A Practical Secure Physical Random Bit Generator

Markus Jakobsson* Elizabeth Shriver* Bruce K. Hillyer* Ari Juels†

Abstract

We suggest a practical and economical way to generate random bits using a computer disk drive as a source of randomness. It requires no additional hardware (given a system with a disk), and no user involvement. As a concrete example of performance, on a Sun Ultra-1 with a Seagate Cheetah disk, it generates bits at a rate of either 5 bits per minute or 577 bits per minute depending on the physical phenomena that we use as a source of randomness. The generated bits are random by a theoretical argument, and also pass a severe battery of statistical tests.

1 Introduction

Randomness is a central aspect of cryptography. It is paramount for key generation, is necessary in several encryption algorithms and in interactive proofs, and is useful for boosting the efficiency of algorithms. It is the pillar on which anonymity rests, and protocol soundness often requires a source of random bits.

Consequently, randomness is a research topic that has been given considerable attention. It has been proven that if a one-way function exists, then, given a random seed, it is possible to generate more randomness (a polynomial amount in the length of the seed; see [14] for a good overview) [10, 11, 9]. A function that amplifies randomness in this manner is called a pseudo-random generator. It is known that if the underlying hardness assumption holds (i.e., that a particular function cannot be inverted in polynomial time in the length of its security parameter) then it is impossible to predict the next bit to be output by the pseudo-random generator with a probability non-negligibly better than 1/2.

However, this guarantee only holds if the seed is unknown to the adversary, and so, we *need* randomness (in the form of the short seed) in order to *produce* randomness. In commercial cryptographic packages, this seed is oftentimes supplied by the user. One approach (used in RSA's toolkit) is to base the seed on keyboard and mouse timings during a period of several minutes during which the user "bangs the keyboard" [19]. This supposedly produces a random seed, but is rather inconvenient to the user. More automated methods have been suggested to avoid this shortcoming. Such methods are based on physical phenomena that in themselves have a large portion of unpredictability. One such method uses the time between observed emissions from a radioactive material [8], another measures the frequency instability of an oscillator [6, 2], and additional techniques derive randomness from quantum mechanical effects in semiconductor devices [1, 18]. Randomness has even been obtained from the blobs in a lava lamp [4]. These techniques require the introduction of new devices (e.g., radioactive material or noisy transistors, and the measuring devices to observe these). This introduces new costs, and also a new potential weakness: the output will not be random if the source or measuring device stops working as it is meant to. A number of random number generators are based on computer clocks (e.g., Truerand [13]), but these generators do not have guarantees. Another potential source of randomness is the unpredictable behavior of the stock market. Although this source avoids the introduction of auxiliary devices and is well-studied, it has several significant weaknesses: the market is sometimes predictable (e.g., during a crash); the market can be manipulated (by a large stock transaction, or by spreading rumors); and finally, its behavior is never secret.

A desirable solution would combine the positive aspects of all the above solutions while avoiding their shortcomings. We want a local source of randomness that does not require user involvement, and is not predictable or manipulable. To limit the costs and to avoid new vulnerabilities, the method for obtaining the random bits should not require any new equipment or any modifications to a computer's operating system. We also want it to be obvious to the user whether the generator is functioning correctly. Additionally, we want to base our random bit generator on a well-studied physical phenomenon, so that we can establish an assurance of the level of randomness of our source. A final important aspect is that of bandwidth—the rate at which bits are produced.

We propose a general method to derive randomness from measurements of access times of storage devices such as tape drives, CD-ROMs, and disk drives, as well as timings of other computer phenomena that exhibit a known degree of unpredictability. We focus on timings of magnetic hard disks for the rest of this paper. We describe how to use variations in disk drive response times as a source of randomness. This source appears to have all of the advantages discussed above, and need not suffer any of the disadvantages. Disks are lo-

*Information Sciences Research Center, Bell Laboratories. {markusj,shriver,bruce}@research.bell-labs.com

†RSA Laboratories, RSA Data Security. E-mail: ari@rsa.com

cally available—already part of most systems. It is obvious to the user whether the disk drive is operational or not. Moreover, the I/O response time has been studied extensively [23, 24, 7].

A way to derive random bits from measured variations in the rotational latency of a disk is presented in [3], but that approach modifies the kernel of the operating system to perform the measurements. Our approach applies to additional devices, requires no operating system modifications, and is accompanied by theoretical guarantees on the strength of the generated bits.

Our solution consists of software that measures the response time for certain I/O requests, and derives the random bits from these measurements. Our algorithm determines an appropriate sequence of disk addresses to access, measures the amount of available Shannon entropy, and calculates how many measurements are needed per output bit to meet a specified error rate given the available entropy. Then the algorithm repeatedly measures the access times for the determined disk addresses, and extracts random bits from these measurements. One way to extract random bits from the readings, which we describe in Section 3, uses the method of randomness purification first proposed by Santha and Vazirani [21], i.e., purifying randomness by the bitwise exclusive-or of biased but random bits.

We consider two different modes of operation for our generator. The first mode, which we dub the "paranoid" mode, generates bits having randomness that is ensured by variations in the speed of rotation of a disk drive. These variations are a consequence of chaotic air turbulence, therefore, this source is guaranteed not to be predictable [3]. In Section 3.1 we describe how to measure the amount of entropy that we can safely attribute to rotational latency, and we give our evidence that the entropy in this measurement is indeed *caused* by the strongly random fluctuations in the rotational latency.

We define the failure rate of our generator to be the maximum probability of success for an adversary who is trying to guess the next output bit. Given the measured value of strongly random entropy, we show how to derive the number of readings required to produce each output bit, in order to achieve a specified maximum failure rate guarantee. For our specific computer configuration, our paranoid-mode generator will limit the adversary to a maximum probability of 2^{-80} in guessing a bit when we derive each output bit from 1494 readings. (This gives a bit-generation rate of approximately 5 bits per minute.)

