AN ABSTRACT OF THE THESIS OF

Adam W. Montville for the degree of Master of Science in Electrical and Computer Engineering presented on May 21, 2003.

Title: Random Number Generation on Handheld Devices for Cryptographic Applications

Abstract Approved:

Redacted for Privacy

Cetin K. Koc

Random number generation is important in many fields today. It is particularly important in the field of cryptography when generating nonce values, cryptographic keys, and other data required in many cryptographic applications. The proliferation of small, handheld devices that are typically connected to large networks via a wireless connection requires stringent security. Because it may be easier to attack a pseudorandom number generator than to attack a particular cryptosystem, it is important that the generation of random numbers on handheld devices be as secure as possible. In order for the random number generator to provide good, secure "random" data, it must first be seeded by a value that, itself, possesses the qualities of a good random sequence. This paper explores several potential seed sources that are available on many current handheld devices.

Random Number Generation on Handheld Devices for Cryptographic Applications

by

Adam W. Montville

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for
the degree of

Master of Science

Presented May 21, 2003
Commencement June 2004

Master of Science thesis of Adam W. Montville presented on May 21, 2003.

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical and Computer Engineering

Redacted for Privacy

Director of School of Electrical Engineering and Computer Science

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Adam W. Montville, Author

ACKNOWLEDGEMENTS

First, I thank God for providing me with the determination, perseverence, and will to complete this thesis. I also thank my Major Professor, Dr. Cetin K. Koc, for his guidance and helpful criticism throughout the research of this topic. In addition, this work could not have been completed, if it were not for my future wife, Virginia. Without her, none of this would have been possible. My gratitude is not complete without thanking my father, who always offered his assistance in proofing and mastering the English language. Finally, I thank Daniel Montville for loaning me hardware test equipment – it may have been old, but it worked!

To those who should have been named, but weren't, please forgive my forgetfulness and know that I appreciate *all* those who lent a helping hand to my research, whether directly or indirectly, including you.

TABLE OF CONTENTS

LIST OF FIGURES

Random Number Generation on Handheld Devices for Cryptographic Applications

1. Introduction

Today's cryptographic applications securing some of our most critical infrastructures and communications rely heavily upon random data. In most cases, they rely upon *pseudo*-random data.

Generation of "random" data can occur in one of two ways: naturally, or deterministically. The latter is the most common because most cryptosystems are employed on digital machines, i.e., computers. Deterministically generated random data is not typically referred to as "random", but as *pseudo-random* [7]. A generator producing pseudo-random data is referred to as a *Pseudo-Random Number Generator* (PRNG), whereas a generator producing natural random numbers is referred to as a *Random Number Generator* (RNG)[1]. Oddly enough, sequences generated by a PRNG often appear to be more random than those generated from naturally occurring sources [12].

A brief review of some cryptographic application specifications demonstrates the reliance upon random data:

[1] In the context of this paper, the term "random" will always refer to a natural or physical source of randomness, and the term "pseudo-random" will always refer to a deterministic source of randomness; in a similar manner, when a RNG is discussed, it will be in the context of natural or physical randomness, and when a PRNG is discussed, it will be in the context of a deterministic method of randomness generation.

Network Protocols: Some network protocols require what is known as a *nonce* (a number that is used once), and further require that such numbers are random; the *ClientHello* and *ServerHello* messages of the SSL/TLS network protocols require randomly generated data, which eventually become the keys used to encrypt the session [16].

Cryptosystems: Virtually all cryptosystems require random data for the purpose of key generation. This is true when examining public-key, private-key, or hybrid cryptosystems. The Advanced Encryption Standard (AES), Data Encryption Standard (DES), and RSA, among others, require randomly generated data [15], [13], [14], [10], [11], [17].

Authentication Systems: Some authentication systems require random data as well. An example of such a scheme is the Unix method of password storage, which may require a *salt* value to store a password in order to prevent identical passwords from appearing identical in the password storage file [4].

Dieter Gollmann, in his book, Computer Security, is concerned about the "layer below" when designing and evaluating computer security systems [4]. His basic concern is that the foundation upon which security measures are constructed should be at least as secure as the application being implemented. If the foundation is flawed, then the security built upon that foundation is flawed. The same reasoning can be applied to cryptographic applications and pseudo-random number generation.

The generation of random data is a layer below relative to a cryptosystem. For example, a PRNG responsible for generating AES keys is the layer below relative to the AES implementation. The PRNG, itself, has a "layer below" known as the *seed.* The seed is that value which is used to start pseudo-random number generation, and should be acquired from a random source. According to NIST, "all true randomness is confined to seed generation" [12]. As long as the seed is not known to an adversary, and the employed PRNG generates acceptable sequences, then the adversary is not likely to predict the output of the PRNG.

Therefore, the layer below for a given PRNG is the seeding of that PRNG. If the seed is somehow known, then the PRNG is predictable. Predictability occurs when an adversary can determine the generated numbers before they are generated. If the PRNG is predictable, then the key may be predictable. If the key is predictable, then the cryptographic application is insecure. Ultimately, the seed must be just as unpredictable as the output of the PRNG [12]. In order to get a seed that is unpredictable without requiring another seed, it is necessary to find some naturally occurring source of random information.

2. Background

2.1. Sources of Randomness

Randomness can come from many places. A person sitting on a park bench flipping a quarter can provide a source of randomness, though that may be (arguably) too slow for any cryptographic application requiring a random bit sequence. A very well known source of randomness comes from the rate of radioactive decay. A computer system running an operating system, and having some form of a user interface, may provide several sources of randomness [8]:

- Hardware-based (external) generators
  - Audio/video input
  - Disk drives
  - Thermal noise (or other hardware noise)
- Software-based (internal) generators
  - Keyboard/mouse movements
  - I/O buffer content
  - Operating system statistics

The first item under *software-based generators* is not, in the author's opinion, properly categorized. In effect, gathering information from the keyboard/mouse of a given system is gathering from an external source, rather than from an internal source. However, there are some caveats associated with such input, which is likely why it has been categorized in such a manner. A common approach to collecting random samples is to gather the timing differences between keystrokes and/or mouse movements. However, [2] has pointed out that many keyboard and mouse inputs are buffered and will therefore not yield proper timing results. However, mouse movements need not be timed. Rather, the coordinate system can be utilized to produce random values.

Using the system clock as a seed source for a PRNG is often a tactic that is used, but not one that should be used for any situation in which an adversary is interested in the value of the seed. As [2] points out, clocks and timers in computer systems vary widely in terms of resolution and the timing of the code execution may alter the perception of the true timing of the clock. A similar argument can be made against I/O buffer content, which may, under a denial of service attack, be well known, or easily guessable. Effectively, these methods of seeding a PRNG (which is deterministic) use deterministically generated seed values.

When considering PRNG seed sources, some of the random sources do not apply, for obvious reasons. Those that *do* apply are the hardware-based methods and the mouse-based method. The keyboard timing method cannot be guaranteed across platforms to be non-buffered, and the software-based methods (with the exception of coordinate-based mouse samples) are all deterministic in nature. This shortens the list of potential sources of randomness to

- Mouse movements
- Audio/video input
- Hardware noise

2.2. Handheld Devices

There are many handheld devices on the market today. Some are extremely specialized and used for rugged, outdoor activity, and others are designed for executives, knowledge workers, and others who need more than simple personal information management. Still other handheld devices are "simple" managers of information that keep track of dates, schedules, and the like. It can be argued, however, that there are two top handheld devices in the United States of America:

- Palm-based platforms
- Microsoft Pocket PC-based platforms

These handhelds are designed with a variety of microprocessors at their core. It appears to be the case, however, that the Palm-based platforms are primarily designed around Motorola DragonBall microprocessors, and that the Microsoft-based platforms are primarily designed around Intel microprocessors (including the SA11xx/SA111x series and the XScale processors)[2].

3. Materials and Methods

3.1. The Seed Goal

It was established in Section 2 that the true source of security for a given, accepted PRNG lies in the seeding of that PRNG. Because the focus is upon the seed, there are slightly different randomness requirements. Though the output of seed and pseudo-random number generators must be unpredictable, the seed generator need not be capable of generating very long random bit sequences. The sequences generated should pass as many randomness tests as possible, however, shorter sequences can be tested.

3.2. Random Bit Collection Source

The purpose of this research is to study some of the (potentially) random sources that may be available in a handheld device. Although there are several potential sources of randomness in handheld devices, only two were chosen for this study: *audio input*, and *touchscreen input*.

[2] A brief review of the datasheets/user-manuals for the DragonBall, XScale, and SA1110 microprocessors reveal that there are no RNGs made available to the systems designed around them [6] [5] [9].

3.3. Handheld Implementation Platform

There are many handheld devices available today, but one of the more popular platforms was chosen for this study – the Microsoft Pocket PC. The Pocket PC operating system runs on several different handheld devices, all with different configurations. However, all Pocket PC devices are required by Microsoft to meet certain specifications. As a result of this mandate, all Pocket PC devices can be expected to have a common denominator of hardware and functionality. Of particular importance is the fact that every Pocket PC device is required to have a source of audio input and a touchscreen as part of the user interface.

The particular Pocket PC device used in this study was a Dell Axim X5 with an Intel XScale microprocessor, 32MB of internal RAM, 32MB of Intel StrataFlash non-volatile memory, and running Pocket PC 2002.

3.3.1. Software Implementation

Several software components were required to complete this study. These components fall into two primary categories: *Pocket PC Software*, and *Test and Evaluation Software*. The Pocket PC Software was originally implemented for the sole purpose of data collection, and has now been slightly rewritten to form a cohesive Application Programming Interface that can be included in derivative works (see the Appendix for source listings).

3.3.1.1. Pocket PC Software

3.3.1.1.1. Touchscreen Data Collection

The touchscreen on a Pocket PC is intended to be the primary user interface on the device. The screen resolution of all Pocket PC devices is specified by Microsoft to be 240 x 320 pixels[3]. Touchscreen data can be collected as entropy by looking at the varying coordinates of the stylus upon the touchscreen as time passes. The implementation used for this study requires the user to scribble on the screen as randomly as they can, and then collects the points of the stylus on the touchscreen at various sample rates, wherein the software collects point data at intervals of approximately 5ms, 10ms, or 15ms.

The coordinates are passed as a single 32-bit value (a DWORD in Microsoft parlance), where the lower 16 bits represent the *x* coordinate and the higher 16 bits represent the *y* coordinate. Each 16-bit value is known as a WORD. The program responsible for collecting entropy represents the collected point data as the exclusive-or of the low half of the *x* WORD with the low half of the *y* WORD.

Let the collected point data be represented by *P*, then

$$P = (LO(coord) \otimes HI(coord)) \wedge 0x00FF.$$

[3] From this point forward, any reference to screen dimension will be in the unit of pixels.

The *coord* value in the previous equation is the value collected when either of the stylus-generated messages are received. The *HI* and *LO* functions return the high WORD and low WORD respectively, and their results are then XOR'd before being AND'ed with a mask designed to collect only the lower eight bits of the representation. Therefore, the point representation used in collecting data from the touchscreen in this study is an 8-bit representation rooted in the x and y coordinates of the original touchscreen event.

3.4. Audio Data Collection

The Pocket PC has a rich audio interface. The Pocket PC Wave API was used to prepare the audio input device, collect the audio data, and release the audio input device appropriately. This method of audio recording requires the use of data buffers. A buffer of bytes is first prepared then "registered" with the audio device. A function is invoked to start recording and will record continuously, thus filling the buffer. When the buffer has been filled, a message is sent to the recording entity, at which time the recording can be stopped, the buffer unprepared (i.e., unregistered), and then used. The use of the WAV file format is beneficial in the sense that the WAV specification does not call for compression of the audio samples [1]; such compression, or other manipulation, would not provide suitable access to the samples.

