Noname manuscript No.
(will be inserted by the editor)

GPU Parallel Statistical and Cube Test Analysis of the SHA-3 Finalist Candidate Hash Functions

Alan Kaminsky

the date of receipt and acceptance should be inserted later

Abstract The 256-bit versions of the SHA-3 finalist candidate hash functions—BLAKE, Grøstl, JH, Keccak, and Skein—were subjected to statistical tests to attempt to disprove the hypothesis that the output bits are uniformly distributed, independent, binary random variables. The hash functions were also subjected to cube tests to attempt to disprove the hypothesis that the superpoly bits are uniformly distributed, independent, binary random variables. The hash functions and test programs were implemented to run in parallel on a 448-core GPU supercomputer; the cube tests in particular require massive amounts of computation and are ideally suited for parallel implementation. Nonrandom behavior was observed at the 0.01 significance level in the BLAKE, JH, Keccak, and Skein hash functions. Nonrandom behavior was not observed at the 0.01 significance level in the Grøstl hash function.

Keywords Hash functions · SHA-3 · BLAKE · Grøstl · JH · Keccak · Skein · statistical tests · cube tests · parallel computing · GPU computing

1 Introduction

In 2007 the U.S. National Institute of Standards and Technology (NIST) inaugurated a competition to choose a new cryptographic hash algorithm [17]. After two rounds of the competition, in December 2010 NIST selected five candidate hash functions—BLAKE, Grøstl, JH, Keccak, and Skein—to advance to the third and final round. The cryptographic community has been and is analyzing the candidate hash functions' security. In 2012 NIST will select the winning algorithm and designate it as SHA-3.

NIST stated that "Hash algorithms will be evaluated against attacks or observations that may threaten existing or proposed applications, or demonstrate some fundamental flaw in the design, such as exhibiting nonrandom behavior and failing statistical tests" [17]. While extensive cryptanalysis of the SHA-3 candidates has been published (see [21] for a bibliography), little statistical analysis of the SHA-3 candidates has been published (see [11,14,23]). This paper's first contribution is a statistical analysis of all five SHA-3 finalist candidates.

The de facto standard for statistical analysis of cryptographic functions is the NIST test suite [20]. It consists of 15 statistical tests performed on binary sequences of length one million bits or more. While the NIST test suite document describes how to apply the statistical tests to such a long binary sequence and how to interpret the results, it does not describe how to evaluate a cryptographic function to generate a long binary sequence in the first place. The method for generating a long binary sequence is particularly unclear for block ciphers, hash functions, and MACs, which produce short, fixed-length outputs (in contrast to stream ciphers, which produce arbitrary-length outputs). This paper's second contribution is a *complete* methodology for statistical analysis of a cryptographic function, especially one with a fixed-length output, based on testing the hypothesis that the function's output bits are uniformly distributed, independent, binary random variables. The methodology prescribes both how to evaluate the function and how to test the function's output bit sequences.

Besides testing the cryptographic function itself, the statistical behavior of the function's internal polyno-

Department of Computer Science, Rochester Institute of Technology, 102 Lomb Memorial Drive, Rochester, NY 14623, USA, E-mail: ark@cs.rit.edu, Tel./Fax: +1 585–568–8705

mial structure can also be probed using the *cube test* [2]. This provides additional insight into the function's statistical behavior beyond just its external black-box outputs. The cube test computes so-called *superpolys* of the function; computing a superpoly requires 2^c evaluations of the function for some c. This paper's statistical analysis methodology includes testing the hypothesis that the function's superpoly bits are uniformly distributed, independent, binary random variables.

Evaluating the cryptographic function to produce binary sequences for the function's output bits and superpoly bits, and executing the statistical tests on these numerous binary sequences, requires extensive computation. However, the computations can be carried out in a massively parallel fashion and are ideally suited to run on a graphics processing unit (GPU) supercomputer. This paper's third contribution is a GPU parallel implementation of each of the SHA-3 finalist candidate hash functions, of the function and superpoly evaluations to produce binary sequences, and of the statistical tests on the binary sequences. The test programs were run on an NVIDIA Tesla C2050 448-core GPU supercomputer.

The paper is organized as follows. Section 2 describes the statistical test methodology. Section 3 describes the GPU parallel implementation. Section 4 presents and interprets the statistical test results on the SHA-3 finalist candidates. Section 5 offers concluding remarks.

2 Statistical Test Methodology

2.1 Hypotheses

The statistical test methodology treats the cryptographic function $F()$ being tested as a black box with w input bits and m output bits (Fig. 1). x designates the vector of input bits treated as a w-bit binary number. $F_i(x)$ designates the function that computes the i-th output bit, $1 \le i \le m$. $F(x) = (F_1(x), F_2(x), \ldots, F_m(x))$ designates the vector of output bits.

The statistical test methodology begins with two hypotheses:

- **Output Randomness Hypothesis.** $F_i(x)$ is a uniformly distributed binary random variable, $1 \le i \le m$.
- **Output Independence Hypothesis.** $F_i(x)$ and $F_j(x)$ are independent random variables, $1 \le i < j \le m$.

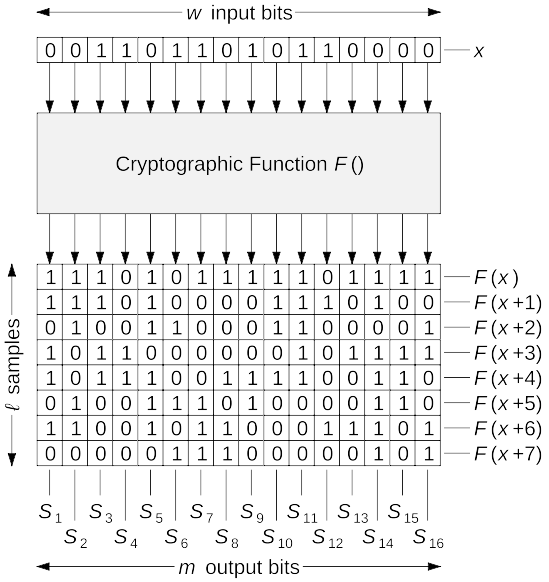

Fig. 1 Generation of binary sequences for cryptographic function output bits for one trial

These hypotheses capture the notion that when a cryptographic function's input changes, the output bits should change randomly.

2.2 Generation of Binary Sequences

Testing the hypotheses begins by generating a number of binary sequences, as follows. A certain number n of trials are performed. For each trial t, $1 \le t \le n$, an initial input value x_t is chosen at random (using a pseudorandom number generator). The cryptographic function is applied to ℓ successive input values starting from x_t (wrapping around to 0 if necessary): $F(x_t), F(x_t + 1), \ldots, F(x_t + \ell - 1)$.

Separating the output bits, we now have a binary sequence of length ℓ for each trial t, $1 \le t \le n$, and each output bit i, $1 \le i \le m$ (Fig. 1):

$$S_i^t = (F_i(x_t), F_i(x_t + 1), \ldots, F_i(x_t + \ell - 1)) \; . \qquad (1)$$

This study used $n = 2000$ trials and $\ell = 10240$ samples per trial. This relatively short sequence length was used so that generating the sequences would not take an excessive amount of time, especially when testing the cryptographic function's superpolys (Sect. 2.6).

2.3 Tests on Binary Sequences

A suite of statistical tests is applied to each binary sequence S_i^t to attempt to disprove the hypotheses. The NIST test suite is not used in its entirety because some

of the statistical tests in that suite require long binary sequences of one million bits or more [20]. Instead, several statistical tests taken from the NIST test suite, from Knuth [15], and from L'Ecuyer and Simard [16] are used.

