathematical model of PUF behaviour that we are testing for. A PUF design is parameterized by a length n, and optionally a row and column size n_1, n_2 such that $n = n_1 \times n_2$. The PUF design defines a distribution $\mathrm{PUF}(n)$ over $\{0,1\}^n$. A PUF output ε is a sample from this distribution $\varepsilon \leftarrow \mathrm{PUF}(n)$. Similarly, a collection of PUFs $\varepsilon_1, \ldots, \varepsilon_k$ is a set of k (potentially not independent) samples from $\mathrm{PUF}_k(n)$. Typically, each bit of PUF output is derived from a separate part of the physical object. Thus, we will sometimes refer to the i$^{\text{th}}$ bit of the PUF output as the i$^{\text{th}}$ cell, denoting it using superscripts as ε^i.

For the purpose of this model, we assume that the PUF output is fully reliable; how the value of each cell is determined is not included. For an ideal PUF, $\mathrm{PUF}(n)$ is precisely the uniform distribution over $\{0,1\}^n$, and our statistical tests will start with the hypothesis that the PUF design being tested is uniform, rejecting this hypothesis if the data lies outside the expected range for uniformly random data. Similarly, the distribution induced by sampling k ideal PUFs of length n, $\mathrm{PUF}_k(n)$, should be equivalent to taking k independent samples from $\mathrm{PUF}(n)$, e.g. $\mathrm{PUF}_k(n) = \mathrm{PUF}(n)^k$, k independent samples from the uniform distribution.

B. Randomness Testing

In the context of this paper, the goal of randomness testing is to determine whether an object is outputting samples according to the expected distribution, which will always be uniform over binary strings of a given dimension. Perhaps the most famous and frequently used randomness test suite is the NIST 800-22 suite [1], which defines a collection of 15 tests for random data. These tests are hypothesis tests, starting with the null hypothesis that the data under test is random, measuring some feature of the test data, and rejecting the null hypothesis if and only if the resulting measurement, a p-value, lies outside an acceptable range. Since these are statistical tests, and thus even samples from a true RNG will fail the tests with some small probability, proper use of the suite requires many samples from the object under test. If around the appropriate number of the samples fail the test, then the evidence supports the hypothesis that the data is truly random. For example, if the p-value is set to 0.001, then we would expect the failure rate of truly random data to be around 1 per 1000 samples. More thoroughly, one can test that the distribution of the resulting p-values is appropriate using standard techniques such as the χ^2 test for uniformity.

1) Summary of 800-22 Tests: We remark that the aim of [1] is not to provide a specific, parameterized, test suite, upon passing which an RNG is deemed to be truly random. Instead, the tests are provided to assist designers of RNGs with a method by which to gather evidence supporting their design. Consequentially, the tests are somewhat open to interpretation by the tester; we will try to minimize this open nature later. However, NIST do provide an open source implementation of the test suite, containing choices of parameters determined by NIST. The NIST 800-22 test suite for statistical randomness contains 15 tests, summarised below; they are described in full detail in the report. However, it is still helpful to present a summary here, as some of the tests will need modification or re-interpretation for the case of PUFs.

1) *Monobit and Frequency within a block tests:* check that the global and local balance of 0's to 1's does not deviate significantly from equal.
2) *Runs and Longest-run-of-ones in a block tests:* check the number of runs of unchanging bits in the data-under-test, globally and in smaller sections, to ensure the data does not oscillate significantly faster or slower than expected.
3) *Binary Matrix Rank test:* considers the input data as a series of matrices and checks there is no significant linear dependence in the input data.
4) *Discrete Fourier Transform test:* looks for periodic features in the tested sequence that might indicate non-randomness.
5) *Non-overlapping template matching test:* searches for particular, aperiodic, patterns in the data.
6) *Serial test:* also checks for patterns of a fixed length to ensure none are appearing substantially more or less than would be expected.
7) *Approximate Entropy test:* calculates occurance frequencies for m and $m + 1$ length patterns in the sequence and checks that the difference is appropriate.
8) *Cumulative Sums test:* searches for overall trends towards ones or zeros in the data by converting all 0's to -1's and taking cumulative sums of the result, searching for any that deviate significantly from 0.

9) *Overlapping Template Matching:* searches for predefined target substrings.
10) *Maurer's Universal Statistic:* detects whether the sequence can be significantly compressed without loss of information.
11) *Linear Complexity:* computes the linear complexity of subsequences and compares the resulting distribution to that expected of a random sequence.
12) *Random Excursion tests:* the partial sums of the (0,1) sequence are adjusted to (-1, +1) to give a random walk; two tests then check whether the number of visits to a state within a random walk exceeds what one would expect.

2) Sample and Block Sizes: The NIST 800-22 Suite assumes that the tested RNG is able to produce essentially arbitrarily many bits as output. Although some tests, particularly the simpler ones such as the Monobit test, do not require a large input, other tests that check for more complicated behaviour, for example Maurer's Universal Statistic, require samples of length at least a million bits. The restrictions on input length are typically due to arguments about the rate of convergence of truly random data to some target distribution that is used to derive test statistics. Additionally, some tests have multiple parameters, as they partition the input string into blocks, and the convergence of the data to a determined distribution may depend on either the size or amount of blocks. For example, the Linear Complexity test requires at least 200 blocks of length between 500 and 5000.