The second mode of our generator, our "utility" mode, derives randomness from the variance of disk access time through all the hardware and software components in the I/O path. Although many of these components are deterministic from an epistemological standpoint, we believe that the complexity of the components and their interactions renders them effectively non-deterministic. From a practical standpoint, a completely accurate timing model of the system is intractable. Utility mode obtains a higher rate of bit production by assuming that all the calculated entropy in the timing measurements is the result of noise that is effectively unpredictable. To obtain a failure rate below 2^{-80} in utility mode, calculated as per Section 3.1 for the parameters of our hardware, our algorithm requires 13 readings per output bit, and thus the output rate is about 577 bits per minute. Standard statistical tests [15] are unable to distinguish the output of our utility mode from a truly random sequence. But because much of the entropy used to produce output bits in utility mode can not be attributed to inherently unknowable sources, our utility bits are not known to be cryptographically strong. Only in the paranoid mode do we have a theoretical argument that the generated bits are truly unpredictable.

The structure of this paper is as follows. In Section 2, we present a very high-level description of magnetic disks, and how these can be used as a source of randomness. In Section 3, we present our algorithm for randomness extraction, and in Section 4 we describe the statistical testing that indicates that our utility bits "look random", even though they are not provably random. Finally, in Section 5, we present conclusions and discuss future work. A brief review of the technical fundamentals of disk I/O appears in Appendix A.

2 Magnetic disks as a random source

We use the variations in the response time of raw read requests for one disk sector to a hard disk as the source of randomness for our generator. (See Appendix A for a brief overview of disk technology and I/O response times.) For the quantitative results in this paper, we use a Seagate Cheetah ST-34501W disk connected to a Sun Ultra-1 computer running Solaris 2.6. Our measurements use the high-resolution timing capability that is generally available in workstations and PCs. In Solaris, this is provided by the `gethrtime()` library function, which has a 0.6 microsecond resolution on our Ultra-1.

The performance of disk systems has been studied extensively. The methods currently used to predict disk performance are analytic modeling [23] and simulation [24, 7]. Both methods are ultimately parameterized by measurements obtained from real disks. Using either of these methods, the time needed to read a block from a local disk drive is an event that currently cannot be predicted with great accuracy. This is because of the complexity of the operating system and of the I/O path. The best models have errors of a few percent in predicting the *mean* response time, and the response times are known to have a large variance, even for repetitive measurements of the same pattern of reads on the same computer and disk drive. (For example, the performance model of [23] is only able to predict the mean response time for disk requests that include a full rotational latency, corresponding to the measurements in column 2 of Table 1, to within a relative error of 3.4%.)

We believe, from the evidence given in [3], that the rotation speed of a disk drive exhibits inherently unpredictable fluctuations due to the mathematically chaotic properties of turbulence in the air flow over the platters. In [3], the amount of randomness is not quantified, so it is not clear whether their method attempts to extract more randomness than is available. In addition, that paper lacks evidence to substantiate the crucial but implicit assumption that the fluctuating disk response time measurements are *caused* by the strong randomness in the disk rotational latency. Their assertion of random output may not hold if the timing variations that they measure are caused by other (not strongly-random) phenomena in the computer system, such as interrupts, cache misses, and memory cycle stealing by device controllers. Perhaps the variations in rotational latency are entirely suppressed by clock-driven regularities in the I/O path, and the timing variations consist entirely of "noise" not known to be strongly random.[1]

We present experimental evidence in Section 2.2 that the fluctuating speed of disk rotation produces a quantifiable

[1] We are indebted to Carl Ellison for pointing out the importance of emphasizing the evidence that bears on this causal link.

amount of randomness in the response time of a read request. Since we wish to incorporate this source of strong randomness in our generator, we ensure that the read operations used for bit generation incur the latency of (nearly) a full rotation of the disk.[2] To obtain this property, we identify pairs of disk addresses (a and b) such that reading a "initializes" the position of the disk head and platter in such a way that if b is requested immediately after a request for a completes, the response time for the read of b will include nearly a full rotation. We now discuss how to find such (a, b) pairs.

2.1 Finding suitable disk addresses

We want the response time for each address to contain a full rotation, so that the response time includes the entropy derived from the rotational latency of the disk. We can find addresses that incur full rotational latency as follows. Consider the scenario in which we read a block from address a, then seek the disk head to a different track on the disk, and immediately request a read from address b on that track. While the disk head is moving, the disk is rotating. When the disk head arrives at the destination track, where is b relative to the disk head? In one extreme case, b is just rotating under the disk head, so that it can be read immediately. In this case, the response time will consist of overhead plus seek time but no rotational latency. In the other extreme case, b has rotated just past the disk head as the head arrives at the destination track, causing the response time to consist of overhead plus seek time plus a full rotational latency. So given an address a, we search for a pair of addresses b and $b + \delta$ such that the response time for address $b + \delta$ is less than the response time for b by about the time of one disk rotation. To find such a b, we start with an initial value that equals a plus twice the average capacity of a disk cylinder (obtained from the disk manufacturer's specification sheet[3]). We then perform a linear search in which we increment b by moderate amounts (say, 20% of the size of a disk track as obtained from the disk drive's specification sheet), observing a monotonic increase in timings,[4] until we encounter a pair b' and b'' such that the time for b'' is less than the time for b' by a large fraction of the rotational latency time (also obtained from the drive's specification sheet). The pair of destinations b' and b'' bracket the desired b that maximizes the rotational latency. We find that b by binary search.

