EVALUATION OF

VIA C3 NEHEMIAH

RANDOM NUMBER GENERATOR

PREPARED BY

Cryptography Research, Inc.
607 Market St., 5th Floor
San Francisco, CA 94105
(415) 397-0123

Last Revision: February 27, 2003

Information in this white paper is provided without guarantee or warranty of any kind. This review was performed for VIA Technologies, Inc. but represents the findings and opinions of Cryptography Research, Inc. and may or may not reflect opinions of VIA Technologies, Inc. or Centaur Technology. Characteristics of the C3 Nehemiah may vary with design or implementation changes.

Contents

1. Introduction

Randomness is required for a variety of computational, statistical, and security-related applications. In particular, random numbers and the processes used to generate them are a critical component of secure protocols and cryptographic key generation. Security processes that lack adequate sources of randomness will have poor security.

Cryptography Research has evaluated the C3 Nehemiah random number generator, which is an on-chip component of the VIA Technologies Nehemiah processor core. When properly used, the generator was found to be a consistent, high-rate source of entropy which we believe is suitable for use in cryptographic and high-assurance applications. This report analyzes the Nehemiah RNG design, provides an entropy analysis of the source, and provides developer recommendations for proper use of the Nehemiah RNG. Cryptography Research provided no Nehemiah design assistance to VIA Technologies or Centaur Technology.

2. Background

2.1. Applications Requiring Randomness

In general, random numbers can be summarized as numbers that are indistinguishable from outcomes that would arise purely by chance. The quality of a random number generator is often measured by the degree to which it produces unpredictable and unbiased output.

Many cryptographic protocols require secret numbers. For example:

- Conventional encryption requires the generation of unguessable keys.
- The computation of a digital signature with the Digital Signature Algorithm requires, besides the signer's private key, a value customarily called k that must be secret, and that must not be re-used.

- Standards for message encryption using the RSA algorithm generally require the use of random numbers to form message padding.
- Many challenge-response protocols require the use of a unique number, or nonce. In practice, a good way to produce a number with a large likelihood of being unique is to use a sufficiently large random number.

Random numbers are also widely used in numerical simulations, gaming, statistical analysis, and distributed computations. While a high quality random source is always best, randomness requirements vary among applications. For example, numerical simulations often require random numbers that are unbiased, but have fewer unpredictability requirements. In contract, cryptographic applications often have extremely strong unpredictability requirements but may be slightly tolerant of biased information. While this report focuses on the use of the Nehemiah RNG for cryptographic applications, the results may be applied to other applications.

2.2. Sources of Randomness

Randomness can be found in several places, the more noteworthy of which we will now discuss.

Quantum Phenomena. Devices have been specifically designed to translate quantum-mechanical uncertainty into random digits, typically harnessing radioactive decay (the classic case being Rand Corporation's book, "A Million Random Digits", published in the 1950's).

Thermal Noise. Excluding quantum mechanics, the behavior of physical systems is deterministic (e.g. given a complete description of a system, one can compute its future behavior). While this appears to make random number generation impossible, the fact that matter is composed of particles endowed with disorganized thermal motion makes it impossible, for practical purposes, to achieve a complete description of a system. Accordingly, the places where thermal motion affects a circuit's behavior offer sources of effective randomness.

For example, where electrons must surmount a potential barrier to move from one conductor to another, each electron's thermal motion may help it or hinder it in crossing the barrier, so that the total charge transferred during some time window varies. Similarly, thermal movements of silicon atoms in a small region of the crystal lattice momentarily affect current flow through that region, again affecting the charge transferred during any given time window.

External Influences. Cosmic rays, temperature fluctuations, supply-voltage variations, and stray electromagnetic fields may change a circuit's behavior and are generally considered to be unpredictable enough to be counted as contributing to cryptographically useful randomness.

Timings of network message arrivals, keystrokes, mouse movements, and disk-operation completions are also often collected to contribute to random number generators' "entropy pools[1]." While this practice can be effective, the process of data collection is often cumbersome to implement. Because the possibility exists for sampled events to be determined or observed by non-random (or hostile) processes, it is difficult to provide a reasonable estimate as to the amount of entropy contributed by each event. Finally, it is difficult to collect entropy in systems where no human is present or in systems where the contents of the entropy pool may be visible to hostile processes.

Chaos. A deterministic system is called chaotic if an infinitesimally small perturbation to its initial conditions produces a change in its behavior that grows exponentially with time. While chaos is a concept completely different from randomness, it is important in random-number generation for the following reason: If an RNG is chaotic, and if there is some inescapable uncertainty in any contribution to its state (e.g., due to thermal noise), then simply by waiting for a certain length of time, namely the time required for the exponential growth of that uncertainty to reach the magnitude of the system's gross state, we can assume that the state of the system is unknowable. By waiting a sufficient length of time between samplings, is may be possible to sample high-quality random bits from a chaotic system that is otherwise deterministic.

3. Design Overview

Cryptography Research obtained RNG design information from tests on Nehemiah processors, as well as documentation, source code, and conversations with engineers at Centaur Technology. This section contains a high-level overview of the RNG design and operation.

3.1. Entropy Source

The RNG hardware comprises two parts: a raw-bit generator that serves as an entropy source and digital post-processing circuitry. The raw-bit generator produces somewhat random bits which the design assumes will have imperfect statistical properties. The post-processing circuitry then uses "whitening" and bit discarding to improve the statistical properties of the imperfect random bits.

Raw bits are generated by using a slow freewheeling oscillator (configured by bias inputs to 20-68 MHz) to clock the sampling of a fast freewheeling oscillator (configured by bias inputs to 450-810 MHz). This approach is good if the jitter in the slow oscillator (i.e., cycle-by-cycle variations in the oscillator's period) is comparable in magnitude to the period of the fast oscillator. In the case of this RNG, the jitter of the slow oscillator is increased by deriving an internal oscillator bias current from two additional fast oscillators. Thermal noise, manufacturing variations, temperature, software settings, and local electrical conditions are expected to cause oscillator variations and contribute entropy to the sampled output. A diagram of the source follows.

[1] See RFC 1750 at http://www.ietf.org/rfc/rfc1750.txt

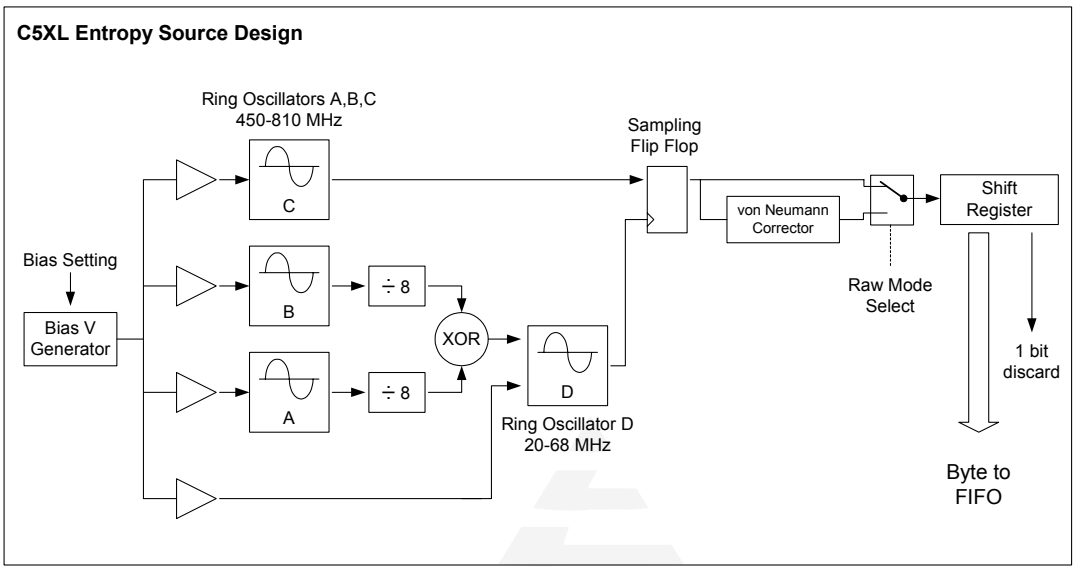

Figure 1: Nehemiah entropy source design

Oscillator operation. As shown above, oscillators A, B, and C are identical free-running ring oscillators, centered at 450-810 MHz. A single software-controlled bias voltage[2] is used to adjust an internal current supply and effectively change the operating frequency of the oscillators. This bias voltage is intended to be static and to be set only on system reset. Oscillators A and B are independently divided by 8, which should generate two independent oscillatory signals with periods of ~14ns (at the default bias setting). These independent signals are combined with the XOR operation, and used in conjunction with the bias voltage to modulate oscillator D. A computer model of the system yielded the following sample result of the XOR output:

[2] Set to one of 8 values (via bits 12:10 of MSR 0x110B).