Some tests in the suite are *chi-square tests.* A chi-square test proceeds as follows. The data in the binary sequence is categorized, and each time an item falls in a certain category, the count in the corresponding *bin* is incremented. The *chi-square statistic* is calculated:

$$\chi^2 = \sum_{i=1}^{b} \frac{(N_i - Np_i)^2}{Np_i} \ , \tag{2}$$

where b is the number of bins, N is the total number of items, N_i is the number of items in bin i, and p_i is the probability that an item will fall in bin i if the hypothesis is true. In the limit, χ^2 obeys a *chi-square distribution* with $b-1$ degrees of freedom (d.o.f.). For finite N, χ^2 only approximately obeys a chi-square distribution; but if the expected count in each bin (Np_i) is 5 or greater, the approximation is acceptable. The *p-value* is calculated; this is the probability that a statistic value greater than or equal to the observed value would occur by chance if the hypothesis is true. The formula for the chi-square p-value is

$$p\text{-value} = 1 - \texttt{gammp}\left(\frac{\text{d.o.f.}}{2}, \frac{\chi^2}{2}\right) \ , \tag{3}$$

where `gammp` is the *incomplete gamma function*,

$$\texttt{gammp}(a,x) = \frac{\int_0^x t^{a-1}\mathrm{e}^{-t}\mathrm{d}t}{\int_0^\infty t^{a-1}\mathrm{e}^{-t}\mathrm{d}t} \ . \tag{4}$$

Procedures for calculating `gammp` are well-known; see [19], for example. If the p-value falls below a certain small *significance* threshold, meaning the observed value of the statistic is very unlikely to occur by chance, the hypothesis is considered disproven.

Seven statistical tests are applied to each binary sequence S_i^t to attempt to disprove the Output Randomness Hypothesis:

Frequency test [20]. This checks whether 0 and 1 bits occur in equal proportions in the binary sequence. It is a chi-square test with two bins and equal bin probabilities of 1/2.

Serial-2 test [15]. This checks whether all possible values of two-bit blocks—00, 01, 10, and 11—occur in equal proportions. The blocks overlap: the first block consists of the first and second bits of the sequence, the second block consists of the second and third bits of the sequence, and so on. The test computes this statistic:

$$Y = \frac{4}{\ell}\sum_{i=0}^{3} N_2(i) - \frac{2}{\ell}\sum_{j=0}^{1} N_1(j) \ , \tag{5}$$

where $N_2(i)$ is the number of two-bit blocks with the value i and $N_1(j)$ is the number of one-bit blocks with the value j. If the hypothesis is true, Y obeys a chi-square distribution with 2 d.o.f.; (3) gives the p-value.

Serial-3 test [15]. This checks whether all possible values of overlapping three-bit blocks occur in equal proportions. The test computes this statistic:

$$Y = \frac{8}{\ell}\sum_{i=0}^{7} N_3(i) - \frac{4}{\ell}\sum_{j=0}^{3} N_2(j) \ , \tag{6}$$

where $N_3(i)$ is the number of three-bit blocks with the value i. If the hypothesis is true, Y obeys a chi-square distribution with 4 d.o.f.; (3) gives the p-value.

Serial-4 test [15]. This checks whether all possible values of overlapping four-bit blocks occur in equal proportions. The test computes this statistic:

$$Y = \frac{16}{\ell}\sum_{i=0}^{15} N_4(i) - \frac{8}{\ell}\sum_{j=0}^{7} N_3(j) \ , \tag{7}$$

where $N_4(i)$ is the number of four-bit blocks with the value i. If the hypothesis is true, Y obeys a chi-square distribution with 8 d.o.f.; (3) gives the p-value.

Gap test [15]. A *gap* in a binary sequence is defined to be a series of zero or more 0 bits between two consecutive 1 bits. The gap test checks whether gaps of different lengths appear in the expected proportions. It is a chi-square test with ten bins and the following bin probabilities:

Gap length	Bin prob.	Gap length	Bin prob.
0	1/2	5	1/64
1	1/4	6	1/128
2	1/8	7	1/256
3	1/16	8	1/512
4	1/32	≥ 9	1/512

The expected count in each bin is the number of gaps in the sequence times the bin probability. If the expected count in the last bin is less than 5, the last category is coalesced into the next-to-last category; this is repeated until the expected count in the last bin is 5 or greater.

Autocorrelation test [16]. This checks whether the binary sequence is uncorrelated with lagged versions of itself at lags of 1 through 9. If they are uncorrelated, the bitwise exclusive-or of the original sequence with the lagged sequence should be half 0s and half 1s. The test computes this statistic:

$$Y = \frac{2}{\ell}\sum_{j=1}^{9}\left(H(S_i^t \oplus \texttt{rot}(S_i^t, j)) - \frac{\ell}{2}\right)^2 \ , \tag{8}$$

where $\texttt{rot}(S_i^t, j)$ is the sequence circularly shifted j positions and $H()$ is the Hamming weight (number of 1s).

If the hypothesis is true, Y obeys a chi-square distribution with 9 d.o.f.; (3) gives the p-value.

Hamming-weight test [16,20]. This partitions the binary sequence into M subsequences of length $N = \ell/M$ and compares the Hamming weight of each subsequence to the expected value, $N/2$. The test computes this statistic:

$$Y = \frac{2}{N} \sum_{j=0}^{M-1} \left(H(S_i^t[jN \,..\, jN + N - 1]) - \frac{N}{2} \right)^2 , \quad (9)$$

where $S_i^t[a \,..\, b]$ is the subsequence of S_i^t from bit indexes a through b inclusive. If the hypothesis is true, Y obeys a chi-square distribution with M d.o.f.; (3) gives the p-value. This study used $M = 100$.

One statistical test is applied to each pair of binary sequences S_i^t and S_j^t, $1 \le i < j \le m$, to attempt to disprove the Output Independence Hypothesis:

Independence test. This checks whether all possible pairs of corresponding bits from the two binary sequences—$(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$—occur in equal proportions. It is a chi-square test with four bins and equal bin probabilities of 1/4.

2.4 Tests on Output Bits

The statistical tests of the Output Randomness Hypothesis in Sect. 2.3 yield a p-value for each of the seven tests, each of the m output bits, and each of the n trials (Fig. 2). The next step in the statistical test methodology is to attempt to disprove the Output Randomness Hypothesis for each output bit based on all the trials' p-values. If the hypothesis is true, the trials' p-values for a given test and output bit should obey a uniform$(0,1)$ distribution. This is checked by applying a *uniformity test:* the $(0,1)$ interval is partitioned into 20 equal subintervals (bins), the trials' p-values are sorted into the bins, and a chi-square test is performed on the bin counts.

In the case of the frequency test, the above procedure does not work. Because the binary sequences are relatively short and there are only two bins, only a certain discrete set of statistic values and p-values was observed, and the p-values were not spread uniformly over the $(0,1)$ interval. Instead, the following procedure is used for the frequency test: If the hypothesis is true, the trials' 0 counts for a given output bit should obey a binomial$(\ell, 0.5)$ distribution. This is checked by partitioning the $(0,\ell)$ interval into 20 subintervals (bins) of approximately equal bin probability (Fig. 3), sorting the trials' 0 counts into the bins, and performing a chi-square test on the bin counts.

The statistical tests of the Output Independence Hypothesis in Sect. 2.3 also yield a p-value for each of the $m(m-1)/2$ output bit pairs and each of the n trials. The statistical test methodology attempts to disprove the Output Independence Hypothesis for each output bit pair based on all the trials' p-values. If the hypothesis is true, the trials' independence test p-values for a given output bit pair should obey a uniform$(0,1)$ distribution. This is checked by applying the 20-bin uniformity test to the trials' p-values.

