Evaluating a GPU based TRNG in an entropy starved virtual Linux environment

by

Christopher Plesiuk

A thesis submitted to

The Faculty of Graduate Studies of

The Univeristy of Manitoba

in partial fulfillment of the requirements

of the degree of

Master of Science

Department of Computer Science

The University of Manitoba

Winnipeg, Manitoba, Canada

May 2016

Thesis advisor

Dr. Parimala Thulasiraman

Author

Christopher Plesiuk

Evaluating a GPU based TRNG in an entropy starved virtual Linux environment

Abstract

A secure system requires cryptography and effective cryptography requires high quality system entropy. Within a virtualized Linux environment the quality and the amount of system entropy can be over overestimated. These virtualized environments can also have difficulty generating entropy data.

To address the problems with entropy in virtualized Linux environments, my thesis investigates and evaluates exposing a unique true random number generator via an entropy-sharing tool called Entropy Broker. Entropy Broker distributes entropy data generated by the true random number generator to several virtualized Linux guest systems to increase the entropy of each system and in turn, increase the security of their cryptographic libraries.

Entropy Broker and the true random number generator are evaluated against the Linux pseudo random number generator, the Haveged pseudo random number generator, and an on chip random number generator developed by Intel.

Contents

List of Figures

List of Tables

Acknowledgments

Thank you to my advisor, Dr. Parimala Thulasiraman, Mr. Jose Mijares Chan, and Mr. Gilbert Detillieux for their support and assistance while I worked on my thesis.

I would also like to thank my committee, Dr. Peter Graham and Dr. Paul Card, for their comments and suggestions.

A special thanks is needed for Dr. John Anderson. Without his his support behind the scenes and words of encouragement this would not have been possible.

Finally, I would especially like to thank my wife, Kori, and our three children, Eli, Sophie, and Clara. Without their encouragement and love, I would have not been able to complete my thesis.

Chapter 1

Introduction

In computer science, entropy is a measurement of randomness. The more uncertain or random a piece of information is, the better or higher the entropy. In Linux environments, entropy is collected from interrupt timings, keyboard events, mouse events, file system activity, and network interrupts. Everspaugh et al. [7] explains that this randomness is exposed in two locations within the file system in Linux, /dev/random and /dev/urandom. The entropy collected in these two locations is exposed to the operating system as random numbers or entropy data. The entropy data is used in multiple cryptographic protocols to secure the system such as public/private key-pairs, salts, padding, and Transport Layer Security (TLS).

Within a virtualized Linux environment, there exists problems with entropy data generation. The primary problem is that each virtualized guest has slow entropy data generation and will overestimate the amount and quality of the entropy data that it has. This problem is due to three issues as documented by John Mechalas from Intel [16]:

- A server hosting virtualized guests in a data centre will not have any keyboard or mouse events limiting the amount of entropy collected.
- Hypervisors virtualize hardware interrupts so entropy will not be collected from those

sources.

- Entropy is shared among several virtualized guests. Each guest does not know the true entropy of the host and each guest assumes that it has full system entropy.

Kerrigan and Chen [13] reinforce this by stating that "..the sum of the entropy of the virtual machines cannot exceed the entropy of the physical host." Overestimating the amount and quality of entropy poses a serious security issue. There are two concerns:

- As pointed out by Smith [22], if there is a problem with the amount of entropy data being generated in the system, applications will typically fail or wait until there is more entropy data available which could take a significant amount of time.

- Explained by Grobauer et al. [8], if there is not enough entropy data and guests do not have an accurate picture of the quality of that entropy, there may be cryptographic vulnerabilities that crop up due to the weak seed of the Linux random number generator.

While I am unaware of any actual exploited vulnerabilities, at least two conceptual attacks were presented by Kerrigan and Chen [13]. The first conceptual attack is a random number pool depletion attack; the second is an entropy data pool poisoning attack. Kerrigan and Chen provided no additional details on the first attack beside from their brief description. The second attack however was explained in greater detail. An entropy data pool poisoning attack attempts to drain the entropy data from a virtual guest and then makes contributions to the entropy data pool by generating network interrupts. As there are so few sources of entropy data generation in a virtualized Linux environment, an attacker can safely assume that network interrupts are one of the only sources of entropy data generation on the system. The attackers goal is to gain knowledge of the virtual guests entropy data pool and use that knowledge to compromise the secure network traffic coming from the guest.

1.1 Contribution:

My thesis investigated the problem of limited entropy data generation within a virtualized Linux environment by utilizing a modification of Chan et al.'s [3] proposed low cost true random number generator (TRNG). The TRNG utilized sources of uncertainty in graphical processing units (GPU) to generate entropy data or random numbers. This entropy data was then exposed to virtualized Linux guests by using a tool developed by Folkert van Heusden called Entropy Broker [10]. As part of my thesis, I wrote a custom server application, using the API's provided by Entropy Broker, to transmit the entropy data generated by Chan's GPU TRNG to virtualized Linux guests.

Entropy Broker and Chan's GPU TRNG were evaluated by reviewing the statistics that the algorithm generates and using the tools *ent* [24] and *rngtest* [17] to test the quality of the entropy data generated from the system. The throughput of the system has also been measured and evaluated against Intel's digital random number generator (DRNG) [16], Haveged's pseudo random number generator (PRNG) [11], and the Linux built in PRNG. Due to hardware constraints available at the University of Manitoba, the system has only been tested with up to four virtualized Linux guests running on a single host.

Chapter 2

Related Work

Cryptography is used to facilitate secure communications between computer systems. High quality random numbers or entropy data is important for implementing secure cryptographic communications.

Schindler and Killmann [21] describe an ideal random number generator as one whose numbers are "independent and uniformly distributed on a finite range". They continue saying that while this is technically impossible, there are three real world subclasses for random number generation: true random number generators (TRNG), pseudo random number generators (PRNG), and hybrid random number generators (HRNG). My thesis is focused on TRNGs which traditionally have two advantages over PRNG and HRNG as pointed out by Chan et al. [3]. These advantages are that TRNGs are inherently unpredictable (which offers better randomness than a PRNG), and that TRNGs do not have any periodicity or data dependencies and there are requirements for good cryptography.

All TRNGs are based on random microcosmic processes. These processes can cause physical effects measured as random noise. Schindler and Killmann [21] provided examples of these as being things such as "... quantum random processes, time between emissions during radioactive decay, inherent semiconductor thermal noise, shot noise from Zener diodes or free

running oscillators." Some work that has been done to develop good physical noise sources include Bagini and Bucci's [1] work on a reliable true random number generator, Maher and Rance's [14] work on random number generators founded on signal and information theory, and the Intel random number generator [12]. The traditional problem with TRNGs, and the reason why they are not wide spread, is accurately pointed out by Chan et al. [3] in that they are usually more expensive, less flexible, and can be slow at generating random numbers. Due to these properties, pseudo random number generators are usually preferred.

2.1 The Linux Pseudo Random Number Generator

Jun and Kocher [12] state "A pseudo random number generator by itself will be insecure without a true random number generator for seeding." They continue to explain that within a deterministic system it is impossible to create a true random source. The Linux answer to secure computing is the Linux random number generator (LRNG) [6, 15]. The LRNG is an open sourced pseudo random number generator project led by Linus Torvalds. The LRNG collects interrupt timings from various sources such as keyboard, mouse, disk read and writes, and network interrupts as described by Gutterman et al. [9] and Kerrigan and Chen [13]. While the study by Gutterman et al. on the LRNG concluded that there were several issues with the LRNG, the overall analysis of the LRNG by one of the authors (Reinman [20]) is that it is "a strong cryptography mechanism for producing random numbers."

Everspaugh et al. [7] state that most virtualized Linux environments usually rely on the internal LRNG including Xen, VMware, and Amazon's EC2. These random numbers are required for cryptographic functions of the operating system. John Mechalas [16] however points out some interesting problems with this. The first is that in a data centre with multiple virtual machines and servers, most systems will not have a keyboard or mouse connected to them. This is important to note as the LRNG relies in part, on these inputs for

entropy data generation. The second issue is that most disk reads, disk writes and network interrupts are virtualized by the hypervisor. This also removes or reduces a source of entropy data generation. Finally, the remaining generated entropy data is shared among the virtual machines. Since these systems do not have a full picture of the actual total entropy coming from the hypervisor or host system, each virtual machine assumes it knows the total system entropy.

2.2 Limited Entropy

There have been some proposed solutions to the problem of reduced entropy data generation in virtualized Linux environments. Examples of this are Intel's Secure Key [16], the Intel Random Number Generator [12], Simtec Electronics Entropy Key [22, 25], the Whirlwind RNG [7], rngd [18], and Haveged [11]. Other options vary based on the operating system and can include such capabilities as /dev/urandom (low quality, infinite entropy) and /dev/arandom (based on the ARC4 algorithm in OpenBSD) [5, 19]. Each method however does not necessarily solve the problem as it either exposes the hypervisors entropy data to its guests, provides sub-par randomness, or cannot be securely delivered to a guest operating system.

2.3 Addressing Limited Entropy Data Generation in a Virtualized Linux Environment

In an effort to take advantage of the positive properties of TRNG and to address the problem with the lack of entropy data generation in virtual Linux environments, my thesis used Chan et al.'s [3] unique TRNG along with a tool called Entropy Broker [10]. Chan et al.'s GPU TRNG used an NVidia graphics card as its physical source for randomness. From the GPU

two digital noise sources are used:

- Timing race conditions that occur within the GPU.
- Samples of the GPU core temperature.

These digital noise sources are then run through a post-processing algorithm. The first post-process is data compression, the second post-process is a basic mapping function that uses modulus, and the third and final post-process takes the generated numbers and applies histogram equalization or exact histogram equalization to obtain a uniform distribution of random numbers.

Entropy Broker [10] is a tool that can be used to distribute entropy data from one location to several others. It uses a client-server architecture to distribute entropy data. The server component of Entropy Broker can reside on the host OS box itself or on any other device. For my thesis, the server component of Entropy Broker resides on a system separate from the system that has the GPU. By combining Chan's unique TRNG and the functionality of Entropy Broker, the problem of reduced entropy data generation on virtualized Linux system is alleviated.

To the best of my knowledge, no one else has used Chan's GPU TRNG to improve the entropy data generation in virtual machines or in combination with Entropy Broker to provide entropy data to virtual machines.

Chapter 3

Entropy Broker

In this chapter, I describe Entropy Broker and its various components in detail. Entropy Broker is used in my thesis to distribute the entropy data generated from Chan's GPU TRNG.

Entropy Broker [10] is a client-server application developed by Folkert van Heusden. It allows the distribution of entropy data to /dev/random on Linux operating systems or it can output entropy data to a file. Entropy Broker consists of three components: the core application, server processes used to generate entropy data, and client processes used to consume entropy data.

Figure 3.1 shows an example of Entropy Broker's architecture.

3.1 Entropy Broker - Core Application

Entropy Broker's core application collects and distributes entropy data from server and client processes. It maintains several pools of entropy data both in memory and on disk. Each pool is 4KB in size.

