Hardware TRNG Report

Kai Demler (kdemler2), Matt Skrzypczyk (skrzypc2)

UIUC CS 460 Final Project, Spring 2016
Source Material can be Found at: http://github.com/mdskrzypczyk/trng

Contents

1 Introduction

Randomness is used in cryptography extensively to guarantee privacy, confidentiality and or anonymity. Random numbers have provided the basis for computer security for decades. Typically computer systems use "non-deterministic" processes to collect a seed, for example mouse movements, Wi-Fi noise and other "analog" interface devices, which is then used for a pseudo-random number generator. Pseudo-random number generators use "random" seeds and create an output that is indistinguishable from a true random number. The issue lies in the generation of the random seed, as it can be the byproduct of deterministic properties, and thus not random. True random number generators generate random bit-streams based on non-deterministic properties of the physical world.

2 Project Overview

This project aims to observe and research the use of electrical circuitry as a source of entropy for generating random numbers. The goal was to create a device that could collect random bit-streams from the circuit and send them to a host-end client via USB communication. Once these numbers were collected one could use them in deterministic PRNGs to generate longer sequences of random numbers as truly random seeds will yield truly random numbers. Initial testing of the circuit was done using an Arduino Duemilanove, a micro-controller that was designed for prototyping, and upon satisfaction of the circuit performance, the circuit was transferred onto an ATTiny2313 micro-controller for sampling and USB data communication. Bit-streams were statistically analyzed using tools like rngtest, ent, and the NIST statistical test suite. Further analysis of the bit-streams were done using compression tests as well as visual analysis by converting the bit-streams to images. The device was capable of generating bit-streams at roughly 40kbps.

3 Design Description

3.1 Avalanche Noise Generator

An avalanche noise generator utilizes the construction of BJT transistors and the quantum behavior of electrons to create white noise signal. When the collector pin of the transistor is hanging and a large voltage is applied between the gate and emitter the electrons can be made to tunnel through the substrate material of the transistor. This leaves holes for mobile charge carriers to occupy and subsequent electrons that try to fill these holes will be "bumped" into tunneling through the material unpredictably. This is where "avalanche" breakdown derives its name, from the avalanche behavior of electrons cascading through the transistor[6]. The resulting small signal behavior can be amplified into a larger space for finer grained sampling. A circuit diagram below shows the design used for this project.

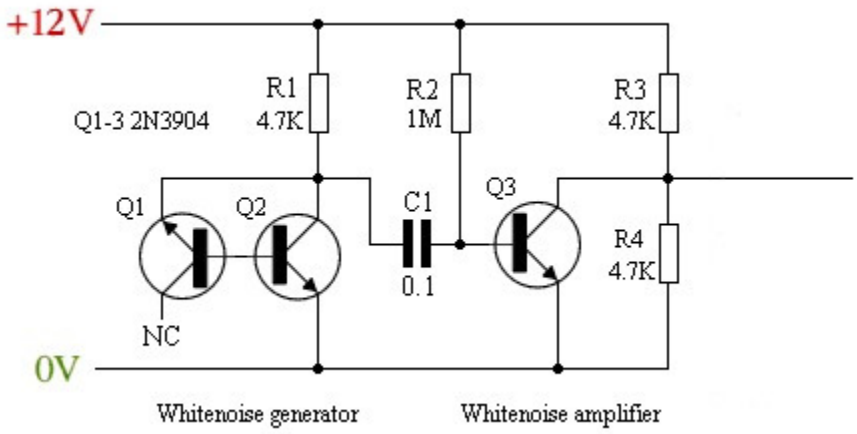

Figure 1: White noise generator circuit for entropy source created by Aaron Logue[7]

Below are screenshots from an oscilloscope used for signal verification. The first three images show the signal on different time resolutions vs. the reference voltage used for distinguishing between a logical 1 or 0 on the microcontroller.