A practical implementation of an audio-based source of randomness would provide a driver that is able to provide a dynamic entropy pool from which seed values can be drawn. The pool would be updated periodically, in order to bolster unpredictability. This means that the audio device would be periodically enabled in order to record sample noises. An owner of a handheld device may be in any number of places, so the study sought to provide a variety of sample locations, in addition to a variety of sample types. However, a simple continuous design was implemented for this study.

The sample locations chosen were: riding in an automobile, dining in a restaurant, attending a lecture, and working in a quiet office. These scenarios were chosen to reflect the potential places and situations a person may find them in throughout the course of any given day. Many people commute to work or otherwise use an automobile of some kind on a day-to-day basis. Dining in a restaurant is thought to have been a good simulation of any busy location, such as an airport terminal. Lecture attendance is not unlike attending a presentation or group meeting where (roughly) one person speaks at a time. Finally, the quiet office is the environment in which only typing, printing, body movements, and background noise are picked up by the audio recording device. These locations are referred to as *location sources* or *sources*.

For each of the collection locations, eight different samples were taken. The Wave API provides for two sample sizes: 8-bit and 16-bit. For each sample size, there are four sample rates available: 8kHz, 11kHz, 22kHz, and 44kHz. This yields a total of eight sample configurations for each of the location sources.

3.4.1.1. Test and Evaluation Software

Several small software components were implemented to assist in organizing the collected data for testing. Without going into too much detail, three tools were implemented for the Linux operating system: *deskew*, *colent*, and *truncfile*. Each of these tools take a user-supplied data file, manipulate the data contained in the file, then write the final result to a new, user-specified file. The *deskew* command is used for removing bias from collected data; *colent* is used to collect one bit of "entropy" for every byte in the collected data file; and *truncfile* is used to trim a given data file to a user-specified size. All of these Linux-based command line tools were implemented using the C programming language. For more information regarding the implementation of these tools, see the Appendix.

3.5. Test Method

Raw data was collected from each of the data sources (touchscreen, and audio input). The collected data was logged and filed prior to distillation. The distillation process performed as many as two steps, which were designed to do two things: collect randomness from the data, and to deskew (remove bias from) the data. Randomness was collected by taking the least significant bit of every byte collected, and deskewing was performed by the method listed in Section 3.5.1.

The collected data was distilled into four distinct categories:

- **Raw Data:** the uncorrected data,
- **Entropy Data:** raw data that has been distilled for randomness
- **Deskewed Data:** raw data that has been distilled for bias correction, and
- **Entropy and Deskewed Data:** entropy data that has been distilled for bias correction.

3.5.1. Deskewing To Remove Bias

The deskewing method used in mentioned in [13] and [8]. This distillation process is necessary because the generator may provide a sequence that contains a greater number of ones or zeros in the bit-sequence. The method of deskewing data is to look at the bits in a sequence t in bit pairs t_i and t_{i+1} for all i from $i = 1, 3, 5, \ldots, n$. If t_i and t_{i+1} are equal, discard the bits and move on to the next pair; otherwise, store t_i as an output of the generator and discard t_{i+1} before iterating. This method of deskewing produces unbiased output from any generator[8].

3.5.2. Selected Suite

The test suite chosen for this study is the ENT Test Suite[4]. The ENT Test Suite is comprised of the Chi-Square Distribution Test, Arithmetic Mean Test, Monte Carlo Estimation of Pi, and the Serial Correlation Test. In addition to these specific tests, the estimated entropy-per-bit is also given by the selected test suite. The test suite was chosen because it is suitable for testing random seed data[5].

3.5.2.1. Chi-Square Distribution Test

The Chi-Square Distribution Test is, perhaps, the most common test of randomness available. It is also used as a foundation for other randomness tests [12]. The distribution is calculated for the input stream and represented as an absolute value and a percentage, where the percentage indicates the frequency at which a truly random sequence (uniformly distributed) would exceed the absolute value [7]. The interpretation of such results lies in the interpretation of the given percentage; the percentage is taken to be the degree to which the sequence is suspected of being non-random [7].

A sequence is judged to be non-random, if the given percentage is greater than 99% or less than 1%. A sequence is suspected of being non-

[4] Much of the information contained in this section is not intended to describe the tests mathematically, but to describe their general operation and to provide proper methods of result interpretation; moreover, much of the information has been adapted from [3].
[5] Much of the information contained in the test description sections was adapted from [3], except where otherwise noted.

random, if the given percentage lies between 95% and 99%, and if the given percentage lies between 1% and 5%. Percentages between 90% and 95% and between 5% and 10% are "almost suspect" [7]. If the percentage given does not fall into any of these ranges, then the sequence can be judged as random.

3.5.2.2. Arithmetic Mean Test

The Arithmetic Mean Test takes the sum of the bits contained in the sequence, then divides the sum by the length of the sequence. This is, in effect, a frequency test that indicates how many ones and zeros exist in the given sequence. For bit sequences, the closer the result of this test comes to 0.5, the more likely it is that the given sequence is random.

3.5.2.3. Monte Carlo Estimation of Pi

The Monte Carlo Estimation of Pi is a test that first gathers the bits of the sequence into bytes. The bytes are then interpreted as successive 24-bit coordinates within a square. If the point falls within a circle inscribed in the square, than that point is registered as a "hit." The hits falling within the circle are then used to estimate the area of the circle, from which the estimation of Pi is derived.

3.5.2.4. Optimal Compression (Estimated Entropy)

The estimation of entropy is derived, in the ENT Test Suite, from the ability to compress a file "optimally." If the file is extremely compressible, then it is judged to be non-random. The result of this test is given as a percentage and is then used to estimate the entropy contained in each bit of the sequence. A value close to 1 is desired.

3.5.2.5. Serial Correlation Test

The Serial Correlation Test measures the extent to which a given bit in the sequence is correlated to past or future bits in the sequence. The result of this test is given on a scale from zero to one, where zero indicates no detected correlation and one indicates definite correlation. The results of this test will be closer to zero for sequences that approach true randomness. Further description of this test can be found in [7].

3.6. Quantity of Collected Data

Approximately 1MB of data was collected for each source configuration. Randomness was then collected from these files using the *colent* command. After randomness was collected, this new data set was deskewed using the *deskew* command. This process resulted in a file significantly smaller in size than the raw data file. The smallest file size after full distillation was just over 15KB, so all of the files used for testing were truncated (using the *truncfile command*) to 15,360 bytes (122,880 bits). As is evident from the data quantity reduction due to the distillation process, it would be advantageous for any practical implementation to avoid, if possible, distillation measures for the sake of processing time reduction, and there for a reduction in power consumption.

4. Results

4.1. Entropy Estimation

The entropy-per-bit range of the results extend from approximately 0.78 to 1.0, which yields approximately 6.24 to 8 bits of entropy (randomness) per byte. Figure 1: Entropy-per-bit of Audio Samples shows the entropy-per-bit of the audio samples at all levels of distillation; and Figure 2: Entropy-per-bit of Touchscreen Samples shows the entropy-per-bit of the touchscreen samples at all levels of distillation.

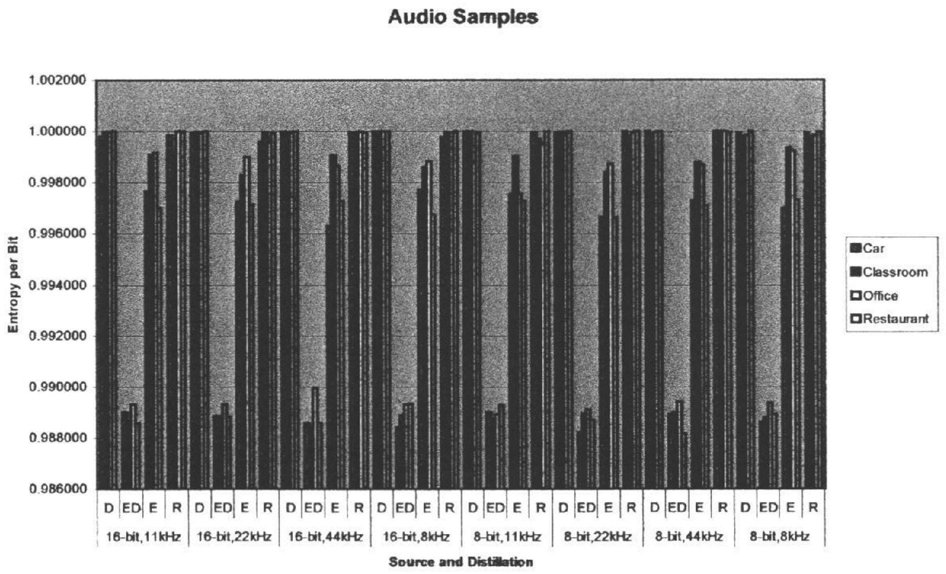

Figure 1: Entropy-per-bit of Audio Samples

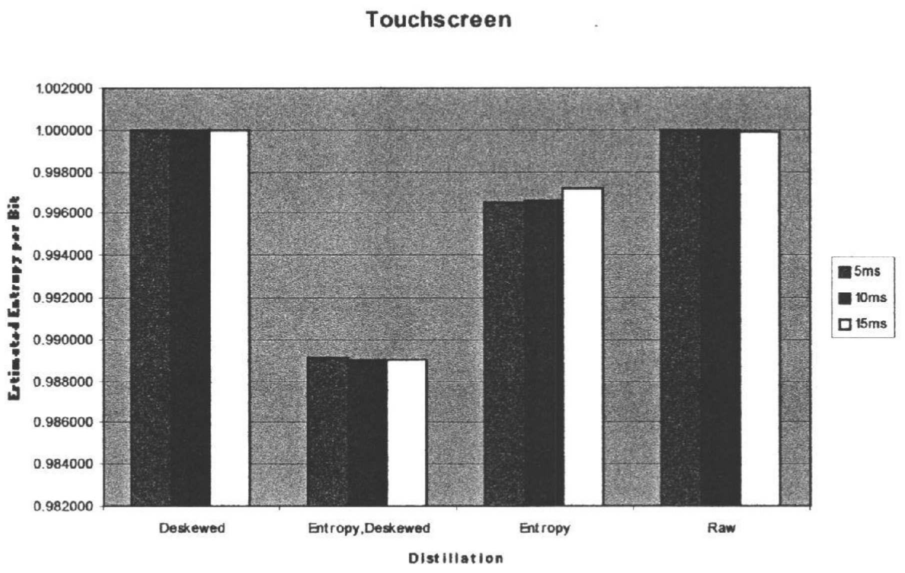

Figure 2: Entropy-per-bit of Touchscreen Samples

Figure 1: Entropy-per-bit of Audio Samples is somewhat difficult to interpret, so Figure 3: Average Entropy-per-bit of Audio Samples shows the average estimated entropy-per-bit of all audio sample configurations, i.e., sample size and rate, at all levels of distillation.

