Random Number Generation: Types and Techniques

David DiCarlo

A Senior Thesis submitted in partial fulfillment
of the requirements for graduation
in the Honors Program
Liberty University
Spring 2012

Acceptance of Senior Honors Thesis

This Senior Honors Thesis is accepted in partial fulfillment of the requirements for graduation from the Honors Program of Liberty University.

_____

Mark Shaneck, Ph.D.
Chairman of Thesis

_____

Terry Metzgar, Ph.D.
Committee Member

_____

Monty Kester, Ed.D.
Committee Member

_____

James Nutter, D.A.
Honors Program Director

_____

Date

Abstract

What does it mean to have random numbers? Without understanding where a group of numbers came from, it is impossible to know if they were randomly generated. However, common sense claims that if the process to generate these numbers is truly understood, then the numbers could not be random. Methods that are able to let their internal workings be known without sacrificing random results are what this paper sets out to describe. Beginning with a study of what it really means for something to be random, this paper dives into the topic of random number generators and summarizes the key areas. It covers the two main groups of generators, true-random and pseudo-random, and gives practical examples of both. To make the information more applicable, real life examples of currently used and currently available generators are provided as well. Knowing the how and why of a number sequence without knowing the values that will come is possible, and this thesis explains how it is accomplished.

Random Number Generation: Types and Techniques

A degree of randomness is built into the fabric of reality. It is impossible to say for certain what a baby's personality will be, how the temperature will fluctuate next week, or which way dice will land on their next roll. A planet in which everything could be predicted would be bland, and much of the excitement of life would be lost. Because randomness is so inherent in everyday life, many researchers have tried to either harvest or simulate its effect inside the digital realm. Before accomplishing this feat, however, many important questions need to be answered. What does it mean to be random? How does a person go about creating randomness, and how can he capture the randomness he encounters? How can someone know if an event or number sequence is random or not? Over generations, the answers to these questions have progressively been developed. This paper takes a look at the current solutions, and attempts to organize the methods for creating chaos.

Defining Random

It is impossible to appreciate a random number generator without first understanding what it means to be random. Developing a well-rounded definition of randomness can be accomplished by studying a random phenomenon, such as a dice roll, and exploring what qualities makes it random. To begin, imagine that a family game includes a die to make things more interesting. In the first turn, the die rolls a five. By itself, the roll of five is completely random. However, as the game goes on, the sequence of rolls is five, five, five, and five. The family playing the game will not take long to realize that the die they received probably is not random. From this illustration, it is

apparent that when discussing randomness, a sequence of random numbers should be the focus of the description, as opposed to the individual numbers themselves (Kenny, 2005). To make sure the next die the family buys is random, they roll it 200 times. This time, the die did not land on the same face every time, but half of the rolls came up as a one. This die would not be considered random either, because it has a disproportionate bias toward a specific number. To be random, the die should land on all possible values equally. In a third scenario, the dice manufacturer guarantees that now all its dice land on all numbers equally. Cautious, a family roles this new die 200 times to verify. Although the numbers were hit uniformly, the family realized that throughout the entire experiment the numbers always followed a sequence: five, six, one, two, etc. Once again, the randomness of the die would be questioned. For the die to be accepted as random, it could not have any obvious patterns in a sequence of dice rolls. If it can be predicted what will happen next, or anywhere in the future, the die cannot truly be random.

From the results of these dice illustrations a more formal definition of randomness can be constructed. A generally accepted and basic definition of a random number sequence is as follows: a random number sequence is uniformly distributed over all possible values and each number is independent of the numbers generated before it (Marsaglia, 2005). A random number generator can be defined as any system that creates random sequences like the one just defined. Unfortunately, time has shown that the requirements for a random number generator change greatly depending on the context in which it is used. When a random number generator is used in cryptography, it is vital that the past sequences can neither be discovered nor repeated; otherwise, attackers would be

able to break into systems (Kenny, 2005). The opposite is true when a generator is used in simulations. In this context, it is actually desirable to obtain the same random sequence multiple times. This allows for experiments that are performed based on changes in individual values. The new major requirement typical of simulations, especially Monte Carlo simulations, is that vast amounts of random numbers need to be generated quickly, since they are consumed quickly (Chan, 2009). For example, in a war simulator a new random number might be needed every time a soldier fires a weapon to determine if he hits his target. If a battle consists of hundreds or thousands of soldiers, providing a random generator quick enough to accommodate it is not trivial. Random numbers are often used in digital games and in statistical sampling as well. These last two categories put very few requirements on the random numbers other than that they be actually random. Inside each of these contexts, requirements even over the additional ones listed can exist depending on the specific application. There is a general definition describing a random number generator, but this definition needs to be tailored for each situation a generator is used in.

Types of Random Number Generators

With a description of randomness in hand, focus can shift to random number generators themselves and how they are constructed. Typically, whenever a random number generator is being discussed, its output is given in binary. Generators exist that have non-binary outputs, but whatever is produced can be converted into binary after the fact. There are two main types of random number generators. The first type attempts to capture random events in the real world to create its sequences. It is referred to as a true

random number generator, because in normal circumstances it is impossible for anyone to predict the next number in the sequence. The second camp believes that algorithms with unpredictable outputs (assuming no one knows the initial conditions) are sufficient to meet the requirements for randomness. The generators produced through algorithmic techniques are called pseudo-random generators, because in reality each value is determined based off the system's state, and is not truly random. To gain an understanding of how these generators work, specific examples from both categories will be examined.

