

# Testing the Randomness of Cryptographic Function Mappings

Alan Kaminsky

ark@cs.rit.edu

Rochester Institute of Technology, Rochester, NY, USA

January 23, 2019

**Abstract.** A cryptographic function with a fixed-length output, such as a block cipher, hash function, or message authentication code (MAC), should behave as a random mapping. The mapping’s randomness can be evaluated with statistical tests. Statistical test suites typically used to evaluate cryptographic functions, such as the NIST test suite, are not well-suited for testing fixed-output-length cryptographic functions. Also, these test suites employ a frequentist approach, making it difficult to obtain an overall evaluation of the mapping’s randomness. This paper describes *CryptoStat*, a test suite that overcomes the aforementioned deficiencies. *CryptoStat* is specifically designed to test the mappings of fixed-output-length cryptographic functions, and *CryptoStat* employs a Bayesian approach that quite naturally yields an overall evaluation of the mappings’ randomness. Results of applying *CryptoStat* to reduced-round and full-round versions of the AES block ciphers and the SHA-1 and SHA-2 hash functions are reported; the results are analyzed to determine the algorithms’ randomness margins.

**Keywords:** Statistical tests, Bayesian model selection, AES block cipher, SHA-1 hash function, SHA-2 hash function

## 1 Introduction

A cryptographic function, loosely speaking, is supposed to produce random-looking outputs. The outputs’ randomness can be evaluated using a pseudorandom number generator (PRNG) statistical test suite, which applies a series of statistical tests to the outputs. Widely used statistical test suites include Diehard [16], Dieharder [3], ENT [24], TestU01 [15], and the NIST test suite [19, 21].

Although the NIST test suite’s title asserts that it is “for cryptographic applications,” the NIST test suite is in fact a general-purpose statistical test suite for evaluating the randomness of *binary sequences*—in contrast to statistical tests on sequences of uniformly distributed integers or real numbers, as are typically found in other test suites. The NIST test suite does not care where the binary sequences come from, whether from a cryptographic function or some other source. The official document [19] just describes the 15 statistical tests in the test suite and discusses how to interpret the test results.

In practice, two problems arise when using the NIST test suite (or any of the popular statistical test suites) to evaluate a cryptographic function that produces a fixed-length output, such as a block cipher, hash function, message authentication code (MAC), or sponge bijection. For brevity, henceforth “block cipher” will refer to other fixed-output-length cryptographic functions as well.

The first problem stems from the fact that a block cipher is not itself a PRNG; it does not generate an *arbitrarily-long* binary sequence. Rather, the block cipher takes a fixed-length plaintext plus a fixed-length key and maps these to a *fixed-length* ciphertext. The mapping is supposed to be random, in two senses: a given key should map different plaintexts to ciphertexts that look as though chosen at random, and different keys should map a given plaintext also to ciphertexts that look as though chosen at random.

To apply the NIST test suite to a block cipher, the block cipher must be made to behave like a PRNG and generate (typically long) binary sequences. The official document [19] does not specify how to do this. NIST did publish a document [22] describing how NIST made the AES finalist candidate block ciphers generate long binary sequences. One technique was to encrypt a long string of 0s using the block cipher in cipher block chaining (CBC) mode with various randomly chosen keys, and to apply the test suite to the resulting ciphertext strings. Another technique was similar, except a long string of mostly 0s with a few 1s was encrypted using the block cipher in electronic codebook (ECB) mode. Several other techniques were used as well; see [22].

However, when the block cipher is turned into a PRNG and the NIST (or other) test suite is applied, it is the PRNG’s randomness that is being tested, not the randomness of the block cipher mapping itself. If a nonrandom bias is detected in the output, it is not clear whether the bias is inherent in the block cipher or is an artifact of the mode in which the block cipher is employed.

For example, a block cipher operated in output feedback (OFB) mode is likely to experience a keystream block collision and start repeating the keystream after encrypting about  $2^{n/2}$  blocks, where  $n$  is the block size (due to the birthday paradox). This does not happen when the block cipher is operated in, say, counter mode. Thus, the slight bias in the OFB keystream arises from the use of OFB mode, not from the block cipher itself; and this bias would be present even if the block cipher mapping was perfectly random. Note that encrypting a string of 0s with a block cipher in CBC mode is identical to generating a keystream with a block cipher in OFB mode; thus, at least one of NIST’s statistical tests on the AES finalist candidates had a slight inherent bias due to the block cipher mode.

As another example, El-Fotouh and Diepold [6] found that AES would fail some of the tests in the NIST test suite when operated in certain modes, but would pass all the tests when operated in other modes. Again, this implies that the bias is due to the mode of operation, not the block cipher mapping.

To summarize, the NIST test suite is not well suited for evaluating block ciphers (or other fixed-output-length cryptographic functions). It would be better to use statistical tests that evaluated the randomness of the block cipher mapping directly, without needing to turn the block cipher into a PRNG.

The second problem that arises in practice stems from the fact that the NIST test suite, like most test suites, takes a *frequentist* approach to statistical testing. That is, from the binary sequence being analyzed, each test calculates a *statistic* and a *p-value*. The *p-value* is the probability that the observed value of the statistic would occur by chance, even if the binary sequence is actually random. If the *p-value* falls below a *significance* threshold, the test *fails* (and indicates nonrandomness), otherwise the test *passes*.

The NIST test suite consists of 15 statistical tests, some of which have multiple variations. Each test and variation is applied to multiple binary sequences produced by the PRNG (block cipher) under test, yielding perhaps several hundred *p-values*. Usually, some of these tests fail. Does this mean the PRNG is nonrandom? Not necessarily. If the PRNG is random, the *p-values* should be uniformly distributed between 0 and 1, and we would expect a certain fraction of them to fall below the significance threshold. So the NIST test suite applies a “second-level” statistical test to the *p-values* for each “first-level” test to determine whether the test’s *p-values* are uniformly distributed. Each second-level test yields a second-level *p-value*, which might pass or fail. From this morass of first- and second-level *p-values*, the analyst must decide whether the binary sequences, and the block cipher that produced them, are random or nonrandom.

The frequentist approach does not specify a procedure for combining multiple *p-values* into a single number that yields an overall random/nonrandom decision. As an alternative, a *Bayesian* testing methodology *does* specify how to combine multiple test results into a single number, the *posterior odds ratio*, which in turn determines whether the block cipher mapping is random or nonrandom. However, none of the widely used statistical test suites takes a Bayesian approach.

This paper’s novel contributions are threefold:

- *Odds ratio tests* are described. The odds ratio tests use the methodology of Bayesian model selection to decide whether a sequence of integers obeys a specified discrete distribution. Multiple individual odds ratio test results are easily aggregated together to yield a single overall test result.
- *CryptoStat*, a randomness test suite for cryptographic functions, is described. *CryptoStat* evaluates a function’s input-to-output mapping directly, without needing to turn the function into a PRNG. *CryptoStat* performs multiple odds ratio tests and aggregates the results to yield a clear random/nonrandom decision.
- *CryptoStat* is applied to reduced-round and full-round versions of the AES block cipher variations and the SHA-1 and SHA-2 hash function variations, and the results are reported. The results reveal each algorithm’s *randomness margin*—the number of rounds for which the algorithm produces random outputs, as a fraction of the full number of rounds.

The rest of the paper is organized as follows. Section 2 summarizes related work. Section 3 reviews the mathematics of Bayesian inference and describes the odds ratio tests. Section 4 describes the *CryptoStat* test suite and the massively parallel programs that speed up *CryptoStat*’s calculations; the programs are

designed to run on the many cores of a graphics processing unit (GPU). Section 5 lists and analyzes CryptoStat’s results for AES, SHA-1, and SHA-2.