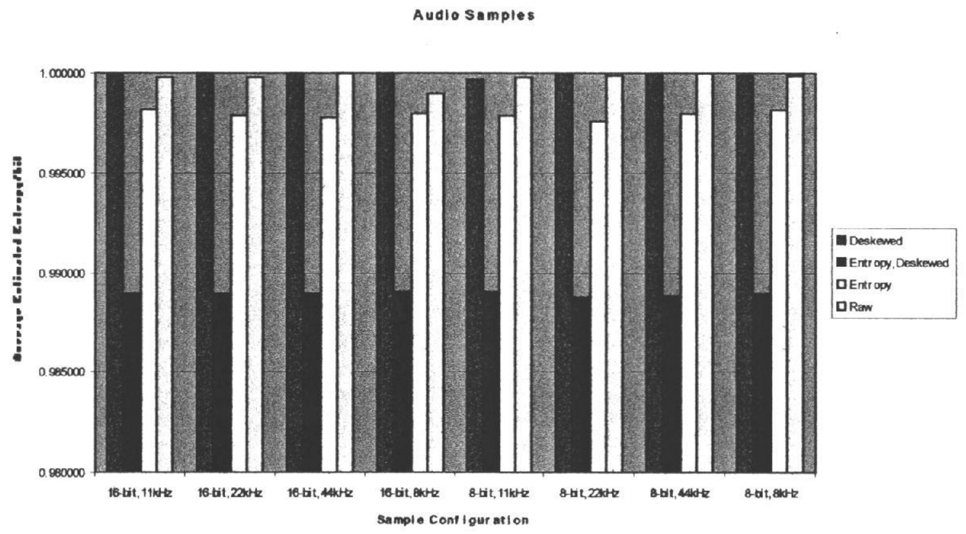

Figure 3: Average Entropy-per-bit of Audio Samples

4.2. Arithmetic Mean

These are the results of the ENT Arithmetic Mean Test. Recall that a value close to 0.5 is desired for a random sequence. Figure 4: Arithmetic Mean of Audio Samples shows the arithmetic mean of the audio samples at all levels of distillation; and Figure 5: Arithmetic Mean of Touchscreen Samples shows the arithmetic mean of the touchscreen samples at all levels of distillation.

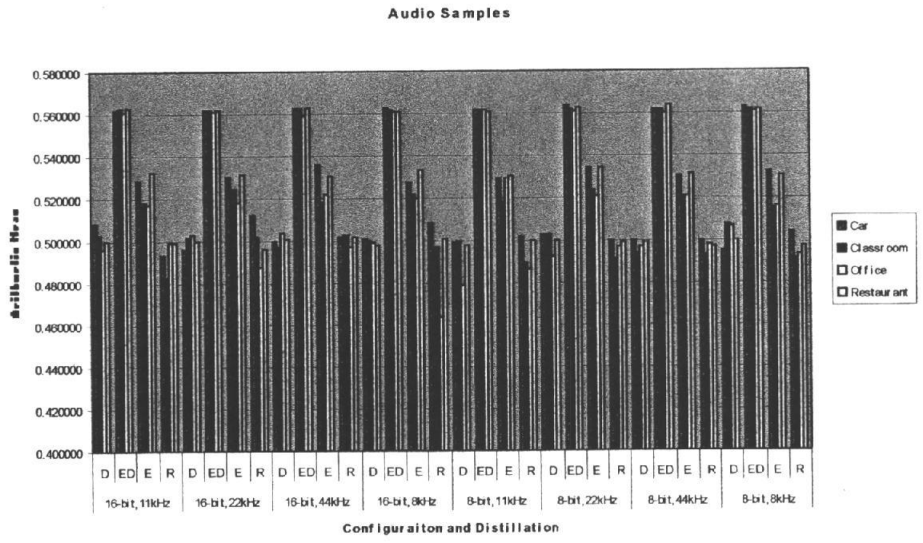

Figure 4: Arithmetic Mean of Audio Samples

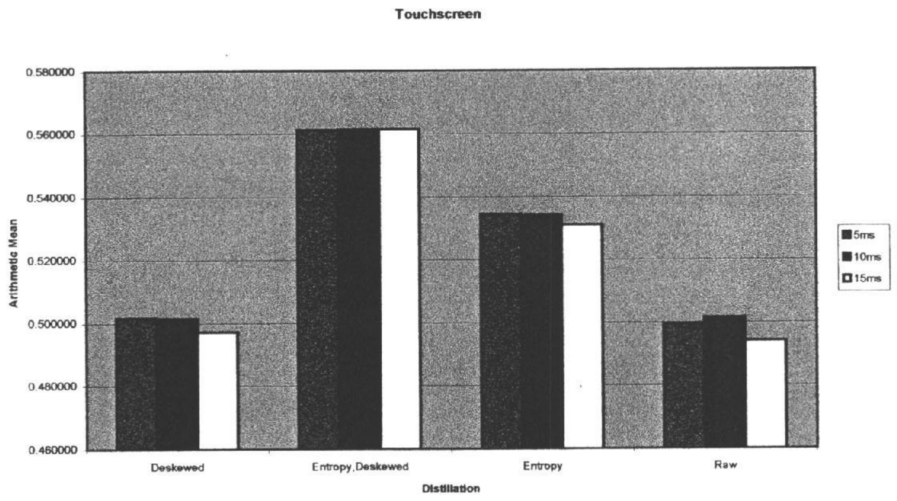

Figure 5: Arithmetic Mean of Touchscreen Samples

4.3. Monte Carlo Estimation of Pi

These are the results of the Monte Carlo Estimation of Pi. The closer the estimation comes to Pi, the more random the sequence is assumed to be. Figure 6: Monte Carlo Estimation of Pi for Audio Samples shows the estimation of the audio samples at all levels of distillation; and Figure 7: Monte Carlo Estimation of Pi for Touchscreen Samples shows the estimation of the touchscreen samples at all levels of distillation.

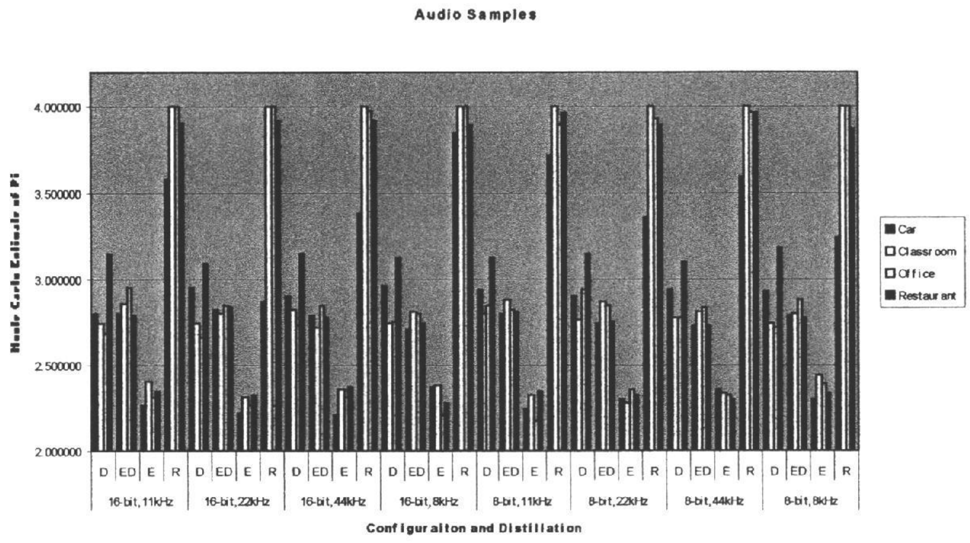

Figure 6: Monte Carlo Estimation of Pi for Audio Samples

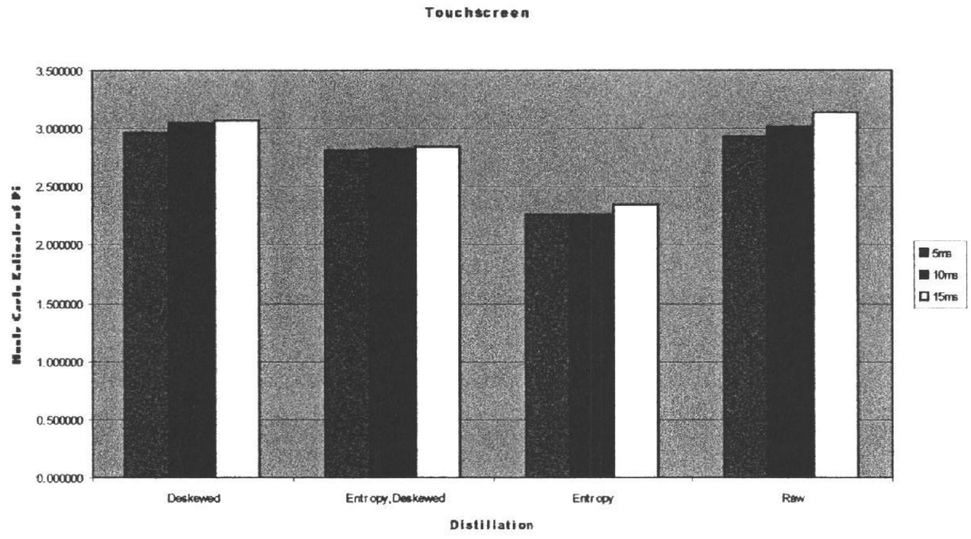

Figure 7: Monte Carlo Estimation of Pi for Touchscreen Samples

4.4. Serial Correlation

These are the results of the ENT Serial Correlation Test. Recall that a value close to 0.0 is desired for a random sequence. Figure 8: Serial Correlation of Audio Samples shows the serial correlation of the audio samples at all levels of distillation; and Figure 9: Serial Correlation of Touchscreen Samples shows the serial correlation of the touchscreen samples at all levels of distillation.

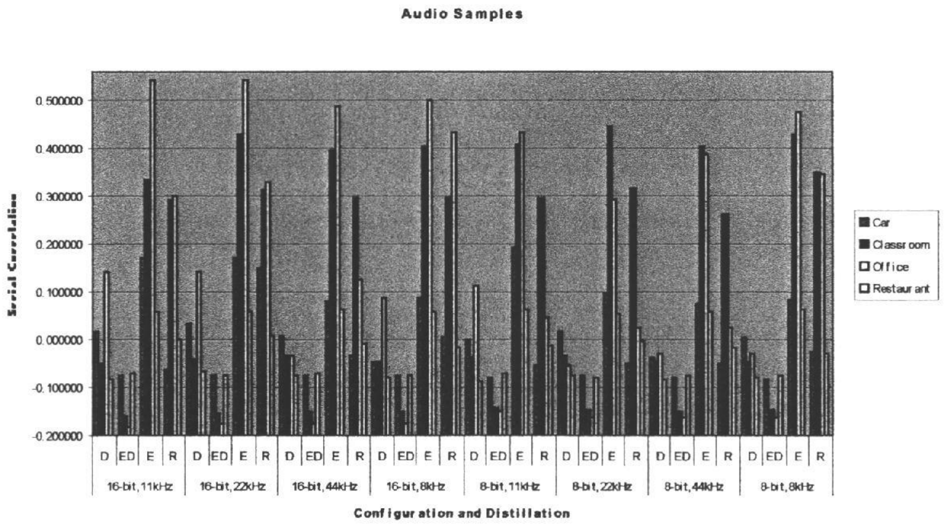

Figure 8: Serial Correlation of Audio Samples

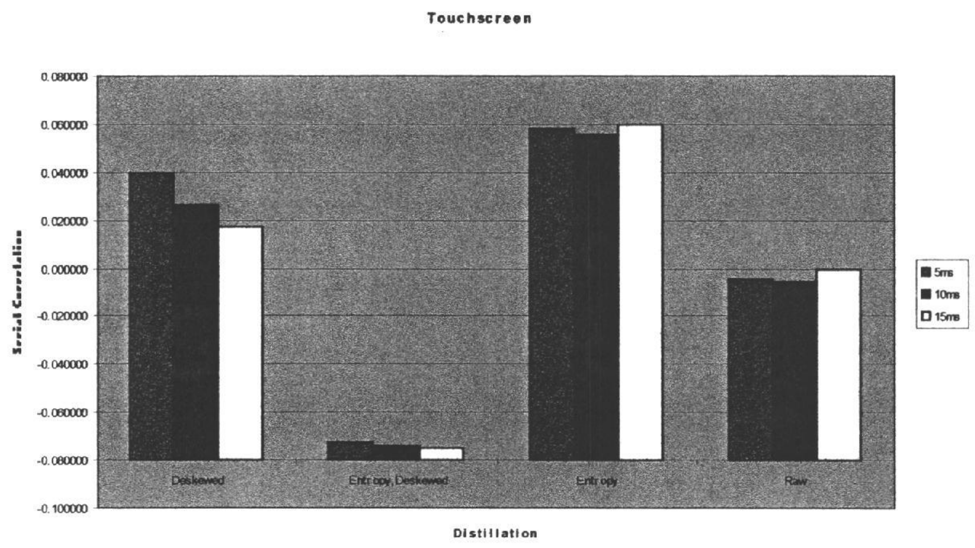

Figure 9: Serial Correlation of Touchscreen Samples

Figure 8: Serial Correlation of Audio Samples is somewhat difficult to interpret, so Figure 10: Average Serial Correlation of Audio Samples shows the *average* estimation of all audio sample configurations, i.e., sample size and rate, at all levels of distillation.

