VIA C5P
Security Application Note
Version 0.98

This is **Version 0.98** of the VIA C5P Security Application Note.

VIA Technologies and Centaur Technology reserve the right to make changes in its products without notice in order to improve design or performance characteristics.

This publication neither states nor implies any representations or warranties of any kind, including but not limited to any implied warranty of merchantability or fitness for a particular purpose. No license, express or implied, to any intellectual property rights is granted by this document.

VIA Technologies and Centaur Technology make no representations or warranties with respect to the accuracy or completeness of the contents of this publication or the information contained herein, and reserves the right to make changes at any time, without notice. VIA Technologies and Centaur Technology disclaim responsibility for any consequences resulting from the use of the information included herein.

LIFE SUPPORT POLICY

VIA processor products are not authorized for use as components in life support or other medical devices or systems (hereinafter called life support devices) unless a specific written agreement pertaining to such intended use is executed between the manufacturer and an officer of VIA.

1. Life support devices are devices which (a) are intended for surgical implant into the body or (b) support or sustain life and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. This policy covers any component of a life support device or system whose failure to perform can cause the failure of the life support device or system, or to affect its safety or effectiveness.

Table of Contents

CHAPTER

1

INTRODUCTION

This chapter introduces the VIA C5P processor security features.

The recent explosive growth of PC networking and Internet based commerce has significantly increased the need for good computer and network security mechanisms.

Good security requires good random numbers. Which can be a problem: it is actually quite hard to generate *good* random numbers on a computer. (For those who don't understand what a random number is, why they are needed in computer applications, what a *good* one is, and how they are generated, Chapter 2 includes background on these topics.)

To address this need for good random numbers in security applications, VIA introduced the C3 Nehemiah processor in January, 2003, which includes a high-performance hardware-based random number generator (RNG) *on the processor die*. This RNG uses random electrical noise on the processor chip to generate highly random values at an extremely fast rate. It provides these numbers directly to the application via a new x86 instruction that has built-in multi-tasking support. By including this unique capability directly within the processor, the VIA C3 Nehemiah processor enables a new level of security for personal computers and embedded devices.

With the C5P processor (stepping 8 of the C3 Nehemiah), VIA has added a second RNG on the processor die, which improves the bit generation speed and effective randomness.

Good security also requires good cryptography. Most people understand the need for cryptography; we've all had our secret decoder rings and thrilled to the stories of espionage and the intrigue of spy versus spy.

Good cryptography is hard, though, conceptually. Getting the algorithms right, and getting the right algorithms, and using them in a secure fashion (or even in just a relatively less vulnerable fashion), is a problem that has plagued the brightest minds for generations. Get it right, and you may or may not get a

footnote in the history books and a raise in your next paycheck; get it wrong and your country may lose the war. Or your husband may find out ... (oops, wrong article).

Fortunately (for the engineers at Centaur), there are now accepted, standard algorithms for performing cryptographic functions. Examples include Rivest-Shamir-Adleman (*RSA*) encryption, the Data Encryption Standard (*DES*) and triple-DES (*3DES*), and more recently, the Advanced Encryption Standard (*AES*), as specified in Federal Information Processing Standards Publication 197 (*FIPS-197*.)

Getting algorithms right is something computer engineers are good at. Making them fast is something we are even better at. Some of our readers may remember calculating trigonometric functions with eight bit integer computers. And how much easier and faster it became when hardware floating point primitives became available on PCs. We can do the same for cryptography.

To address the need for *fast* and *easy-to-use* cryptographic primitives, VIA introduces on the C5P processor a high performance implementation of the Advanced Encryption Standard. This new feature, the *Advanced Cryptography Engine (ACE)*, provides increased cryptographic capabilities for personal computers.

1.1 RANDOM NUMBER GENERATOR

The VIA C5P processor RNG has many powerful and unique features:[1]

High Quality

■ **Hardware-Based.** Truly random numbers require a non-deterministic source (as opposed to a programmed software algorithm). The VIA C3 Nehemiah processor random bits are obtained from electrical random noise on the processor chip and meet the requirements of truly random numbers: statistically good, unpredictable, and unreproducible.

■ **Quality vs. Bit Rate Load Option:** Application code can control a hardware filter that selects and returns one-of-N bits generated (to reduce potential bit-to-bit correlation). Four levels of quality filtering can be selected. The slowest setting is required for best randomness in the returned values. The faster settings provide good entropy for applications that use the generated bits as a seed to another (software) algorithm which produces the random values actually used by the application.

■

■ **Designed for Verification and Test.** Because the RNG allows direct access to the entropy source, third parties can perform their own tests of the output.

■

High Performance

■ **High Bit Rate.** The VIA C5P processor RNG produces bits at a variable rate. Typical rates vary from approximately 800K to 1600K bits per second (depending on whether both noise sources have

[1]Patents pending.

been selected) at the best quality rate, to over 100M bits per second at the fastest setting. This is from 10 times to 1000 times faster than the next fastest RNG hardware solution that we know of: the Intel RNG included in the 82802 Firmware Hub. Our higher rates address the needs of applications requiring high bit rates and can be used with software that algorithmically increases the quality (randomness) of the numbers produced, for example by applying hashing algorithms to the output.

■

■ **Asynchronous Multi-byte Generation.** The hardware generates random bits at its own pace. These accumulate into hardware buffers with no impact to program execution. Software may then read the accumulated bits at any time. This asynchronous approach allows the hardware to generate large amounts of random numbers completely overlapped with program execution. This is opposed to good software generators, which can be fast but consume a significant number of cycles.

■

An Efficient and Secure Programming Interface

■ **Application-level Access:** Application code can store the random numbers directly from the hardware, without the intervention of privileged device drivers. A new x86 instruction is provided to store random data and the associated valid byte count. In addition, a `REP` version of the instruction allows large amounts of random data to be collected with a single instruction. Direct access ensures that applications receive data directly from the random number generator, avoiding the possibility of receiving nonrandom data due to malicious or accidental device driver flaws.

■

■ **Transparent to the Operating System:** The operations involved in obtaining random bytes and verifying the configuration operate in user mode, without operating system knowledge or involvement. For security purposes, changes to the configuration require a higher privilege level.

■

■ **Inherent multitasking (Atomic Instruction):** Storing random data is a single atomic instruction within an application. If a task switch occurs, and a different application wants to store random data, it gets completely new bits.

■

Many Optional Features

■ **Contiguous String Filter:** This diagnostic feature, when enabled, ensures that no contiguous string of zeroes or ones longer than a certain user-specified number (up to 64) can be generated. *Please note that this feature may be discontinued in subsequent processors.*

■

■ **Raw Bit Access.** The raw internally generated random bits may be accessed. Normally the raw bits are filtered through a *whitener* to obtain desirable random characteristics.

■

■ **Controllable Voltage Bias.** The internal voltage bias to the various oscillators can be adjusted.

1.2 ADVANCED CRYPTOGRAPHY ENGINE

The VIA C5P processor ACE also has many powerful and unique features.[2]

General Characteristics

■ **World-Class Data Rate.** Operating on a long series of data blocks, the ACE can encrypt or decrypt data at a sustained rate of 12.8 Gb/sec.[3] This is faster than any known commercial AES hardware implementation, and several times faster versus software implementations. For a single encryption or decryption, the effective rate can be even faster: up to 21 Gb/sec.[4]

■

■ **It's Free.** The unique ACE implementation is so small (approximately 0.6 mm^2) that its inclusion on the VIA C5P processor did not change the die size or any other processor cost factor.