In practice, if our generator were to use only a pair of addresses a and b, our generator would spend half of its time on "initialization" by requesting a.[5] To reduce the overhead, we use a sequence of addresses, not just a pair of addresses. This sequence is determined one step at a time, as above, so the first address initializes the position of the disk head and platters, and each subsequent address incurs nearly a full rotational latency when the requests are issued one at a time in rapid succession. We denote the list of requested addresses by $r_1, \ldots, r_n$. We take high-resolution time measurements immediately before issuing each read request and immediately after that request is completed. The difference is the timing that we use as a source of randomness. So, from a list of addresses $r_1, \ldots, r_n$ we obtain a list of timings. We iterate the timing run repeatedly (typically 100 iterations), and thereby obtain 100 timings for each address $r_2, \ldots, r_n$.

[2] This maximizes the amount of strongly unpredictable rotational latency incorporated in each reading.

[3] Most of the disk vendors have detailed web pages that publish values of disk parameters such as the average number of bytes per cylinder and the rotations per minute.

[4] A timing that is inconsistent with the expected smooth increase is re-measured in case it reflects a delay from other system activity. If repeated measurements continue to show an unreasonable value, perhaps because of a remapped bad block, we discard that address and resume the search using a different nearby address.

[5] Section 3.1 reveals that it also is necessary to choose the number of addresses to be larger than the number of disk cache segments so that the disk mechanism is forced to perform a mechanical read operation for every request.

2.2 Entropy in response time

If we are to use the response time of disk reads to generate random bits, we must have confidence that the response times have a random component. In this section we first describe how to quantify the amount of randomness that is available for extraction from timing measurements collected as per the description in the previous section. Then we develop a sequence of arguments and experimental results that forge a causal link from the strongly-random "chaotic air turbulence" described by [3] to the variations in timing measurements actually obtained by our procedures.

We can quantify the amount of randomness by calculating a (lower-bound) estimate of the entropy of the source. We use different computations to ascertain the available entropy for paranoid and utility modes, but both methods use the standard definition of entropy:

$$E = -\sum_k \mathsf{f}(k) \log_2 \mathsf{f}(k) \qquad (1)$$

where $\mathsf{f}(k)$ represents the frequency of a value k.

When we take 100 high-resolution timings for an address r_i, we obtain a list of 100 values, all close to the mean, but differing in the low-order bits (the microseconds and nanoseconds). In practice, this list of readings almost never contains duplicate values. Blindly applying equation 1 to a list of n unique values yields $E = \log_2 n$; a function of the number of readings, independent of the behavior of the disk. This is not what we want. We need a way to round the timings to eliminate nanosecond jitter while retaining meaningful fluctuations in the disk latency. As we vary the quantization of rounding, the value of entropy ranges from 0 (all items rounded into the same bucket) to $\log_2 n$ (n unique values). We use the following technique to determine an intermediate quantization for rounding that gives a useful value of entropy.

Recall that in picking our a, b pairs, we find pairs of destinations b and $b + \delta$ such that the timings for the b contain a full rotational latency, whereas the timings for the $b + \delta$ do not. Given a quantization, we can calculate the entropy for the timings of b and also for the timings of $b + \delta$. The difference of these entropies is a function of the quantization. Intuitively, if all the timings fall into one bucket only (which is a very drastic rounding-off), there is only one outcome of the experiment, and the entropy of the experiment is zero, hence the difference of the entropies is zero. Similarly, if the rounding is so mild that each timing is assigned to a distinct bucket, the entropy of the experiment is $\log_2 n$ (because of n unique values), so the difference between the entropies of the two experiments is again zero. However, choosing an intermediate bucket size gives a positive difference of entropies. We search for the quantization that maximizes the difference in entropy for the b and $b + \delta$, because this quantization erases the confounding effects of the timing mechanism, while revealing the entropy that results from the difference between no rotational latency and full rotational latency.

Table 1: Mean response time and computed entropy of read times.

destination address	*mean response time*	*total entropy*	*rotational latency entropy*
802816	7.50 ms	3.17	0.16
808448	2.15 ms	3.01	-
1080329	7.61 ms	3.08	0.11
1081344	1.76 ms	2.97	-
105387008	9.50 ms	3.17	0.13
105401344	4.43 ms	3.04	-
1048622592	12.95 ms	3.13	0.13
1048623104	7.16 ms	3.00	-
1050628096	12.99 ms	3.10	0.10
1050628608	7.03 ms	3.00	-

Table 1 shows five pairs of b and $b+\delta$.[6] Column 3 shows the entropy for each b and $b+\delta$, computed using equation 1 on timings rounded as above. Column 4 shows the difference. We attribute the difference between these entropies to rotational latency, and we offer the following arguments in support of the claim that this difference in entropy is a *consequence* of rotational latency, and not a result of other activity in the computer system.

First, we observe that the entropy values are consistently higher for b than for $b+\delta$ (i.e., destinations that incorporate nearly a full rotation have higher entropy than destinations that have minimal rotational latency). We observed this in numerous measurements; five examples appear in Table 1. In our system, the destinations that include a full rotational latency have an average of 0.13 greater entropy.

