An Architecture for Learning Stream Distributions with Application to RNG Testing

Alric Althoff, Ryan Kastner
Department of Computer Science and Engineering
University of California, San Diego
{aalthoff,kastner}@eng.ucsd.edu

ABSTRACT

Learning cumulative distribution functions (CDFs) is a widely studied problem in data stream summarization. While current techniques have efficient software implementations, their efficiency depends on updates to data structures that are not easily adapted to FPGA or ASIC implementation. In this work, we develop an algorithm and a compact hardware architecture for learning the CDF of a data stream and apply our technique to the problem of on-chip run-time testing for bias in the output of random number generators (RNGs). Unlike previous approaches, our method is successful regardless of the expected output distribution of the RNG under test.

1. INTRODUCTION

Computing the kth smallest element in a list is a common problem in computer science, and in this work, we discuss an approximate algorithm for doing so. Such statistics are frequently used in stream mining and database analysis to determine percentiles of network latency, assist server load balancing, and rank streaming objects.

As mentioned in [1], learning manually spaced data quantiles doesn't necessarily provide sufficient knowledge about a distribution. Without prior knowledge of some properties of the stream manual spacing may ignore areas where greater curvature is present in the cumulative distribution of the stream. One of the objectives of this paper is to address this in constant space and time per update in a way that does not depend upon the number N of elements in the stream. To the best of our knowledge this is the first algorithm implementing such an adaptive strategy in a way easily amenable to efficient hardware implementation.

While myriad applications exist for run-time quantile and cumulative distribution function (CDF) estimation, in this paper we apply our work to nonparametric run-time testing for bias in random number generators (RNGs). Other recent work [2, 3, 4, 5, 6, 7] focuses on implementing previous well-regarded tests [8] for bit-wise uniformity and independence. Our architecture generalizes this work in that it can be used when testing for bias in random values drawn from any distribution. This is particularly useful when many stages of post-processing are involved prior to use of the variates in question. For example, assume that a trusted uniform and independent bit-generating pseudorandom number generator (PRNG) generates integer variates used in an inversion sampler [9] to allow sampling from a Poisson distribution. Testing the output of the PRNG is useful, but the error and/or attack surface also includes the implementation of the inversion sampling algorithm and all other post-processing stages. Without a testing framework in place *immediately prior to the consumer* of the random numbers such an error could go undetected for a disturbingly long time.

In terms of cryptographic techniques, the quality of the random or pseudorandom number generation routine is of utmost importance. Over the past several years many security flaws have been at least in part caused by bias in either the method used to generate initial random variates, or simple coding errors in the interpretation of random values. E.g., as discussed in [10], a fencepost error created by using $\leq$ instead of $<$ can significantly compromise the security of a supposedly secure platform. While for off-line error checking it is often feasible to use a more time consuming and resource hungry technique such as an Anderson-Darling test [11], a straightforward hardware implementation would add considerable area to designs that may already be resource-starved. Our approach addresses these concerns, and can mitigate attacks that rely on manipulating the environmental or algorithmic surface involved in either random and pseudorandom number generation.

It is important to realize that the dangers of attacks or errors involving random number generation routines are not limited to cryptographic scenarios. Consider a hypothetical situation where one or several investing institutions use Monte Carlo simulation routines—which often require variates from specific non-uniform distributions—running on FPGAs to model properties of financial indicators. It is a well known result from the theory of dynamical systems that small periodic perturbations at well-placed intervals can cause a system previously thought to be stable to diverge dramatically [12]. If an attacker were to introduce a subtle bias in the random number generation routine at the right times, the model could be made to behave either erratically, or more interestingly, according the the whims of the attacker. Testing for bias immediately prior to use would mitigate such concerns.

DAC '17, June 18-22, 2017, Austin, TX, USA
 ISBN 978-1-4503-4927-7/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3061639.3062199

Algorithm 1 A Simple Quantile Learning Algorithm

Input: $\vec{\alpha}$, $\hat{Q}_0$, λ
Result: $\hat{Q}_t \approx Q$
$t \leftarrow 0$
while x_t **exists**
  for every $j \in [n]$
    $\hat{Q}_t(\alpha_j) \leftarrow \hat{Q}_{t-1}(\alpha_j) - \lambda \text{sgn}_\alpha(\hat{Q}_{t-1}(\alpha_j) - x_t)$
  end for
  $t \leftarrow t + 1$
end while

In summary, this work provides

- A novel algorithm for adaptively spacing CDF estimation points, designed to obtain higher precision in the tails of completely unknown distributions.
- An efficient FPGA hardware architecture for quantile and CDF estimation, requiring only constant storage independent of the length of the stream.
- A technique and architecture for discovering bias in random number generation pipelines, regardless of source, expected distribution, and true vs. pseudorandomness.

