NUMBER 8 — SEPTEMBER 3, 1998

RSA Laboratories' Bulletin

News and advice on data security and cryptography

Preliminary Analysis of the BSAFE 3.x Pseudorandom Number Generators

Robert W. Baldwin
RSA Data Security
San Mateo, California

Abstract

An enormous number of commercial applications (over 350 million copies) rely on the BSAFE and JSAFE toolkits from RSA Data Security to generate cryptographically strong pseudorandom numbers for keys, initialization vectors, challenges, etc. This paper describes the algorithms used by these toolkits, discusses their design, analyzes their resistance to various attacks, and presents results from statistical tests. The algorithms appear to be well suited for cryptographic applications.

Introduction & Background

The amazing feature of cryptography is that it reduces the problem of protecting a large amount of data to the problem of protecting a small amount of keying material. However, generating even a small amount of keying material is hard. The trouble is that gathering good randomness (bits that cannot be predicted or influenced by an attacker) can take several thousand milliseconds, which is unacceptable for most applications. The usual solution is to rely on a good pseudorandom number generator (PRNG) to quickly produce keying material from an initial seed of good randomness.

This paper presents a preliminary analysis of two PRNGs called MD5Random and SHA1Random that are based on the MD5 [15] and SHA1 [14] digest functions. These PRNGs are implemented in the JSAFE and BSAFE cryptographic toolkits from RSA Data Security. These algorithms are used in over 350 million copies of commercial software to generate keys (both public key pairs and symmetric keys), cipher initialization vectors, and random values for challenge-response protocols.

After describing the algorithms, we discuss their design and identify various assumptions that are made about the properties of the underlying digest functions. The discussion explains why these PRNG are believed to be cryptographically strong. The next section discusses how the generators resist various attacks in order to explore the implications of the design choices. The attack section is similar to the analysis reported in [9] for other PRNG algorithms. The statistical properties of the PRNG are then presented based on both classical randomness tests [2, 6, 7, 8, and 10] and the Marsaglia Diehard tests [12].

Algorithm Description

These PRNGs are defined as abstract objects with three operations:

The `B_RandomInit` operation creates the PRNG algorithm object by allocating and zeroing the state and buffers plus initializing the related digest object (MD5 or SHA1).

The `B_RandomUpdate` operation adds seed bytes to initialize or update the state. This is called initially before generating output, and may be called after producing output in order to add more seeding to the state. For backward compatibility with BSAFE versions 1 and 2, an algorithm similar

Robert Baldwin is a Technical Director at RSA Data Security; he can be reached at baldwin@rsa.com or baldwin@lcs.mit.edu.

RSA Laboratories
A Division of RSA Data Security

to the one in RSAREF [17] is used if `B_RandomUpdate` is only called once before calling `B_GenerateRandomBytes`. This paper describes the generation algorithm used when `B_RandomUpdate` is called two or more times before the first generate call.

The `B_GenerateRandomBytes` operation returns a sequence of generated bytes and updates the internal state and buffers as needed.

Operators:
"+" is unsigned integer addition
"*" is unsigned integer multiplication
"**" is unsigned integer exponentiation
"||" is concatenation
"| x |" is the length of x in bits

Variables:
X_j where $j > 0$ is a sequence of seed material blocks passed to one or more state update operations between output generation operations. For example, X_1 contains all the bytes passed via all the calls to `B_RandomUpdate` before the first call to `B_GenerateRandomBytes`. The length of X_j is an arbitrarily long multiple of 8 bits.

Y_{ji} is the ith block of output from the generator after seeding with blocks X_1 through X_j.

S_{ji} is the state used to generate Y_{ji}.

$L = |Y_{ji}| = |S_{ji}|$ is 128 bits for MD5Random and 160 bits for SHA1Random.

$H(x)$ is the digest of x, which is either MD5 (x) or SHA1 (x).

C = Odd (H ("")) is an odd L-bit constant computed by hashing a zero length bit string and setting the least significant (right most) bit of the result to one.

Algorithm Initialization (`B_RandomInit`):
$j = 1$, $i = 0$, and buffered output is cleared

State updating (`B_RandomUpdate`):
$S_j = H(S_{j-1} \,||\, X_j)$, $j = j + 1$, $i = 0$
where S_0 = zero length bitstring and buffered output is cleared

The value of the X_j seed block can be passed to the PRNG using one or more calls to the BSAFE `B_RandomUpdate` function. The resulting new state does not depend on how the X_j bytes are chopped into sub-blocks passed to the individual update calls. In effect, the PRNG buffers all the seed bytes in X_j until the next call to the generate function. In practice, only the most recent 64 bytes (512 bits) are buffered since the rest are run through the digest's compression function.

Output generation (`B_GenerateRandomBytes`):
$Y_{ji} = H(S_{ji})$, $i = i + 1$, $S_{ji} = (S_j + (C * i)) \bmod 2^L$

The PRNG returns successive bytes of the Y_{ji} values and it buffers unused bytes of the Y_{ji} to satisfy future calls to `B_GenerateRandomBytes`. When the buffer is exhausted, the parameter, i, is incremented and new values for S_{ji} and Y_{ji} are generated. Calling `B_RandomUpdate` resets the parameter, i, to zero.

Design Goals, Assumptions and Considerations

The primary goals for these PRNG algorithms are listed below.

