The Importance of Reliable Randomness

A Note on Random Number Generation and Combinatorial Sampling

Adam Young, Ph.D.
This Cigital white paper addresses common issues surrounding random number generation and combinatorial sampling. A practice on how to “mix” or “filter” weak sources of randomness to provide seeding material involves using a cryptographic hash function as an entropy extractor. This white paper shows a weakness in this approach due to the existence of alternate techniques with provably secure properties. Also, it is common practice to apply John von Neumann’s original algorithm to unbias potentially biased sources of entropy. This white paper presents an overview of and references to more efficient methods of doing so. Finally, it is common practice to generate random permutations over n objects by sampling n objects uniformly, then n-1 objects, then n-2 objects and so on to arrive at a random permutation. This approach is inefficient since better approaches from Algorithmic Combinatorics can be used. Such methods are described herein.

Key words: Random number generator, pseudorandom number generator, seed, RNG, PRNG, TRNG, biased distributions, sampling, combinatorial algorithms, computational intractability, quadratic residuosity problem, SHA-1, one-way hash function, entropy.

1 Introduction
Understanding how to generate and properly use random numbers is at once germane to Computer Science. Quality random number generation affects the quality of all randomized algorithms in practice, including, for instance, Monte Carlo algorithms. Consider a ubiquitous algorithm such as Quicksort. When it is employed with very poor random numbers, the quality of its operation may be suspect. Until recently, random number generation was relied upon to test the primality of large numbers [SS77, Mi76, Ra80]. Yet it was recently proven that primality testing can, in fact, be performed without relying on random numbers [ASK02], thus settling a question that plagued even the great mathematician Johann Carl Friedrich Gauss1. The fact that randomness is no longer needed to test primality helps prevent primality testing from being implemented improperly, as was the case in OpenSSL [Yo02]. Large prime numbers are essential for implementing such cryptosystems as RSA [RSA78] and the Diffie-Hellman key exchange [DH76]. There exist numerous problems that are known to be solvable using randomization. However, it is an open question as to whether or not randomization is even needed to solve many of these problems. For example, it is still not known whether randomness is needed to compute square roots modulo a prime p when p has no special form.

Generating and using random numbers correctly is subtle at best, and can completely undermine the security of a software application when performed incorrectly [AHM]. Since Cigital Labs is committed to improving software quality and reliability across the board, this white paper was written to bring scientists and practitioners alike up to speed on some of the common approaches to generating and using random numbers efficiently and securely. It is envisioned that this white paper will assist in a wide range of applications, including, but not limited to: online and off-line gambling software, smart-card applications, and cryptographic implementations. Gambling companies need reliable random number generation since it directly affects their bottom line. If a coalition of users were able to subvert an online casino, it could have lasting effects on both the revenue stream and reputation of the Internet casino.

This white paper is broken down into several sections that show how to utilize multiple sources of weak randomness to compute pseudorandom bits and pseudorandom permutations. Section 2 addresses the generation of truly random numbers. A specific true random number generator is described that can be implemented entirely in software. It was developed at AT&T and has not received the industry attention that it deserves. It is an incredibly clever way to derive randomness in software, provided that the underlying motherboard utilizes a real-time clock crystal and a CPU crystal. It is understood that this is a potentially biased source of randomness and, as such, methods are needed to eliminate any existing bias.

Section 3 addresses algorithms that are capable of removing the bias that might exist in random bit generators. Neumann’s celebrated algorithm for unbiasing a biased coin is given, followed by recent improvements to his approach. By applying Neumann unbiasing to a source of physical randomness, truly random bits result.

Section 4 addresses pseudorandom number generators (PRNG). A PRNG is an algorithm that takes as input a relatively small amount of truly random bits and outputs an even longer sequence of bits that are pseudorandom. In

---

1 Born 1777, died 1855. Made revolutionary discoveries in Mathematics (including Number Theory), Physics and Astronomy.
short, a PRNG is capable of “stretching” a sequence of truly random bits into a sequence that “looks random.” Technically speaking, a sequence of bits qualifies as being pseudorandom if it is computationally indistinguishable from a truly random sequence by all probabilistic poly-time Turing Machines.

Section 5 addresses how to select permutations over \( n \) objects uniformly at random. For example, by choosing a permutation over \( \{1,2,3,\ldots,52\} \) uniformly at random, a random shuffle of a deck of cards is being chosen. This section covers well-known results from Combinatorial Algorithmics.

Section 6 shows a way to combine all of these primitives together in a provably secure fashion, provided that multiple sources of weak randomness are available. The end result is a pseudorandom bit generator and algorithm that outputs combinatorial objects that are selected in a provably pseudorandom fashion.

Finally, Section 7 shows a common technique that programmers use to extract “entropy” from sources of potentially weak randomness. It explains how cryptographic hash functions such as SHA-1 are used as entropy extractors. SHA-1 was not designed for this purpose and has not been proven to operate in this fashion.

Several sections of this white paper were taken from a forthcoming book that the author is writing with Dr. Moti Yung from Columbia University. The book deals heavily with cryptography and, as a result, includes a significant amount of material on random number generation.

2 Physical Sources of Entropy

In 1995, Jack Lacy of AT&T laboratories gave a guest lecture at Columbia University in a graduate level Computer Security course taught by Dr. Matt Blaze. Lacy’s lecture covered AT&T’s cryptographic library CryptoLib, which he worked on while at AT&T. This library is described in a paper that was published in USENIX ’93 [LMS93]. One of the topics covered was the true random number generator called truerand, which is described in the USENIX paper as well as the original paper on Cryptotvirology [YY96]. The truerand algorithm is a software random number generator that exploits motherboard architectures to generate physically random bits. It exploits the fact that most motherboards come equipped with two crystal timing devices. One crystal is used to generate the real-time clock signal and the other is used to generate the CPU clock signal. The CPU crystal operates at a much higher frequency than the clock crystal. Truerand pits these two oscillators against one another and extracts randomness based on their discrepancies.