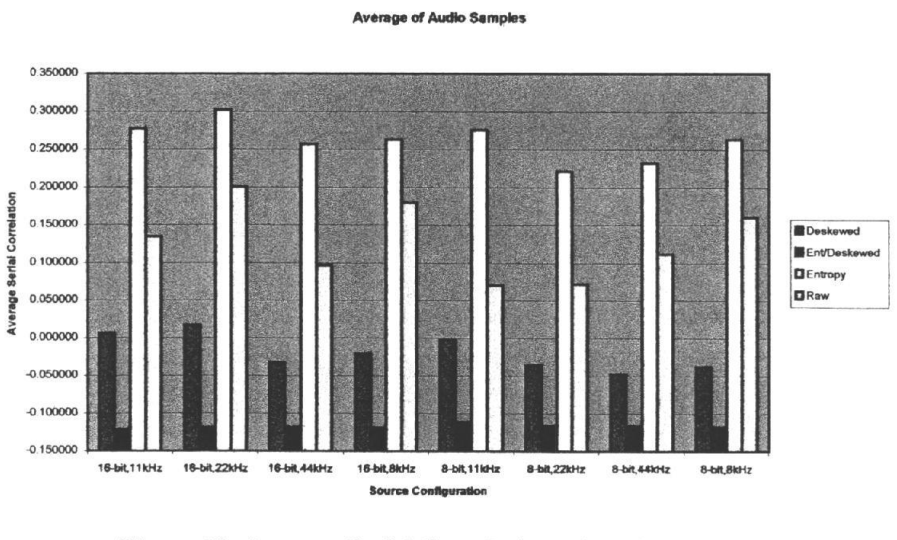

Figure 10: Average Serial Correlation of Audio Samples

5. Discussion

5.1. Analysis of Audio Sampling

Prior to testing, it was thought that the audio samples would require extensive distillation in order to achieve randomness. One reason for this prejudice stems from the thought that the audio input hardware on the handheld device was likely to have some filtering to reduce noise. Another reason is that the audio hardware is not directly accessible. Rather, the operating system had to be used as the interface to the audio hardware, and it was thought that the operating system might further filter the sample input, or otherwise alter the data prior to testing. As mentioned in [1], the WAV file format does not compress audio samples, however.

The audio samples held up well to most of the ENT test suite. Of all the samples, however, the deskewed and raw samples seemed to perform the best. Of these, the 16-bit samples at either 8kHz or 44kHz (see Figure 1: Entropy-per-bit of Audio Samples) proved superior. The audio samples did not fair as well as the touchscreen in the Monte Carlo estimation of Pi. However, it appears that the deskewed and fully distilled audio samples performed the best against this test.

The audio samples produced widely variant results when up against the Serial Correlation Test, and very few of them fell in the range given by [7]. The only audio samples that satisfied the given acceptable range were:

- 16-bit, 11kHz restaurant,
- 16-bit, 8kHz car,
- 8-bit, 11kHz car,
- 8-bit, 22kHz restaurant, and
- 8-bit, 8kHz car

A potential reason for this is that large quantities of audio data were collected at once which is not likely to be the preferred implementation for cryptographic purposes. It is likely that the data collected from a dynamic pool of audio samples over an extended period of time will produce much better correlation results. Such a dynamic pool may be implemented as a stream driver in the Pocket PC operating system, which then would not need to be statically (or dynamically) linked to any particular cryptographic application, but available to all.

5.2. Analysis of Touchscreen Sampling

As with the audio Samples, the touchscreen sampling performed well at all sample resolutions, but the 15ms resolution proved to be superior in most cases. In all cases, the deskewed or raw touchscreen data samples appeared to yield the best test results, with the deskewed samples showing slightly better results in all but the Serial Correlation Test. Unfortunately, the only touchscreen samples that passed the Serial Correlation Test according to the range given in [7] were the raw samples collected at all resolutions. A potential reason for this deviance may be due to the implementation of the touchscreen collection algorithm.

The touchscreen resolution on a Pocket PC device is 240 x 320, and the coordinate system on the Pocket PC is such that the *x* and *y* coordinate values are placed into 16-bit WORDs. This is necessary because of the range on the *y*-axis (320). The collection method (given in 3.3.1.1.1) provides for an 8-bit collection result, which means that the collection will represent a maximum of 256 values for the *y*-axis. The consequence of this implementation is that those *y* values that range from 256 to 339 are mapped to the first 64 values of the *y*-axis.

Another possibility for the correlation of the touchscreen sampling may lie in the fact that a user supplied the input. If, at some point during the data collection, the user interacting with the program began scribbling in a patterned way, then it may be the case that the serial correlation numbers would grow further from zero, thus showing a higher degree of correlation from bit to bit. Yet another possibility for the correlation is the fact that the screen resolution, and therefore its binary representation, is bounded. Such bounds are known to provide, in some cases, a higher degree of correlation [2].

The serial correlation is likely to be of greater importance to the decision of whether to use the touchscreen for a random bit sequence, because it is not as dynamic as the audio can be when implemented as a driver. The touchscreen could, if the Original Equipment Manufacturer (OEM) of a Pocket PC device were to incorporate his into their native touchscreen driver, collect data periodically form the screen as the device is used throughout the day. However, this may provide an unwanted bias in the sense that the Pocket PC invites a particular pattern of use. The "Start" button is tapped often, and the resulting pop-up menu is bound to have some preferred applications (i.e., "tap points"). In addition, the command bar and other menu items are typically restricted to the top and bottom of the display. Essentially, the typical use of a Pocket PC is not likely to provide a good source of randomness, which means that the user will need to intervene and generate a new seed manually, or when prompted to do so. Either way is invasive and not likely to be adopted by users.

5.3. Chi-Square Distribution Analysis

Because the Chi-Square Distribution is, perhaps, the most sensitive test in the ENT Test Suite, it has been given special consideration in this discussion. The results have thus far indicated that 16-bit audio samples taken at either 8kHz or 44kHz rates yield the best random data in this study (when serial correlation is not considered). Moreover, the deskewed and raw samples provided the best random data out of all the 16-bit, 8/44kHz samples. Figure 11: Chi-Square Test Results for 16-Bit Audio Samples (RAW) shows the percentage of the time that raw audio samples of the given configuration will exceed the Chi-Square Distribution value. Recall that very high or very low percentages are concluded to be non-random, and that the closer to 50% the percentage is, the more likely it is that the sequence under test is random [3] [7].

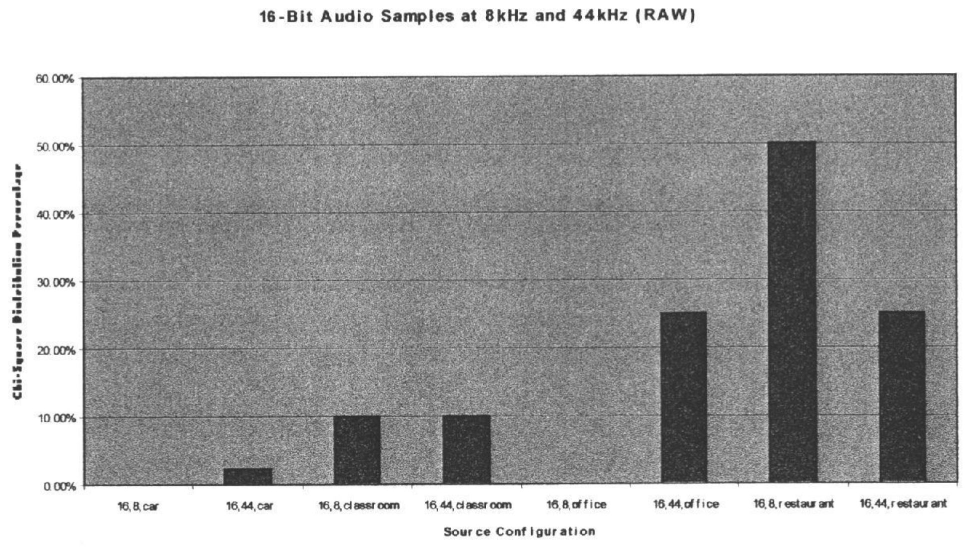

Figure 11: Chi-Square Test Results for 16-Bit Audio Samples (RAW)

Similarly, Figure 12: Chi-Square Test Results for 16-Bit Audio Samples (DESKEWED) shows the percentage of the time that the deskewed audio samples of the given configuration will exceed the Chi-Square Distribution value.

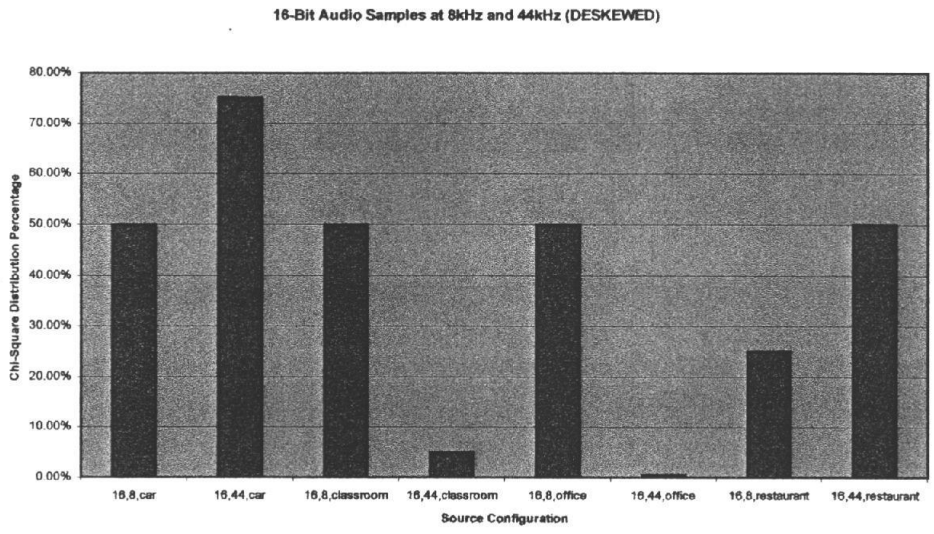

Figure 12: Chi-Square Test Results for 16-Bit Audio Samples (DESKEWED)

As is clearly evident in Figure 11 and Figure 12, the deskewed, 16-bit audio samples taken at a rate of 8kHz provide the most satisfactory results when the Chi-Squared Distribution is utilized. Out of eight sample configurations, the two that were most unacceptable came from 16-bit 44kHz samples (in the classroom and office). The only time the 44kHz samples faired better than the 8kHz samples was in the restaurant location. Otherwise, the 16-bit, 8kHz samples were consistently near 50%, and therefore performed very well against the Chi-Squared Distribution test.