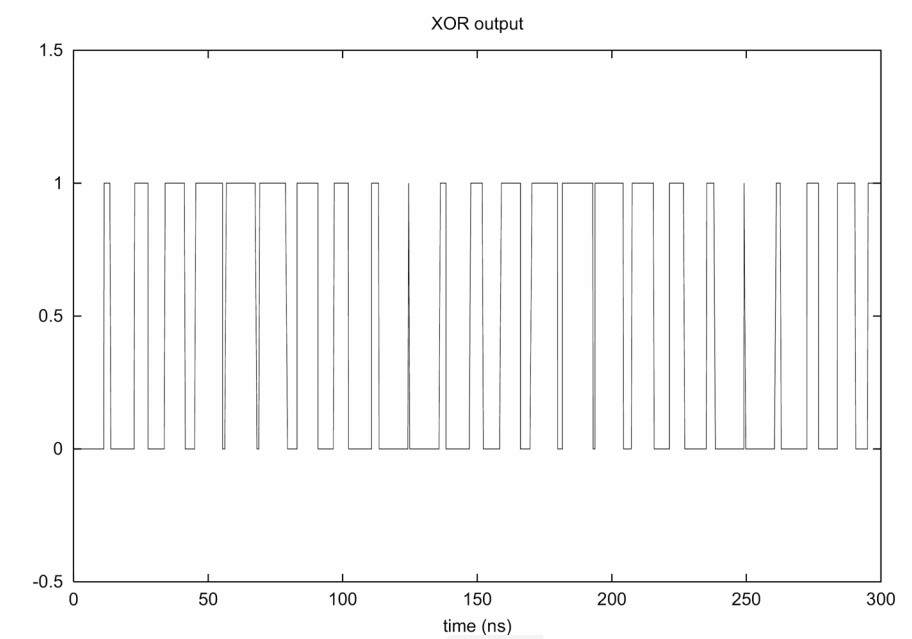

Figure 2: Output from software-based model of oscillator behavior

Oscillator D is a slower version of the above oscillators, centered at approximately 45 MHz. It serves as the RNG sampling clock and is used to clock a flip-flop that samples oscillator C. The bias voltage for oscillator D incorporates the same bias voltage as the above oscillators, but also includes the combined XOR output from oscillators A and B.

Whitener. Pairs of bits in the sampled output are then passed through a von Neumann whitener, which operates on two new bits at a time (once for every 2 periods of oscillator D). The von Neumann whitener is designed to reduce bias from the sampled data and operates in the following fashion:

Input (bits discarded after use)	**Corrector Output**
0 0	No output (no shift register activity)
0 1	Output = 0 (shift "0" into shift register)
1 0	Output = 1 (shift "1" into shift register)
1 1	No output (no shift register activity)

The whitening circuitry supports a "raw bit bypass" mode, in which each bit sampled from oscillator C is clocked directly into the shift register. This mode provides

convenient access to RNG internals for testing purposes. (Most of the entropy tests performed by Cryptography Research were performed on the raw source.)

Shift register operation. The shift register shifts in whitened bits as they become available. After 8 bits of information have entered the shift register, one byte is transferred to the digital subsystem FIFO and the next incoming bit is discarded.

Power control. Control logic can instruct the entropy source to enter a low-power mode, where oscillator internal bias currents are removed (effectively removing power from the oscillators). When exiting low-power mode, the contents of the shift register are discarded and the next complete byte destined for the shift register is also discarded.

3.2. Digital Hardware and Firmware

The datapath and microcode subsystem are primarily responsible for supplying the microprocessor with random bytes when they are requested. The digital block shares the primary microprocessor clock, maintains a 32-byte FIFO queue (comprised of four 8-byte buffers), controls software reads, and permits adjustment of RNG parameters.

Registers and commands. The RNG is controlled through a 32-bit status/configuration register (MSR 0x110B), which is accessible only to ring 0 code. RNG data are read via the XSTORE instruction, a Nehemiah specific x86 command. Based on the 2-bit divisor setting, 1 to 8 bytes of the FIFO memory are copied to user memory on each successful XSTORE call. The XSTORE command also copies the contents of the MSR to user memory.

Queue management. Data is transferred from the entropy source to the FIFO on a byte-wise basis. Each XSTORE instruction consumes either zero or 8 bytes of FIFO memory. (Zero bytes are consumed if not enough data is available. Otherwise, 8 are consumed.) The hardware prevents "stale" data from being re-read, and also marks all FIFO data stale when any RNG configurations are changed. The entropy source oscillators are powered down when the FIFO is full (see power control section below).

Power control. The digital subsystem powers off the entropy source when the RNG is not enabled, or when the FIFO is full.

Selectable filter: XSTORE divisor. The XSTORE command consumes 8 bytes of FIFO memory unless fewer than 8 FIFO bytes are available, in which case no bytes are returned. The XSTORE command takes a 2-bit divisor parameter, which is used to discard ($2^{divisor} - 1$) bits for each bit returned to user memory. For example, a divisor of 0 will result in 8 bytes of output data, while a divisor of 3 will result in 1 byte of XSTORE output.

Selectable filter: String filter[3]. When activated, the string filter discards new bytes from the entropy source if the new byte would otherwise cause bytes entering the FIFO to exceed a maximal number of consecutive "1" or "0" bits (where the limit is a 6-bit configurable parameter). The filter also has an "alarm" bit that is set if any bytes have been discarded.

4. Entropy Analysis

4.1. Background on Entropy

Entropy is used by cryptographers as a measure of unguessability or incalculability. If a secret value (such as a key) is chosen by a process that gives n bits of entropy, then the average number of tries required to guess that value by exhaustive search is 2^{n-1}. Entropy is conveniently additive: the entropy of the combination of two independently chosen values is the sum of their individual entropies. Because of this additivity, it makes sense to speak of the entropy per bit of a bit source and to compute the number of bits that must be combined to make a secret key of a given strength.

When a bit generator produces serially correlated bits, successive bits are not independent, and their entropies cannot be meaningfully added. However, the useful

[3] Use of this feature is not recommended except for testing purposes.

notion of per-bit entropy can be restored by calculating the entropy of the distribution of strings of n successive bits (for some n large enough to encompass the effects of the serial correlation) and dividing by n. This per-bit estimate may understate the entropy of individual bits or of short strings of bits, but this conservative estimate is appropriate in security applications.

In an analysis of this kind, one generally doesn't arrive at a specific figure for the per-bit entropy of a bit source. Instead, one establishes an upper bound on the entropy, typically by identifying unequal distributions or patterns in sequences of output bits; and sometimes one derives a lower bound on the entropy, perhaps from theoretical arguments or perhaps by reasoning from analog measurements on the system. Just as no statistical test can prove the randomness of a bit source, no statistical test on output bits can establish a lower bound on entropy.

The upper bound on the entropy estimate allows us to warn convincingly against procedures that are demonstrably inadequate. For example, if a hypothetical bit generator has been shown to provide only 1/2 bit of entropy per output bit, it is easy to dissuade an application programmer from constructing an 80-bit secret key by simply reading 80 bits from that bit generator. At the other extreme, a lower bound is the only firm assurance for the most cautious user, who will typically divide his entropy requirements by the lower bound to determine the number of RNG bits that he should hash together to make a key. Establishing any sort of lower bound is difficult, and it is not unusual for the upper and lower bounds to differ by orders of magnitude.

4.2. Tools and Approaches

In the studies described here, the RNG was run in its most vulnerable mode: the von Neumann whitener was turned off, and every bit clocked into the shift register was read and examined. As expected, the output in this mode is unsuitable for numerical applications; the bits produced are statistically imbalanced and fail all sensitive statistical tests. However, this mode makes it is easier for to find any nonrandom behavior that might cause problems and would be more difficult to detect when operating in other

modes . (In particular, the prudent user will run the RNG with the von Neumann whitener turned on, and may set the EDX divider to use a subset of the output bits, both of which are designed to improve the quality of the random bits produced by discarding bits that are more likely to be correlated.)

To find an upper bound on per-bit entropy, we have used two general approaches:

- *Computing entropies of distributions*. Given a large sample of RNG output, we tally the number of occurrences of all n-bit patterns, for n up to 24. We then compute the entropy of this distribution and divide by the number of bits.
- *Measuring the "predictability" of bits*. Given a large sample of RNG output, we build a predictor that attempts to predict the value of any bit from the values of bits immediately around it in the sequence. If the success rate of the predictor is p, the entropy of the bit being predicted is taken as $-p\log_2(p)-(1-p)\log_2(1-p)$.

4.3. Aspects Studied

Several different aspects of RNG operation were studied:

- *Normal operation*. Tests were performed to determine if non-random behavior was detectable in the raw output. Output sufficiently biased to introduce weaknesses in security applications will normally make an RNG fail standard statistical tests.
- *Bias variation*. The 3-bit bias setting[4] affects the periods of all four source oscillators. Variations in the bias setting may affect the RNG output.
- *Chip variation*. Differences in manufacturing lot or wafer position may affect the RNG source operation. The authors received four Nehemiah chips of varying speed grades and performed testing on all four chips.

[4] Note: the bias setting can be treated as a 2's complement number, where bias=0 is the center and bias=3 / bias=4 represent bias extremes.

- *Startup behavior*. Because the oscillators are stopped when the accumulation buffers are full, XSTORE operations are likely to include data from freshly started oscillators. It seems reasonable to suspect that oscillator startup is to some extent reproducible, so initial RNG bytes may tend to be the similar across XSTORE calls returning startup data.
- *Die temperature*. The temperature of the processor die could affect the operation of the RNG oscillators in ways that reduce the quality of RNG output.
- *Local correlation*. In multi-application environments, a potentially hostile process could access the RNG immediately before or after a secure process executes the XSTORE operation. Therefore, any bit correlations across consecutive XSTORE operations may detract from the quality of RNG output.
- *Varying processor activity*. Varying levels of CPU activity may affect the properties of the RNG output.