According to Crocker and Schiller's [4] Randomness Requirements for Security (RFC

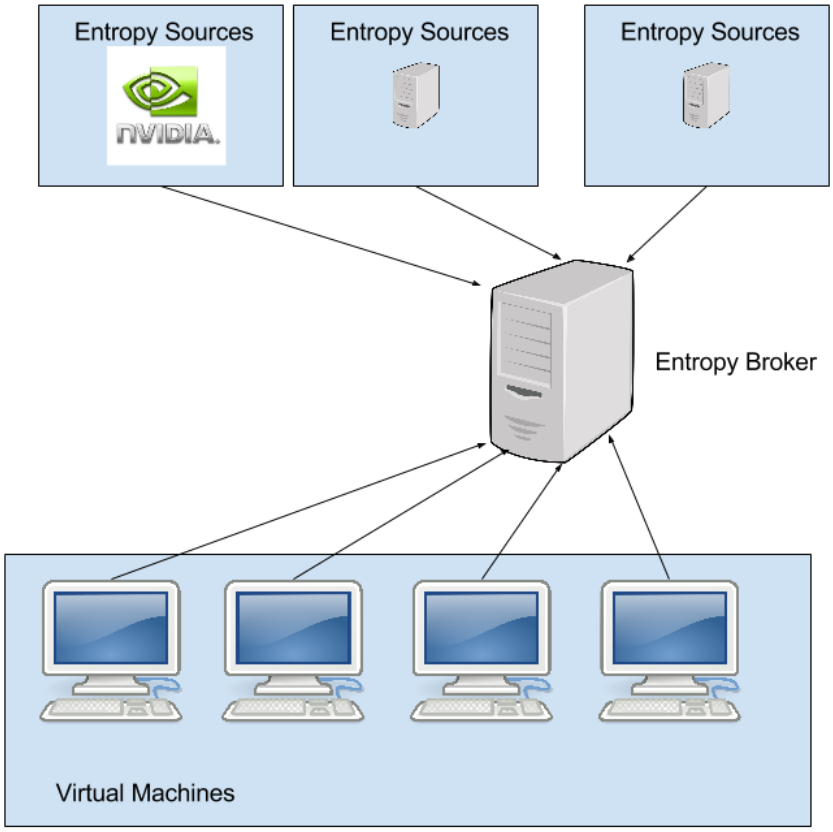

Figure 3.1: Entropy Broker Architecture

4086), one way to produce a strong sequence of randomness or high quality entropy data is to take an encryption algorithm, such as AES in the case of Entropy Broker, and use a random key to encrypt data. Crocker and Shiller call this method *stirring*. Entropy Broker uses stirring with the pools of entropy data by using the information retrieved from server processes as the random key and encrypting pools of entropy data using AES.

When entropy data is requested from client processes, the core application hashes its pool files using SHA512 and then 'un-stirs' the files by decrypting the pool file with the hash as the key. The hashed value is folded in half, and then returned as entropy data to the client process.

When sending or receiving entropy data, the local area network is used. The network stream is encrypted using 3DES. The data sent is hashed using MD5 in order for the client process to verify that it was not altered in transit. SHA512 is used to authenticate the handshake between Entropy Broker and its processes.

3.2 Server Process

An Entropy Broker server process is an Entropy Broker component that is used to produce entropy data and send it to the main Entropy Broker application. Entropy Broker has several built-in server processes such as:

Timers measures the jitter between two timer-frequencies.

Audio measures the noise in an audio-stream.

Video measures the noise in a video signal.

USB measures the difference between the local system clock and a USB device clock.

For the purpose of my thesis, a custom server process was created that uses a modified

version of Chan's GPU TRNG. This modified version of Chan's GPU TRNG is described in some detail in the next chapter.

3.3 Client Process

Client processes are an Entropy Broker component that consumes entropy data. There are two distinct client processes built into Entropy Broker:

- the first type of client process consumes entropy data and writes it to a user specified file. The user is able to determine how much entropy data should be written before the process stops by specifying the number of bytes.

- the second type of client process writes to the Linux operating system's entropy data pool at /dev/random until the user ends the process. Writing to /dev/random is only done when the Linux operating system's entropy data pool reaches a minimum threshold. The application knows that this threshold has been met when /dev/random becomes writable. For most Linux operating system's this occurs when there is less than 128 bits left in its entropy data pool.

Chapter 4

Combining GPU TRNG with Entropy Broker - Implementation and Evaluation

In this chapter, I first describe Chan's GPU TRNG and the modifications made to it in order to evaluate it and make it work with Entropy Broker. I then explain the experiments and the evaluation performed to evaluate the GPU TRNG. The chapter concludes with a summary of the evaluation.

4.1 Chan's GPU TRNG

Chan's GPU TRNG required several modifications function with Entropy Broker and to allow testing of the algorithm's performance. See appendix A for the custom Entropy Broker server process code.

In order to test the algorithm, three variables are defined that affect the GPU TRNG race condition and the random numbers generated:

Max Iterations Used in the race condition to determine how many times the race condition should occur.

Max Repetitions Used to determine how much data is generated at once.

Number of Loops This value was only set for the initial testing. It determined how many times the algorithm should run.

Two variables were not modified and stayed set at their default values:

Size how many threads should take part in the race condition.

Compression Level the level of compression that should be applied to each value returned from the race condition timing.

In addition to the modifications, a problem was identified when trying to use Chan's GPU TRNG with Entropy Broker. Chan's GPU TRNG generates a vector of floats when completed, however Entropy Broker works with bytes. When floats are interpreted as groups of bytes, the entropy of the data is significantly lower due to the large amount of repetition in the binary representation of the floats.

To compensate for the issue of interpreting floats, two methods were selected. The methods were an XOR function and a linear transform. The XOR function broke each float into four sets of bytes and XOR'd the results. The linear transform maps each float to a single byte by multiplying the float by the max value of a byte and then rounding the result. Both methods were evaluated as part of testing Chan's GPU TRNG.

The linear transform was considered, as it is an obvious candidate for mapping a series of large numbers to a series of smaller numbers. XOR was chosen based on a paper by Suresh and Burleson [23] in which they explain that XOR can be used as a simple entropy extractor.

Chan's GPU TRNG was also modified to disable the exact histogram equalization portion of the algorithm to test the performance and the quality of generation.

Finally, the algorithm was wrapped in a custom Entropy Broker server interface. This interface allowed Chan's GPU TRNG to connect to the Entropy Broker application and provide it with entropy data. The custom interface included the following components:

- displaying the performance of the entropy data generation
- writing entropy data to a file
- connecting to an external Entropy Broker server

The experiments performed and the evaluation of the performance and the quality of the entropy data generated by the modified algorithm are presented in the next section.

4.2 First Phase of GPU TRNG Experiments

The first group of experiments focused on testing Chan's GPU TRNG in various configurations to determine two things:

1. how the GPU TRNG performs with different settings
2. the amount of entropy in the data generated

Secondary testing was also performed to evaluate the quality of the entropy data generated. The performance, not the quality of the entropy data is the primary focus of my thesis.

Once Chan's GPU TRNG was modified to allow for testing, the experimentation took part in two phases. The first phase of the experiments were designed and ran with one-hundred different configurations. The configuration options were as follows:

Max Repetitions 256, 512, 1024, 2,048, 4,096

Max Iterations 4,096, 8,192, 16,384, 32,768, 65,536

Exact Histogram Equalization Enabled Yes/No

XOR or Linear Transform XOR/Linear

Each experiment combination was run ten times to generate a decent sample size of entropy data to evaluate. Each individual test was configured to generate 64KB of experimental data.

The results of these tests were parsed, aggregated, and evaluated. Three methods were used to evaluate the entropy data. The first method utilized metrics built into Entropy Broker that output timing and the amount of data written. The second and third methods used RNGTEST and ENT to run through the generated entropy data. RNGTEST and ENT evaluated the data at the binary level and were used to evaluate the quality of entropy data generated.

4.2.1 RNGTEST

RNGTEST reviews the data in chunks of 20,000 bits at a time using FIPS 140-2 [2] tests. The following tests were used:

Monobit

Review the number of ones and zeroes within the chunk of data and see if the values falls within an expected interval. If it does not, it fails the test.

- *x:* the number of ones in the bit stream.
- *FIPS 140-2 Acceptance Range:* $9,725 < x < 10,275$

Poker

Divide the 20,000 bits into four 5,000 bit segments. Walk through the data four bits at a time. The total number of possible unique four bit segments is $2^4 = 16$. For each

5,000 bit segments, count the number of each pattern that occurs. The number of occurrences then must fall within an expected interval. If it does not, it fails the test.

- Let g_i be the number of occurrences of each pattern where i represents each unique pattern.
- *x:* $(16/5000) \times \sum_{i=0}^{15} g_i^2 - 5000$
- *FIPS 140-2 Acceptance Range:* $2.16 < x < 46.17$

Runs

A run is a sequence of consecutive values of ones or zeroes. Count the number of runs and determine if the number of runs of length one through six or higher fall within the expected interval. If it does not, it fails the test.

- *x:* the number of runs of each length that appear in the bit stream
- *FIPS 140-2 Acceptance Range:* Table 4.1 provides the acceptance ranges.

Length of Run	FIPS 140-2 Acceptance Range
1	$2,315 \leq x \leq 2,685$
2	$1,114 \leq x \leq 1,386$
3	$527 \leq x \leq 723$
4	$240 \leq x \leq 384$
5	$103 \leq x \leq 209$
6+	$103 \leq x \leq 209$

Table 4.1: FIPS 140-2 Acceptance Range for the Runs RNGTEST

Long Run

A simple test which is passed as long as there is no run longer than twenty six consecutive ones or zeroes.

- *x:* the length of the longest run.
- *FIPS 140-2 Acceptance Range:* $x < 26$

4.2.2 ENT

The following built in ENT tests were used to evaluate the experiments entropy data:

Entropy

Entropy of the data is calculated by attempting to losslessly compress the data that ENT is evaluating. The closer to zero percent compression, the higher the entropy. The value that ENT presents is between zero and eight. This expresses the information density as bits per character.

Mean

All of the bytes in the file are added, and then divided by the length of the file. The closer to 127.5, the more random the sequence.

Monte-Carlo Pi

The entropy data is used to calculate PI using a Monte-Carlo simulation. The closer to PI, the better the quality of entropy data.

Serial Correlation

This test measures the extent to which each byte in the data depends on the previous byte. For random data, this should approach zero.

4.2.3 Throughput Summary

To measure the throughput of the GPU TRNG algorithm, every few seconds statistics are written to a file on disk. These statistics include how long the current test has been running for and how much entropy data has been generated.

The first observation of the experiment results is that there is no performance difference between xor, linear transform, and whether or not the exact histogram equalization is

performed. The biggest impact and bottleneck to performance is the number of iterations performed.

Table 4.2 shows the results for each experiment configuration. Two results are shown for each experiment: average duration in seconds and average throughput in bytes per second.

		Iterations				
		256	512	1024	2,048	4,096
Repetitions	4,096	5.21s 12,577.54 B/s	7.02s 9,332.98 B/s	10.67s 6142.44 B/s	17.91s 3,659.74 B/s	32.42s 2,021.64 B/s
	8,192	4.78s 13,709.74 B/s	6.59s 9,949.91 B/s	10.23s 6,406.31 B/s	17.47s 3,752.14 B/s	31.98s 2,049.45 B/s
	16,384	4.56s 14,371.76 B/s	6.37s 10,291.18 B/s	10.01s 6,549.24 B/s	17.25s 3,799.86 B/s	31.76s 2,063.64 B/s
	32,768	4.45s 14,732.22 B/s	6.26s 10,470.39 B/s	9.90s 6,622.74 B/s	17.14s 3,824.25 B/s	31.65s 2,070.98 B/s
	65,536	4.40s 14,902.56 B/s	6.20s 10,565.20 B/s	9.85s 6,654.65 B/s	17.08s 3,836.40 B/s	31.60s 2,074.29 B/s

Table 4.2: Duration and Throughput Results

Figure 4.1, you can see that there is a direct relationship between the number of iterations, and the performance of the algorithm. The unexpected result of this experiment, is that the number of repetitions had a slight impact on performance. However, I believe this can be explained by the fact that there is an increased amount of statistical output for the smaller repetitions to disk. An experiment that has the number of iterations set to 4096 has to write statistics each time it completes an iteration where as if the number of iterations is set to 65536 it only has to write statistics once.

Reviewing the timings in table 4.2 it can be determined that writing statistics takes between 0.05 and 0.06 seconds.