True Random Number Generators

A true random number generator uses entropy sources that already exist instead of inventing them. Entropy refers to the amount of uncertainty about an outcome. Real word events such as coin flips have a high degree of entropy, because it is almost impossible to accurately predict what the end result will be. It is the source of entropy that makes a true random number generator unpredictable. Flipping coins and rolling dice are two ways entropy could be obtained for a generator, although the rate at which random numbers could be produced would be restricted. Low production rate is a problem that plagues most true random number generators (Foley, 2001). Another major disadvantage of these generators is that they rely on some sort of hardware. Since they use real world phenomena, some physical device capable of recording the event is needed. This can make true random generators a lot more expensive to implement, especially if the necessary device is not commonly used. It also means that the generators are vulnerable to physical attacks that can bias the number sequences. Finally, even when there are no

attackers present, physical devices are typically vulnerable to wear over time and errors in their construction that can naturally bias the sequences produced (Sunar, Martin, & Stinson, 2006). To overcome bias, most true random number generators have some sort of post processing algorithm that can compensate for it. Despite these disadvantages, there are many contexts where having number sequences that are neither artificially made nor reproducible is important enough to accept the obstacles. For security experts, there is a peace of mind that comes with knowing that no mathematician can break a code that does not exist. In the next sections, four major true random generators: Random.org, Hotbits, lasers, and oscillators will be covered.

Random.org. A widely used true random number generator is hosted on a website named Random.org. Random.org freely distributes the random sequences it generates, leading to a varied user base (Haahr, 2011). Applications of these numbers have ranged everywhere from an online backgammon server to a company that uses the numbers for random drug screenings (Kenny, 2005). However, since the numbers are obtained over the Internet, it would be unwise to use them for security purposes or situations where the sequence absolutely needs to stay private. There is always the risk that the transmission will be intercepted. The random number generator from this site collects its entropy from atmospheric noise. Radio devices pick up on the noise and run it through a postprocessor that converts it into a stream of binary ones and zeroes. Scholars have pointed out that the laws governing atmospheric noise are actually deterministic, so the sequences produced by this generator are not completely random (Random.org, 2012). The proponents of this claim believe that only quantum phenomena are truly

nondeterministic. Random.org has countered this argument by pointing out that the number of variables that would be required to predict the values of atmospheric noise are infeasible for humans to obtain. Guessing the next number produced would mean accurately recoding every broadcasting device and atmospheric fluctuation in the area, possibly even down to molecules. It has been certified by several third parties that the number sequences on this site pass the industry-standard test suites, making it a free and viable option for casual consumers of random numbers.

HotBits. The other popular free Internet-based random number generator is referred to as HotBits. This site generates its random number sequences based off of radioactive decay. Because this is a quantum-level phenomenon, there is no debate over whether the number sequences are truly non-deterministic. At the same time, the process involved in harvesting this phenomenon restricts HotBits to only producing numbers at the rate of 800 bits (100 bytes) per second (HotBits, 2012). Although the HotBits server stores a backlog of random numbers, the rate at which random sequences can be extracted is still limited in comparison to other options. As with Random.org, random numbers obtained from this generator are sent over the Internet, so there is always the possibility that a third party has knowledge of the sequence. This makes it unsuitable for security-focused applications, but Hotbits is useful when unquestionably random data is necessary.

Lasers. The use of lasers allows for true random number generators that overcome the obstacle of slow production. In laser-based generators, entropy can be obtained by several different means. Having two photons race to a destination is one

method that is currently implemented (Stefanov, et al., 2008). Another high-speed technique is measuring the varying intensity of a chaotic laser. Prototype systems in this second category have been created that can produce random bits at rates of over ten Gigabits per second (Li, Wang, & Zhang, 2010). The prototypes exhibited a natural bias toward one value over the other, so a post processor needed to be applied to create truly random sequences. A commonly used tactic is to take several bits at a time and exclusive-or them together to remove the unwanted bias. The stream of bits that emerged from this process was able to pass the most stringent randomness tests that are used for generators dealing with cryptography. Laser generators are capable of increased speeds, but they are complex to install and prohibitively expensive. Care needs to be taken during construction and installation that no bias is introduced, and the natural wear of the laser could also lead to it subtly producing more biased results over time. It is difficult to imagine laser-based generators being used in practical applications.