Truerand operates as follows. Initially, a global Boolean flag \( F \) is set to false and a local variable \( i \) is set to zero. Then, the address of a call-back function is passed as an argument to an operating system call that sets an alarm. The alarm is set to go off in one tick, which is typically \( 1/60 \) of a second. The call-back function sets \( F \) to true and then terminates. After the operating system call is made, a while loop is entered in which \( i \) is incremented by one in each iteration. The loop terminates only when \( F \) is set to true, which occurs when the call-back function is invoked by the operating system. The key to truerand’s operation is that it will not always take exactly one tick for the timer to go off, and hence the number of increments of \( i \) is dictated by the number of CPU instructions executed within that time. In theory, the inconsistencies in the total number of CPU instructions executed until the alarm goes off will be reflected most in the least significant bits of \( i \). In multitasking operating systems, great care must be taken to verify that the counter gets incremented a sufficient number of times.

In truerand, the 16 least significant bits of the counter are assumed to contain some true randomness. However, it is not realistic to think that for each of these bits a binary zero occurs with exactly the same probability as a binary “1.” Intuitively, one would think that the least significant bit would be selected with a probability that is closer to \( \frac{1}{2} \) than the other bits. However, closer is not good enough. To be a true random number generator, a binary 1 must be selected with probability \( \frac{1}{2} \): no more and no less. This observation implies that some method is needed to remove the biases in these bits if biases are in fact present.
Another well-studied physical source of entropy is physical phenomena associated with disk drives. It has been shown that chaotic air-turbulence is a good source of randomness in computers [DIPF94, JSHJ99]. This may be measured by taking into account seek-times as well as rotational latency in disk drives.

3 Unbiasing Biased Probability Distributions

This section addresses the problem of unbiasing biased coins, i.e., distributions over \{0,1\}. It then shows how to use a fair coin to sample from any set of elements in a uniform fashion. The section concludes with a description of how to remove any existing bias in the output of AT&T’s truerand. Intel also used the use of Neumann’s algorithm to “purify” the output of a physical source of randomness in their hardware RNG [JK99].

3.1 Neumann’s Algorithm

A biased coin is a coin that has a fixed probability of heads, denoted by \( p_h = \frac{1}{2} + \Delta \), where \( \Delta \) is a Real number contained in the interval \([-\frac{1}{2}, \frac{1}{2}]\). The value \( \Delta \) is referred to as the bias of the coin. Let \( p_t \) be the probability of tails. Hence, \( p_t = 1 - p_h \). The problem of unbiasing a biased coin is one whose fundamental nature is matched only by its elegant solution. The classic approach to removing bias is to use Neumann’s algorithm [Ne63] (see also [JJSH00]). If a source can be used to produce pairs of coin tosses where the same coin is used in each pair of tosses, then Neumann’s algorithm can be applied. If not, then Neumann’s algorithm cannot be correctly applied to the source. A variable bias may result from entropy sources that are sensitive to temperature changes, electromagnetic field fluctuations, etc.

It is instructional to consider the problem using a concrete example. Consider the problem of generating a fair coin toss given a coin that comes up heads with probability 5/8. Neumann’s solution is as follows: A series of experiments are performed in which the coin is tossed twice in succession. If in a given experiment the result is heads followed by tails, then the result of the toss is heads and no more experiments are performed. If in a given experiment the result is tails followed by heads, then the final outcome is tails and no more experiments are performed. If in a given experiment the tosses are the same, then another experiment is performed. This is depicted in the table below.

<table>
<thead>
<tr>
<th>Outcome</th>
<th>Probability</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heads-Heads</td>
<td>25/64</td>
<td>do over</td>
</tr>
<tr>
<td>Tails-Tails</td>
<td>9/64</td>
<td>do over</td>
</tr>
<tr>
<td>Heads-Tails</td>
<td>15/64</td>
<td>Heads</td>
</tr>
<tr>
<td>Tails-Heads</td>
<td>15/64</td>
<td>Tails</td>
</tr>
</tbody>
</table>

Table 1: Example of Neumann’s Algorithm

Observe that the probability of heads is the same as the probability of tails. So, given that the two tosses are not the same the answer will be heads with probability \( \frac{1}{2} \). The answer is therefore always correct when an experiment terminates. An interesting aspect of this algorithm is that the numerical value of the bias is not needed to generate fair coin tosses. The probability that a given experiment will terminate with an answer is 15/64 + 15/64 = 30/64. Although it is possible that this method will never terminate when executed, the chances of this are negligible for a bias of 1/8. Since the algorithm might never halt but always outputs the correct answer when it does halt, it belongs to a class of algorithms known as Las Vegas algorithms. The algorithm is tractable for any bias \( \Delta \) that is not overwhelmingly close to \( \frac{1}{2} \) or \(-\frac{1}{2}\).
Neumann's algorithm will now be formally analyzed.

**Lemma 1:**

When the input bits to Neumann's algorithm are generated by a biased coin,

Neumann's algorithm outputs a heads with probability $\frac{1}{2}$.

**Proof:** Consider the case that a value is output in iteration $j$. Since the two input bits for iteration $j$ are generated by a biased coin, it follows that the two flips are independent events. In iteration $j$, Neumann's algorithm outputs heads with probability $(\frac{1}{2} + \Delta)(\frac{1}{2} - \Delta)$. It outputs tails with probability $(\frac{1}{2} - \Delta)(\frac{1}{2} + \Delta)$. It follows from the commutative property that these probabilities are equal. QED.

### 3.2 Iterating Neumann's Algorithm

It has been shown that one can remove bias in a biased coin in a more efficient way than Neumann's original approach. This improvement effectively iterates John von Neumann's algorithm [Pe92].

<table>
<thead>
<tr>
<th>Input</th>
<th>Neumann</th>
<th>Improved Neumann</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>0011</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0100</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0101</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0110</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>0111</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1001</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1010</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>1011</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
<td>00</td>
</tr>
</tbody>
</table>
To show that Neumann's method can be improved, consider the table above. The middle column shows the output when Neumann's algorithm is applied to the input nibble\(^2\). For example, if the input is 0001, then the first two zeros cause a rejection and the 01 causes a binary zero to be output.

The rightmost column shows the output when Neumann's method is generalized to taking four bits as input. Let \(p_h\) denote the probability that an input bit is heads, where a heads is denoted by a binary "1." Let \(p_t\) denote the probability that an input bit is tails. So, \(p_t = 1 - p_h\). For example, \(p_h = 51/100\) implies that the input bits are biased towards being a "1" with probability 51%.