4.4. Results

4.4.1. Normal Operation

To study RNG behavior under normal (continuous) operation, samples of slightly more than 5 gigabytes of raw output were taken from several different processor chips at all 8 possible bias settings. These samples were analyzed for the apparent entropy of the distribution of 16-bit values, the bias of single-bit frequencies, and the predictability of individual bits within 16-bit fields. The results for one of the processor chips are presented in the following figures.

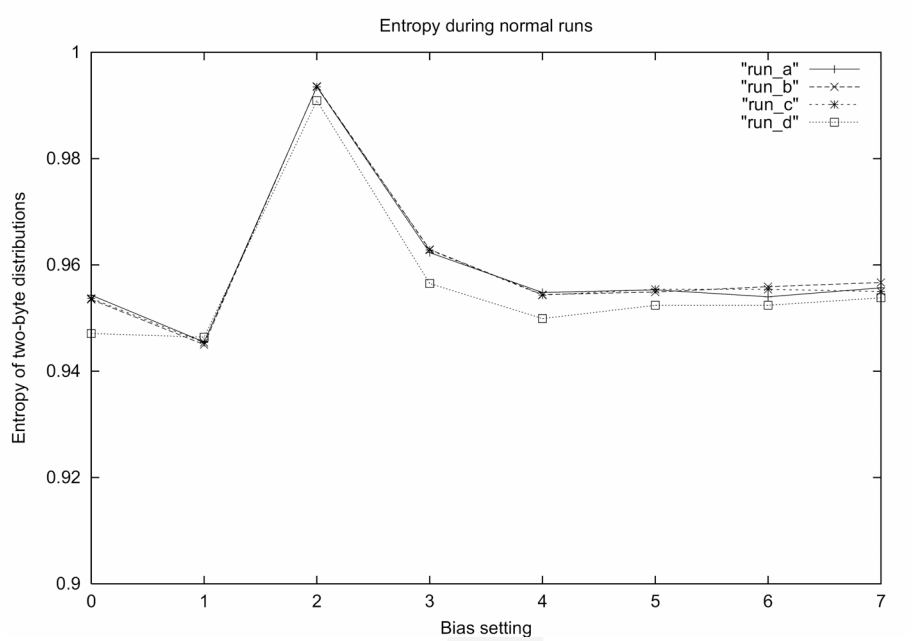

Figure 3: Entropy during continuous RNG operation (raw output)

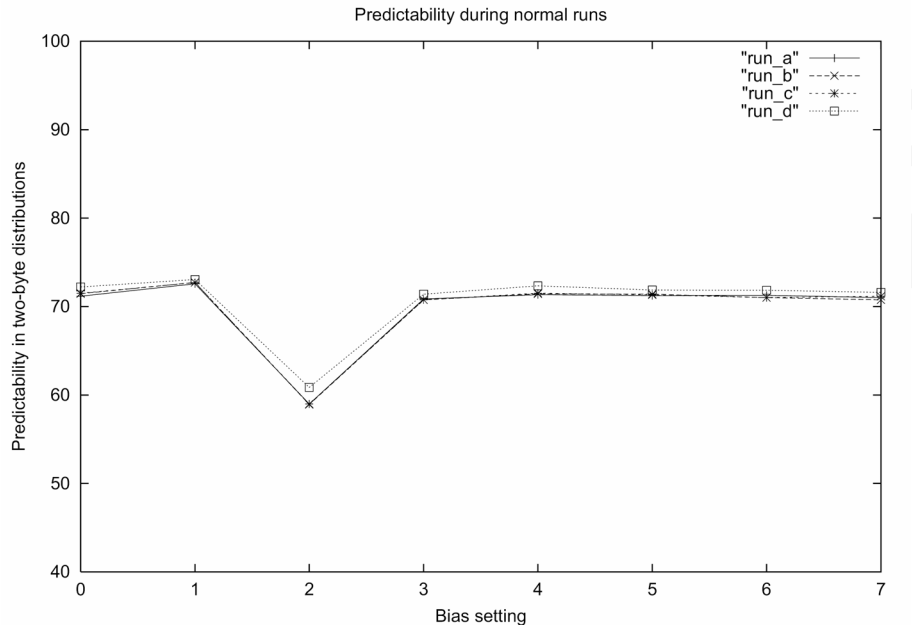

Figure 4: Predictability during continuous RNG operation (raw output)

A worst-case predictability of 72% implies an entropy of only 0.855 bit, notably smaller than the 0.95-bit-per-bit entropy of the overall distribution. This difference is expected, since the displayed predictability value is the highest of the 16 bit positions, while the overall entropy figure reflects also the 15 other, apparently less predictable, bit positions. In addition, the predictability measure also assumes knowledge of all other bits.[5] Thus, a large difference between the entropy of the distribution and the entropy inferred from the highest predictability indicates differences in bit-to-bit predictability.

We say that other bits are *apparently* less predictable, because the RNG generates bits for all positions in the same way, so we do not expect that some bit positions within our arbitrary 16-bit framework could be intrinsically more predictable than others. The appearance of a difference stems from the fact that we are predicting the value in one bit position based on the values in the other 15 positions, and the predictive value of those other 15 positions presumably varies depending on their dispositions relative to the bit being predicted. In practice, the most predictable positions tend to be near the center of the 16-bit window. It is important to note that other bit positions would probably be just as predictable, if we kept a larger context of adjacent bits on which to base our predictions. It is also noteworthy that the RNG discards a bit between bytes. As a result, bits adjacent to the byte boundary are less predictable because only one neighboring bit is known. To be conservative, one should assume that the worst predictability score applies to all bit positions.

The following lessons were drawn from the test results:

- The entropies of the distributions are consistent with a raw source of generally good quality. These tests yield candidate upper and lower entropy bounds of 0.85 to 0.99 bits per bit of raw output, respectively.

[5] A simple example is the case where a two-bit value is evenly distributed between the two outcomes {0,0} and {1,1}. The distribution's entropy is 1 bit (0.5 bit per output bit), but the predictability of each bit would be 1, implying an entropy of zero. Since one cannot be certain that adversaries will not have access to prior or subsequent output bits, the entropy implied by the predictability is a more conservative measure of the entropy of the output

- There is a fairly consistent tendency for bits to be predictable at about the 70% level. While some bias settings (in particular, bias = 2) appear to yield higher levels of unpredictability, the relatively small sample size does not allow conclusive evidence of a trend. Accordingly, the conservative interpretation is that raw, continuous bits from the RNG can be predicted at a 70% predictability rate, yielding approximately 0.88 bits of entropy per bit of raw output.
- The single-bit distribution tends to be a few percentage points off the desired 50-50 (not shown). This discrepancy is trivial compared with the predictability, and is substantially reduced (but not completely eliminated by) the whitener.

The predictability figures indicate that the RNG favors certain patterns of behavior. Some questions about these patterns come to mind.

- Do these patterns persist over time?
- Do the same patterns apply at different bias settings?
- Do different chips have the same patterns?

To study the patterns' persistence over time, four separate samples of 5.24 gigabytes each were taken. The samples were named 1a, 1b, 1c, and 1d. Patterns found in samples 1b, 1c, and 1d were used to attempt to predict bits in sample 1a. This process was repeated at each of the 8 bias settings. The results appear in the following figure.

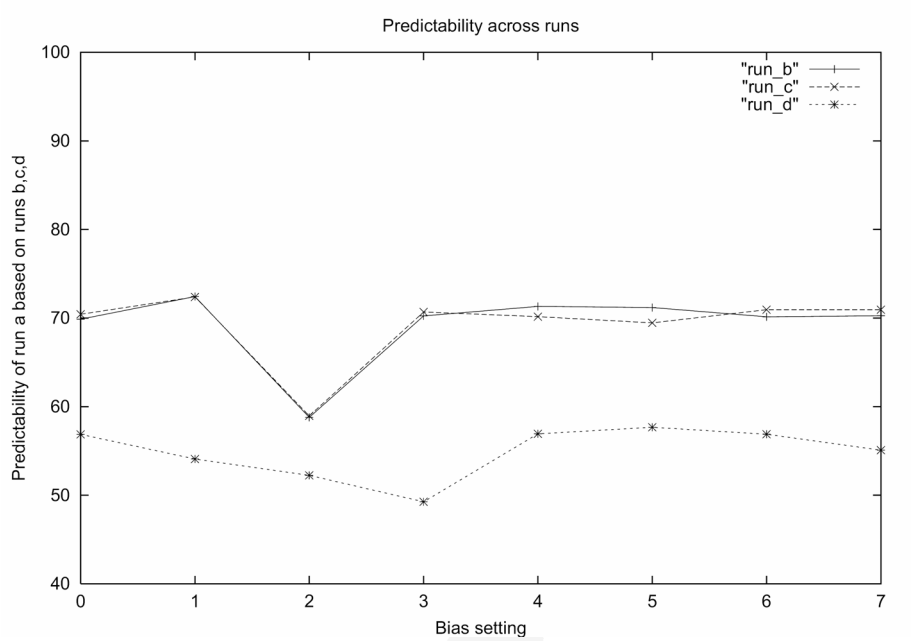

Figure 5: Predictability across runs (raw output)

Observations:

- Patterns from runs 1b and 1c are relatively effective at predicting the bits in run 1a, while run 1d is conspicuously less good. Since runs 1a and 1d were spaced further apart, likely patterns may be dependent on local environmental conditions.
- The bias setting of 2 is again seen to be less predictable than other bias settings, which would indicate that bias=2 may yield higher quality output.