Algorithm Goals

1. Output indistinguishable from true random sequence.
2. Knowledge of some outputs does not help predict future or past outputs.
3. Make good use of the "entropy" in the seeding material.
4. Guaranteed long cycle length.
5. Sufficiently large internal state to avoid exhaustive search.
6. Good performance.
7. Simple algorithm.

The goals provide different yard sticks for measuring the algorithm, but do not provide well-defined objectives that must be met. The objectives were:

Algorithm Objectives

1. Allow re-seeding interspersed with output generation.
2. Generated bytes depend on all preceding seed bytes.
3. State depends on order of the seeding bytes passed to update function.
4. State does not depend on how seed bytes are chopped into buffers that are passed to the update function. For example, calling `B_RandomUpdate` twice with the seeds "*ab*"

and "*cdef*" produces the same state as passing "*abc*" then "*def*". This property simplifies the analysis of the PRNG seeding.

5. State update and output generation runs in constant time to avoid timing attacks.
6. Generated sequence does not depend on the order in which groups of bytes are requested. For example, if two generators start in the same state and the first one is asked to return two bytes, then asked to return four bytes, the same six byte sequence is returned as would be by asking the second generator to return five bytes and then one byte. This property simplifies the analysis of the PRNG output.
7. Use at most one underlying digest algorithm object.
8. Avoid the use of encryption functions to simplify export compliance issues.

Design Discussion

It would be nice to design a PRNG based on structures with provable security properties. For example, the ability to distinguish the PRNG output from truly random bytes should be related to some properties of the digest function or its underlying compression operator. Unfortunately, we did not know of any such structures or proofs.

However, there are several features in the design of these algorithms that are intended to rely on accepted features of digest functions to thwart cryptanalysis. For example, digest functions are assumed to be hard to invert, so given the output Y_{ji} it is cryptographically difficult to find any value, Z, such that $Y_{ji} = H(Z)$. This means that finding the specific L-bit state value, S_{ji}, that generated Y_{ji}, should be hard. Thus one of the design assumptions is that this hard to invert property is true even if the input values to H are restricted to the 2^L possible state values.

The state update formula, $S_{ji} = (S_j + (C * i)) \bmod 2^L$, was chosen to avoid possible related input attacks against the digest function by changing a large and irregular number of bits between successive state values. The motivation for this comes from viewing the compression function as an encryption operator with a 512-bit key and an L-bit input. When the input to MD5 or SHA1 is less than 56 bytes, as is the case with these algorithms, the digest function can be defined as:

$$H(S_{ji}) = IV + G(IV, S_{ji} \,||\, \text{zeros} \,||\, LC)$$

Where $G(M, K)$ is the compression function, IV is a published constant, and LC is an 8-byte value that encodes the length of S_{ji} in bits. The LC value is a constant for these PRNGs. The compression function is assumed to be a good random permutation that maps the L-bit M value into an unrelated L-bit result based on the 512-bit value K. The G function effectively encrypts M using K as the key. The compression functions for MD5 and SHA1 are constructed to be permutations. The "goodness" that is assumed by these PRNG algorithms is that it is cryptographically hard to find K given M and $G(M,K)$.

This design assumed it is hard to find K given a known fixed M and a known output. For example, we assume that there are no statistical characteristics of the output blocks, Y_{ji}, that would help an attacker recognize when a trial value of K is close to the actual S_{ji} value. The design assumes that the fixed known value of M, which is IV, does not help the attacker.

However these generators were designed to avoid excessive reliance on the G function being a perfect pseudorandom permutation. The PRNGs were designed to avoid related key weaknesses in the G function. Published results [16, 4] on the compression function of MD5 and the round functions that are shared between MD5 and SHA1 suggest that this precaution is prudent. Thus one design criteria was to update the generator state, which becomes the varying part of G's K input, in a manner that produces large and irregular Hamming differences between successive K values.

Several ways of updating the state were considered. A linear feedback shift register (LSFR) was considered but eliminated because the first round of SHA1 involves a rolling exclusive-or that is a kind of LSFR. Adding a constant is also a linear operation, but it is over Z_2^L rather than GF_2^{32} as in SHA1, so it was accepted. An odd constant guarantees that the PRNG will have a maximum cycle length, which is of 2^L blocks of L bytes each.

The generators are also designed to prevent an attacker from seeing several M values encrypted with the same K value for the compression function $G(M, K)$. In these algorithms, M is always the known constant IV. If there are cryptographic weaknesses in the G function when viewed as an encryption function, then limiting the number of known plain-

... there are several features in the design of these algorithms that are intended to rely on accepted features of digest functions to thwart cryptanalysis.

text-ciphertext pairs available to the attacker should help. For example, it would be hard to mount a linear or differential cryptanalysis attack against G since each key value is only used once. Of course, if G is viewed as a function that encrypts K with the fixed key M, then many ciphertext blocks are known and the attacker must find one of the matching plaintext blocks. We assume this is hard for MD5 and SHA1.

These PRNGs can also be defined in terms of a pseudorandom function, F, which maps L-bit values to L-bit values, where

$F(x) = G(IV, x \mid\mid \text{zero} \mid\mid LC)$,
where IV and LC are known constants.