Recall from probability theory that if A and B are two events in a uniform probability space, the conditional probability of A given B is defined as follows.

\[
\Pr[A \mid B] = \frac{\Pr[A \cap B]}{\Pr[B]}
\]

The correctness of the improved Neumann algorithm will now be proven.

\[
\Pr[0 \text{ is output } \mid \text{ one bit is output}] = \frac{(p_t p_h^2)}{(2 p_h^2 p_t^2)} = \frac{1}{2}
\]

\[
\Pr[1 \text{ is output } \mid \text{ one bit is output}] = \frac{(p_h^2 p_t^2)}{(2 p_h^2 p_t^2)} = \frac{1}{2}
\]

It remains to consider the case that two bits are output. Below, the probability for 00 is derived.

\[
\Pr[00 \text{ is output } \mid \text{ two bits are output}] = \frac{(p_t^3 p_h + p_t^2 p_h^2 + p_t p_h^3 p_t)}{(4(p_t^3 p_h + p_t^2 p_h^2 + p_t p_h^3 p_t))} = \frac{1}{4}
\]

It is straightforward to show that the conditional probability is \(\frac{1}{4}\) for 01, 10, and 11 as well.

Why is this method an improvement on Neumann's algorithm? Observe that in each row in the table, the entry in the rightmost column has at least as many bits as the entry in the middle column, sometimes more. It therefore has a higher throughput than Neumann's algorithm.

It was mentioned by Jun and Kocher that the Intel RNG is capable of producing random bits at a rate of over 75 kbit/sec [JK99]. However, it was also mentioned that the Intel device is using Neumann's algorithm. This implies that Intel is not using the best available algorithm to remove bias. If Intel were to modify their design to employ an iterative version of Neumann's algorithm, it is quite possible that their throughput would be significantly improved.

### 3.3 Uniform Sampling

Suppose that we are interested in simulating the roll of a die but do not have one available, and suppose further that only a fair coin is available. The die roll can be simulated perfectly as follows: The coin is flipped three times to obtain a three-bit number. This number is uniformly distributed between 0 and 7 inclusive. If the number is between 0 and 5 then the number is output. Otherwise, the three bits are thrown out and this procedure is repeated. Since all six faces of the die are equally likely, this method samples \(\{0,1,2,3,4,5\}\) uniformly at random. Uniform sampling techniques are

---

\(^2\) One nibble equals four bits.
needed when a random bit generator is available and when elements must be sampled from a set that has a
cardinality that is not a power of two.

This general uniform sampling algorithm is as follows: Let RBG(i) denote a perfectly random bit generator that returns
a string consisting of i > 0 truly random bits.

Input: Integer N ≥ 1

Output: R chosen uniformly at random from [0,N-1]

Choose1toNRandomly(N):
1. Let T be the smallest power of two, such that T ≥ N
2. compute R = RBG(log₂(T))
3. if R < N then output R and halt
4. goto step (2)

Lemma 2:
Assuming the existence of RBG(i), Choose1toNRandomly() outputs R drawn uniformly at random from [0,N-1].

Proof: Observe that the strings which are output by RBG are chosen independently at random. Suppose that
Choose1toNRandomly() halts with R in iteration j. Let p_i,j denote the probability that R = i with 0 ≤ i < N in iteration j.
Clearly p_i,j = 1/T. Since step (3) is the only step which outputs a value, and since this value is R, it follows that
Choose1tnNRandomly outputs R drawn uniformly at random from [0,N-1]. QED.

In a given iteration of Choose1toNRandomly(), the value R will be less than N with probability N/T. So, the probability
that a given iteration causes a repeat in step (4) is 1-N/T. Since T is the smallest power of 2 such that T ≥ N, it follows
that N/T > ½. So, the probability that no answer is found after j iterations is,

\[(1 - N/T)^j < (1 - ½)^j = ½^j\]

It follows that Choose1toNRandomly() is an efficient Las Vegas algorithm.

3.4 Removing Bias in AT&T's truerand
Neumann's algorithm can be used to remove bias in the output of truerand using the following algorithm: In the first
step truerand is invoked to obtain the 16 least significant bits of the counter i. Let s_1 denote these 16 bits. Truerand is
then invoked again to obtain another set of 16 bits s_2. The least significant bits from each of these two sets are used
as a Neumann experiment, then the two penultimate bits are used as a second experiment, and so on. For
concreteness, suppose that s_1 and s_2 are as follows:

s_1 = 0100111010001011
s_2 = 0101111100101010
Experiment 1 results in 10, experiment 2 results in 11, experiment 3 results in 00, etc. A result of 10 is interpreted as a randomly chosen binary 1 and a result of 01 is interpreted as a randomly chosen binary 0. A result of 00 or 11 is regarded as a failed experiment. Let $p_{i,j}$ denote the probability that a binary 1 occurs in the $j^{th}$ bit of the $i^{th}$ trial, where $j = 0,1,...,15$ and $i = 1,2,3,4,...$. Consider the following two assumptions:

1. $p_{i,j} = p_{i+1,j}$ for $i = 1,3,5,7,...$ and $j = 0,1,2,...,15$.
2. For all $i$ there exists a $j$ contained in $\{0,1,2,...,15\}$ such that $p_{i,j}$ is neither overwhelming nor negligible.

Assumption (1) implies that truerand behaves consistently across pairs of invocations, which is necessary for correctness. Assumption (2) implies that in each trial $i$ there is at least one probability $p_{i,j}$ capable of producing a fair toss. For example, suppose that $p_{i,15} = 1$ for $i = 1,2,3,...$. In this case, the random number generator will never output a bit based on bit position $j = 15$. If for all $i$, $p_{i,j} = 1$ for $j = 0,1,2,...,15$, then truerand would not provide any randomness at all. It would, of course, be more efficient to purify the output of truerand by using the iterative version of Neumann's algorithm.

This example application illustrates an important aspect of properly using Neumann's algorithm: The two bits that are supplied to Neumann's algorithm must be produced with the same bias. Consider the problem of employing keyboard latency. Some writers, for instance, may produce keystrokes in bursts whenever they have an epiphany. The extraction of randomness should be immune to this possibility. One approach is to utilize the difference in time between keystrokes. Also, only the least significant bits should be used.

For example, if keystrokes occur at times $t_1,t_2,t_3,...,t_m$ then the differences will be $\Delta_1 = t_2 - t_1$, $\Delta_2 = t_3 - t_2$ and so forth. Let $b_1,b_2$ be the least significant bits of $\Delta_1$ and let $b_3,b_4$ be the least significant bits of $\Delta_2$. It is important to supply $(b_2,b_4)$ to Neumann's algorithm to get a bit for example, and then supply $(b_1,b_3)$. One would expect that by matching the significance of the bits, the biases if present would be matched as well.