To study the persistence of patterns from one bias setting to another, we used the 8 "1a" samples from the preceding study, and measured how well the tables built at one bias setting predicted bits at another bias setting. The results appear in the following table.

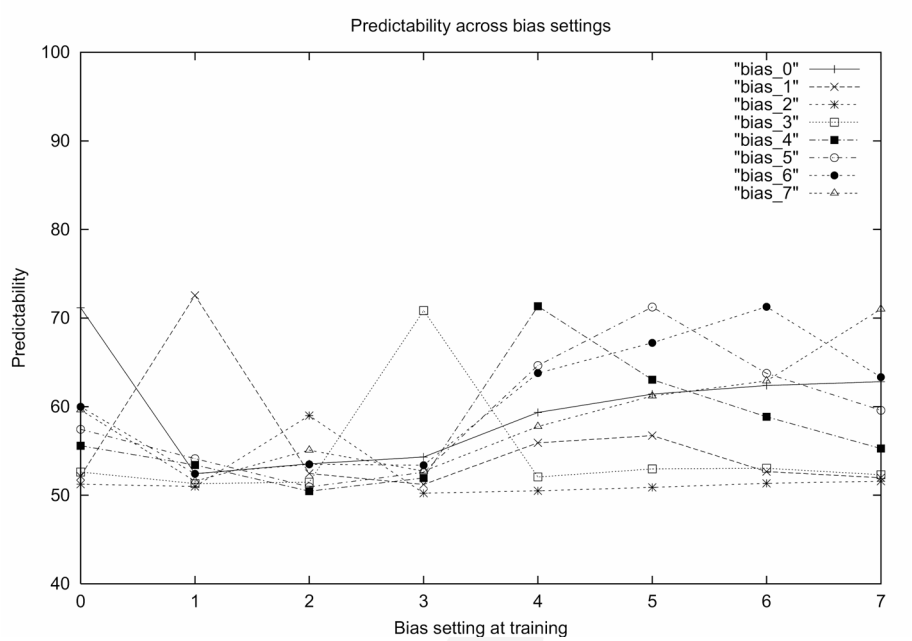

Figure 6: Predictability across bias settings (raw output)

Observations:

- Changes in bias yield gradual changes in RNG output patterns. For example, results from bias 5 are most effective at predicting the results from bias 5, are somewhat less effective at predicting results from biases 4 and 6, and are progressively less effective at predicting other bias settings.[6]
- Biases 1 and 3 are relatively individualistic, while bias 2 is, again, especially unpredictable. This suggests that bias 2 may provide a "local maximum" for pattern unpredictability.

To study the commonality of patterns between chips, we compared, at each bias setting, 5.24-gigabyte samples taken from chips arbitrarily named 1, 3, and 4. (For chips 1 and 3 we had multiple data sets, of which we used the first, 1a and 3a.) The results appear in the following figure.

[6] Note: the bias setting can be treated as a 2's complement number, where bias=0 is the center and bias=3 / bias=4 represent bias extremes.

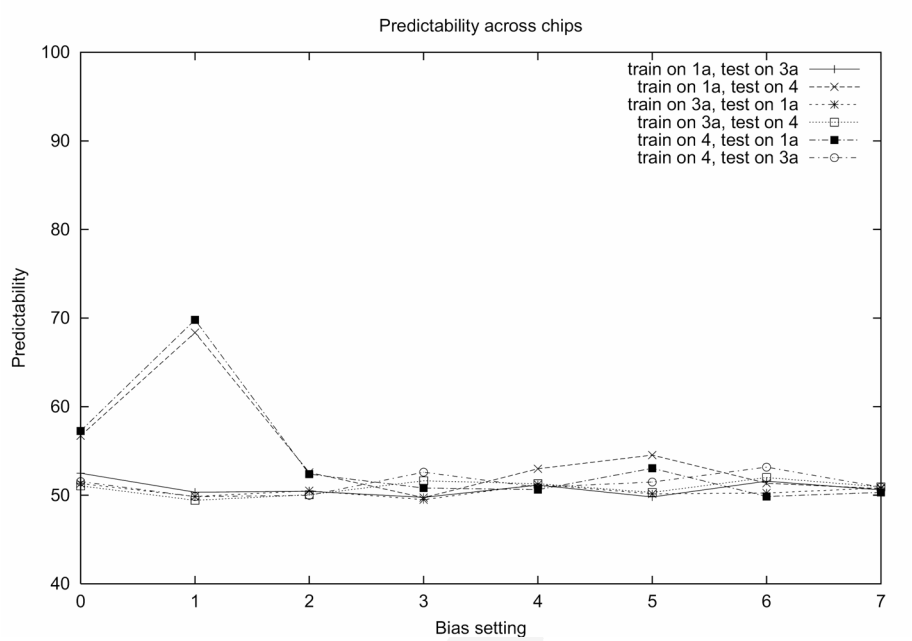

Figure 7: Predictability across chips (raw output)

Observations:

- With one exception, one chip's patterns are not particularly useful for predicting another chip's bits. (A 50% chance of predicting a bit correctly means that the bit is effectively random.)
- At bias=1, chips 1 and 4 appear to bear a strong resemblance to each other.
- Bias = 2 appears to be notably less predictable in all three chips. At this bias, the self-predicting success rate is below 61% for all three chips.

We can conclude that continuous usage of the RNG yields mildly predictable RNG behavior, yielding a "worst case" entropy of about 0.85 bits per raw output bit under predictability analysis. While predictability patterns were found to vary with oscillator bias settings, no bias settings were particularly poor (although the bias=2 setting appears to yield somewhat higher entropy). In addition, the results also suggest that RNG behavior may be affected by chip-specific differences, such as manufacturing or die temperature, although a larger sample size is required to assert this claim.

4.4.2. Startup Behavior

A program was written that reads 8 bytes from the RNG, then pauses long enough to ensure that the accumulation buffers become full and the oscillators are therefore stopped before reading the next 8 bytes, repeating 12.5 million times. Counters tally the number of occurrences of every possible 16-bit value in each of the seven pairs of adjacent bytes, a total of 7 * 65,536 counters. For each of the seven byte pairs, the table of 65,536 tallies was processed to compute the classical Shannon entropy of the distribution. Also for each set of tallies, the best single-bit predictor was found and its success rate computed. For example, it might be found that the bit in the 0x0020 position can be correctly predicted with 68% success based on the remaining 15 bits.[7]

The results are presented in the following Figure. As described previously, the distribution entropy is significantly higher than entropy implied by the predictability the target bits.

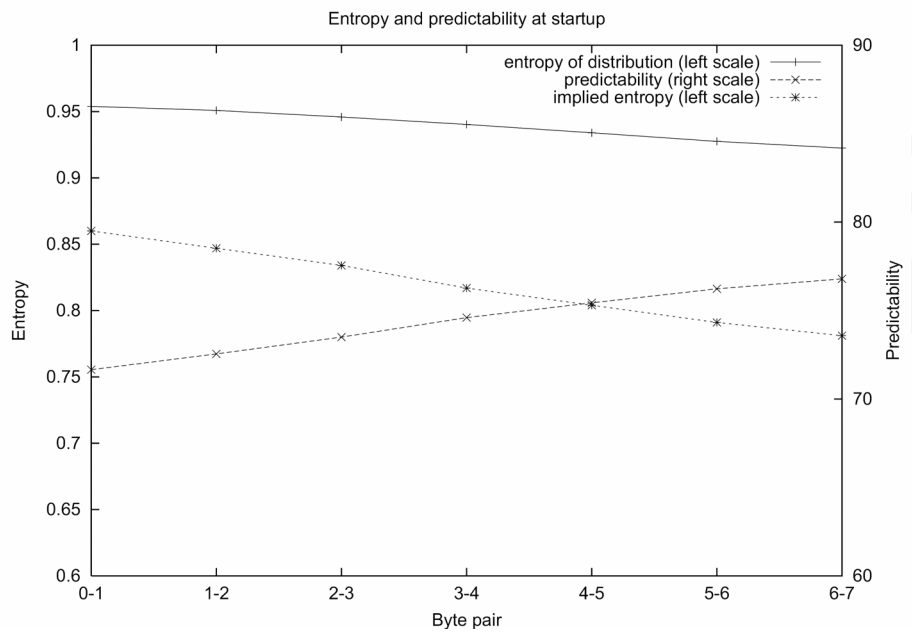

Figure 8: Entropy and predictability at startup (raw output)

[7] This measure overstates the actual predictability, since the predictor was used on the same data that was applied to train it. Fair tests, applying the predictor to an independent set of tallies, were occasionally done to assess the magnitude of the overstatement.