The design assumes that virtually all output values are generated by $F(x)$ when x is varied over all L-bit values. Without this assumption, the PRNG would produce a subset of the possible output blocks. That is, F is assumed to be nearly a pseudorandom permutation. This property is not obvious from the construction of G for either SHA1 or MD5, but it appears to follow from the collision resistance properties of the hash functions.

Resistance to Attacks

Exhaustive Seeding Search

All PRNGs can be compromised if the attacker can guess the seed bytes that initialized the generator. The attacker can try all possible seed byte values and check the computed result. The check could be direct against an observed value like a CBC mode initialization vector, or indirect such as using the computed result as a triple-DES key and checking to see if it decrypts a message correctly.

The success of a seeding search attack depends on how the application gathers seeding material and how much access the attacker has to the system while the generators are being seeded. This attack does not depend on the PRNG algorithms. It depends on the volume and quality of the seeding material.

There are many poor sources of seed material, such as clock values, that the attacker can guess, or user keystrokes that can be observed over a network connection. It may also be possible for the attacker to force the system into a known state or at least into a state for which there is less entropy. For example, on a multi-user computer, network statistics are hard to predict if the machine has been running for a long time, but easy to predict if the system is rebooted and the counters are reset to zero.

System statistics bring up another problem. The attacker may be able to directly observe the statistics by having a process running on the machine while the PRNG is being seeded. See [1, 5] for further discussion of good seed gathering.

Exhaustive State Search

Rather than guessing at the seeding material, the attacker can guess at the internal state of the generator. These generators have L bits of state, which is 128 bits for MD5Random and 160 bits for SHA1Random. Exhaustive search of such a large space is considered to be impractical both now and into the foreseeable future.

Despite the impractical nature of these attacks, they do set a theoretical limit on the strength of a cryptographic system that uses these PRNG to generate keys. For example, triple-DES with three keys has 168 bits of key material, so you might assume that an attacker would have to try all 2^{168} keys for exhaustive search. However, if MD5Random generates the triple-DES key, then at most 2^{128} different triple-DES keys are used by the system, so the attacker's work is reduced. The same analysis holds for RSA or DSA key generation. For example, for 2048 bit RSA keys, it is definitely easier to try all 2^{128} states for MD5Random than to factor this large modulus.

Despite the impractical nature of these attacks, they do set a theoretical limit on the strength of a cryptographic system that uses these PRNG to generate keys.

One design criterion was to make the internal state of the PRNG be large enough to make exhaustive state search impractical. The 128 and 160 bit state sizes meet this criterion. For example, a 128 bit state can take on 3 x 10**38 different values. Assuming that an attacker had a billion computers (10**9) that each could try one billion (10**9) states per second running all year long (3 x 10**7 seconds per year), it would take 10**13 years to try all states. By comparison, the visible universe is only about 4 x 10**9 years old.

Pre-computation

An attacker can precompute the Y_{ji} output values produced by different S_{ji} state values. This reduces the time needed to find the state for any observed or deduced Y_{ji} value, but increases the amount of precomputation and storage required.

A space-time trade-off is possible with this attack if the PRNGs are used to produce a long sequence of outputs. The attacker could pre-compute every N^{th} value of the state and wait for one of those to be used.

Again, the large state size makes this attack infeasible.

Malicious Software Attacks

If a PRNG is implemented in software on a device that allows other software to be loaded (e.g., programmable smart cards, PCs, Web Servers, upgradeable cryptographic hardware), then a virus or other malicious software might be able to read the internal state of the PRNG. The state S_{ji} and the unused bytes of Y_{ji} are in memory. The virus might also be able to observe any new seed bytes. A virus that cannot communicate this information to the attacker is harmless, however the impact of being able to communicate is explored in the next two sections.

Compromise Forward Tracking

If the PRNG state is compromised, the attacker can compute future values of the generator by adding the known constant, C. We considered adding an unknown constant, which would in effect increase the size of the state to $2L$ bits, but decided against it for several reasons. First, the state was already large enough to prevent exhaustive search, so a state compromise is unlikely to occur unless a virus is present, in which case the unknown constant would be captured along with the L bits of state. Second, the constant would have to be generated from the same seed material as the initial state. That is, the PRNG would have to generate two L-bit values from the same X_j values. Several systems have attempted to solve this problem by putting fixed values before or after the seed material being digested, but there is no analytic basis for this. In the end, the desire for simplicity eliminated this choice.

The forward tracking of the PRNG is halted when the application adds new seed material, which causes the state to become $H\ (S_{ji}\ ||\ X_{j+1})$, provided that X_{j+1} is not revealed to the attacker. Even if X_{j+1} is not revealed, the attacker may be able to guess the new seed bytes and confirm a guess by observing or deducing actual outputs of the generator as discussed in the section on exhaustive seed searching. For this reason RSA recommends that seed material be added in large blocks with enough unpredictability to thwart an exhaustive seeding search [1].

Compromise Back Tracking

A state compromise allows limited back tracking. The state, $S_{ji} = S_j + C*i$, can be rolled backwards to S_{j0} by subtracting C. The attacker can compute backwards from Y_{ji} to Y_{j0}, but the hash function that relates S_j and S_{j-1} prevents the attacker from going back any further, even if the previous seed bytes X_{j-1} are known since $S_j = H\ (S_{j-1}\ ||\ X_j)$ and H is hard to invert.