2. RELATED WORK

While a naïve algorithm for determining a quantile is $O(N \log N)$—sort ascending and pick the kth element—for the large number N of elements found in many databases, or when $N \to \infty$ as in a stream of data which is later summarized and discarded, such an algorithm is impossible to apply. Even substantially less intuitive approaches such as *Quickselect* [13] require $O(N)$ space. To address this issue, many algorithms have been developed for quantile approximations, (see [14, 15, 16] for recent examples,) that require a small fraction of the space of prior work. While these algorithms may have a very efficient software implementation, an efficient FPGA architecture would require implementing and maintaining the update algorithms and complex data structures that make these approaches possible, and hence require substantial overhead. In this work we develop an FPGA architecture—and introduce several nontrivial extensions—for the algorithm presented in [17] and [18] that only require constant space and time for both storage and updates. This algorithm is not as precise as those in the work mentioned above, and only achieves minimal error when the data stream elements are processed in time independent order. An algorithm without this requirement would return an estimate within ϵ of the true quantile function even when given a sorted stream—which is very nearly the worst case for our algorithms. In many situations this is a serious drawback, but in several important applications this can be quite advantageous, such as our bias tester in section 3.4, or other applications where testing for time independence is among the goals.

Hardware for testing RNGs related to our work here has been developed in many recent publications [3, 4, 6, 7, 5, 2]. The method of [7] implements several of the NIST [8] tests using dynamic reconfiguration due to the large hardware requirements of all 15 implementations of these tests. In [5] the authors implement versions of two of the NIST tests and optimize them by identifying common operations between tests and approximating the statistical thresholds used. The work of [4]—which is closest to our own—extends this by efficiently implementing eight of the 15 tests recommended by NIST, and approximating the thresholds. The work of [6] and [2] address this issue and provide extension to true—and in [2], non-ideal—random sources, but these works focus on RNGs where the output can be only one of two values, and are potentially insensitive to programming errors or post-processing-based attacks. In [3] the authors of [2] address environmental attacks on true RNGs—those RNGs extracting randomness from their physical environment—and use empirical tests to determine the behavior of the statistical features they extract. Several of the papers mentioned above present work designed to detect problems at the output of the RNG in accordance with the *health test* paradigm discussed in the relevant U.S. National Institute of Standards and Technology recommendation [19]. Our contribution addresses a gap in the taxonomy: nonparametric testing of random variables from any distribution at the *point of use*—as opposed to near the RNG.

3. BACKGROUND AND METHODOLOGY

3.1 Quantiles

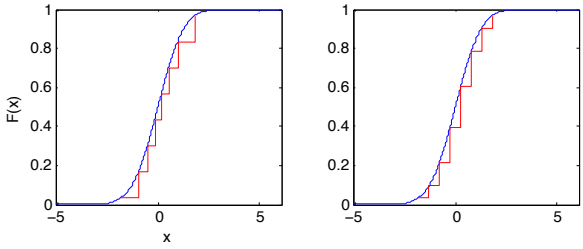

Figure 1: Results from Algorithms 1 (left) and 2. The blue curved line is the ideal true CDF. The points of the red steps are the learned result for $n = 8$, $\lambda = 10^{-2}$, and $\zeta = 10^{-4}$. Algorithm 1 misses detail in the tail regions, while Algorithm 2 starting from a equally spaced α vector adapts to represent the tails more densely.

An empirical quantile of some random variable $X \sim f_X$, where f_X is a probability density, is the value at a location in a sorted array L of unique entries drawn from f_X, where "location" is stated in terms of a fraction of the length of the array. That is to say, if Q is the quantile function of a probability density f_X, and N is the number of values drawn from f_X, sorted, and placed in L, then $Q(\alpha) = L[\lceil N \cdot \alpha \rceil]$ is the empirical α-quantile of f_X. This implies that the median equals the $\alpha = 0.5$ quantile, and quartiles are the $\alpha = [0.25, 0.5, 0.75]$ quantiles.