4.2.4 RNGTEST Results Summary

The next step in evaluating Chan's GPU TRNG was to review the results from RNGTEST. The first result based on the tests is that every configuration that had exact histogram

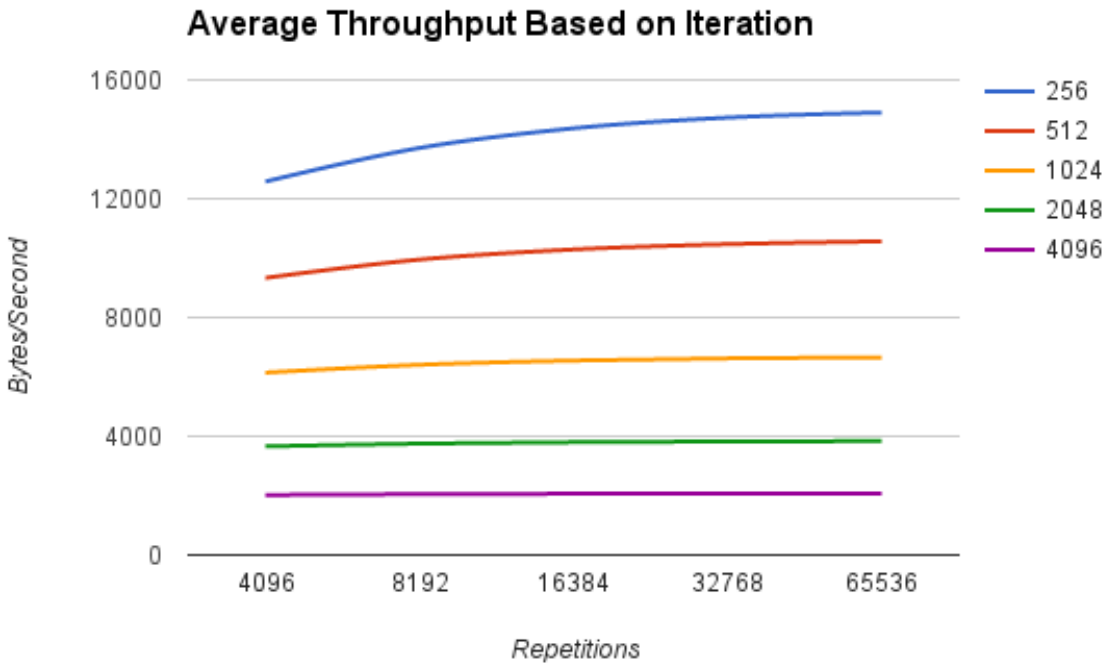

Figure 4.1: Average Throughput Based on Iterations

equalization disabled failed. These tests failed primarily on the poker and runs test. Chan et al. chose exact histogram equalization to improve the entropy of the data, and this determines that they were correct. Without the exact histogram equalization step, the quality of the entropy data is poor.

With exact histogram equalization enabled, the tests that used XOR rather than linear transform were the ones that consistently passed most RNGTEST tests. However, linear transform typically failed on the poker and run tests. The reason for why linear transform failed the poker and run tests more frequently is likely due to the fact that when mapping from a float to a byte, there are only 255 byte values to map to. Any numbers that are close to each other in value will map to the same byte using linear transform. This means that there is an increased potential to have several repetitions of the same byte. While XOR has the same problem in that there is compression, it is not as apparent with RNGTEST because numbers that are close together do not necessarily map to the same byte value.

The number of successful passes for each iteration and repetition combination was evaluated next (Figure 4.2). Based on the results, the number of iterations did not seem to have any impact on the number of RNGTEST successes.

Table 4.3 shows the RNGTEST results for all of the tests that had exact histogram equalization and XOR enabled. Long runs were a rare occurrence in the data generated and most fails were due to the entropy data failing the poker test. Long runs were rare probably due to the fact that twenty six zeros in data that has had it's entropy improved through exact histogram equalization and then processed with XOR would be uncommon. The poker failures are probably due to the loss of entropy from the compression quality of the XOR algorithm.

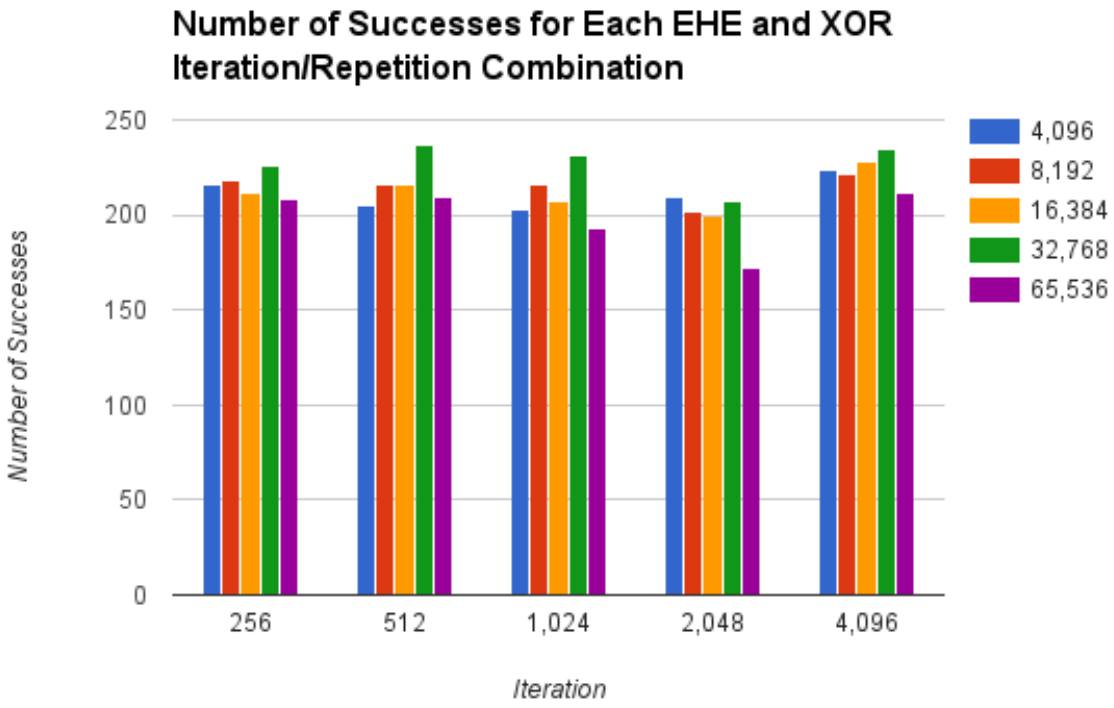

Figure 4.2: Number of Successes for Exact Histogram Equalization and XOR

Iterations	Repetitions	Success	Failure	Monobit	Poker	Runs	Long Run
256	4,096	216	44	11	38	4	1
256	8,192	218	42	9	38	8	0
256	16,384	212	48	11	46	8	0
256	32,768	226	34	6	30	0	0
256	65,536	208	52	8	49	10	0
512	4,096	205	55	10	51	2	1
512	8,192	216	44	4	41	5	0
512	16,384	216	44	7	35	6	0
512	32,768	237	23	3	20	1	0
512	65,536	209	51	9	43	9	0
1,024	4,096	203	57	12	53	4	0
1,024	8,192	216	44	16	35	7	0
1,024	16,384	207	53	12	45	8	1
1,024	32,768	231	29	4	24	2	1
1,024	65,536	193	67	11	64	6	0
2,048	4,096	209	51	11	44	5	0
2,048	8,192	202	58	13	52	10	0
2,048	16,384	200	60	12	47	14	0
2,048	32,768	207	53	12	45	5	1
2,048	65,536	172	88	18	83	8	0
4,096	4,096	224	36	3	35	6	0
4,096	8,192	221	39	6	34	5	0
4,096	16,384	228	32	6	28	10	0
4,096	32,768	235	25	7	17	4	0
4,096	65,536	212	48	12	36	9	1

Table 4.3: Exact Histogram Equalization and XOR RNGTEST Results

4.2.5 ENT Results Summary

The ENT results were similar to the throughput results in that the number of repetitions did not have a significant impact. The deviation between the various repetitions was minimal. The ENT results were also similar to the RNGTEST results in that unless XOR and exact histogram equalization were used, the entropy was low. An average of the various repetitions for each iteration was collected. The results from the XOR and exact histogram equalization tests can be found in Table 4.4.

For each ENT test, the entropy was close to the maximum value of eight. The results for

Iterations	Entropy	Mean	Monte-Carlo-PI	Serial Correlation
256	7.999	127.4	3.14317892	0.06384519
512	7.999	127.4	3.14059327	0.05589011
1,024	7.999	127.4	3.14622231	0.04856509
2,048	7.999	127.4	3.14350853	0.0246228
4,096	7.999	127.4	3.18024541	-0.02285363

Table 4.4: Exact Histogram Equalization and XOR ENT Results

mean, Monte-Carlo-PI and serial-correlation values were also excellent.

4.2.6 Summary and Conclusions

Based on the results from the previous experiments, five combinations of test configurations were chosen for additional testing. While the focus of the thesis is throughput, quality was still a worthwhile secondary evaluation criteria. With that in mind, only XOR and exact histogram equalization enabled experiments qualified to be selected for additional testing since they provided the best quality of entropy data.

The winner for throughput was the test configuration that had the number of iterations set to 256. I decided that rather than look at all of the various 256 iterations again in a second round of testing, I would select an additional four different test configurations.

I chose the top performing configuration to be the one with 256 iterations and 32,768 repetitions. The 256 iteration series of tests had the highest performance, and the 32,768 repetition had the highest number of RNGTEST successes.

The Euclidean Distance was then calculated between the top performing test configuration and all other XOR test configurations. The following criteria, with equal weight, were used in the Euclidean distance calculation:

- Speed to Complete
- Total RNGTEST Successes

- ENT - Entropy
- ENT - Mean
- ENT - Monte-Carlo-Pi
- ENT - Serial-Correlation

Only the first match for each iteration value was chosen to continue for further testing. They were:

- Iteration = 256, Repetition =4096
- Iteration = 512, Repetition = 32,768
- Iteration = 1,024, Repetition = 8,192
- Iteration = 4,096, Repetition = 16,384

The iterations with 2,048 were not included as it scored just below the 4,096 configuration and only five additional configurations were chosen to be tested along with the top configuration.

4.3 Second Phase of GPU TRNG Experiments

A follow-up experiment was run based on the results from the previous tests. The top five variations of Chan's GPU TRNG from the previous testing were selected for additional testing. The additional testing was to confirm the previous experiments results with a larger sample size and to confirm that the top configuration was in fact the correct choice.

In this extended testing, each algorithm was run one hundred times and each time generated one megabyte of entropy data. The results of that testing was parsed, aggregated, and

evaluated using the same three metrics as in the previous experiment. Those three metrics were throughput, RNGTEST, and ENT.

The oddity experienced in the previous tests continued here where the lower number of repetitions caused an increase in overhead due to writing statistics to disk more frequently than configurations with a higher number of repetitions. This writing to disk was performed only for these experiments and was not enabled later when testing Entropy Broker.

The five chosen test configurations are described in Table 4.5.

Iterations	Repetitions
256	4,096
256	32,768
512	32,768
1,024	8,192
4,096	16,384

Table 4.5: Test Configurations

4.3.1 Throughput Summary

Average throughput results for the second round of testing are given in Table 4.6. Based on those results, the configuration with 256 iterations and 32,768 repetitions performed the best. However, as explained in the previous sections, having less repetitions caused some additional overhead when measuring throughput due to the additional statistics that are written for this phase of testing. This caused the 256 iteration configuration with 4,096 repetitions to perform worse than the configuration with 32,768 repetitions.

4.3.2 RNGTEST Results Summary

RNGTEST results for the second round of testing are provided in Table 4.7. The RNGTEST results were similar to the previous experiment results. Overall, there is a higher number of

Iteration	Repetition	Speed (B/s)	Previous Experiment Speed (B/s)	Runtime (s)
256	32,768	17,213.45	14,732.22	60.91
256	4,096	14,368.18	12,577.54	72.97
512	32,768	11,544.28	10,470.39	90.83
1,024	8192	6,849.76	6406.31	153.08
4,096	16,384	2,106.75	2,063.64	497.72

Table 4.6: Second Round of Tests Throughput Results

total failures. Due to the small size of the previous experiments, I had expected to see this result.

Iteration	Repetition	Successes	Failures	Monobit	Poker	Runs	Long Run
256	32,768	35,258	6,642	2,087	5,365	778	21
256	4,096	32,071	9,829	2,382	8,552	1,559	20
512	32,768	37,451	4,449	1,483	3,438	534	24
1,024	8192	31,455	10,445	2,072	9,487	1,389	10
4,096	16,384	33,240	8,660	2,153	7,490	1,394	23

Table 4.7: Second Round of Tests RNGTEST Results

The amount of failures for the various tests is consistent with the previous experiments. Most of the RNGTEST failures are due to the poker and monobit tests. There are very few long run test failures.