These results contradict the hypothesis that the oscillators might come up in some repeatable relationship, and support information received from Centaur on the relatively fast startup-timings for the RNG oscillators. In particular, the computed entropies (by distribution and by predictability measurement) are higher for earlier startup bytes. Therefore, applications reading RNG data generated at oscillator startup do not seem to be adversely affected.

4.4.3. Temperature

By regulating power to the processor cooling fan, the surface temperature of the chip was held within 3 degrees Celsius while consecutive samples of 512MB were read from the RNG. Occurrences of all pairs of consecutive bytes were tallied, resulting in 524,287,999 tallies in 65,536 bins. From these tallies, the following entropy and predictability figures were computed.

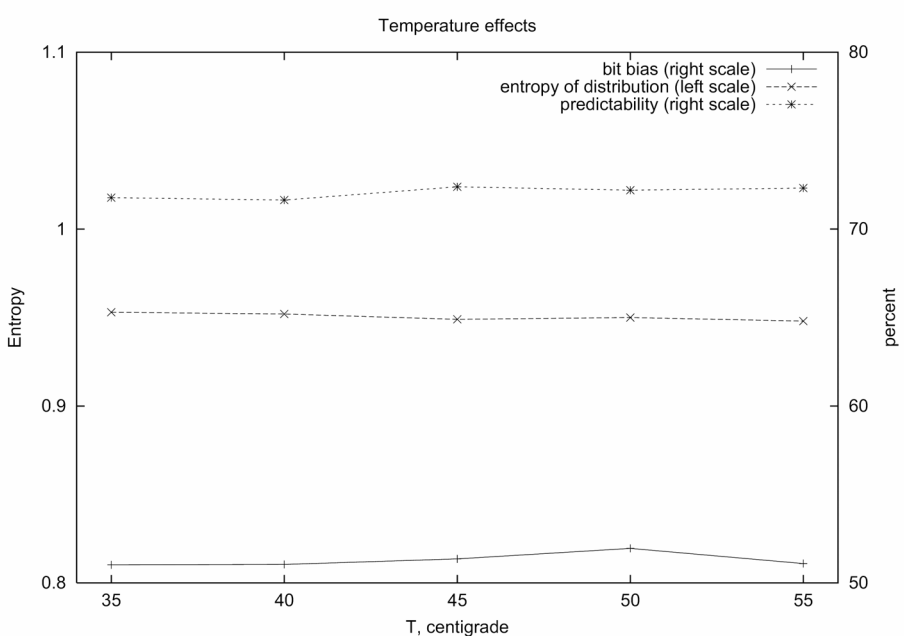

Figure 9: Temperature effects (raw output)

A very mild deterioration of the statistics is observed as the temperature rises.

4.4.4. Processor Activity

Three runs with consecutive samples of 512MB were made under different processor loading conditions. The results are presented in the following table.

CPU Load	single-bit bias	per-bit entropy	predictability
none	50.99	0.955	71.0
low-priority competing process	50.99	0.952	71.8
high-priority competing process	51.00	0.952	71.79

In general, competing processes are expected to increase the number of shutdown/startup cycles of the RNG oscillators, as process timeslicing increases the delays between subsequent XSTORE operations. We conclude that competition for the processor does not significantly affect the quality of the RNG output.

4.4.5. Modeled Operation

A program was written to simulate the simplest complete model of this RNG. This model system is deterministic and is characterized by the periods of the three fast oscillators, the two periods of the slow oscillator, and the four oscillators' phases. Changes to the slow oscillator's period were considered to happen instantaneously, and were made in such a way as to preserve the phase angle of the oscillator at the instant of change. An example of the modeled output of the XOR is shown in Section 3.1.

The simulator was used to study the effects of RNG parameters on the RNG output and to analyze the characteristics of deterministic models of the system. The simulator generates output with spectral content that differs substantially from the RNG output and from the output of a cryptographic PRNG.[8] However, the simulator output passes the FIPS-140 test suite fairly consistently (14 failures out of 320 tests). With whitening enabled and EDX=2, the simulator output passed all tests in the FIPS-140 suite.

[8] Spectral content was measured with FFT and autocorrelation tests.

Tests with the simulator provide a cautionary note: as with most oscillator-based RNG designs, there is no obvious test that can prove that the output is truly random instead of the result of a complex and unknown deterministic process. Although our simulator analysis and statistical tests are suggestive of high quality, RNG users for security applications should always make the most conservative choices possible and, if practical, incorporate entropy from all available sources.

4.4.6. Results from Centaur Technologies Testing

Cryptography Research was provided with an assortment of statistical tests that were run on the output of various chips running in various modes on various processors. About 14 different chips were tested. About half the tests (179,514) were run with undersampled output (EDX = 3) and the von Neumann whitener on. The remaining tests (134,301) were run at the full sampling rate (EDX = 0), the von Neumann whitener off, and SHA mixing of the output bits.

These tests measure the suitability of the RNG for numerical applications. They are not particularly relevant to security applications, since insecure pseudorandom bit generators have been known to pass these statistical tests.

The tests run with SHA mixing all produced excellent statistics. This is unsurprising, since good SHA mixing produces excellent statistics for all known input streams.

Tests run without SHA mixing tended to fail the demanding statistical tests that were applied. This does not prove that the random bits are particularly poor, since tests encompassing so much data will detect very small flaws in an RNG; but it does indicate that applications requiring high quality randomness should not use this RNG without mixing.

It is possible that by experimenting with the bias settings, a setting may be located that produces random bits with better statistics. However, every chip might not have such an optimal bias setting, and chip performance may drift with time and/or temperature. To minimize the likelihood of a problem, users of this RNG (as well as virtually any other

hardware-based entropy source) who require high quality randomness should use appropriate mixing functions.

4.5. Source Whitening: Extrapolating Test Results

The above RNG tests were performed with the von Neumann whitener deactivated. The usefulness of von Neumann whitening depends on the characteristics of the input bitstream and the intended use of the output bits. Specifically,

- von Neumann whitening works well on bits from memory-less generators.
- It works less well on bit streams with serial correlations.
- Its bit-balancing effect is important for numerical work.
- Under normal circumstances, it will improve the entropy of the output, although in some unusual (largely theoretical) situations it can reduce entropy.
- Its bit-balancing effect is often less important for cryptographic applications where post-processing is used to produce high-entropy output from lower-entropy sources.

To illustrate these points, consider the following simple examples.

Biased, independent bits. The whitener is ideally suited for input bits that are independent but biased. If each bit is 1 with probability p, then the entropy per input bit is $-p\log_2(p)-(1-p)\log_2(1-p)$, and the average number of output bits produced per input bit is $p(1-p)$. These two functions agree nicely, as shown in the following figure.

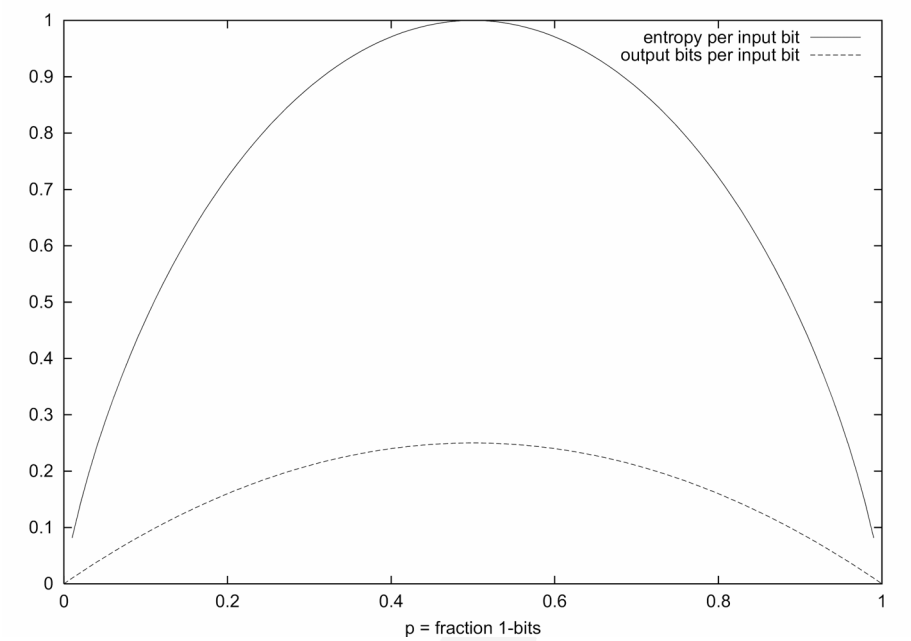

Figure 10: von Neumann correction of biased, independent bits

The rate at which output bits are produced never exceeds the rate at which input bits can be imagined as "bringing entropy" in, so it seems reasonable (and is in fact true) that each output bit holds a bit of entropy. In other words, each output bit from a von Neumann corrector brings one bit of entropy if the raw bits are independent but biased with any probability p between 0 and 1.