Put somewhat more formally, Q is the inverse of the CDF F_X of f_X. Assuming we treat a stream element x_t, taken at time t from the stream, as a sample from f_X, the CDF of that data stream is $F(z) = \Pr(x_t \leq z)$, and so the quantile function can be written

$$Q(\alpha) = F_X^{-1}(\alpha) = \inf \{x \in \text{supp}(F_X) : \alpha \leq F_X(x)\} \quad (1)$$

3.2 Stochastic Approximation of Quantiles

For each stream element x_t we use the update equation

defined in [18] for our approximation. This is

$$\hat{Q}_t(\alpha) \leftarrow \hat{Q}_{t-1}(\alpha) - \lambda \text{sgn}_\alpha(\hat{Q}_{t-1}(\alpha) - x_t) \tag{2}$$

where

$$\text{sgn}_\alpha(z) = \begin{cases} -\alpha & \text{if } z < 0 \\ 1-\alpha & \text{if } z \geq 0 \end{cases} \tag{3}$$

Given a family of α_j values at which to approximate the quantile function of a univariate stream we can run this update equation in parallel for each α_j.

Theorem: $\hat{Q}_t(\alpha)$ in Algorithm 1 converges to the α-quantile of the data stream generating x_t.

Proof Sketch: Informally, we begin with $\alpha = 0.5$, and the reasoning for all $\alpha \in [0, 1]$ flows naturally from this case. Let x_t be a stream element available at time t. Recall that by assumption x_t and x_{t-1} are independent in time for all t, and so have an equal chance of being both above the median, both below the median, or one on either side of the median. Assume $\hat{Q}_t(0.5)$ has converged. Then half of the time x_t cancels out x_{t-1} because $\text{sgn}_{0.5}(x_t) \in \{-0.5, 0.5\}$, and so $\hat{Q}_t(0.5)$ stays in place on average.

Next, assume $\hat{Q}_t(0.5)$ has not converged. If $\hat{Q}_t(0.5)$ is below the median, then there is a probability greater than 0.5 that $x_t \geq \hat{Q}_t(0.5)$ and so $\hat{Q}_t(0.5)$ will increase by 0.5λ at a rate equal to that probability. This will continue until convergence, i.e. until $x_t \geq \hat{Q}_t(0.5)$ with probability 0.5, which occurs exactly when $\hat{Q}_t(0.5)$ is within λ of the median of the observed stream elements. When considering any particular α the intuition remains the same.

3.3 Learning Cumulative Distributions

Algorithm 1 is useful when we desire $\hat{Q}(\alpha)$ for a fixed set of α values that we choose a priori. Now we will introduce a modification that allows us to learn α under the constraint that for $j \in [n]$ the $\hat{Q}(\alpha_j)$ values be equally spaced through the range of the distribution, with the exception that both α_1 and α_n remain fixed at their a priori values. These recovered α values under this constraint form the CDF computed at equally spaced points over the domain of f_X. This implies that the α values that we learn have relatively greater density in areas of low probability, which leads to more accurate tail estimates.

We accomplish this adaptation by adding the second finite difference of all $\hat{Q}_t(\alpha)$, denoted $\Delta^2[\hat{Q}]$ in the sequel, to the set of αs at each recursive step after attenuation to a small value in the range of the CDF. That is to say

$$\alpha_j \leftarrow \alpha_j + \zeta \Delta^2[\hat{Q}] \tag{4}$$

where ζ is a suitable step size. We advise practitioners to take care when selecting ζ. A heuristic is to set $\zeta = (C \cdot n)^{-1}$, for some large constant C. This is because (a) all CDFs are sharply bounded in $[0, 1]$, and (b) we are simultaneously learning $\hat{Q}_t(\alpha)$, and too great a change in α can destabilize other algorithmic components. Note that if ζ is not sufficiently small then this algorithm can fail, and oscillations in the results indicate that a smaller step size is required. For a visual comparison between the results of Algorithms 1 and 2 see Figure 1.