4 Pseudorandom Number Generators

It is an unfortunate fact that programmers often implement pseudorandom number generators in an insecure fashion. This may be because they are not aware of the various approaches, or perhaps because it is perceived that the problem is not particularly difficult. Be that as it may, no one should be left to re-invent the wheel, especially when it comes to security-sensitive implementations. This is the reason that PRNGs have appeared in peer-reviewed and well-respected conferences and this is why an approach has even undergone standardization.

Subsection 4.1 presents a standardized approach that is heuristic in nature. This approach is ideal for applications that demand a high volume of pseudorandom bits per second. Subsection 4.2 covers PRNGs that have provably secure properties under well-accepted intractability assumptions.

4.1 Heuristic Pseudorandom Number Generation

Algorithm 5.11 below [MOV] is a U.S. Federal Information Processing Standard (FIPS)-approved method to pseudorandomly generate keys and initialization vectors for use with DES. It is from the ANSI X9.17 standard [X917].

---

3 This assumption can actually be weakened a little since truerand need only provide such a probability sufficiently often.
Algorithm 5.11: ANSI X9.17 pseudorandom bit generator

input: a random 64-bit seed \( s \), integer \( m \), and a DES EDE key \( k \)

output: \( m \) pseudorandom 64-bit strings \( x_1, x_2, \ldots, x_m \)

1. compute \( I = E_k(D) \) where \( D \) is a 64-bit representation of the date/time in as fine a resolution as possible.

2. for \( i = 1 \) to \( m \) do:
   3. \( x_i = E_k(I \ XOR \ s) \)
   4. \( s = E_k(x_i \ XOR \ I) \)

5. output \((x_1, x_2, \ldots, x_m)\)

This approach has undergone much scrutiny and is a sound way to generate pseudorandom bits provided that DES is replaced by a more modern cipher such as AES. Any company that employs this method for pseudorandom bit generation by replacing DES with AES is employing sound primitives.

Other conventional wisdom involves running DES (or AES) in output feedback mode and outputing half the bits in each “ciphertext” block as part of the overall pad that is output [Yu]. These approaches are often fast enough for even the most computationally demanding applications.

4.2 Pseudorandom Number Generation Based on Reduction Arguments

A definitive source for provably secure techniques regarding pseudorandom number generation is *Pseudorandomness and Cryptographic Applications*, by Princeton University Press [Lu96]. The definition of a pseudorandom bit generator (PRBG) is given below.

**Definition:** Let \( k, L \) be positive integers such that \( L \geq k+1 \) and \( L \) is a specified polynomial function of \( k \). A \((k,L)\)-PRBG is a function from \( k \)-bit strings to \( L \)-bit strings that can be computed in polynomial time (in \( k \)). The input to the PRBG is a \( k \)-bit seed \( s_0 \) and the output is an \( L \)-bit string which is pseudorandom.

A simple and relatively fast \((k,L)\)-PRBG is the Blum-Blum-Shub generator [BBS86]. It uses the parameters \( p, q \) and \( n \). The values \( p \) and \( q \) be two large distinct primes and \( n = pq \). These two primes must be kept secret. The Blum-Blum-Shub generator is defined as follows: Let \( s_0 \) be a quadratic residue modulo \( n \). The pseudorandom bit stream is found by computing \( z_i \) for \( i = 1, 2, 3, \ldots, L \).

\[
z_i = (s_0^i (2^i) \ mod \ n) \ mod \ 2
\]

It was originally shown that the output of the Blum-Blum-Shub generator could be \( \varepsilon \)-distinguished from \( L \) truly random bits if and only if there exists an unbiased Monte Carlo algorithm that solves the quadratic residues problem\(^4\) having an error probability of at most \( \Delta \), for any \( \Delta > 0 \). An even stronger result was shown by Vazirani and Vazirani [VV84]. They proved that this PRBG is secure under the weaker assumption that factoring is intractable.

\(^4\) The quadratic residues problem is to distinguish squares from pseudosquares modulo \( n \). The value \( r \) is a pseudosquare mod \( n \) if it is not a square and its Jacobi symbol is unity with respect to \( n \).
The Blum-Blum-Shub PRBG is also regarded as being secure when the $\log_2(\log_2(n))$ least significant bits of $s_0^{2^i}$ mod $n$ are used (instead of just the least significant bit). So, when $n$ is a 768 bit composite, the nine least significant bits can be used in the pseudorandom bit stream.

Why is this a favorable approach to generating bits pseudorandomly? The answer to that question is simple: Whereas DES, AES, etc. have only been around for a few decades or less, brilliant mathematicians have been trying to solve the factoring problem for centuries... and all have failed.

5 Random Permutation Generation

5.1 Shuffling Cards by Repeated Sampling
The following algorithm shuffles a deck of cards "in-place" and is thereby very memory efficient. Initially, deck[i] = i for $i = 1,2,3,...,52$.

CardShuffle():
1. for $i$ = 1 to 51 do:
2. $j$ = number drawn uniformly at random between $i$ and 52 inclusive
3. card = deck[$j$]
4. deck[$j$] = deck[$i$];
5. deck[$i$] = card

Assuming that a random (or pseudorandom) bit generator is available, step (2) above can be implemented using Neumann's method. For example, when $i = 1$ a 6 bit number $r$ is generated. If the number is between 0 and 51 inclusive, then we set $j = r + 1$. Otherwise, we generate another 6 bits and repeat. From Lemma 2 it follows that $j$ will be drawn from the correct probability distribution.