The response time for b, which contains a full rotational latency, is larger than the response time for $b+\delta$, which has no rotational latency. One could hypothesize that the larger response time for b leads to higher entropy because of noise from events such as processor interrupts and other system activity not known to be strongly random. But the evidence does not support that hypothesis: our second observation is that the increased entropy is not increasing as a function of seek distance or total response time. The examples in Table 1 are typical of our measurements: in the five pairs of b and $b+\delta$, the response times for b vary by nearly a factor of 2, and by a factor of 4 for the $b+\delta$, yet the absolute entropy values and the differences in entropy change little. In particular, observe in Table 1 the four measurements that have total response times near 7 milliseconds. Two of these measurements have short seeks and nearly a full rotational latency, and the other two measurements have long seeks with almost no rotational latency. All four of these measurements have the same behavior from the point of view of the computer: a request is sent to the disk, and after a 7 millisecond delay, the disk responds with one sector of data. These I/Os all have the same opportunity for the computer system to introduce noise into the calculation of entropy, but the requests that incur a full rotational latency consistently have the higher value of entropy. This observation holds true over many other b, $b+\delta$ pairs.

We believe that the experiments described above give good evidence for a causal link from variations in rotational latency to variations in the response time of read requests. We now describe an additional experiment that provides further evidence of this causal link. We correlate the speed of disk rotation during a read request with the response time of the disk request, in the following way. We connect a high-speed measuring instrument (a logic analyzer) to the "index pulse" test point on the disk drive's circuit board. A voltage appears on that test point each time the disk platters reach angular position zero. By measuring the time interval between adjacent index pulses, we can directly observe the speed of disk rotation. We find that when the disk is idle, the speed of disk rotation is constant, with about 5.980 milliseconds between index pulses. When the disk is servicing requests, we observe fluctuations in the time between index pulses. We conclude that disk arm movement disturbs the smooth airflow inside the disk drive, inducing the chaotic turbulence described by [3]. When the disk arm becomes idle, the intervals usually oscillate above and below the value of 5.980 on successive rotations, until the drive can dampen out the speed fluctuations and return to its resting rate. A complicating factor is that when the disk is very busy, some index pulses are not generated—we observe times between index pulses of about 12 or 18 or 24 milliseconds.

The experiment is as follows. We start the recording of index pulse intervals on the logic analyzer. We then cause the computer to issue a read at address a to initialize the position of the disk head and platters, then to sleep 12 milliseconds[7] and then to issue a read at address b, which is chosen to incur (almost) a full rotational latency and a very short seek. Since we have measured the response time of requests a and b, and we know the duration of the sleep between them, we can calculate the number of disk rotations (at the average rate) from the start of servicing a until the start of servicing b. The next rotation occurs during the servicing of b. We then examine the data collected from the logic analyzer. We observe values of 5.980 (preceding a). We say that the first disturbed index pulse timing corresponds with the start of a, and from this we identify the measurement m_1 that should be prior to and partially overlapping the rotational latency of b, and the very next rotation m_2, during which the rotational latency of b completes. Note that the request for a is an asynchronous event with respect to the angular position of the disk: the application that requests a and b is started manually.

Operating and extracting data from the logic analyzer is manually intensive, so we have only been able to obtain 71 trials of the experiment. Given a list of 71 read response times for b, a list of the corresponding m_1 measurements, and a list of the corresponding m_2 measurements, we calculated correlation coefficients. Because of the small number of trials, the results are only suggestive, but the results are consistent with our claim of a causal link between the rotational latency and the response time of a disk read. In particular, the correlation coefficient between b and m_1 is approximately -0.24, and the correlation coefficient between b and m_2 is approximately 0.27. The change of sign between adjacent disk rotations is consistent with our observation that the disk usually oscillates between "too fast" and "too slow" on successive rotations as the disk controller attempts to drive the speed of rotation to the nominal value.

[6] The value of $a = 524288$ was used for this experiment, and each measurement was repeated 100 times. The "bucket size" of 3.8856 milliseconds was used to compute the entropy.

[7] Some experiments used a sleep of 18 ms. This delay of 2 or 3 rotation times enables the disk to stabilize to some extent, reducing the number of missed index pulses. Sometimes, because of other system activity, the operating system takes longer than the requested 12 or 18 milliseconds to reawaken our process. We discard all experimental trials that encounter such delays.

3 Algorithm

This section describes the technique that we use to extract random bits from the measurements of disk access times. The only difference between the paranoid mode and the utility mode is the number of readings that are required to produce one output bit.

Not all of the total measured entropy is guaranteed to correspond to actual unpredictable bits: what looks random to us may not look random to a skilled adversary. This is the reason that, in the paranoid mode, we determine the number of readings per output bit using only the entropy attributed to rotational latency. Because this entropy is a consequence of chaotic air turbulence, the output bits in paranoid mode are *a priori* unpredictable by a strong adversary.

In utility mode, we bound the amount of randomness available for extraction by the full entropy of the response time of request b, although the actual algorithm in Section 3.1 uses a direct derivation of bias from the measurements. For utility mode we could, instead, derive randomness from the response time of a faster operation, such as a cache hit in the disk controller, or the response to a `TEST UNIT READY` command. This would produce bits at a faster rate (preliminary tests suggest 2100 bits per minute), but we would not be able to attribute any of the entropy to an inherently unpredictable source such as chaotic variation in rotational latency.

Our technique has 3 major steps. The *calibration* step measures properties of the particular computer and disk drive that will serve as a source of randomness, to select a set of disk addresses to use as targets of read commands, and to calculate a safe bound on the available entropy. The *query* step performs a large number of read operations, to obtain the response times from which the random bits will be derived. The *filter* step applies a function to transform the sequence of response times into a sequence of random bits. We now describe each of these steps in detail.

3.1 Calibration

The calibration of the computer and disk system has 3 stages. First, we determine a good sequence of disk addresses to use for timing measurements. Then we calculate the amount of entropy per disk timing that we can safely extract, and finally we derive how we will group raw response times during the filter stage to make good use of the available entropy.