Theorem: $\hat{Q}_t(\alpha)$ in Algorithm 2 converges to equally spaced points over between $\hat{Q}_t(\alpha_1)$ and $\hat{Q}_t(\alpha_n)$, while these two converge to those α-quantiles of the data stream generating x_t.

Algorithm 2 A Distribution Learning Algorithm

Input: $\vec{\alpha}$, $\hat{Q}_0$, λ, ζ
Result: $\vec{\alpha} \approx F_X$
$t \leftarrow 0$
while x_t **exists**
  for every $1 < j < n$
    $\hat{Q}_t(\alpha_j) \leftarrow \hat{Q}_{t-1}(\alpha_j) - \lambda \text{sgn}_\alpha(\hat{Q}_{t-1}(\alpha_j) - x_t)$
    $\alpha_j \leftarrow \alpha_j + \zeta \Delta^2[\hat{Q}_t]$
  end for
  $t \leftarrow t + 1$
end while

Proof Sketch: This is easy to see when considering that adding $\zeta \Delta^2[\hat{Q}_t]$, for small ζ, forces the second finite difference toward zero, $\zeta \Delta^2[\hat{Q}_t]$ toward a constant, and thus $\hat{Q}_t$ towards linearity. If $\hat{Q}_t(\alpha)$ are equally spaced, then α are equal to points on the CDF F_X, where $Q(\alpha) = F_X^{-1}(\alpha)$, and $\hat{Q}_t(\alpha) \approx Q(\alpha)$.

3.4 PRNG and RNG Monitoring

To determine whether our RNG under test is biased, we may compute at regular intervals a p-value as

$$z_j = \alpha_j - \frac{|\hat{Q}_t(\alpha_j) - Q(\alpha_j)|}{\lambda t} \tag{5}$$

$$p = \min_j \left\{ 2n \left(\left(\frac{\alpha_j}{z_j} \right)^{z_j} \left(\frac{1-\alpha_j}{1-z_j} \right)^{1-z_j} \right)^t \right\} \tag{6}$$

where t is the number of values tested so far. Incorporating the factor of $2n$ corrects the probability for both the two-tailed nature of the test, and the fact that $1 \leq j \leq n$. We derive p from the Chernoff-Hoeffding tail bound on sums of independent Bernoulli random variables.

If the input stream is arriving in an independently and identically distributed manner with true α-quantile equal to $Q(\alpha)$, then p from Eqn. (6) will be on the order of $2n$. If the input stream is biased, then p will be very small. So for a fixed false rejection probability θ

$$\text{Reject} = p \leq \theta \tag{7}$$

To simplify this computation, a user can choose the update interval for p to be as long as they wish. For the example in Figure 6 p is computed once per thousand observations. This has little effect other than to make the time until detection last until the end of an interval—recall that $\hat{Q}(\alpha)$ is updated continuously—and allows us to take a leisurely approach to updating p using whatever method is most practical. In practice, a user can numerically compute the values of $|\hat{Q}_t(\alpha_j) - Q(\alpha_j)|$ for which Eqn. (6) equals their chosen θ. For example, minimizing

$$J(z_j) = \left(2n \left| \left(\frac{\alpha_j}{z_j} \right)^{z_j} \left(\frac{1-\alpha_j}{1-z_j} \right)^{1-z_j} \right|^t - \theta \right)^2 \tag{8}$$

with respect to $|\hat{Q}(\alpha_j) - Q(\alpha_j)|$ will yield a threshold for the absolute difference instead, and simplify computation at each interval. The values of $\hat{Q}(\alpha_j)$ can be reset after