Perhaps the most interesting result is that there still seems to be a large variety to the number of successes between the various configurations. I would have assumed that with the larger number of tests, I would have found the values closer together. This is something that should be looked at in future work as it suggests that the number of iterations and the number of repetitions may have an impact on the quality of the entropy data generated after all.

4.3.3 ENT Results Summary

The ENT results are provided in Table 4.8 and provided no surprises. The entropy was the same and the serial-correlation result was similar to the previous experiments.

Iterations	Repetitions	Entropy	Mean	Monte-Carlo-PI	Serial Correlation
256	4,096	7.999	126.147705	3.16302282	0.03370154
256	32,768	7.999	127.6656865	3.11860995	0.07519801
512	32,768	7.999	127.665649	3.11679655	0.06818049
1,024	8,192	7.999	128.162476	3.11217471	0.05329567
4,096	16,384	7.999	127.1698	3.13200837	0.05074492

Table 4.8: Exact Histogram Equalization and XOR ENT Results

The mean and Monte-Carlo-PI results were less than expected. This warranted further investigation. After reviewing the raw results, I determined that in the current batch of tests, the Monte-Carlo-PI and mean were very similar to one another. In the previous experiments, there were less results, and the average was taken across tests with the same iterations. This leads me to conclude that the number of repetitions must have an impact on the quality of the entropy data generated. As before, this warrants examining in more detail in future work.

4.3.4 Summary and Conclusions

In the previous sub-sections, I provided a review of the additional testing that was performed on the five selected configurations for Chan's GPU TRNG. Performance was the primary factor that was being evaluated. The quality of the entropy data generated was also evaluated.

Based on the experimental results, the fastest performing configuration was the same configuration as from the first batch of experiments. The configuration had its values set to 256 iterations and 32,768 repetitions. As noted above, it is difficult to compare the two 256 iteration configurations throughput due to the way that performance statistics are generated. Regardless, it is still fair to select the 32,768 version based on the measured performance. It will at least be no worse than the 4,096 version throughput performance.

Even with the larger sample size, RNGTEST results were similar to the previous experi-

mental results. ENT had similar results for the amount of entropy and the serial-correlation values but the mean and Monte-Carlo-PI results were less than expected. This was due to the way the previous experiment averaged the results.

Based on the throughput and quality analysis, the configuration with 256 iterations and 32,768 repetitions was selected for use.

Chapter 5

Analyzing Entropy Broker

In this chapter, I explain the experiments and evaluation performed with Entropy Broker. These final experiments evaluated a modified Entropy Broker server process that utilized Chan's GPU TRNG to transmit entropy data to several virtualized Linux guests. The top performing algorithm from the previous chapter was chosen to be used in these experiments. That algorithm had the number of iterations set to 256 and the number of repetitions to 32768.

5.1 Architecture

For these experiments, three Linux servers at the University of Manitoba were provisioned. The server names were Parrot, Partridge, and Cuckoo-04. See Tables 5.1, 5.2, and 5.3 for the server configurations.

Parrot was used to host the virtualized Linux environment. VirtualBox 4.3.34 was installed and configured for up to four virtualized guests. Each guest was setup to have 100% of a single CPU, 512MB of RAM, and had Ubuntu 14.04.02 installed with no graphical user interface. Parrot's CPU has two cores, each core is able to run up to two threads.

Server Name:	Parrot
Description:	Server running Oracle VM VirtualBox
Operating System:	Scientific Linux release 6.6
CPU:	Intel 661 @ 3.33 GHz
RAM:	8 GB

Table 5.1: Parrot Configuration

Server Name:	Partridge
Description:	Server running the host Entropy Broker application
Operating System:	CentOS release 5.11
CPU:	Intel Q9550 @ 2.83 GHz
RAM:	4 GB

Table 5.2: Partridge Configuration

Server Name:	Cuckoo-04
Description:	Server used to run modified Chan's GPU TRNG to generate entropy data for Entropy Broker
Operating System:	CentOS release 5.11
CPU:	Intel Q9550 @ 2.83 GHz
RAM:	4 GB

Table 5.3: Cuckoo Configuration

partridge.cs.umanitoba.ca	*entropy_broker -n*
cuckoo-04.cs.umanitoba.ca	*./server_gpu_trng -X password.txt -I partridge.cs.umanitoba.ca -n*
guest	*sudo ./eb_client_linux_kernel -n -X password.txt -I partridge.cs.umanitoba.ca timeout duration rngtest < /dev/random*

Table 5.4: Server Commands

The configurations for each virtualized Linux guest were identical and included OpenSSH, Entropy Broker, and the required libraries for Entropy Broker to function. VirtualBox was configured to allow SSH from the local host.

Partridge was used to host the core functionality of Entropy Broker. A decision was made early in my experiments to separate the core Entropy Broker application from the application that generated entropy data. This was done to avoid any bottlenecks that the CPU might introduce.

Cuckoo-04 was used to generate entropy data. An nVidia GeForce GTX 260 graphics card was used to generate entropy data using Chan's GPU TRNG and the Entropy Broker APIs.

The Entropy Broker application was started first on Partridge, and then the Entropy Broker server process was launched on Cuckoo-04. Before consuming any of the entropy data in Entropy Broker, the pools that store the entropy data on the Entropy Broker server were filled. This ensured a consistently available amount of entropy data for the start of every experiment. Figure 5.1 displays the architecture used.

Table 5.4 shows the commands used on each server to start the Entropy Broker experiments.

Once the entropy data pools in Entropy Broker were full, the client processes on the virtualized Linux guests were connected to the main server. Each client process requests entropy data whenever its internal entropy data pool (stored at /dev/random) hit a certain threshold. The client process would then begin requesting entropy data. Entropy data was

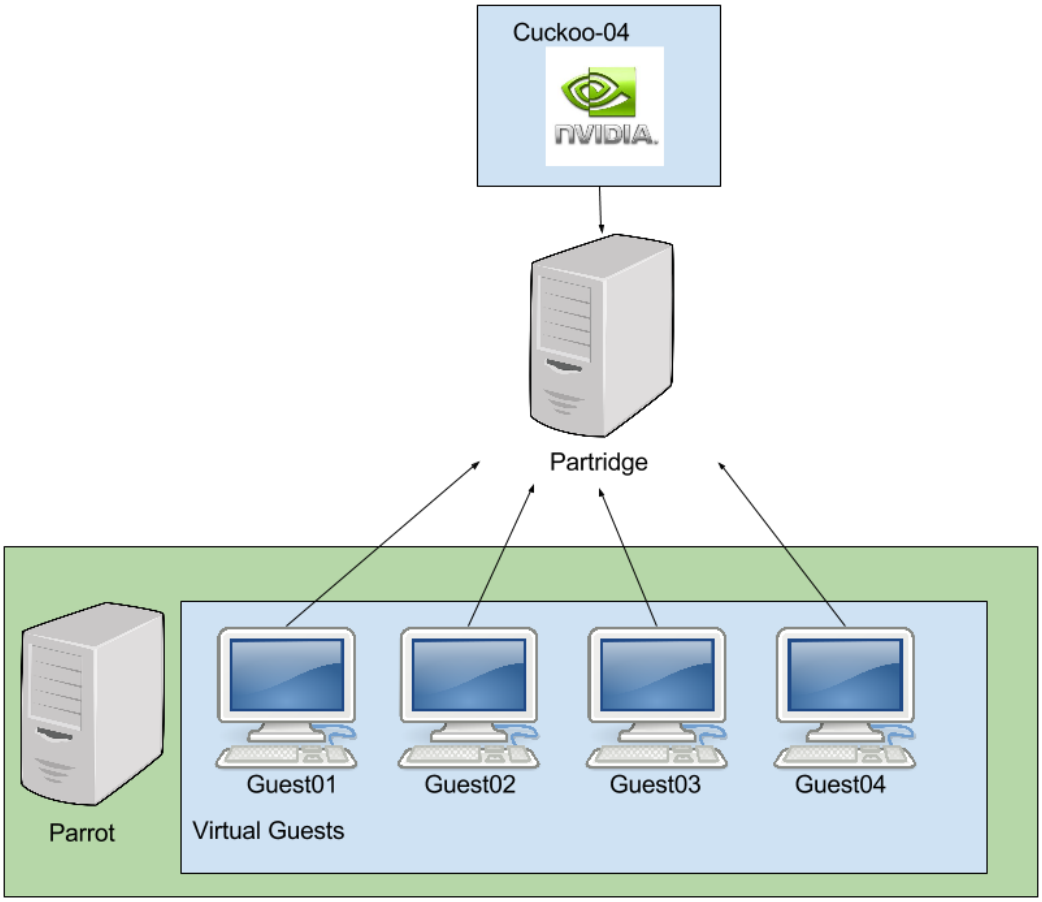

Figure 5.1: Entropy Broker Architecture

requested by using the built-in timeout functionality in Linux, a timeout duration, and the RNGTEST application. The guest running the client process is forced to drain its entropy data pool using RNGTEST, the client process then re-requests entropy data from Entropy Broker.

To test Entropy Broker a variety of configurations were developed to test as many use cases as possible. The various configurations are detailed below.

Each use case was run with the following four configurations:

One Virtual Guest

A single virtual guest has full access to all of the available resources from the host. There is no competing for CPU cycles as only one core will be utilized.

Two Virtual Guests

Two virtual guests should still have full access to all of the available resources from the host. There should be little competing for CPU cycles as both cores will be utilized but there are still available threads.

Three Virtual Guests

Three virtual guests will begin to stress the host. All cores will be utilized and one core will run two threads. There will start to be some competition for resources.

Four Virtual Guests

Four virtual guests will stress the host server. All cores and threads will be utilized. Each guest will compete for resources.

The following sections detail the various experiments.

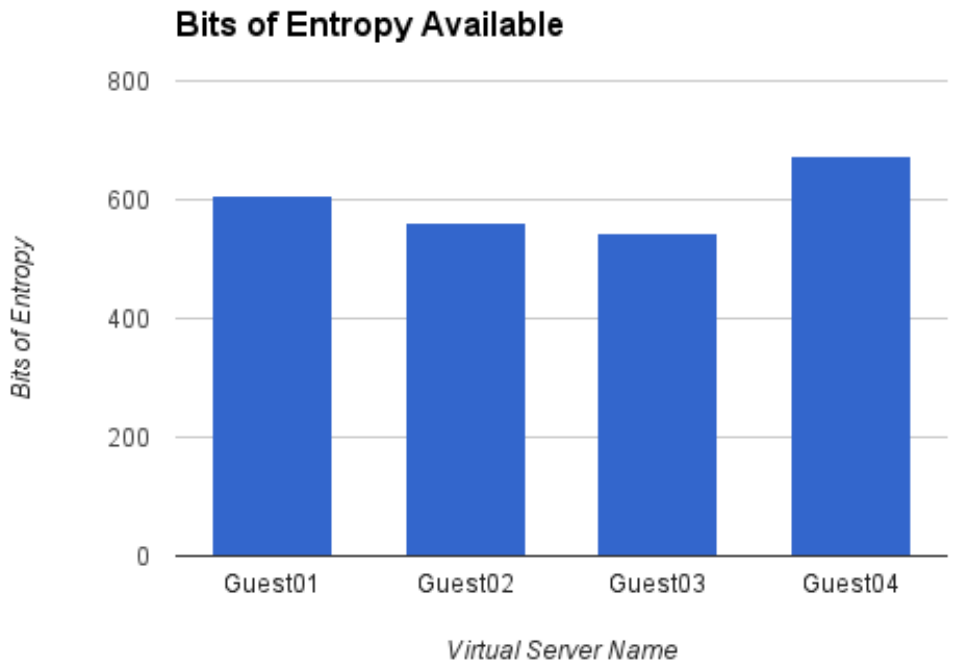

Figure 5.2: Bits of Entropy Data Available Over Thirty Minutes

5.2 Linux Virtual Environment Experiments

5.2.1 No Entropy Improvement

The first experiment that was performed had no entropy data improvement and was used to establish a baseline of available entropy data in the virtual environment. Each virtual machine configuration was run through this experiment. Each virtual guest was tested to determine how much entropy data was available over thirty minutes of RNGTEST.

Based on the results of this first experiment, it was determined that there was no difference between having one, two, three, or four virtual machines running. The amount of entropy data available stayed the same. Figures 5.2 and 5.3 show the amount of entropy data available in two different ways.