3) AIS Autocorrelation Test: Another suite of popular tests are those found in [2]. This suite has significant overlap with the NIST 800-22 suite, but there is one specific test typically of interest to PUFs that we introduce here. The Autocorrelation test measures the correlation between bits a distance d from each other by examining the sum of the XORs of these bits, that is

$$\mathrm{ACF}_d(\varepsilon) = \sum_{i=1}^{n-d} \varepsilon^i \oplus \varepsilon^{i+d}.$$

Assuming the input distribution is uniform, each term of the summand is an independent sample from the Bernoulli distribution with parameter p, such that $\mathrm{ACF}_d(\varepsilon)$ approximates a normal distribution for large n. From this, a p-value is determined and each sample is deemed a pass or a fail. It is relatively common practice to remove the parameter d and interpret the ACF test as a single test for parameter $d = 1$, testing the correlation of adjacent bits. However, for the case of a PUF with a rectangular array of size $n_1 \times n_2$, there are two obvious parameters of interest: $d = 1$ and $d = n_1$, corresponding to row and column adjacent bits. Regardless, we view this test as having too much overlap with the Runs test from the NIST 800-22 suite. Thus, despite being commonly applied to PUF testing, we believe this test would increase the size of the suite at no real gain in functionality.

4) Critiques of NIST 800-22 Suite: Recently, NIST announced their decision to review the contents of the document defining the 800-22 test suite [14]. As part of this process, there was a request for public comments. Since our suite reuses ideas from the 800-22 test suite, we feel it prudent to acknowledge the more substantial comments here. The main contributions of the public review can be understood as follows:

1) Repeated claims that testing of p-values derived from binomial distributions for uniformity is inappropriate, and should instead be replaced by the testing of so-called Q-values (see e.g. [15]).
2) Concern that a suite of statistical tests cannot determine the suitability of an object to generate random numbers for cryptographic applications. Rather, the focus should be on stochastic models explaining the entropy content of the source as well as health and reliability testing.
3) The false positive rate of the tests are too high for uniformly random data.
4) Proposal of an additional test for "Specious" randomness, which can be somewhat understood as the presence of artificially balanced randomness, in the sense that the proportion of certain patterns and substrings does not deviate sufficiently from the average.

The first of these comments we agree with wholeheartedly, and will recommend that Q-values are used rather than p-values for the appropriate tests. For point 2, we agree that a careful model explaining the source of randomness is irreplaceable for any cryptographic application. Correspondingly, we stress that a PUF design capable of passing our test suite should not be immediately accepted as a source of high entropy randomness. Instead, the designer should have a clear proposal explaining how output bits are produced that indicates the design as a reliable source of entropy. However, a comprehensive test suite compliments a model well, especially to aid analysis of the hardware. The test suite can be used to detect anomalies in the output, assuming that the model indicates that the PUF under test should output strings that are sufficiently close to uniform. Additionally, if the physical design deviates from the model, a min-entropy estimator such as the 800-90B suite [16] does not inform the designer what the underlying issue is. Instead, a failed statistical test such as the monobit test may point towards more specific flaws in the design.

For the third point, we believe the issue is primarily one of methodology, especially in the PUF use case. Rather than putting a single string through the tests, it is preferable to put many strings from the same generator through the tests. In our case, the many strings should correspond to individual PUF instances. Then, a set of p-values (or Q-values) obtained for each test can be checked against the uniform distribution to confirm that they are as expected; false positives on this check should be less likely than running a single string through all 15 tests. Additionally, the comment claims that a single, uniformly random string run through all tests at a p-value of 0.01 should fail at least one test with probability around 0.14 using a simple independence and binomial argument. However, we find this argument to be somewhat flawed, as the assumption of independence between the p-values of the tests is inaccurate, as shown in other comments in [14]. For example, a skewed $0 : 1$ clearly affects the outcome of the majority of the tests.

For the final point, again we believe that some of the issue could be solved by correct testing methodology. Testing the spread of p-values from many inputs rather than just observing the passes and fails should indicate whether or not the output data is artificially balanced, as well as unacceptably imbalanced. Additionally, the specific inputs considered in the proposed test have input length $N \cdot 2^N$ for N at least 12, which is too large for most PUF designs. However, we are open to including a test for specious randomness in our suite if such a test is deemed prudent by NIST and can be parameterized for input lengths more suited to PUF outputs.

III. Per-Array Randomness

In this section we describe the randomness tests that are suitable to run on a per-array basis. These tests are used to provide evidence that the set of PUF outputs behave individually as one would expect for random data. We base our randomness testing on those tests from the NIST 800-22 suite that are suitable for the case of PUFs. For two tests, the Binary Rank Matrix and Template Matching tests, we construct novel variants of the tests using precise calculations of the behaviour of uniformly random data to lower the required input length. We will also explain how to input the PUF data into the tests, as some of the tests are sensitive to the ordering of the input data, and there are multiple ways to order an array of PUF output into a single string.

The tests in this section do not say anything about possible correlations within the set of PUF reads. For example, if we fix one length n string ε_1 that passes all randomness tests at a significance level of 0.001, and another ε_2 that fails all tests at the same level, a PUF design whose outputs were either ε_1 with probability 0.999 and ε_2 otherwise would pass these tests[1], but this PUF should clearly not be used to provide cryptographic keying material. The possibility of correlations across various arrays will be handled in the next section.

A. Ordering PUF Data

Although some randomness tests only care about global features of the data, many are sensitive to the ordering in which the data is input to them. For example, the strings $00110011\ldots0011$ and 000000...111111 (e.g. $n/2$ 0s followed by $n/2$ 1s) both have an even mix of 0 and 1 but the first performs much better on the Autocorrelation test than the second. Many PUFs produce output that is most naturally interpreted in an $n = n_1 \times n_2$ array. Since the randomness tests require input in the form of a string, there are two obvious choices for inputting the data: row-ordering and column-ordering, where the data is either concatenated row-by-row or column-by-column. Given the previous example, we will recommend that some tests are performed in both orders: otherwise a PUF whose output displays small correlation between row-adjacent bits but not column-adjacent bits would perform differently on the tests depending on input order. This may lead to misleading results if the tests are run in only a single order, as ultimately bad performance in either row or column ordering is evidence that the PUF is not sufficiently random.