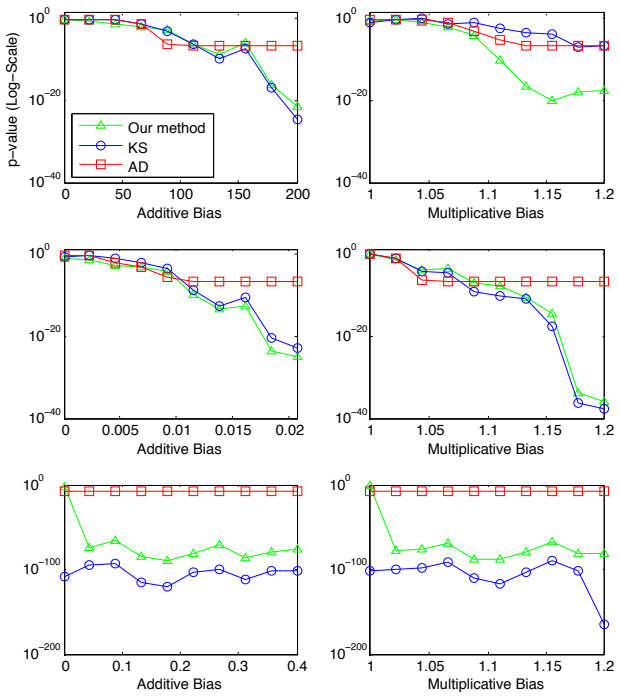

Figure 2: A comparison of p-values from Anderson-Darling (AD) and Kolmogorov-Smirnov (KS) tests against those of our statistic (see Eqn. (6)). Each row are results given a different initial probability distribution, points are p-values at varying degrees of bias. The top row is from a normal distribution with $\mu = 0$ and $\sigma = 1,000$, second is a beta distribution with parameters $(a, b) = (2, 10)$, and the bottom row is a Poisson distribution with parameter 4. Each point is a p-value computed given a sequence of three thousand observations with $n = 6$ and $\lambda = 10^{-4}$. Note that the AD test for p-values $< 6 \times 10^{-9}$ are thresholded to 6×10^{-9} due to the excessive computation necessary to determine smaller p-values.

a reasonable number of intervals to avoid storing many of these thresholds for different t.

Because the Chernoff bound assumes independence, and the movements of $\hat{Q}(\alpha)$ are dependent on the underlying CDF, Eqn. (6) is imprecise. In order to compute the precise probability of a deviation $\hat{Q}(\alpha)$ under the null hypothesis we would need to compute the probability at every step. However, if λ is small enough such that a λ-step of $\hat{Q}(\alpha)$ does not appreciably change the probability of $\hat{Q}(\alpha)$ increasing or decreasing, then Eqn. (6) holds quite well. "Small enough" will depend on the probability distribution under test, but there is no λ too small for this application, barring obvious numerical issues.

We can see in Figure 2 that the statistic closely follows the well-known Kolmogorov-Smirnov (KS) and Anderson-Darling (AD) test statistics with varying bias for several distributions. Note also that neither the KS or AD test has a trivial extension to a streaming environment. Additionally, our test is sensitive to autocorrelation—an issue that is not possible to detect with either a KS or AD test.

4. IMPLEMENTATION

4.1 A $\hat{Q}(\alpha)$ Functional Unit

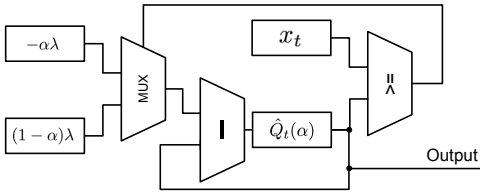

Figure 3: A hardware estimator for a single $\hat{Q}(\alpha)$

A $\hat{Q}(\alpha)$ unit with $n = 1$ leads to the hardware architecture depicted in the block diagram of Figure 3. All $\hat{Q}(\alpha)$ hardware units are independent except for the read-only value x. On account of this, increasing n, and hence the number of α values, creates a group of nearly identical $\hat{Q}(\alpha)$ units. We implemented our designs using Xilinx Vivado High Level Synthesis (HLS) targeting a Zynq-7020 SoC.

Referring to the HLS code for this operation, shown in Figure 4, we initialize $\lambda(1-\alpha)$ and $-\lambda\alpha$—in Figure 3 these are arrays *pos* and *neg* respectively—only once before use, and these values remain constant throughout. In practice this loop may be unrolled manually or using #pragma HLS UNROLL to minimize impact on latency. Also note that the ternary statement is necessary to ensure that the multiplexer is generated as desired. FPGA area and performance estimates are listed in Table 1.