An application that wants to prevent backtracking can call `B_RandomUpdate` after each call to `B_GenerateRandomBytes` to force an application of the hash function to the internal state. In this case the generator becomes:

$$Y_j = H\ (S_j) \text{ and } S_{j+1} = H\ (S_j\ |\ X_{j+1})$$

Even if X_j is known, the non-invertability of H makes it cryptographically infeasible for an attacker to find S_j from S_{j+1}.

If the PRNG state is compromised, the attacker can compute future values of the generator by adding the known constant, C.

Input Entropy

Ideally a PRNG would extract as much entropy out of the seeding material as possible. These generators extract at most L bits of entropy due to the use of a digest function. For MD5 and SHA1, L is considered to be large enough to thwart exhaustive state search. We considered using a universal hash function like an LSFR, but ruled against it due to the possibility of an attacker controlling a portion of the seed input as discussed below. For an LSFR it would be easy to compute the impact on the state of changing a portion of the seeding material.

The initial state value is the digest of all the seed material supplied in any number of calls to `B_RandomUpdate` before the first call to `B_GenerateRandomBytes`. Specially:

$$S_1 = H\ (X_1)$$

where X_1 is the concatenation of all the bytes passed to the seeding routine before the first call to the generate routine.

This is an improvement over the BSAFE 2.x algorithm which produces different states based on how the seed material is chopped into the blocks and passed to `B_RandomUpdate`. More seriously, the state for the old PRNGs did not depend on the order in which the blocks were supplied. This problem

was noticed by Paul Kocher while consulting for RSA Labs and was fixed in the next release of BSAFE. It is described further in [11, 1].

The algorithm for adding new seed material into the state after some bytes have been generated is:

$$S_{j+1} = H\,(S_{ji} \mid\mid X_{j+1})$$

This is algorithm links the old state and the new state using the digest function to ensure a cryptographic mixing of the two. The mixing helps the algorithms thwart an attacker who can influence the range of values used for new seed bytes. One effect of this algorithm is that the state of the generator depends on both the seed bytes and the number and location of calls to `B_GenerateRandomBytes` within the stream of seed bytes. For example,

$$S_2 = H\,((H\,(X_1) + n{*}C) \mid\mid X_2)$$

Where n is the number of times the PRNG needed to update its state to generate all the bytes requested by calls to `B_GenerateRandomBytes` before the new block of seeding, X_1, was added.

One alternative design was to make the state be the hash of all the seed bytes seen so far. That is, $S_j = H\,(X_1 \mid\mid X_2 \mid\mid \ldots \mid\mid X_j)$. This links together the seed bytes in the traditional way of digest functions and makes the state independent of calls to the generate function. However, this design was eliminated for implementation reasons. The final step in computing a digest is to append a bit length value and perform the last compression operation. These steps destroy the internal buffers that would be necessary to compute the state after the next block of seeding is added. Retaining the internal buffers would require direct access to the compression function's inputs (G's inputs), which is not possible for hardware implementations of MD5 and SHA1. In order to allow BSAFE to work with hardware co-processors, the PRNGs algorithms could not be based on internal access to the digest functions, so this design was eliminated.

Chosen Seed Input

An attacker may be able to control some number of the seed bytes. What advantage could this give? For example, the attacker could force a web server to reboot by crashing it, and any seed bytes that were based on system counters would be predictable. The algorithm used to reduce seed bytes to internal state involves a cryptographic hash function, which is assumed to have the property that small changes to the input will produce large and unpredictable changes in the output. As long as the uncontrolled seed bytes provide a sufficiently large amount of unpredictability, this attack is not effective.

Cycle Shortening

Without adding new seed material, the generators will have a cycle length of 2^L L-bit blocks due to the additive constant C. To shorten the cycle length, the attacker would need to influence the new seed material in specific ways. For example, if an attacker can find a value Z such that $S_1 = H\,(S_1 \mid\mid Z)$, then the cycle length will drop to one L-bit block. The cryptographic properties of digest functions make this attack infeasible. The collision free property, which is assumed for strong digest functions, means that it is infeasible to find any values $Z1$ and $Z2$ such that $H\,(Z1) = H\,(Z2)$. In this case, $S_1 = H\,(X_1)$ so the attacker must solve $H\,(X_1) = H\,(H\,(X_1) \mid\mid Z)$, which seems harder even with a known X_1 value.

Timing Attacks

A Kocher-style [11] timing attack can be mounted against a PRNG if the running time of its operations, either seed updating or output generation, depends on the input or state values. The implementations of these PRNG in BSAFE and JSAFE take constant time, so they are not susceptible to timing attacks. In particular, the operation that adds the large odd constant to the state after each output block always executes the same number of instructions regardless of the input values. There are no optimizations for operands or carry bits that are zero.

Overall, the algorithms appear to be well suited for applications that require a pseudorandom number generator with good cryptographic properties.

Summary of Cryptanalysis Discussion

The different attacks discussed above illustrate features and limitations of these algorithms. Overall, the algorithms appear to be well suited for applications that require a pseudorandom number generator with good cryptographic properties.

Statistical tests

The previous sections have discussed the cryptographic features of these PRNG algorithms. Their purpose was to identify the assumptions and design features that support the hypothesis that an attacker cannot accurately predict the actual output of the

PRNGs. This section presents information that supports the hypothesis that the output of these generators is not distinguishable from a truly random sequence. Intuitively, cryptographic strength seems to imply good statistical properties, but it is reassuring to have the results from actual statistical tests.