Serial correlation. Consider a bit generator where the probability that any two consecutive bits differ is p. If $p = \frac{1}{2}$, this is a perfect random bit generator. If p is small, it will produce long runs of identical values, and the whitener will seldom output a bit. If p is large (i.e., near 1), it will produce long patterns of alternating 0's and 1's, and the whitener will almost always output a bit. The corresponding plot for this generator follows. Toward the right-hand side of the chart, the rate at which bits are output exceeds the rate at which entropy enters the system; and indeed, the entropy per output bit falls sharply. In this region, a strong serial correlation remains among the output bits..

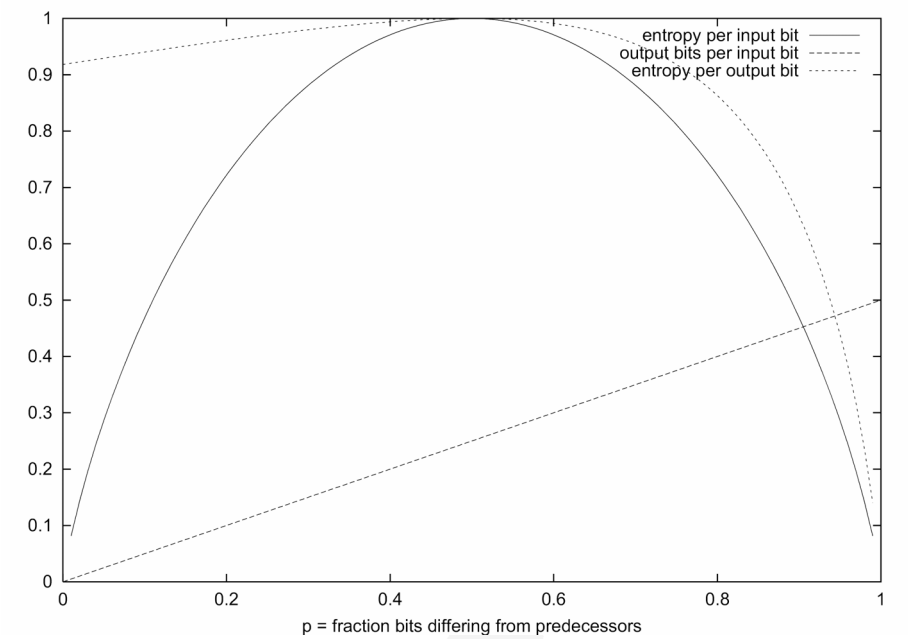

Figure 11: von Neumann correction of serially correlated bits

While the whitener is less effective at handling serially correlated bit data, these diagrams suggest that the whitener will at least *improve* the per-bit entropy of whitened bits in this case.

A series of whitened 512MB samples were taken from a single processor. As expected, the results had bit biases that were substantially smaller (within 0.015% of an expected 50-50 distribution), but still detectably imbalanced.[9] The worst-case bit predictability (calculated by predicting an output bit with knowledge of the 15 neighboring bits) was reduced to 55.1%, yielding entropy measurements of 0.99 bits of entropy per bit. The serial correlations of the output bits also diminish more quickly, as illustrated in the following two diagrams.[10]

[9] For this sample size, one standard deviation is 0.00156%, so the whitened distribution can still be considered statistically imbalanced.

[10] Serial correlation is viewed by plotting the percentage of 1's as a function of position after a 1 in the high-order bit position of a byte. (Ordinarily, positions after all 1 bits would be computed, but this weighing is used to avoid corruption by the missing bit after every byte.)

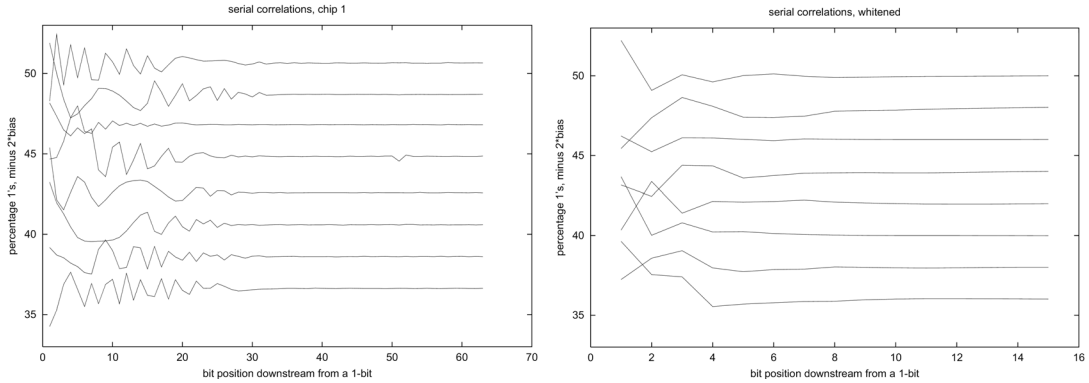

Figure 12: Serial correlations diminish more quickly when whitener is enabled

It is important to note that van Neumann correction can, at least in theory, reduce the entropy from a source. For example, a source whose output alternates between "0" bits and random bits contains 1 bit of entropy for each two output bits. Applying the whitener to this source would provide an output sequence of all zeroes (i.e., containing no entropy). Our tests did not reveal any patterns like this or any other output characteristics that would cause the whitening step to reduce the output entropy.

4.6. Entropy Conclusions

In a limited set of tests, the raw RNG source was found to have measured entropies between 0.78 bits per output bit and 0.99 bits per output bit. Changes in bias, temperature, stop/start behavior, and processor were not found to significantly reduce the RNG output quality. Use of the von Neumann corrector is expected to improve the quality of the output and was observed to do so rather substantially.

Overall, we believe that the Nehemiah RNG provides a high-quality and consistent source of entropy. In normal operation (whitener enabled, EDX divisor set to zero), users should typically obtain over 0.99 bits of entropy per output bit. In applications where the quality of the randomness is critical, developers should make more conservative assumptions about the randomness of the output. Our most conservative interpretation of our results suggests that sensitive applications should be able to assume that the Nehemiah RNG, after whitening and with EDX divisor set to 0, provides at least 0.75 bits of entropy per output bit. (In practice, we recommend that developers make even more

conservative assumptions, since the high data rates offered by the Nehemiah RNG allow the collection of extra output without any significant performance penalty.)

5. Usage Recommendations

5.1. Usage Modes

This section describes recommended procedures for the use and initialization of the Nehemiah RNG. It is addressed to three primary audiences:

- *Developers of OS-level resources*. The Nehemiah RNG serves as a fast, high-quality entropy source for existing randomness resources (the /dev/random and /dev/urandom UNIX resources are perhaps the most well known). Because these resources can be used by a variety of applications, developers of OS-level randomness resources must incorporate the RNG in a manner that produces cryptographically strong random numbers, makes appropriate entropy estimates, operates efficiently, and resists attacks where "hostile" and "trusted" code may share the same resource.
- *Developers of high-security applications*. Some high-security applications (such as key generation and challenge-response protocols) require that individual applications be provided with a cryptographically strong source of randomness. Developers of these applications may want to increase assurance by obtaining random data directly from the Nehemiah RNG as (or in addition to) their primary randomness source.
- *Developers of non-secure applications with lower entropy requirements*. Some applications (such as those that currently use the C "`rand()`" function) typically have less stringent entropy, security, and statistical requirements.

Our usage recommendations are primarily designed for applications where an extremely high quality source of randomness is required. Developers of non-secure applications that do not require such high quality randomness (the third category) can always elect to use these numerically-conservative methods for random number generation. Alternately, developers can use OS-level randomness resources that incorporate output from the

Nehemiah RNG. Section 5.5 of this document provides guidelines for developers of non-secure applications who prefer to extract data directly from the Nehemiah RNG.

5.2. Feeding an Entropy Pool with the Nehemiah RNG

As with any hardware entropy source, it is recommended that the Nehemiah RNG be used as a source of seed data for a cryptographically strong PRNG or hash function. If implemented correctly, cryptographic functions can produce high-quality random data (ideally with one bit of entropy per output bit) from a larger quantity of source data with less than one bit of entropy per bit.

In such a design, output from XSTORE calls is incorporated into a buffer by a process referred to as *seeding* an entropy pool. The entropy pool may be occasionally compressed or "stirred" with a *mixing* function, which combines entropy across RNG data. Finally, application requests for random numbers are served by an *output* function, which transforms data in the entropy pool to generate outputs that can be used as random numbers by the application.