■

■ **It's Non-Intrusive.** Although the ACE provides a significant amount of function with many options, only one new x86 opcode is used. This is the same new x86 opcode that the VIA C3 Nehemiah processor introduced for the random number generator.

■

■ **Direct Application-level Access.** Application code can directly encrypt and decrypt blocks without the intervention of any privileged code. One new non-privileged x86 instruction provides all of the ACE functions *without any need for a device driver or for a change to the operating system.* In addition to facilitating the ACE availability, this application-level capability improves security by ensuring that all related cryptographic information is kept within the application.

■

■ **Inherent multitasking.** In addition to any single application being able to directly use the ACE features, any number of tasks can use the ACE concurrently without supplemental task management by the application or by the operating system. Although the ACE implementation contains additional x86 state, the using tasks do not need to save and restore this state - the hardware manages the additional state in a transparent fashion.

■

Significant Function Provided

■ **Performs Both Encryption & Decryption.** This seems obvious, but many hardware AES engines support encryption only. In addition to providing both encryption and decryption, the ACE provides both with the same performance.

■

[2] Patents pending.

[3] Asymptotic REP instruction performance, ECB mode, 1 GHz C5P processor. Future VIA processors running at 2 Ghz will encrypt at over 25 Gb/sec.

[4] The timing of the encryption and decryption operations is effectively variable since a large portion of the time can be overlapped with subsequent instruction execution. This rate assumes the maximum possible instruction overlap.

■ **All Three AES Key Sizes Are Supported Directly in Hardware.** FIPS-197 defines three key sizes (128-bits, 196-bits, and 256-bits) but requires only that one key size be supported. Many hardware AES engines support only the basic 128-bit key size. The VIA C5P processor ACE directly supports all three key sizes in hardware with the same performance.

■

■ **Four Operating Modes Supported Directly in Hardware.** The most common encryption *operating modes* are implemented in hardware. These modes are *Electronic CodeBook* (ECB), *Cipher Block Chaining* (CBC), 128-bit *Cipher FeedBack* (CFB), and 128-bit *Output FeedBack* (OFB).[5] Providing these operating modes directly in hardware improves performance (versus implementing them in software) and improves the integrity and security of the encrypted data. Most other AES devices support only one or two of these modes in hardware.

■

■ **Multi-block pipelined.** For the ECB operating mode, the ACE hardware is pipelined such that it concurrently operates on two data blocks at the same time, thus doubling the effective data rate.

■

■ **High-Performance REP Function.** The basic encryption and decryption instructions all use the x86 `REP` prefix, to allow large amounts of data to be encrypted or decrypted with only one instruction. Unlike x86 string operations, where multiple string operations are (slightly) faster than REP forms, the REP encryption and decryption instructions are much faster than a series of comparable single encryption instructions - generally 50% or more.

■

Significant Flexibility Provided (Optional Features)

■ **Intermediate Round Results Provided.** The ACE hardware implements a special mode that returns the result of any particular round of the multi-round AES algorithm. Properly calculating this intermediate result is not obvious since the algorithm for intermediate rounds is different from the last round of the normal algorithm.

■

■ **User-Defined Round Count.** The user specifies the round count used in the encryption or decryption. Normally, this will be the round count defined in the AES standard, but some users may prefer a custom round count for research, improved performance, or increased security.

■

Two Extended Key Generation Options. The ACE hardware has a feature that directly performs the AES-defined expansion of the user-supplied key into the extended keys used by the AES rounds. In addition, the application may directly provide a standard or non-standard extended key sequence directly to the hardware. This may be useful for cryptographic research or using a private encryption algorithm.

[5] As defined in *NIST Special Publication 800-38A, Recommendations for Block Cipher Modes of Operation,* 2001

CHAPTER

2

ON RANDOM NUMBERS

This chapter provides background material on random numbers for computers.

For those who do not already know what random numbers are, what applications use them, how they are generated, and why most current generators do not really produce random numbers, this chapter provides a general background.

Apology to mathematicians, statisticians, security experts, and information scientists: this background introduction intentionally simplifies some very complex technical topics and skips many important considerations.

2.1 THE NEED FOR RANDOM NUMBERS

To help understand the definition of a random number for computer applications, we need to understand something about how they are used. Historically, many computer applications have required random numbers as an essential component of their logic. For example,

- Monte Carlo simulations of physical phenomena in physics, chemistry, mechanical engineering, electrical engineering, meteorology, statistics, aerodynamics, hydrodynamics, and other similar disciplines.
- Numeric analysis solutions for many mathematical problems such as solving integral and differential equations, solving linear and non-linear equations, etc.
- Casino games and online gambling (to simulate card shuffling, dice rolling, etc.)
- Creation of lottery numbers
- Generating data for statistical analysis
- Generating data for psychological testing
- Computer games (choosing random behaviors, dealing cards, etc.)
- Computer music generation
- ■

These applications have different requirements for their random numbers. For example, computer games generally require neither high performance in generating random numbers nor particularly high degrees of randomness. Other applications, such as psychological testing, require higher degrees of randomness, but do not require high performance generation. Large-scale Monte Carlo-based simulations, however, use millions, billions, or more random numbers, and thus have very high performance requirements.[6] These applications also require good statistical properties of the random numbers, but unpredictability is not particularly important. Other applications, such as online gambling, have very stringent randomness requirements as well as stringent non-predictability requirements (the next number cannot be easily predicted from the previous numbers).

While these historical applications are important to PCs, today the major need for quality random numbers is *computer security*. The recent explosive growth of PC networking and Internet based commerce has significantly increased the need for a wide variety of computer security mechanisms. High quality random numbers are essential to all of the following major components of computer security:[7]

Confidentiality. Data encryption is the primary mechanism for providing confidentiality. Many different encryption algorithms exist (symmetric, public-key, one-time pad, etc.), but all share the critical characteristic that the *encryption/decryption key must not be easily predictable*. The cryptographic strength of an encryption system depends on the strength of the key: that is, the difficulty of predicting, guessing, or calculating the key. To ensure that keys cannot be easily guessed, random number generators are used to produce cryptographic keys and other secret parameters in virtually all serious security applications. Many

[6] James E. Gentle, *Random Number Generation and Monte Carlo Methods,* Springer, 2002

[7] This document assumes some knowledge of security concepts, functions, and protocols. If you need some technical background on these areas and more about random numbers, we recommend the following three classics:

Bruce Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, John Wiley & Sons, 1996.

Alfred J. Menezes et al., *Handbook of Applied Cryptography*, CRC Press, 1997

Donald E. Knuth, *The Art of Computer Programming: Volume 2, Seminumerical Algorithms,* AddisonWesley, 1998. Chapter 3 covers random number theory in some detail.

public key encryption protocols, such as RSA encryption using PKCS #1, also require reliable sources of randomness.