A separate report [16] describes the details of statistical tests that were performed on MD5Random and SHA1Random. They are summarized in this section. The first tests are classics from [10, 2, 3, 6, and 7]. The second tests come from the Diehard software [12], which was developed by Professor Marsaglia specifically for testing pseudorandom number generators.

Classical Statistical Tests

The following tests are based on performing a pass/fail statistical test on 1000 sequences of 340 bytes each produced by the PRNG. The actual number of samples that pass the tests are compared to the theoretically expected number of passes. For example, on average, 800 random bits should have 400 ones and 400 zeros, but we expect some variation, so we can construct a test that only passes if the number of ones is between 480 and 520. From probability theory we can predict the percentage of samples that will pass this test if the sequences are truly random. In this case the bit count range is one standard deviation, so about 67% should pass. A generator is weak if the actually number of samples passing the test is too high or too low. For example, a generator that produced an alternating sequence of ones and zeros will have the correct average number of ones, but would fail this test, since all of the samples would pass this test when only 67% are expected to pass.

The tests are summarized in the following table and briefly described below. Each entry in the table below gives the percentage of the 1000 sequences that passed each test at the 95% significance level.

The tests can be split into two groups. The first group consists of tests on the appearance of the output from a pseudorandom source:

The ***frequency*** test checks that the balance between 1 and 0 bits lies within acceptable limits.

The ***serial*** test checks that the distribution of pairs of adjacent bits, 00, 01, 10 and 11, is acceptable.

Table 1. *Classical Statistical Tests Results*

	Pass Rates at 5% Significance Level		
Test	**Expected**	**MD5Random Actual**	**SHA1Random Actual**
Frequency	95%	95.2%	95.7%
Serial	95%	95.0%	95.0%
Runs	95%	94.3%	95.8%
Poker (3)	100%	98.8%	99.0%
Poker (4)	98%	96.9%	95.9%
Poker (5)	77%	82.2%	81.3%
Auto-correlation (14)	100%	100.0%	100.0%
Auto-correlation (16)	100%	99.8%	99.5%
Auto-correlation (18)	92%	92.4%	92.9%
Auto-correlation (20)	36%	35.7%	37.4%
Linear Complexity	95%	92.5%	92.4%
Jump Test	95%	95.5%	94.5%
Distribution of Jumps	90.3%	92.4%	90.7%

The ***runs*** test checks that the distribution of runs of lengths m is acceptable, where m varies between 1 and 5. A run is a sequence of 1 or 0 bits. This range of values for m is, again, a function of the number of bits analyzed, though all possible values are tested within the same test giving a single result, unlike the poker test which gives a result for each value of m independently.

The ***poker*** test considers each successive m-bit subsequence (without overlap) and checks that each of the 2^m possible patterns appears an acceptable number of times. The range of values of m for which we perform the poker test is a function of the number of bits in the sequence. In these tests we restricted ourselves to $2 \leq m \leq 5$.

The ***auto-correlation*** test checks that the sequence does not have too much agreement or *correlation* when compared with itself offset by d positions. The auto-correlation was tested for $1 \leq d \leq 20$.

The second set of tests attempts to assess the complexity of some output. They are all based around the use of the *linear complexity* of a sequence and they have been well studied in the literature [3, 10, and 13].

The ***linear complexity*** of a sequence offers some measure of how easy it might be to reproduce some se-

quence of bits. The *linear complexity profile* is obtained by plotting the linear complexity against the number of bits analyzed. The appearance of this profile for a perfectly random source has been evaluated and so a pseudorandom bit generator can be tested against this measure [10, 13. The *linear complexity profile test* counts the number of jumps in the profile of a sequence. The number of jumps is then checked to see that it lies within acceptable bounds for the experiment at the chosen significance level.

The ***jumps*** test is performed only if a sequence passes the linear complexity profile test. Whilst the latter checks that an acceptable number of jumps has occurred in the profile, the jumps test assesses the frequency of the different sizes of these jumps and checks that they lie within acceptable bounds.

In summary, the following linear complexity related tests were performed:

- The *linear complexity* of the sequence was evaluated to see if it lay within acceptable limits.
- The *linear complexity profile* test was conducted to see if the linear complexity profile had an acceptable number of jumps.
- The *jumps* test was completed which tests the distribution of the jumps within the linear complexity profile.

Diehard Statistical Tests

The next sets of tests were designed by Professor Marsaglia to identify weaknesses in many common non-cryptographic PRNG algorithms. They operate on a single large sample from the generator (11 megabytes) that is usually broken into 32-bit words before performing tests. These tests examine the most significant bits of these words, the least significant bits, and inter-bit correlations. The results are expressed in terms of a "*p*" value that should be uniformly distributed between zero and one. Bad PRNGs will produce *p* values that are within 0.00001 of zero or one. The test was run 10 times and the table reports the average and sample standard deviation of the ten results. The average should be 0.5 and the standard deviation should be the square root of one twelfth (about 0.289). A description of the tests appears after the table.

As an example of a bad PRNG, the output of a linear congruent generator (LCG) is included. The equation for this LCG is X' = 65 * X + 3 mod 2**32. The multiplier and additive constants follow the guideline in [10] for maximal length.