6. Conclusion

The results of this study are both promising and concerning. On some levels, the results appear to be promising, and should elicit further study in these areas. On other levels, the results seemed to warrant that the research direction should be altered to other potential sources of randomness.

6.1. General Conclusions

As mentioned in Section 4, the touchscreen and audio implementations were not ideal, and even though the samples appear promising for the purpose of random number generation, the implementation issues should be fixed and the data resampled and retested prior to further exploration of the subject. Nonetheless, these tests indicate that the chosen sources of randomness may be suitable for cryptographic applications, which is contradictory to the findings of [1], where it is claimed that microphone input is not an adequate source of entropy on any computing system.

Of all the sources examined, the audio source appears to be the best when tested with the ENT Test Suite. In particular, the 16-bit audio samples at either 8kHz or 44kHz seemed to provide the best results in most areas when raw or deskewed data was used. It is expected that a dynamic pool implementation of the audio collection will provide a correction to the relatively sporadic serial correlation showing of the audio samples, and that a larger ample collected as would be in a practical application would provide a better estimate of Pi via the Monte Carlo method.

All things considered, handheld devices may possess adequate random sources for the purpose of seeding deterministic PRNGs. In particular, those handheld devices capable of recording audio data are likely to provide a suitable means of gathering seed data.

6.2. Recommendations for Future Work

It can be concluded from this study that 16-bit audio samples taken at a sampling rate of 8kHz, when deskewed, provide the best source of randomness from those examined on the Dell Axim X5 Pocket PC handheld device. Further inference of these results may be examined in future work. A continuation of this work may be interested in a more practical implementation of an audio RNG system with emphasis placed on audio sources with additional, more stringent testing. Further, future work should be interested in performing identical tests across multiple devices. Currently, the results are only applicable to a Dell Axim X5.

The 16-bit audio samples at 8kHz and 44kHz sample rates performed well against the Arithmetic Mean, Estimated Entropy and the Chi-Square tests. These samples performed only marginally well against the Monte Carlo Estimate of Pi, and did not fair well at all against the Serial Correlation Test. It is recommended that further study be performed in the area of attempting to massage better Serial Correlation results by using a more practical implementation, and by using a larger amount of data.

REFERENCES

[1] Giles Cotter. Generation of pseudorandom numbers from microphone input in computer devices, March 2002.

[2] J. Schiller, D. Eastlake, S. Crocker. Randomness recommendations for security. Technical report, Internet Engineering Task Force, December 1994.

[3] http://www.fourmilab.ch/random/, 2003. A search for "ENT random" on Google will turn up more references to the same content than that which is listed here.

[4] Dieter Gollmann. *Computer Security.* John Wiley and Sons, Inc., 1999.

[5] Intel Corporation. *Intel XScale Core Developer's Manual*, December 2000.

[6] Intel Corporation. *Intel StrongARM SA-1100 Microprocessor Developer's Manual*, October 2001.

[7] D.E. Knuth. *The Art of Computer Programming, Volume 2.* Addison-Wesley, Third Edition, 1998.

[8] A. Menezes. *Handbook of Applied Cryptography.* CRC Press, 1996.

[9] Motorola, Inc. MC68328 (DragonBall) Integrated Processor User's Manual, 1995.

[10] National Institute of Standards and Technology. *Specifications for the Data Encryption Standard (DES)*, October 1999.

[11] National Institute of Standards and Technology. *Specification for the Advanced Encryption Standard (AES)*, November 2001.

[12] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elain Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, San Vo. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, National Institute of Standards and Technology, May 2001.

[13] Bruce Schneier. *Applied Cryptography.* John Wiley and Sons, Inc., Second Edition, 1996.

[14] William Stallings. *Cryptography and Network Security Principles and Practice.* Prentice-Hall, Inc., Second Edition, 1999.

[15] Douglas R. Stinson. *Cryptography Theory and Practice.* Chapman and Hall, Second Edition, 2002.

[16] Stephen Thomas. *SSL and TLS Essentials.* John Wiley and Sons, Inc., 2000.

[17] U.S. Department of Commerce/National Institute of Standards and Technology. *Digital Signature Standard (DSS)*, January 2000.

APPENDICES

This section contains the source listings for all code developed for this research. The code is divided into two sections: Pocket PC and Linux.

Pocket PC Source Listings

Resource.h

```
#define IDS_APP_TITLE                   1
#define IDS_HELLO                       2
#define IDC_AUDIO                       3
#define IDI_AUDIO                       101
#define IDM_MENU                        102
#define IDD_ABOUTBOX                    103
#define IDM_FILE_EXIT                   40002
#define IDM_HELP_ABOUT                  40003
#define IDM_RUN_1MB                     40004
#define IDM_RUN_2MB                     40005
#define IDM_RUN_4MB                     40006
#define IDM_RUN_8MB                     40007
#define IDM_RUN_16MB                    40008
#define IDM_RUN_32MB                    40009
#define IDM_SETTING_16_BIT              40010
#define IDM_SETTING_8_BIT               40011
#define IDM_SETTING_8_KHZ               40012
#define IDM_SETTING_11_KHZ              40013
#define IDM_SETTING_22_KHZ              40014
#define IDM_SETTING_44_KHZ              40015
#define IDM_RUN_TOUCH                   40016
#define IDM_RUN_OS_RAND                 40017
#define IDM_RUN_IR                      40018
#define IDM_SETTING_GEN_DATA            40021
#define IDM_RUN_RAND_AUD                40022
#define IDM_RUN_RAND_TS                 40023
#define IDM_RUN_RAND_OS                 40024
#define IDM_RUN_RAND_IR                 40025
#define IDM_SETTING_TS_25               40026
#define IDM_SETTING_TS_15               40027
#define IDM_SETTING_TS_5                40028
#define IDM_INSTRUCTION                 40029
#define IDM_SETTING_AMT_1K              40031
#define IDM_SETTING_AMT_5K              40032
#define IDM_SETTING_AMT_10K             40033
#define IDM_SETTING_AMT_50K             40034
#define IDM_SETTING_AMT_100K            40035
#define IDM_SETTING_AMT_250K            40036
#define IDM_SETTING_TS_10               40037
#define IDM_SETTING_AMT_512K            40039
#define IDM_SETTING_AMT_1M              40040
#define IDM_SETTING_AMT_5M              40041
```

```
// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        104
#define _APS_NEXT_COMMAND_VALUE         40042
#define _APS_NEXT_CONTROL_VALUE         1001
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

StdAfx.h

```
// stdafx.h : include file for standard system include files,
// or project specific include files that are used
// frequently, but
// are changed infrequently
//

#if
!defined(AFX_STDAFX_H__A9DB83DB_A9FD_11D0_BFD1_444553540000__I
NCLUDED_)
#define
AFX_STDAFX_H__A9DB83DB_A9FD_11D0_BFD1_444553540000__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define WIN32_LEAN_AND_MEAN        // Exclude rarely-used
stuff from Windows headers

// Windows Header Files:
#include <windows.h>
#include <commctrl.h>
#include <stdio.h>


// Local Header Files
#include "audio.h"
#include "sound_recorder.h"
#include "file_writer.h"
#include "os_rand.h"
#include "ir_rand.h"
#include "prng_ppc_sha.h"

// TODO: reference additional headers your program requires
here
```

```
//{{AFX_INSERT_LOCATION}}
// Microsoft eMbedded Visual C++ will insert additional
declarations immediately before the previous line.

#endif //
!defined(AFX_STDAFX_H__A9DB83DB_A9FD_11D0_BFD1_444553540000__I
NCLUDED_)
```

StdAfx.cpp

```
// stdafx.cpp : source file that includes just the standard
includes
//    audio.pch will be the pre-compiled header
//    stdafx.obj will contain the pre-compiled type
information

#include "stdafx.h"

// TODO: reference any additional headers you need in STDAFX.H
// and not in this file
```

audio.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////
///////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "newres.h"

//////////////////////////////////////////////////////////////
///////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////
///////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32
```

```
//////////////////////////////////////////////////////////////
///////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure
application icon
// remains consistent on all systems.
IDI_AUDIO                ICON    DISCARDABLE     "audio.ICO"

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////
///////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""newres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED


//////////////////////////////////////////////////////////////
///////////////
//
// Menubar
//

IDM_MENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "E&xit",                        IDM_FILE_EXIT
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About",                          IDM_HELP_ABOUT
        MENUITEM "Instructions",
IDM_INSTRUCTION
    END
    POPUP "Run"
    BEGIN
        MENUITEM "Collect raw Audio",               IDM_RUN_1MB
        MENUITEM "Collect raw Touchscreen",         IDM_RUN_TOUCH
        MENUITEM "Collect PRNG from OS",
IDM_RUN_OS_RAND
        MENUITEM "Collect raw IR",                  IDM_RUN_IR
        MENUITEM SEPARATOR
        MENUITEM "Get PRN from Audio",
IDM_RUN_RAND_AUD
        MENUITEM "Get PRN from Touchscreen",
IDM_RUN_RAND_TS
        MENUITEM "Get PRN from OS",
IDM_RUN_RAND_OS
        MENUITEM "Get PRN from IR",
IDM_RUN_RAND_IR
    END
    POPUP "Settings"
    BEGIN
        POPUP "AUDIO"
        BEGIN
            MENUITEM "16 Bit Samples",
IDM_SETTING_16_BIT
            MENUITEM "8 Bit Samples",
IDM_SETTING_8_BIT
            MENUITEM "8 kHz",
IDM_SETTING_8_KHZ
            MENUITEM "11 kHz",
IDM_SETTING_11_KHZ
            MENUITEM "22 kHz",
IDM_SETTING_22_KHZ
            MENUITEM "44 kHz",
IDM_SETTING_44_KHZ
        END
        POPUP "Touchscreen"
        BEGIN
            MENUITEM "15 ms delay",
IDM_SETTING_TS_15
            MENUITEM "10 ms delay",
IDM_SETTING_TS_10
            MENUITEM "5 ms delay",
IDM_SETTING_TS_5
        END
        POPUP "Data Collection Amount"
        BEGIN
```

```
            MENUITEM "1 KB",
IDM_SETTING_AMT_1K
            MENUITEM "5 KB",
IDM_SETTING_AMT_5K
            MENUITEM "10 KB",
IDM_SETTING_AMT_10K
            MENUITEM "50 KB",
IDM_SETTING_AMT_50K
            MENUITEM "100 KB",
IDM_SETTING_AMT_100K
            MENUITEM "250 KB",
IDM_SETTING_AMT_250K
            MENUITEM "512 KB",
IDM_SETTING_AMT_512K
            MENUITEM "1 MB",
IDM_SETTING_AMT_1M
        END
    END
END


//////////////////////////////////////////////////////////////
///////////////
//
// Dialog
//

IDD_ABOUTBOX DIALOG DISCARDABLE  0, 0, 125, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
EXSTYLE 0x80000000L
CAPTION "About audio"
FONT 8, "System"
BEGIN
    ICON            IDI_AUDIO,IDC_STATIC,11,17,20,20
    LTEXT           "audio Version
1.0",IDC_STATIC,38,10,70,8,SS_NOPREFIX
    LTEXT           "Copyright (C) 2003",IDC_STATIC,38,25,70,8
END


//////////////////////////////////////////////////////////////
///////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_ABOUTBOX, DIALOG
    BEGIN
```

```
        LEFTMARGIN, 7
        RIGHTMARGIN, 118
        TOPMARGIN, 7
        BOTTOMMARGIN, 48
    END
END
#endif    // APSTUDIO_INVOKED


//////////////////////////////////////////////////////////////
///////////////
//
// Accelerator
//

IDC_AUDIO ACCELERATORS DISCARDABLE
BEGIN
    "/",            IDM_HELP_ABOUT,          ASCII,  ALT,
NOINVERT
    VK_F4,          IDM_FILE_EXIT,           VIRTKEY, ALT,
NOINVERT
END


//////////////////////////////////////////////////////////////
///////////////
//
// String Table
//

STRINGTABLE DISCARDABLE
BEGIN
    IDS_APP_TITLE           "audio"
    IDS_HELLO               "Hello World!"
    IDC_AUDIO               "AUDIO"
END

#endif    // English (U.S.) resources
//////////////////////////////////////////////////////////////
///////////////



#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////
///////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//
```

```
////////////////////////////////////////////////////////////////
////////////////
#endif    // not APSTUDIO_INVOKED
```