Table 1: Area and Performance Estimates for a Single $\hat{Q}(\alpha)$ Unit. Latency (Lat) is in cycles, Throughput (Tp) in MHz

	Lat	Tp	FF	LUT	DSP
8-bit int	2	164	33	67	0
32-bit int	2	164	129	267	0
32-bit float	7	143	400	867	2

Users should make sure to select matching data types for both $\hat{Q}(\alpha)$ and the arrays *pos* and *neg* as shown in Figure 4. If these data types do not match, the user (or tool) will have to insert a (relatively) expensive conversion to the highest precision type, which is often floating point, for the subtraction. This is almost never beneficial or necessary. The most common case, where $\hat{Q}(\alpha)$ is integer and $\lambda\alpha$ is not, can be easily solved by rounding $\lambda\alpha$ and interpolating to find something near the true value. If $\lambda\alpha$ loses an unacceptable amount of precision by doing this, it is very likely that the domain of the distribution of interest is small enough to require a fixed or floating point version. Now is a good time to reenforce that the resulting $\hat{Q}(\alpha)$ *estimate* can be as much as λ away from $Q(\alpha)$ at any time when the input is completely independent in time, which would be the ideal case.

While Algorithm 1 can be implemented efficiently in hardware with relative ease, Algorithm 2 is somewhat more chal-

```
for (int j = 0; j < n; ++j)
  Q_hat[j] -= (Q_hat[j] > x) ? pos[j] : neg[j];
```

Figure 4: HLS Code for Algorithm 1. See discussion in section 4.1.

Algorithm 3 An Approximation of Algorithm 2

Input: $\vec{\alpha}$, $\hat{Q}_0$, λ, ζ
Result: $\vec{\alpha} \approx F_X$
$t \leftarrow 0$
while x_t **exists**
  for $1 < j < n$

$$b \leftarrow \begin{cases} \lambda & \text{if } \hat{Q}_{t-1}(\alpha_j) \geq x_t \\ 0 & \text{otherwise} \end{cases}$$

$$g \leftarrow \begin{cases} \zeta & \text{if } \Delta^2[\hat{Q}_t](\alpha_j) \geq 0 \\ -\zeta & \text{otherwise} \end{cases}$$

$\hat{Q}_t(\alpha_j) \leftarrow \hat{Q}_{t-1}(\alpha_j) - (b - [\lambda\alpha]_j)$
$[\lambda\alpha]_j \leftarrow [\lambda\alpha]_j + g$
  end for
  $t \leftarrow t + 1$
end while

Table 2: Comparison with Previous Work [4]

	This work*	[4] all 8 tests
FF	774	Sum of 8 = 519
LUT	402	Sum of 8 = 934
Throughput (MHz)	164	Min of 8 = 132

* The architecture shown in Figure 3 with $n = 6$ replications.

lenging. Our adaptations are described in the next section and result in Algorithm 3.

4.2 A Hardware Friendly Approximation

Algorithm 3 is an approximation of Algorithm 2. To make Algorithm 3 hardware amenable, we eliminate multiplications completely, and replace the very small $\zeta\Delta^2[\hat{Q}_t]$ values with small constants. Algorithm 3 updates $\lambda\alpha$ at every time step instead of α, and so in in order to obtain the α values as viable points on the CDF in question, α must be divided by λ. If a group of α values are needed at every time step, then this division might be prohibitive, but if it is possible for us to choose λ to be a power of two, then these concerns are obviated for the most part. Replacing $\zeta\Delta^2[\hat{Q}_t]$ with ζ has the asymptotic effect of adding an error of at most ζ to the output.

The hardware block diagram in figure 5 shows a three-point CDF estimator. α of the two extreme end points are fixed, and so we reuse the $\hat{Q}(\alpha)$ functional units as shown in Figure 3 for them. Like the $\hat{Q}(\alpha)$ units, the area of this architecture scales as $O(n)$, as all elements shown in the figure must be duplicated for each additional α, minus one subtract unit and the end-point $\hat{Q}(\alpha)$ units. For area and performance results for a an implementation when $n = 3$ see Table 3

We compare our implementation to the closest prior work [4]—an implementation of approximations of eight tests from the NIST Statistical Test Suite—in Table 2. We have made an effort to make the comparison fair by working under the assumption that all eight of these tests are implemented simultaneously on the same FPGA, and (as stated in [4]) are not sharing resources. So we compare to the sum of the area consumption of the eight tests from [4]. The throughput of the test supporting the slowest clock will dictate the rate at which other tests, and the RNG under test, can be run. Thus we take the minimal throughput from [4] for our comparison. For our work, we select a 32-bit implementation of $\hat{Q}(\alpha)$ with $n = 6$ replications with differing αs. This is not the most hardware efficient implementation we have proposed, but it is the most fair comparison.