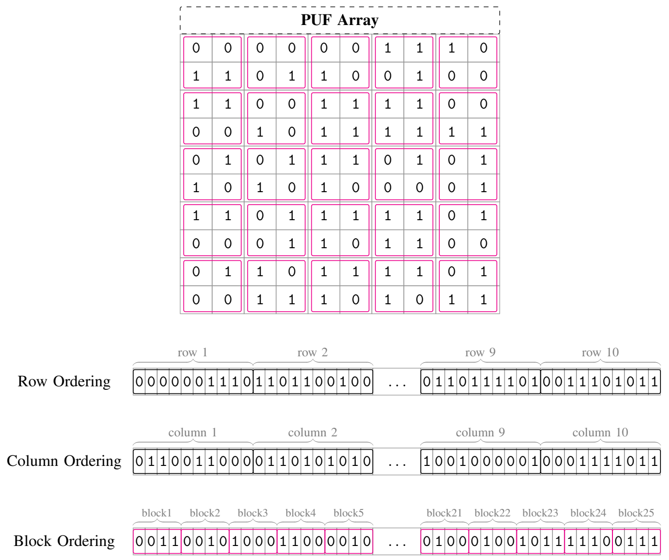

Fig. 1. A figure showing different possible orderings of a PUF read with $n_1 = n_2 = 10, b_1 = b_2 = 2, n_{blocks} = 25$.

We further suggest that these two orderings oversimplify a potential spatial aspect of PUFs: nearby cells may exhibit some correlated behaviour. To counteract this, we suggest a third ordering for some tests, which we call block ordering. In this order, an $n_1 \times n_2$ PUF is split into n_{blocks} evenly sized $b_1 \times b_2$ blocks, so that $n_{blocks} \cdot b_1 \cdot b_2 \approx n$. Since there is not a standard dimension for PUF arrays, we do not suggest explicit choices for b_1, b_2. Instead, the tester must exercise some discretion: for PUFs with sufficiently favourable dimensions, it will be easy to split the output into many square blocks e.g. if $n_1 = n_2$ and are perfect squares. For PUFs with less accommodating dimensions, some bits may need to be removed or the blocks may need to be rectangular. However, one bound to be aware of is an upper bound on n_{blocks}. Since these blocks will later be input in to the Block Frequency Test, which accepts at most 100 blocks of input, b_1 and b_2 should be chosen so that n_{blocks} does not exceed 100. Regardless of the explicit parameter settings, the blocks should be arranged so that the PUF splits into around $\frac{n_1}{b_1} \cdot \frac{n_2}{b_2}$ blocks laid out in an $\frac{n_1}{b_1} \times \frac{n_2}{b_2}$ matrix. Since practical choices of parameters depend greatly on the exact dimensions of the PUF, we leave the final choice up to the tester; in absence of specific parameters, we suggest square blocks of dimension 4 as a reasonable starting point. We remark that for the selected tests, whether the blocks are then concatenated into a single string by row or by column should not substantially effect the results.

B. Applying Randomness Tests To PUFs

Of the 16 tests presented in Section II, not all are appropriate to apply to PUFs. Additionally, some tests should be run in different orderings, whereas others only examine global features of the data. In Table I we present a summary of

[1]At least, as long as one does not test the resulting p-values thoroughly and instead just counts the pass rate.

TABLE I
THE RECOMMENDED ORDERINGS FOR EACH OF THE RANDOMNESS TESTS CONSIDERED

Test	**Recommended Orderings**		
	Row?	**Column?**	**Block?**
Monobit	✓	✗	✗
Block Frequency	✓	✓	✓
Runs	✓	✓	✗
Longest Runs	✓	✓	✗
Binary Matrix Rank	✓	✓	✓
Discrete Fourier Transform	✓	✓	✗
4-bit Template Matching	✓	✓	✗
Serial	✓	✓	✗
Approximate Entropy	✓	✓	✗
Cumulative Sums	✓	✓	✓
Non-Overlapping Template Matching	✗	✗	✗
Overlapping Template Matching	✗	✗	✗
Maurer's Universal Statistic	✗	✗	✗
Linear Complexity	✗	✗	✗
Random Excursions	✗	✗	✗
Autocorrelation Parameter 1	✗	✗	✗

which tests should be run in which orders, following which we explain our reasoning.

1) Tests Requiring Only One Order: The only global test is the Monobit test: this test checks that the balance of 0s and 1s in the whole string is consistent with a uniform distribution. Re-ordering the input does not effect the output of the Monobit test, and so it only needs to be run once.

2) Row and Column Ordered Tests: The next set of tests are the ones we recommend running in row and column order. These tests are sensitive to the order in which the data is input, but do not measure aperiodic local behaviour, which is the main focus of the block ordering. Additionally, tests that are interested in short runs and short templates do not need to be tested in block order, as the row (or column) length of the block may be long enough to capture the behaviour that these tests analyze, and the performance will be unchanged by moving from row ordering to block ordering. Periodic behaviours should be tested in both row and column order in case the PUF is constructed in such a way that rows behave differently to columns. Of the tests in this category, six test for highly localized behaviour: the Runs, Longest-run-of-ones in a Block, 4-bit Template Matching, and Approximate Entropy tests all concern themselves with short patternic behaviour. The final test, the Discrete Fourier Transform test, analyzes periodic behaviour at a depth that makes sense to be measured on a per-row or per-column basis, but not per block.

3) Tests Requiring Block Ordering: The final category of useful tests are those that analyze local behaviour on a scale suitable for inputting the data in block order. Some PUF designs may exhibit local behaviour, in the sense that cells physically close to each other exhibit correlated values, more generally than by row and column. The Block Frequency test examines the local $0 : 1$ balance of the data by splitting it into blocks, which we argue is meaningful on a per-row, per-column, and per-block basis, as all three orderings present natural blocks to examine the balance of 0's and 1's. The other test, the Cumulative Sums test, tests for peaks in the random walk defined by the input data. Inputting the data in block format will lead to large peaks if some blocks contain too strong a preference towards 0 or 1, which would be the case if there was a local bias in either direction.