## 2 Related Work

Several authors have studied the randomness of block ciphers by turning the block cipher into a PRNG and applying standard statistical tests to the PRNG. As already mentioned, NIST turned the AES finalist candidate block ciphers into PRNGs and applied the NIST test suite [22]. Hellekalek and Wegenkittl [8] turned AES into a PRNG and applied several tests from the NIST test suite as well as a new test, the gambling test, to AES. The designers of the HIGHT [9] and TWIS [17] lightweight block ciphers turned their ciphers into PRNGs and applied the NIST test suite. Sulak, Doğanaksoy, Ege, and Koçak [23] treated a block cipher as a PRNG that generates *short* binary sequences (128–256 bits); they modified seven of the 15 statistical tests in the NIST test suite to operate on such short binary sequences, and they applied the modified tests to the AES finalist candidate block ciphers. The technique has also been used on cryptographic functions other than block ciphers. Wang and Zhang turned the SHA-3 candidate BLAKE hash function into a PRNG and applied the NIST test suite [25].

Some authors (such as [9] and [17]) merely make a vague statement to the effect that “We applied the NIST test suite, and our cipher passed,” leaving the impression that the NIST test suite yields an unequivocal yes/no, random/nonrandom decision about the block cipher under test. In fact, the NIST test suite does not yield such a decision (nor do similar test suites). The authors have not reported the detailed test results and the nuanced analysis of those results that led them to conclude that their ciphers “passed.”

CryptoStat analyzes the randomness of a cryptographic function’s mapping directly, without treating the function as a PRNG. Other authors have used this approach as well. Filiol [7] defined a statistical test based on comparing a cryptographic function’s algebraic normal form to that of a random Boolean function, and applied the test to the DES and AES block ciphers as well as several stream ciphers and hash functions. Katos [13] defined a statistical test to measure the diffusion of the block cipher’s mapping, but did not apply the test to any actual block ciphers. Doğanaksoy, Ege, Koçak, and Sulak [4] defined four statistical tests based on the block cipher mapping—strict avalanche criterion test, linear span test, collision test, and coverage test—and applied these tests to the AES finalist candidate block ciphers. The same authors also applied the methodologies of [23] and [4] to the compression functions of the SHA-3 second-round candidate hash functions [5].

All of the above works used a frequentist methodology (based on  $p$ -values) to analyze the statistical test results. To my knowledge, CryptoStat is the first to use a Bayesian methodology (based on odds ratios). The Bayesian methodology quite naturally combines the results of individual tests into an overall odds ratio,

yielding an unequivocal decision about whether the reduced-round or full-round cryptographic function is random or nonrandom (refer to Section 3.4).

### 3 Odds Ratio Tests

Odds ratio tests are an alternative to frequentist statistical tests such as the chi-square test. A strong point of odds ratio tests is that the results of multiple independent tests can easily be *aggregated* to yield a single overall result. Odds ratio tests use the methodology of *Bayesian model selection* applied to binomial distributions. For more information about Bayesian model selection, see [12].

#### 3.1 Bayes Factors and Odds Ratios

Let  $H$  denote a *hypothesis*, or *model*, describing some process. Let  $D$  denote an experimental *data sample*, or just *sample*, observed by running the process. Let  $\text{pr}(H)$  be the probability of the model. Let  $\text{pr}(D|H)$  be the conditional probability of the sample given the model. Let  $\text{pr}(D)$  be the probability of the sample, apart from any particular model. Bayes's Theorem states that  $\text{pr}(H|D)$ , the conditional probability of the model given the sample, is

$$\text{pr}(H|D) = \frac{\text{pr}(D|H) \text{pr}(H)}{\text{pr}(D)}. \quad (1)$$

Suppose there are two alternative models  $H_1$  and  $H_2$  that could describe a process. After observing sample  $D$ , the *posterior odds ratio* of the two models,  $\text{pr}(H_1|D)/\text{pr}(H_2|D)$ , is calculated from Equation (1) as

$$\frac{\text{pr}(H_1|D)}{\text{pr}(H_2|D)} = \frac{\text{pr}(D|H_1)}{\text{pr}(D|H_2)} \cdot \frac{\text{pr}(H_1)}{\text{pr}(H_2)},$$

where the term  $\text{pr}(H_1)/\text{pr}(H_2)$  is the *prior odds ratio* of the two models, and the term  $\text{pr}(D|H_1)/\text{pr}(D|H_2)$  is the *Bayes factor*. The odds ratio represents one's belief about the relative probabilities of the two models. Given one's initial belief before observing any samples (the prior odds ratio), the Bayes factor is used to *update* one's belief after performing an experiment and observing a sample (the posterior odds ratio). Stated simply, posterior odds ratio = Bayes factor  $\times$  prior odds ratio.

Suppose two experiments are performed and two samples,  $D_1$  and  $D_2$ , are observed. Assuming the samples are independent, it is straightforward to calculate that the posterior odds ratio based on both samples is

$$\begin{aligned} \frac{\text{pr}(H_1|D_2, D_1)}{\text{pr}(H_2|D_2, D_1)} &= \frac{\text{pr}(D_2|H_1)}{\text{pr}(D_2|H_2)} \cdot \frac{\text{pr}(H_1|D_1)}{\text{pr}(H_2|D_1)} \\ &= \frac{\text{pr}(D_2|H_1)}{\text{pr}(D_2|H_2)} \cdot \frac{\text{pr}(D_1|H_1)}{\text{pr}(D_1|H_2)} \cdot \frac{\text{pr}(H_1)}{\text{pr}(H_2)}. \end{aligned} \quad (2)$$

In other words, the posterior odds ratio for the first experiment becomes the prior odds ratio for the second experiment. Equation (2) can be extended to any number of independent samples  $D_i$ ; the final posterior odds ratio is just the initial prior odds ratio multiplied by all the samples' Bayes factors.

*Model selection* is the problem of deciding which model,  $H_1$  or  $H_2$ , is better supported by a series of one or more samples  $D_i$ . In the Bayesian framework, this is determined by the posterior odds ratio (2). Henceforth, "odds ratio" will mean the posterior odds ratio. If the odds ratio is greater than 1, then  $H_1$ 's probability is greater than  $H_2$ 's probability, given the data; that is, the data supports  $H_1$  better than it supports  $H_2$ . The larger the odds ratio, the higher the degree of support. An odds ratio of 100 or more is generally considered to indicate decisive support for  $H_1$  [12]. If on the other hand the odds ratio is less than 1, then the data supports  $H_2$  rather than  $H_1$ , and an odds ratio of 0.01 or less indicates decisive support for  $H_2$ .

### 3.2 Models with Parameters

In the preceding formulas, the models had no free parameters. Now suppose that model  $H_1$  has a parameter  $\theta_1$  and model  $H_2$  has a parameter  $\theta_2$ . Then the conditional probabilities of the samples given each of the models are obtained by integrating over the possible parameter values [12]:

$$\text{pr}(D|H_1) = \int \text{pr}(D|\theta_1, H_1) \pi(\theta_1|H_1) d\theta_1 \quad (3)$$

$$\text{pr}(D|H_2) = \int \text{pr}(D|\theta_2, H_2) \pi(\theta_2|H_2) d\theta_2 \quad (4)$$

where  $\text{pr}(D|\theta_1, H_1)$  is the probability of observing the sample under model  $H_1$  with the parameter value  $\theta_1$ ,  $\pi(\theta_1|H_1)$  is the prior probability density of  $\theta_1$  under model  $H_1$ , and likewise for  $H_2$  and  $\theta_2$ . The Bayes factor is then the ratio of these two integrals.