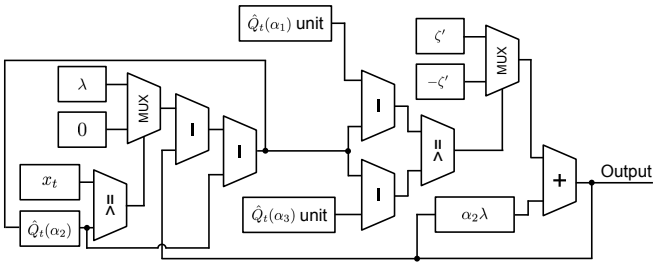

Figure 5: A hardware architecture for CDF estimation at three points.

Table 3: Area and Performance for a 3-point CDF Estimator. Latency (Lat) is in cycles, Throughput (Tp) in MHz

	Lat	Tp	FF	LUT	DSP
32-bit float	25	117	1,030	2,107	4

5. EXPERIMENTAL RESULTS

We compare three of the PRNGs mentioned in [6] in our experiment (see Table 4). LFSR and BigMod are known to be biased. LFSR over-represents small values, while BigMod has a slightly "lumpy" histogram. Note that we have verified that neither of these passes the NIST Statistical Test Suite (STS), (0/15 for LFSR, 1/15 for BigMod), while Twister passes 14/15 of these tests with test parameters set to the defaults. For our STS tests we used 550 groups of 16,000 bits each, making 8.8×10^6 bits in all.

5.1 Inversion Sampling Bias Test

For our example, we test the three PRNGs listed in Table 4. Our null hypothesis is that these PRNGs produce a uniformly random sequence of integers in the range $[1, 2^{31} - 1]$. Most importantly, we do not test the outputs x of the PRNGs directly, instead, we post-process each, to make x', using inversion sampling according to the following:

$$x' = \hat{F}_X^{-1}(x/(2^{31} - 1)) \cdot (2^{31} - 1) \quad (9)$$

where $\hat{F}_X^{-1}$ is an fine grained approximation to the quantile function of a standard normal distribution and we set $Q(\alpha)$—our null model—accordingly. $\hat{F}_X^{-1}$ is approximate in that we bin the $2^{31} - 1$ unique values from the PRNGs into $22,734$ bins equally spaced over $[-1 \times 10^{10}, 1 \times 10^{10}]$.

Table 4: PRNGs Scored in Figure 6.

PRNG	Implementation
Twister	Mersenne Twister [20] generating 31 bit integers
LFSR	$x \leftarrow x^{31} + x^{28} + 1$
BigMod	$x \leftarrow 1583458089 \cdot x \mod (2^{31} - 1)$

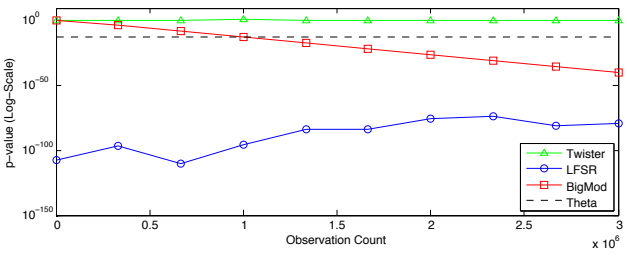

Figure 6: A comparison of p-values over a sequence of three million observations from the PRNGs in Table 4, with $n = 6$. θ (dashed line) is a constant threshold computed for a false positive rate equal to 2^{-40}. See section 5.1 for discussion.

p-values computed at intervals of 10^3 for $n = 6$ during three million observations are shown in Figure 6. See equation (6) for the details of p-value computation.

Using the rather relaxed $\theta = 2^{-40}$, we reject the null hypothesis the first time the test statistic crosses the reject threshold θ. We can see from Figure 6 that even a significantly larger θ would not reject Twister, but would reject the others considerably sooner.

6. CONCLUSION

In this work we have developed an algorithm for learning values, $\hat{Q}(\alpha)$, and probabilities, α, of an unknown CDF, and included modifications to ease hardware implementation. We demonstrated the usefulness of the technique for uncovering bias in sources of randomness when considering the entire random number generation pipeline as opposed to only the RNG or PRNG. We have also shown that our work compares favorably in terms of both area consumption and throughput with previous less flexible approaches to run-time bias detection.