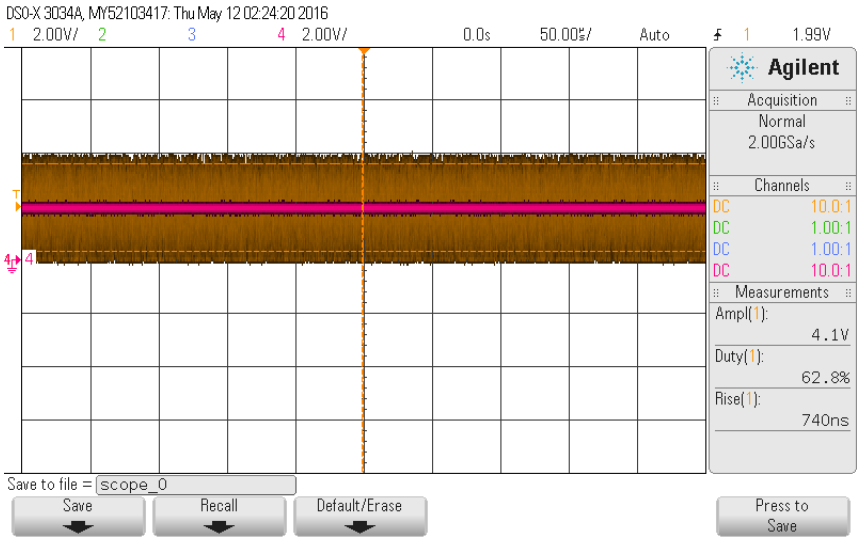

Figure 2: White-noise signal (yellow) vs. reference voltage

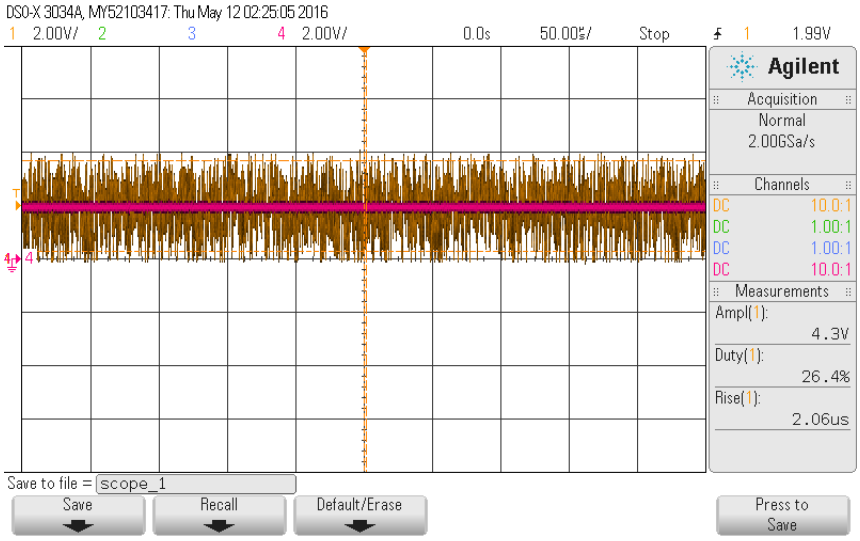

Figure 3: White-noise signal (yellow) vs. reference voltage

Figure 4: White-noise signal (yellow) vs. reference voltage

This image shows the frequency power spectrum of the white noise signal. One can clearly see that the white noise signal observes a constant power spectrum after about 20MHz. One may notice that below 20MHz there is a significant increase in power level but this may be attributed to the low frequencies used to offset the signal to fit in the voltage range on the microcontroller.