Many successful attacks against cryptographic algorithms have focused not on the encryption algorithm but instead on its source of random numbers. As a well-known example, an early version of Netscape's Secure Sockets Layer (SSL) collected data from the system clock and process ID table to create a seed for a software pseudo-random number generator. The resulting random(?) number was used to create a symmetric key for encrypting session data. Two graduate students broke this mechanism by developing a procedure for accurately guessing the random number in less than a minute and thus guessing the session key.[8]

Authentication. Challenge/response authentication protocols require that challenge values be unpredictable to ensure that attackers cannot re-use data from previous authentication transactions. The strength of passwords used to protect access and information also depends on the difficulty of predicting or guessing the password. As a result, strong random number generators are necessary to automatically generate strong passwords.

Integrity. Digital signatures and message digests are used to guarantee the integrity of communications over a network. Random numbers are often used in digital signature algorithms to make it difficult for a malicious party to forge the signature. Many signing algorithms, including the U.S. Government's Digital Signature Standard[9] also require random sources to ensure the security of the signing keys.

In summary, good security requires good random numbers.

2.2 WHAT IS A RANDOM NUMBER

The notion of a random number is a non-intuitive concept. What is randomness? Can any *single* number be considered random? Can any *finite sequence* of numbers be considered random? Is it the *numbers* themselves, or the *process* that generated the numbers that defines randomness? These questions are the subject of many technical papers, doctoral theses, and bar fights among philosophers, mathematicians, statisticians, and information scientists.

One essential fact emerges from this work: ***there is no such thing as a random number***: there are only methods to produce random numbers.[10] A number by itself (or any finite sequence of numbers) is not random, or non-random. (Is zero any less random than 42?) A practical definition of randomness must focus on the characteristics of the random number generator as well as the characteristics of numbers it generates.

To identify the desired characteristics, we consider the theoretical definition of a perfect random number generator. A *truly random number* generator is one where the probability of each generated bit being a zero (or one) is exactly ½, regardless of what it has generated before.[11] For example, an unbiased coin flip should generate truly random numbers. This definition has many implications beyond the statistical characteristics of the bits generated. It implies that the generator is *unpredictable* (one can't predict the next coin toss better than ½ the time, and *unreproducible* (two separate coin flipping experiments give independent results).

Entropy is a related term that we will use a lot in this document. In general, entropy is the measurement of randomness or chaos. Something that has high entropy is chaotic and unpredictable. Something that is ordered has low entropy. In cryptography, entropy is usually measured in bits. A truly random one bit number has an entropy of one bit. Entropy is directly connected to the information content of a message. For

[8] http://www.demailly.com/~dl/netscapesec/wsj.txt

[9] Digital Signature Standard (DSS)", Federal Information Processing Standard Publication 186, May 19, 1994.

[10] John von Neumann

[11] There are many different definitions used in the literature, but they are all equivalent to this definition.

example, a sequence of *n* random bits (i.e., a sequence having an entropy of *n* bits) cannot on average be expressed or defined in less than *n* bits.

The problem with the theoretical definition of randomness is that the amount of entropy cannot be measured perfectly. It can be easy to demonstrate that a particular generator is *not* random based on its design (for example, its design causes the same sequence of generated numbers to recur ever so often). Unfortunately, demonstrating randomness is much more difficult. Indeed, it is impossible to *prove* that the output from a given random number generator is truly random. As a result, the best we can do is to analyze the statistical characteristics of the previous generated bits and compare them to a mathematically perfect generator.

For security applications. the most important characteristics of random numbers are (paraphrasing Schneier[12]):

Unbiased Statistical Distribution. Truly random numbers have certain statistical characteristics. These characteristics can be calculated for a theoretical random generator and compared to the statistics for a generated sample. Based on this comparison, the probability that the sample came from a biased source can be calculated.

For example, one important statistical characteristic is the number of 0 or 1 bits in the sample. For a sample of 20,000 bits from a truly random generator, the probability is 0.0001 that the number of 1 bits is less than 9,725 or more than 10,725. Thus, if we analyze 10,000 samples of 20,000 bits each, and find many samples that deviate from these limits, we should suspect that our generator is not truly random (we should expect only one deviation out of 10,000 tests from a truly random generator).

Of course, this test does not prove randomness. For example, if our generator is stuck at generating alternate 1s and 0s, it would pass this monobit test even though the output is clearly non-random. Thus many different statistical characteristics must be analyzed in order to feel comfortable about a generator. Our alternating 1s and 0s sequence would quickly fail many other tests such as a bit-to-bit correlation test, or a multi-bit block frequency test.

Security applications minimally require good statistical characteristics of the generated random numbers. A generator that meets only this non-biased requirement is called *pseudo-random*.

Unpredictability. In addition to good statistical properties, truly random numbers must be unpredictable; that is, the probability of correctly guessing the next bit of a sequence of bits should be exactly ½ regardless of the values of the previous bits. Some applications do not care about this unpredictability characteristic, but it is critical to security applications.

An example of a predictable sequence that has good statistical characteristics would be a generator that produces, for example, 20,000 statistically good random bits and then repeats the exact sequence again. To identify this problem would require analyzing the samples larger than 20,000 bits. A worse problem with predictability is the use of typical software generators. For a software generator to be unpredictable requires that the software algorithm or its initial values be *hidden*. (If we know the algorithm and initial values, we can produce the same sequence as the software generator.) From a security viewpoint, a *hidden algorithm* or *magic number* approach is very weak if it is possible for someone to predict the output. For example, the previous section mentioned a well-known case where the security of a predictable (software) random number generator failed because the starting values could be guessed easily.

A generator which deterministically derives its output from a starting value, but which meets both the statistical and unpredictability requirement is called *cryptographically secure pseudo-random*. (To be precise, these RNGs meet the unpredictability requirement by making it *computationally infeasible* to predict

[12]Schneier (1996) p.45-46

output bits. An adversary with unlimited computational power could guess possible values for the starting state until one is found that matches the output.)

While some statistical tests may indicate a degree of predictability, whether a generator is considered unpredictable is primarily based on it's inherent design.

Unreproducibility. To meet this requirement, two of the same generators (or the same generator at two different times), given the same starting conditions, must produce different outputs. Clearly, software algorithms do not meet this requirement; only a hardware generator based on random physical processes can generate values that meet this stringent security requirement. There no test to determine unreproducibility: this determination is based on the inherent design of the generator.

A generator that meets all three requirements is called *really,* or *truly, random*.

2.3 HOW TO GENERATE RANDOM NUMBERS

We now know what random numbers are and why they are critical to many applications, especially security. Why do we need anything special, however, to generate them— Isn't there a random number generator included in every programming library? Well, there is, but as Schneier points out: "these [software] random number generators are almost definitely not secure enough for cryptography, and probably not even very random. Most of them are embarrassingly bad.[13] This section explores this problem and describes the three basic approaches to generating random numbers: software, physical sources, and quantum randomness, and some combinations of these techniques.

Software Generators

The most common approach to generating random numbers is using a deterministic algorithm implemented by a computer program. Such deterministic algorithms cannot generate truly random numbers. (At best they are predictable and reproducible, and at worst, have bad statistical characteristics.) The quality of software based random number generators is often summarized with John Von Neumann's famous quote: "Anyone who considers arithmetic methods of producing random digits is, of course, in a state of sin."[14] Thus, software generators are usually called *pseudo-random* or *quasi-random* generators. There is, however, a wide range of quality among software generators with some being considerably better for security purposes than others.