### 3.3 Odds Ratios for Binomial Models

Suppose an experiment performs  $n$  Bernoulli trials, where the probability of success is  $\theta$ , and counts the number of successes  $k$ , which obeys a binomial distribution. The values  $n$  and  $k$  constitute the sample  $D$ . With this as the model  $H$ , the probability of  $D$  given  $H$  with parameter  $\theta$  is

$$\text{pr}(D|\theta, H) = \binom{n}{k} \theta^k (1 - \theta)^{n-k} = \frac{n!}{k! (n-k)!} \theta^k (1 - \theta)^{n-k} .$$

Consider the odds ratio for two particular binomial models,  $H_1$  and  $H_2$ .  $H_1$  is that the Bernoulli success probability  $\theta_1$  is a certain value  $p$ , the value that the success probability is "supposed" to have. Then the prior probability density

of  $\theta_1$  is a delta function,  $\pi(\theta_1|H_1) = \delta(\theta_1 - p)$ , and the Bayes factor numerator (3) becomes

$$\text{pr}(D|H_1) = \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k} .$$

$H_2$  is that the Bernoulli success probability  $\theta_2$  is some unknown value between 0 and 1, not necessarily the value it is “supposed” to have. The prior probability density of  $\theta_2$  is taken to be a uniform distribution:  $\pi(\theta_2|H_2) = 1$  for  $0 \leq \theta_2 \leq 1$  and  $\pi(\theta_2|H_2) = 0$  otherwise. The Bayes factor denominator (4) becomes

$$\text{pr}(D|H_2) = \int_0^1 \frac{n!}{k!(n-k)!} \theta_2^k (1-\theta_2)^{n-k} d\theta_2 = \frac{1}{n+1} .$$

Putting everything together, the Bayes factor for the two binomial models is

$$\frac{\text{pr}(D|H_1)}{\text{pr}(D|H_2)} = \frac{(n+1)!}{k!(n-k)!} p^k (1-p)^{n-k} .$$

Substituting the gamma function for the factorial,  $n! = \Gamma(n+1)$ , gives

$$\frac{\text{pr}(D|H_1)}{\text{pr}(D|H_2)} = \frac{\Gamma(n+2)}{\Gamma(k+1)\Gamma(n-k+1)} p^k (1-p)^{n-k} .$$

Because the gamma function’s value typically overflows the range of floating point values in a computer program, we compute the logarithm of the Bayes factor instead of the Bayes factor itself:

$$\begin{aligned} \log \frac{\text{pr}(D|H_1)}{\text{pr}(D|H_2)} &= \log \Gamma(n+2) - \log \Gamma(k+1) - \log \Gamma(n-k+1) \\ &\quad + k \log p + (n-k) \log(1-p) . \end{aligned} \quad (5)$$

The log-gamma function can be computed efficiently (see [18] page 256), and mathematical software libraries usually include log-gamma.

### 3.4 Odds Ratio Test

The preceding experiment can be viewed as a *test* of whether  $H_1$  is true, that is, whether the success probability is  $p$ . The log (posterior) odds ratio of the models  $H_1$  and  $H_2$  is the log prior odds ratio plus the log Bayes factor (5). Assuming that  $H_1$  and  $H_2$  are equally probable at the start, the log odds ratio is just the log Bayes factor. The test *passes* if the log odds ratio is greater than 0, otherwise the test *fails*.

When multiple independent runs of the preceding experiment are performed, the overall log odds ratio is the sum of all the log Bayes factors. In this way, one can *aggregate* the results of a series of individual tests, yielding an overall odds ratio test. Again, the aggregate test passes if the overall log odds ratio is greater than 0, otherwise the aggregate test fails.

Note that the process of aggregating the individual test results automatically deals with the situation where most of the individual tests passed but some of the individual tests failed (or vice versa). As long as the overall log odds ratio is positive, the aggregate test passes, despite the existence of individual test failures. The analyst need only consider the aggregate result, not the individual results.

Note also that the odds ratio test is not a frequentist statistical test that is attempting to disprove some null hypothesis. The odds ratio test is just a particular way to decide how likely or unlikely it was that a series of observations came from a Bernoulli( $p$ ) distribution, by calculating a posterior odds ratio. While a frequentist statistical test could be defined based on odds ratios, I am not doing that here.

### 3.5 Odds Ratio Test for a Discrete Distribution

Consider a discrete random variable  $X$ . The variable has  $B$  different possible values (“bins”),  $0 \leq x \leq B - 1$ . Let  $\text{pr}(x)$  be the probability of bin  $x$ ; then the variable’s cumulative distribution function is

$$F(x) = \sum_{i=0}^x \text{pr}(i) , \quad 0 \leq x \leq B - 1 . \quad (6)$$

Suppose an experiment with  $n$  trials is performed. In each trial, the random variable’s value is observed, and a counter for corresponding bin is incremented. If the variable obeys the distribution (6), the counter for bin  $x$  should end up at approximately  $n \cdot \text{pr}(x)$ .

An odds ratio test for the random variable calculates the odds ratio of two hypotheses:  $H_1$ , that  $X$  obeys the discrete distribution with cumulative distribution function (6), and  $H_2$ , that  $X$  does not obey this distribution. First calculate the observed cumulative distribution of  $X$ :

$$F_{\text{obs}}(x) = \frac{1}{n} \sum_{i=0}^x \text{counter}[i] , \quad 0 \leq x \leq B - 1 .$$

Let  $y$  be the bin such that the absolute difference  $|F(y) - F_{\text{obs}}(y)|$  is maximized. (This is similar to what is done in a Kolmogorov-Smirnov test for a *continuous* distribution.) The trials are now viewed as Bernoulli trials, where incrementing a bin less than or equal to  $y$  is a success, the observed number of successes in  $n$  trials is  $k = n \cdot F_{\text{obs}}(y)$ , and the success probability if  $H_1$  is true is  $p = F(y)$ . An odds ratio test for the discrete distribution ( $H_1$  versus  $H_2$ ) is therefore equivalent to an odds ratio test for this particular binomial distribution, with Equation (5) giving the log Bayes factor. If the log Bayes factor is greater than 0, then the observed distribution is close enough to the expected distribution that  $H_1$  is more likely than  $H_2$ , and the test passes. Otherwise, the observed distribution is far enough away from the expected distribution that  $H_2$  is more likely than  $H_1$ , and the test fails.



**Table 1.** Discrete uniform distribution odds ratio test data; test passes

$x$	counter[ $x$ ]	$F_{\text{obs}}(x)$	$F(x)$	$ F(x) - F_{\text{obs}}(x) $
0	99476	0.099476	0.1	0.000524
1	100498	0.199974	0.2	0.000026
2	99806	0.299780	0.3	0.000220
3	99881	0.399661	0.4	0.000339
4	99840	0.499501	0.5	0.000499
5	99999	0.599500	0.6	0.000500
6	99917	0.699417	0.7	0.000583
7	100165	0.799582	0.8	0.000418
8	100190	0.899772	0.9	0.000228
9	100228	1.000000	1.0	0.000000

**Table 2.** Discrete uniform distribution odds ratio test data; test fails

$x$	counter[ $x$ ]	$F_{\text{obs}}(x)$	$F(x)$	$ F(x) - F_{\text{obs}}(x) $
0	101675	0.101675	0.1	0.001675
1	101555	0.203230	0.2	0.003230
2	100130	0.303360	0.3	0.003360
3	99948	0.403308	0.4	0.003308
4	99754	0.503062	0.5	0.003062
5	99467	0.602529	0.6	0.002529
6	99355	0.701884	0.7	0.001884
7	99504	0.801388	0.8	0.001388
8	99306	0.900694	0.9	0.000694
9	99306	1.000000	1.0	0.000000

A couple of examples will illustrate the odds ratio test for the case of a discrete uniform distribution. I queried a pseudorandom number generator one million times; each value was uniformly distributed in the range 0.0 (inclusive) through 1.0 (exclusive); I multiplied the value by 10 and truncated to an integer, yielding a bin  $x$  uniformly distributed in the range 0 through 9; and I accumulated the values into 10 bins, yielding the data in Table 1. The maximum absolute difference between the observed and expected cumulative distributions occurred at bin 6. With  $n = 1000000$ ,  $k = 699417$ , and  $p = 0.7$ , the log Bayes factor is 5.9596. In other words, the odds are about  $\exp(5.9596) = 387$  to 1 that this data came from a discrete uniform distribution, and the test passes.