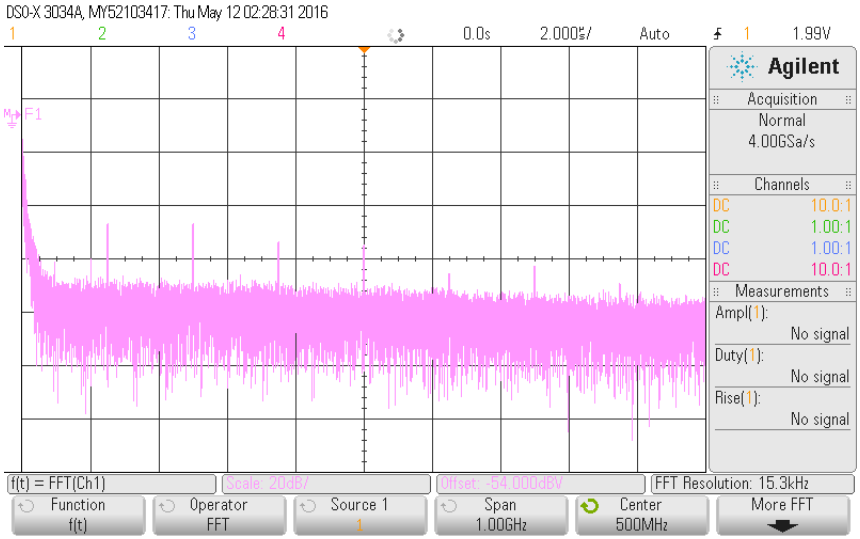

Figure 5: White-noise signal frequency spectrum

3.2 ATTiny2313 Micro-controller

The ATTiny2313 is an inexpensive, low-power CMOS 8-bit micro-controller containing 2KB of In-System programmable flash memory and 128 bytes of RAM. While the micro-controller does not have pins that are directly capable of measuring analog voltage signals it does have two pins that are used for voltage comparison that set internal registers[5]. The ATTiny2313 was chosen as the sampling/communication programs did not need to be lengthy and the remaining RAM space was enough for storing any sampled data for transmission to the host. USB communication was handled using V-USB[3], a USB library for AVR micro-controllers that provides a very simple interface as well as libusb[2] for client side communication. The 2313 takes discrete samples of its internal Analog Comparator Output (ACO) for indication of a logical 0 or 1 on the white-noise signal and constructs 64 byte packets of data to send to the linux client. Once the linux client begins receiving this data it can be piped into /dev/urandom or into files for analysis.

4 Performance Analysis

4.1 Statistical Analysis

Rngtest, ent, and the NIST statistical test suite were used to perform statistical analysis of the generated bit-streams. rngtest is a Linux program that operates on 20,000 bit blocks at a time and runs FIPS 140-2 tests on the streams. Upon completion, rngtest displays results of the number of successes and failures found within the entire stream. Successes indicate passing the FIPs tests while failures indicate the opposite. Below is a table summarizing the results on 4 independent bit-streams.[9]

File Size	10MB(1)	10MB(2)	10MB(3)	10MB(4)	10MB(5)
Type	TRNG	TRNG	TRNG	TRNG	PRNG
Successes	3994	3995	3994	3995	3994
Failures	3	2	5	4	5

One can see that the samples from the TRNG device created for this project are equally as good if not better than random number streams from a pseudo random number generator.

Ent is another Linux program used for testing the entropy in a bitstream. While it is usually used in testing pseudorandom number generators, it can be used for analyzing the generated bitstreams. Ent computes entropy, arithmetic mean, monte carlo approximations of pi, and serial correlation coefficient using the input bitstreams and displays statistics. Below is a table summarizing the results on the same 4 independent bitstreams as well as a short summary of each test.

File Size	10MB(1)	10MB(2)	10MB(3)	10MB(4)
Entropy ($\frac{bits}{bit}$)	0.999996	0.99995	0.999993	0.999993
Arithmetic Mean	0.4988	0.4998	0.4984	0.4984
Monte Carlo Pi Error	0.14%	0.14%	0.2%	0.2%
Serial Correlation Coefficient	-0.000016	0.000023	-0.000120	-0.000373

The following test summaries were taken from the ent manpage:
Entropy:The information density of the contents of the file, expressed as a number of bits per character.
$Arithmetic\,Mean$: This is simply the result of summing the all the bits in the file and dividing by the file length.
$Monte\,Carlo$: Each successive sequence of six bytes is used as 24 bit X and Y co-ordinates within a square. If the distance of the randomly-generated point is less than the radius of a circle inscribed within the square, the six-byte sequence is considered a "hit". The percentage of hits can be used to calculate the value of Pi. For very large streams (this approximation converges very slowly), the value will approach the correct value of Pi if the sequence is close to random.
$Serial\,Correlation\,Coefficient$: This quantity measures the extent to which each byte in the file depends upon the previous byte. For random sequences, this value (which can be positive or negative) will, of course, be close to zero[8].