newres.h

```
#ifndef __NEWRES_H__
#define __NEWRES_H__

#if !defined(UNDER_CE)
      #define UNDER_CE _WIN32_WCE
#endif

#if defined(_WIN32_WCE)
      #if !defined(WCEOLE_ENABLE_DIALOGEX)
            #define DIALOGEX DIALOG DISCARDABLE
      #endif
      #include <commctrl.h>
      #define  SHMENUBAR RCDATA
      #if defined(WIN32_PLATFORM_PSPC) && (_WIN32_WCE >= 300)
            #include <aygshell.h>
      #else
            #define I_IMAGENONE           (-2)
            #define NOMENU                0xFFFF
            #define IDS_SHNEW       1

            #define IDM_SHAREDNEW        10
            #define IDM_SHAREDNEWDEFAULT 11
      #endif
#endif // _WIN32_WCE


#ifdef RC_INVOKED
#ifndef _INC_WINDOWS
#define _INC_WINDOWS
      #include "winuser.h"           // extract from windows
header
#endif
#endif

#ifdef IDC_STATIC
#undef IDC_STATIC
#endif
#define IDC_STATIC      (-1)

#endif //__NEWRES_H__
```

file_writer.h

```
/**************************************************************
***********

  FILE:                  file_writer.h
  PURPOSE:         Header information for file_writer.c
  DATE:                  April 2003
  AUTHOR:          Adam W. Montville
                         Information Security Laboratory
                         Oregon State University
                         montviad@ece.orst.edu

**************************************************************
**********/

#ifndef __FILE_WRITER
#define __FILE_WRITER

#include "stdafx.h"

// FUNCTION DECLARATIONS (See Implementation for descriptions)
int FileWriterInitialize(char *);
int FileWriterWrite(char);
int FileWriterClose(void);
int FileWriterNewline(void);
int FileWriterComma(void);

#endif // __FILE_WRITER
```

file_writer.cpp

```
/**************************************************************
***********

  FILE:                  file_writer.c
  PURPOSE:         Interface for writing files when collecting
PRNG seed
                         data.  This interface uses the
standard C-style file
                         functions.
  DATE:                  April 2003
  AUTHOR:          Adam W. Montville
                         Information Security Laboratory
                         Oregon State University
                         montviad@ece.orst.edu

**************************************************************
**********/

#include <stdio.h>
```

```
#include "file_writer.h"


FILE *outFile;                      // File pointer



/**
 * FUNCTION:      FileWriterInitialize
 * DATE:          April 2003
 * PURPOSE:       Initialize the file writing component for
 *                      operation.
 * PARAMS:
 *                      filePath -- character pointer to file
 * RETURN:
 *                      0 -- Failure
 *                      1 -- Success
 */
int FileWriterInitialize(char *filePath) {
      int result = 1;

      outFile = fopen (filePath, "w");          // open for
writing
      if(outFile == NULL)
            result = 0;

      return result;
}


/**
 * FUNCTION:      FileWriterWrite
 * DATE:          April 2003
 * PURPOSE:       Writes a single unisigned integer to the
file
 *                      pointed to by outFile.
 * PARAMS:
 *                      data -- unisgned integer data to be
written
 * RETURN:
 *                      -1 -- Not initialized
 *                       0 -- Failure
 *                       1 -- Success
 */
int FileWriterWrite(char data) {
      int result = 1;

      if(outFile != NULL) {
            if(fprintf(outFile, "%c", data) != sizeof(data))
                  result = 0;
      } else {
```

```
		result = -1;
	}

	return result;
}


int FileWriterNewline(void) {
	int result = 1;
	if(outFile != NULL) {
		fprintf(outFile, "\n");
	} else {
		result = 0;
	}
	return result;
}

int FileWriterComma(void) {
	int result = 1;
	if(outFile != NULL) {
		fprintf(outFile, ",");
	} else {
		result = 0;
	}
	return result;
}

/**
 * FUNCTION:	FileWriterClose
 * DATE:	April 2003
 * PURPOSE:	Closes the file pointer and frees any used
 *			memory that may have been allocated
 * PARAMS:
 *			NONE
 * RETURN:
 *			-1 -- Not initialized
 *			 0 -- Failure
 *			 1 -- Success
 */
int FileWriterClose(void) {
	int result = 1;

	if(outFile != NULL) {
		if((fclose(outFile)) != 0)
			result = 0;
	} else {
		result = -1;
	}

	return result;
}
```

audio.h

```
#if
!defined(AFX_AUDIO_H__68F81C24_E17E_412B_99F8_7BCF206FE788__IN
CLUDED_)
#define
AFX_AUDIO_H__68F81C24_E17E_412B_99F8_7BCF206FE788__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "resource.h"


#endif //
!defined(AFX_AUDIO_H__68F81C24_E17E_412B_99F8_7BCF206FE788__IN
CLUDED_)
```