Oscillators. Oscillators, the final category of true random number generator to be discussed, make use of basic hardware, resulting in more convenient installation. An oscillator is a simple circuit obtained by placing an odd number of inverter gates in a loop. The final output of this configuration is undefined, since the current oscillates in a sine wave pattern over time. However, manufacturing is never perfect and defects always cause a slight and random deviation from a sine wave. These deviations are referred to as jitter, which is a common source of entropy in simple random number generators (Sunar, Martin, & Stinson, 2006). Because oscillators rely on manufacturing defects that cannot be replicated, they are part of a broader group of physical unclonable functions, or PUF

(Gassend, Clarke, Dijk, & Devadas, 2002). PUF are simple hardware that rely on unrepeatable idiosyncrasies during production to create random patterns. There are many types of PUF; oscillators are part of the delay category since jitter is caused by delay introduced by differences in the wires and silicon. To increase the randomness of jitter, oscillators of different lengths can be combined and evaluated together. Oscillators are cheap to install and use in comparison to other types of physical devices since their components are commonly used.

Random number generators based off of oscillators are vulnerable to many types of attacks. Environmental effects such as temperature changes and power surges can influence the jitter of a system. Attackers can change these variables intentionally to influence the random sequence for a limited time. These techniques are known as non-invasive attacks because they don't require direct contact to initiate. Invasive attacks can also be launched against oscillators. These attacks attempt to inflict a permanent defect inside the oscillator system, which will break the circuit and force a nonrandom output (Sunar, Martin, & Stinson, 2006). Fortunately, complexity can be added into oscillator-based generators that can thwart both types of attacks. When compared to pseudo-random generators that use tamper-proof algorithms, true random generators can appear to be very fragile. The actual case is a tradeoff between passive and active attacks. A passive attack occurs when the attacker does not need to alter the system, and is much harder to detect then active attacks, which often leave a footprint. Although true random number generators can suffer from active attacks, the pseudo-random generators that will be described are all vulnerable to passive attacks where the entire sequence past and future

can be predicted. In high-risk situations like cryptography, the potential setbacks of true generators are often preferable.

Pseudo Random Number Generators

Random number generators that do not rely on real world phenomena to produce their streams are referred to as pseudo random number generators. These generators appear to produce random sequences to anyone who does not know the secret initial value. In a minimalistic generator, the initial value will be the only time entropy is introduced into the system. Unlike true random number generators that convert entropy sources directly into sequences, a pseudo random needs to find entropy to use to keep itself unpredictable. Classic tactics for accomplishing this include taking the time of day, the location of the mouse, or the activity on the keyboard. Another way of explaining these sources is that they use the entropy of human interaction. This approach is frowned on in secure settings, because an attacker could purposely manipulate physical interactions to bias the system (Gutternman, Pinkas, & Reinman, 2006). If no human users interface with the hardware, generators are normally able to use other components on the system, such as hard drives, to generate entropy. Regardless, pseudo random number generators are limited by the entropy in their host device. These entropy sources all but determine the quality of the resulting sequences.

If the initial values of a pseudo random generator are known, every value in the sequence can be easily determined and even recalculated. This makes securing pseudo random generators against attackers of pivotal importance. The generator should be designed so that determining the internal variables at any given time is an infeasible task.

Going further, assuming that an attacker is able to determine the internal variables at a point in time, pseudo random generators should still be able to protect themselves. Forward security is the term used to describe a generator where knowing the internal state of a generator at a point in time will not help an attacker learn about previous outputs (Gutternman, Pinkas, & Reinman, 2006). If random number generators are being used for purposes like password creation, keeping up forward security becomes vital. Backward security denotes that an attacker who learns the state of the generator at a point in time will not be able to determine future numbers that will be produced. Backward security is only possible if the generator introduces some level of entropy into its equation. True random number generators always have forward and backward security, because they have no deterministic components.

Outside attackers are not the only problem inherent in pseudo random generators. At times honest or malicious mistakes can render an entire generator insecure. For example, the National Institute of Standards and Technology, or NIST, periodically publishes a list of pseudo random generators it deems secure enough for cryptography. In their 2007 publication, one of the four generators listed was championed by the National Security Agency, or NSA. It was called Dual_EC_DRBG (Schneier, 2007). Independent researchers quickly discovered that this generator contained a backdoor. Theoretically, there exists a set of constant numbers that, when known, would allow an attacker to predict every value of NSA's generator after collecting thirty-two bytes of random output. Although it is impossible to tell if the NSA possessed that set of constants, or even knew that such a thing existed, this served as a reminder that all pseudo random

generators should be thoroughly examined by experts before they are trusted. Since random numbers are used in many important applications such as setting up secure Internet communications, there are many groups that desire to crack the generators involved. Pseudo random number generators generally lack entropy after initialization, so once they are broken the attacker requires no additional effort to monitor the system. Additional complexity and thorough security checks need to be included in all pseudo random number generators to make them safe for public use.

Aside from vulnerability to attacks, all pseudo random generators share fundamental limitations. Without continual entropy, random generators can only create sequences based off of a limited set of initial conditions. Sequences created this way can only last a limited amount of time before they reach their starting point and repeat themselves exactly. The length a sequence can extend before it repeats itself is referred to as its period (Chan, 2009). A major consideration in the choice of a pseudo random number generator is the size of its period, because this directly affects the frequency that a generator can be used. Pseudo random generators are capable of producing sequences at rapid speeds, so in applications that use vast quantities of randomness, the threat of exceeding the period is not trivial. The field of cryptography also contributes to the creation of pseudo random numbers. Good encryption techniques that make messages undecipherable can also be used to encrypt a starting value into seemingly random numbers. Most encryption techniques have a poor production rate however, so using encryption techniques in generators is generally a bad idea (Trappe & Washington, 2006). In the next sections, several of the more common algorithms used in random

number generation, namely linear congruential, lagged fibonacci, and feedback shift registers, will be discussed.