A notable drawback of our algorithm is that in order to function optimally it requires the stream under analysis to generate variates in a time independent manner. While this is ideal for testing RNGs, it will give suboptimal results for ordered data. In our future efforts we will work to mitigate this algorithmic deficiency, and apply our approach to more general run-time on-chip statistical monitoring.

7. ACKNOWLEDGEMENTS

The authors would like to thank the reviewers for their valuable feedback which improved the final version of this paper. This work was supported by the NSF under Grants CNS-1563767 and CNS-1527631.

8. REFERENCES

[1] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Effective computation of biased quantiles over data streams," in *21st International Conference on Data Engineering (ICDE'05)*. IEEE, 2005, pp. 20–31.

[2] B. Yang, V. Rožić, N. Mentens, and I. Verbauwhede, "On-the-fly tests for non-ideal true random number generators," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2015, pp. 2017–2020.

[3] B. Yang, V. Ro, N. Mentens, W. Dehaene, I. Verbauwhede *et al.*, "Total: Trng on-the-fly testing for attack detection using lightweight hardware," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 127–132.

[4] F. Veljković, V. Rožić, and I. Verbauwhede, "Low-cost implementations of on-the-fly tests for random number generators," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012, pp. 959–964.

[5] A. Vaskova, C. Lopez-Ongil, A. Jimenez-Horas, E. San Millan, and L. Entrena, "Robust cryptographic ciphers with on-line statistical properties validation," in *2010 IEEE 16th International On-Line Testing Symposium*.

[6] A. Tisserand, "Circuits for true random number generation with on-line quality monitoring," in *Claude Shannon Institut Workshop on Coding and Cryptography*, 2011.

[7] D. Hotoleanu, O. Cret, A. Suciu, T. Gyorfi, and L. Vacariu, "Real-time testing of true random number generators through dynamic reconfiguration," in *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*. IEEE, 2010, pp. 247–250.

[8] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," DTIC Document, Tech. Rep., 2001.

[9] L. Devroye, "Sample-based non-uniform random variate generation," in *Proceedings of the 18th conference on Winter simulation*. ACM, 1986, pp. 260–265.

[10] P. Ducklin. (2013) Anatomy of a pseudorandom number generator - visualising cryptocat's buggy prng. [Online]. Available: https://goo.gl/UF1BGk

[11] T. W. Anderson and D. A. Darling, "A test of goodness of fit," *Journal of the American statistical association*, vol. 49, no. 268, pp. 765–769, 1954.

[12] O. H. Amman, T. von Kármán, and G. B. Woodruff, "The failure of the tacoma narrows bridge," 1941.

[13] C. A. Hoare, "Algorithm 65: find," *Communications of the ACM*, vol. 4, no. 7, pp. 321–322, 1961.

[14] M. Greenwald and S. Khanna, "Space-efficient online computation of quantile summaries," in *ACM SIGMOD Record*, vol. 30, no. 2. ACM, 2001, pp. 58–66.

[15] D. Felber and R. Ostrovsky, "A randomized online quantile summary in o($1/\varepsilon$ log $1/\varepsilon$) words," *CoRR*, vol. abs/1503.01156, 2015. [Online]. Available: http://arxiv.org/abs/1503.01156

[16] Z. S. Karnin, K. Lang, and E. Liberty, "Almost optimal streaming quantiles algorithms," *CoRR*, vol. abs/1603.05346, 2016. [Online]. Available: http://arxiv.org/abs/1603.05346

[17] L. Tierney, "A space-efficient recursive procedure for estimating a quantile of an unknown distribution," *SIAM Journal on Scientific and Statistical Computing*, vol. 4, no. 4, pp. 706–711, 1983.

[18] J. Kim, W. B. Powell, and R. A. Collado, "Quantile optimization for heavy-tailed distribution using asymmetric signum functions," *Princeton University*, 2011.

[19] M. S. Turan, E. Barker, J. Kelsey, K. McKay, M. Baish, and M. Boyle, "Recommendation for the entropy sources used for random bit generation," *NIST Special Publication 800-90B (2nd Draft)*, 2016.

[20] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.