Over a period of thirty minutes the average amount of entropy data available was 596

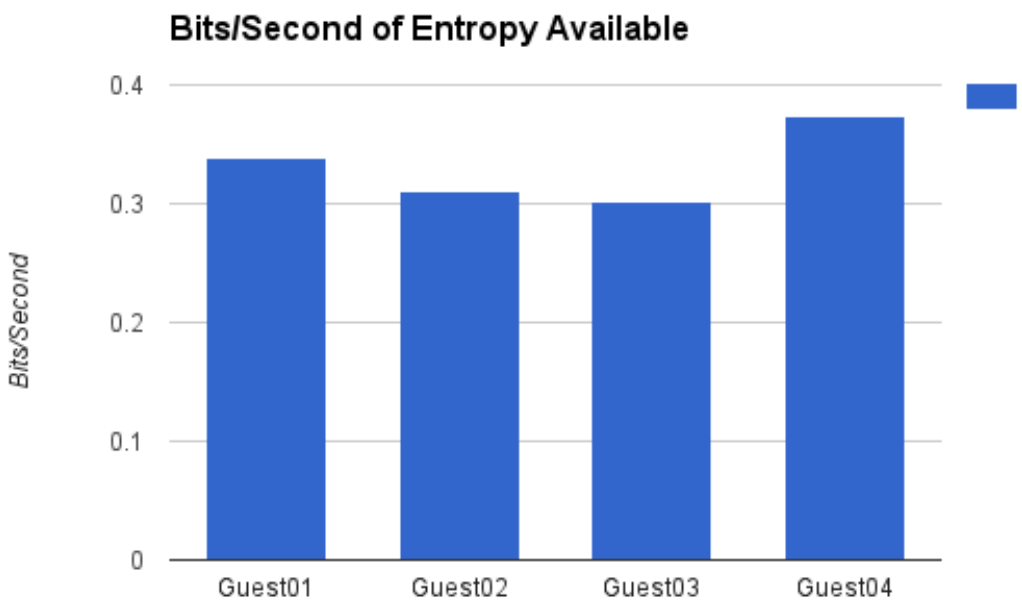

Figure 5.3: Entropy Data Available in Bits per Second

bits, the largest amount of entropy data available was 672 bits on Guest04. The minimum amount of entropy data available was 544 bits on Guest03. The average available bits per second across all virtual machines was 0.33 bps.

5.2.2 GPU TRNG and Entropy Broker

The second experiment contained four Entropy Broker scenarios to get a sense for what Entropy Broker is capable of as far as improving entropy data generation in a virtualized Linux environment.

5.2.2.1 Experimental Scenario One

The first experiment scenario entailed configuring Entropy Broker, the client processes, and the server processes properly in the environment and measuring the performance. Each virtual machine configuration was run through this experimental scenario.

Based on the results, the very first thing identified was that the amount of entropy data the virtual machines consumed was not limited by the generation speed of Chan's GPU TRNG. In fact, it seems that the way that Entropy Broker is designed and configured caused the biggest bottleneck. An e-mail was sent to the owner and developer of the tool requesting some additional information about this but at the time of writing my thesis I had not yet heard a reply. The bottleneck aside, Entropy Broker still increased the available entropy data in the virtual Linux guests.

Figure 5.4 shows the total amount of entropy data generated in bits per second for each test case. The following equation was used to calculate total consumed entropy data:

$$TotalAvailableEntropyData = TotalAvailableEntropyData/TotalDurationTests$$

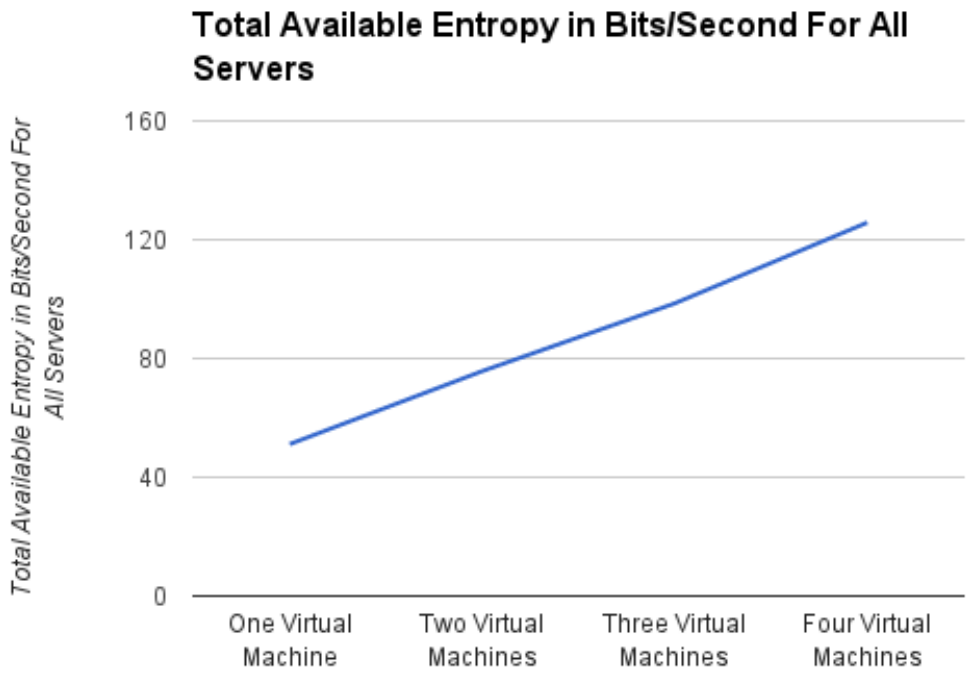

Figure 5.4: Total Available Entropy Data in Bits per Second for all Servers

The amount of entropy data consumed increased linearly based on the number of virtual machines and fits the line of best fit of $y = 24.67x + 26.11$.

Figure 5.5 displays the average available entropy data in bits per second for each server based on the number of virtual machines in the test case. The average entropy data in bits/second decreases significantly at first but levels off once it hits three and four virtual machines.

Reviewing the raw results, an interesting effect was noticed. For two virtual machines, the amount of entropy data generated for each server was generally equal. However, for three or four servers, the amount of entropy data generated for each server seemed to favor one or two virtual guests with no decrease of overall entropy data for the other servers entropy data generation. Based on my investigation, there does not seem to be a reason for this favortism. As before, an e-mail was sent to the owner and developer of the tool requesting

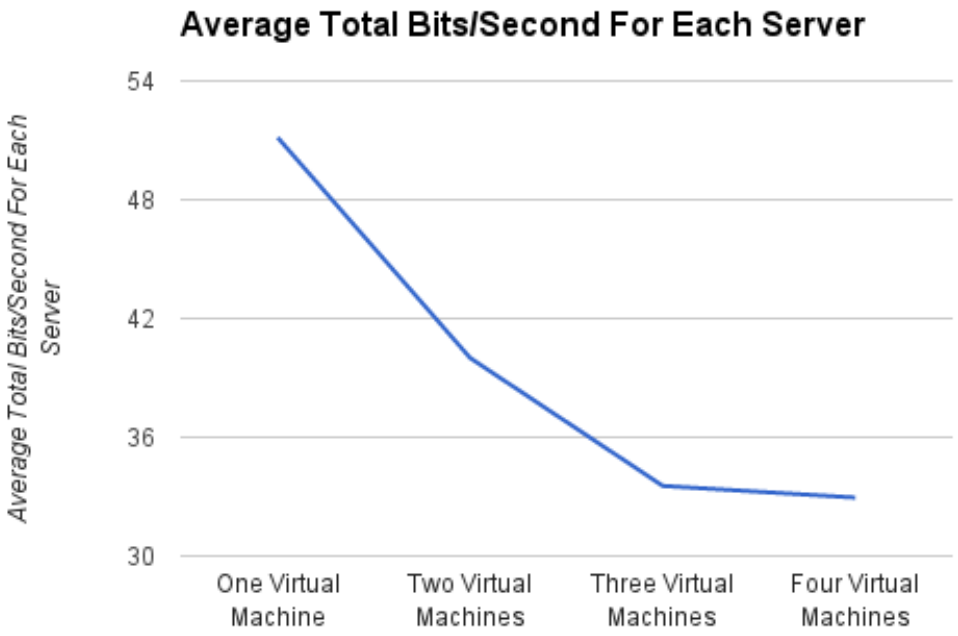

Figure 5.5: Average Total Bits per Second For Each Server

some additional information about this but I have not yet heard back.

The maximum available entropy data in bits per second was consistent across all test cases. Generally at least one server was able to consume at least 45 bits per second up to a maximum of 51 bits per second. The minimum amount of entropy data consumed was 14 bits per second but usually averaged 18 bits per second.

Figure 5.6 shows that the average throughput increases almost linearly as well. Average throughput was calculated by computing the total entropy data for each test scenario and averaging the total entropy data across the other tests within that scenario.

The average total throughput for all servers also increases linearly based on the number of virtual machines and fits the line of best fit of $y = 25.05x + 25.59$.

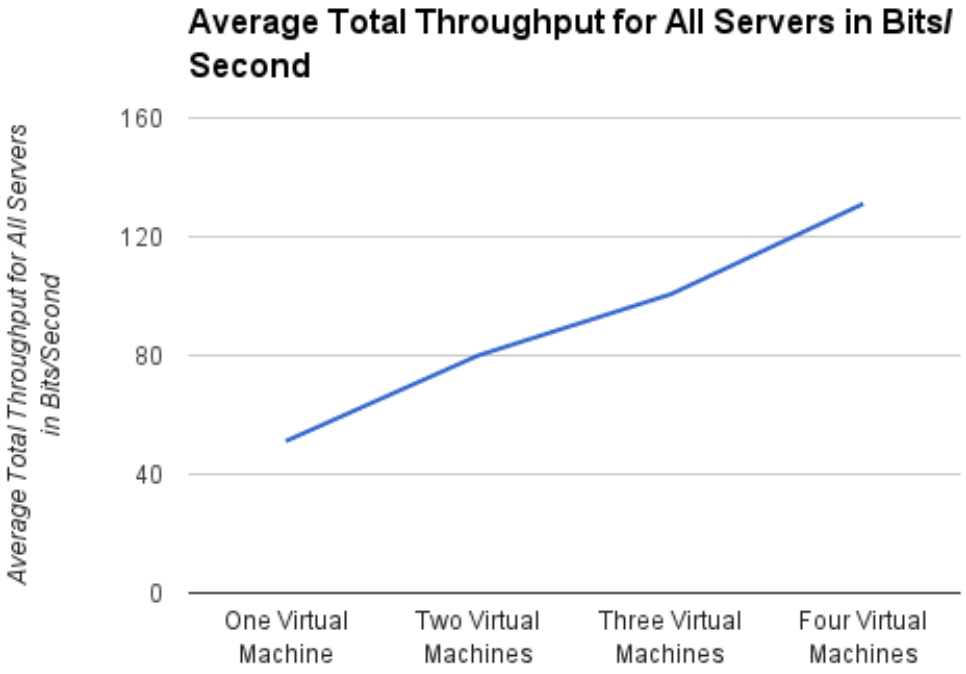

Figure 5.6: Average Total Throughput for all Servers in Bits per Second

Average Entropy Data for All Servers	161.13 bps
Average Entropy Data Consumed per Server	40.28 bps

Table 5.5: Four Virtual Guests and Full Pools of Entropy Data - Entropy Broker Enabled

5.2.2.2 Experimental Scenario Two

The second experimental scenario filled up the pools of entropy data in Entropy Broker and disabled the GPU TRNG server process. This was done to determine how long Entropy Broker was able to function before requiring entropy data input from the server process. Only the four virtual guest configuration was used. This was done to attempt to drain the pools as quickly as possible within my experimental limits. A timeout of six hours was used.

Table 5.5 show the performance with four servers enabled. After the six hours, there was still entropy data available in Entropy Broker. No further testing was conducted as this confirms that the way Entropy Broker distributes entropy data is the bottleneck for entropy data transfer.