Determining the sequence of disk addresses. Our generator measures the response times for single-sector reads from a sequence $r_1, \ldots, r_n$ of disk addresses. To force the disk mechanism to perform a physical read from the disk surface for each request, the value of n must be at least 2, and must be larger than the number of sequential streams that can be held in the disk drive's cache (i.e., larger than the number of *cache segments*). It is easy to determine the number of cache segments experimentally [25], or (in many cases) by reading the disk manufacturer's specification sheet. Some disks support a command that sets the maximum number of cache segments, in which case we set it to 1.

We want the response time for each address to contain a full rotation, so that the response time includes the entropy derived from the rotational latency of the disk. We find addresses that incur full rotational latency using the method discussed in Section 2.1.

We choose an arbitrary address r_1 as a starting point, and using this as a, search for a good b relative to a, i.e., a value b incurring a large rotational latency. This b is r_2. We iterate this process to find a good r_i relative to each r_{i-1}.

Calculating the available entropy. We discussed how to calculate entropy in Section 2.2. In summary, for paranoid mode we measure the response times of single-sector disk reads from a sequence of addresses $r_1, \ldots, r_n$. We denote these response times $t_1, \ldots, t_{n-1}$, where t_i is the response time for a read to address r_{i+1} immediately after a read to address r_i. We round the t_i's to obtain q_i's, and calculate the entropies E_{r_i} from these q_i. In paranoid mode we only use the entropy attributable to rotational latency. This value is obtained by finding the entropies $E_{r_i+\delta}$, where $r_i + \delta$ is the destination near r_i that had no rotational latency, and then by setting the available entropy $E_i = E_{r_i} - E_{r_i+\delta}$. (Typical values of E_i for our hardware configuration appear in column 4 of Table 1.)

In the utility mode, we use a value of entropy that may be greater than the entropy of the rotational latency used in paranoid mode, but that is definitely less than the full measured entropy, E_{r_i}. (Column 3 of Table 1 gives typical values of the full measured entropy on our hardware.) The particular value of entropy that we use for utility mode is a consequence of the grouping determined in the next section.

Now that we have determined the amount of entropy available for extraction, we describe how to calculate the number of readings that will be required to generate each output bit. First we discuss paranoid mode, then utility mode.

Determining grouping. Given the lower bound on entropy attributable to rotational latency, the final calibration step determines k, the number of readings required to compute one output bit.

The entropy of a bit is calculated as

$$E = -p \log_2 p - (1-p) \log_2 (1-p) \qquad (2)$$

where p is the probability of the bit being 0. If a bit contains less than one bit of entropy, this corresponds to a bias: one of the outcomes is more likely. We first compute the bit probabilities that correspond with the measured entropies, (e.g., the entropies illustrated in Table 1). We then determine how many of these biased bits need to be exclusive-ored together to drive the bias of the resulting bit below any specified threshold, by extending the technique introduced by Santha and Vazirani [21].

Consider two readings, each consisting of a single bit that takes the value zero with probabilities $p_1 = 1/2 + \epsilon_1$ and $p_2 = 1/2 + \epsilon_2$, respectively. (More generally, $p_i = 1/2 + \epsilon_i$.) The result is zero with probability $p_1 p_2 + (1-p_1)(1-p_2) = 1/2 + 2\epsilon_1 \epsilon_2$. Generalizing this result, we can compute the bias after exclusive-oring k such bits (for an even k) as

$$\epsilon' = 2^{k-1} \prod_{i=1}^{k} \epsilon_i.$$

Since $\epsilon_i < 1/2$ for all i, this expression goes to zero as k increases. Given an entropy computed over a sequence of 1000 readings, we substitute ϵ (the average value of ϵ_i) into the previous expression, and get

$$\epsilon' = 2^{k-1} \epsilon^k. \qquad (3)$$

To solve equation 3 for k with a maximum failure rate of 2^{-80}, we set ϵ' to 2^{-80}, rearrange terms, and take logs, getting

$$k = \frac{-79}{1 + \log_2 \epsilon}. \quad (4)$$

Let us consider an example of how to determine parameters in paranoid mode. For this example we choose $n = 2$ (i.e., we use a sequence of 2 addresses). Looking at Table 1, we choose addresses $r_1 = 524288$ and $r_2 = 105387008$ since this pair gives us a large rotational entropy. From the 4th column of Table 1, we see that $E = 0.13$. Using equation 2, we can approximate p as 0.982, hence $\epsilon = 0.482$ (since $\epsilon = p - 1/2$). Substituting this value of ϵ into equation 4, we get $k = 1494$. This derivation shows that if 1494 readings are "collapsed" to a single bit, and the entropy per reading is 0.13, then the output will have a bias that is upper-bounded by 2^{-80}. Note that our method does not require any knowledge of which specific bits in a reading are the ones that contain the entropy attributable to rotational latency.

In the utility mode, we use entropy derived from additional elements of the computer I/O architecture, and thereby obtain significantly faster bit generation, although these bits are not known to be strongly random. Our measurements show a total entropy of approximately 3 bits of entropy per reading (see the 3rd column in Table 1), but this entropy is not concentrated in three particular bits of the timing measurement. The entropy may be spread over all the bits of the response time. Also, the bits in the timing measurement are correlated to each other, and so cannot be treated as independent readings. For utility mode, we find it simplest to determine k directly from the the bias ϵ, as follows. First, we exclusive-or all the bits of a reading to obtain a single bit *reduced reading*. Then we count the number of 0 bits and 1 bits in the set of reduced readings to determine the bit probabilities (p is the ratio of 0 bits to total bits). This leads directly to k from equation 4 (since $p = 1/2 + \epsilon$). We computed $\epsilon = 0.007$ for 1000 readings using addresses similar to row 5 in Table 1. From equation 4 we get $k = 12.83$, which we round to 13. Using equation 2, we see that this value of p corresponds with an entropy of about 1, which is conservative by comparison with the 3 bits of total entropy seen in the 3rd column of Table 1.