Linear congruential generator. A simple example of a pseudo random number generator is the linear congruential generator. The formula for the algorithm it uses is: $\mathbf{s}_{i+1} = (a^* \mathbf{s}_i + c) \bmod m$ (Chan, 2009). This algorithm requires an initial value for $\mathbf{s}_o$, and chosen constants a, c, and m. If the constants are selected in accordance with the well-established rules publicly available, such as choosing an m value that is prime, then this algorithm will produce every value zero through m exactly once in its period, which is m, and is uniformly distributed. This algorithm demonstrates the point that pseudo random generators are deterministic. The entropy for this generator is the initial $\mathbf{s}_o$ value, since a, c, and m will most likely remain constant between uses. Once $\mathbf{s}_o$ is selected, the entire resulting sequence is solidified even before it is calculated. Although the sequence is unpredictable initially, once a seed value for the system is reused or an attacker recognizes the number sequence, then subsequent values can be predicted. The Boost library, a popular programming package, includes an implementation of a linear congruential generator (Chan, 2009).

Lagged Fibonacci generator. A major problem with simple linear algorithms is that the period is limited by m. With the lagged Fibonacci algorithm, much larger periods can be obtained. The basic form of this algorithm is $\mathbf{s}_{i+1} = \mathbf{s}_{i-p} \pm \mathbf{s}_{i-q} \bmod m$ (Chan, 2009). This algorithm can be used in conjunction with addition, subtraction, multiplication, or even exclusive-or (XOR). Because multiple past values are used instead of just the previous value as in the linear algorithms, values can be repeated inside

the period without indicating a loop. In the Boost library implementation of this algorithm, p is set to 44497, q is 21034, and the resulting period is around $2^{2300000}$ (Chan, 2009). A period of this size greatly reduces the likelihood of pattern recognition and secures the generator from the more basic forms of passive attacks.

Feedback shift registers. At times, it can be easier to describe pseudo random number generators in terms of hardware instead of their mathematical form. This is the case with feedback shift registers. These are best visualized as a string of n bits sitting inside a register in hardware. An even number of positions are selected: such as indexes five, seven, nine, and n. These generators operate by performing XOR on the bits at these positions, taking the result as the new leftmost bit, and shifting the rest of the string right by one (Dichtl, 2003). The bit that gets shifted out is the next random bit in the generators output. Mathematically, this algorithm can be written as $\mathbf{x}^p = \mathbf{x}^{p-t}_1 + \cdots + \mathbf{x}^{p-t}_n + \mathbf{x}^0$, where x is the bit string of length n, and t decides the index positions (Sunar, Martin, & Stinson, 2006). An advantage of these types of generators is that entropy can be easily added into the system, simply by including the new information into the XOR operation. Without entropy being introduced, shift registers have a period of $2^n - 1$, because zero will never be the result.

The Mersenne Twister is a very popular and widely used example of feedback shift registers in simulation and modeling (Nishimura, 2000). It is a good first choice when picking a pseudo random number generator. The Mersenne Twister can be classified as a twisted generalized feedback shift register (TGFSR), which has algorithms more tightly tied to matrices than strings. Adaptations exist both for making the generator

faster and making it secure enough for cryptography. The benefits of this generator are rapid number generation, highly random sequences, and a large period. Because this generator is so popular, implementations and source code examples are easily available.

Case Study: Linux Random Number Generator

This sub-section examines a concrete example of a widely used random number generator, specifically, the one used in the Linux operating system. Linux allows users to utilize its internal random numbers, and uses them itself for functions such as generating SSL keys, TCP sequence numbers, and random identifiers (Gutternman, Pinkas, & Reinman, 2006). The Linux pseudo random generator consists of three stores of random bits. The first store serves as the primary source of entropy. It uses signals sent from peripherals as its source of randomness. Whenever there is a keystroke, mouse movement, or interrupt, a 32 bit message is sent containing a timestamp and the details of the event that occurred. Each store keeps track of the number of entropy bits it currently contains. Whenever a store of bits needs to be used, its entropy counter is decremented by the amount of bits read, the extracted bits are encrypted with SHA-1 encryption, and new bits are added to the store that are derived from the bits it currently possess. New bits are added using a twisted generalized feedback shift register, similar to the Mersenne Twister. During each round, entropy is added to the string by inserting new random bits at a random location inside the register. Both the entropy bits to add and the position to insert them are determined by captured hardware events. The two remaining stores of bits are actually used by calls to the system's random functions. The first of these stores is for insecure random numbers, referenced by the /dev/urandom command (Gutternman,

Pinkas, & Reinman, 2006). This command will provide users with a specified amount of random bits. If there is not enough entropy in the urandom store to produce the bits, it will attempt to draw from the primary store of bits. In the event that there is still not enough entropy in the system, the urandom store will simply use the feedback shift register to produce pseudo-random bits to fill the gap. The third store of bits, which is used for the /dev/random command, is intended for high entropy sequences. In the event that this pool cannot find enough entropy for a request, it will block the operation until it can. Users receive random bits in groups of 32, a unit known as a word.