It remains to consider the running time of CardShuffle(). Suppose that a fair coin is available. How many times would one expect to have to flip it to get a heads? The answer is two. Suppose that a biased coin is available which comes up heads with probability $3/4$. How many times would one expect to have to flip it to get heads? The answer is $4/3$. The expected number of random bits needed to shuffle using CardShuffle() is found by summing the expected number of bits needed in each iteration. The expected number of bits needed per iteration is given below. The number immediately before the colon is $i$.

$$1: 6*(64/52) \quad 2: 6*(64/51) \quad 3: 6*(64/50) \quad \ldots \quad 20: 6*(64/33)$$
$$21: 5*(32/32) \quad 22: 5*(32/31) \quad 23: 5*(32/30) \quad \ldots \quad 36: 5*(32/17)$$
$$37: 4*(16/16) \quad 38: 4*(16/15) \quad 39: 4*(16/14) \quad \ldots \quad 44: 4*(16/9)$$
$$45: 3*(8/8) \quad 46: 3*(8/7) \quad 47: 3*(8/6) \quad 48: 3*(8/5)$$
$$49: 2*(4/4) \quad 50: 2*(4/3)$$
Cigital > The Importance of Reliable Randomness

51: 1*(2/2)

For example, in the first iteration when i = 1, one would expect to have to use 6(64/52) bits to draw the first card in the shuffle. Observe that 6*64 = 384 factors out of the top row, 5*32 = 160 factors out of the second row, etc. The top row is equivalent to the following:

\[ 6^*64(1/52 + 1/51 + 1/50 + \ldots + 1/33) \]

The rightmost term is the difference between two Harmonic numbers, namely H_{52} and H_{32}. Recall that the n^{th} Harmonic number, H_{n}, is defined by:

\[ H_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n} = \text{sum from 1 to n of } (1/k) \]

This definition was taken from *Concrete Mathematics*, by D. Knuth et al. [GPK94]. Clearly all Harmonic numbers are rational. The above expression is equivalent to,

\[ 384(H_{52} - H_{32}) + 160(H_{32} - H_{16}) + 64(H_{16} - H_8) + 24(H_8 - H_4) + 2 + 8/3 + 1 \]

After simplifying, the following expression is obtained.

\[ 384 H_{52} - 224 H_{32} - 96 H_{16} - 40 H_8 - 24 H_4 + 5 + 2/3 \]

These five Harmonic numbers are given below [Gi].

\[ H_4 = 25/12 \]
\[ H_8 = 761/280 \]
\[ H_{16} = 2436559/720720 \]
\[ H_{32} = 586061125622639/144403552893600 \]
\[ H_{52} = 14063600165435720745359/3099044504245996706400 \]

H_{32} is approximately 4.0585 and H_{52} is approximately 4.53804. The expected number of random bits needed to shuffle a single deck is 355.9 using this method.

5.2 Shuffling Cards using Trotter-Johnson

Algorithms to generate and enumerate permutations fall under the category of Algorithmic Combinatorics. Ranking and Unranking algorithms allow computer scientists to efficiently store, generate and use combinatorial objects. Consider the problem of storing a particular permutation of n objects in a computer. If the objects are numbers for instance, one could simply store them in an array. However, a more efficient way is to establish a bijection between the n! combinatorial objects and the natural numbers from 0 to n!-1 and subsequently store the natural number that uniquely identifies the object.

When a Ranking function is supplied with a combinatorial object it returns the object’s rank, i.e., a natural number that uniquely represents the object. When a rank is given to an Unranking function, it returns the corresponding combinatorial object. An example of such a bijection will go a long way toward illustrating this concept. Consider the
problem of establishing a bijection between \( \{0,1,2,\ldots,9\} \) and the 5 choose 3 subsets of \( \{1,2,3,4,5\} \) containing three elements. The table below depicts a co-lex ordering that defines such a bijection.

<table>
<thead>
<tr>
<th>T</th>
<th>Rank(T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3,2,1]</td>
<td>0</td>
</tr>
<tr>
<td>[4,2,1]</td>
<td>1</td>
</tr>
<tr>
<td>[4,3,1]</td>
<td>2</td>
</tr>
<tr>
<td>[4,3,2]</td>
<td>3</td>
</tr>
<tr>
<td>[5,2,1]</td>
<td>4</td>
</tr>
<tr>
<td>[5,3,1]</td>
<td>5</td>
</tr>
<tr>
<td>[5,3,2]</td>
<td>6</td>
</tr>
<tr>
<td>[5,4,1]</td>
<td>7</td>
</tr>
<tr>
<td>[5,4,2]</td>
<td>8</td>
</tr>
<tr>
<td>[5,4,3]</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 3: Example of Co-Lex Ordering

The Trotter-Johnson algorithm is a minimal change algorithm for generating the \( n! \) permutations. It is a well-known algorithm developed in 1962 by Trotter and 1963 by Johnson [Tr62,Jo63]. In this algorithm, \( \pi \) is a permutation over \( \{1,2,3,\ldots,n\} \). The value \( r \) is a rank of such a permutation. Hence, \( r \) is contained in \( \{0,1,2,\ldots,n!-1\} \). The description below is due to Kreher and Stinson [KS99]. Denote by floor\((x)\) the smallest integer less than or equal to \( x \).

Algorithm 2.18: TrotterJohnsonUnrank\((n,r)\):

1. \( \pi[1] = 1 \)
2. \( r_2 = 0 \)
3. for \( j = 2 \) to \( n \) do:
4. \( r_1 = \text{floor}(r j! / n!) \)
5. \( k = r_1 \cdot j r_2 \)
6. if \( r_2 \) is even then

\[ \text{See also [Ev73].} \]
for $i = j - 1$ down to $j \cdot k$ do:

7. $\pi [i + 1] = \pi [i]$

8. $\pi [j - k] = j$

9. else

10. for $i = j - 1$ down to $k + 1$ do:

11. $\pi [i + 1] = \pi [i]$

12. $\pi [k + 1] = j$

13. $r_2 = r_1$

14. return $(\pi)$

This algorithm is the natural way to select a particular ordering of the 52 cards in a deck. The actual value of $52!$ is needed to implement shuffling based on unranking. When expressed in decimal the value of $52!$ is,

$$8065817517094387571660636856403766975289505440883277824000000000000$$

Expressed in hexadecimal, the value of $52!$ is

$$2FDE529A3274C649CFEB4B180ADB5CB9602A9E0638AB2000000000000$$

The abundance of zeros on the right side of these numbers is due to the fact that every other number in $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdots \cdot 52$ is evenly divisible by two and every other fifth number is evenly divisible by five. The number of binary digits needed to express $52!$ is exactly 226. The program used to compute these numbers is given in Appendix A.

Given the above value for $52!$, Trotter-Johnson unranking can be used to shuffle as follows: A random 226 bit number $r$ is chosen. If the number is less than $52!$ then it is supplied to Algorithm 2.18 with $n = 52$. The algorithm will return the shuffle $\pi$. If $r \geq 52!$, then $r$ is discarded, another 226 bits are chosen randomly and this process repeats.