5.2.2.3 Experimental Scenario Three

The third scenario disabled Entropy Broker once the virtual guests stopped requesting entropy data. This was done to determine how much of an initial impact Entropy Broker had on the virtual machine's entropy data pool. Only the four virtual guest configuration was used in this test. The timeout was fifteen minutes. As the goal of this test was only to evaluate the initial impact of Entropy Broker, a longer test was not required.

Figures 5.7 and 5.8 shows the increase in initial performance. This performance was compared to what was shown in figures 5.2 and 5.3. An average increase of 878 bits of available entropy data was seen along with an average increase of 1.307 bps of entropy data consumption. This performance increase would not occur over the long term as there would be no additional entropy data provided by Entropy Broker once the initial Linux entropy data pool was depleted.

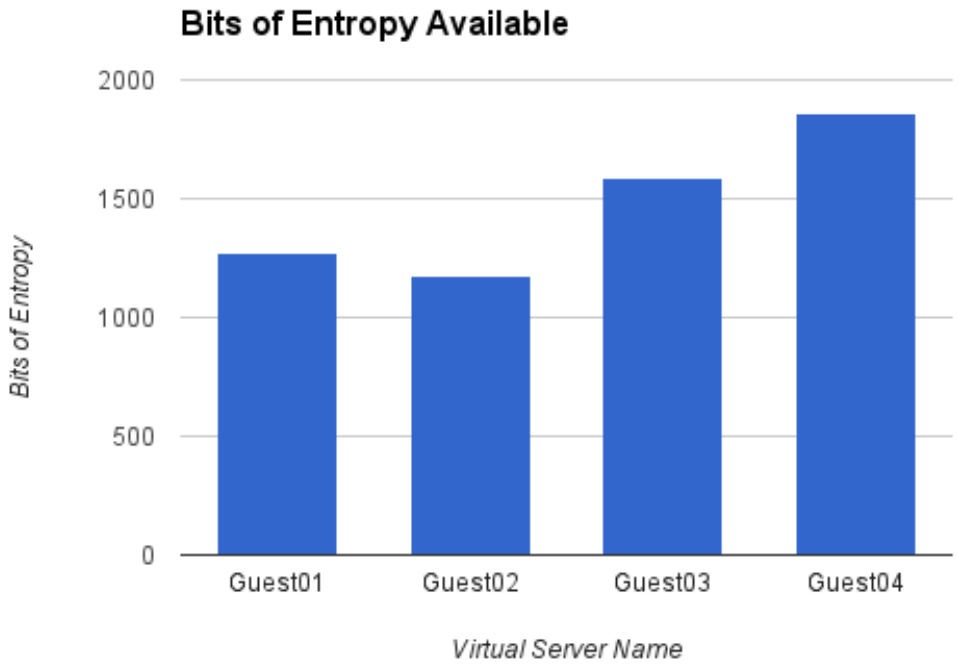

Figure 5.7: Disabled Entropy Broker Total Bits Available per Server

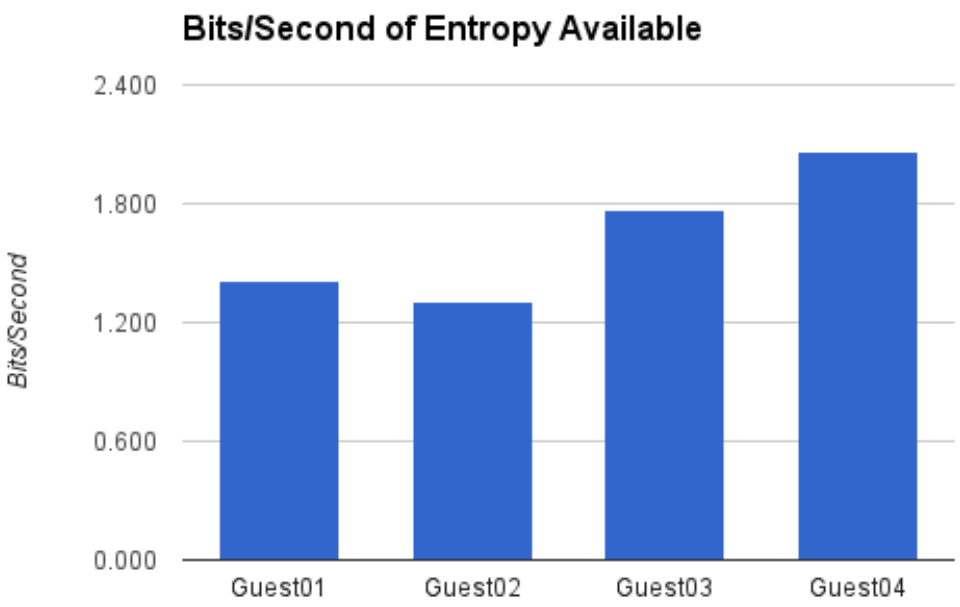

Figure 5.8: Disabled Entropy Broker Bits per Second per Server

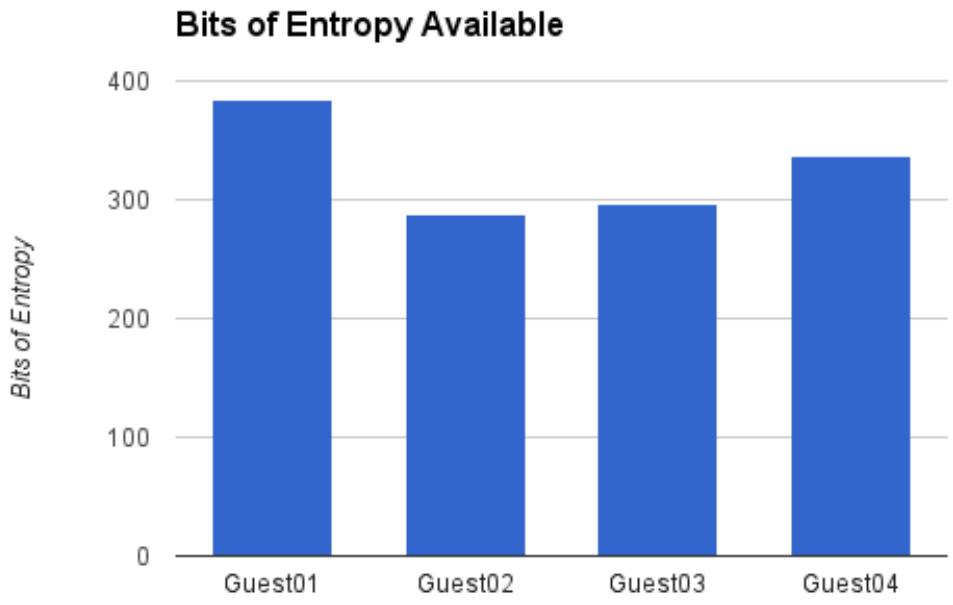

Figure 5.9: Entropy Broker and No Pooled Entropy Data - Total Bits Available per Server

5.2.2.4 Experimental Scenario Four

The fourth scenario had Entropy Broker enabled, but no pooled entropy data and no server process to provide entropy data to Entropy Broker. The purpose of this experiment was to determine the impact, if any, Entropy Broker had on the virtual machines without it's server processes or pooled entropy data. Only the four virtual guest configuration was used in this test. The timeout was fifteen minutes and the impact of the experiment was readily apparent within a short period of time.

The results can be seen in Figures 5.9 and 5.10 and confirmed that Entropy Broker requires pooled entropy data to function and has no impact on the virtual guests entropy data pools.

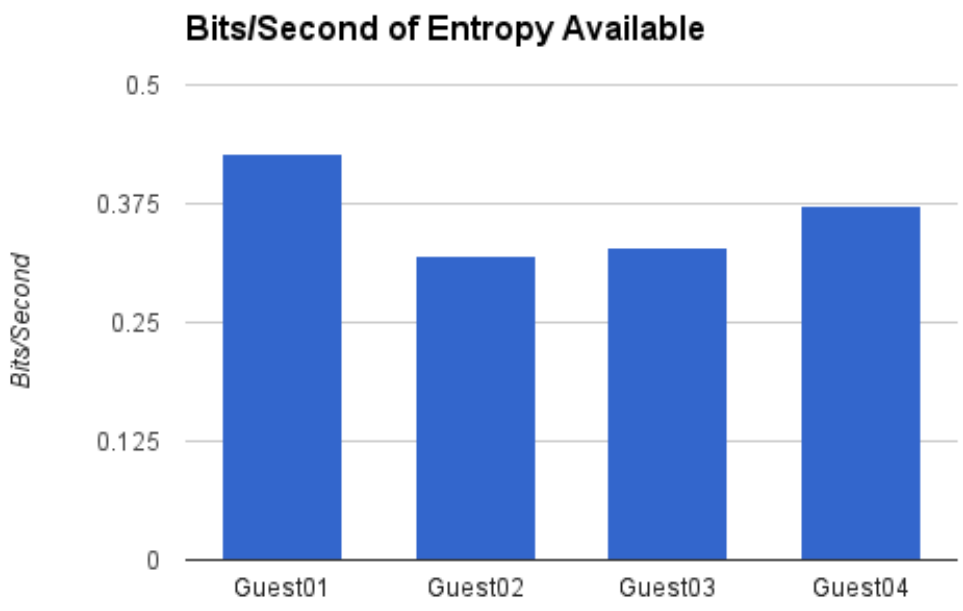

Figure 5.10: Entropy Broker and No Pooled Entropy Data - Total Bits per Second per Server

Setting	Old	New
max_number_of_mem_pools	512	1,024
min_store_on_disk_n	5	10
default_max_get_bps	4,096	8,192
max_get_put_size	32,768	65,536
kernelpool_filled_interval	3,600	5

Table 5.6: Entropy Broker Configuration Differences

	New	Old
Average Entropy Data Available for All Servers	944.81 bps	125.79 bps
Average EntropyData Available per Server	248.70bps	32.95 bps

Table 5.7: Entropy Broker Original vs. Modified Configuration for Four Virtual Guests'

5.2.3 Entropy Broker Configuration Modification

The final step in evaluating Entropy Broker was to see the impact that minor modifications to configuration file would have on entropy data generation and the virtual Linux guests' performance. The differences between the original and modified configurations are found in Table 5.6.

Based on Table 5.7 you can see that with the modification in Table 5.6 performance was increased by approximately 7.5x. Additional modifications should be possible to further increase the performance of Entropy Broker.

5.2.4 Other Entropy Broker Sources

The developer of Entropy Broker, Folkert van Heusden, has published several statistics for other sources of entropy data generation for Entropy Broker. These statistics can be found in Table 5.8. These statistics are only from using a CPU to generate entropy data and not how quickly Entropy Broker is able to distribute the entropy data.

As you can see from the Table 5.8, the highest performing system is the one labeled as having it's motherboard as Pheonix and it's processor as AMD Athlon XP. It is able to generate 43,024 bps of entropy data. Chan's GPU TRNG significantly outperforms this by

Name	CPU	Clock	bps
GPU TRNG			137,707.6
Intel DG965WH	Intel Q6600	2.4GHz	2,268
Asus P5K WS	Intel E6600	2.4GHz	358
Gigabyte 965G-DS3	Intel Q9450 Via Nehemiah C3 Via C7	2.66GHz 530MHz 1.8GHz	1,751 2,442 1,491
Phoenix	AMD Athlon XP	1.6GHz	43,024
ASUS A8N-E	AMD Athlon 64 X2 4200+	2.3GHz	117
ASUS M3A	AMD Athlon 64 X2 4800+	2.6GHz	112
Via TUV4X	Tualatin Celeron III	800 Mhz	2,447
IBM xSeries 330 8654-52G	P3 Coppermine	1GHz	4,440
IBM x3800 8865-2RG	Intel Xeon	3.6GHz	3,238
IBM 9113-550	Power5 (in an LPAR)	1.5GHz	23
SGI O200 (running IRIX)	MIPS R12000	270MHz	191
PowerEdge R200 (running FreeBSD)	Intel Xeon CPU 3065	2.3GHz	189
VMware Workstation on Lenovo T61	i386 Emulation	2.4GHz	245

Table 5.8: Entropy Broker CPU Entropy Data Generation Throughput

being able to generate 17,213.45 B/s or 137,707.6 bps of entropy data.