Many programmers have contributed to the Linux random number generator, but since it is only pseudo random, attacks have been found that can be used against it. The Linux generator lacks forward security. If the state of any of the stores is known at a point of time, all the previous outputs of that store can be computed back to the last time entropy was added (Gutternman, Pinkas, & Reinman, 2006). This stems from the fact that each store remains virtually the same between extractions, the only change being 96 bits around the random index *j*. To brute force the last random bit, an attacker would need to take the 2^{96} possible previous states of the store and see which one results in the current state after an extraction. The previous random bit is determined when the previous state is found, because it is a by-product of the transitional algorithm. Once the previous state is known, it is again possible to determine the 32 bits produced two calls ago. The only end to this cycle comes when the store is refreshed with random data, which leaves no deterministic algorithm that can crack it. In the world of computer security, any algorithm with more than 2^{80} possibilities is considered secure. With current technology, the time it

would take to try 2^{80} or more options is infeasible no matter how vital the data. Fourteen of the possible indexes, namely 18-31, are special in that they affect less of the pool than normal however. If it happens that the index *j* is one of these, then the complexity of determining the previous output falls down to 2^{64}, which is considered insecure for security purposes. At the same time, it is a relatively complex attack to use for only 32 bits of data, so the threat is not overwhelming.

Aside from the mathematical approach, there are side-channel attacks that can be used against the Linux random number generator. Unless the particular distribution of the operating system changes the setting, there is no limit to the amount of data that can be requested from the urandom command. An attacker can exploit this and request an infinite amount of data. The result is that all of the entropy in the primary and the urandom stores will be constantly depleted. Calls to random will be denied indefinitely as a result, and other users of urandom will get deterministic output, which can be figured by the previously described attack. Most likely, a monitored Linux system would be safe from this attack because the call for that much random data would set off red flags for administrators. A more far-fetched attack has also been described (Gutternman, Pinkas, & Reinman, 2006). If the Linux operating system is booted from a CD to a computer without a hard drive, then the only initial source of entropy will be keyboard events. Most often, the first thing entered into the keyboard is a user's password. Therefore, the entropy introduced to the pool would be exclusively generated by the user's password, and an attack on the store could theoretically extract it. The researchers who proposed the attack were not able to successfully complete it, although the machine they experimented

with had a hard drive (Gutternman, Pinkas, & Reinman, 2006). Because Linux uses a pseudo random generator, more attacks than the ones discussed likely exist. It is this threat of outsiders calculating secret random numbers that make the system less than truly random.

Case Study: RSA Key Generation

Encryption means nothing at all if the random number generators it relies on are poorly constructed. This fact became relevant on a global scale when a standard for encryption named RSA was shown to be predictable in some cases. RSA is a public-key algorithm that uses secret prime values to compute a modulus *n.* This *n* value used in both the encryption and decryption of messages, and security relies on its prime factors remaining secret. The range of possible prime numbers is large enough that they should never be reused or guessed. However, researchers found that they were able to remotely break the encryption on .4% of all the RSA signed certificates in their study, which included 5.8 million samples (Lenstra, et al., 2012). To break the encryption, researchers simply needed to use the Euclidean algorithm on all the samples to see if any of them shared a common factor in the modulus. If they did, then both of the RSA certificates could be broken, because the prime numbers were discovered. After studying the problem with RSA, the researchers found that its root was in the random number generators being used for these certificates. The repeated primes were mostly isolated to cases where the RSA certificate was created by an embedded device, such as a firewall, router, or printer. These devices typically do not have enough entropy to create unpredictable prime numbers. Also, devices from the same manufacturer likely have the same starting

condition for their generators, meaning that the first random numbers produced will be similar. Therefore, when these devices were used to create random prime numbers a couple primes were frequently repeated. That allowed the attackers to do their pair-wise comparison of certificates and break the resulting encryptions. This phenomenon reinforced the fact that strong random number generators form the backbone of modern cryptography.

Testing a Random Number Generator

Many sequences that appear random may actually be easy to predict. For this reason, it is important to thoroughly test any generator that claims to produce random results. A fitting description of random in this context is as follows: a random sequence is one that cannot be described by a sequence shorter than itself (L'Ecuyer, 2007). Attempting to find these patterns by intuition would be difficult if not impossible. Fortunately, many tests exist that can suggest if numbers in a sequence are random, and the algorithms in these tests are widely used outside of random numbers. Whenever professional forecasters make data-driven predictions, they apply formulas to determine the probability that the results were not random. These formulas can likewise be used to verify that a result was random, and the only thing that must change is the passing criteria. No test can conclusively prove randomness; the best that can be accomplished is that with enough testing, users of the generators can be confident that the sequence is random enough. There are scholars who believe that the source of a random sequence should dictate what tests to run. For example, true random generators tend to exhibit bias toward values, and this trait worsens as the hardware wears down. As a result, these

scholars claim that if a random sequence comes from a true random generator, extra tests should be performed that check for bias (Kenny, 2005). Other scholars believe that random is random regardless of where it came from, and that it is appropriate to test all random sequences the same. In the following subsections several statistical and exploratory tests will be examined, as well as the major test suites NIST and Diehard.