Multiprecision libraries such as OpenSSL contain routines for testing if one big number is less than another. To test if $x < y$ the algorithm scans the bits of $x$ from the most significant bits to least significant bits. The algorithm does this with $y$ at the same time. The algorithm makes a determination if and when two bits differ.

Now consider the running time of this approach. Observe that,

$$52! \text{ is approximately } 80.658175 \times 10^{22}$$

$$2^{226} \text{ is approximately } 107.839787 \times 10^{22}$$

So, one would expect to have to generate a sequence of 226 bits about 1.337 times. It follows that using TrotterJohnsonUnrank, about 302.2 random bits will be needed on average to shuffle a single deck. Since $302.2 < 355.6$, it is more efficient to use TrotterJohnsonUnrank to shuffle cards. When 50,000 shuffles are performed, we
would expect TrotterJohnsonUnrank to use up about 1.8 Megabytes of randomness. The iterated approach would use up approximately 2.12 megabytes of randomness. The unranking algorithm helps to pave the way for using PRNGs based on rigorous mathematical foundations, which tend to be more computationally demanding than ad hoc constructions.

6 Putting it all together

6.1 A Solid PRBG from Weak Entropy Sources
Consider a true RNG predicated on the “randomness” derived from keyboard latency. What if the user breaks his or her arm and has to type methodically with one hand? This may eliminate some if not all of the randomness for a time. What if a solution based on the Intel RNG suffers a breakdown, thus rendering the Intel RNG unavailable? Such a solution would then have to rely entirely on other sources of randomness until the failure is rectified. Fault tolerance in random number generation is an important issue to address in building any robust system that requires access to random bits.

A solid approach to generating pseudorandom numbers based on physical randomness is given in Figure 1, above. The three dark grey boxes are separate sources that are believed to provide a measurable amount of true entropy. Each source is treated separately and the output of each source is sent through an unbiasing algorithm as indicated by the light grey boxes. The Intel hardware RNG uses two free-running oscillators: one that is fast and second that is much slower. A thermal noise source is used to modulate the frequency of the slower clock signal. The slower clock triggers measurements of the faster clock. The drift between the two is used to generate bits, which may be biased.

---

6 This is using the common definition of a kilobyte in which 1 kilobyte equals 1,024 bytes (not 1,000 bytes).
7 Look for key, press key, look for key, press key. Commonly referred to as “hunting and pecking.”
The Intel hardware RNG applies Neumann’s algorithm internally to produce an unbiased stream (in theory). This is why the output of the Intel RNG does not go through a light grey box in Figure 1.

The four bit streams produced by the four light grey boxes are then bitwise exclusive-or’ed. This is indicated by the circle with cross hairs. Observe that if one of the physical sources of randomness behaves like a biased coin then exactly one of the streams going into the bitwise XOR operation will be completely random. This is the heart of the fault-tolerant random number generation method. Suppose that all but one of the sources is compromised, either due to device failure or a malicious adversary. As long as one of the streams is truly random, the output of the XOR operation will be truly random. This is readily apparent based on the methodology used in the Vernam Cipher\(^8\). In fact, it is sufficient that in the bitwise XOR operation, one bit in each bit position be completely random. If a device were to shut down, then the stream that it would normally contribute to the XOR operation can be set to all zeros. This will have a null effect on the bitwise XOR operation.

A provably secure PRNG will operate in a secure fashion if and only if the following three conditions hold:

1. The secret PRNG parameters must be kept secret. This includes the initial seed (and the primes as in the case of the Blum-Blum-Shub PRNG, etc.).
2. The parameters for the PRNG must be correct. This means that the seed must be chosen perfectly at random (and the primes in Blum-Blum-Shub must be chosen correctly, etc.).
3. The underlying computational intractability assumption (or assumptions) must hold. In the case of the Blum-Blum-Shub PRNG, this means that factoring must in fact be hard.

If any of the above conditions do not hold, the PRNG may be compromised.

For example, 32 bytes may be collected from each unbiasing algorithm. These four sets of 32 bytes are exclusive-or’ed together and the result is used to seed and run a pseudorandom bit generator (more bytes may be required if suitable PRNG parameters cannot be sampled from the 32 bytes alone). Once seeded, exactly L bits may be taken from the pseudorandom bit generator. If more bits are necessary, then this whole process can be repeated. This differs from the approach Intel took by constantly providing the PRNG with fresh randomness, yet it has the same overall effect.

Finally, if combinatorial objects need to be selected uniformly at random, then the output of the pseudorandom bit generator can be used as input into a combinatorial unranking algorithm. To generate random permutations, the Trotter-Johnson unranking algorithm can be used. Like the unbiasing algorithm, the unranking algorithm is of the Las Vegas type. It may have to reject sequences of random bits in order to produce an output drawn from the proper probability distribution.

### 6.2 Proof of Correctness

This entire system is provably correct assuming that (1) one of the physical sources of randomness acts like a biased coin and (2) the underlying intractability assumption(s) hold. The redundancy of using multiple physical sources for seed material makes the overall solution robust against failure.

Let \( B_i = b_{i,1} b_{i,2} ... b_{i,k} \) denote the bit string which is output by Neumann’s algorithm when applied to entropy source \( i \). Each string \( B_i \) is \( k \) bits in length. In Figure 1, it follows that \( 1 \leq i \leq 4 \). The strings \( B_1, B_2, B_3, \) and \( B_4 \) are bitwse exclusive-or’ed to obtain a \( k \)-bit seed \( S = s_1 s_2 ... s_k \) using the following algorithm:

\[
\text{XORTheBitStrings}(B_1, B_2, B_3, B_4):
\]

1. for \( j = 1 \) to \( k \) do:

---

\(^8\) Also known as the one-time pad.
2. \[ s_i = b_{1i} \oplus b_{2i} \oplus b_{3i} \oplus b_{4i} \]

3. output the bit string \( S = s_1 s_2 ... s_k \)

The value \( S \) is used to seed the pseudorandom bit generator\(^9\).

**Theorem 1:**

If for all \( j \) where \( 1 \leq j \leq k \) there exists an \( i \) contained in \([1,4]\) such that \( b_{ii} \) is chosen uniformly at random, then \( S \) is a truly random \( k \)-bit string.