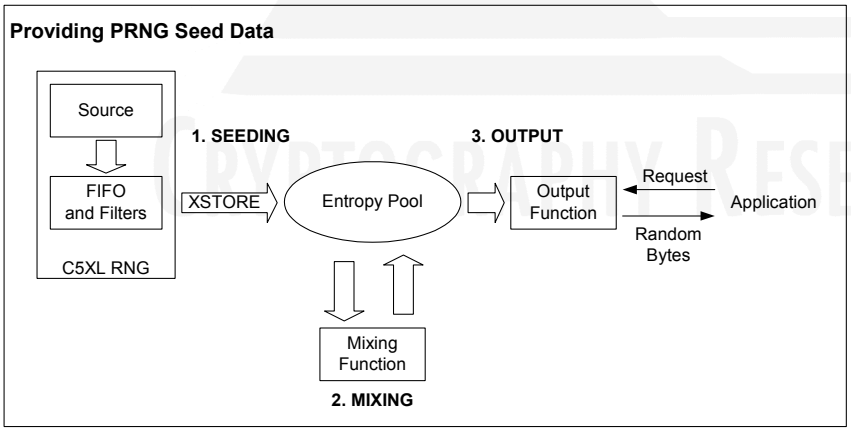

Figure 13: Seeding a PRNG with the Nehemiah RNG

Because the *mixing* and *output* functions are a series of deterministic operations, this construction is referred to as a Pseudo Random Number Generator (PRNG) if the design allows the number of output bits to exceed the entropy of the seed data. Cryptographic functions can also be employed as part of a true RNG to compress larger amounts of partially-random seed data to produce higher-quality output. A well-designed PRNG

serves as an entropy buffer in which the various rates of *seeding* and *output*, as well as the quality of the seed material, determine the quality of the output. Cryptographically-based PRNG designs often use hash functions (such as SHA-1) or modular arithmetic, which have desirable properties for *mixing* and *output* functions. A complete discussion on PRNGs is outside the scope of this document.[11] The use of cryptographic processing offers the following advantages:

- *Improved per-bit entropy*. The per-bit entropy of the Nehemiah output can be improved by integrating RNG output into a seed pool at a rate greater than the rate at which entropy is extracted from the seed pool.
- *The entropy pool has inertia*. The state within the seed pool of a PRNG enables bursty applications (such as SSL key generation) to consume random data at a rate that temporarily exceeds the bitrate of the source.
- *Protection against some failure modes*. It is difficult to certify that an RNG will never experience "hiccups" or local conditions that may result in comparatively low entropy output. A properly-seeded PRNG can "ride out" catastrophic source RNG failures while maintaining levels of unpredictability that are suitable for many applications.
- *An additional margin of safety*. A properly-constructed mixing function will combine entropy that has been collected over time with "fresh" entropy from the source. This will result in output entropy that should be better (or at least no worse) than using the source directly.

PRNGs with better sources of seed data can provide higher-assurance output. In the UNIX /dev/random resource, event and timing data from low-rate events (such as mouse movement or process timings) is commonly used for seeding when no hardware source is available. Because estimating the entropy provided by such events is difficult and because they occur relatively infrequently, the amount and quality of the seed data available to /dev/random is limited. The Nehemiah RNG provides a consistently high

[11] More information on PRNGs can be found in *Yarrow-160: Notes on the Design and Analysis of the*

level of entropy at substantially higher rates. Using the Nehemiah RNG as a source of PRNG seed data should improve the assurance of applications that require randomness.

5.3. RNG Command overview

This section introduces specific Nehemiah RNG commands. Considerably more detailed information is available in Sections 3 and 4 of the *VIA C5XL Processor Random Number Generator Application Note*.

5.3.1. The Configuration MSR

The RNG is controlled through a 32-bit status/configuration MSR (0x110B). While the MSR itself is only accessible to ring 0 code, most MSR contents are copied to EAX on every completed XSTORE operation, and some MSR bits can be probed by examining the Centaur Extended CPUID Feature Flags. It is therefore possible for user applications to view (but not change) the status of the MSR.

Cryptography Research recommends setting the MSR to 0x00000020 (RNG enable, all other settings 0) as part of the operating system boot process. Multi-application systems should avoid making any further modifications to this register. The MSR register settings are shown below:

MSR 0x110B

31:22	21:16	15	14	13	12:10	9:8	7	6	5	4:0
Reserved	String Filter Count	String Filter Failed	String Filter Enable	Raw bits enable	DC bias	Reserved	Reserved	RNG enable	Reserved	Current Byte Cout
	Read/ Write	Read/ Write	Read/ Write	Read/ Write	Read/ Write			Read/ Write		Read only

Source: Centaur Technology

Disable string filter (clear MSR bits 21:13). Because this feature can introduce significant biases in the output, it should only be used for testing purposes and should be disabled during normal operation. (While the alarm bit feature, described in Section 3.2, could be used in some applications to detect a catastrophic RNG failure, it is difficult to properly handle such alarm conditions in multi-application environments. As a result, applications should implement any required quality monitoring in software.)

Yarrow Cryptographic Pseudorandom Number Generator, by J. Kelsey, B. Schneier, and N. Feguson.

Activate von Neumann corrector (clear MSR bit 13). As discussed in Section 3.1, the corrector discards bits to restore a uniform per-bit frequency. The corrector also increases the temporal separation between consecutive bits, which should improve the per-bit entropy.

Use default bias setting (clear MSR bits 12:10). The oscillation frequency of each RNG oscillator is regulated by the 3-bit bias voltage setting. While the choice of bias setting is somewhat arbitrary, the default setting should be used for cross-application consistency.[12] Secure applications should also check that the bias setting does not change from the default.

Reserved bits. Centaur reports that the reserved bits are currently ignored. Applications should never set any reserved bits.

5.3.2. The XSTORE Operation

The XSTORE operation is an extended x86 instruction that can be run from any application. A complete set of documentation for the XSTORE command can be found in the *VIA C5XL Processor Random Number Generator Application Note*. The following recommendations apply to the use of the XSTORE command in security applications:

- *Prior to calling XSTORE, verify that the RNG is enabled*. Using XSTORE without a present and enabled RNG will result in an exception or otherwise unpredictable behavior.
- *Specify the correct divisor*. On each XSTORE call, the 2 lowest bits of EDX specify the divisor, or the rate at which random data is to be returned. Applications that fail to specify the divisor correctly may receive fewer random bytes then expected.
- *Returned data*. XSTORE may return (or overwrite) up to 8 bytes. At least 8 bytes of output memory must be allocated to avoid buffer overruns. At EDX=0, the RNG returns bytes in the following order: older bytes are stored in lower byte addresses; within each byte, the most significant bit is the oldest.[13]

[12] While somewhat better statistics have been measured at a bias setting of 2, the bias setting must be shared by all applications running on the system. Future versions of the Nehemiah and successor parts may also have slightly different "optimal" settings. As a result, it is advisable to use the declared default value of 0 (as written in the *VIA C5XL Processor Random Number Generator Application Note*).
[13] Note: the relative byte and bit ordering may change when the EDX divider is nonzero.

- *Inspect RNG configuration on each call*. Before interpreting data returned by the XSTORE call, the application should inspect the MSR configuration (as copied to EAX) to ensure that the RNG configuration is correct and unchanged. Because the MSR configuration should be checked on each XSTORE call, the REP prefix should not be used.

5.3.3. Determining RNG Availability

Before using or initializing the RNG, developers should ensure that a VIA Nehemiah processor containing the RNG feature is present. User-level applications should ensure that the RNG is enabled and that SSE instructions are enabled before performing XSTORE operations. Section 3.2 of the *VIA C5XL Random Number Generator Application Note* defines procedures for:

- Determining if a VIA Nehemiah processor is present
- Determining if the RNG feature is present (Centaur Extended CPUID Flag EDX[2])
- Determining if the RNG is activated (Centaur Extended CPUID Flag EDX[3])
- Determining if SSE instructions are enabled

5.4. Recommended Usage Procedures

5.4.1. Boot-Time Initialization

At system boot, OS code (ring 0) should initialize the RNG in the following sequence:

(1) The code should verify that that the processor contains a Nehemiah core with a RNG that may be enabled. This is done by examining the Centaur Extended CPUID Flags. (Consult Section 3.2.2 of the *VIA C5XL Random Number Generator Application Note*.)

(2) SSE instructions should be enabled, if necessary. (The RNG feature requires that SSE instructions be enabled. For details, consult Section 3.2.1 of the *VIA C5XL Random Number Generator Application Note*.)

(3) Ring 0 code should write the value 0x00000020 to MSR 0x110B. This will enable the RNG, turn on the von Neumann corrector, set the offset bias to zero, and turn off string filtering (see Section 5.3.1 for more information about MSR settings).

(4) Finally, the initialization code should also perform a simple RNG usage test (see *"Application Start Sequence"* below).

5.4.2. Application Start Sequence

At application start, the RNG should be verified in the following sequence. An error or unexpected result in any step should result in a user-visible error message and should prevent the RNG from being used within the application.

(1) The code should verify that that the processor contains a Nehemiah core with an enabled RNG. This is done by examining the Centaur Extended CPUID Flags, particularly EDX[2] and EDX[3]. (Consult Section 3.2.2 of the *VIA C5XL Random Number Generator Application Note*.)

(2) The RNG feature requires that SSE instructions be enabled. (Consult Section 3.2.1 of the *VIA C5XL Random Number Generator Application Note*.)

(3) The application should perform at least one XSTORE operation. This should be done as listed in "*Extracting Random Data"* below. This section discusses how the MSR contents should be validated before RNG use is permitted.

(4) The application may optionally perform tests of the entropy source. At a minimum, applications should read 16 bytes (using "*Extracting Random Data"* below) and ensure that the bytes are not all "0xFF" and not all "0x00". (The probability of a correctly-operating RNG failing this test is 2^{-127}, or one in 170 trillion trillion trillion.)

(5) If the application requires the initial seeding of an entropy pool, "*Extracting Random Data"* (below) should be repeatedly run until a suitable level of starting entropy is achieved.

(6) If any errors occurred in performing these steps, the Nehemiah RNG should not be used and, if possible, an error message should be presented to the user.