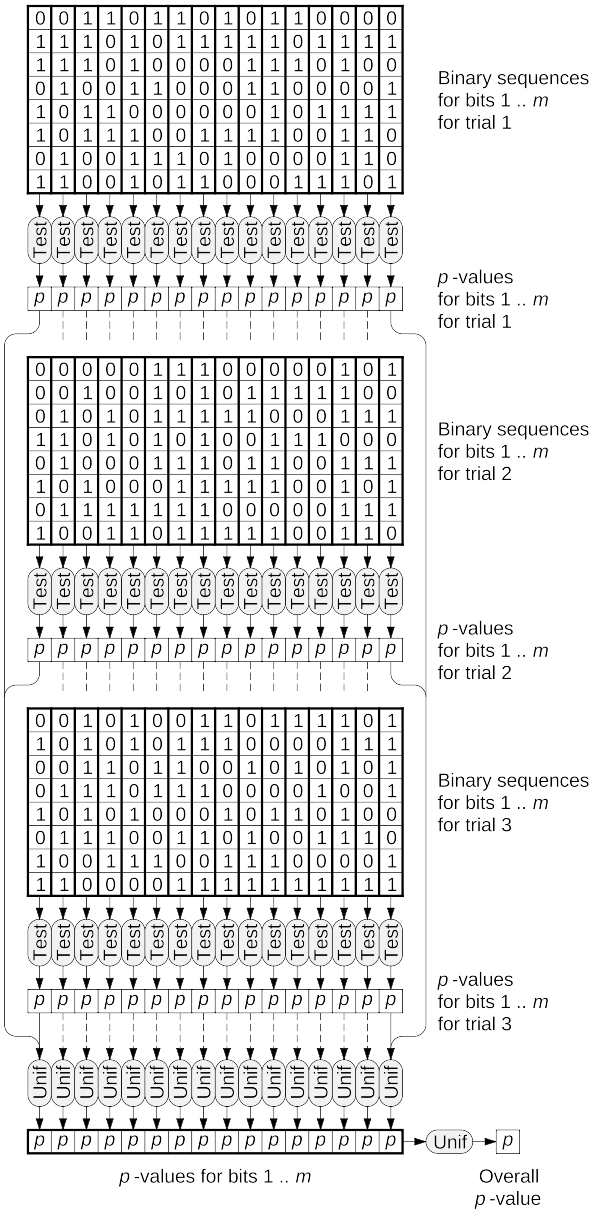

Fig. 2 Methodology for testing the Output Randomness Hypothesis. A certain statistical test (oval labeled "Test") is applied to the length-ℓ binary sequence for each output bit and each trial, yielding a p-value. A uniformity test (oval labeled "Unif") is applied to the n trials' p-values for each output bit, yielding another p-value. One final uniformity test is applied to the m output bits' p-values, yielding an overall p-value. This is repeated for each of the seven statistical tests

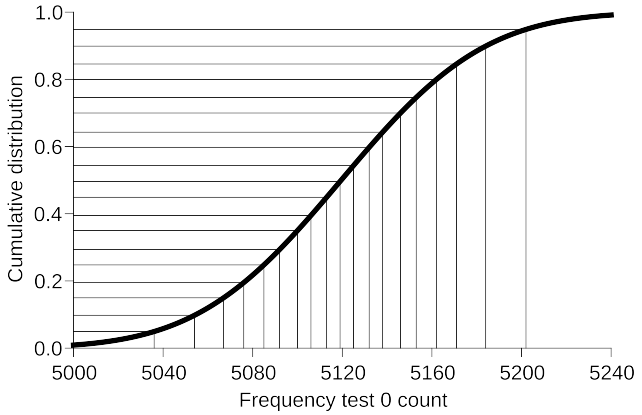

Fig. 3 Bins for a chi-square test of the frequency test 0 counts. The dark line plots the cumulative distribution function of the binomial$(10240, 0.5)$ distribution. The vertical lines show the boundaries of the 20 bins. The horizontal lines show that the bin probabilities are all approximately the same (because the distribution is discrete, they are not all exactly 1/20)

2.5 Tests on the Entire Function

The statistical tests of the Output Randomness Hypothesis in Sect. 2.4 yield a p-value for each of the seven tests and each of the m output bits (Fig. 2). The final step in the statistical test methodology is to attempt to disprove the Output Randomness Hypothesis for the entire cryptographic function based on all the output bits' p-values. If the hypothesis is true, the output bits' p-values for a given test should obey a uniform$(0, 1)$ distribution. This is checked by applying the 20-bin uniformity test to the output bits' p-values.

The statistical tests of the Output Independence Hypothesis in Sect. 2.4 also yield a p-value for each of the $m(m-1)/2$ output bit pairs. The statistical test methodology attempts to disprove the Output Independence Hypothesis for the entire cryptographic function based on all the output bit pairs' p-values. If the hypothesis is true, the output bit pairs' independence test p-values should obey a uniform$(0, 1)$ distribution. This is checked by applying the 20-bin uniformity test to the output bit pairs' p-values.

Applying the statistical test methodology to a cryptographic function yields, finally, eight overall p-values, one for each of the frequency, serial-2, serial-3, serial-4, gap, autocorrelation, Hamming-weight, and independence tests. If any p-value falls below a significance threshold chosen by the experimenter, the corresponding hypothesis is disproven, and the function exhibits nonrandom behavior. Sect. 4 gives the p-values observed when the statistical test methodology was applied to the BLAKE, Grøstl, JH, Keccak, and Skein hash functions.

2.6 Superpoly Tests

Dinur and Shamir introduced the notion of a *superpoly* in conjunction with the *cube attack* on a cryptographic function [9]. Aumasson, Dinur, Meier, and Shamir showed how superpolys can be used in statistical tests of cryptographic functions, which they dubbed *cube tests* [2].

A superpoly is defined as follows: Consider the function $F_i(x)$ that computes one output bit of a cryptographic primitive. Designate a number c of the input bits to be *cube inputs*, and let $y = (y_1, y_2, \ldots, y_c)$ be the vector of cube input bits. Designate a number s of the input bits to be *superpoly inputs* (different from the cube inputs), and let $z = (z_1, z_2, \ldots, z_s)$ be the vector of superpoly input bits. Evaluating $F_i(x)$ for given values of y and z, with the remaining input bits set to 0, results in another function $F'_i(y, z)$, which can be viewed as some GF(2) polynomial in the input bits. This polynomial is expressed as

$$F'_i(y, z) = y_1 y_2 \cdots y_c Q_i(z) + R_i(y, z) \ . \tag{10}$$

The first part of (10) consists of the terms in F'_i that include *all* the cube inputs. The cube inputs are factored out, leaving some GF(2) polynomial Q_i in just the superpoly inputs. Q_i is called the "superpoly of F_i" with respect to the chosen cube and superpoly input bits. The second part of (10) consists of the remaining terms in F'_i, which is some other GF(2) polynomial R_i in the cube and superpoly inputs.

The value of the superpoly Q_i can be calculated, without even knowing its formula, due to: [9]

Theorem 1

$$Q_i(z) = \sum_{y=00\ldots00}^{11\ldots11} F'_i(y, z) \ . \tag{11}$$

Proof In the summation over the 2^c values of y, each term in R_i is added whenever the term's cube inputs are all 1. This happens an even number of times, because no term in R_i contains all the cube inputs. Therefore, in GF(2) arithmetic, the terms in R_i sum up to 0. The terms in Q_i, however, are added in only once, when all the cube inputs are 1. Therefore, the summation yields just Q_i. □

Thus, for a certain set of cube input bits and superpoly input bits, the superpolys of F_i, $1 \le i \le m$, can be computed by evaluating the cryptographic function 2^c

times, with the same superpoly input values each time and different cube input values from $00\ldots00$ through $11\ldots11$ each time, then bitwise exclusive-oring (i.e., adding modulo 2) the outputs together.

Viewed as polynomials, a cryptographic function's output bits F_i ought to be *random* polynomials [2]. Accordingly, the superpolys of F_i ought to be random polynomials also, and a sequence of values of a superpoly Q_i ought to be a random binary sequence just like a sequence of values of F_i itself. The following hypotheses capture this notion:

- **Superpoly Randomness Hypothesis.** $Q_i(z)$ is a uniformly distributed binary random variable, $1 \le i \le m$.
- **Superpoly Independence Hypothesis.** $Q_i(z)$ and $Q_j(z)$ are independent random variables, $1 \le i < j \le m$.

The statistical test methodology tests these hypotheses exactly as described in Sects. 2.2–2.5, except the binary sequences are generated by calculating superpolys of the cryptographic function $(Q_i(z_t), Q_i(z_t + 1), \ldots, Q_i(z_t + \ell - 1))$ using (11). For each trial, the specific cube and superpoly input bits are chosen at random (using a pseudorandom number generator), as is the starting superpoly input value z_t. This study examined $n = 2000$ superpolys (trials) and $\ell = 10240$ samples per trial. The number of cube bits c varied from 1 to 20 and the number of superpoly bits s varied from 15 to 34, with 5 trials per (c, s) combination. Sect. 4 gives the p-values observed when the statistical test methodology was applied to the superpolys of the BLAKE, Grøstl, JH, Keccak, and Skein hash functions.