audio.cpp

```
// audio.cpp : Defines the entry point for the application.
//

#include "stdafx.h"


#define MAX_LOADSTRING 100

// Global Variables:
HINSTANCE               hInst;                          //
The current instance
HWND                    hwndCB;                         //
The command bar handle
HWND                    g_hWnd;
TCHAR                   szOut[MAX_LOADSTRING];  // String to
be output
BOOL                    dataCollected = FALSE;  // indicates
whether data has been collected
BOOL                    first = TRUE;                //
Indicates first run.
BOOL                    working = FALSE;
BOOL                    audioDataCollected;
BOOL                    trackEnabled = FALSE;
BOOL                    audioEnabled = FALSE;
BOOL                    irEnabled = FALSE;
BOOL                    osEnabled = FALSE;
enum                    TASKS {NONE, AUDIO, IR, OS, TRACK};
CHAR                    *seedBuffer;
TASKS                   running = NONE;
```

```
UINT                    dataCount;
int                           currentBits;
int                           currentHertz;
char                    *outFileName = "rand.txt";
TCHAR                   errorMessage[MAX_LOADSTRING];
BOOL                    errorPresent = FALSE;
UINT                    myError;
SEED_DATA               prngData;
UINT                    tsDelay = 25;                //
default
UINT                    dataCollectionAmount = 1024;        //
default
DWORD                   threadId;
DWORD                   threadExitCode;


// Forward declarations of functions included in this code
module:
ATOM                    MyRegisterClass   (HINSTANCE, LPTSTR);
BOOL                    InitInstance      (HINSTANCE, int);
LRESULT CALLBACK  WndProc                 (HWND, UINT, WPARAM,
LPARAM);
LRESULT CALLBACK  About             (HWND, UINT, WPARAM,
LPARAM);
void                    AudioCallback     (void);
LRESULT                       DoMouseMain       (HWND hWnd,
UINT wMsg, WPARAM wParam, LPARAM lParam);
void                    LoggerFunction    (LPTSTR);
void                    PackSeedData      ();
DWORD WINAPI            AudioPowerManageSubversion(LPVOID
lpData);




int WINAPI WinMain(     HINSTANCE hInstance,
                              HINSTANCE hPrevInstance,
                              LPTSTR    lpCmdLine,
                              int       nCmdShow)
{
      MSG msg;
      HACCEL hAccelTable;

      // Perform application initialization:
      if (!InitInstance (hInstance, nCmdShow))
      {
            return FALSE;
      }
      hAccelTable = LoadAccelerators(hInstance,
(LPCTSTR)IDC_AUDIO);

      // Main message loop:
      while (GetMessage(&msg, NULL, 0, 0))
      {
            if (!TranslateAccelerator(msg.hwnd, hAccelTable,
&msg))
            {
                  TranslateMessage(&msg);
                  DispatchMessage(&msg);
            }
      }

      return msg.wParam;
}

//
//  FUNCTION: MyRegisterClass()
//
//  PURPOSE: Registers the window class.
//
//  COMMENTS:
//
//    It is important to call this function so that the
application
//    will get 'well formed' small icons associated with it.
//
ATOM MyRegisterClass(HINSTANCE hInstance, LPTSTR
szWindowClass)
{
      WNDCLASS    wc;

    wc.style                  = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc            = (WNDPROC) WndProc;
    wc.cbClsExtra       = 0;
    wc.cbWndExtra       = 0;
    wc.hInstance        = hInstance;
    wc.hIcon                  = LoadIcon(hInstance,
MAKEINTRESOURCE(IDI_AUDIO));
    wc.hCursor                = 0;
    wc.hbrBackground    = (HBRUSH)
GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName           = 0;
    wc.lpszClassName    = szWindowClass;

      return RegisterClass(&wc);
}

//
//  FUNCTION: InitInstance(HANDLE, int)
```

```
//
//  PURPOSE: Saves instance handle and creates main window
//
//  COMMENTS:
//
//    In this function, we save the instance handle in a
global variable and
//    create and display the main program window.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
      HWND  hWnd;
      TCHAR szTitle[MAX_LOADSTRING];                   // The
title bar text
      TCHAR szWindowClass[MAX_LOADSTRING];             // The
window class name
      int         result;
      hInst = hInstance;           // Store instance handle
in our global variable
      // Initialize global strings
      LoadString(hInstance, IDC_AUDIO, szWindowClass,
MAX_LOADSTRING);
      MyRegisterClass(hInstance, szWindowClass);

      LoadString(hInstance, IDS_APP_TITLE, szTitle,
MAX_LOADSTRING);
      hWnd = CreateWindow(szWindowClass, szTitle, WS_VISIBLE,
            CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
CW_USEDEFAULT, NULL, NULL, hInstance, NULL);

      if (!hWnd)
      {
            return FALSE;
      }

      ShowWindow(hWnd, nCmdShow);
      UpdateWindow(hWnd);
      if (hwndCB)
            CommandBar_Show(hwndCB, TRUE);


      g_hWnd = hWnd;
      currentBits = 8;
      currentHertz = 44100;

      return TRUE;
}

//
//  FUNCTION: WndProc(HWND, unsigned, WORD, LONG)
//
```

```
//  PURPOSE:  Processes messages for the main window.
//
//  WM_COMMAND    - process the application menu
//  WM_PAINT      - Paint the main window
//  WM_DESTROY    - post a quit message and return
//
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM
wParam, LPARAM lParam)
{
      HDC hdc;
      int wmId, wmEvent;
      PAINTSTRUCT ps;
      RECT rt;
      int result;
      GetClientRect(g_hWnd, &rt);


      switch (message)
      {
            case WM_COMMAND:
                  wmId    = LOWORD(wParam);
                  wmEvent = HIWORD(wParam);
                  // Parse the menu selections:
                  switch (wmId)
                  {
                        case IDM_HELP_ABOUT:
                           DialogBox(hInst,
(LPCTSTR)IDD_ABOUTBOX, hWnd, (DLGPROC)About);
                           break;
                        case IDM_FILE_EXIT:
                           DestroyWindow(hWnd);
                           break;
                        case IDM_SETTING_TS_10:
                              tsDelay = 10;
                              break;
                        case IDM_SETTING_TS_15:
                              tsDelay = 15;
                              break;
                        case IDM_SETTING_TS_5:
                              tsDelay = 5;
                              break;
                        case IDM_SETTING_AMT_1K:
                              dataCollectionAmount = 1024;
                              break;
                        case IDM_SETTING_AMT_5K:
                              dataCollectionAmount = 5120;
                              break;
                        case IDM_SETTING_AMT_10K:
                              dataCollectionAmount = 10240;
                              break;
                        case IDM_SETTING_AMT_50K:
                              dataCollectionAmount = 51200;
                              break;
                        case IDM_SETTING_AMT_100K:
                              dataCollectionAmount = 102400;
                              break;
                        case IDM_SETTING_AMT_250K:
                              dataCollectionAmount = 256000;
                              break;
                        case IDM_SETTING_AMT_512K:
                              dataCollectionAmount = 512000;
                              break;
                        case IDM_SETTING_AMT_1M:
                              dataCollectionAmount = 1048576;
                              break;
                        case IDM_RUN_1MB:
                              dataCollected = FALSE;
                              running = AUDIO;
                              CreateThread(NULL, NULL,
&AudioPowerManageSubversion, NULL, 0, &threadId);
                              InvalidateRect(g_hWnd, &rt,
TRUE);
                              result =
SoundRecorderInitialize(outFileName, dataCollectionAmount,
(DWORD)AudioCallback);
                              if(result != -1) {
                                    SoundRecorderStart();
                                    working = TRUE;
                              } else {
                                    running = NONE;
                                    InvalidateRect(g_hWnd,
&rt, TRUE);
                              }
                              break;
                        case IDM_RUN_TOUCH:
                              if(dataCollected == TRUE) {
                                    dataCount = 0;
                              }
                              dataCollected = FALSE;
                              running = TRACK;
                              InvalidateRect(g_hWnd, &rt,
TRUE);
                              break;
                        case IDM_RUN_OS_RAND:
                              dataCollected = FALSE;
                              running = OS;
                              InvalidateRect(g_hWnd, &rt,
TRUE);
                              break;
                        case IDM_RUN_IR:
                              dataCollected = FALSE;
                              running = IR;
                              InvalidateRect(g_hWnd, &rt,
TRUE);
                              break;
                        case IDM_SETTING_8_BIT:
                              SoundRecorderSetFormat(8,
currentHertz);
                              break;
                        case IDM_SETTING_16_BIT:
                              SoundRecorderSetFormat(16,
currentHertz);
                              break;
                        case IDM_SETTING_8_KHZ:

      SoundRecorderSetFormat(currentBits, 8000);
                              break;
                        case IDM_SETTING_11_KHZ:

      SoundRecorderSetFormat(currentBits, 11025);
                              break;
                        case IDM_SETTING_22_KHZ:

      SoundRecorderSetFormat(currentBits, 22050);
                              break;
                        case IDM_SETTING_44_KHZ:

      SoundRecorderSetFormat(currentBits, 44100);
                              break;

                        default:
                           return DefWindowProc(hWnd, message,
wParam, lParam);
                  }
                  break;
            case WM_CREATE:
                  hwndCB = CommandBar_Create(hInst, hWnd, 1);

                  CommandBar_InsertMenubar(hwndCB, hInst,
IDM_MENU, 0);
                  CommandBar_AddAdornments(hwndCB, 0, 0);
                  break;
            case WM_PAINT:
                  RECT rt;
                  hdc = BeginPaint(hWnd, &ps);
                  GetClientRect(hWnd, &rt);
                  rt.top = rt.top + 30;               // make
room for command bar


                  if(errorPresent) {
```

```
                              DrawText(hdc, errorMessage,
_tcslen(errorMessage), &rt, DT_LEFT | DT_WORDBREAK);
                              errorPresent = FALSE;
                    } else {

                              // NOTHING IS YET RUNNING
                              if(running == NONE) {
                                   swprintf(szOut, TEXT("Select the
desired settings for your capture method from the \"Settings\"
menu,"));
                                   DrawText(hdc, szOut,
_tcslen(szOut), &rt, DT_LEFT | DT_WORDBREAK);
                                   rt.top = rt.top + 15;
                                   swprintf(szOut, TEXT("then
select the desired capture method from the \"Run\" menu."));
                                   DrawText(hdc, szOut,
_tcslen(szOut), &rt, DT_LEFT | DT_WORDBREAK);

                              // OS COLLECTION IS RUNNING
                              } else if (running == OS) {
                                   if(!dataCollected) {
                                        swprintf(szOut,
TEXT("Collecting data from OS."));
                                        DrawText(hdc, szOut,
_tcslen(szOut), &rt, DT_LEFT | DT_WORDBREAK);

     OsRandInitialize(outFileName, 0, dataCollectionAmount);
                                        OsRandStart();
                                        dataCollected = TRUE;
                                        InvalidateRect(g_hWnd,
&rt, TRUE);
                                   } else {
                                        OsRandClose();
                                        swprintf(szOut, TEXT("Data
from OS has been collected."));
                                        DrawText(hdc, szOut,
_tcslen(szOut), &rt, DT_LEFT | DT_WORDBREAK);
                                   }

                              // IR COLLECTION IS RUNNING
                              } else if (running == IR) {
                                   if(!dataCollected) {
                                        swprintf(szOut,
TEXT("Collecting data from IR device."));
                                        DrawText(hdc, szOut,
_tcslen(szOut), &rt, DT_LEFT | DT_WORDBREAK);


     IrRandInitialize(outFileName, dataCollectionAmount,
(DWORD) LoggerFunction);
                                        IrRandStart();
```

```
                                    dataCollected = TRUE;
                                    InvalidateRect(g_hWnd,
&rt, TRUE);
                              } else {
                                    IrRandClose();
                                    swprintf(szOut, TEXT("IR
data has been collected."));
                                    DrawText(hdc, szOut,
_tcslen(szOut), &rt, DT_LEFT | DT_WORDBREAK);
                              }

                        // TS COLLECTION IS RUNNING
                        } else if (running == TRACK) {
                              if(!dataCollected) {
                                    if(dataCount == 0)
                                          myError =
FileWriterInitialize(outFileName);

                                    if(!myError) {
                                          swprintf(szOut,
TEXT("COULD NOT INIT OUTFILE"));
                                    } else {
                                          swprintf(szOut,
TEXT("Scribble as randomly as possible."));
                                    }
                                    DrawText(hdc, szOut,
_tcslen(szOut), &rt, DT_LEFT | DT_WORDBREAK);
                              } else {
                                    FileWriterClose();
                                    swprintf(szOut,
TEXT("Touchscreen data collected."));
                                    DrawText(hdc, szOut,
_tcslen(szOut), &rt, DT_LEFT | DT_WORDBREAK);
                              }

                        // AUDIO COLLECTION IS RUNNING
                        } else if (running == AUDIO) {
                              if(!dataCollected) {
                                    swprintf(szOut,
TEXT("Collecting audio data"));
                                    DrawText(hdc, szOut,
_tcslen(szOut), &rt, DT_LEFT | DT_WORDBREAK);
                              } else {


      GetExitCodeThread(&threadId,&threadExitCode);

      ExitThread(threadExitCode);
```

```
                                    swprintf(szOut, TEXT("Data
has been collected."));
                                    DrawText(hdc, szOut,
_tcslen(szOut), &rt, DT_LEFT | DT_WORDBREAK);
                              }
                        }
                  }
                  EndPaint(hWnd, &ps);
                  break;
            case WM_LBUTTONDOWN:
            case WM_MOUSEMOVE:
                  if(running == TRACK) {
                        if(!dataCollected) {
                              DoMouseMain(hWnd, message,
wParam, lParam);
                        }
                  }
                  break;
            case WM_DESTROY:
                  CommandBar_Destroy(hwndCB);
                  PostQuitMessage(0);
                  break;
            default:
                  return DefWindowProc(hWnd, message, wParam,
lParam);
   }
   return 0;
}

// Mesage handler for the About box.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam,
LPARAM lParam)
{
      RECT rt, rtl;
      int DlgWidth, DlgHeight;      // dialog width and height
in pixel units
      int NewPosX, NewPosY;

      switch (message)
      {
            case WM_INITDIALOG:
                  // trying to center the About dialog
                  if (GetWindowRect(hDlg, &rtl)) {
                        GetClientRect(GetParent(hDlg), &rt);
                        DlgWidth    = rtl.right - rtl.left;
                        DlgHeight   = rtl.bottom - rtl.top ;
                        NewPosX           = (rt.right -
rt.left - DlgWidth)/2;
                        NewPosY           = (rt.bottom -
rt.top - DlgHeight)/2;
```

```
                        // if the About box is larger than the
physical screen
                        if (NewPosX < 0) NewPosX = 0;
                        if (NewPosY < 0) NewPosY = 0;
                        SetWindowPos(hDlg, 0, NewPosX,
NewPosY,
                              0, 0, SWP_NOZORDER |
SWP_NOSIZE);
                  }
                  return TRUE;

            case WM_COMMAND:
                  if ((LOWORD(wParam) == IDOK) ||
(LOWORD(wParam) == IDCANCEL))
                  {
                        EndDialog(hDlg, LOWORD(wParam));
                        return TRUE;
                  }
                  break;
      }
    return FALSE;
}


//
//  FUNCTION: AudioCallback
//
//  PURPOSE:  Receives the callback when the WAVEHDR object
//            is full and ready to be read.
//
void AudioCallback(void) {
      dataCollected = TRUE;
      SoundRecorderClose();
      RECT rt;
      GetClientRect(g_hWnd, &rt);
      InvalidateRect(g_hWnd, &rt, TRUE);
}




/**
 * FUNCTION:      DoMouseMain
 * DATE:          April 2003
 * PURPOSE:       Handle touchscreen events for pen tracking.
 * PARAMS:
 *                      hWnd --
 *                      wMsg --
 *                      wParam --
 *                      lParam --
 *
 * RETURN:
 *                      LRESULT
 *
 * NOTES:          Adapted from code found in Douglas Boling's
book
 *                      "Programming Microsoft Windows CE"
(2nd Ed.).
 */
LRESULT DoMouseMain (HWND hWnd, UINT wMsg, WPARAM wParam,
LPARAM lParam) {
      POINT ptM;
      UINT uPoints = 0;
      HDC hdc;

      CHAR temp;

      ptM.x = LOWORD(lParam);
      ptM.y = HIWORD(lParam);

      if(dataCount != dataCollectionAmount) {
            temp = (ptM.x & 0x00FF) ^ (ptM.y & 0x00FF);
      // collect the xor of low bytes.
            FileWriterWrite(temp);
            dataCount++;
      } else {
            dataCollected = TRUE;
            dataCount = 0;
            RECT rt;
            GetClientRect(g_hWnd, &rt);
            InvalidateRect(g_hWnd, &rt, TRUE);
      }

      hdc = GetDC(hWnd);

      SetPixel (hdc, ptM.x, ptM.y, RGB(0,0,0));
      SetPixel (hdc, ptM.x+1, ptM.y, RGB(0,0,0));
      SetPixel (hdc, ptM.x, ptM.y+1, RGB(0,0,0));
      SetPixel (hdc, ptM.x+1, ptM.y+1, RGB(0,0,0));

      ReleaseDC(hWnd, hdc);

      Sleep(tsDelay);

      return 0;
}
```

```
/**
 * FUNCTION:      PackSeedData
 * DATE:          April 2003
 * PURPOSE:       Take one bit of entropy collected and pack
these
 *                      bits into words for use in PRNG.
 * PARAMS:
 *                      NONE
 * RETURN:
 *                      NONE
 * NOTE:          Assumes that the raw seed data has already
been
 *                      collected.
 */
void PackSeedData() {
      int i, j, k, temp;
      int mask = 0x00000001;

      prngData.c_len = 60;
      prngData.t_len = 20;

      for(i = 0; i < 60; i++) {
            for(j = i*8; j < (i+1)*8; j++) {
                  temp = seedBuffer[j] & mask;
                  prngData.c[i] = prngData.c[i] | temp;
                  prngData.c[i] <<= 1;
            }
      }


      // i now equals 60
      for(i; i < 80; i++) {
            for(k = 0, j = i*8; j < (i+1)*8; j++, k++) {
                  temp = seedBuffer[j] & mask;
                  prngData.t[k] = prngData.t[k] | temp;
                  prngData.t[k] <<= 1;
            }
      }

}




/**
 * FUNCTION:      LoggerFunction
 * DATE:          April 2003
 * PURPOSE:      Allows other components to "write back" to
this
 *                        component and then prints on the
screen.
 * PARAMS:
 *                        message -- a pointer to the message
(null term string)
 * RETURN:
 *                        NONE
 */
void LoggerFunction(LPTSTR message) {
      swprintf(errorMessage, message);
      errorPresent = TRUE;
      RECT rt;
      GetClientRect(g_hWnd, &rt);
      InvalidateRect(g_hWnd, &rt, TRUE);
}



DWORD WINAPI AudioPowerManageSubversion(LPVOID lpData) {
      while(!dataCollected) {
            SystemIdleTimerReset();
            Sleep(2000);
      }
      return ERROR_SUCCESS;
}
```