We note that the value k determined by the above technique *guarantees* the randomness of the output, assuming that the value of entropy used here is less than the true entropy of the source, since k puts an upper-bound on the bias of the output bits. Implicitly, a proper choice of k therefore corresponds to a proof that the output of the generator is *random*, given a sufficiently good estimation of the entropy.

3.2 Query

After the three calibration steps are completed, we collect the timings from which random bits will be generated. We repeatedly issue the sequence $\{r_i\}$ of single-sector read requests determined during calibration, to obtain corresponding response times t_i. Because the number of r_i is greater than the number of cache segments, the disk is forced to serve all of the r_i from the disk surface (after the first time through the sequence of r_i, which we discard).

3.3 Filter

We extract random bits from the timings t_i by a *filtering function*. One common way to do this is to compute a cryptographic hash on a batch of input data (e.g., the timings obtained in the query step), such that the aggregate entropy of the input bits is large enough to imply the security of the output bits.

In this paper, we use a simple and computationally efficient technique that has very strong theoretical underpinnings.[8] We group the t_i into contiguous batches of size k, and reduce each batch to a single bit by the exclusive-or of all the bits in all the t_i in the batch. k is the value calculated from the entropy during calibration, namely 1494 for paranoid mode, and 13 in utility mode for our particular hardware.

4 Validation

Physical analysis is one way to demonstrate the integrity, i.e., the truly random nature, of a random source. Physical analysis of a random source involves modeling based on knowledge of underlying physical properties. For example, if a random number generator makes use of a radioactive source, then results from experimental and theoretical physics regarding radioactive decay offer evidence of the integrity of the generator. We presented a physical analysis of magnetic disks in Section 2. The operational characteristics of magnetic disks, as we have explained, are evidence in support of the integrity of this generator.

It is also possible to test a random source empirically. Empirical testing involves the application of tests to an output sequence S derived from the source, comparing the results of these tests to the results likely to be obtained from a source that is truly random. Although statistical testing can not prove that a source is random, it is prudent to validate the output of a source that is asserted to be random according to a physical argument. A necessary (not sufficient) condition for a good random generator is that the output "looks random" to a battery of empirical tests.

In this section we describe empirical techniques for testing the randomness of bits, and present the results of a particular battery of tests applied to the output of our generator in utility mode (i.e., $k = 13$, and also in a weaker mode, namely $k = 2$).

4.1 Statistical testing

Empirical randomness tests may be roughly divided into three categories: traditional statistical tests, complexity-based tests, and spectral tests. Traditional statistical tests are simple, ad hoc tests of randomness, such as the well known examples in [12]. In a traditional statistical test, a collection of random variables $x_1, \ldots, x_n$ is derived from the sequence S such that the distribution of the $\{x_i\}$ is easily computed for a truly random sequence. (For example, the $\{x_i\}$ might be independent and identically distributed over truly random sequences S.) These $\{x_i\}$ then form the basis of a chi-square test. The Runs Test given by Knuth is a well known example of a traditional statistical test. In one version of this test, the lengths of monotonically increasing sub-sequences of S are tabulated for evaluation by a chi-squared test.

Complexity-based tests are statistical tests based on theoretical characterizations of the complexity of a random sequence. An example of such a test is the Universal Statistical Test [17, 16], which is based on characterizations of the per-bit entropy of a source. Complexity-based tests are

[8] In another paper, we describe a filtering function that asymptotically extracts all the entropy available in the input, and that has very good performance in practice.

similar in structure to traditional statistical tests, but are lengthy and complicated. Spectral tests involve analysis of Fourier or Walsh transforms of the sequence S or on the autocorrelation function of S. Spectral tests are generally not applied as statistical tests, but are used to make qualitative assessments of purportedly random sequences.

No single empirical test offers strong evidence, *per se*, of the random nature of a source. Even linear congruential generators, which for cryptographic purposes were broken two decades ago, will pass most empirical statistical tests, as well as a range of spectral tests [22]. Thus, effective testing requires the use of a battery of empirical tests. Even then, empirical testing can only be regarded as a kind of sanity check: the fact that a random source passes a battery of empirical tests is evidence of its integrity, but not a guarantee. To guarantee the strength of a random generator requires techniques such as physical analysis, together with evidence that the physical analysis describes phenomena actually measured by the practical implementation. A good overview of empirical tests of randomness may be found in [5].

For empirical testing of our random generator, we use a software package called Diehard [15] that has been designed for this purpose. Diehard includes a battery of about 200 instances of fifteen traditional statistical tests. Although sophisticated pseudo-random generators successfully pass the test, the author of Diehard asserts in 1996 that, to the best of his knowledge, no physical random source has ever passed the full battery of Diehard tests.

4.2 The results

The standard input file to diehard contains 91,750,400 bits. We generated a file of this size using our algorithm with $k = 2$. (This gives weaker guarantees than our utility and paranoid modes, but increases the rate of bit production for testing.) These bits passed the full suite of Diehard tests. We also generated utility-mode bits ($k = 13$) for testing, but only generated 30,384,320 of these because of time limitations. To test these bits with diehard, we overwrote these bits into the leading portion of the $k = 2$ file. This hybrid file (30 million $k = 13$ bits and 62 million $k = 2$ bits) also passed the full suite of diehard tests. See `http://www.bell-labs.com/~shriver/random` for the results of the Diehard tests.) Bits produced by our generator in utility and paranoid mode are generated by exactly the same mechanism, except that paranoid mode exclusive-ors a larger number of readings together. Since equation 3 shows that increasing values of k lead to tighter bounds on bias in the output, we expect that bits generated in paranoid mode ($k = 1494$) would also pass the Diehard tests.