2.7 Related Work

The NIST test suite [20] includes 15 statistical tests that are applied to a number of binary sequences generated by the cryptographic function under test. The NIST test suite document recommends a minimum sequence length for each test to yield meaningful results; some tests can be performed on sequences as short as 100 bits, while other tests require sequences of one million bits or more. The NIST test suite document does not describe how to evaluate the cryptographic function to generate the binary sequences. NIST also provides a C program that implements the test suite. Given a number of binary sequences, the program calculates the p-value for each test and sequence. For each test, the program reports the number of sequences that passed the test at an experimenter-chosen significance level (where "pass" means the p-value was greater than the significance); the default significance is 0.01. In addition, for each test, the program applies a 10-bin chi-square uniformity test to the sequences' p-values and reports the uniformity test's p-value.

Sulak, Doğanaksoy, Ege, and Koçak [22], noting that the output length of a typical block cipher or hash function (128 to 256 bits) is too short to apply the NIST test suite, proposed an alternative statistical test methodology. Their methodology includes seven of the 15 NIST statistical tests. For each test, they calculated the exact probability that the test's p-value would fall into each of ten equal-sized bins in the $(0, 1)$ interval, given a sequence length of 128, 160, or 256 bits. These (typically unequal) bin probabilities are used in a chi-square uniformity test that is specific to each statistical test and sequence length. To apply one of the seven statistical tests to a block cipher, one million integers from 0 to 999999 are encrypted using an all-zero key; each plaintext is exclusive-ored with its ciphertext to form a binary sequence whose length is the cipher's block size; the statistical test's p-value is calculated for each sequence; and the 10-bin uniformity test is applied to the sequences' p-values using the appropriate bin probabilities. To apply one of the seven statistical tests to a hash function, a similar procedure is followed, except one million integers are hashed.

Unlike the methodology of [22], this paper's statistical test methodology focuses on the cryptographic function's output bits separately, hypothesizing that each individual output bit is a uniform, independent, binary random variable. Statistical tests and uniformity tests are applied to each output bit and each output bit pair separately; only at the last step are the output bit and pair results combined in one final uniformity test. This lets the experimenter diagnose nonrandomness of individual output bit positions and nonindependence of individual output bit pairs, which is not done in the methodology of [22].

This paper's statistical test methodology uses binary sequences that are much longer than the methodology of [22]—10240 bits versus 128 to 256 bits. With these longer sequences, the statistical tests' p-values fit much better into a simple uniform$(0, 1)$ distribution (except for the frequency test), obviating the multiplicity of uniformity tests with differing bin probabilities in the methodology of [22] and simplifying the implementation.

On the other hand, this paper's statistical test methodology uses binary sequences that are much shorter than the one-million-bit sequences required by some tests in the NIST test suite. This makes it possible to calculate binary sequences for a cryptographic function's superpolys for larger numbers of cube bits, and thus to probe more of the cryptographic function's

internal polynomial structure, within a reasonable running time.

More recently, Doğanaksoy, Ege, Koçak, and Sulak [10] proposed another statistical test methodology for block ciphers and hash functions whose tests focus on desirable cryptographic properties of those primitives, rather than general tests that could apply to any binary sequence. The methodology includes the strong avalanche criterion test, linear span test, collision test, and coverage test. The authors applied the methodology to reduced-round versions of each of the five AES finalist candidate block cipher algorithms to determine the smallest number of rounds needed for the algorithm to pass every test, and compared that with the actual number of rounds for the algorithm to determine a security margin. Depending on the particular block cipher, only 12 to 40 percent of the full number of rounds was sufficient.

Unlike the methodology of [10], this paper's methodology uses general tests that could apply to any binary sequence. Furthermore, it is not concerned with finding the minimum number of rounds needed to pass all the tests; it treats the cryptographic primitive as a black box and always tests the full number of rounds. (The authors of [10] do not state whether the full-round block cipher algorithms passed the tests.)

Finally, while the other reported methodologies examine only the function samples, this paper's methodology examines both the function samples and the superpoly samples. This provides a stronger assessment of the cryptographic primitive's statistical behavior; a primitive that passes the statistical tests on the function samples might not pass the statistical tests on the superpoly samples or vice versa.

3 GPU Parallel Implementation

The statistical test methodology's computations can be executed in a massively parallel fashion. For example, generating the binary sequences to test the Output Randomness and Independence Hypotheses for 2000 trials and 10240 samples per trial requires evaluating the cryptographic function more than twenty million times; however, each evaluation is completely independent of all the others and can be done in parallel with all the others. Likewise, generating the binary sequences to test the Superpoly Randomness and Independence Hypotheses for $c = 1$ to 20 cube bits, $s = 15$ to 34 superpoly bits, 5 trials per (c, s) combination, and 10240 samples per trial requires evaluating the cryptographic function more than two trillion times (about 2^{41} times); these evaluations also can be done in parallel.

Graphics processing units (GPUs), typically offering many more computational cores at a much lower cost per core than conventional multicore CPUs, are ideally suited for these kinds of massively parallel calculations. Accordingly, the statistical tests on the SHA-3 finalist candidate hash functions were carried out on an NVIDIA Tesla C2050 448-core GPU supercomputer. However, the GPU's architecture differs considerably from a conventional CPU's architecture. The GPU's unique characteristics must be taken into account when implementing GPU parallel programs in order to achieve the highest performance.

Section 3.1 describes the GPU's architecture at a high level, emphasizing the characteristics that affect GPU parallel programs. Section 3.2 describes the software architecture of the statistical test suite. Section 3.3 describes the GPU implementations of the SHA-3 finalist candidate hash functions.

3.1 GPU Architecture

Fig. 4 shows the architecture of the NVIDIA Tesla C2050 GPU supercomputer. It has 14 *multiprocessor* chips. Each multiprocessor chip has 32 processor *cores,* a high-speed *register* file with 32768 registers, and a slower *shared memory* with 49152 bytes of storage. There is also an off-chip *global memory* with 3 GiB of DRAM storage accessible by all the multiprocessors. A bus connects the GPU's global memory with the host CPU's main memory, allowing data to be transferred between the GPU and the CPU. Other GPUs have a similar architecture, with differing numbers of multiprocessors, cores, and registers, and differing amounts of memory. NVIDIA refers to this architecture as the Compute Unified Device Architecture (CUDA). NVIDIA provides an API and compiler for writing CUDA programs in Fortran, C, and C++.

While the GPU's memory hierarchy—fast registers, slower shared memory, very slow global memory—superficially resembles that of a conventional CPU, the GPU's storage is not cached in hardware. Instead, the software executing on the GPU must explicitly move data between the memory areas. A GPU program typically moves input data from global memory into shared memory; performs calculations on the data read from shared memory, using the registers for intermediate storage; writes the results back to shared memory, possibly combining results from separate calculations together in shared memory (*parallel reduction*); and moves the output data from shared memory back into global memory. The chip area that would normally be occupied by cache hardware is instead devoted to increasing the number of processor cores.

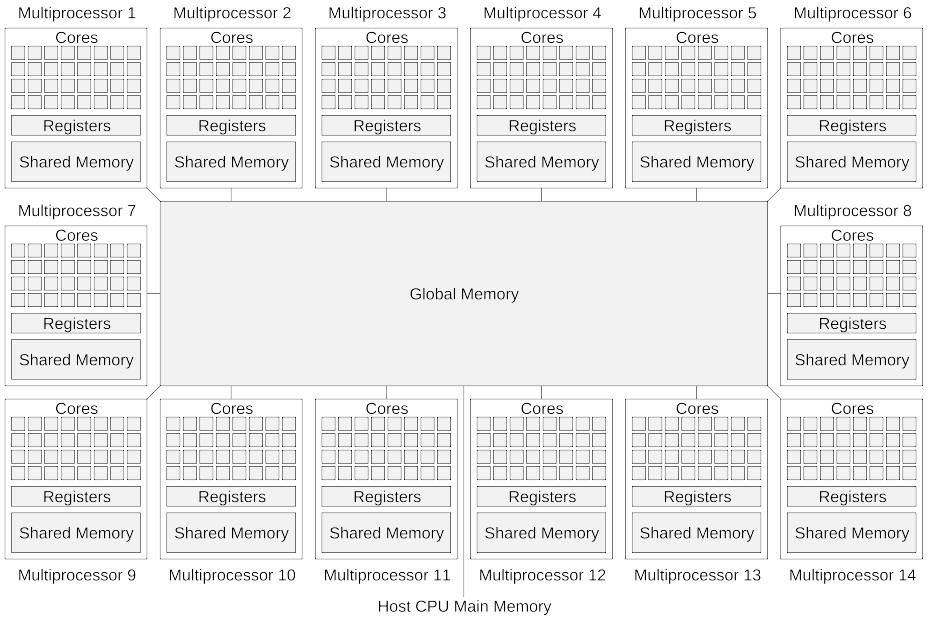

Fig. 4 NVIDIA Tesla C2050 GPU supercomputer architecture

Because neither the shared memory nor the global memory is cached, every memory access incurs the full memory latency. Thus, a GPU program achieves its highest performance when most of the instructions involve only the registers and the number of memory accesses is minimized. (When programming a GPU in a high level language, the compiler typically places a function's "local" or "automatic" variables in registers.) Consequently, on a GPU, it can take less time to recalculate a quantity whenever it is needed than to look it up in a table.