**Proof:** Consider bit position \( j \) where \( 1 \leq j \leq k \). Without loss of generality, let \( b_{ij} \) be a bit that is chosen uniformly at random, with \( 1 \leq i \leq 4 \). Let \( p_h \) denote the probability that the three bits in bit position \( j \) other than \( b_{ij} \) exclusive-or to the value “heads” where heads corresponds to 1. Hence, \[ p_h = Pr[b_{1j} \oplus b_{2j} \oplus b_{3j} \oplus b_{4j} \oplus b_{ij}] = 1 \]

The probability that \( s_j = 1 \) is \( (\frac{1}{2})p_h + (\frac{1}{2})(1 - p_h) = \frac{1}{2} \). This is the probability that bit \( b_{ij} = 0 \) and \( p_h \) results in heads plus the probability that bit \( b_{ij} = 1 \) and \( p_h \) results in tails. QED.

Observe that the perfect randomness in each bit in \( S \) is closely related to the security of each plaintext bit in a Vernam ciphertext\(^10\). The unconditional security of the One-Time Pad holds regardless of the probability distribution over the message space.

The following theorem surmises Cigital's provably secure approach to pseudorandom bit generation based on multiple sources of randomness. For concreteness, it is assumed that the Blum-Blum-Shub pseudorandom bit generator is used.

**Theorem 2:**

If one of the four sources of entropy is sampled to form a biased coin that is not overwhelmingly biased towards heads or tails\(^11\) and if the integer factorization problem in intractable, then the pseudorandom bit generator outputs \( L \)-bit pseudorandom strings.

**Proof:** Suppose that entropy source \( i \) where \( 1 \leq i \leq 4 \) is sampled to form a biased coin that is not overwhelmingly biased towards heads or tails. It follows from Lemma 1 that Neumann's algorithm will produce a truly random \( k \)-bit string \( B \). Furthermore, from Theorem 1 it follows that \( S \) will be a truly random \( k \)-bit string. Observe that \( S \) is used to seed a \((k,L)\)-PRBG. Since the Blum-Blum-Shub PRBG is secure if and only if factorization is hard and since it is assumed that factoring is hard, the theorem is proved. QED.

In addition to being provably secure when a biased coin is available, this solution produces provably quasi-random seeds when such a coin is not available, provided that enough sources of weak entropy are used. This follows immediately from the work of Santha and Vazirani [SV84], who weaken the assumption that a biased coin is available. The pseudorandomness results from the fact that several weakly random bit streams are bitwise exclusive-or'ed with each other.

The basic idea may be illustrated using several examples. Consider the two entropy sources in the figure below.

---

\(^9\) Other \( k \)-bit strings may need to be obtained as well, e.g., to compute the primes \( p \) and \( q \) in Blum-Blum-Shub.

\(^10\) Gilbert Vernam published the One-Time pad cryptosystem in 1926 [Ve26] and it wasn't proven to provide perfect secrecy until some 30 years later [Sh49].

\(^11\) As illustrated by Theorem 1 this assumption can be relaxed a bit.
Consider the first row in the table above. A final result of heads occurs if source 1 results in heads and source 2 results in tails, or if source 1 results in tails and source 2 results in heads. This is \((5/8)(2/8) + (3/8)(6/8) = 28/64 = 7/16\). A coin flip that results from the XOR of the bits from the two sources is in every case the same or better than the best flip in either of the two sources.

The Santha-Vazirani algorithm is a good way to combine weak entropy sources to guarantee that the randomness improves. However, the state of the art in entropy extraction has advanced considerably in recent years. There is a wealth of scientific literature on the subject [Ta96, TUZ01, SU01, TZS01, NT99]. Entropy extraction is an area of theoretical computer science unto itself.

6.3 A variant that utilizes an avalanche effect

It may be argued that this approach is vulnerable to specific scenarios. For example, suppose that all sources of randomness fail except the keyboard latency source. Furthermore, suppose that, for whatever reason, the first output bit after unbiased can always be predicted correctly. This could imply that a particular bit in the seed is known. This systems-level consideration can be addressed in practice by changing the design slightly. For example, the output of the bitwise XOR operation can be run through a cryptographic hash function such as SHA-1. This will “mix” the input bits considerably and help eliminate the possibility of correlations.

However, choosing the appropriate hash function is critical. For example, suppose that an adversary where to provide a black-box and say that it is a cryptographic hash function. Suppose it has a 32-byte input and produces a 160-byte hash. Even if the 32 input bytes are truly random, this does not necessarily mean that the 160 output bits will be truly random as well. To see this, note that the device could for example take the last three bytes of the input and throw the rest away. It could pad these three bytes with leading zeros, supply the result to SHA-1, and then output that which SHA-1 outputs. To an outside observer, the 20 output bytes may in fact look “completely random.” However, they are clearly not.

The point of this argument is to show that it may be perilous to assume that the outputs of a cryptographic hash function are random simply because they look random. SHA-1 was designed to be difficult to invert based on a toolbox of cryptographic and cryptanalytic know-how developed over decades. It was not designed to generate random numbers per se, and to date it has not exhibited the provably secure properties of algorithms such as Blum-

\[12\] A coin flip that is heads with probability 3/8 is in some sense just as erroneous as one with probability 5/8 since the absolute values of the biases are identical.
Blum-Shub. Its primary function is to hash messages in the Digital Signature Standard to prevent existential forgeries of digital signatures. 

7 How Not To Extract Entropy From Entropy Sources

It is not an uncommon practice to "collect" entropy from various sources to produce a byte-stream and then hash this byte stream using SHA-1 for instance, to derive a "random" seed. This heuristic is very poor indeed since provably secure methods exist that use weak entropy sources to generate perfectly random or perfectly quasirandom bits.

Consider SHA-1 for these purposes. SHA-1 is based directly on the MD4 algorithm. MD4 is a customized hash function designed with the explicit purpose of hashing with optimized performance in mind. The original MD4 design goals were:

1. To make finding a collision difficult, requiring approximately $2^{64}$ operations to do so (collision intractability); and
2. To make finding a message yielding a pre-specified hash value difficult, requiring approximately $2^{128}$ operations to do so (non-invertability).

Despite the fact that finding collisions was supposed to be difficult, a collision in MD4 was in fact found [Do95]. This demonstrates the frailty of relying on primitives that cannot be proven to be correct.