4) Tests Excluded Due to Excessive Input Requirements: The remaining tests are not suitable for PUFs due to restrictions on acceptable input lengths. With the exception of the Binary Rank Matrix test, each requires at least a million bits of input. We are unaware of many mainstream PUFs that achieve this length, and thus consider these tests not practical for examining PUF randomness.

5) Simplifying the Template Matching Tests: Both template matching tests within the NIST test-suite require an unreasonably large amount of input bits for PUF applications. Specifically, [1] recommends at least a million bits for the overlapping test. For the non-overlapping variant, no stringent requirements are given, although it seems to be relatively common knowledge in randomness literature that approximately a million bits are also required for this test; this mismatch has led to the test being erroneously applied to PUFs e.g. [12]. Instead, we recommend a novel, more compact version of the test, based on the version from [17]. In [17], they construct template matching tests for 4-bit templates, demonstrating that the appearance probabilities of the patterns depend only on the degree of overlap, defined as the maximum length $i < 4$ such that the first and last i bits of the template are the same.

Rather than relying on convergence arguments, the test of [17] splits the input string into blocks of length 128, counting the number of occurrences of the target template in each block. Finally, the appearance counts are compared with the expected probabilities using a χ^2 test and direct calculation of the occurrence probabilities. In order for the χ^2 test to have a sufficient amount of expected occurences per bucket, the test requires a minimum of 43 blocks, or 5504 bits. Our test is similar to theirs, except that in order to further reduce the number of required bits we use 64-bit blocks. We also find errors within their occurrence probability proofs, which we are unable to fix analytically. Instead, we efficiently computed the exact occurrence probability for given templates in a 64-bit string via a recursive "brute force" computation. More details on this calculation are given in the Appendix. We also do not include wrap around occurrences for the template (where the first 3 bits of each block are appended to the end of the block) as we believe this does not well-represent spatial aspects of a PUF design. As a result of the new parameters, our test requires slightly under 2000 bits; more specifically 1920 or more. This ensures that the expected frequency of each bucket in Table II is at least 5. A complete description of our test is given below.

As with the template matching tests from the NIST specification, this test is actually one test for each possible template. Since there are 16 possible 4 bit templates, this results in 16 distinct tests.

Function Call: Template(t, n), where:

t	The chosen 4-bit template
n	The length of the device output

TABLE II
BINS AND PROBABILITIES FOR DIFFERENT OVERLAP LENGTHS

Overlap Length							
0-bit		**1-bit**		**2-bit**		**3-bit**	
0–2	0.188312	0–2	0.237221	0–2	0.275492	0–2	0.388599
3	0.23798	3	0.215906	3	0.199207	3–4	0.26515
4	0.26006	4	0.216319	4	0.18832	5–6	0.176063
> 4	0.313648	> 4	0.330554	> 4	0.336981	> 6	0.170188

Additional input required by the function:

ε The sequence of bits of the device under test.

Test Statistic: $\chi^2(\varepsilon)$, which measures the appearance frequency of the template per 64-bit block against the expected frequencies.

Description:

1) Split the input string into $M = \left\lfloor \frac{n}{64} \right\rfloor$ many 64-bit blocks $B_1, \ldots, B_M$, discarding any excess bits.
2) For each block, count the number c_j of (possibly overlapping) occurrences of the template string t in block B_j.
3) Compute the overlap of the string $t = t_1t_2t_3t_4$, defined as the largest $i < 4$ such that the prefix and suffix of t of length i are equal.
4) Locate the appropriate column of Table II corresponding to the overlap of t, and transform the list of counts c_j into a frequency count, letting #Buckets[2] denote the number of buckets and F_i the frequency with which c_j lies in the ith bucket for $j = 1, \ldots, M$.
5) Apply the appropriate χ^2 goodness of fit test by computing
$$\chi^2(\varepsilon) = \sum_{i=1}^{\#\text{Buckets}} \frac{(F_i - M \cdot p_i)^2}{M \cdot p_i}$$
and corresponding p-value $p_{template} =$ **igam**$((\#\text{Buckets-1})/2, \chi^2(\varepsilon)/2)$. Here p_i denotes the probability that the number of occurrences c_j of the template t in a 64-bit block lies in the ith bucket, whose value may be found in Table II.
6) Compare the p-value with the chosen threshold to determine whether the test passes or fails.

Here **igam** denotes the usual incomplete gamma function.

6) Reparametrizing the Binary Matrix Rank Test: The NIST test-suite recommends running the Binary Matrix Rank test with at least 38 matrices, each of dimension 32×32. This parameterisation is inapplicable to the per-PUF setting as it requires far too many test bits. We recommend the use of at least 44 matrices of dimension 4×4 instead, which we justify as follows. Firstly, for this test in a block ordering we are interested in highly localized non-random behaviour. Therefore, we posit that a dimension of 4 suffices to capture these localised correlations. Note also that linear dependencies in dimension higher than 4 imply linear dependencies in dimension 4, meaning that running the test in dimension 4 captures some of the information obtained if a higher dimension was to be used. We also recommend using at least 44 matrices. This is due to the fact that a Chi-squared test is applied to the ranks of the test matrices, the null-hypothesis of which is that the matrices are uniformly distributed. In order to justify the use of a Chi-squared test, one should follow the guideline that the expected number of occurrences in each category is at least 5. Categorising the test matrices as having either rank $4, 3$ or ≤ 2 leads to expected probabilities of 0.308, 0.577 and 0.115 for the respective categories (see [1] for an explicit probability formula). Therefore, using 44 matrices ensures that the expected number of occurrences of each category is at least 5, justifying the applicability of a Chi-squared test. Otherwise, the test is the same as described in Section 2.5 of [1], with the chosen matrices and hard coded probability values for ranks replaced with our own described above.