The following test descriptions are largely copied from the documentation that accompanies the Diehard program.

Table 2. *Diehard Statistical Tests Results*

Test		Expected	SHA1Random	MD5Random	LCG
Birthday Spacing	Avg:	0.500	0.473	0.450	0.782
	Std Dev:	0.289	0.290	0.284	0.295
5-Permutations	Avg:	0.500	0.475	0.452	1.000
	Std Dev:	0.289	0.357	0.345	0.000
Rank 31x31	Avg:	0.500	0.553	0.626	0.516
	Std Dev:	0.289	0.213	0.231	0.257
Rank 32x32	Avg:	0.500	0.615	0.626	0.419
	Std Dev:	0.289	0.228	0.226	0.014
Rank 6x8	Avg:	0.500	0.490	0.526	0.991
	Std Dev	0.289	0.295	0.294	0.044
Missing 20bit words	Avg:	0.500	0.499	0.495	0.000
	Std Dev:	0.289	0.293	0.286	0.000
Missing 10bit pairs	Avg:	0.500	0.485	0.523	1.000
	Std Dev:	0.289	0.292	0.286	0.000
Missing 5bit quads	Avg:	0.500	0.494	0.491	0.789
	Std Dev:	0.289	0.282	0.279	0.408
Missing 2bit tens	Avg:	0.500	0.518	0.466	1.000
	Std Dev:	0.289	0.281	0.290	0.000
Count-Ones All Bytes	Avg:	0.500	0.444	0.450	1.000
	Std Dev:	0.289	0.275	0.237	0.000
Count-Ones Specific	Avg	0.500	0.562	0.527	1.000
	Std Dev	0.289	0.295	0.305	0.000
Parking Lot	Avg	0.500	0.507	0.460	1.000
	Std Dev	0.289	0.264	0.291	0.000
Minimum Distance	Avg	0.500	0.520	0.611	1.000
	Std Dev	0.289	0.271	0.226	0.000
Smallest 3D Sphere	Avg	0.500	0.508	0.512	0.122
	Std Dev	0.289	0.300	0.293	0.146
Squeeze Iterations	Avg	0.500	0.377	0.531	1.000
	Std Dev	0.289	0.299	0.335	0.000
Overlapping Sums	Avg	0.500	0.476	0.506	0.486
	Std Dev	0.289	0.292	0.300	0.352
Up-Down Runs	Avg	0.500	0.495	0.456	0.692
	Std Dev	0.289	0.283	0.322	0.311
Craps Game	Avg	0.500	0.469	0.576	1.000
	Std Dev	0.289	0.289	0.295	0.000

For the ***Birthday Spacing*** test choose *m* birthdays in a year of *n* days. List the spacing between the birthdays. If *j* is the number of values that occur more than once in that list, then *j* is asymptotically

Poisson distributed with mean m**3/(4n). This test uses n=2**24 and m=2**9, so that the underlying distribution for j is taken to be Poisson with lambda=2**27/(2**26)=2. A sample of 500 j's is taken, and a chi-square goodness of fit test yields a p value. The first test uses bits 1-24 from integers in the specified file. Then the file is closed and reopened. Next, bits 2-25 are used to provide birthdays, then 3-26 and so on to bits 9-32. Each set of bits provides a p-value.

The ***5-Permutations*** tests looks at a sequence of one million 32-bit random integers. Each set of five consecutive integers can be in one of 120 states, for the 5! possible orderings of five numbers. Thus the 5th, 6th, 7th, ... numbers each provide a state. As many thousands of state transitions are observed, cumulative counts are made of the number of occurrences of each state. Then the quadratic form in the inverse of the 120x120 covariance matrix yields a test equivalent to the likelihood ratio test that the 120 cell counts came from the specified (asymptotically) normal distribution with the specified 120x120 covariance matrix (with rank 99). This version uses 1,000,000 integers, twice.

The ***Rank 31x31*** matrix test uses the leftmost 31 bits of 31 random integers from the test sequence to form a 31x31 binary matrix over the field {0,1}. The rank is determined. That rank can be from 0 to 31, but ranks < 28 are rare, and their counts are pooled with those for rank 28. Ranks are found for 40,000 such random matrices and a chi-square test is performed on counts for ranks 31,30,29 and <=28.

The ***Rank 32x32*** matrix test is like the 31x31 test except that the matrix has 32 rows of 32-bits each. Ranks less than 29 are pooled with the rank 29 count.

The ***Rank 6x8*** matrix test uses six random 32-bit integers from the test sequence, a specified 8-bit byte is chosen, and the resulting six bytes form a 6x8 binary matrix whose rank is determined. Within the 32-bit word, all 24 starting bit positions for the 8-bit byte are tested. The resulting rank can be 0 to 6, but ranks 0,1,2,3 are rare; their counts are pooled with those for rank 4. Ranks are found for 100,000 random matrices, and a chi-square test is performed on counts for ranks 6,5 and <=4.

The ***Missing 20bit words*** test treats the test sequence as a stream of "letters", which are either 0 or 1 and examines the overlapping of 20-letter "words". Thus the first word is bits 1 to 20; the second is bits 2 to 21, and so on. The bitstream test counts the number of missing 20-letter (20-bit) words in a string of 2**21 overlapping 20-letter words. There are 2**20 possible 20 letter words. For a truly random string of 2**21+19 bits, the number of missing words j should be (very close to) normally distributed with mean 141,909 and sigma 428. Thus (j–141909)/428 should be a standard normal variate (z score) that leads to a uniform [0,1) p value. The test is repeated twenty times.