Many different software algorithms are used, but all comprise two basic steps for each value generated:

- **Choose a starting value or *seed.*** All software generators start with some initial value, or seed. Some set of arithmetic operations (the mixing algorithm) are performed on this initial seed to produce the first random result. In most cases, this result is then used as the seed to produce the second result, and so forth. In some cases, a new seed is constructed for each value based on external (to the random number generator) events (for example, the time-of-day clock).

- ***Mix* the bits from the seed in an attempt to increase its *apparent* entropy.** The mixing algorithms can be grouped into two distinct groups. Conventional generators such as typically provided for the C language `rand()` function perform simple arithmetic operations. Newer algorithms designed specifically for cryptography perform much more complicated (and time-consuming) operations. Each algorithm has

[13] Schneier (1996) p44

[14] John Von Neumann (1951). Quoted in Knuth, 1968, Vol. 2

different performance characteristics (bit-generation rate), and different randomness characteristics. Although the mixing operation does not technically increase the total entropy (deterministic process cannot create entropy), it is believed that properly-designed mixing operations can make it *computationally infeasible* to distinguish the output from a true random source.

-

Let's look at the conventional approach, which represents most extent generators. These algorithms take the previous value generated and they apply some arithmetic operation to it to produce the next value. Historically, little theory was applied to the mixing algorithm: if it seemed to mix up the seed bits a lot, it was considered a good algorithm. As an example, Knuth references an *extremely* complicated algorithm he invented in 1959 that, when executed, quickly converged to a constant value. This led him to conclude: "random numbers should not [be] generated with a method chosen at random. Some theory should be used."[15] Subsequently, a significant amount of theory has been applied to defining and analyzing good mixing algorithms. The result is a set of *linear* mixing algorithms that are relatively fast and have reasonably statistical characteristics.

A basic example of a linear algorithm is widely used in modern software generators. It is the *linear congruential method*. The calculation is simple:

$$

where the constants a, c, and m meet some very specific mathematical constraints and the calculations are performed in the word size of the processor. If the constants are chosen correctly, then the cycle length of the generator is m. That is, after m numbers, the generated sequence repeats itself. This is, for example, the approach used in the Microsoft Visual C++ 5.0 library implementation of `rand()` where $m = 2^{31}$ (0x80000000).

There are several other linear algorithms that have good theoretical underpinnings. These theoretical characteristics are not intuitively obvious, however. Inventing your own algorithm or making minor changes to a known good algorithm can yield very bad randomness.

Although the cycle length of the Microsoft `rand()` example is too short for Monte Carlo simulations and other applications requiring large amounts of random, linear algorithms *can* have a reasonably long cycle. Similarly, well-designed linear algorithms *can* generate numbers with reasonable statistical characteristics. In general, however, the most common software algorithms do not have statistical characteristics as good as truly random values.

Even worse, these algorithms are easily predictable; that is, knowing the algorithm and the constant values (which are often fixed in the software), one can predict the entire sequence. Even if the seed value isn't constant, it can be guessed or, knowing the last generated value (or any other previously-generated value), one can trivially predict the sequence. These generators are also reversible: given the algorithm and a value, previous values can be easily calculated. These cyclical and predictable characteristics are a disaster for security applications.

The good news is that extensive research, publication, and analysis has been done for years on software random number algorithms for use in security applications. One result is the design of some improved non-linear software generators that closely match the statistical characteristics of a truly random generator and are somewhat unpredictable. These are called *cryptographically secure pseudorandom bit* generators (CSPRBG). They use large internal calculation sizes (typically 64- to 1024-bits) and use cryptographically strong *one-way* algorithms. They use values that are determined dynamically rather than built into the code.

[15] Knuth, pg 6

Strong one-way functions make it is impracticable to calculate subsequent or previous values given the current value unless the generator's state is known. Even if an adversary discovers the state at a particular point, the algorithms are not reversible, making it infeasible to determine the number prior to the compromise. Of course, knowing the state of the RNG does allow future results to be calculated. As a result, these algorithms rely on secrecy of the initial conditions and computational intermediates to ensure unpredictability.

A simple example of such a CSPRBG is the Blum-Blum-Shub generator.[16] Here is an abstract:

- Start by selecting two large primes, p and q, with certain characteristics and calculate the hidden value $n = pq$. Choose some seed, x_0, with certain characteristics relative to n.
- Calculate each new random value by

$$z_{i+1} = \textit{the least significant bit of } x_{i+1}$$

  Even if n is known, its particular numeric characteristics make the difficulty of calculating the previous value very high.

As good as these CSPRBGs are compared to conventional linear software algorithms, however, they still have some problems:

- They are not widely available. They are also prone to typical software mistakes in the implementation.
- They are often computationally very slow.
- They are predictable in the sense described above (i.e., forward prediction if the initial conditions are known).
- Like all software algorithms, they are reproducible: they produce the same results starting with the same input

Their security relies on an assumption that adversaries do not have enough computational power to perform certain mathematical operations. As a result, their irreversibility cannot be proved, although, in practice, this is not a major problem.

There is a way to improve the situation here. Rather than relying solely on a software generator to produce all of the randomness, a software generator can be used in conjunction with a source of true randomness (entropy). Instead of using the previous result as the seed for the next number to be calculated, output from the independent source of entropy can be used as the seed. Even if the entropy source is imperfect (provides less than one bit of entropy per output bit), the results in this case can be more robust than what could be generated by either the software or the entropy source alone.

An example is using a hardware random number generator (see the next section) to produce the seed for a software algorithm. Although a variety of designs are used, the software algorithms are often based on cryptographic *hash functions.* A hash algorithm mixes up its input bits in a fashion such that its output bits should be unpredictable unless the entire input is known. Thus, a good hash algorithm effectively destroys any statistical correlations among the input bits and makes it computationally infeasible to recover the input from the output. The most commonly used cryptographic hash algorithm, the *Secure Hash Algorithm (SHA-1)*[17] is a U.S. government standard and has been extensively analyzed.

The primary reasons for using a hybrid approach (as opposed to a pure hardware-based or pure entropy based approach) are (1) the historically slow speed of entropy and hardware generators, and (2) the lack of perfect randomness in the hardware or entropy generators. While the VIA C3 Nehemiah processor RNG provides

[16] (In choosing an example, I couldn't resist this name.) See Menezes et. al, pg. 186

[17] FIPS 180-1, Secure Hash Standards, U.S. Department of Commerce/NIST, 1995

both high performance and high quality output, users may choose to use it in conjunction with software based algorithms.

Physical Sources

In terms of randomness characteristics, *entropy generators* based on physical phenomena fall in between software generators and quantum based hardware generators. The idea here is to sample some *random(?)* physical process and assume the samples are truly random. Unfortunately, computers and software tend to be very predictable, so designers must be very careful to ensure that they collect good entropy.

For example, a widely used PC encryption program derives its keys from several seconds of mouse movements and keystroke timing (directed by the program). The Linux operating system has random number generators (/dev/random and /dev/urandom) that use entropy generated by the keyboard, mouse, interrupts, and disk drive behavior as the seed. Microsoft's CryptGenRandom function (part of the Microsoft CryptoAPI) is similar. It uses, for instance, mouse or keyboard timing input, that are then added to both the stored seed and various system data and user data such as the process ID and thread ID, the system clock, the system time, the system counter, memory status, free disk clusters, the hashed user environment block, as the seed. Many other similar environmental sources of randomness have been tried.