5.3 Haveged

Haveged was tested within the virtual Linux environment to show the affect a pseudo random number has on entropy data generation. All four virtual guest configurations were run. Haveged was tested briefly to determine the impact it had on available entropy data and the virtual guests. The timeout was fifteen minutes for each experiment as the impact of Haveged is quickly noticed. The experiments were setup the same way as for Entropy Broker where entropy data was requested by using the built-in timeout functionality in Linux, a timeout duration, and the RNGTEST application.

Haveged relies on the virtual guests' CPU to generate entropy data. In this virtual Linux environment, 33% of the CPU was required for Haveged to run for each virtual guest. For the configuration with four virtual machines, one CPU was completely dedicated to entropy

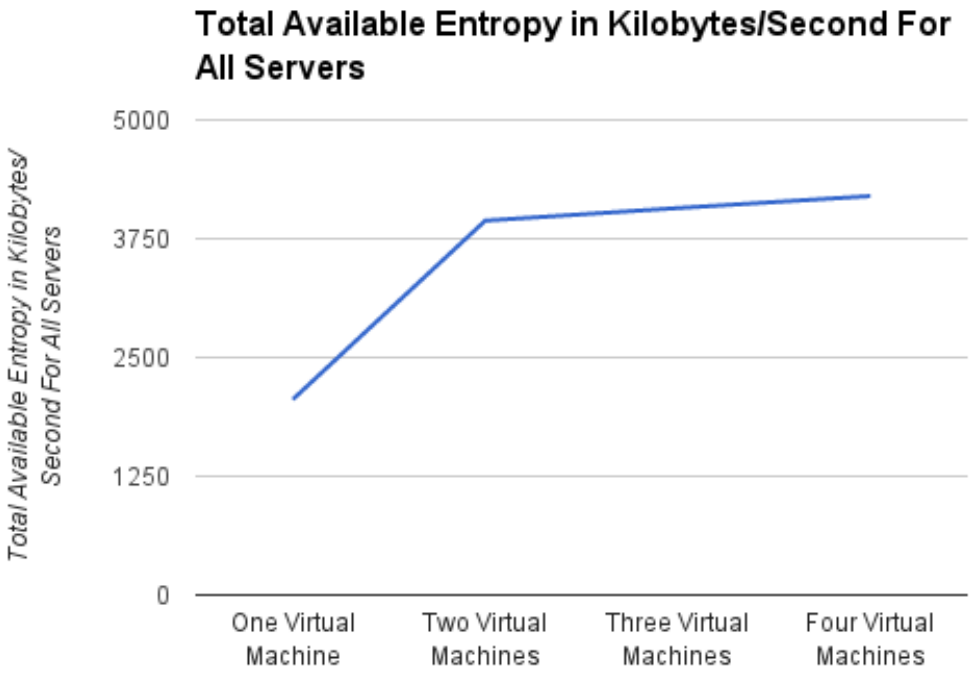

Figure 5.11: Haveged - Total Available Entropy Data per Configuration

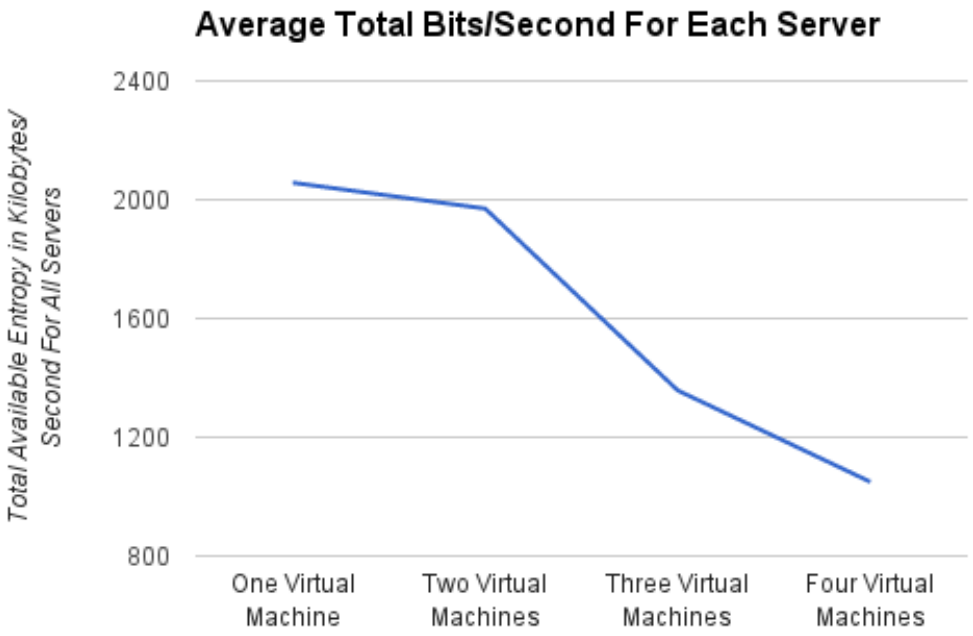

Figure 5.12: Haveged - Average Throughput per Configuration

data generation and 33% of a second CPU was also required. This is reflected in Figures 5.11 and 5.12 in that the total amount of entropy data consumed plateaus after more than two virtual machines are running, and the total average entropy data consumption drops significantly.

Haveged generates significantly more entropy data than Entropy Broker is able to share and significantly more entropy data than Chan's GPU TRNG. This isn't really a fair test as Haveged is a pseudo random number generator where Chan's GPU TRNG is not. In addition, Entropy Broker and Chan's GPU TRNG was setup in a client-server infrastructure where as Haveged was installed on the virtual machine where you require additional entropy data. However, it does accurately show the advantage in throughput that pseudo random number generators have over the GPU based TRNG.

1-10 Virtual Machines Average Entropy Data Available per Server	146 Kb/sec
11-60 Virtual Machines Average Entropy Data Available per Server	$1,520/n$ Kb/sec

Table 5.9: Intel's Digital Random Number Generator Throughput

5.4 Intel's Digital Random Number Generator

The Intel digital random number generator was not experimented with as there was no hardware available, but the statistics that Intel has published have been included in Table 5.9. For Intel's digital random number generator to function, it requires a CPU equipped with the digital random number generator, a virtual environment that is able to expose the entropy data to guests, and a Linux operating system that is capable of accepting the entropy data from the digital random number generator.

Entropy Broker in it's current configuration functions slower than Intel's generator, though Intel's generator requires a specific CPU to function. Intel's generator also can not compete with Haveged's performance but provides a better baseline to compare Chan's GPU TRNG against.

5.5 Summary

In analyzing Entropy Broker with Chan's GPU TRNG, two conclusions were reached. The first is that there is a bottle neck in the way Entropy Broker shares entropy data with its' client processes that prevented Chan's GPU TRNG from reaching its' maximum throughput. Even with that bottle neck, the amount of entropy data available for virtual machines greatly increases with Entropy Broker.

The second, is that while both Haveged and Intel's digital random number generator outperformed Entropy Broker, they did so by either requiring CPU cycles from the virtual

guest, or by requiring a unique CPU, the correct virtual environment, and a specialized Linux operating system.

Though outperformed by Haveged and Intel's digital random number generator, Entropy Broker and Chan's GPU TRNG provided a unique and useful method of sharing entropy data across systems using a TRNG and greatly increased the amount of entropy data available in the virtual Linux environment.

Chapter 6

Conclusion

For secure computing, good cryptography is required. For good cryptography, a system must have good entropy data. In this thesis, I worked with Chan's GPU TRNG and Entropy Broker to provide a solution to address the problem of limited entropy data within a virtual Linux environment. Chan's GPU TRNG was modified to test various effects of configuration changes. Chan's GPU TRNG was then wrapped in a custom Entropy Broker interface that I wrote to allow it to function with the client-server infrastructure provided by Entropy Broker.

Various configurations of settings were used to test Chan's GPU TRNG. Through evaluation of the results the setting values that provided the best throughput were selected. The selected values were then used in various virtual guest configurations to test Entropy Broker using Chan's GPU TRNG to generate entropy data. Four virtual guest configurations were tested. These results were compared against a pseudo random number entropy data generator called Haveged, Intel's digital random number generator test results, and published Entropy Broker test results.

While Chan's GPU TRNG can not currently beat the entropy data generation peformance of Haveged or Intel's digital random number generator, it is significantly faster than

the entropy data generation of a typical virtual Linux guest and the other published Entropy Broker performance results. The client-server infrastructure has the advantage that it offloads the generation of entropy data from the system that needs it removing the need for additional processing power on the virtual guest. It also does not require a special chipset, as any nVidia graphics card that is CUDA enabled is able to be used to generate entropy data. Entropy Broker also does not require a special CPU, virtual environment, or Linux operating system to function. Entropy Broker can also be distributed across multiple systems.

Chapter 7

Future Work

There are several areas of future work that I would like to pursue. The first area involves further experimentation with Chan's GPU TRNG. This includes:

- Determining the theoretical maximum performance that could be reached using Chan's GPU TRNG to generate entropy data.
- Investigating the effect that different GPU's have on Chan's GPU TRNG as nVidia constantly improves the architecture and performance of their GPU's.
- Further experimentation with the variables in Chan's GPU TRNG to determine their effects.
- Investigate and experiment with different ways of converting or compressing the floats that the GPU TRNG generates with the goal of improving the quality of entropy data generated.
- Determine if there is a relationship between iterations and repetitions.

The second area of future work revolves around combining Entropy Broker and Chan's GPU TRNG. Additional modifications and study of the Entropy Broker configuration should

provide further performance gains. Another area of research would be to investigate whether adding more virtual machines would still allow the performance to continue to scale linearly, and to determine the impact adding more GPU's would have.

The final area of future work would be to investigate other methods of transferring entropy data and not to rely on Entropy Broker. Entropy Broker was designed to work with several different types of entropy data generation methods but creating a solution that was customized uniquely to Chan's GPU TRNG could, in theory, provide better performance.

Bibliography

[1] Vittorio Bagini and Marco Bucci. "A design of reliable true random number generator for cryptographic applications". In: *Cryptographic Hardware and Embedded Systems*. Springer. 1999, pp. 204–218.

[2] Tom Caddy. "FIPS 140-2". In: *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 468–471.

[3] Jose Juan Mijares Chan et al. "True random number generator using GPUs and histogram equalization techniques". In: *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*. IEEE. 2011, pp. 161–170.

[4] S Crocker and J Schiller. *RFC 4086-randomness requirements for security*. 2011.

[5] Theo De Raadt et al. "Cryptography in OpenBSD: An Overview." In: *USENIX Annual Technical Conference, FREENIX Track*. 1999, pp. 93–101.

[6] */dev/random — Wikipedia, The Free Encyclopedia*. 2015. URL: `http://en.wikipedia.org/?title=/dev/random`.

[7] Adam Everspaugh et al. "Not-So-Random Numbers in Virtualized Linux and the Whirlwind RNG". In: ().

[8] Bernd Grobauer, Tobias Walloschek, and Elmar Stocker. "Understanding cloud computing vulnerabilities". In: *Security & privacy, IEEE* 9.2 (2011), pp. 50–57.

[9] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. "Analysis of the linux random number generator". In: *Security and Privacy, 2006 IEEE Symposium on*. IEEE. 2006, 15–pp.

[10] Folkert van Heusden. *Entropy Broker*. 2014. URL: http://www.vanheusden.com/entropybroker/.

[11] *How to Setup Additional Entropy for Cloud Servers Using Haveged*. 2013. URL: https://www.digitalocean.com/community/tutorials/how-to-setup-additional-entropy-for-cloud-servers-using-haveged.

[12] Benjamin Jun and Paul Kocher. "The Intel random number generator". In: *Cryptography Research Inc. white paper* (1999).

[13] Brendan Kerrigan and Yu Chen. *A study of entropy sources in cloud computers: random number generation on cloud hosts*. Springer, 2012.

[14] David P Maher and Robert J Rance. "Random number generators founded on signal and information theory". In: *Cryptographic Hardware and Embedded Systems*. Springer. 1999, pp. 219–230.

[15] *Manual Page for random*. 2015. URL: http://man7.org/linux/man-pages/man4/random.4.html.

[16] John Mechalas. *Intel Data Protection Technology with Secure Key in the Virtualized Environment*. 2013. URL: https://software.intel.com/en-us/articles/intel-data-protection-technology-with-secure-key-in-the-virtualized-environment.