The ***Missing 10bit pairs*** test, also called OPSO for Overlapping Pairs Sparse Occupancy, treats the test sequence as a stream of 10-bit "letters", which can take on 1024 different values. Each letter is determined by a selected ten bits from each 32-bit word from the test sequence. All 22 starting positions for this 10-bit letter are tried within the 32-bit word. The number of overlapping 2-letter pairs that do not occur in the entire sequence are counted. Those counts should be very close to normally distributed with a mean of 14909 and sigma of 290.

The ***Missing 5bit*** *quads* test, also called OQSO for Overlapping Quad Sparse Occupancy, is similar to the OPSO test except that it considers 4-letter words from an alphabet of 32 letters. One letter is chosen from each 32-bit word of the test sequence and the test is run for all 27 different starting positions of the 5-bit letter within the 32-bit word.

The ***Missing 2bit tens*** test, also called the DNA test, is similar to the OPSO test except that it considers 10-letter words from an alphabet of 4 letters (like the four DNA bases C, G, A and *T*). One letter is chosen from each 32-bit word of the test sequence and the test is run for all 30 different starting positions of the 2-bit letter within the 32-bit word.

The ***Count-Ones All Bytes*** test treats the test sequence as a stream of bytes (four per 32-bit word). Each byte can contain 0 to 8 ones with probabilities 1,8,28,56,70,56,28,8,1 over 256. The bytes are converted to one of five letters, *A* through *E*, based on the number of ones in that byte. The letter *A* is generated by 0, 1, or 2 bits set to one. The *B* means 3 bits were set, 4 yields C, 5 yields *D*, and 6, 7, or 8 yield *E*. There are 5**5 possible five letter words and strings of 256,000 overlapping 5-letter words create the frequency count of each word. The quadratic form

in the weak inverse of the covariance matrix of the cell counts provides a chi-square test Q5-Q4, the difference of the naïve Pearson sums of (OBS-EXP)**2/EXP on the counts for 5- and 4-letter cell counts.

The ***Count-Ones Specific*** test is similar to the previous test except that only one 8-bit byte per 32-bit word is used. The test is run for all 24 different starting positions of the byte within the word.

The ***Parking Lot*** test is based on a square of side 100. Randomly "park" the first car, which is viewed as a circle of radius 1, at a location determined from two 32-bit words of the test sequence. Then try to park a 2nd, a 3rd, and so on, each time parking "by ear". That is, if an attempt to park a car causes a crash with one already parked, try again at a new random location. To avoid path problems, consider parking helicopters rather than cars. Each attempt leads to either a crash or a success; the latter followed by an increment to the list of cars already parked. A simple characterization of this experiment is, k, the number of cars successfully parked after n=12,000 attempts. Simulation shows that k should average 3523 with sigma 21.9 and is very close to normally distributed. Thus (k–3523)/21.9 should be a standard normal variable, which, converted to a uniform variable, provides input to a KSTEST based on a sample of 10.

The ***Minimum Distance*** test does the following 100 times. Choose n=8000 random points in a square of side 10000. Find d, the minimum distance between the (n**2–n)/2 pairs of points. If the points are truly independent uniform, then d**2, the square of the minimum distance should be (very close to) exponentially distributed with mean 0.995. Thus 1–exp (–d**2/0.995) should be uniform on [0,1) and a KSTEST on the resulting 100 values serves as a test of uniformity for random points in the square. The KSTEST is based on the full set of 100 random choices of 8000 points in the 10000x10000 square.

The ***3D Sphere*** test chooses 4000 random points in a cube of edge 1000. At each point, center a sphere large enough to reach the next closest. Then the volume of the smallest such sphere is (very close to) exponentially distributed with mean 120pi/3. Thus the radius cubed is exponential with mean 30. The mean is obtained by extensive simulation. The 3DSPHERES test generates 4000 such spheres 20 times. Each min radius cubed leads to a uniform variable by means of 1–exp (–r**3/30.), then a KSTEST is done on the 20 p-values.

The ***Squeeze*** test converts the test sequence of 32-bit integers into floating point numbers to get uniforms on [0,1). Starting with k=2**31=2147483647, the test finds j, the number of iterations necessary to reduce k to 1, using the reduction k'=ceiling (k*U), with U provided by floating integers from the file being tested. Such j's are found 100,000 times, then counts for the number of times j was <=6,7,47, >=48 are used to provide a chi-square test for cell frequencies.

In the ***Overlapping Sums*** test the 32-bit integers are floated to get a sequence $U(1)$,$U(2)$,... of uniform [0,1) variables. Then overlapping sums, $S(1)$=$U(1)$+...+$U(100)$, $S2$=$U(2)$+...+$U(101)$,... are formed. The S's are virtually normal with a certain covariance matrix. A linear transformation of the S's converts them to independent standard normals, which are converted to uniform variables for a KSTEST.