I queried a pseudorandom number generator one million times again, but this time I raised each value to the power 1.01 before converting it to a bin. This introduced a slight bias towards smaller bins. I got the data in Table 2. The maximum absolute difference between the observed and expected cumulative distributions occurred at bin 2. With  $n = 1000000$ ,  $k = 303360$ , and  $p = 0.3$ , the log Bayes factor is  $-20.057$ . In other words, the odds are about  $\exp(20.057) =$

514 million to 1 that this data did not come from a discrete uniform distribution, and the test fails.

## 4 CryptoStat Test Suite

CryptoStat<sup>1</sup> is a suite of Java programs, Java classes, and GPU kernels that uses odds ratio tests to analyze the randomness of cryptographic functions. To perform an analysis run, the following items are specified:

- The cryptographic function to be analyzed.
- The series of values for the function’s  $A$  input.
- The series of values for the function’s  $B$  input.
- The series of values to test for randomness, derived from the function’s  $C$  output values.
- The bit groups to test in the test data values.

These items are explained in the subsections below.

### 4.1 Cryptographic Function

In CryptoStat, a cryptographic function  $F$  has two inputs,  $A$  and  $B$ , and one output,  $C = F(A, B)$ . Each input and output is an integer of a fixed bit size. The interpretation of the inputs and output depends on the particular cryptographic function; for example:

- Block cipher:  $A$  = plaintext,  $B$  = encryption key,  $C$  = ciphertext.
- Tweakable block cipher:  $A$  = plaintext,  $B$  = encryption key + tweak,  $C$  = ciphertext.
- Hash function:  $A$  = first message block,  $B$  = second message block,  $C$  = digest.
- Salted hash function:  $A$  = message block,  $B$  = salt,  $C$  = digest.
- MAC:  $A$  = message,  $B$  = authentication key,  $C$  = tag.

If  $F$  has only one input, the input must be split into two pieces  $A$  and  $B$ ; typically there is a natural way to do this, such as splitting a hash function’s one input, the message, into two message blocks. If  $F$  has more than two inputs, some of the inputs must be combined together; such as concatenating the key and tweak inputs of a tweakable block cipher. As will be seen, CryptoStat can specify separate values for different portions of the  $A$  and  $B$  inputs.

CryptoStat can also analyze a cryptographic function with inputs or outputs of unlimited size, by fixing the  $A$ ,  $B$ , or  $C$  bit sizes; for example:

- Stream cipher:  $A$  = message ( $n$  bits),  $B$  = encryption key + nonce,  $C$  = ciphertext ( $n$  bits).

---

<sup>1</sup> Source code and documentation for CryptoStat are available at <https://www.cs.rit.edu/~ark/parallelcrypto/cryptostat/>.

- Authenticated stream cipher:  $A$  = header data ( $h$  bits) + message ( $n$  bits),  
 $B$  = encryption key + nonce,  $C$  = ciphertext ( $n$  bits) + tag.

If  $F$  has more than two outputs, the outputs must be combined together; such as concatenating the ciphertext and tag outputs of an authenticated stream cipher. As will be seen, CryptoStat tests the randomness of various portions of the output separately.

CryptoStat defines every cryptographic function to have two inputs and one output to simplify the design of the test suite and to provide a unified framework for analyzing any cryptographic function, regardless of the actual number of the function's inputs or outputs.

A cryptographic function  $F$  typically consists of multiple rounds. Thus,  $F$  is parameterized by a round number:  $F_r(A, B)$  is the function reduced to  $r$  rounds,  $1 \leq r \leq R$ , where  $R$  is the full number of rounds. CryptoStat analyzes  $F$ 's randomness for all reduced-round versions as well as the full-round version.

The cryptographic function to be analyzed is implemented as a Java class (a subclass of the Function abstract base class) and a GPU kernel. The Java class encapsulates the function's characteristics, such as the  $A$  input bit size, the  $B$  input bit size, the  $C$  output bit size, and the full number of rounds  $R$ . The GPU kernel contains the code to calculate  $C_r = F_r(A, B)$ ,  $1 \leq r \leq R$ . The inputs and outputs are implemented as multiple-precision integers of any (fixed) bit size as required by the cryptographic function.

During an analysis run, CryptoStat constructs a *function object* (an instance of a subclass of the abstract Function class) that implements the desired cryptographic function. CryptoStat uses the function object to evaluate  $F$ , feeding in a series of  $A$  and  $B$  input values and recording the  $C$  output values. The individual function evaluations are executed in parallel on the GPU's many cores, thus speeding up the analysis run.

The CryptoStat distribution includes function classes and GPU kernels for all variations of AES, SHA-1, and SHA-2. The user can define additional function classes and GPU kernels to test other cryptographic functions.

## 4.2 Input Value Series

To perform a CryptoStat analysis run on a cryptographic function  $F$ , a series of values for the  $A$  input must be specified, and a series of values for the  $B$  input must be specified. It is important for the input values to be highly *nonrandom* [2]. If random inputs are applied to the function and the function's outputs are random, the randomness in the outputs might be due to the randomness in the inputs, rather than to the randomness of the function itself. When nonrandom inputs nonetheless yield random outputs, the outputs' randomness must be coming from the randomness of the function's mapping.

CryptoStat lets the user specify three basic kinds of input value sequences, illustrated below in hexadecimal. Each input value's bit size is the bit size of the  $A$  or  $B$  input to which the value will be applied.

- A given number of sequential values starting from 0:  
0000 0001 0002 0003 0004 0005 0006 0007 ...
- A given number of values in a Gray code sequence starting from 0; each value differs from the previous value in one bit position:  
0000 0001 0003 0002 0006 0007 0005 0004 ...
- One-off values starting from 0; each value differs from the first value in one bit position:  
0000 0001 0002 0004 0008 0010 0020 0040 ...

CryptoStat lets the user modify one of these basic input value series in various ways, including:

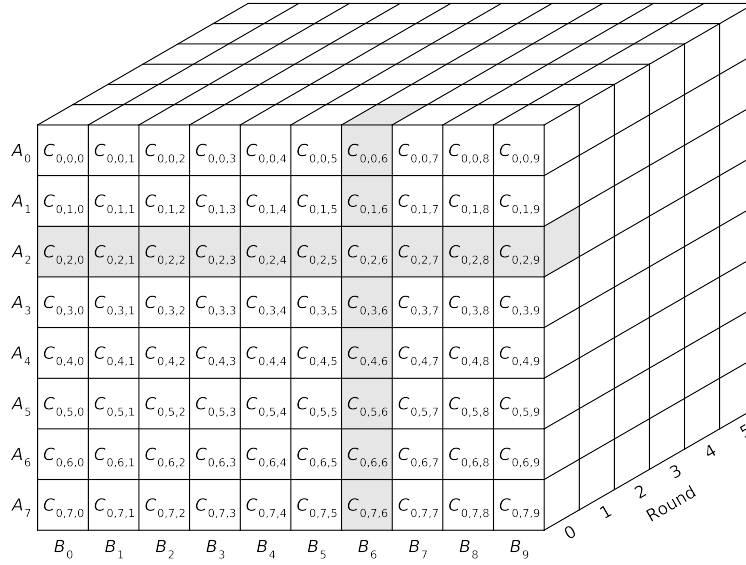
- Add a given constant to each input value; for example, add `abcd` to sequential values:  
`abcd abce abcf abd0 abd1 abd2 abd3 abd4 ...`
- Exclusive-or a given constant into each input value; for example, exclusive-or `5555` into Gray code values:  
`5555 5554 5556 5557 5553 5552 5550 5551 ...`
- Left-shift each input value a given number of bit positions; for example, left-shift Gray code values by 8 bits:  
`0000 0100 0300 0200 0600 0700 0500 0400 ...`
- Complement each input value; for example, complement one-off values:  
`ffff fffe fffd fffb fff7 ffef ffd ffbf ...`

CryptoStat lets the user combine multiple input value series together in various ways, including:

- Concatenate the input values; for example:  
`0000 0001 0002 0003 concatenated with ffff fffe fffd fffc yields`  
`0000 0001 0002 0003 ffff fffe fffd fffc`
- Interleave the input values in a round robin fashion; for example:  
`0000 0001 0002 0003 interleaved with ffff fffe fffd fffc yields`  
`0000 ffff 0001 fffe 0002 fffd 0003 fffc`
- For each combination of values from the input series, exclusive-or the values together; for example:  
`0001 0002 0003 combined with 0400 0500 0600 yields`  
`0401 0402 0403 0501 0502 0503 0601 0602 0603`

Where the *A* or *B* input consists of several portions, such as the key and tweak of a tweakable cipher's *B* input, the last option lets the user specify input values separately for each portion of the input; CryptoStat then automatically generates all combinations of these to feed into the analysis.