Linux Source

Deskew.c

```
/**
 *   DESKEW.C
 *
 *   AUTHOR:     A. W. Montville
 *   VERSION:    1.0
 *
 *   Given an input file which has entropy bits collected,
i.e., the lowest
 *   bit from collected raw bytes, the output file will
contain deskewed information.
 *
 *   The input data (read from the input file supplied at the
command line) is
 *   parsed in bit pairs starting with the first byte of the
file and moving toward
 *   the last byte of the file.  The first bit of the pair, a,
is compared to the
```

```
 *   second bit of the pair, b.  If they are the same, both
bits are discarded, and
 *   the algorithm moves on to the next bit pair.  If they are
different, then the
 *   bit a is kept as the output bit of the generator which
produced the input data.
 *
 *   This method is described in "Handbook of Applied
Cryptography", by A. Menezes,
 *   et. al. (see page 173).
 *
 *   Copyright (c) 2003 A. W. Montville
 */
 #include <fcntl.h>
 #include <stdio.h>
 #include <stdlib.h>
 #include <sys/stat.h>
 #include <sys/types.h>
 #include <unistd.h>


#define MAX_DESKEW_BUFFER_SIZE      1048576            // 1
Meg.

void DeskewUsage(void);

int main(int argc, char* argv[]) {

      char                 *inFilePath = argv[1];
      char                 *outFilePath = argv[2];
      unsigned char        readBuffer[MAX_DESKEW_BUFFER_SIZE];
      unsigned char        writeBuffer[MAX_DESKEW_BUFFER_SIZE];
      int                  eofFlag = 0;

      unsigned char        temp[1];
      unsigned char        a;
      unsigned char        b;

      mode_t                     mode;

      int                  inFile;                          //
handle to input file
      int                  outFile;                   // handle to
output file
      int                  count;                           //
counts bytes read from input file
      int                  packCount;                 // tracks bits
that have been packed
      int                  idx;                       // index for
output buffer
```

```
	int			i;				//
byte loop index
	int			j;				//
bit loop index


	unsigned char maskArray[8] = {	0x01,		// 0000
0001	[0]
							0x02,
		// 0000 0010	[1]
							0x04,
		// 0000 0100	[2]
							0x08,
		// 0000 1000	[3]
							0x10,
		// 0001 0000	[4]
							0x20,
		// 0010 0000	[5]
							0x40,
		// 0100 0000	[6]
							0x80
	// 1000 0000	[7]
						};


	// Check that enough arguments were supplied, otherwise
print usage
	if(argc >= 3) {

		mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
S_IROTH;

		inFile = open(inFilePath, O_RDONLY, mode);
		outFile = open(outFilePath, O_WRONLY | O_EXCL |
O_CREAT, mode);

		// Detect error opening input file
		if(inFile == -1) {
			perror("inFile");
			return 1;
		}

		// Detect error opening output file
		if(outFile == -1) {
			perror("outFile");
			return 1;
		}


		packCount = 0;
```

```
            #ifdef DEBUG
                  printf("Starting process...\n");
            #endif

            while(1) {
                  count = 0;
                  i = 0;
                  idx = 0;

                  #ifdef DEBUG
                        printf("Reading Input file: %s",
inFilePath);
                  #endif

                  do {
                        count = read(inFile, temp, 1);
                                    // Read one byte from the
file
                        if(count == 0) {
                                    // If byte not read (or
error), then

                                                      //    set EOF
flag to indicate such.
                              eofFlag = 1;

                              #ifdef DEBUG
                                    printf("count == EOF\n");
                              #endif
                        }


                        #ifdef VERBOSE_DEBUG
                              printf("readBuffer[%d] being
assigned\n", i);
                        #endif

                        if(eofFlag)
                                          // If EOF flag is
set, then break out
                              break;
                                                      //    of the
loop prior to writing.

                        readBuffer[i] = temp[0];
                        i++;
                  } while (i < MAX_DESKEW_BUFFER_SIZE);

                  count = i;
                  #ifdef DEBUG
                        printf("%d bytes read so far, writing
to output file...\n", count);
                  #endif

                  for(i = 0; i < count; i++) {

                        for(j = 7; j >= 1; j -= 2) {
                              a = maskArray[j] &
readBuffer[i];
                              b = maskArray[j-1] &
readBuffer[i];


                              // If a is not equal to zero and
b is zero, or if a is
                              // equal to zero and b is not,
then we have different
                              // bits and should collect the
entropy.
                              if( (a && !b) || (!a && b) ) {

                                    // If a is not equal to
zero, then the bit was set and
                                    // a 1 is written to the
lsb of writeBuffer[idx]; otherwise
                                    // a is zero, and the
buffer remains unchanged.
                                    if(a) {
                                          writeBuffer[idx] |=
0x1;
                                    }


                                    // Keep track of how many
bits have been packed into
                                    // the character.  If we
have been through to 6, then
                                    // the last shift takes
place.  Otherwise, the index is
                                    // incremented and the
packCount variable is reset.
                                    if(packCount < 7) {
                                          packCount++;
                                          writeBuffer[idx] <<=
1;
                                    } else {
                                          packCount = 0;
                                          idx++;
                                    }
                        }
                    }
                }


                write(outFile, writeBuffer, idx-1);


                if(eofFlag)
                    break;
            }

            close(inFile);
            close(outFile);

        } else {
            DeskewUsage();
        }

        return 0;
    }


void DeskewUsage() {
        printf("\n\n");
        printf("*** USAGE ***\n");
        printf("   deskew <infile> <outfile>\n");
        printf("\n\n");
}
```

collect_entropy.c

```
/**
 *   COLLECT_ENTROPY.C
 *
 *   AUTHOR:    A. W. Montville
 *   VERSION:   1.0
 *
 *   Given an input file, this creates an output file
containing 1 bit of
 *   data for every 8 bits collected.  The least-significant
bit is collected.
 *
 *   Copyright (c) 2003 A. W. Montville
 */
 #include <fcntl.h>
 #include <stdio.h>
 #include <stdlib.h>
 #include <sys/stat.h>
 #include <sys/types.h>
 #include <unistd.h>

void CollectEntropyUsage(void);

int main(int argc, char* argv[]) {

      char* inFilePath = argv[1];
      char* outFilePath = argv[2];
      char temp[1];
      char toWrite[1];
      mode_t mode;
      int inFile, outFile;
      int count = 0;
      char mask = 0x0001;


      // Check that enough arguments were supplied, otherwise
print usage
      if(argc >= 3) {

            mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
S_IROTH;
            inFile = open(inFilePath, O_RDONLY, mode);
            outFile = open(outFilePath, O_WRONLY | O_EXCL |
O_CREAT, mode);

            if(inFile == -1) {
                  perror("inFile");
                  return 1;
            }

            if(outFile == -1) {
                  perror("outFile");
                  return 1;
            }

            count++;
            while(read(inFile, temp, 1) > 0) {
                  toWrite[0] = toWrite[0] | (temp[0] & mask);
                  if(count != 8) {
                        toWrite[0] = toWrite[0] << 1;
                        count++;
                  } else {
                        count = 1;
                        write(outFile, toWrite, 1);
                  }
            }



            close(inFile);
            close(outFile);
        } else {
            CollectEntropyUsage();
        }

        return 0;
  }


void CollectEntropyUsage() {
        printf("\n\n");
        printf("*** USAGE ***\n");
        printf("   colent <infile> <outfile>\n");
        printf("\n\n");
}
```

trunc_files.c

```
/**
 *   TRUNC_FILES.C
 *
 *   AUTHOR:    A. W. Montville
 *   VERSION:   1.0
 *
 *   Given an input file, this will truncate the file to a
 *   user-defined byte length
 *
 *   Copyright (c) 2003 A. W. Montville
 */
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

#ifndef TRUNC_FILE_MAX_BUFFER_SIZE
#define TRUNC_FILE_MAX_BUFFER_SIZE  2097152
      // 2 MB buffer
#endif



// Function Prototypes
void TruncFileUsage(void);

// Main program entry
int main(int argc, char* argv[]) {
```

```
char *inFilePath = argv[1];
char *outFilePath = argv[2];
char temp[1];
char buffer[TRUNC_FILE_MAX_BUFFER_SIZE];
int inFile, outFile;
int i = 0;
int count = 0;
int amtRead = 0;
int truncAmt = 0;
mode_t mode;


// Check that enough arguments were supplied, otherwise
// print usage
if(argc >= 4) {

      truncAmt = atoi(argv[3]);

      // Open file handles
      mode = S_IRUSR | S_IWUSR | S_IRGRP | \\
             S_IWGRP | S_IROTH;
      inFile = open(inFilePath, O_RDONLY, mode);
      outFile = open(outFilePath, O_WRONLY | \\
                                  O_CREAT, mode);

      // check inFile
      if(inFile == -1) {
            perror("inFile");
            return 1;
      }

      // check outFile
      if(outFile == -1) {
            perror("outFile");
            return 1;
      }


      // Read the entire file into buffer[]
      while((amtRead = read(inFile, temp, 1)) > 0) {
            buffer[i] = temp[0];
            count += amtRead;
            i++;
      }

      // Return an error if the input file contains
      // fewer
      // bytes than the truncation amount.
      if(count < truncAmt) {
            perror("inFile too small");
            return 1;
            }

            // Write truncated amount to outFile
            write(outFile, buffer,truncAmt);

            // Close file streams
            close(inFile);
            close(outFile);

      } else {

            // Display proper usage
            TruncFileUsage();
      }

      return 0;
 }


void TruncFileUsage() {
      printf("\n\n");
      printf("*** USAGE ***\n");
      printf("   truncfile <infile> <outfile_prefix> \\
                 <truncation_amount_in_bytes>\n");
      printf("   NOTE: The last argument to truncfile \\
                 cannot be checked for errors\n");
      printf("         therefore the output file size \\
                 must be checked prior to use.\n");
      printf("\n\n");
}
```