The ***Up-Down Runs*** test counts runs up, and runs down, in a sequence of uniform [0,1) variables, obtained by floating the 32-bit integers in the test sequence. This example shows how runs are counted: .123,.357,.789,.425,.224,.416,.95 contains an up-run of length 3, a down-run of length 2 and an up-run of (at least) 2, depending on the next values. The covariance matrices for the runs-up and runs-down are well known, leading to chi-square tests for quadratic forms in the weak inverses of the covariance matrices. Runs are counted for sequences of length 10,000. This is done ten times.

The ***Craps Game*** test plays 200,000 games of craps, finds the number of wins and the number of throws necessary to end game. The number of wins should be (very close to) a normal with mean 200000p and variance 200000p(1–p), with p=244/495. Throws necessary to complete the game can vary from 1 to infinity, but counts for all>21 are lumped with 21. A chi-square test is made on the number-of-throws cell counts. Each 32-bit integer from the test file provides the value for the throw of a die, by floating to [0,1), multiplying by 6 and taking 1 plus the integer part of the result.

Summary of Statistical Tests

The MD5Random and SHA1Random algorithms produce output streams that pass all of the statistical

tests described in this paper. However, some of the tests such as *Poker-5* and *Rank 32x32* produced results that were not as good as the others. Additional testing could investigate whether these results indicate a real problem or are just statistical anomalies.

Conclusions

This paper has described the algorithms used for the BSAFE and JSAFE pseudorandom number generators, which are widely used in commercial products. The design goals were presented and the discussion explained the extent to which they were achieved. The algorithm analysis highlighted the assumptions that are made about the underlying hash functions to ensure good cryptographic properties for the PRNGs. The statistical tests provide additional assurances about the suitability of these algorithms as random number generators. The preliminary analysis presented here indicates that these are good algorithms for generating cryptographically strong pseudorandom numbers.

References

[1] R.W. Baldwin. *Proper Initialization for the BSAFE Random Number Generator*. RSA Labs Bulletin 3, Redwood City, California, January 1996. http://www.rsa.com/rsalabs/pubs/updates/bull-3.pdf

[2] H. Beker and F. Piper. *Cipher Systems*. Van Nostrand, London, 1982.

[3] G.D. Carter. *Aspects of local linear complexity*. Ph.D. thesis, University of London, 1989.

[4] H. Dobbertin. *Cryptanalysis of MD4*. In Proceedings of the 3rd Workshop on Fast Software Encryption, Cambridge, U.K., pages 53-70, Lecture Notes in Computer Science 1039, Springer-Verlag, 1996.

[5] D. Eastlake 3rd, S. Crocker, J. Schiller. *Randomness Recommendations for Security*. RFC 1750, IETF. December 1994. Also at http://ds.internic.net/rfc/rfc1750.txt

[6] E.D. Erdmann. *Empirical Tests of Binary Keystreams*. Master's thesis, University of London, 1992.

[7] S.W. Golomb. *Shift Register Sequences*. Holden-Day, San Francisco, 1967.

[8] I.J. Good. *The serial test for sampling numbers and other tests for randomness*. Proc. Camb. Phil. Soc., 49:276-284, 1953.

[9] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. *Cryptanalytic Attacks on Pseudorandom Number Generators*. Fast Software Encryption, Fifth International Workshop Proceedings (March 1998), Springer-Verlag, 1998, to appear. Also available at http://www.counterpane.com/pseudorandom_number.html

[10] D.E. Knuth. *The Art of Computer Programming*. Volume 2, Addison-Wesley, Reading, Mass., 2nd edition, 1981.

[11] P. Kocher. *Timing Attacks on Diffie-Helman, RSA, DSS and Other Systems*. In Proceedings of Advances in Cryptology – Crypto 96, Santa Barbara, California. Pages 104 to 113 in Lecture Notes in Computer Science #1109, Springer-Verlag. 1996.

[12] G. Marsaglia. *Diehard Statistical Tests*. http://stat.fsu.edu/~geo/.

[13] H. Niederreiter. *The linear complexity profile and the jump complexity of keystream sequences*. In I.B. Damgård, editor, *Advances in Cryptology - Eurocrypt '90*, pages 174 - 188, Springer-Verlag 1991.

[14] NIST, FIPS PUB 180-1: *Secure Hash Standard*, http://csrc.nist.gov/fips/fip180-1.txt (ascii). April 1995.

[15] R. Rivest. *The MD5 Digest Algorithm*. RFC 1321. http://ds.internic.net/rfc/rfc1321.txt April 1992.

[16] M.J.B. Robshaw. *On Recent Results on MD2, MD4, and MD5*. RSA Labs Bulletin 4, Redwood City, California, November 1996. http://www.rsa.com/rsalabs/pubs/updates/bull-4.pdf

[17] RSA Data Security Inc. *What is RSAREF?* In RSA's Cryptography FAQ. http://www.rsa.com/rsalabs/newfaq/q174.html

For more information on this and other recent security developments, contact RSA Laboratories at one of the addresses below.

RSA Laboratories
2955 Campus Drive, Suite 400
San Mateo, CA 94403 USA
650/295-7600
650/295-7599 (fax)
rsa-labs@rsa.com
http://www.rsa.com/rsalabs/

The preliminary analysis presented here indicates that these are good algorithms for generating cryptographically strong pseudorandom numbers.

Copyright © 1998 RSA Laboratories, a division of RSA Data Security, Inc., a Security Dynamics Company. All rights reserved.