To obtain a value series for the *A* input, CryptoStat constructs a *generator object*, then queries the generator object to get the *A* input values. CryptoStat does the same for the *B* input. The *A* and *B* generator objects might or might not be the same. As will be seen, certain kinds of input generators are typically coupled with certain kinds of randomness tests.



**Fig. 1.** Example output value calculation

Each generator object is an instance of a subclass of the Generator abstract base class. The CryptoStat distribution includes generator subclasses for the previously described input series and others. The user can define additional generator subclasses to generate other kinds of input value series.

### 4.3 Output Value Calculation

Given a cryptographic function  $F$  consisting of  $R$  rounds, a series of  $M$  values for the function's  $A$  input, and a series of  $N$  values for the function's  $B$  input, CryptoStat calculates a three-dimensional array of  $C$  output values as

$$C_{r,i,j} = F_r(A_i, B_j), \quad 0 \leq r \leq R - 1, \quad 0 \leq i \leq M - 1, \quad 0 \leq j \leq N - 1.$$

Fig. 1 illustrates the array of  $C$  output values calculated for a function with six rounds, eight  $A$  input values, and ten  $B$  input values. (An actual CryptoStat run would have much longer input series; on the order of 500 to 1000 input values for  $A$  and for  $B$ , say.) The calculations are performed in parallel on the GPU.

Next, CryptoStat groups the individual  $C$  output values into a number of output value series. As will be seen, randomness tests are performed separately on each output value series. For each round  $r$ , there is one output value series consisting of the  $C$  values calculated from the particular value  $A_i$  and all  $B$  values, designated as  $C_{r,i,*}$ . There is also one output value series consisting of the  $C$  values calculated from the particular value  $B_j$  and all  $A$  values, designated as  $C_{r,*,j}$ . Fig. 1 highlights the output value series  $C_{0,2,*}$  and  $C_{0,*,6}$ .

Organizing the output value series this way lets CryptoStat separately analyze the effect of each input on the function’s output. For example, for a block cipher with  $A = \text{plaintext}$ ,  $B = \text{key}$ , and  $C = \text{ciphertext}$ , analyzing the  $C_{r,i,*}$  output series yields insight into the ciphertext’s behavior when the plaintext is held constant and the key is changed. Analyzing the  $C_{r,*,j}$  output series yields insight into the ciphertext’s behavior when the key is held constant and the plaintext is changed.

CryptoStat’s organization of the output value series is inspired by the “array-based approach” of Bajorski *et al.* [2].

#### 4.4 Test Data Series

From each  $C_{r,i,*}$  output data series CryptoStat derives a test data series designated  $T_{r,i,*}$ ; and from each  $C_{r,*,j}$  output data series CryptoStat derives a test data series designated  $T_{r,*,j}$ . CryptoStat then performs odds ratio tests on these test data series.

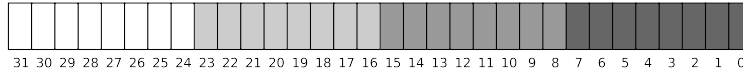
CryptoStat lets the user specify the method to derive the  $T$  data series from the  $C$  output series in various ways, including:

- Direct. The test data series is the  $C$  output series:  $T_{r,i,j} = C_{r,i,j}$ .
- Difference. Each test data value is the difference (bitwise exclusive-or) between a  $C$  output value and the previous  $C$  output value in the series. Specifically, for the  $T_{r,i,*}$  test data series,  $T_{r,i,j} = C_{r,i,j} \oplus C_{r,i,j-1}$ ,  $j \geq 1$ ; for the  $T_{r,*,j}$  test data series,  $T_{r,i,j} = C_{r,i,j} \oplus C_{r,i-1,j}$ ,  $i \geq 1$ .
- Avalanche. Each test data value is the difference (bitwise exclusive-or) between a  $C$  output value and the first  $C$  output value in the series. Specifically, for the  $T_{r,i,*}$  test data series,  $T_{r,i,j} = C_{r,i,j} \oplus C_{r,i,0}$ ,  $j \geq 1$ ; for the  $T_{r,*,j}$  test data series,  $T_{r,i,j} = C_{r,i,j} \oplus C_{r,0,j}$ ,  $i \geq 1$ .

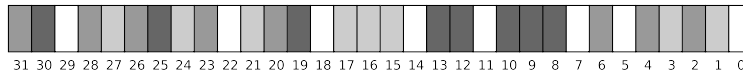
The test data series specifications can be paired with the input value series specifications to observe certain kinds of cryptographic function behavior. Pairing the “one-off” input value series with the “avalanche” test data series observes how the function’s outputs change when a single bit is flipped in the first input value. Pairing the “Gray code” input value series with the “difference” test data series observes how the function’s outputs change when a single bit is flipped in the preceding input value. Ideally, in both cases, the *avalanche effect* should be observed: a small change to the inputs should result in a large, random change in the outputs.

The test data series derivation is implemented as a Java class (a subclass of the Test abstract base class) and a GPU kernel. The Java class encapsulates the test data series’ characteristics, such as its name. The GPU kernel contains the code to derive the series of test data values from the series of  $C$  output values, as well as the code to perform odds ratio tests on the test data series.

During an analysis run, CryptoStat constructs a *test object* (an instance of a subclass of the abstract Test class) that derives the desired test data series. CryptoStat uses the test object to perform odds ratio tests on the test data



**Fig. 2.** Example of groups of adjacent bits



**Fig. 3.** Example of groups of scattered bits

series. The individual odds ratio tests are executed in parallel on the GPU’s many cores, thus speeding up the analysis run.

The CryptoStat distribution includes classes and GPU kernels for the direct, difference, and avalanche test data series. The user can define additional test classes and GPU kernels to derive other test data series.

#### 4.5 Output Bit Groups

CryptoStat partitions the  $C$  output (and  $T$  test data) bit positions into disjoint subsets called *bit groups*. CryptoStat performs a randomness test on each separate bit group in each of the  $T$  data series. The user can specify the number of bits in a bit group, from one to eight bits (all bit groups are the same size). The user can specify the locations of the bits in each bit group in various ways, including:

- The bits in each bit group are adjacent to each other. For example, Fig. 2 depicts four groups of eight adjacent bit positions in a 32-bit  $C$  output value, namely bits 0–7, 8–15, 16–23, and 24–31.
- The bits in each bit group are scattered across the  $C$  output value. For example, Fig. 3 depicts four groups of eight scattered bit positions, namely bits (8, 9, 10, 12, 13, 19, 25, 30), (2, 4, 6, 20, 23, 26, 28, 31), (1, 3, 15, 16, 17, 21, 24, 27), and (0, 5, 7, 11, 14, 18, 22, 29).