Statistical Tests

One of the approaches to testing random number generators is leveraging the wide array of statistical formulas. With this approach, each test examines a different quality that a random number generator should have. For example, a random generator would go through a chi-squared test to ensure a uniform distribution, and then a reverse-arrangements test to see if the sequences contained any trends. Confidence in the generator's randomness is only gained after it passes an entire suite of tests which comes at it from different directions. These tests should be run on more than just a single sequence to ensure that the test results are accurate. Making the act of testing more difficult is the fact that failing a test does not indicate that a generator is not random. When outputs are truly random, then there will be some isolated sequences produced that appear non-random (Haahr, 2011). Tests need to be picked carefully and tailored to the context generators are needed in. The chi-squared, runs, next bit, and matrix based tests will be examined because of their popularity.

Chi-squared test. The chi-squared test is used to ensure that available numbers are uniformly utilized in a sequence. This test is easy to understand and set up, so it is commonly used (Foley, 2001). The formula for the test is: $x = \sum (O_i - E_i)^2 / E_i$, where

the summation is over all the available categories. O represents the actual number of entries in the category, and E is the expected number of entries. For example, if the random numbers were scattered one through six, then there would be six categories, and the number of times each value appeared in the sequence would become the values of O. When the resulting value is above the chosen significance level, then it can be said that the values in the sequence are uniformly distributed. Because this test is simple, it can be run many times on different sequences with relative ease to increase the chance of its accuracy (Foley, 2001).

The runs test. An important trait for a random sequence is that it does not contain patterns. The test of runs above and below the median can be used to verify this property (Foley, 2001). In this test, the number of runs, or streaks of numbers, above or below the median value are counted. If the random sequence has an upward or downward trend, or some kind of cyclical pattern, the test of runs will pick up on it. The total number of runs and the number of values above and below the median are recorded from the sequence. Then, these values are used to compute a z-score to determine if numbers are appearing in a random order. The formula for the score is: $z = ((u \pm 0.5) - \mathbf{MEAN}_u) / \sigma_u$, where σ denotes the standard deviation of the sequence. (Foley, 2001). Just like the chi-squared test, a test for runs is easy to implement, and can be run frequently.

Next bit test. When testing pseudo random generators for cryptography applications, the next bit test is a staple. In its theoretical form, the next bit test declares that a generator is not random if given every number in the generated sequence up to that point there is an algorithm that can predict the next bit produced with significantly greater

than 50% accuracy (Lavasani & Eghlidos, 2009). This definition makes the next bit test virtually impossible to implement, because it would require trying every conceivable algorithm to predict the next bit. Instead, it can be used after a pattern is discovered to cement the fact that a generator is insecure. Several attempts have been made to alter the next bit test so that it can be used as an actual test. The universal next bit test developed in 1996 was the first to allow the next bit test to be administered, but it was shown that this test would pass non-random generators. Later, the practical next bit test was developed and was shown to be as accurate as the NIST test suite at the time, if not more so (Lavasani & Eghlidos, 2009). However, this test required a large amount of resources to run, limiting its usefulness. The next bit test remains relevant in cryptography because it has been proven that if a generator can pass the theoretical next bit test, then it will pass every other statistical test for randomness.

Matrix-based tests. Although the previous statistical test looks at the sequence of numbers linearly, a large portion of the more advanced testing techniques view it in terms of vectors and matrices. These tests will take the values of the generator, and sequentially add them into a hypercube, which is a cube with k dimensions (L'Ecuyer, 2007). From here the tests vary, although many of them conclude with a chi-squared calculation on their output (Chan, 2009). In a nearest pair test, Euclidean distances are computed and the nearest pairs are used to look for time-lagged patterns. It has been proposed that a large number of pseudo random generators based on circuits will fail the nearest pairs test, and it is one of the more stringent tests available. Multidimensional tests are often specifically tailored to look at random number generators, unlike the linear tests which are

multipurpose. The calculations in matrix-based tests are complex, and will not be presented in this thesis.

Exploratory Analysis

Although it is true that determining randomness by intuition is a poor choice, visualizing random sequences is a good preliminary method to understand the data. There are several ways that sequences can be plotted in an attempt to bring out any oddities inside the random generator. Figure 1 is a bitmap obtained from the rand() function in PHP on Windows (Haahr, 2011). By turning the sequence into a plot, it becomes more apparent that the given generator has some patterns in its sequences. In an ideal generator, the bitmap would seem like complete static, but in this example sections of black and white are clearly grouped. From this point, more specific tests can be decided on to determine just how severe the patterns actually are.