The final statistical analysis performed on the generated samples was the NIST Statistical Test Suite. This suite provides a plethora of statistical tests that compute a P-Value for a given sample. The P-Value is the probability that a perfect random number generator would have produced a

sequence less random than the sequence that was tested, given the kind of nonrandomness assessed by the test. If a P-value for a test is determined to be equal to 1, then the sequence appears to have perfect randomness. A P-value of zero indicates that the sequence appears to be completely non-random[1]. Because the test suite runs so many tests on the data and computes a unique P-Value for each test, the data is summarized with a few of the P-Values as well as the average minimum pass rate for each bitstream when specified to be broken up into 1,000 smaller streams of 10KB.

File Size	10MB(1)	10MB(2)	10MB(3)	10MB(4)
Average Minimum Pass (Out of 1,000)	980	980	980	980
P-Value Frequency Test	0.0001	0.0529	0.171	0.0815
P-Value CumulativeSums Test	0.0295	0.0342	0.31	0.579
P-Value LongestRun Test	0.506	0.005	0.645	0.437

Summaries of the individual tests can be found in the NIST Statistical Test Suite documentation.

4.2 Compression Analysis

The next method used to evaluate the bitstreams was compression analysis. Because most compression mechanisms rely on patterns in data and bitstream frequency in order to efficiently compress data, they fail when the data is random. Below is a table describing a few of the compression results.

File Size	1KB	10.2KB	102.4KB	1,024KB
Compressed File Size	1.2KB	10.4KB	102.7KB	1,024.514KB

Compression was done on the 10MB bitstream samples as well, the amount of data increase was on the order of hundreds of bytes so this becomes negligible, but what is important is that the compressed file never *decreased* in size relative to the original file.

4.3 Visual Analysis

Another way to differentiate a random number and non-random numbers is by representing the random bits as an image. To accomplish this, a python script was developed to iterate over an input bit-stream and project each bit as a white pixel for a 1, or a black pixel for a 0. When this is conducted on a truly random bit-stream, the resulting image will look like static with no discernible patterns. A non-random bit-stream may have discernible patterns.

We conducted this visual analysis technique on 4 different data sets of size 2KB; the first is an output from our TRNG circuit, the second is a known random number sample from RANDOM.ORG[4], the third is a pseudo-random sample from /dev/urandom with a random seed, and the fourth is a pseudo-random sample from a PRNG with a non-random seed. The seed that was used in this bit-stream was the current system time of the machine. The images that were constructed from the sample are represented below:

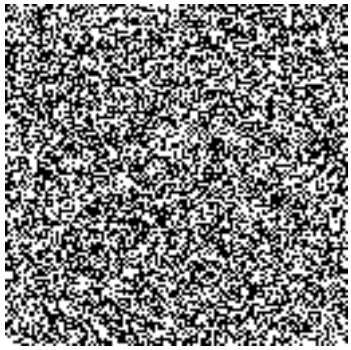

Figure 6: Our TRNG bit-stream

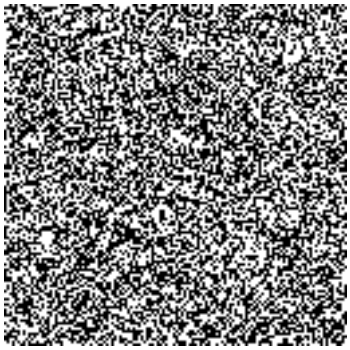

Figure 7: RANDOM.ORG bit-stream

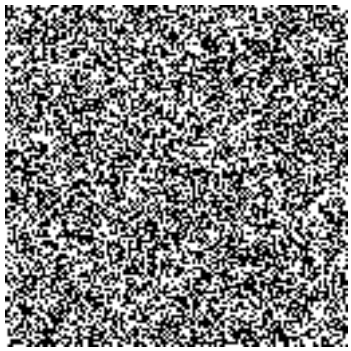

Figure 8: PRNG with random seed

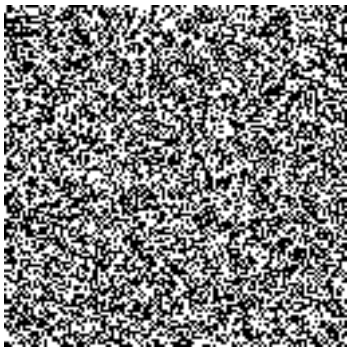

Figure 9: PRNG with non-random seed

After conducting the visual analysis, there appears to be no discernible difference between all four images. Even the PRNG with the system time seed appears to be random. This shows how effective PRNGs are at generating numbers that are "random enough." Furthermore, this test shows that our TRNG was as good as a true random number bit-stream as well as a PRNG. In addition, it shows that visual analysis is a poor method at differentiating a PRNG from a TRNG, or a true random sample.

5 Additional Observations

Before we moved our TRNG circuit from the arduino prototype, the TRNG circuit was able to pass all of RNG tests, however when we transferred the circuit to the SoC board, we began to fail about 4 out of 4000 tests.

One possible source for this reduced entropy is the 5 to 12 volt boost converter that we used. This boost converter was used to increase the 5 volt voltage source from the USB, to 12 volts, which is the voltage required for the TRNG circuit to function. One hypothesis is that when the boost converter increases the voltage, some predictable, periodic or otherwise non-random noise was introduced to the circuit. This would cause the output of the TRNG to be predictable. If this is the case, the noise could be reduced with a noise reduction circuit, to provide clean voltage to the

TRNG circuit.

Another possibility for this introduced error is the reference voltage used to quantize the TRNG output signal. With the Arduino prototype, micro-controller uses a reference voltage to determine if the analog output of the TRNG circuit is greater than or less than that reference. With the SoC design, a reference voltage had to be generated using resistors and the voltage divider rule. The initial voltage is taken from the regulated 3.3 volts, used to power the SoC which is subsequently stepped down to approximately 2.3 volts using a potentiometer. With the potentiometer set to create a reference voltage at approximately the middle of the magnitude of the TRNG waveform, the result fails fewer tests than when the potentiometer is off by a small fraction. This indicates a need to "tune" the device for maximum entropy.

6 Conclusion

The TRNG that we developed was able to produce a random bit-stream at 40kbps. This data can be used as a seed for a PRNG to significantly increase the throughput and or decrease the latency of the random bit-stream. This source of random bits can be used by various computer systems to aide their security with high entropy random seeds. The simple circuit we utilized can be replicated easily and is comprised of low cost, off the shelf components, and can be made and introduced into computer systems easily over USB.

Although our final SoC device did fail some of the randomness tests, the original circuit did not. More time is needed to further investigate the source of this predictability. Once found, the application of this circuit can be used as a standalone TRNG seed for any computer system.

One key limitation in our implementation was the limited bit-rate. While 40kbps of random data would be enough for most applications, many enterprise uses of random data may requiure a higher bit-rate. In our design, the limiting factor was the amount of dedicated RAM available on the SoC. Furthermore, the USB interface would become a bottle neck, once the RAM limitations were met. To solve this issue, a large array of TRNG circuit could be manufactured as a PCI-E device with a theoretical throughput of 16 GBps.

Manufacturing a simple TRNG is valuable way of ensuring that powerful adversary has not tampered with off the shelf TRNGs. This project is meant to illustrate the ease at which one can be self manufactured, to use in many computer systems. As more and more of our personal and private information gets stored on computers and on the Internet, TRNGs or self manufactured TRNGs will become ubiquitous to ensure security of our data.

References

[1] http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html. Nist statistical test suite.

[2] http://libusb.info/. A cross-platform user library to access usb devices.

[3] https://www.obdev.at/products/vusb/index.html. V-usb library.

[4] https://www.random.org/bytes/. True random number service.

[5] http://www.atmel.com/images/doc2543.pdf. Attiny2313 datasheet.

[6] http://www.circuitstoday.com/pn-junction-breakdown characteristics. Pn junction breakdown characteristics.

[7] http://www.cryogenius.com/hardware/rng/. Hardware random number generator.

[8] http://www.fourmilab.ch/random/. Ent, a pseudorandom number sequence test program.

[9] http://www.linuxcommand.org/man_pages/rngtest1.html. rngtest.