CryptoStat analyzes the randomness of each bit group separately. By testing one-bit groups, CryptoStat can detect nonrandomness in individual  $C$  output bit positions. By testing two-bit or larger bit groups, CryptoStat can detect correlations among multiple  $C$  output bit positions. Where the  $C$  output consists of several portions, such as the ciphertext and tag of an authenticated cipher’s output, CryptoStat can diagnose nonrandomness in the bit groups in each separate portion of the output.

To partition the  $C$  output bit positions into bit groups, CryptoStat constructs a *bit group object* (an instance of a subclass of the abstract `BitGroup` class), then queries the bit group object to get the bit positions in every bit group. The CryptoStat distribution includes bit group subclasses for bit groups consisting

of adjacent bit positions and bit groups consisting of randomly chosen scattered bit positions. The user can define additional bit group subclasses to generate other kinds of bit groups.

#### 4.6 Randomness Tests

CryptoStat performs randomness tests for each bit group in each test data series. The bit group values are integers in the range 0 through  $2^b - 1$ , where  $b$  is the bit group size. CryptoStat hypothesizes that the bit group values are uniformly distributed, as would be expected of a cryptographic function that is supposed to produce random-looking outputs.

Many statistical tests have been proposed to test a series of discrete values for uniformity; see [14] for a survey. One fairly simple test is a *frequency test*, in which there are  $2^b$  bins, one for each possible bit group value, and the probability of each bin is the same,  $\text{pr}(x) = 2^{-b}$ .

However, the frequency test has two major drawbacks. First, the frequency test cannot distinguish a random sequence from a nonrandom sequence in which the bit group values nonetheless appear with the expected frequencies. For example, for two-bit groups, the frequency test would pass when applied to sequences such as (0 1 2 3 0 1 2 3 ...) or (3 3 2 2 1 1 0 0 ...), even though such sequences are manifestly not random. Second, the frequency test requires an exponentially growing amount of storage to hold all the bin counters and an exponentially growing amount of CPU time to compute log odds ratios as the bit group size increases. This in turn limits the degree of parallelism that can be achieved when running on the GPU and increases the time needed for analysis runs.

Instead of frequency tests, CryptoStat uses *run tests* and *noncolliding block tests* to test the bit group value sequences for uniformity. These tests are designed to fail, not only when the bit group values themselves fail to appear with the correct frequencies, but also when certain *patterns* in the bit group values fail to appear with the correct frequencies.

The run test examines consecutive nonoverlapping blocks of four values ( $v_1 v_2 v_3 v_4$ ) in a bit group value series and checks whether  $v_1 < v_2$  or  $v_1 \geq v_2$ ,  $v_2 < v_3$  or  $v_2 \geq v_3$ , and  $v_3 < v_4$  or  $v_3 \geq v_4$ . For each block, this yields one of eight possible comparison patterns: ( $<<<<$ ), ( $<<\geq$ ), ( $<\geq<$ ), ( $<\geq\geq$ ), ( $\geq<<$ ), ( $\geq<\geq$ ), ( $\geq\geq<$ ), and ( $\geq\geq\geq$ ). Table 3 gives the probability of each comparison pattern (bin) for bit group sizes from one through eight bits, assuming that the bit group values are uniformly distributed. (I derived these probabilities by enumerating all possible four-value blocks.) The run test performs an odds ratio test for a discrete distribution with these probabilities as described in Section 3.5, yielding a log odds ratio.

Unlike the frequency test, in cases where the bit group values by themselves appear with the expected frequencies but there are nonrandom patterns in the bit group values, the run test tends to fail because the eight comparison patterns do not appear with the expected frequencies. Also unlike the frequency test, the run test requires the same amount of storage (eight bin counters) regardless of the bit group size.



**Table 3.** Run test, bin probabilities

Bin	Bit group size							
	1	2	3	4	5	6	7	8
<<<<	0	1	70	1820	35960	635376	10668000	174792640
<<<≥	0	15	378	7140	122760	2031120	33028128	532668480
<≥<<	1	35	714	12580	210056	3428880	55396384	890576960
<≥>>	3	45	630	9180	139128	2162160	34076640	541056960
≥<<<	0	15	378	7140	122760	2031120	33028128	532668480
≥<≥	4	65	966	14620	226424	3559920	56444896	898965440
≥><<	3	45	630	9180	139128	2162160	34076640	541056960
≥>>>	5	35	330	3876	52360	766480	11716640	183181376
Denom.	2 <sup>4</sup>	2 <sup>8</sup>	2 <sup>12</sup>	2 <sup>16</sup>	2 <sup>20</sup>	2 <sup>24</sup>	2 <sup>28</sup>	2 <sup>32</sup>

(Note: The bin probability is the table entry divided by the denominator in the last row.)

**Table 4.** Noncolliding block test, noncolliding block probabilities

$b$	$m$	$p = m!/m^m$
1	2	$5.0000 \times 10^{-1}$
2	4	$9.3750 \times 10^{-2}$
3	8	$2.4033 \times 10^{-3}$
4	16	$1.1342 \times 10^{-6}$
5	32	$1.8004 \times 10^{-13}$
6	64	$3.2203 \times 10^{-27}$
7	128	$7.2993 \times 10^{-55}$
8	256	$2.6544 \times 10^{-110}$

The noncolliding block test examines consecutive nonoverlapping blocks of  $m = 2^b$  values in a bit group series, where  $b$  is the bit group size and  $m$  is the number of possible bit group values. A *colliding block* is a length- $m$  block with one or more repeated values (collisions); a *noncolliding block* is a length- $m$  block with no repeated values. In other words, in a noncolliding block every possible value appears just once. Of the  $m^m$  possible arrangements of  $m$  possible values in a length- $m$  block, there are  $m!$  arrangements where no value is repeated. Therefore, the probability of a noncolliding block in a sequence of uniformly distributed values is  $m!/m^m$ . Each block can be viewed as a Bernoulli trial, where a noncolliding block is a success. Let  $n$  be the number of blocks,  $k$  be the number of noncolliding blocks, and  $p = m!/m^m$ ; then the noncolliding block test's log odds ratio is given by Equation (5). Table 4 lists the noncolliding block probabilities for bit groups of one through eight bits.

The noncolliding block test is inspired by the observation that some cryptographic functions, especially reduced-round versions, when presented with a series of  $2^b$  consecutive input values, generate a series of output values with those input values merely arranged in a different order; that is, a noncolliding block. For example, eight-bit groups in the ciphertext outputs of the early rounds of AES behave this way. The noncolliding block test would fail in this case because there would be too many noncolliding blocks, whereas the frequency test would pass.

CryptoStat combines (adds) together the run test's and the noncolliding block test's log odds ratios and reports the sum as the test result.

#### 4.7 Aggregate Test Results

CryptoStat computes a separate test result, namely the combined log odds ratio for the run and noncolliding block tests, for each bit group in each output series. Let  $L_{r,i,*,g}$  be the log odds ratio for bit group  $g$  in output series  $C_{r,i,*}$ ,  $0 \leq g \leq G-1$ , where  $G$  is the number of bit groups. Let  $L_{r,*,j,g}$  be the log odds ratio for bit group  $g$  in output series  $C_{r,*,j}$ ,  $0 \leq g \leq G-1$ .

A typical CryptoStat run would calculate many, many log odds ratios. For example, consider the AES block cipher, where input  $A$  is the 128-bit plaintext, input  $B$  is the 128-bit key, output  $C$  is the 128-bit ciphertext, there are 10 rounds, and the user has specified 500  $A$  input values, 500  $B$  input values, and 128  $C$  output bit groups of one bit each. Then CryptoStat would calculate  $10 \times 500 \times 128 = 640000$   $L_{r,i,*,g}$  log odds ratios and an equal quantity of  $L_{r,*,j,g}$  log odds ratios.

To gain insight into the cryptographic function's randomness from all these individual test results, CryptoStat computes and prints the following aggregate test results. (Recall that log odds ratios are aggregated by adding them together.)