7) Choosing Parameters and Block Sizes: Many of the suggested tests have further input parameters as well as the length n of the input string. Although these tests have some flexibility in how these parameters are selected, the convergence arguments underlying the tests enforce restrictions, and these parameters must not be chosen arbitrarily. Typically, restrictions come in the form of an upper bound on the ratio between block sizes and length of input string, such as in the Approximate Entropy test, where the block size m must be less than $\lfloor \log_2(n) \rfloor - 5$. Block size restrictions are not a problem for RNGs capable of outputting arbitrarily many bits, but, in the case of PUFs the fixed output length can make choosing block sizes challenging, especially when trying to choose block sizes with some spatial interpretation. For most tests a natural block size would be to choose m corresponding to either row length or column length, but this is unlikely to be compatible with restrictions such as $m < \lfloor \log_2(n) \rfloor - 5$ unless there is a huge gap between row and column sizes of the underlying PUF design.

Restrictions on parameter settings vary for specific tests may be found in [1], and interact with the dimensions of the PUF being tested. Again, it is challenging to make concrete recommendations for how they are selected. Other tests, such as the Serial test, contain pattern sizes: these tests check for the appearance behaviour of specific patterns within the tested data. For these tests, pattern sizes are typically small, and the choice of parameter is upper bounded by a function of the input length. However, we provide a list of recommendations below which we hope should be sufficiently general to apply to most PUFs and tests. These recommendations assume the data is being concatenated by row; for data concatenated in column order, replace each instance of the word row with column

1) If possible, select block sizes equal to the row length. If this is not possible, choose a size that divides the row length or is close to a divisor of the row length. This ensures that blocks correspond as closely as possible to rows or parts of rows.
2) For Template Matching tests, use an exact table of probabilities for statistical testing such as Table II.
3) For the Binary Matrix Rank test in block order, split the PUF output into 4×4 matrices, discarding any excess

[2]Note #Buckets = 4 in all cases; we leave it open as one could imagine parameterizations of the test where this need not be the case.

bits if necessary.

4) For the remaining relevant tests, always ensure that all parameters chosen are compatible with the restrictions present in Section 2 of [1].

IV. Cross Device Testing

In the previous section, we explained the tests for randomness on a per-device level: each test takes as input the read of a single device, and returns a pass or fail depending on the properties of that device. However, it is also important that each device is independent from the other devices, so that not only is each output random when considered individually, but also when considered in relation to the other devices. It is typical to examine this behaviour by considering the statistics of the inter-array Fractional Hamming Distance of the test devices as in [12], [18]. For a pair of device outputs $\varepsilon_1, \varepsilon_2$, define their Fractional Hamming Distance FHD$(\varepsilon_1, \varepsilon_2)$ as the usual definition for binary strings:

$$\text{FHD}(\varepsilon_1, \varepsilon_2) = \frac{1}{n} \cdot \sum_{i=1}^{n} {\varepsilon_1}^i \oplus {\varepsilon_2}^i$$

where ${\varepsilon_1}^i$ denotes the ith value of ε_1.

For an ideal PUF of length n, the distribution of FHDs of pairs of PUF devices converges to the normal distribution with mean $1/2$ and standard deviation $1/(2\sqrt{n})$ as n and the number of PUFs converge to infinity. Consequentially, the usual method to examine the independence of k PUF outputs is to take the set of all $\binom{k}{2}$ FHD pairs, each assumed to be sampled from the appropriate normal distribution, and compare the sample mean and standard deviation with the ideal values. However, this technique has the drawback of not providing pass-fail behaviour. Even for a good quality PUF design, it is likely that the statistics will not perfectly match the ideal values due to standard features of finite sample sizes. Although one could envision various cut-off points, it is not clear at what point the statistics are far enough from the ideal values to conclude that the PUF reads are not sufficiently independent from one another. Instead, we would like to have a test of independence that can be applied on a per-device basis, so that it is easy to collect a large amount of p-values and apply pass-fail and check p-value spread as usual.

A. Inter-Array FHD Test

In order to have a statistical test with a per-device pass-fail behaviour, we suggest a novel test based on the FHD statistics of the entire set of devices. Although this test assigns a p-value and a pass or fail to each device individually, it still requires the output values of all devices as input. We will describe the test in detail below, following which we explain why we believe it to be a good test for our goals.

B. Test Description

Function Call: FHD(k, n), where:

k	The number of devices
n	The length of each device output

Additional input required by the function:

ε_1	The sequence of bits of the device under test.
$\varepsilon_2, \ldots, \varepsilon_k$	The sequence of bits of the other devices.

Test Statistic: $\chi^2(\varepsilon_1)$, which measures how well the set of FHDs between the device under test and the other devices matches the expected normal distribution of truly random data.

Description:

1) Compute $\text{FHD}_{1,i} = \text{FHD}(\varepsilon_1, \varepsilon_i)$ for $i = 2, \ldots, k$, the Fractional Hamming Distance between the array under test and each other array.
2) Compute the χ^2 statistic, $\chi^2(\varepsilon_1) = 4 \cdot n \cdot \sum_{i=2}^{k} (\text{FHD}_{1,i} - 1/2)^2$.
3) Compute the p-value $p_{FHD} = \textbf{igam}((k-1)/2, \chi^2(\varepsilon_1)/2)$.
4) Compare the p-value with the chosen threshold to determine whether the test passes or fails.

Here **igam** denotes the usual incomplete gamma function.

C. Test Explanation

The purpose of the test is to determine whether or not the distribution of FHDs between the tested device and each other device in the data set follows what one would expect for independent uniformly random strings. Assuming the initial hypothesis that the input data is truly random, the test statistic $\chi^2(\varepsilon)$ should follow a χ^2 distribution with $k-1$ degrees of freedom. The incomplete gamma function is then used to transform this distribution into a uniform distribution, following which the result can be compared with the chosen p-value.