[17] Henrique de Moraes Holschuh. *Manual Page for rngtest*. 2015. URL: http://www.linuxcommand.org/man_pages/rngtest1.html.

[18] Calomel Org. *Entropy and Random Number Generators*. 2014. URL: https://calomel.org/entropy_random_number_generators.html.

[19] *RC4 — Wikipedia, The Free Encyclopedia*. 2015. URL: http://en.wikipedia.org/wiki/RC4.

[20] Tzachy Reinman. "On linux random number generator". PhD thesis. The Hebrew University, 2005.

[21] Werner Schindler and Wolfgang Killmann. "Evaluation criteria for true (physical) random number generators used in cryptographic applications". In: *Cryptographic Hardware and Embedded Systems-CHES 2002*. Springer, 2003, pp. 431–449.

[22] Andy Smith. *Adventures in entropy part 1*. 2010. URL: http://strugglers.net/~andy/blog/2010/06/06/adventures-in-entropy-part-1/.

[23] Vikram B Suresh and Wayne P Burleson. "Entropy extraction in metastability-based TRNG". In: *Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 135–140.

[24] John Walker. *ent - A Pseudorandom Number Sequence Test Program*. 2015. URL: http://www.fourmilab.ch/random/.

[25] Brian Warner. *Remote Entropy*. 2014. URL: https://blog.mozilla.org/warner/2014/03/04/remote-entropy/.

Appendices

Appendix A

Custom Entropy Broker Module Source Code

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/device_ptr.h>
#include <thrust/sequence.h>
#include <thrust/copy.h>
#include <thrust/host_vector.h>
#include <thrust/sort.h>
#include <thrust/functional.h>
#include <thrust/transform.h>

//Basic Libraries//
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <signal.h>
#include <arpa/inet.h>
#include <string>
#include <map>
#include <sys/time.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>

//Entropy Broker//
#include "defines.h"
#include "error.h"
#include "random_source.h"
#include "utils.h"
#include "log.h"
#include "encrypt_stream.h"
#include "hasher.h"
#include "protocol.h"
#include "server_utils.h"
#include "statistics.h"
```

```
#include "statistics_global.h"
#include "statistics_user.h"
#include "users.h"
#include "auth.h"

//CUDA//
#include <cuda.h>

#define SIZE 16
#define MAX_ITERATIONS 256
#define MAX_REPETITIONS 32768
#define LOOP 32

const char *server_type = "server_gpu_trng_v";
const char *pid_file = "/home/cs/grad/umplesiu/server_gpu_trng.pid
    ";

bool do_exit = false;

using namespace std;

/**
 * CUDA kernel function that reverses the order of bits in each
     element of
the array.
 */
```

```
__global__ void TRNGkernel(int M, int S) {
        extern __shared__ int shared[];
        int *s = &shared[0];
        int *out = &shared[S+1];
        int k=0;
        int id =threadIdx.x;
        do{
                s[id] = id;
                out[id]=s[id+1];
                k++;
        } while(k<M);
}

//Signal Handler//
void sig_handler(int sig)
{
        fprintf(stderr, "Exit due to signal %d\n", sig);
        do_exit = true;
}

//Help file//
void help(void)
{
        printf("-I host    entropy_broker host to connect to\n");
        printf("           e.g. host\n");
        printf("                host:port\n");
```

```
        printf("                [ipv6 literal]:port\n");
        printf("           you can have multiple entries of this\n"
            );
        printf("-o file    file to write entropy data to\n");
        printf("-l file    log to file 'file'\n");
        printf("-L x       log level, 0=nothing, 255=all\n");
        printf("-n         do not fork\n");
        printf("-s         log to syslog\n");
        printf("-S         show bps (mutual exclusive with -n)\n");
        printf("-P file    write pid to file\n");
        printf("-X file    read username+password from file\n");
}

void generate_raw_numbers(thrust::host_vector<float> *RN , int S,
    int M, int N, float CL)
{
        int i;
        cudaEvent_t start, stop;
        float kernelTime;

        for(i=0;i<N;i++)
        {
                // activate the timer in the graphic card
                cudaEventCreate(&start);
                cudaEventCreate(&stop);
                cudaEventRecord( start, 0 );
```

```
            // kernel with the race conditions
            TRNGkernel<<<1,S,(2*S+1)*sizeof(int)>>>(M, S);

            // Wait for the GPU launched work to complete
            cudaThreadSynchronize();

            cudaEventRecord( stop, 0 );
            cudaEventSynchronize( stop );
            cudaEventElapsedTime( &kernelTime, start, stop );
            cudaEventDestroy(start);
            cudaEventDestroy(stop);

            // stores the raw numbers
            (*RN)[i] = fmod(kernelTime*CL,float(1.0));
        };


        cudaDeviceReset();
}

struct saxpy_functor
{
    const float a;

    saxpy_functor(float _a) : a(_a) {}
```

```
    __host__ __device__
        float operator()(const float& x) const {
            return x/a;
        }
};

//Exact Histogram Equalization//
thrust::host_vector<float> EHE (thrust::host_vector<float>
   RawNumbers, int N_REPETITIONS)
{
        int N = N_REPETITIONS;
        thrust::device_vector<float> D = RawNumbers;
        thrust::device_vector<float> I(N);
        thrust::device_vector<float> I2(N);
        thrust::device_vector<float> Itop(N);
        thrust::device_vector<float> Ibottom(N);
        thrust::sequence(I.begin(), I.end(), 0.0f, 1.0f);
        thrust::sequence(I2.begin(),I2.end(),0.0f,1.0f);
        thrust::transform(I.begin(), I.end(), I.begin(),
           saxpy_functor(float(N)));
        Ibottom = I;
        thrust::copy(Ibottom.begin()+1,Ibottom.end(),Itop.begin())
           ;
        Itop[N-1]=1.0f;
        thrust::sort_by_key(D.begin(), D.end(), I2.begin());
```

```
        thrust::copy(I.begin(),I.end(),D.begin());
        thrust::sort_by_key(I2.begin(), I2.end(), D.begin());
        thrust::copy(D.begin(), D.end(), RawNumbers.begin());

        return RawNumbers;
}



void printData(unsigned char byte, float RawNumber)
{
        int number;
        number = byte;
        cout << "Raw: " << RawNumber << "Int Value:" << number <<"
            Value: " << byte << endl;
}

//Actual Working Function//
void main_loop(std::vector<std::string> * hosts, char *bytes_file,
    char show_bps, std::string username, std::string password,
   char print_details, char xor_transform, char histogram)
{
        //Configure Variables///
        int N_REPETITIONS = MAX_REPETITIONS;
                                        //4 bytes per float//
        unsigned char bytes[N_REPETITIONS];
        thrust::host_vector<float> RawNumbers(N_REPETITIONS);
```

```
            //Initialize raw numbers variable.
protocol *p = NULL;
                                                        //
   Required for Entropy broker setup.
float CompressionLevel = 100.00;
                                        //Compression level for
    algorithm.

lock_mem(bytes, sizeof bytes);

//Create new protocol.
if (!hosts -> empty())
{
        p = new protocol(hosts, username, password, true,
            server_type, DEFAULT_COMM_TO);
}

//Set BPS Timer//
init_showbps();
set_showbps_start_ts();


for(;!do_exit;)
//for(int k = 0 ; k < LOOP ; k++)
{
        //Generate Random Numbers//
```

```
RawNumbers.clear();
RawNumbers.resize(N_REPETITIONS);

generate_raw_numbers(&RawNumbers,SIZE,
    MAX_ITERATIONS,N_REPETITIONS, CompressionLevel)
    ;
if(histogram) {RawNumbers = EHE(RawNumbers,
    N_REPETITIONS);}

//XOR Mapping//
if(xor_transform)
{
        unsigned char byte[sizeof(float)];
        float * f_buf = (float*)byte;

        for(int i = 0; i < RawNumbers.size(); i++)
        {
                f_buf[0] = RawNumbers[i];
                bytes[i] = byte[3] ^ byte[2] ^
                    byte[1] ^ byte[0];        //XOR
                    Each float.
                if (print_details) {printData(
                    bytes[i],RawNumbers[i]);}
        }
}
else    //Linear Transform//
```

```
{
        for(int i = 0; i < RawNumbers.size(); i++)
        {
                bytes[i] = static_cast<unsigned
                    char>(round(RawNumbers[i
                    ]*255.0));
                if (print_details) {printData(
                    bytes[i],RawNumbers[i]);}
        }
}


if (show_bps)
{
        update_showbps(RawNumbers.size());
}
if (bytes_file)
{
        emit_buffer_to_file(bytes_file, bytes,
            RawNumbers.size());
}

if (p && p -> message_transmit_entropy_data(bytes,
    RawNumbers.size(), &do_exit) == -1)
{
        dolog(LOG_INFO, "connection closed");
```

```
                        p -> drop();
                }

                memset(bytes, 0x00, sizeof bytes);
                set_showbps_start_ts();
        }

        if (show_bps)
        {
                update_forceshowbps(0);
        }



        //Clean-up//
        cudaThreadExit();
        delete p;

}


//Main Function//
int main(int argc, char *argv[])
{
        //Initialize Values//
        int c;
```

```
int log_level = LOG_INFO;

bool do_not_fork = false;
bool log_console = false;
bool log_syslog = false;
bool show_bps = false;
bool print_details = false;
bool xor_transform = false;
bool histogram = true;

char *log_logfile = NULL;
char *bytes_file = NULL;

std::string username, password;
std::vector<std::string> hosts;

//Parse Arguments//
while((c = getopt(argc, argv, "I:hX:P:So:L:l:sndxE")) !=
   -1)
{
        switch(c)
        {
                //Host to connect to.
                case 'I':
                        hosts.push_back(optarg);
                        break;
```

```
//File to read username and password from.
case 'X':
        get_auth_from_file(optarg,
           username, password);
        break;

//File to write PID to.
case 'P':
        pid_file = optarg;
        break;

//Show bytes per second.
case 'S':
        show_bps = true;
        break;

//File to write entropy data to.
case 'o':
        bytes_file = optarg;
        break;

//Log to Syslog?
case 's':
        log_syslog = true;
        break;
```

```
//Set log level.
case 'L':
        log_level = atoi(optarg);
        break;

//Log file location.
case 'l':
        log_logfile = optarg;
        break;

//Do not Fork.  Log to Console.
case 'n':
        do_not_fork = true;
        log_console = true;
        break;

//Print Extra Details
case 'd':
        print_details = true;
        break;

//Perform XOR Transform
case 'x':
        xor_transform = true;
        break;
```

```
                //Do not do EHE//
                case 'E':
                        histogram = false;
                        break;

                //Otherwise, display help.
                default:
                        help();
                        return 1;
        }
}

//Check if there is a username and password//
if (!hosts.empty() && (username.length() == 0 || password.
   length() == 0))
{
        cout << "Hosts:" << hosts[0] << endl;
        cout << "Username:" << username << endl;
        cout << "Password:" << password << endl;
        error_exit("Please select a file with
           authentication parameters (username + password)
           using the -X switch.");
}

//Check if there is a host file to connect to//
```

```
if (hosts.empty() && !bytes_file)
{
        error_exit("No host to connect to or file to write
           to given.");
}

(void)umask(0177);        //Set permissions for file create
   //
no_core();                          //No seg core dumps//

set_logging_parameters(log_console, log_logfile,
   log_syslog, log_level);                  //set logging
   level

//Attempt to fork, if allowed.//
if (!do_not_fork && !show_bps)
{
        if (daemon(0, 0) == -1)
                error_exit("fork failed");
}

write_pid(pid_file);      //Write PID to file

signal(SIGPIPE, SIG_IGN);                  //Signal Pipe ->
   Ignore signal//
signal(SIGTERM, sig_handler);     //Signal Terminate
```

```
        signal(SIGINT , sig_handler);    //Signal Interrupt//
        signal(SIGQUIT, sig_handler);    //Signal Quit//


        //Actual Work//
        main_loop(&hosts, bytes_file, show_bps, username, password
            , print_details, xor_transform, histogram);
        //End Actual Work//

        unlink(pid_file);
        fprintf(stderr, "Program  Successfully Terminated.\n");

        return 0;
}
```