The *kernel* of a GPU program—the portion that executes in parallel on the cores of the GPU—consists of a number of *thread blocks,* each thread block containing a number of *threads.* Each thread block executes on a separate multiprocessor; typically, there are more thread blocks than multiprocessors, and the hardware schedules thread blocks to execute on the multiprocessors automatically. Each thread within a thread block executes on a separate core within a multiprocessor; typically, there are more threads than cores, and the hardware schedules threads to execute on the cores automatically. Only threads within the same thread block can access shared memory locations. All threads in all thread blocks can access global memory locations. The threads hide memory latency; if a thread stalls waiting for a memory access to finish, another ready-to-execute thread can be scheduled.

A GPU program achieves its highest performance when all the threads execute the *identical* instruction sequence on possibly different data sequences, known as *single instruction stream multiple data stream* (SIMD) parallelism. Different threads are allowed to execute different instruction sequences, but the hardware cannot execute such threads in parallel; they must be executed sequentially, diminishing the program's performance.

3.2 Statistical Test Suite Architecture

Fig. 5 shows the software architecture of the statistical test suite. There are four principal source files written in C for CUDA: EvalFunction, EvalSuperpoly, CryptoFunction, and StatTest.

The EvalFunction program generates data to test the Output Randomness and Independence Hypotheses for a certain cryptographic function. The program performs a specified number n of trials, one at a time. For each trial, the program does a specified number ℓ of cryptographic function evaluations $F(x_t), F(x_t + 1), \ldots, F(x_t + \ell - 1)$ (see Fig. 1) in parallel on the GPU. Each evaluation is performed in a single, separate thread. For each evaluation, the thread generates the input value, stores it in a shared memory table, executes the cryptographic function which stores the output value back into the shared memory table, and copies the output value into a global memory table. (The code for the cryptographic function itself is in a separate source file as described later.) When all the threads have finished, the program copies the trial's results from the GPU's global memory to the CPU's memory, then writes the results to a file. The program

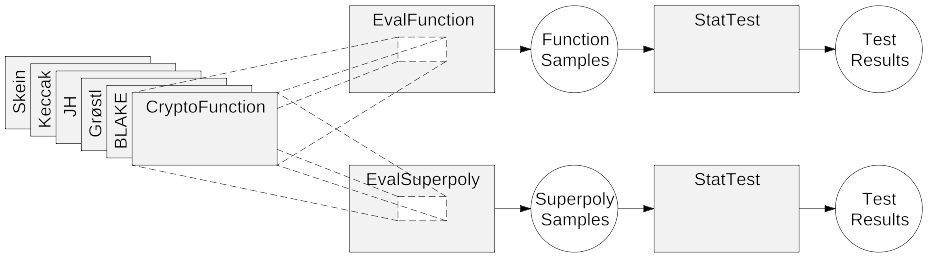

Fig. 5 Statistical test suite software architecture

flushes the file to disk after each trial in order to checkpoint the computation. If the program aborts for some reason, upon restarting it can read the file to see how far it got and resume the trials from that point.

The EvalSuperpoly program generates data to test the Superpoly Randomness and Independence Hypotheses for a certain cryptographic function. The program examines numbers of cube bits c within a specified range, numbers of superpoly bits s within a specified range, and a specified number of trials per (c, s) combination, one trial at a time. For each trial, the program does a specified number ℓ of superpoly evaluations—requiring $2^c \cdot \ell$ cryptographic function evaluations, see (11)—in parallel on the GPU. Each evaluation is performed in a single, separate thread. For each evaluation, the thread generates the input value, stores it in a shared memory table, and executes the cryptographic function which stores the output value back into the shared memory table. The threads in the thread block that are working on the same superpoly computation then synchronize with each other and do a shared memory exclusive-or parallel reduction to sum their results together. One thread adds this partially-reduced result into a global memory table using an atomic exclusive-or operation. When all the threads have finished, the global memory table holds the fully-reduced superpoly results. The rest of the EvalSuperpoly program is the same as the EvalFunction program (they both use the same output file format).

The code for the cryptographic function being evaluated resides in a separate CryptoFunction source file that is included into the EvalFunction and EvalSuperpoly programs. For this study, five CryptoFunction source files implemented the BLAKE, Grøstl, JH, Keccak, and Skein hash functions.

Each CryptoFunction source file evaluates the hash function in a single thread, with the algorithm's state preferentially stored in multiprocessor registers (local variables). Although each of the five hash algorithms studied exhibits some degree of internal parallelism, running time measurements demonstrated that using multiple threads to compute one evaluation invariably took longer than using a single thread. There are several reasons for this. With multiple threads computing one evaluation, the algorithm's state cannot be stored in the fast registers; instead, it must be stored in the slower shared memory so all the threads can access it. Also, the multiple threads computing one evaluation typically must synchronize with each other at several points in the algorithm, further increasing the running time. Finally, the different threads computing one evaluation typically must execute different instruction sequences at various points in the algorithm, eliminating SIMD parallelism at those points and increasing the running time. When there is only one thread per evaluation and different threads compute different evaluations, each thread's state does not need to be shared and so can be stored in fast registers, the threads do not need to synchronize with each other, and every thread executes the identical instruction sequence.

Each CryptoFunction source file also specifies the number of threads per thread block, NT, to use when computing the particular hash function's evaluations in parallel on the GPU. Choosing the proper NT is crucial for achieving the best performance. If NT is too small, there are not enough threads in the thread block to hide the memory access latency, and the running time increases. If NT is too large, the multiprocessor does not have enough on-chip resources to accommodate all the threads—in particular, if all the threads' local variables do not fit in the fast register file, some of the threads' local variables are placed in the slow global memory instead—and the running time increases.

The final program, StatTest, reads the EvalFunction or EvalSuperpoly program's output file, performs the statistical tests on all the samples for all the trials as shown in Fig. 2, and prints the test results. For each trial, the StatTest program reads the trial's samples from the file, copies them from the CPU's main memory to the GPU's global memory, and executes a series of parallel kernels on the GPU:

– For each output bit, each statistical test (frequency test through Hamming-weight test), and each sam-

ple, the appropriate statistical test bin stored in a global memory table is incremented in parallel.
- For each output bit and each statistical test (frequency test through Hamming-weight test), the statistic and p-value are computed in parallel and stored in global memory tables.
- For each output bit pair and each sample, the appropriate independence test bin stored in a global memory table is incremented in parallel.
- For each output bit pair, the independence test statistic and p-value are computed in parallel and stored in global memory tables.

After executing the above kernels, the StatTest program copies the bins, statistics, and p-values from the GPU's global memory to the CPU's main memory, then prints some or all of the results (as specified by command line arguments). After processing all the trials, the StatTest program executes another series of parallel kernels on the GPU:

- For each output bit and each statistical test (frequency test through Hamming-weight test), the uniformity test statistic and p-value are computed in parallel and stored in global memory tables.
- For each output bit pair, the independence test's uniformity test statistic and p-value are computed in parallel and stored in global memory tables.

After executing the above kernels, the StatTest program copies the uniformity test statistics and p-values from the GPU's global memory to the CPU's main memory, then prints some or all of the results (as specified by command line arguments). Finally, for each statistical test, the StatTest program computes (on the CPU) and prints the overall uniformity test statistic and p-value.

3.3 Hash Algorithm Implementations

For this study, the SHA-3 finalist candidate hash algorithms were implemented with a fixed input message size of 256 bits and a fixed output digest size of 256 bits. The variants recommended by the developers for 256-bit SHA-3 were implemented.