There is overlap between the problems that graphs can bring out and the problems that statistical techniques look for. A run sequence plot is designed to look for trends in a sequence, much like the test for runs. To create this plot, actual sequence values on the y-axis are compared against the values' indexes in the sequence (Foley, 2001). If there are trends in the sequence, this plot will make them easier to see. A histogram plot has a comparable purpose to the chi-squared test. It is composed of a bar graph, with the different categories on the x-axis plotted against the number of times that value appears on the y-axis. Any kind of non-uniformity will be brought out quickly this way, signaling that more tests concerning uniform distribution should be run.

Other exploratory graphs exist that look for unique types of problems. A lag plot graphs a value on the y-axis against the value that came before it on the x-axis (Foley, 2001). The purpose of this plot is to expose outliers in the data. If the lag plot has too many outliers, there is most likely a problem with the generator. Another unique exploratory tool is the autocorrelation plot. An autocorrelation plot examines the correlation of a value to the values that came before it at various intervals, called lags. If the plot displays no correlations between values at any lag, then the numbers are most likely independent of each other, which is a good indication of randomness. Using these exploratory plots allows analysts to get a feeling for the faults of a generator and better decide on which tests to run on the number sequences.

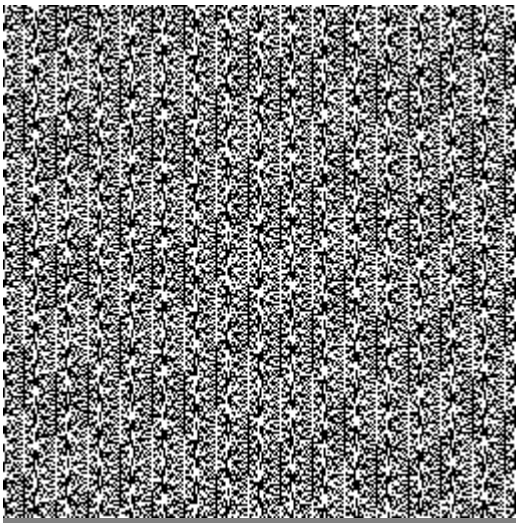

Figure 1. Obtained from Random.org

NIST

Of the available suites for testing random number generators, the NIST suite reigns as the industry standard (Kenny, 2005). The NIST suite was designed to test bit sequences, with the idea that passing all NIST tests means that a generator is fit for cryptographic purposes. Even new true random number generators have their preliminary results run through the NIST battery to demonstrate their potential (Li, Wang, & Zhang, 2010). The NIST suite contains fifteen well-documented statistical tests (NIST.gov, 2008). Because cryptography has the most stringent requirements for randomness out of

all the categories, a generator that passes the NIST suite is also random enough for all other applications. However when a generator fails the NIST suite, it could still be random enough to serve in areas such as gaming and simulation, since the consequences of using less than perfectly random information is small. NIST does not look at factors such as rate of production, so passing the NIST suite should not be the only factor when determining a generator's quality.

Diehard

Another widely used suite of random number tests is known as Diehard. This suite was invented by George Marsaglia in 1995 (Kenny, 2005). It was made to be an update for the original random number test suite, Knuth. Knuth is named after Donald Knuth and was published in the 1969 book *The Art of Computer Programming, Volume 2*. Knuth's tests were designed before cryptography became a major industry, and the suite was later considered to be too easy to pass for situations where vast quantities of random numbers were needed. Diehard was designed to be more difficult to pass than Knuth's suite, fulfilling the role of a general-purpose battery for detecting non-randomness. All of the tests are available free online, so they can be easily used to test any number sequence (Marsaglia, 2005). The Diehard suite has not been updated since its inception in 1995, but is still a widely used test suite (Kenny, 2005).

Conclusion

Many techniques are used to create the various types of random number generators. Although the idea of constructing a system that produces randomness can seem like a contradiction, decades of research has refined the art. On one end of the

spectrum, random number generators can serve as a funnel; they take random events from the real world as input and convert them into sequences of random numbers. Classically, these truly random number generators have been labeled as slow and difficult to install. However, the advent of laser-based generators is helping to solve the speed problem, while circuit-based generators are being designed that utilize existing hardware. True random number generators can be delicate though. Because they are constantly collecting feedback from outside phenomenon, care needs to be taken so attackers do not disrupt their environment.

On the other end of the spectrum, mathematicians and cryptographers have developed many algorithms that are unpredictable under certain circumstances. The pre-determined yet unforeseen sequences that result from these methods have been labeled pseudo random. Easily set up and able to produce values quickly, pseudo random generators are most commonly used. Normally, these generators need to keep their initial conditions and parameters a secret, or else anyone could compute the same number sequence. Even assuming that the initial conditions are not disclosed, pseudo random generators need to be designed in a way that recording part of the sequence or discovering the state of the generator does not allow new information to be computed. Unless these pseudo random generators have some method of refreshing themselves with real world entropy, they will eventually repeat themselves. If the designer of the random algorithm has malicious intent, it is possible for a backdoor to be installed that would allow outsiders to start predicting the numbers. Fortunately, all of these concerns about

security do not affect many applications that need random generators. Most times, a grab-and-go pseudo random generator can meet the needs of an application.