In generating our bits, we used $n = 20$ disk addresses. The average reading time was 8 ms, which gives us 577 utility bits per minute. We may be able to greatly increase the generation rate for utility bits by using disk commands that are faster than `read`. For instance, preliminary experiments show that the SCSI disk command called `TEST UNIT READY` can be used with our technique to generate about 2100 utility bits per minute. A similar improvement will not be available in paranoid mode until a physical argument can establish the strongly random nature of some fast disk command.

5 Conclusions

We have presented a practical method to generate strong random bits. The randomness is derived from the entropy present in measurements of the disk I/O response time in a computer. Our method does not require any addition or changes of hardware, or any modifications of the computer operating system.

The generator has two modes. In one mode, random bits are generated from a source with known chaotic behavior, and therefore these bits are known not to be predictable. In the second mode, bits are generated from a source that may be partly deterministic from an epistemological point of view, but that, in practice, is acknowledged to be predictable only to a limited degree. The first mode is appropriate for the generation of secret keys, seeds for pseudo-random generators, or wherever we require that the bit string be perfectly random against the strongest possible adversary. The second mode is appropriate for applications that require random bits in the absence of a strong adversary, for settings where the power of the adversary is known not to be sufficient, or where the cost to attack the system with a reasonable success probability greatly exceeds the potential gain of a successful attack.

The contributions of this paper are as follows. We suggest a practical and useful setting for random bit generation, requiring a minimum of changes and user involvement. We specify design goals giving an inexpensive and practical product that can easily be integrated into existing cryptographic packages, and that uses only the hardware present in a typical workstation or PC. We develop methods to detect and measure the entropy of an inherently unpredictable origin given a source with a much larger amount of total entropy. In our case, the strong entropy is derived from perturbations in the rotational speed of a disk drive caused by air turbulence over the disk platters, and the remainder comes from the variance in disk access time attributable to other sources, such as the operating system, the I/O bus access, and the disk controller. Furthermore, we devise methods to decide, given the amount of available entropy and the required failure probability, how many measurements are needed to ensure the strength of each random bit generated. We combine these techniques to produce a random bit generator that achieves our previously stated design goals. In particular, the strength of the generator is established on theoretical grounds, and is confirmed by a well-known battery of statistical tests.

Directions for future work include the evaluation of a wide range of physical sources for available randomness; adaptations of our methods to fit other devices; and the specification of a strict adversarial model, followed by rigorous proofs that our random generator satisfies the requirements, given a minimum of assumptions on the random source, along with a careful analysis of random sources to establish that the stated assumptions are valid. We are especially interested in finding phenomena that support higher generation rates for strongly random bits via a larger product of readings per second $\times$ strong entropy per reading.

The efficiency and security—two opposing goals—both depend directly on the number of readings, k, that are used to produce one output bit. With a smaller value of k, efficiency increases, but below a certain threshold, security decreases (since output bits will become partially predictable to a sufficiently knowledgeable adversary). We believe that it is an important topic of future research to develop methods to determine the smallest value of k that produces the

desired level of security. Lacking such a method, we have been forced to be very conservative in our determination of k, thereby obtaining strong bits, but suffering a consequential impairment in the bit generation rate.

In our current calibration phase, we assume that all the variance in rotational latency corresponds to true entropy. Although we believe that this assumption models reality well, it may not be entirely true. For instance, the disk controller works to dampen out fluctuations in the speed of rotation, driving the disk speed towards the target value. It is possible that some of the acceleration applied by the disk controller may potentially be perceived as random behavior by our estimator of the value k. Developing a better understanding of the influence of such behavior on our estimate, and being able to counter the same, will improve our security guarantees. This may lead to a somewhat lower efficiency.

On the other hand, our estimation method gives a *lower bound* of the entropy of the source (based on the above assumption). This may be an unnecessarily crude lower bound. Developing more precise methods for determining the entropy due to rotational latency may enable lower choices of k to be made, leading to a more efficient random generator.

Finally, a better understanding of the distribution of the raw readings obtained from our physical randomness source may enable us to make significant efficiency improvements while maintaining the desired level of security, by supporting the design of more efficient methods for the extraction of randomness.

Acknowledgments. We wish to thank Andreas Jakobsson and Rafi Ostrovsky for inspiring discussions and useful feedback, Mario Suttora and Cliff Martin for helping set up the logic analyzer, Andres Tellez for collecting logic analyzer measurements, and Peter Fiandra of Seagate for technical information such as the identification of the index pulse test pin. Many thanks to Carl Ellison, the shepherd for this paper at CCS5.

References

[1] Agnew, G. B. Random sources for cryptographic systems. In *Advances in Cryptology – Eurocrypt '87*, D. Chaum and W. L. Price, Eds., Springer Verlag, pp. 77–81. Published in Lecture Notes in Computer Science v. 304, 1988.

[2] AT&T. T7001 random number generator. Data Sheet.

[3] Davis, D., Ihaka, R., and Fenstermacher, P. Cryptographic randomness from air turbulence in disk drives. In *Advances in Cryptology – CRYPTO '94* (Santa Barbara, CA), Springer Verlag, pp. 114–120. Published in Lecture Notes in Computer Science v. 839.

[4] Computing random numbers. Light headed. *The Economist* (31 May 1997), 74–75. See also `http://www.lavarand.sgi.com/`.

[5] Erdmann, E. D. Empirical tests of binary keystreams. Master's thesis, University of London, 1992.