For each hash algorithm, the number of threads per thread block, NT, to use when computing the evaluations in parallel on the GPU was determined by running time measurements. Table 1 gives the running time in microseconds per evaluation for each hash function, along with the value of NT that minimized the running time. The running time per evaluation is the time required to compute a large number of evaluations in parallel on the GPU, multiplied by the number of cores on the GPU (448), divided by the number of evaluations. This estimates the time it would take to compute the 256-bit digest of one 256-bit message on one GPU core at the NVIDIA Tesla C2050 GPU's processor clock frequency of 1.15 GHz.

Table 1 Running times for GPU parallel hash algorithm implementations

Algorithm	Running time (μsec/eval)	NT
BLAKE	1.94	64
Grøstl	424	512
JH	21.2	128
Keccak	39.0	64
Skein	6.63	64

The following hash algorithm variants were implemented:

BLAKE [3]. Developed by Aumasson, Henzen, Meier, and Phan, BLAKE uses the HAIFA hash function construction [8] with a local wide-pipe compression function derived from the ChaCha stream cipher [4]. This study tested the BLAKE-256 variant, which uses a 512-bit message block, a 256-bit chaining value, and a 512-bit state inside the local wide-pipe compression function. BLAKE also provides for a 128-bit salt input, but the salt was not used in this study.

Since BLAKE's compression function requires no table lookups, the BLAKE hash algorithm can be computed entirely in registers on the GPU and runs very quickly on the GPU.

Grøstl [12]. Developed by Gauravaram, Knudsen, Matusiewicz, Mendel, Rechberger, Schläffer, and Thomsen, Grøstl is a wide-pipe hash function with a compression function built from two permutations dubbed P and Q, followed by an output transformation built from permutation P. The design of the permutations P and Q is based on the design of the AES block cipher [1]. This study tested the Grøstl-256 variant, which uses a 512-bit message block and a 512-bit chaining value.

The Grøstl hash algorithm proved difficult to implement so as to obtain high performance on the GPU. Like AES, Grøstl generally achieves its best performance in software when implemented using table lookups. Several implementations of Grøstl were tested, and a table-lookup-based implementation similar to the Grøstl designers' optimized 32-bit implementation did indeed yield the smallest running time. Replacing some of the table lookups with calculations caused the running time to increase. However, because table lookups from global memory are slow on the GPU, Grøstl's running time on the GPU is much larger than the other hash functions' running times.

JH [24]. Developed by Wu, JH uses a variation of the sponge construction [5,6]. JH's compression function uses a permutation, dubbed E_8, whose round function is built from a substitution layer, a maximum distance separable code mixing layer, and a bit transposition layer. This study tested the JH-256 variant, which uses a 512-bit message block and a 1024-bit chaining value.

JH's compression function can be computed mostly in registers on the GPU, with round constants looked up from a global memory table indexed by the round number. Unlike Grøstl, which accesses its tables in a data-dependent (*i.e.,* random) pattern, JH accesses its tables in a data-independent, regular pattern. This access pattern proves to be more efficient in the GPU hardware, resulting in a smaller running time for JH than for Grøstl. However, because table lookups from global memory are slow on the GPU, JH's running time on the GPU is larger than BLAKE's and Skein's running times.

Keccak [7]. Developed by Bertoni, Daemen, Peeters, and Van Assche, Keccak uses the sponge construction [5,6]. All the variants of Keccak recommended for SHA-3 use the same permutation, dubbed Keccak-f, whose round function is built from five layers that perform various linear and nonlinear mappings. This study used the $\lfloor \text{Keccak}[1088, 512] \rfloor_{256}$ variant, which uses a 1088-bit message block and a 1600-bit chaining value, and which generates a 256-bit hash digest.

Keccak's compression function has the largest state size (twenty-five 64-bit variables; 1600 bits) of all the SHA-3 finalist hash algorithms. Consequently, due to limitations on the number of local variables in GPU kernel functions, the state variables had to be allocated in shared memory rather than registers. This causes Keccak's running time on the GPU to be longer than that of BLAKE, JH, and Skein. However, Keccak's running time is much smaller than that of Grøstl because the shared memory, where Keccak's state was stored, is much faster than the global memory, where Grøstl's lookup tables were stored.

Skein [13]. Developed by Ferguson, Lucks, Schneier, Whiting, Bellare, Kohno, Callas, and Walker, Skein is a flexible cryptographic primitive that can be configured to operate as a hash function, MAC, stream cipher, pseudorandom number generator, and others. At its core is a new block cipher, Threefish [13], offering large block and key sizes of 256, 512, and 1024 bits. When operated as recommended for SHA-3, Skein is a wide-pipe hash function whose compression function uses Threefish in Matyas-Meyer-Oseas mode [18], followed by an output transformation that also uses Threefish. This study tested the Skein-512-256 variant, which uses a 512-bit message block and a 512-bit chaining value with 512-bit Threefish.

Since Skein's compression function requires no table lookups, the Skein hash algorithm can be computed entirely in registers on the GPU and runs very quickly on the GPU. Skein is slower than BLAKE on the short messages this study tested because Skein requires two applications of the compression function (one to absorb the input message, one to generate the hash digest) while BLAKE requires only one; and because Skein's compression function has more rounds than BLAKE's (72 versus 14 rounds), although each Skein round is simpler.

4 Statistical Test Results

4.1 Overall P-Values

The statistical test methodology described in Sect. 2 was applied to the 256-bit versions of the five SHA-3 finalist candidate hash algorithms described in Sect. 3. In the Output Randomness Hypothesis and Output Independence Hypothesis tests, the same starting input values x_t were used in each trial for every hash algorithm. In the Superpoly Randomness Hypothesis and Superpoly Independence Hypothesis tests, the same cube input bits y, the same superpoly input bits z, and the same starting superpoly input values z_t were used in each trial for every hash algorithm. Fig. 6(a) lists the overall p-value (refer to Fig. 2) for each statistical test, computed from the function samples for each hash algorithm. Fig. 6(b) lists the overall p-value for each statistical test, computed from the superpoly samples for each hash algorithm.

Nonrandom behavior was observed in the BLAKE, JH, Keccak, and Skein hash algorithms, as evinced by the following statistical test failures. Failure is defined to be an overall p-value at or below a significance of 0.01 (the default significance in the NIST test suite [20]).

- BLAKE, function samples, serial-2 test, p-value = 0.00876
- JH, function samples, independence test, p-value = 0.00053
- Keccak, superpoly samples, serial-2 test, p-value = 0.00557
- Skein, superpoly samples, independence test, p-value = 0.00067

Nonrandom behavior was not observed in the Grøstl hash algorithm; every statistical test's overall p-value was above a significance of 0.01.