While these physical activities may look random, their randomness cannot be proven, and they run the risk of generating poor entropy (or no entropy) if the sampled physical activity is dormant or repetitive. There are several potential security vulnerabilities when using such physical activities. For example, in networked applications such as browsers, the application traffic between a client and server effectively publishes the locations and sequence of the client's mouse-events. Similarly, users may enable "snap-to" options that center the mouse pointer in the center of the button to be pressed and make the click locations predictable. As a result, the entropy from mouse movements in these environments could be far less than an RNG designer expected. Similarly, asking the user to create entropy using the keyboard creates bias since humans tend to follow certain patterns in typing (such as a tendency to use the center of the keyboard).

In summary, it is dangerous to use the entropy values *directly* as random numbers. Instead, the entropy values are usually used as the seed to a good hash algorithm to produce the final random numbers. (In the Microsoft CryptoAPI and Linux examples, a SHA-1 algorithm is applied to the entropy seeds). Other common problems with entropy generators on computers are that they require hooks in the operating system, they are difficult to test, they often require some user involvement, and they are slow (since they are based on macro physical events).

An unusual example of a physically-based random source is available through www.random.org, which continually provides generated random numbers using atmospheric noise (sampled as radio frequency noise). The rate varies, but this generator typically produces about 40,000 bits per second.

True Hardware Generators

By now you should be convinced that good security requires good random numbers, and while it is possible to generate pretty good statistically random numbers using software, it is difficult, slow, and subject to numerous security pitfalls. The only truly random generator is some mechanism that detects quantum behavior at the sub-atomic level. This is because randomness is inherent in the behavior of sub-atomic particles (remember quantum mechanics, Heisenberg's Uncertainty Theorem, etc.).

A quantum based hardware generator is actually practical. Examples that have been used are:

The interval between the emission of particles during radioactive decays. This technique is used at a web site to generate random bits that are available on-line from the decay of Krypton-85.[18] This source generates only 30 bytes per second and requires a cumbersome (and dangerous?) pile of hardware.

The thermal noise across a semiconductor diode or resistor. This is the approach most often used in add on PC hardware.

The frequency instabilities of multiple free running oscillators. This approach is the basis of the VIA C3 Nehemiah RNG approach. While implemented differently than the resistor based approach, ultimately, the source of randomness is the same.

The charge developed on a capacitor during a particular time period.

These sources have been used in a few commercially available add on random number generator devices, none of which have achieved much visibility or use. Since they are peripheral devices such as PCI cards and serial port devices, these commercial hardware generators are expensive and cumbersome. In addition, they are all made by universities or small companies with limited marketing and distribution capability. Some examples of such add-on hardware generators are found in these references.[19]

There is one notable exception to the add on strategy. Intel provided a hardware random number generator in the 82802 Firmware Hub.[20]

2.4 HOW TO VERIFY RANDOMNESS

Given the differences in generation algorithms and the inherent theoretical problems with software generators, the obvious question is: how do we compare or quantify random number generator *goodness*? For any serious application or generator design, it is essential to have tests that *quantify* randomness.

In one sense, the definition of a random number is simple: the probability that each bit generated is a one (or zero) should be ½. Unfortunately, that definition doesn't help us directly since we can't actually test the *probability* for the next bit to be generated. All we can do is to study bits that are actually generated and analyze their characteristics to see how likely it is that the generator is random based on what it has generated. This approach turns out to be complicated, and in fact, there is no single or simple set of tests that can accurately identify randomness.

The underlying difficulty here is that truly random numbers occur randomly. That is, every possible value can be generated by a truly random source. If we sample 64 bits, for example, from a truly random source, there is a tiny but non zero probability that all 64 bits will be zero. In that case, should we reject the source as not random? A potential answer is to collect many 64 bit samples and check to see how many times all zeros appears. Statistically, we should expect one all zero set per 2^{64} samples. It is usually impractical to collect such a comprehensive set of samples, however. Indeed, because all outputs should be equally unlikely, no particular output provides *proof* of non-randomness. As a result, even the appearance of multiple all zero sets would not prove that the source is nonrandom. Conversely, what if the sample values are 0, 1, 2, 3, 4, etc.? The values

[18] http://www.fourmilab.ch/hotbits/

[19] http://faust.irb.hr/~stipy/random/hg324.html
http://comscire.com/index.htm
http://mywebpages.comcast.net/orb/
http://www.protego.se/sg100_en.htm

[20] Benjamin Jun et al., *The Intel Random Number Generator,* Cryptography Research, Inc., 1999, available from http://www.cryptography.com/resources/whitepapers/IntelRNG.pdf

and bits seem to be appearing in the right statistical frequency, but do we really believe that the source is random?

Solutions to this fundamental problem are complex and rely on some understanding of statistical concepts. This testing topic is very important, however, and requires some attention in order to determine the proper use of any RNG, including the VIA C3 Nehemiah processor RNG.

CHAPTER

3

TESTING RANDOMNESS

This chapter summarizes many methods used for quantifying randomness. A summary of the testing performed by Centaur is provided. Users, of course, may want to perform their own evaluations.

3.1 BACKGROUND

There is no simple way to answer to the question: is a particular generator random? The only answers are statistical in nature and are only meaningful for relatively large sample sizes. We shall explore testing for randomness in some detail in this chapter, including a few equations. The intent is to provide adequate background to understand out randomness testing results In particular, this background section introduces technical concepts (*P-value* and α) that are extensively used in subsequent sections in this chapter.

While all randomness testing is statistical in nature, there are several different technical approaches. We primarily follow the approach proposed by the U.S government's National Bureau of Standards (NIST).[21] The essence of this approach is as follows:[22]

Identify *multiple* statistical characteristics of truly random numbers for which to test. No single statistical characteristic is adequate (for example, a sequence of 1, 2, 3, etc. satisfies a numeric distribution test, but obviously will fail a basic bit-to-bit correlation test). The basic theory here is that if something has tail feathers like a duck, walks like a duck, and quacks like a duck, then we may substitute it for a real duck. Unfortunately, there are many possible characteristics of a duck, and many possible statistical characteristics

[21] Juan Soto, *Statistical testing of Random Number Generators,* NIST: 1999. From Proceedings of the 22nd National Information Systems Security Conference, 10/99. http://csrc.nist.gov/rng/nissc-paper.pdf.

[22] Andrew Rukhin, et al*., A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications,* NIST Special Publication 800-22, 2001. http://csrc.nist.gov/rng/SP800-22b.pdf. This is a good description of the theory and practice behind random number testing, albeit quite technical.

of random numbers. A particular generator may have a particular flaw that shows up on some characteristics, but looks random on others. Following the analogy, it may look just like a duck in all characteristics except for the webbing on its feet. For some applications (swimming), this difference may be catastrophic.

Fortunately, a significant amount of research has been performed and documented relative to the desirable random number statistics. There are several sets of tests that been shown to be reasonably effective at identifying flaws in sources of randomness.