The method for selecting and appraising the most appropriate random number generator is highly dependent on context. If a high security application needs random numbers, then running a candidate generator through the NIST test suite would be appropriate. Preferably, this would be accompanied by researching the known attacks that can be launched against it. Perfect randomness and security is not the final say in selection however. When picking a generator for a simulation, quantity could win out over quality. Having the best randomness is not always relevant. In these cases, exploratory plots of the random generator could be used to determine what tests inside a suite such as Diehard should be run. Known vulnerabilities might be ignored entirely. Gaming takes a middle ground, because the random sequences only need to be good enough to keep players from predicting them. How hard the potential players are going to try is the baseline for how much testing and security analysis needs to be done. The growing demand for digital unpredictability has led the field of random number generation to grow rapidly in breadth and complexity. Fortunately, the types and techniques at the core of random number generators have remained stable for decades.

References

Callegari, S., Rovatti, R., Setti, G. (2005). Embeddable ADC-based true random number generator for cryptographic applications exploiting nonlinear signal processing and chaos. *IEEE Transactions On Signal Processing, 53*(2), 793-805.

Chan, H. (2009). *Random number generation.* Retrieved 10/16/2011 from http://fuchun00.dyndns.org/~mcmintro/random.pdf

Dichtl, M. (2003). How to predict the output of a hardware random number generator. *CHES 2003, 2779*, 181-188. Retrieved 10/16/2011 from http://www.springerlink.com/content/kkpghbg30r1xce4m/fulltext.pdf

Eastlake, D. (2005). Randomness requirements for security [RFC]. Retrieved 10/16/2011 from http://wiki.tools.ietf.org/html/rfc4086

Foley, L. (2001). Analysis of an on-line random number generator. Retrieved 10/16/2011 from http://www.random.org/analysis/Analysis2001.pdf

Gassend, B., Clarke, D., Dijk, M., Devadas, S. (2002). Silicon physical random functions. *Proceedings of the Computer and Communications Security Conference.*

Gutternman, Z., Pinkas, B., Reinman, T. (2006). Analysis of the linux random number generator. *Proceedings of the 2006 IEEE Symposium on Security and Privacy.*

Haahr, M. (2011). Random.org. Retrieved 10/16/2011 from http://www.random.org

Jin, A., Ling, D., Goh, A. (2004). Biohashing: Two factor authentication featuring fingerprint data and tokenised random number. *Pattern Recognition, 37*, 2245-2255.

Kenny, C. (2005). Random number generators: An evaluation and comparison of random.org and some commonly used generators. Retrieved 10/16/2011 from http://www.random.org/analysis/Analysis2005.pdf

Lavasani, A., Eghlidos, T. (2009). Practical next bit test for evaluating pseudorandom sequences. *Scientia Iranica*, *16*(1), 19-33.

Law, A., Kelton, D. (2000). *Simulation modeling and analysis* (3rd ed). AZ: McGraw-Hill Higher Education.

L'Ecuyer, P. (1992). Testing random number generators. *Proceedings of the 1992 Winter Simulation Conference (IEEE Press), pp. 305-313.*

Lenstra, A., Hughes, J., Augier, M., Bos, J., Kleinjung, T., Wachter, C. (2012). Ron was wrong, whit is right. Retrieved from http://eprint.iacr.org/2012/064

Li, P., Wang, Y., Zhang, J. (2010). All-optical fast random number generator. *Optics Express*, *18*(19).

Marsaglia, G. (2005). Random number generation. Retrieved 10/16/2011 from dl.acm.org/ft_gateway.cfm?id=1074752&ftid=634693&dwn=1&CFID=60160965&CFTOKEN=82346114

Marsaglia, G. (2005). The marsaglia random number CDROM including the diehard battery of tests of randomness. Retrieved 2/4/2012 from http://stat.fsu.edu/pub/diehard/

Nishimura, T. (2000). Tables of 64-bit mersenne twisters. *ACM Transactions on Modeling and Computer Simulation*, *10*(4), 348-357.

NIST (2008). Random number generation. Retrieved 10/16/2011 from http://csrc.nist.gov/groups/ST/toolkit/rng/index.html

Park, S., Miller, K. (1988). Random number generators: Good ones are hard to find. *Communications of the ACM*, *31*(10), 1192 - 1201.

Reeds, J. (1977). “Cracking” a random number generator. *Cryptologia, 1*(1), 1-6.

Schneier, B. (November, 2007). Did NSA put a secret backdoor in new encryption standard? *Wired*.

Stefanov, A., et. al. (2008). Optical quantum random number generator. Retrieved 10/16/2011 from http://arxiv.org/pdf/quant-ph/9907006v1

Sunar, B., Martin, W., Stinson, D. (2006). A provably secure true random number generator with built-in tolerance to active attacks. Retrieved 10/16/2011 from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.69.5788&rep=rep1&type=pdf

Trappe, L., Washington, L. (2006). *Introduction to cryptography with coding theory* (2nd ed). Upper Saddle River, NJ: Pearson.

Walker, J. (2006). HotBits: Genuine random numbers, generated by radioactive decay. Retrieved 10/16/2011 from http://www.fourmilab.ch/hotbits/