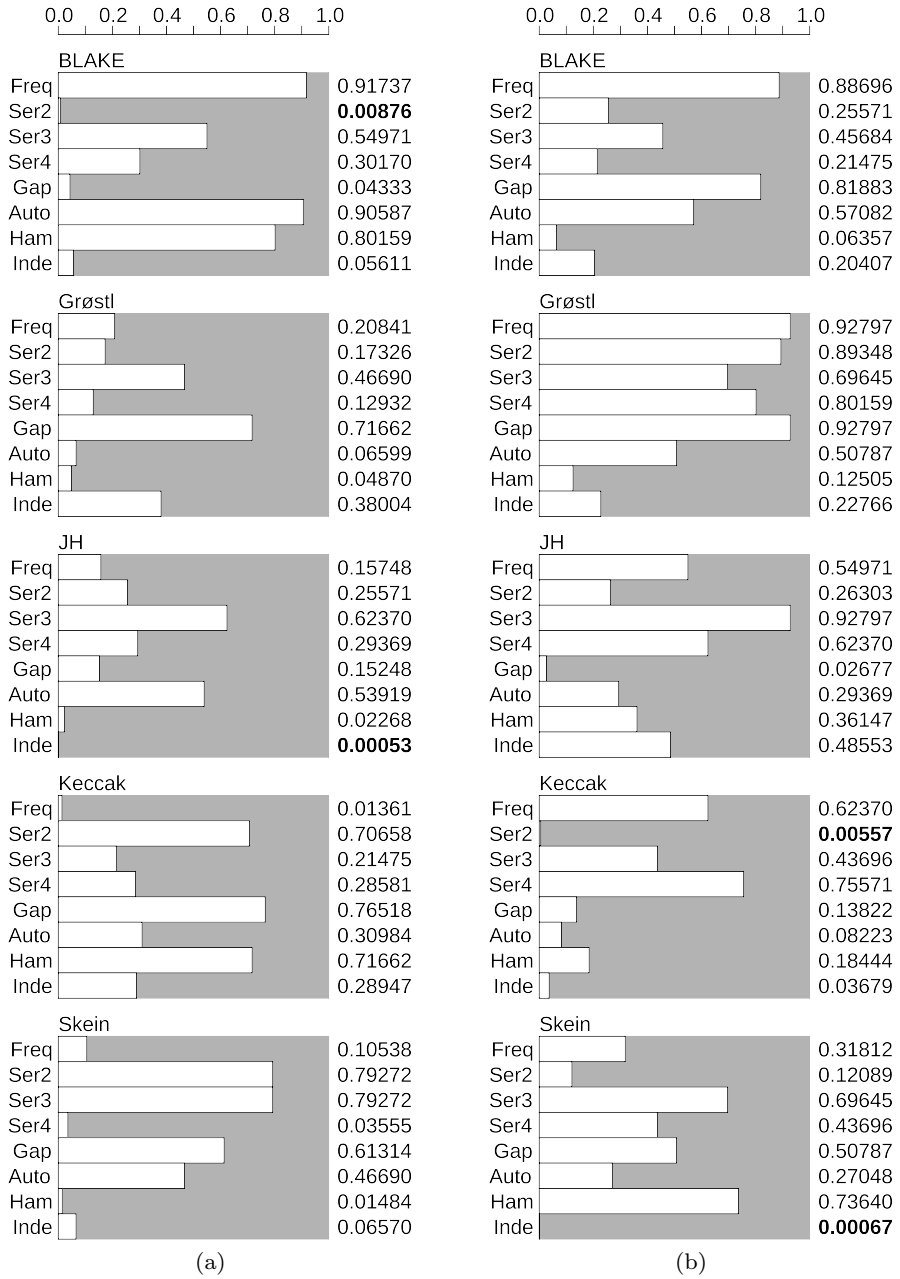

Fig. 6 Results of statistical tests on the SHA-3 finalist candidate hash algorithms. The overall p-value is listed for each hash algorithm and statistical test (see Fig. 2). P-values less than or equal to a significance of 0.01, indicating nonrandom behavior, are shown in **bold.** (a) Results for function samples. (b) Results for superpoly samples

4.2 Statistical Test Failures

Fig. 7 focuses more closely on the aforementioned statistical test failures.

Consider the serial-2 test on the BLAKE function samples. 2000 trials were performed from different starting points, each trial yielding a 10240-sample binary sequence for each of the 256 hash output bits. In the first step of the statistical analysis, for each output bit and each trial, the serial-2 test was applied to the binary sequence, yielding a p-value. In the second step, for each output bit, a 20-bin chi-square uniformity test was applied to the 2000 trials' p-values, yielding another p-value. In the third step, a 20-bin chi-square uniformity test was applied to the 256 output bits' p-values, yielding the final overall p-value listed in Fig. 6(a). Fig. 7(a) shows the bins for this final uniformity

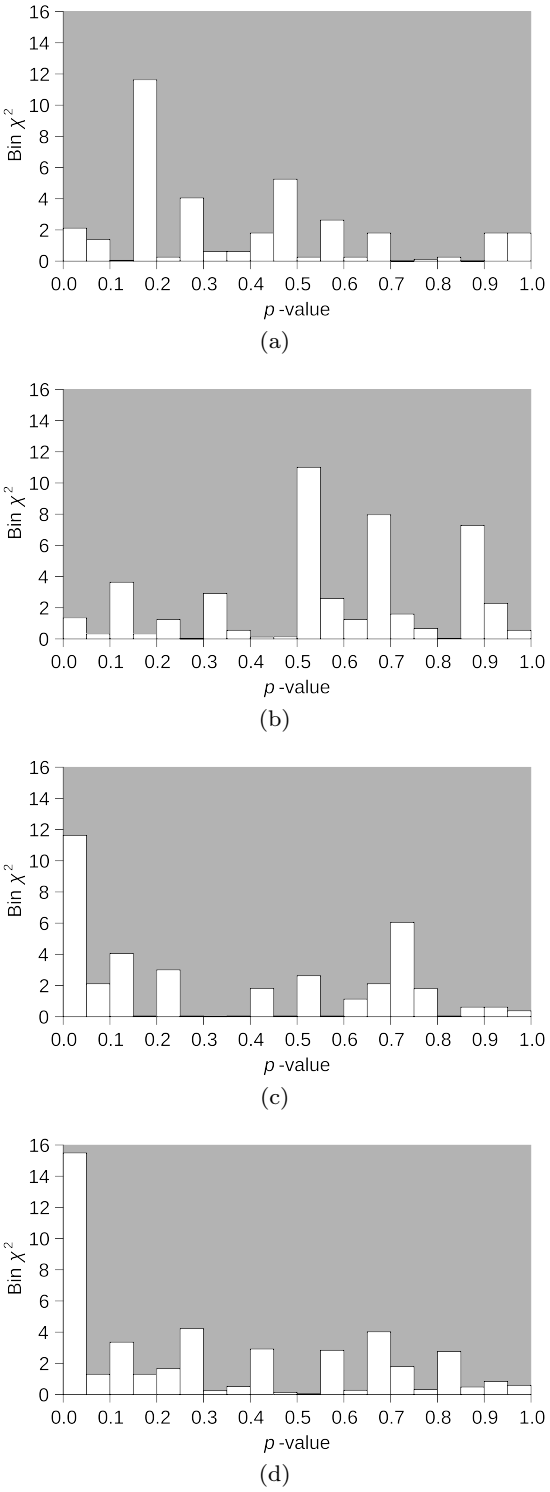

Fig. 7 Histograms of χ^2 values for overall uniformity test bins for: (a) BLAKE, function samples, serial-2 test; (b) JH, function samples, independence test; (c) Keccak, superpoly samples, serial-2 test; (d) Skein, superpoly samples, independence test

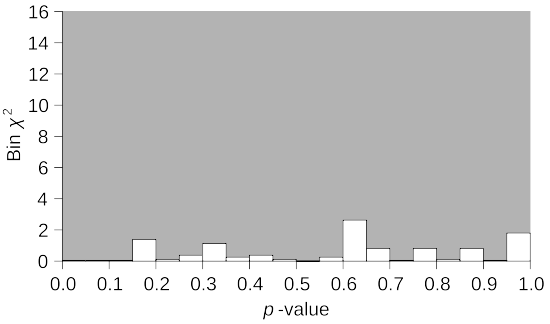

Fig. 8 Histogram of χ^2 values for overall uniformity test bins for BLAKE, function samples, frequency test

test. The horizontal axis plots the 20 bins. The vertical axis plots the quantity

$$\frac{(N_i - Np_i)^2}{Np_i} \tag{12}$$

for each bin, which is the bin's contribution to the χ^2 statistic (2). If the Output Randomness Hypothesis—that the output bits are uniformly distributed binary random variables—is true for the BLAKE hash algorithm, then the output bits' p-values should be uniformly distributed in the $(0, 1)$ interval, and the quantity (12) should be close to zero for each bin. Instead, Fig. 7(a) shows that the output bits' p-values are far from being uniformly distributed, leading to a large χ^2 value and a p-value below the significance threshold of 0.01. This disproves the Output Randomness Hypothesis for the BLAKE hash algorithm (at a significance of 0.01).

Likewise, Fig. 7(b) shows how the independence test disproves the Output Independence Hypothesis for the JH hash algorithm. Fig. 7(c) shows how the serial-2 test disproves the Superpoly Randomness Hypothesis for the Keccak hash algorithm. Fig. 7(d) shows how the independence test disproves the Superpoly Independence Hypothesis for the Skein hash algorithm.

For comparison, Fig. 8 shows the bins for the frequency test on the BLAKE hash algorithm's function samples, a test that did not fail at a significance of 0.01 (p-value = 0.91737). The bins' χ^2 values are all near zero, showing that the trials' p-values fall close to a uniform distribution.