[6] Fairfield, R. C., Mortenson, R. L., and Koulhart, K. B. An LSI random number generator (RNG). In *Advances in Cryptology – CRYPTO '84* (Santa Barbara, CA, 1984), pp. 203–230.

[7] Ganger, G. R. *System-oriented evaluation of I/O subsystem performance.* PhD thesis, University of Michigan, Ann Arbor, MI, June 1995.

[8] Gude, M. Concept for a high-performance random number generator based on physical random phenomena. *Frequenz 39* (1985), 187–190.

[9] Håstad, J. Pseudo-random generators under uniform assumptions. In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing* (Baltimore, MD, 14–16 May 1990), pp. 395–404.

[10] Håstad, J., Impagliazzo, R., Levin, L., and Luby, M. Pseudo-random generation based on one-way functions. To appear in SIAM Journal of Computing. A preliminary version appears in STOC, 1989.

[11] Impagliazzo, R., Levin, L. A., and Luby, M. Pseudo-random generation from one-way functions (extended abstract). In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing* (Seattle, WA, 15–17 May 1989), pp. 12–24.

[12] Knuth, D. *The Art of Computer Programming, Seminumerical Algorithms*, second ed. Addison Wesley, Reading, MA, 1981.

[13] Lacy, J. B., Mitchell, D. P., and Schell, W. M. Cryptolib: Cryptography in software. In *USENIX Security Symposium IV Proceedings* (Santa Clara, CA, October 1993), USENIX Association, pp. 1–17. See also `ftp://research.att.com/dist/mab/librand.shar`.

[14] Luby, M. *Pseudorandomness and Cryptographic Applications.* Princeton University Press, New Jersey, 1996.

[15] Marsalgia, G. Diehard. `http://stat.fsu.edu/~geo/diehard.html`.

[16] Maurer, U. A universal statistical test for random bit generators. *Journal of Cryptology* (1992), 89–105.

[17] Maurer, U. M. A universal statistical test for random bit generators. In *Advances in Cryptology – CRYPTO '90* (Santa Barbara, CA, 1990), Springer Verlag, pp. 409–420. Published in Lecture Notes in Computer Science v. 537.

[18] Richter, M. *Ein Rauschgenerator zur Gewinnung von Quasi-idealen Zufallszahlen für die Stochastische Simulation.* PhD thesis, Aachen University of Technology, 1992. In German.

[19] RSA Data Security, Inc. *RSA SecurPC for Windows 95 Users Manual*, 1997.

[20] Ruemmler, C., and Wilkes, J. An introduction to disk drive modeling. *IEEE Computer 27*, 3 (March 1994), 17–28.

[21] Santha, M., and Vazirani, U. V. Generating quasi-random sequences from slightly-random sources (extended abstract). In *25th Annual Symposium on Foundations of Computer Science* (Singer Island, FL, 24–26 Oct. 1984), IEEE, pp. 434–440.

[22] Schneier, B. *Applied Cryptography*, second ed. John Wiley & Sons, Inc., 1996.

[23] Shriver, E. *Performance modeling for realistic storage devices.* PhD thesis, New York University, New York, NY, May 1997. Available at `http://www.bell-labs.com/~shriver/`.

[24] WILKES, J. The Pantheon storage-system simulator. Tech. Rep. HPL-SSP–95–14, Storage Systems Program, Hewlett-Packard Laboratories, Palo Alto, CA, December 1995.

[25] WORTHINGTON, B. L., GANGER, G. R., PATT, Y. N., AND WILKES, J. On-line extraction of scsi disk drive parameters. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Ottawa, Canada, May 1995), ACM Press, pp. 146–56.

A Basic disk knowledge

See [23, 20] for additional disk information.

The access time for a read request to a local disk is the sum of queuing delays and service times in many physical components, such as the host device controller, the bus, the disk controller, and the disk mechanism. When a disk request is issued, it enters the operating system, which then hands it off to the host device controller. If the disk is busy, the request is put on a queue in the device controller; the queue is sorted by a scheduling algorithm that attempts to improve response times. One commonly-used class of scheduling algorithms are the *elevator* algorithms, where the requests are serviced in the order that they appear on the disk tracks. Once the request reaches the head of the queue, the request is sent to the bus controller which requests the bus. The request is then sent to the disk, and might be queued there if the disk mechanism is busy. This queue is also sorted to improve response time; one commonly-used scheduling algorithm is Shortest Positioning Time First, which services requests in an order intended to minimize the sum of the *seek time* (i.e., the time to move the head from the current track to the desired track) and the *rotational latency* (i.e., the time needed for the disk to rotate to the correct sector once the desired track is reached).

When the request reaches the head of the queue, the disk cache is checked to see if the data is in cache. If not, the disk mechanism moves the disk head to the desired track (seeking) and waits until the desired sector is under the head (rotational latency). The desired data is then read into the disk cache. The disk controller then contends for access to the bus, and transfers the data to the host from the disk cache at a rate determined by the speed of the bus controller and the bus itself. Once the host receives the data and copies it into the memory space of the application, the application process is awakened. At this time, is the read is said to be completed.

The disk cache (also called the *disk buffer*) is used for multiple purposes. One is as a pass-through speed-matching buffer between the disk mechanism and the bus. Most disks do not retain data in the cache after the data has been sent to the host. A second purpose is as prefetching buffer. Data can be prefetched into the disk cache to service future requests. Most frequently, this is done by the disk saving in a cache segment the data that comes after the requested data. Modern disks such as the Seagate Cheetah only prefetch data when the requested addresses suggest that a sequential access pattern is present. The disk cache is divided into *cache segments*. Each segment contains data prefetched from the disk for one sequential stream. The number of cache segments usually can be set on a per-disk basis; the typical range of allowable values is 1–16.