For each statistical test identified, perform statistical analysis to identify the probability that a truly random generator could have produced a sequence less random than the sample tested (for that particular test characteristic). This probability is called the P-value. Another way of stating this is that the P-value is the probability of obtaining a test statistic as large or larger than the one observed *if* the sequence is truly random.

That is, the higher the P-value for a random number sequence, the more the sequence looks like it came from a random generator (for the particular test statistic). Hence, small P-values are interpreted as evidence that the sample is unlikely to come from a random source. For example, if the P-value for a particular sample is 0.3, it means that it is 30 % likely that this sample came from a random generator (relative to the particular test characteristic).

This statistical analysis and resulting P-value calculation require serious mathematics, but the approach is well documented in the literature.

Identify the acceptable (to the application) probability that a test might reject a sample from a truly random generator. This acceptable probability is called the *level of significance*, or α. Typical values of α used in cryptography applications are from 0.01 to 0.001. An α of 0.005 means that one would accept that, on average, one sample out of every 200 samples from a truly random generator would be rejected by a particular test. The same α is generally used for all tests. Larger values for α increase the rate at which good RNGs will get rejected, while smaller values for α increase the likelihood that a bad RNG may pass the tests.

The connection between of the calculated P-value and the target α is as follows: if the P-value $\geq \alpha$, then the sequence should be considered random (by the particular application for the particular test). A P-value $< \alpha$ would be considered a failed sample.

Perform all the tests on lots of *samples* (a contiguous sequence of generated bits). The result from a single sample (especially for the normal case of P-value $\geq \alpha$,) does not adequately identify whether a generator is truly random or not. The number of samples required depends on the test's requirements, but a careful analysis will typically include at least $1/\alpha$ number of samples, and often many more.

For the number of samples tested, calculate the statistical significance of the set of P-values across all samples and the failure rate. This analysis of the many samples finally gives us an overall confidence factor about the randomness of the generator.

An detailed example should make this process more clear. Let us consider the simplest test for randomness: the number of 1 and 0 bits in the sampled sequence. Let n be the number of sampled bits ($n > 1000$) and S be the absolute value of the difference between the number of 1 bits and the number of 0 bits. Calculate (we won't derive the mathematics here)

$$s = \frac{S}{\sqrt{n}}, \text{ and}$$

$$P - value = erfc\left(\frac{s}{\sqrt{2}}\right) \text{ where } erfc(z) = 1 - \frac{2}{\sqrt{\pi}}\int_0^z e^{-u^2} du$$

(The $erfc(z)$ functions is a commonly used statistical function called the *complementary error function.)* Solving these equations for a P-value = 0.0001, we find that the number of one bits (or zero bits), x, in a

20,000 bit sample must be between 9,725 and 10,275 (exclusive) in order for the resulting P-value to be ≥ 0.0001. That is, there is only a 0.0001 probability that a truly random sample of 20,000 bits will not have a value x such that $9{,}725 < x < 10{,}275$. That is, if our sample falls outside of these bit extremes, we could consider the sample not random with a 99.99 % confidence level.

Why did we pick a target of 0.0001? This was merely our choice as to how stringent our tests for random numbers should be. (Actually, this happens to be a specific example from an important government standard, FIPS 140-2, discussed below).

This was merely one sample, however. Two important facts about random number testing that are inherent in the statistical nature of the tests are:

If a particular sample (or set of samples) passes all statistical tests, it may still come from a non-random generator.

If a particular sample (or set of samples) fails some statistical test, it may still come from a random generator, although for the likelihood of a false-negative may be vanishingly small).

Thus, to properly evaluate a generator, many different samples must be analyzed. By measuring more samples and by running more tests we can increase our confidence in the test results. In the example above, we should probably run at least 1/0.0001 = 10,000 tests, each of 20,000 bits, and expect to see around one failure from a truly random generator. There are several statistical approaches for analyzing the results from many samples in order to give a probability of generator randomness. For example, first consider the passing fraction (those with P-value ≥ α,) for some test from m samples. One *confidence interval* that we use is:

$$q \pm 3\sqrt{\frac{q(1-q)}{m}} \quad \text{(where } q = 1-\alpha \text{)}$$

If the passing fraction falls outside this confidence interval, there is evidence that the generator *may* not be truly random.

As an example, let us assume that we test 10,000 samples from the above example and one test failed (had a P-value < α). Our passing fraction is 0.9999. In our example, the confidence level (for α = 0.0001) is calculated as 0.9996 - 1.0002. Since the passing fraction is within the confidence interval, the samples should be considered random (relative to this one test). If we had only had 5,000 samples with three failures, our passing fraction would be 0.9994 and the confidence interval would be 0.999476 - 1.00032. In this case, we would be outside the limits and probably should conclude that the samples (or the generator) may not be truly random.

This particular approach to analyzing the failure rate of multiple samples is only one example of many that can be used. Another important approach is to look at the P-values themselves. Given a large number of samples, the theory is that the P-values should have a uniform distribution between zero and one. The actual distribution can be analyzed using a chi-square function to indicate how close the distribution comes to a theoretical distribution. A common test used to so analyse the P-values is the Kolmogorov-Smirnov test (*KS test*)[23]

3.2 IMPORTANT TESTS

There are *many* aspects of randomness that can be tested. Accordingly, there are many tests defined and in use somewhere. A few of these are robustly defined and have achieved acceptance by the technical community. Some of the most important tests are:

[23]This test is described by the NIST at http://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm

NIST Test Suite. The Computer Security Resource Center (CSRC) is a part of the US government's National Bureau of Standards (NIST) organization. The CSRC has defined 16 statistical tests for verifying randomness.[24] Each test addresses a different potential type of flaw in the generator. In addition to the definitions, NIST provides a suite of programs that can be used for implementing these tests.

FIPS-140. This is a government cryptography standard, which includes a four-test subset of the full NIST suite with a specific numeric range of acceptability for each test.

The DIEHARD Tests. These are a suite of 18 statistical tests defined by George Marsaglia of the University of Florida.[25] These are widely used in academia and referenced in many papers on RNGs, but they are not as well documented or supported as the NIST tests. There is a great deal of overlap between the NIST tests and the DIEHARD tests, but each contains some unique tests.

Knuth's Tests.[26] The tests described in Knuth's classic reference are primarily interesting from a historical or theoretical perspective since the tests described are basically covered by modern suites like the NIST. The reference is a good source for understanding the theory behind random number testing.

Many other tests used. Almost everyone who dabbles in random number generation has their own favorites, and, in fact, usually makes up their own tests.

When testing an RNG, it may also be appropriate to construct additional tests based on the design of a particular RNG. For example, if an RNG was constructed by flipping ten coins in order then repeating such that every tenth bit was produced using the same coin, it would be appropriate to apply tests that look at every tenth bit.

3.3 OUR METHODOLOGY

The statistical test suites described in the previous section were developed to test software random number generators. These generators are necessarily deterministic, and provide only as much entropy as found in the initial seed - *regardless of the number of "random bits" generated*. While Centaur Technology has made extensive use of these tests, we have found that simpler tests that check for a biased statistical distribution are more appropriate and more effective for our hardware generator.