4.3 Related Work

Doğanaksoy, Ege, Koçak, and Sulak [11] applied their statistical test methodologies of [10] and [22] to the 14 SHA-3 second-round candidate hash algorithms. They examined only the compression functions, not the complete hash algorithms, and they examined only reduced-

round versions of the compression functions to determine the minimum number of rounds needed to pass all the tests. They found that the Fugue, Hamsi, and Shabal hash algorithms' compression functions failed to pass some of the tests even with the full number of rounds. However, those algorithms were eliminated during the second round of the SHA-3 competition. They found that every SHA-3 finalist candidate hash algorithm's compression function passed all the tests with a reduced number of rounds. As in their previous papers, they did not report results for the full-round compression functions. In contrast, this paper reports results for the full hash algorithms, not just the compression functions; this paper reports results for hash algorithms with the full-round compression functions; and this paper applies a different methodology with different statistical tests from [11].

In a precursor to the work reported in this paper, Kaminsky [14] applied three statistical tests—balance test, off-by-one test, and independence test—to binary sequences computed from superpolys of the SHA-3 second-round candidate hash algorithms CubeHash and Skein. The balance test is the same as this paper's frequency test; the independence test is the same as this paper's; the off-by-one test was not used in this paper. The balance test and off-by-one test did not reveal nonrandom behavior at a significance of 0.001 in CubeHash or Skein. The independence test did reveal nonrandom behavior at a significance of 0.001 in both CubeHash and Skein. Those results agree with this paper's results for Skein's superpoly samples.

Wang and Wang [23] used the NIST statistical test suite [20] to test the randomness of BLAKE-32 (the 256-bit version submitted to the second round of the SHA-3 competition), operating as a pseudorandom number generator in the following manner. Given a seed value S, the hash of a message consisting of the message blocks $(S, S+1, S+2, \ldots)$ is computed, and the concatenation of all the intermediate chaining values (outputs of the BLAKE compression function) is taken as the output pseudorandom sequence. They used this process to generate one billion pseudorandom bits, partitioned them into 1000 one-million-bit sequences, and applied the NIST test suite to these sequences. The smallest overall uniformity test p-value observed for any statistical test was 0.007584; all the others were greater than 0.01. In contrast, this paper studies the SHA-3 finalist candidate algorithm, BLAKE-256; this paper analyzes binary sequences derived from the external hash digests, not the internal chaining values; and this paper applies a different methodology with different statistical tests from [23]. Still, both [23] and this paper observed one statistical test failure at a significance of 0.01 in the respective BLAKE variants studied.

5 Conclusion

Applying statistical tests of the hypotheses that the output bits and superpoly bits of the cryptographic functions are uniformly distributed, independent, binary random variables, nonrandom behavior was observed at a significance of 0.01 in the 256-bit versions of the SHA-3 finalist candidate hash functions BLAKE, JH, Keccak, and Skein. Nonrandom behavior was not observed at a significance of 0.01 in the 256-bit version of the SHA-3 finalist candidate hash function Grøstl. The statistical tests were performed on a 448-core NVIDIA Tesla C2050 GPU supercomputer; without the high degree of parallel processing afforded by the GPU, the statistical tests—especially the cube tests—could not have been completed in a practical amount of time.

This paper's results show the importance of performing statistical tests on both the function samples and the superpoly samples of a cryptographic function. Two of the hash functions, BLAKE and JH, exhibited nonrandom behavior in the function samples but not in the superpoly samples. Two other hash functions, Keccak and Skein, exhibited no nonrandom behavior in the function samples but did exhibit nonrandom behavior in the superpoly samples. If the superpoly samples had not been tested, the nonrandom behavior in the latter two functions would not have been observed.

This paper makes no claims about the SHA-3 finalist candidate hash functions' security based on the statistical test results. No one knows at this time how to translate, say, a failure of the independence test on a hash function into, say, a breach of a digital signature scheme using that hash function. However, this paper's results might suggest caution when using certain hash functions as high-quality cryptographic pseudorandom number generators.

Future work includes adding more statistical tests to the methodology; generating binary sequences by applying the cryptographic function to other sequences of input values, such as off-by-one sequences; investigating the statistical behavior of additional SHA-3 finalist candidate hash function variants, such as the 224-, 384-, and 512-bit versions, and other cryptographic functions using the methodology; and determining whether certain aspects of a hash algorithm's design correlate with the hash algorithm's statistical behavior.

The program source files and analysis output files described in this paper are available on the

author's web site at http://www.cs.rit.edu/~ark/parallelcrypto/sha3test01/.

Acknowledgements I would like to thank Michael Kurdziel, Marcin Łukowiak, and Stanisław Radziszowski for their helpful comments on earlier drafts of this paper.

References

1. Advanced Encryption Standard (AES). FIPS Publ. 197 (2001)
2. Aumasson, J., Dinur, I., Meier, W., Shamir, A.: Cube testers and key recovery attacks on reduced-round MD6 and Trivium. In: Dunkelman, O. (ed.) Fast Software Encryption—FSE 2009. LNCS 5665, 1–22. Springer, Heidelberg (2009)
3. Aumasson, J., Henzen, L., Meier, W., Phan, R.: SHA-3 Proposal BLAKE, Version 1.3. http://131002.net/blake/blake.pdf (2010). Accessed 13 July 2011
4. Bernstein, D.: ChaCha, a Variant of Salsa20. http://cr.yp.to/chacha/chacha-20080128.pdf (2008). Accessed 13 July 2011
5. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the indifferentiability of the sponge construction. In: Smart, N. (ed.) Advances in Cryptology—EUROCRYPT 2008. LNCS 4965, 181–197. Springer, Heidelberg (2008)
6. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Cryptographic Sponge Functions, Version 0.1. http://sponge.noekeon.org/CSF-0.1.pdf (2011). Accessed 13 July 2011
7. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The Keccak Reference, Version 3.0. http://keccak.noekeon.org/Keccak-reference-3.0.pdf (2011). Accessed 13 July 2011
8. Biham, E., Dunkelman, O.: A framework for iterative hash functions—HAIFA. Cryptology ePrint Archive, Report 2007/278 (2007)
9. Dinur, I., Shamir, A.: Cube attacks on tweakable black box polynomials. In: Joux, A. (ed.) Advances in Cryptology—EUROCRYPT 2009. LNCS 5479, 278–299. Springer, Heidelberg (2009)
10. Doğanaksoy, A., Ege, B., Koçak, O., Sulak, F.: Cryptographic randomness testing of block ciphers and hash functions. Cryptology ePrint Archive, Report 2010/564 (2010)
11. 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)
12. Gauravaram, P., Knudsen, L., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.: Grøstl—a SHA-3 Candidate, Version 2.0.1. http://www.groestl.info/Groestl.pdf (2011). Accessed 13 July 2011
13. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein Hash Function Family, Version 1.3. http://www.skein-hash.info/sites/default/files/skein1.3.pdf (2010) Accessed 13 July 2011
14. Kaminsky, A.: Cube test analysis of the statistical behavior of CubeHash and Skein. Cryptology ePrint Archive, Report 2010/262 (2010)
15. Knuth, D.: The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3rd ed. Addison-Wesley, Boston (1998)
16. L'Ecuyer, P., Simard, R.: TestU01: a C library for empirical testing of random number generators. ACM Trans. Math. Softw. 33, 22 (2007)
17. National Institute of Standards and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. Fed. Regist. 72, 62212–62220 (2007)
18. Matyas, S., Meyer, C., Oseas, J.: Generating strong one-way functions with cryptographic algorithms. IBM Tech. Discl. Bull. 27, 5658–5659 (1985)
19. Press, W., Teukolsky, S., Vetterling, W., Flannery, B.: Numerical Recipes: The Art of Scientific Computing, 3rd ed. Cambridge University Press, Cambridge (2007)
20. 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 (2010)
21. The SHA-3 Zoo. http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo (2011). Accessed 13 July 2011
22. 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 6338, 309–319. Springer, Heidelberg (2010)
23. Wang, H., Wang, H.: A fast pseudorandom number generator with BLAKE hash function. Wuhan Univ. J. Nat. Sci. 15, 393–397 (2010)
24. Wu, H.: The Hash Function JH. http://www3.ntu.edu.sg/home/wuhj/research/jh/jh_round3.pdf (2011). Accessed 13 July 2011