D. Convergence and Independence

Although it is true that, for an ideal PUF, testing each device outputs a p-value that is uniformly distributed, it is not necessarily true that the collection of p-values derived from testing all devices is independent. Since the test compares each device against each other device, some of the FHD comparisons contribute to the test statistic for multiple devices. A simple example of this is that FHD$(\varepsilon_1, \varepsilon_2)$ contributes to the test statistic for the first two devices. Indeed, this is not the end of the story, as there are more complicated dependencies in the set of FHDs: another example is that knowing the values of FHD$(\varepsilon_1, \varepsilon_2)$ and FHD$(\varepsilon_2, \varepsilon_3)$ may provide some information on FHD$(\varepsilon_1, \varepsilon_3)$[3].

Nonetheless, we conjecture that the distribution of the set of p-values converges to the uniform distribution over $[0, 1]$ as both n and k converge to infinity. As a slightly stronger claim, we suspect that the set of χ^2 statistics converges to a set of independent observations from the χ^2 distribution with $k-1$ degrees of freedom, from which it follows that the p-values satisfy the desired convergence. Note that we require convergence to infinity in both n and k; it is relatively straightforward to see how fixing one to be relatively small

[3]An easy way to see this is to set the first two values to 0, upon which the third must also be 0.

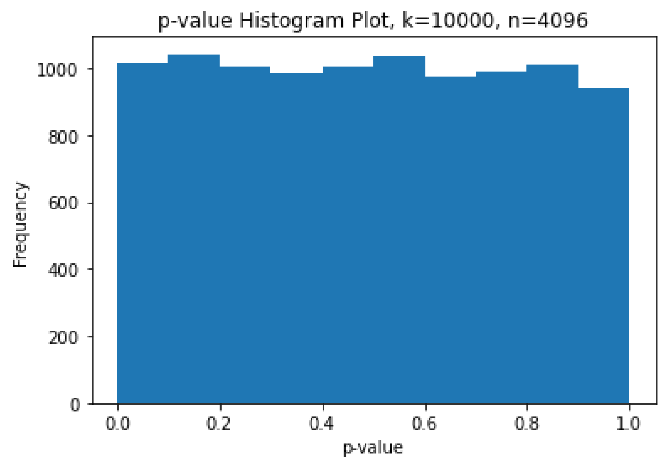

Fig. 2. A histogram of FHD p-values for $k = 10,000$ uniformly distributed bit-strings of length $n = 4096$.

and letting the other converge to infinity does not result in independent χ^2 statistics.

Unfortunately, we are unable to prove this property of our new test, although simulated outcomes of the test for large n and k corroborate our belief that the test functions as intended. We make it clear that this is unproven and formally state our conjecture as follows.

Conjecture 1: Let $p_1, \ldots, p_n \leftarrow \text{FHD}(k, n)$ denote the set of p-values derived from performing the FHD test on a set of k samples from the uniform distribution over $\{0,1\}^n$. Then, as $n, k \to \infty$, the distribution of $p_1, \ldots, p_n$ converges to $\mathcal{U}(0,1)^n$.
To support this conjecture, we provide a plot (Fig. IV-D) of the p-values for $\text{FHD}(k, n)$ exhibiting an almost uniform nature using $k = 10,000$ and $n = 4096$. The plotted p-values have a χ^2 test score of 0.501 when tested for uniformity.

We note that the same dependencies occur in the usual method of comparing the sample mean and standard deviation of the set of inter-array FHD scores against the ideal values, as is typically done to analyze the independence of the PUF arrays. However, the statistics are still compared to the mean and standard deviation of the normal distribution, which implicitly assumes that the FHD scores are independent.

V. Test Suite

In the previous two sections, we defined the tests that will form our suite. Since the tests were spread out and introduced along with some justifications, in this section we provide a comprehensive summary of the test suite. We leave the details of the individual tests to the previous sections; the main contributions of this section will be in explaining how to interpret the test results and how to choose the input values from an unstable PUF. All tests in our suite take as input a binary string, so for each PUF in the test set the input values must be fixed before the tests are run. The tests will then output a collection of p-values to be interpreted.

A. *Interpreting the Results*

Each result described in the suite will return a p-value. Under the assumption that the input data is truly random, the p-values for each test should be uniformly distributed over the interval $(0, 1)$. Before beginning the test suite, the significance level α should be determined by considering the number of devices in the data set. For those unfamiliar with statistical testing, we recommend choosing 0.01 for a small data set (say below 1000 devices) and 0.001 otherwise. For a given α, each test outputs a pass or a fail on each input string. If the proportion of devices failing a test is not close to α, this evidence suggests that there is a design flaw with the PUF design under test. If all tests have a pass-fail rate close to α, the test suite supports the hypothesis that each PUF outputs a string of uniformly random bits. For a given test, an acceptable proportion of failures at significance level α and k sample devices is determined by the value

$$(1-\alpha) \pm 3\sqrt{\frac{(1-\alpha)\cdot\alpha}{k}}$$

and if the proportion is outside this interval, the tests provide evidence that the data tested, and thus the PUF design, is not uniformly random. It should be noted that this interval is determined by following a binomial approximation to normal, which is most appropriate for values of k approximately 10 times as large as α.

B. *p-values and Q-values*

A more thorough tester should also compare the distribution of p-values resulting from a test to the uniform distribution. This may be done using a standard χ^2 goodness of fit test; for those unfamiliar, this technique is described in detail in Section 4.2.1 of [1]. If the p-values are not sufficiently close to the uniform distribution, this is also evidence that the input data was not truly random. However, to complete this analysis, we must split the tests in our suite into two classes: those for which we recommend following the recommendations of Section 4.2 of [1] to analyze the p-values, and those for which we suggest using Q-values as in [15]. Details on Q-values may be found in [15], but we summarize the key information here. For tests whose statistic follows a binomial distribution, the distribution of p-values derived from applying the test to uniform data does not follow a uniform distribution. Instead, the statistic follows a folded normal distribution. Since we would like to recommend comparing p-value spreads to the uniform distribution, this is unsuitable for our use case. To improve this, the Q-value is introduced as:

$$Q = \frac{1}{2}\text{erfc}\left(\frac{d}{\sqrt{2}}\right)$$

where erfc denotes the standard complementary error function and d the statistic of the test in question. For completeness, we note that there is a simple relationship between p- and Q-values, with

$$p = \begin{cases} 2Q, & Q \leq 0.5 \\ 2(1-Q) & Q \geq 0.5 \end{cases}.$$

The tests whose statistic d follows a binomial distribution should have the Q values examined for uniformity rather than the p-values. These tests include:

- The Monobit Test
- The DFT Test
- The Runs Test.