During the initial development of the RNG, while we were getting the bugs out and trying to understand the characteristics of the part, we tested the device with the NIST suite, the DIEHARD suite, the FIPS-140 tests (and many others). These tests proved valuable in helping us to understand some of our earlier failures, and in comparing the C3 Nehemiah processor RNG with other hardware random number devices and software algorithms.

However, once we had stable hardware, we determined that a simple chi-square test of the distribution of the 256 possible byte values was more sensitive to deviations from "randomness" - *for our hardware RNG* - than any other test. That is to say that the NIST tests, and the DIEHARD tests, and the FIPS-140 tests were frequently *passing* (the P-values were within the confidence levels at the expected frequency) when the chi-square test was *screaming* failure.

We continue to use the above test suites (particularly the DIEHARD and the FIPS-140 tests) on a regular basis and have incorporated them into our comprehensive test harness. But they are no longer part of the routine testing.

---

[24] Ibid.

[25] http://faust.irb.hr/~stipy/random/manual/html/Diehard.html

[26] Donald E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms,* Vol2, Chapter 5, Addison Wesley, 1998

Again, remember that many of the tests in the above test suites are designed to catch flaws in deterministic random generators that were already known to have good statistical distributions - to find those generators that deliver a sequence like 0, 1, 2, 3, ... repeatedly, which will pass the statistical distribution tests (like the chi-square test).

As noted in the introduction, the raw bits delivered by a VIA C3 Nehemiah processor RNG display a small bit-to-bit correlation. The (software selectable) hardware one-of-N filter reduces this correlation to small levels, such that statistical tests performed on samples of (say) 10MB of random bits will pass all of the above statistical tests, allowing for the fact that - as Marsaglia says - "P happens". A failure is to be expected, occasionally, if you perform enough tests.

However, even with the hardware one-of-N filter set at the maximum value, the measured entropy of the bits from the VIA C3 Nehemiah RNG is still slightly less than 1. That means that a suitably designed test, on a *sufficently large* sample of random bits, will *consistently* fail.

Once our standard methodology was in place, *extensive* testing of production steppings of the VIA C3 Nehemiah processor RNG feature was performed. Centaur Technology has sampled and tested in excess of 10TB (yes, that is almost 10^{14} bits) of data produced by the VIA C3 Nehemiah processor RNG. The tests were (*and continue to be*) run on many different chips running on several different PC platforms running several different operating systems, but most tests have been performed using Red Hat Linux 7.1 (kernel version 2.4.2-2)

Many thousands of samples, usually of 10 MB each, were taken from each chip/system, using all eight DC bias settings. The samples were taken with the hardware one-of-N filter set to the maximum value, to obtain the best possible results. When the C5P processor stepping became available, we performed the tests on each noise source separately and with both noise sources active.

Many thousands of samples, also of 10MB each, were taken with the hardware one-of-N filter off, and with the whitener disabled. While the statistical properties of these samples of raw bits was poor, that is to say they consistently failed the tests, these samples allowed us to measure the entropy of the raw bits produced by the RNG.

A smaller set of samples was taken at intermediate settings, with the whitener enabled but with the one-of-N filter set to less than the maximum value. Most of these samples were taken early in our development process, which enabled us to determine the appropriate maximum value for the one-of-N filter.

For <u>every</u> sample, the standard chi-square test on the byte distribution was performed. The cumulative distribution of the delivered bits was also subjected to the chi-square test, and a KS test on the distribution of the chi-square values for the individual samples was performed. We also measured the effective generated bit rate. And on some samples we also performed FIPS-140, DIEHARD, NIST, or other statistical tests.

As noted above, the entropy of our RNG is not a perfect 1.00 bits of entropy per bit, but somewhat less. Thus, we have seen that for *very large samples* of bits, the chi-square test for the entire sample, or the KS test of the distribution of the P values from the chi-square tests of the individual 10MB samples (or both), will eventually fail - *even though each 10MB sample passes any and all statistical tests at an acceptable rate*.

Now, *when* the cumulative distribution chi-square test or the KS test will fail varies considerably with the DC bias setting, from part to part, and slightly on other environmental conditions such as temperature or what the author had for breakfast that morning.

Typically, we set an arbitrary limit on the P-value for the cumulative chi-square statistic, and on the KS statistic. Then, when either test exceeded that limit (sometimes we required that both tests have P values outside our confidence level), testing at that bias setting was terminated, a different bias value was set, and the

tests restarted. Of course, the reports of the test harness were saved to disk. Generally, however, the random bits themselves were (are) not saved, except when we intend to (later) evaluate them with the NIST or DIEHARD test suites.

(Hey, do *you* have 10TB of disk drives lying around? We can fill a 30 GB drive with raw bits in a little more than 30 minutes - if the drive controller can keep up with us.)

This procedure would repeat, continuously, for a week or two, when a different chip would be placed in the system and the entire process began anew. This can be thought of as trying to estimate a "Mean Time Before (statistical) Failure" for the RNG. Evaluation of these data allowed us to select, as the default DC bias setting for the production stepping of the VIA C3 Nehemiah processor, that bias setting that gave the "best" results, on average.

Most tests were run at ambient conditions. A few were run and analyzed at worst case temperature conditions and at intermediate temperatures. The bottom line is that temperature is just one of the variables that contribute to the entropy of the VIA C3 Nehemiah processor RNG. Furthermore, the effects of changes in temperature on the bit generation rate varies with the DC bias setting.

As an interesting comparison, and to validate our test harness, the same statistical tests were also performed on random numbers from other sources. One source is the standard Linux software random number generator **/dev/urandom** running on the same PC platforms. This generator gathers environmental noise from device drivers and other sources into an entropy pool. From this entropy pool random numbers are created using a SHA-1 hash.

A second alternate source is random data provided by www.random.org. This data is derived on atmospheric noise sampled as electrical noise (static) on a radio receiver.

3.4 RESULTS AND USAGE RECOMMENDATIONS

An independent evaluation of the VIA C3 Nehemiah Random Number Generator has been performed by Cryptography Research, Inc., of San Francisco. The attentive reader will recall that these are the same security consultants who evaluated the Intel 82802 Firmware Hub.

A copy of the report is available at: www.cryptography.com/resources/whitepapers/VIA_rng.pdf

From their closing commentary:

> *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.*

We suspect that most applications, those that have a need for relatively small samples of random numbers, will want the best randomness available with the least effort. Since the hardware speed is very fast even with the one-of-N filter set to the maximum value, these applications should so set the filter and directly use the numbers returned from the RNG. This option provides very good randomness with no additional software effort at, typically, a fast 800K bits/second (or 1600K bits/second if both noise sources are selected).

So long as your sample size is on the order of 10MB or smaller, the VIA C3 Nehemiah processor RNG will deliver very good randomness.

For samples on the order of 1GB or larger, you can expect that the slight bit-to-bit correlation will exhibit itself in statistical test failures. For intermediate sample sizes, on the order of 100MB, we recommend that you run your own tests to determine whether the unmixed output of the RNG is suitable for your needs.

> **NOTE**: we have *no idea* how this compares with other hardware generators. The VIA C3 Nehemiah processor RNG is so much faster than other generators that we haven't been able to obtain enough large samples to test. **We do not consider pre-calculated samples as acceptable for testing**.