- For each round  $r$  and each input  $A_i$ , compute and print

$$L_{r,i,*,*} = \sum_{g=0}^{G-1} L_{r,i,*,g}$$

This yields insight into the function's randomness for specific  $A$  input values, aggregated across all bit groups.

- For each round  $r$  and each input  $B_j$ , compute and print

$$L_{r,*,j,*} = \sum_{g=0}^{G-1} L_{r,*,j,g}$$

This yields insight into the function's randomness for specific  $B$  input values, aggregated across all bit groups.

- For each round  $r$  and each bit group  $g$ , compute and print

$$L_{r,*,*,g} = \sum_{i=0}^{M-1} L_{r,i,*,g} + \sum_{j=0}^{N-1} L_{r,*,j,g}$$

This yields insight into the function’s randomness for specific bit groups, aggregated across all  $A$  input values and all  $B$  input values.

- For each round  $r$ , compute and print

$$L_{r,*,*,*} = \sum_{g=0}^{G-1} L_{r,*,*,g}$$

This yields insight into the function’s overall randomness, aggregated across all  $A$  input values, all  $B$  input values, and all bit groups.

(CryptoStat normally does not print the individual test results  $L_{r,i,*,g}$  and  $L_{r,*,j,g}$ . The user can optionally turn on verbose output to print these in addition to the aggregate test results.)

#### 4.8 Randomness Margin

All of CryptoStat’s test results are reported separately for each reduced-round version of the cryptographic function, from one round up to the full number of rounds. Typically, a cryptographic function exhibits nonrandom behavior (negative log odds ratios) for the smaller numbers of rounds, but eventually exhibits random behavior (positive log odds ratios) once enough rounds are computed.

The function’s *randomness margin* is the number of rounds exhibiting random behavior as a fraction of the full number of rounds. CryptoStat calculates the randomness margin from the overall aggregate log odds ratios. Let  $r$  be the largest round such that  $L_{r,*,*,*} < 0$ ; then the randomness margin is  $(R - r)/R$ .

The randomness margin gives a quick overall indication of the function’s randomness. Randomness margins closer to 1 are preferred. A randomness margin closer to 0 suggests a weakness in the function’s design; possibly the function is not randomizing its inputs adequately, or possibly the function needs more rounds. The various aggregate test results ( $L_{r,i,*,*}$  and  $L_{r,*,j,*}$  and  $L_{r,*,*,g}$ ) or the individual test results ( $L_{r,i,*,g}$  and  $L_{r,*,j,g}$ ) might help identify the design weakness.

#### 4.9 Analysis Programs

The preceding sections have described what happens during one CryptoStat analysis run: the user specifies the cryptographic function, a generator for some number of  $A$  input values, and a generator for some number of  $B$  input values; the function is evaluated on those  $A$  and  $B$  values, yielding  $C$  output series; bit groups in each test data series derived from the  $C$  output series are subjected to randomness tests. A Java program named **Analyze** in the CryptoStat distribution performs one analysis run and prints the results.

However, a single analysis run exercises only a limited portion of the cryptographic function’s mapping. To get more coverage, one wants to do multiple analysis runs with  $A$  input value series of various kinds,  $B$  input value series of

**Table 5.** Cryptographic functions analyzed

Cryptographic function	$A$ input bit size	$B$ input bit size	$C$ output bit size
AES-128	128	128	128
AES-192	128	192	128
AES-256	128	256	128
SHA-1	256	160	160
SHA-224	256	160	224
SHA-256	256	160	256
SHA-384	512	320	384
SHA-512	512	320	512
SHA-512/224	512	320	224
SHA-512/256	512	320	256

various kinds,  $C$  output bit groups of various sizes and locations, and various test data series.

To automate such a series of analysis runs, the CryptoStat distribution includes another Java program, named `AnalyzeSweep`. For this program the user specifies a cryptographic function, a *list* of  $A$  input value generators, a *list* of  $B$  input value generators, a *list* of  $C$  output bit group specifications, and a test data series specification. The `AnalyzeSweep` program then automatically does an analysis run for every combination of an  $A$  input value generator, a  $B$  input value generator, and a  $C$  output bit group specification from the lists. For each combination, the program prints the number of nonrandom rounds detected. Finally, the program prints the maximum number of nonrandom rounds detected over all the combinations as well as the resulting randomness margin.

The Java programs and classes in the CryptoStat distribution utilize the Parallel Java 2 Library [10, 11], a class library for multicore, cluster, and GPU parallel programming in Java. The GPU kernels are written in C using Nvidia Corporation’s CUDA.

## 5 AES, SHA-1, and SHA-2 Analysis Results

I used CryptoStat to analyze all the versions of the AES block cipher [1] and the SHA-1 and SHA-2 hash functions [20]. For the block ciphers, the  $A$  and  $B$  inputs were the plaintext and the key; the  $C$  output was the ciphertext. For the hash functions, a fixed-length message consisting of a single message block was hashed; the  $A$  input was the first half of the message block; the  $B$  input was the second half of the message block, leaving space at the end to add the hash function’s padding; the  $C$  output was the digest. Table 5 lists the cryptographic functions analyzed and the  $A$ ,  $B$ , and  $C$  bit sizes.

I used the `AnalyzeSweep` program to test the randomness of each cryptographic function. I did one program run with direct test data series, one program run with avalanche test data series, and one program run with difference test

data series (refer to Section 4.4). Files containing the program runs' outputs, which include the `AnalyzeSweep` command executed, the list of *A* input value series, the list of *B* input value series, and the list of *C* output bit groups, are posted on the Web.<sup>2</sup>

## 5.1 Direct Tests

For the direct tests on each block cipher and hash function, I specified a list of *A* input value series. The first series in the list consisted of 512 sequential input values starting from 0 (refer to Section 4.2). The next series consisted of 512 sequential input values left-shifted by 8 bits; the series after that, 512 sequential input values left-shifted by 16 bits; and so on until the input values hit the most significant end of the *A* input. The *A* input value series list also included each of the above input value series, complemented.

I specified a list of *B* input value series, in the same manner as the *A* input value series.

I specified a list of the following 13 *C* output bit group definitions (refer to Section 4.5):

- Adjacent 1-bit groups
- Adjacent 2-bit groups
- Three different scattered 2-bit groups
- Adjacent 4-bit groups
- Three different scattered 4-bit groups
- Adjacent 8-bit groups
- Three different scattered 8-bit groups

I executed the `AnalyzeSweep` program to perform multiple direct test runs (refer to Section 4.4), one run for each combination of an *A* input value series, a *B* input value series, and a *C* output bit group. Ideally, each output bit group should exhibit a uniform distribution of values.

## 5.2 Avalanche Tests

For the avalanche tests on each block cipher and hash function, I specified one *A* input value series, consisting of the concatenation of the following:

- One-off inputs starting from 000...000 hexadecimal
- One-off inputs starting from 555...555 hexadecimal
- One-off inputs starting from aaa...aaa hexadecimal
- One-off inputs starting from fff...fff hexadecimal

---

<sup>2</sup> <https://www.cs.rit.edu/~ark/parallelcrypto/cryptostat/#testingtherandomness>

I specified one  $B$  input value series, also consisting of the concatenation of the above one-off inputs. I specified a list of the same 13  $C$  output bit group definitions as the direct tests.

I executed the `AnalyzeSweep` program to perform 13 avalanche test runs, one run for each  $C$  output bit group along with the  $A$  input series and the  $B$  input series. Recall that pairing the “one-off” input value series with the “avalanche” test data series observes how the function’s outputs change when a single bit is flipped in the first input value. Ideally, the avalanche effect should be observed: a small change to the inputs should result in a large, random change in the outputs.

### 5.3 Difference Tests

For the difference tests on each block cipher and hash function, I specified a list of  $A$  input value series. The first series in the list consisted of 512 input values in a Gray code sequence starting from 0. The next series consisted of 512 input values in a Gray code sequence left-shifted by 8 bits; the series after that, 512 sequential input values in a Gray code sequence left-shifted by 16 bits; and so on until the input values hit the most significant end of the  $A$  input. The  $A$  input value series list also included each of the above input value series, complemented.