Collision resistance and non-invertability were also the design criteria used in devising SHA [FIPS92]. SHA was not designed to output "random looking hash values" and it was not designed to "extract truly random bits from inputs with a sufficient amount of Entropy." The fact that it may appear to do so on the surface is immaterial from the standpoint of security, but nonetheless probably explains why so many programmers opt to use it for these purposes. Incidentally, properties (1) and (2) above were designed to provide sufficient conditions for using a hash function to heuristically protect against existential forgeries in digital signature schemes.

The use of SHA-1 to extract 160 "random" bits from the collected entropy strings is perilous since there is no evidence to suggest that SHA-1 does in fact do this. This use of SHA-1 assumes that it is a "magic box" that can magically extract entropy from the input string and output a truly random 160-bit string.

This view is shared by Dr. Ari Juels (Principal Research Scientist, RSA), Dr. Markus Jakobsson (Principal Research Scientist, RSA), Dr. Liddy Shriver (MTS, Bell Labs) and Dr. Bruce Hillyer (MTS, Bell Labs). The following quote is from their paper, titled How to Turn Loaded Dice into Fair Coins [JJSH00]:

"Timings of human interaction with a keyboard or mouse are currently the most common source of random seeds for cryptographic applications on PCs. After a sufficient amount of such timing data is gathered, it is generally hashed down to a 128-bit or 160-bit seed. This method relies for its security guarantees on unproven or unprovable assumptions about the entropy generated by human users [RFC1750] and the robustness of hash functions as entropy extractors."

The danger of using a "complex" algorithm to produce strong random numbers in the absence of a theoretical foundation or analysis was noted by D. Eastlake (DEC), S. Crocker (Cybercash), and J. Schiller (MIT) in RFC 1750 [RFC1750]:

"Another serious strategy error is to assume that a very complex pseudo-random number generation algorithm will produce strong random numbers when there has been no theory behind or analysis of the algorithm."

---

13 Even this use is heuristic in nature, as illustrated by Pointcheval and Stern's more rigorous approach to proving the security of signature schemes based on the Forking Lemma.

14 Markus served on the author's Ph.D. committee and informed him of [JJSH00] soon after it was published.
There has been no theory or analysis behind SHA-1’s ability to extract entropy from its input. The complexity of the SHA-1 algorithm in no way justifies its use as a magic box.

Consider the case in which entropy is taken from two sources a weak source derived from user input (e.g., message arrival, mouse inputs, etc.) and possibly even guessable inputs (e.g., system time), and a strong source that is a hardware RNG, for instance. There is strong reason to suspect that the clock input is poor indeed. The following is from RFC 1750 [RFC1750]:

"Computer clocks, or similar operating system or hardware values, provide significantly fewer real bits of unpredictability than might appear from their specifications."

Suppose that 32 bytes are derived solely from the weak source and 32 bytes are taken directly from the strong source. Finally, suppose that these two sequences are concatenated and fed as input to SHA-1 to produce a 20-byte value. The use of SHA-1 in this fashion could in fact result in a seed which is significantly less random than a 160-bit string that is taken directly from the Intel RNG. It is conceivable that SHA-1 mixes this low-quality randomness with the high-quality randomness and outputs bits having a degree of randomness that is somewhere between the two extremes. Again, since there is no proof regarding SHA-1’s true behavior as an entropy extractor, any arguments against this possibility can only be pure speculation.

An even more extreme possibility exists. Consider the case that not enough user input is obtained and padding bytes are used to supplement the weak source of entropy to derive 32 bytes of weak entropy. Since a hash function that maps 64 bytes to 20 bytes acts like a compression function, it is information theoretically possible that the hash function will output a value that draws most of its entropy from the weak source and the padding (which has no entropy at all). In this case, the resulting value may have even less entropy than the bytes provided by the weak source of entropy.

8 Conclusion
A series of techniques encompassing the use of physical sources of entropy, unbiasing probability distributions, cryptographically stretching random seeds, and randomly sampling combinatorial objects was presented herein. It was shown that reliable randomness can be obtained from weak sources of randomness, even when a true biased coin is not available. It is hoped that these methods will be adopted by application developers and system architects alike. In bringing these scientific methods to the forefront, Cigital hopes that software practices can be improved for the betterment of everyone involved in Information Sciences.

Adam Young, Ph.D. is a Research Scientist at Cigital. He can be reached at ayoung@cigital.com.

About Cigital, Inc.
Cigital helps companies build better software faster. For more than a decade, our Software Quality Management solutions have enabled Global 1,000 companies to repeatedly deliver better software by dramatically increasing software reliability, performance and security. Cigital improves “speed-to-quality” by collaborating with clients to identify and fix software flaws early in the development process, where they are more easily and cost-effectively corrected.

Based on a host of patent-pending technologies invented by the computer scientists of Cigital Labs, our award-winning research facility, our solutions provide decision support to executives who are responsible for delivering quality software products to market as quickly and cost-effectively as possible.

Cigital is headquartered in Dulles, VA with additional offices in Boston.

www.cigital.com
References


[Yo02] Adam Young. Found bug in Rabin-Miller probabilistic primality test in OpenSSL. It was fixed in distributions 0.9.6a and later.

Appendix A: Program to Compute 52!

The following program was used to compute the value of 52! in both decimal and hexadecimal. It also computes the exact number of bits needed to represent 52! in binary form. It was compiled using gcc in Cygwin [Cygwin]. The OpenSSL 0.9.7a-1 multiprecision library was used to perform the multiprecision arithmetic [OpenSSL].

```
#include <stdio.h>
#include <openssl/bn.h>

int main(int argc,char **argv)
{
  int i;
  BIGNUM *x,*y;
  BN_CTX *thectx;
  thectx = BN_CTX_new();
  x = BN_new();y = BN_new();
  BN_set_word(x,1);
  for (i=1;i<=52;i++)
    {
    BN_set_word(y,i);
    BN_mul(x,x,y,thectx);
    }
  printf("x_10 = %s.\n",BN_bn2dec(x));
  printf("x_16 = %s.\n",BN_bn2hex(x));
  printf("# bits in 52! = %d.\n",BN_num_bits(x));
  BN_clear_free(x);BN_clear_free(y);
  BN_CTX_free(thectx);
  return 0;
}
```