For those applications that require a faster bit generation rate, or require *very large* samples of random numbers, or that have cryptographic requirements for the highest possible entropy, Centaur Technology recommends turning the whitener and the one-of-N filter *off*, turning both noise sources *on*, and mixing the raw bits returned from the C5P RNG (at up to 100M bits/second) with the Secure Hash Algorithm (SHA-1). If you have need for the fastest possible bit generation rate, it will be worthwhile experimenting with the DC bias setting. We have observed bit rates near 120M bits/second on some chips at non-default bias settings.

The SHA-1 always outputs 20 bytes. Depending on your security requirements and your estimate of the entropy of the raws bits from the RNG, you will probably want to mix more than 20 bytes of raw RNG output with the SHA-1 algorithm.

In the above mentioned report from Cryptography Research (if you have *any* security or cryptographic requirements for an RNG, that report is recommended reading), they measured the per-bit entropy of the raw bits from the VIA C3 Nehemiah processor RNG, and found that the value lies between 0.78-0.99 bits of entropy per raw bit. These numbers correspond closely with entropy measures taken during our internal testing. We generally measured numbers at the higher end of that range; Cryptography Research was creative in setting up a test environment that produced the lower measured entropy. (Again: go read that report. I'll wait.)

Thus, depending on the paranoia required of your application, you may wish to mix 20, 24, 28, or even 32 bytes (or more!) of raw bits to generate 20 random bytes.

During our testing we frequently mixed raw bits with SHA-1, mixing from 20 to 30 bytes of raw bits to produce 20 bytes worth of random bits. Because of the asynchronous nature of the VIA C3 Nehemiah processor RNG, the delivered bit rate is only slightly sensitive to the number of input bytes when mixed with the SHA-1, and we have measured sustained data rates of over 50M bits/second on a 1.2 Ghz C5P with this technique. *The bits mixed with SHA-1 are indistinguishable from random on all tests, including our chi-square and KS tests on samples as large as a terabyte. We are currently collecting larger samples.*

Provided that the above entropy measures of the VIA C3 Nehemiah processor RNG are accurate, and that there are no cryptographic flaws involved in mixing bits with the SHA-1, these random bits should be considered cryptographically secure, or truly random.

As delivered.

In addition to the Linux **/dev/urandom** device discussed above, there is also a **/dev/random** generator on Linux platforms, that functions the same as **/dev/urandom**, except that the estimated entropy of the output random numbers does not exceed the estimated entropy of the underlying entropy pool. Thus these bits are considered cryptographically secure. The bit rate of **/dev/random** on a C5P with the 2.4.2-2 kernels is on the order of a few 100 bits/second.

It is Centaur Technology's understanding that newer Linux kernels (version 2.6 and later), and OpenBSD (version 3.3 and later), have incorporated the VIA C3 Nehemiah processor RNG into their entropy pools, when they are running on our processor. For customers with these operating system kernels, the advantages of the VIA C3 Nehemiah processor RNG are already part of your system.

Centaur may be able to provide advice or assistance to customers with unusual requirements based on our experience. Please refer to our RNG Programmers Guide for usage details and contact information.

CHAPTER

4

ACE VALIDATION

Validating a cryptographic algorithm is - compared with testing a random number generator - almost trivially easy. After all, it's just an algorithm, with a large number of publicly available data sets with known input and result values. And unlike random tests, where a small number of test failures is to be expected, the cryptography engine is either correct or it is not. Any test failure is one too many.

Centaur Technology has developed a comprehensive test package incorporating many of the available data sets, involving both encryption and decryption, at all three key sizes, in all four supported operating modes, with varying numbers of input blocks, using both methods of key generation, and with special tests for every corner case we have been able to think up. We have tested a number of C5P chips in a variety of multi-tasking environments, with as many as 10 tasks concurrently running version of our ACE test package.

No ACE failures have been observed in production steppings of the VIA C5P.

But don't take our word for it. Cryptography Research, Inc., has recently completed a (brief) security evaluation of the Advanced Cryptography Engine. A copy of the report is available at:

www.somesite.org

From their summary:

"Marketing quote from CRI"

Customers interested in testing the ACE for themselves should obtain a copy of the VIA C5P Advanced Cryptography Engine Programmers Guide for programming details. They may also request (please refer to the Programmers Guide for contact information) a copy of our test package.

Centaur Technology has learned that OpenBSD (version 3.4) has incorporated the ACE in it's kernel cryptographic services.

CHAPTER

5

CENTAUR SECURITY ROADMAP

In addition to our presence on the desktop, VIA processors are used in a growing number of embedded products. In particular, within the networking infrastructure, where many products aggregate connections to support SSH, SSL, IPSEC, etc., there is a current need for *fast* cryptographic processing.

VIA has built, and plans to continue building, devices that meet these existing pent-up needs. Our Random Number Generator, and the AES encryption/decryption primitives of the Advanced Cryptography Engine, can be used *now*. Not next month, not next year, and not before some new protocol and infrastructure has been introduced and deployed, but immediately.

And they can be used directly by your applications; they do not require support from your favorite operating system.

5.1 WHAT'S NEXT?

What's next for VIA processors with respect to our security initiative?

SHA-1

The new VIA Esther processor, successor to the Nehemiah (also called the C5I at Centaur), will incorporate all the security features of the C5P and will add one new security feature: a hardware implementation of the Secure Hash Algorithm (SHA-1)[27] In addition to other, more mundane (to security geeks) features and improvements in the processor. Like running at 2 Ghz.

[27]Defined in FIPS180-2

The SHA-1 logic is already working correctly in simulation, and has been integrated with the Random Number Generator. The one-of-N filter will be removed; it's functionality will be replaced by using the internal SHA-1 logic to mix the raw bits, such that the higher settings of the filter will require more raw bits input into the hash function than the lower settings.

Using the SHA-1 as a mixing function, in the hardware, will make the improved statistical properties of combining a fast entropy source with a good cryptographic hash algorithm, and the improved security of delivering only as many random bits as the entropy pool contains, available to your applications regardless of your operating system support.

Nevertheless, the raw (unwhitened, unfiltered) bits will still be available to customers who want them; device drivers incorporating such raw bits in their entropy pools will find that the C5I processor RNG will function identically to the C5P.

You may ask - why is this so wonderful? I can get 50M bits per second of truly random bits from a C5P today. Yes - but your processor will be running flat-out doing all that SHA-1 mixing. If you can get the timing right (and ask for random bits just as they become available from the noise sources), you will be able to achieve a 50M bits per second (or higher) generation rate of truly random bits using about 2% of processor cycles.

The software API will remain unchanged with a single exception, having to do with memory alignment but only when the SHA-1 mixer is selected. Interestingly, using the hardware SHA-1 is slightly faster than the current hardware one-of-N filter. Thus the overhead (in terms of CPU cycles) of using a VIA processor RNG will be reduced.

Of course, there are other uses for a good hash algorithm besides mixing your random bits. Perhaps in our next Security Application Note we will discuss them.

We are interested in your opinions and ideas regarding security functionality on our processors. However, before you call Centaur Technology with your suggestions, please take a look at our Programmers Guides to the RNG and ACE units. They will give you insight into our current implementation and will perhaps help you to understand what will be (relatively) an easy or a hard functionality to add.

Also, the Programmers Guides provide contact information.