For these tests, the analysis is the same as the others, except that the Q-value should be used in place of the p-value.

C. Test Suite Instructions

1) Stage 1: Determine the Dimensions and Input Values: First, the tester must determine the dimension parameters and input values used by the test. This should be done by analyzing the design of the PUF, including the intended post-processing stage, if any. The length n of the input strings should be equal to the number of bits output by each instance of the PUF. If the PUF bits are produced by individual cells that are laid out in a matrix-like array, this determines the row and column length n_1, n_2 with $n = n_1 \cdot n_2$. Otherwise, there is only a single parameter n. If the PUF cells are produced in a matrix-like array, the tester must also determine the block dimensions b_1, b_2 as described in Section III.

Next, the tester must determine the values to be input for each array. The natural way to do this is to take a single read of each PUF cell, and set the value of the cell to the value of that read. This measures the randomness of a read of the PUF; however, if the intended use case of the PUF involves some per-bit error processing, it may be more important to measure the randomness of the error-corrected bit values. In this case, it may be preferable to take many reads of each cell and determine its 'true' value by majority voting. Whether or not a single read or a majority vote value is appropriate to be tested depends on the use case of the PUF, but in absence of a known use case we recommend that the randomness of a single read is analyzed, as this tests the randomness of the raw PUF output.

2) Stage 2: Randomness Tests in Row-Order Only: Next, we begin to run the randomness tests. For these tests, each device is tested separately, with the $n = n_1 \times n_2$ bit string input concatenated in row-order if applicable. The only test in this category is:

1) The Monobit Test.

3) Stage 3: Randomness Tests in Row- and Column-order: For the following tests, each device should have two strings input to the test: with the input concatenated into a single string by row and by column. If the device does not have rows and columns, then only one input is provided to each test. The tests in this category are:

1) The Runs Test.
2) The Longest-Runs-of-One in a Block Test.
3) The Serial Test.
4) The Discrete Fourier Transform Test.
5) The Approximate Entropy Test.
6) The 4-bit Template Matching Tests.

Note that the details of the 4-bit Template Matching Tests are given in Section III-B5, and that this is one distinct test for each of the 16 possible 4-bit templates.

4) Stage 4: Randomness Tests in Row-, Column-, and Block-Order: The following tests should be run three times, with the input concatenated by row, column, and block. Once again, if the device does not have rows and columns, then only one input is provided. The tests in this category are:

1) The Block Frequency Test.
2) The Cumulative Sums Test.
3) The Binary Matrix Rank Test.

Note that the Binary Matrix Rank test will have different block sizes than the other two, as described in Section III-B6 and Section III-B7.

5) Stage 5: Testing for Device Independence: Finally, the set of devices should be tested for independence. The order in which the data is input to this test does not matter, as long as it is consistent across all devices in the dataset. The test in this category is:

1) The FHD Test.

VI. Conclusion

In this paper, we introduced a randomness test suite specialized to the case of testing PUF randomness. Our suite is built by adapting pre-existing tests designed for RNGs to the case of PUFs, taking into account changes required resulting from their physical layout and limited output length. These considerations include applying tests on different orderings of the input data and choosing parameters that take into account the spatial properties of the PUF cells. Additionally, we introduced a per-device test of independence, providing a statistical test of whether or not devices behave independently from one another.

For future work, we are open to the possibility that our test suite may need to be expanded, as there may be as-yet-unknown modes of failure for a PUF that our test suite does not pick up on. Additionally, we were unable to prove that our novel test of device independence, the Inter-Array FHD test, produces a set of independent p-values; there would be some value to proving the conjecture of Section IV. If the conjecture was instead disproved, this would effect the use of FHD statistics in determining the independence of PUF arrays used in previous work.

Another direction of future research would be to develop an entropy estimator for a PUF, the PUF analogy of that provided for min-entropy of RNGS in NIST Special Publication 800-90B [16]. This may be somewhat more challenging than providing statistical tests, as the algorithm of NIST 800-90B takes significant advantage of the fact that an RNG provides a continuous string of output bits, and that it is easy to request more bits as required. However, there are substantial advantages to having an estimate of PUF entropy: randomness extraction is a common part of a fuzzy extractor, and tight, or at least conservative, bounds on the min-entropy of the PUF output allows for stronger arguments about the entropy of the post-processed output.

Overall, we expect our suite to prove valuable to novel PUF designers as a method to validate that their PUF performs as expected. Our suite is intended to be defined so that the designer does not need to understand the low level details of each test. Of course, if they find that their PUF performs badly against a specific test, understanding the properties of that test will help explain any corresponding weaknesses of the PUF. Finally, we once again stress that randomness tests should not be used in place of a model of the process by which PUF bits are created. Without a good model, there is no initial hypothesis for the tests, which require the assumption that the data under test is a string of uniformly random bits. This initial hypothesis is only applicable when derived from a model justifying that the output bits from the PUF are expected to have good entropy, and thus the PUF is suitable to use as a source of randomness.

References

[1] L. E. B. III, A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. L. Banks, A. Heckert, J. Dray, and S. Vo, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," National Institute of Standards & Technology (NIST), Special Publication 800-22 Rev. 1a., 2010.