I specified a list of  $B$  input value series, in the same manner as the  $A$  input value series. I specified a list of the same 13  $C$  output bit group definitions as the direct tests.

I executed the `AnalyzeSweep` program to perform multiple difference test runs, one run for each combination of an  $A$  input value series, a  $B$  input value series, and a  $C$  output bit group. Recall that pairing the “Gray code” input value series with the “difference” test data series observes how the function’s outputs change when a single bit is flipped in the previous input value. Ideally, the avalanche effect should be observed: a small change to the inputs should result in a large, random change in the outputs.

### 5.4 Test Results

For each cryptographic function, Table 6 lists the full number of rounds and the largest number of nonrandom rounds detected by the `AnalyzeSweep` direct test runs, avalanche test runs, and difference test runs. Recall that, for a particular test run, the number of nonrandom rounds corresponds to the largest round with a negative aggregate log odds ratio (refer to Section 4.8). The avalanche and difference test results show that each function does in fact exhibit the avalanche effect, after computing a sufficient number of rounds.

For each cryptographic function, Table 7 lists the full number of rounds, the largest number of nonrandom rounds detected by all the `AnalyzeSweep` test runs, and the cryptographic function’s randomness margin (refer to Section 4.8).

As expected, the full-round versions of the AES block cipher input-output mappings and the SHA-1 and SHA-2 hash function input-output mappings exhibit random behavior. Beyond that, the data shows that each function has a

**Table 6.** Number of nonrandom rounds detected

Cryptographic function	Full rounds	Direct test	Avalanche test	Difference test
AES-128	10	2	1	2
AES-192	12	3	2	2
AES-256	14	3	2	2
SHA-1	80	23	20	22
SHA-224	64	17	15	17
SHA-256	64	17	16	17
SHA-384	80	17	16	17
SHA-512	80	18	16	18
SHA-512/224	80	18	16	17
SHA-512/256	80	18	16	17

**Table 7.** Randomness margins

Cryptographic function	Full rounds $R$	Nonrandom rounds $r$	Randomness margin $(R - r)/R$
AES-128	10	2	0.800
AES-192	12	3	0.750
AES-256	14	3	0.786
SHA-1	80	23	0.713
SHA-224	64	17	0.734
SHA-256	64	17	0.734
SHA-384	80	17	0.788
SHA-512	80	18	0.775
SHA-512/224	80	18	0.775
SHA-512/256	80	18	0.775

substantial randomness margin, with 71 to 80 percent of the rounds exhibiting random behavior, indicative of a conservative design from a randomness point of view.

## References

1. *Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication 197 (2001)
2. Bajorski, P., Kaminsky, A., Kurdziel, M., Lukowiak, M., Radziszowski, S.: Array-based statistical analysis of the MK-3 authenticated encryption scheme. In: *IEEE Military Communications Conference (MILCOM)* (2018)
3. Brown, R., Eddelbuettel, D., Bauer, D.: Dieharder: a random number test suite (2013). <http://www.phy.duke.edu/~rgb/General/dieharder.php>, retrieved 10-Apr-2013

4. Doğanaksoy, A., Ege, B., Koçak, O., Sulak, F.: Cryptographic randomness testing of block ciphers and hash functions. Cryptology ePrint Archive, Report 2010/564 (2010)
5. Doğanaksoy, A., Ege, B., Koçak, O., Sulak, F.: Statistical analysis of reduced round compression functions of SHA-3 second round candidates. Cryptology ePrint Archive, Report 2010/611 (2010)
6. El-Fotouh, M., Diepold, K.: Statistical testing for disk encryption modes of operations. Cryptology ePrint Archive, Report 2007/362 (2007)
7. Filiol, E.: A new statistical testing for symmetric ciphers and hash functions. In: Deng, R., Bao, F., Zhou, J., Qing, S. (eds.) *4th International Conference on Information and Communications Security (ICICS 2002)*. LNCS, vol. 2513, pp. 342–353. Springer, Heidelberg (2002)
8. Hellekalek, P., Wegenkittl, S.: Empirical evidence concerning AES. *ACM Transaction on Modeling and Computer Simulation* 13, 322–333 (2003)
9. Hong, D., Sung, J., Hong, S., Lim, J., Lee, S., Koo, B., Lee, C., Chang, D., Lee, J., Jeong, K., Kim, H., Kim, J., Chee, S.: HIGHT: a new block cipher suitable for low-resource device. In Goubin, L., Matsui, M. (eds.) *8th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2006)*. LNCS, vol. 4249, pp. 46–59. Springer, Heidelberg (2006)
10. Kaminsky, A.: The Parallel Java 2 Library: Parallel Programming in 100% Java. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC14)*, Poster Session. (2014) [http://sc14.supercomputing.org/sites/all/themes/sc14/files/archive/tech\\_poster/tech\\_poster\\_pages/post116.html](http://sc14.supercomputing.org/sites/all/themes/sc14/files/archive/tech_poster/tech_poster_pages/post116.html), retrieved 25-Jan-2018
11. Kaminsky, A.: The Parallel Java 2 Library. <https://www.cs.rit.edu/~ark/pj2.shtml>
12. Kass, R., Raftery, A.: Bayes factors. *Journal of the American Statistical Association* 90, 773–795 (1995)
13. Katos, V.: A randomness test for block ciphers. *Applied Mathematics and Computation* 162, 29–35 (2005)
14. Knuth, D.: *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition* (1998)
15. L’Ecuyer, P., Simard, R.: TestU01: a C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software* 33, 22 (2007)
16. Marsaglia, G.: Diehard battery of tests of randomness. <http://i.cs.hku.hk/~diehard/cdrom/>, retrieved 03-Apr-2013
17. Ojha, S., Kumar, N., Jain, K., Sangeeta: TWIS—a lightweight block cipher. In Prakash, A., Sen Gupta, I. (eds.) *5th International Conference on Information Systems Security (ICISS 2009)*. LNCS, vol. 5905, pp. 280–291. Springer, Heidelberg (2009)
18. Press, W., Teukolsky, S., Vetterling, W., Flannery, B.: *Numerical Recipes: The Art of Scientific Computing, Third Edition*. Cambridge University Press, Cambridge (2007)
19. Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., Vo, S., Bassham, L.: A statistical test suite for random and pseudorandom number generators for cryptographic applications. *NIST Special Publication 800-22 Revision 1a* (2010). <http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf>, retrieved 03-Apr-2013
20. *Secure Hash Standard (SHS)*. Federal Information Processing Standards Publication 180-4 (2015)



21. Soto, J.: Statistical testing of random number generators. *22nd National Information Systems Security Conference* (1999). <http://csrc.nist.gov/groups/ST/toolkit/rng/documents/nissc-paper.pdf>, retrieved 03-Apr-2013
22. Soto, J., Bassham, L.: Randomness testing of the Advanced Encryption Standard finalist candidates. *NIST IR 6483* (2000). <http://csrc.nist.gov/groups/ST/toolkit/rng/documents/aes-report-final.doc>, retrieved 03-Apr-2013
23. Sulak, F., Doğanaksoy, A., Ege, B., Koçak, O.: Evaluation of randomness test results for short sequences. In: Carlet, C., Pott, A. (eds.) *Sequences and Their Applications—SETA 2010*. LNCS, vol. 6338, pp. 309–319. Springer, Heidelberg (2010)
24. Walker, J.: ENT: a pseudorandom number sequence test program (2008). <http://www.fourmilab.ch/random/>, retrieved 10-Apr-2013
25. Wang, H., Zhang, H.: A fast pseudorandom number generator with BLAKE hash function. *Wuhan University Journal of Natural Sciences* 15, 393–397 (2010)