5.4.3. Extracting Random Data

The following sequence should be run whenever random data is required:

(1) *Prepare for XSTORE call*. ES:EDI should point to a low-end address of a buffer of at least 8 bytes.[14] EDX should be cleared (EDX=0) to specify reading all 8 bytes from the queue.

(2) *Execute XSTORE*. The XSTORE instruction (opcode 0x0F 0xA7 0xC0) will return either 0 or 8 bytes of data in ES:EDI. It will also return the state of the MSR register in EAX. Note that an Invalid Instruction exception will occur if XSTORE is executed when the RNG is not enabled.

(3) The contents of the MSR (which is placed into bits 31:5 of register EAX) should be inspected to verify that:

   a. RNG is activated (bit 6 is set).
   b. von Neumann corrector is turned on (bit 13 is cleared).
   c. String filter is deactivated (bit 14 is cleared).
   d. Bias voltage is left at the default setting (bits 12:10 are cleared).

   If any of these criteria are not met, the RNG should not be used and, if possible, an error message should be presented to the user. The MSR should be inspected on *each* XSTORE call, as it is possible that another application may have changed the MSR settings. For the same reason, XSTORE should generally not be executed with the REP prefix, as MSR changes during a context switch may not be noted.

(4) If bits 4:0 of register EAX=0 (no bytes read), return to step 2 unless the request has timed out. If bits 4:0 of register EAX=8, then proceed. If EAX is a value other than 0 or 8, an error message should be presented to the user and the RNG should no longer be used by the application.

At the successful completion of this process, the 8 bytes of RNG output (stored from the initial value of ES:EDI) are available for contribution to the application entropy pool.

[14] In general, to avoid buffer overflow problems, the XSTORE return buffer should always contain at least 8 allocated bytes (regardless of EDX setting).

For systems that perform entropy estimates, the application may assume that the 8-bytes of output contain 48 bits of entropy.[15]

5.4.4. Maintaining an Entropy Pool

Output from the Nehemiah RNG should normally be used to seed an entropy pool. The output can be used to directly seed an entropy pool as performed by the UNIX /dev/random driver.[16] The output may also be used to seed an intermediate entropy pool that is periodically hashed and contributed to a primary entropy pool, as in the Yarrow PRNG.[17] While relative abundance of high quality output provided by the Nehemiah RNG can ease design concerns, developers should always avoid re-designing security code if a free and well-reviewed design already exists. At a minimum, a single secure application may use a cryptographically strong "whitening" function to mix RNG outputs together. A suitable example would be the SHA-based random-number generator described in FIPS 186-2.[18]

The following are usage recommendations are provided:

- The developer should define a value for entropy_min, or the minimally acceptable level of entropy. For security applications, entropy_min should be at least 128 bits. As a security bound, no more than $2^{entropy\_min/2}$ bits (e.g., 2^{64} bits) should be extracted from the PRNG before it has been completely reseeded. (This is not generally a practical concern, as entropy_min should be large enough that it is infeasible to obtain $2^{entropy\_min/2}$ output bits.)
- The PRNG should have an entropy pool at least twice the size of the level of security desired. For security applications, the PRNG should have a pool of 128 bits at an absolute minimum, and a pool of at least 256 bits is strongly recommended.

[15] This is consistent with an entropy estimate of 0.75 bits per whitened output bit. Sensitive applications can further derate this entropy estimate with minimal penalty.
[16] The Linux programmers manual for /dev/random can be found at http://www.rt.com/man/random4.html.
[17] *Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator*, by J. Kelsey, B. Schneier, and N. Ferguson, http://www.counterpane.com/yarrow.html.
[18] Appendix 3.1 of FIPS 186-2, http://csrc.nist.gov/publications/fips.

- On initialization, the Nehemiah RNG should contribute four times[19] the minimally acceptable level of entropy to the entropy pool. For example, if a minimum entropy level of 256 bits is required, 22 XSTORE calls should be performed.[20] The initialization code should also verify that the RNG does not output all "0" or all "1" bits. The complete initialization process should finish in under 0.5ms.
- Because the XSTORE command returns quickly if no data is present in the FIFO, time-sensitive resources (such as /dev/random) may call XSTORE multiple times before adding the contents of the return buffer to the pool (4 consecutive calls to XSTORE generally exhausts the FIFO).

5.5. Random Numbers for Non-Secure Applications

Although post-processing is recommended for all hardware-based entropy sources, some developers of non-secure applications may elect to use the Nehemiah RNG without hashing or mixing. Such developers applications should consider the following procedures:

- *Boot-time initialization*. At system boot, the RNG should be initialized as recommended above.
- *Application initialization*. At application start, the presence of the RNG should be verified as recommended above, with the exception that calls to extract random data from the RNG should use the procedures below.
- *Extracting random data*. The XSTORE command should be setup and called as recommended above, except that:
    - When preparing for the XSTORE call, the two lowest bits of EDX should be set (EDX=0x00000003). This discards FIFO data (7 of every 8 bits are discarded) to increase the temporal distance between adjacent bits.

[19] The suggestion of a 4X safety factor does not reflect a specific concern about the RNG design. Instead, the high performance of the Nehemiah makes it practical to add a sizable safety margin for critical operations.

[20] 256 bits entropy * 4X safety factor / 0.75 bits entropy per output bit / 64 bits per XSTORE call = 21.33 XSTORE calls.

- XSTORE returns 0 or 1 byte of data (instead of 0 or 8 bytes).
- If the MSR register (as copied to register EAX) does not match the criteria listed in Section 5.4.3, it may still be possible to use the data byte. (This would be an application-specific decision).
- Although the data should be of sufficient quality to pass most randomness tests, the application should not be provided with a formal entropy estimate.

6. Closing Commentary

The Nehemiah random number generator meets the overall design objective of providing applications with a high-performance, easy-to-use random number generator.

Entropy. Cryptography Research tested multiple processors in a range of environmental conditions, oscillator stop/start usage patterns, and conditions found in multi-application environments. In tests of the raw source (with the whitener disabled), Cryptography Research obtained entropy estimates of 0.78 to 0.99 bits of entropy per raw output bit. Operation with the whitener enabled is believed to provide 0.99 bits (typical) of entropy per output bit. We recommend that applications requiring high quality randomness use the RNG as a source of random seed data for a cryptographically strong PRNG or mixing function. When seeding, applications should assume that the RNG provides 0.75 bits of entropy (or less) per output bit, although even more conservative values should be used if there is no significant performance impact.

High output rates. The RNG generates output at significantly higher rates than most PC-based randomness resources. Raw bits are produced at rates of 30 to 50 Mbits/sec, and whitened bits were observed at rates of 4 to 9 Mbits/sec. We estimate that PRNGs seeded with the Nehemiah RNG can sustain output in excess of 2 Mbits of entropy per second, which should eliminate blocked PRNG reads in virtually all applications. Even more important, the high bit rates enable sensitive applications to use more conservative per-bit estimates of entropy with little performance penalty.

On-die location. Unlike all PC-based random number sources known to the authors, the Nehemiah RNG is located entirely on the microprocessor die. This location provides applications with convenient and well-controlled access to the RNG. It is also the most logical location to place a system security component. Although the Nehemiah processor is not designed to serve as a platform for tamper-resistant applications, this location also increases the effort required to physically tamper with the RNG subsystem.

Multi-application support. The RNG is well designed to support simultaneous use by multiple applications. Only OS-level (ring 0) code is provided with write access to the RNG configuration register. Because XSTORE copies the MSR settings to user space, applications may validate the RNG configuration on each XSTORE call. RNG data is only output once, and altering any MSR settings flushes any unread RNG data. Future hardware versions can further improve application separation by discarding more bytes (and therefore increasing temporal distance) between filling the 8-byte FIFO buffers.

Excellent visibility of raw source. The ability to disable the RNG whitener and adjust the EDX divider enables a wide set of tests to be meaningfully applied. The designers should be commended for including raw source access and 1:1 divider access, enabling application developers to perform source validation and improve their levels of source assurance. These features will enable a significantly larger population of developers and researchers to study the RNG source.

User options. The provisioning of many user settings (bias, string filter, EDX divider, and whitener settings) is a double-edged sword. Some features, such as bias adjustment and string filtering, are beneficial for testing and for advanced developers of embedded systems with unusual requirements, but could cause problems if used in multi-application systems. The Nehemiah balances these challenges by limiting configuration changes to ring 0 code while allowing all applications to verify that the device is properly configured.

Manufacturing test. Documentation states that Nehemiah parts failing RNG manufacturing tests will have the RNG permanently disabled. While application software can detect a disabled RNG, the presence of Nehemiah parts without an RNG

capability may contribute to end-user confusion. (This manufacturing choice provides VIA with the assurance that the RNG feature will not reduce the manufacturing yield. As VIA gains experience manufacturing RNG-enabled parts, procedures for handling parts that fail manufacturing tests may change.)

Our analysis indicates that the Nehemiah core Random Number Generator is a suitable source of entropy for use in cryptographic applications. The RNG can be easily incorporated within existing software applications and operating systems and functions well within a multi-application environment. The device meets the overall design objective of providing applications with a high-performance, high-quality, and easy-to-use random number generator.