[2] W. Killmann and W. Schindler, "A proposal for: Functionality classes for random number generators," Bundesamt für Sicherheit Informa-tion–stechnik, Technical Report, 2011.

[3] G. Marsaglia, "DIEHARD: a battery of tests of randomness," 1995, http://www.stat.fsu.edu/pub/diehard/.

[4] R. G. Brown, "DIEHARDER: A random number test suite," 2003, https://webhome.phy.duke.edu/~rgb/General/dieharder.php.

[5] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas, "Silicon physical random functions," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, ser. CCS '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 148–160. [Online]. Available: https://doi.org/10.1145/586110.586132

[6] G. E. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation," in *2007 44th ACM/IEEE Design Automation Conference*, 2007, pp. 9–14.

[7] J. Delvaux, D. Gu, I. Verbauwhede, M. Hiller, and M.-D. M. Yu, "Efficient fuzzy extraction of puf-induced secrets: Theory and applications," in *Cryptographic Hardware and Embedded Systems – CHES 2016*, B. Gierlichs and A. Y. Poschmann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 412–431.

[8] M. Shiozaki, Y. Hori, and T. Fujino, "Entropy estimation of physically unclonable functions with offset error," Cryptology ePrint Archive, Report 2020/1284, 2020, https://ia.cr/2020/1284.

[9] A. Vijayakumar, V. C. Patil, C. B. Prado, and S. Kundu, "Machine learning resistant strong puf: Possible or a pipe dream?" in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2016, pp. 19–24.

[10] U. Rührmair and J. Sölter, "Puf modeling attacks: An introduction and overview," *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, 2014.

[11] Y. Dodis, L. Reyzin, and A. Smith, "Fuzzy extractors: How to generate strong keys from biometrics and other noisy data," in *Advances in Cryptology - EUROCRYPT 2004*, C. Cachin and J. L. Camenisch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 523–540.

[12] K.-H. Chuang, E. Bury, R. Degraeve, B. Kaczer, D. Linten, and I. Verbauwhede, "A physically unclonable function using soft oxide breakdown featuring 0

[13] R. Maes, P. Tuyls, and I. Verbauwhede, "A soft decision helper data algorithm for sram pufs," in *2009 IEEE International Symposium on Information Theory*, 2009, pp. 2101–2105.

[14] NIST, "Decision to revise NIST SP 800-22 rev. 1a," 2022, https://csrc.nist.gov/News/2022/decision-to-revise-nist-sp-800-22-rev-1a.

[15] S. Zhu, Y. Ma, J. Lin, J. Zhuang, and J. Jing, "More powerful and reliable second-level statistical randomness tests for nist sp 800-22," in *Advances in Cryptology – ASIACRYPT 2016*, J. H. Cheon and T. Takagi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 307–329.

[16] E. Barker and J. Kelsey, "NIST DRAFT special publication 800-90b recommendation for the entropy sources used for random bit generation," no. 800-90b, 2012.

[17] F. SULAK, "New statistical randomness tests: 4-bit template matching tests," *TURKISH JOURNAL OF MATHEMATICS*, vol. 41, pp. 80–95, 01 2017.

[18] M.-Y. Wu, T.-H. Yang, L.-C. Chen, C.-C. Lin, H.-C. Hu, F.-Y. Su, C.-M. Wang, J. P.-H. Huang, H.-M. Chen, C. C.-H. Lu, E. C.-S. Yang, and R. S.-J. Shen, "A puf scheme using competing oxide rupture with bit error rate approaching zero," in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, 2018, pp. 130–132.

Appendix

In order to reduce the number of bits required for Template Matching tests, an m-dependent Central Limit Theorem is replaced by a calculation of the *exact* distribution of template matches in the analysis. Suppose we consider a template $t = t_1 \dots t_m$ and uniform strings of length n. Since 2^n is quite large for the recommended value of $n = 64$, it is difficult to enumerate over all strings to derive the exact distribution. Instead, we use a recursive length-doubling strategy to efficiently compute the distribution of template matches. Of course, it would be preferable to provide an analytic formula for the distribution as in [17], where an explicit formula for the probability of k occurrences of a pattern in an n bit string is given. However, we find their formula to be inaccurate, due to an error in the proof relating to the possible circular symmetry of strings, and testing their formula for small values of n, k reveals inconsistencies in the output.

Instead, our recursive strategy begins as follows. Suppose we consider length n' strings where $n' \geq 2m$ and a fixed template. We consider the first $m-1$ bits as its prefix, and the last $m-1$ bits as its suffix. Note that there are 2^{m-1} prefixes and 2^{m-1} suffixes. Now suppose we have a $2^{m-1} \times 2^{m-1} \times n'$ matrix of probabilities $P^{(n')}$ where the (i, j, k)-th entry is the probability that a uniformly chosen n'-bit string has prefix index i, suffix index j and exactly k template matches. Define the discrepancy of suffix j and prefix i, denoted $D(j, i)$, as the number of template matches in the concatenation of suffix j and prefix i. Suppose we have two length n' bit strings: s_1 with suffix j and C_1 template matches, and s_2 with prefix i and C_2 template matches. The number of template matches in the concatenation of $s_1||s_2$ is then $C_1 + D(j, i) + C_2$. We can use this to count the number of concatenations of length n' string pairs that have prefix i, suffix j and k template matches. In terms of probabilities, we may conclude that

$$P^{(2n')}_{i,j,k} = \sum_{\substack{j_1,k_1,i_2,k_2: \\ k_1+k_2+D(j_1,i_2)=k}} P^{(n')}_{i,j_1,k_1} \cdot P^{(n')}_{i_2,j,k_2}.$$

Therefore, we can calculate $P^{(2n')}$ (which gives us the template count distribution over uniform strings of length $2n'$) efficiently from $P^{(n')}$ without enumerating over $2^{2n'}$ strings. This allows us to efficiently compute the template count distribution for $m = 4, n = 64$ in a matter of seconds by starting from enumerating over all strings of length 16.