For the kid playing space station in the school yard.

Abstract

Cheap micro-controllers, such as the Arduino or other controllers based on the Atmel AVR CPUs are being deployed in a wide variety of projects, ranging from sensors networks to robotic submarines. In this paper, we investigate the feasibility of using the Arduino as a true random number generator (TRNG). The Arduino Reference Manual recommends using it to seed a pseudo random number generator (PRNG) due to its ability to read random atmospheric noise from its analog pins. This is an enticing application since true bits of entropy are hard to come by. Unfortunately, we show by statistical methods that the atmospheric noise of an Arduino is largely predictable in a variety of settings, and is thus a weak source of entropy. We explore various methods to extract true randomness from the micro-controller and conclude that it should not be used to produce randomness from its analog pins.

1 INTRODUCTION

Various aspects in our lives may seem random — so thinking that generating randomness might seem easy at first glance. But when one inquires further one quickly realizes that due to the deterministic nature of CPUs, it is impossible for them to generate random numbers.

However, there is a great need for unpredictable values in cryptography. Kerckhoff's principle states that "a cryptosystem should be secure even if everything about the system, except the key, is public knowledge". Almost all encryption schemes rely on the notion of secret keys so those keys must be generated in an unpredictable way, or else the encryption scheme is useless. Examples of this are the keystream in a one-time-pad, the primes in the RSA algorithm and the challenges used in a challenge-response system [8, 1]. Many secure encryption protocols use nonces (numbers used once) to add "noise" to messages [1]. If these numbers are predictable, the nonces do not serve much purpose.

Since regular computers are unable to produce truly random numbers, psuedorandom number generators (denoted PRNG) are the name of the game. A PRNG is a one-way function f the generates random sequnces, of either integers or bits, from an intial seed s and then applies the function iteratively to generate the sequence [8]. In a cryptographic system, a weak source for the seed weakens the whole system. It may allow an adversary to break it, as was perhaps most notably demonstrated by breaking the method that the Netscape browser used to seed its PRNG [5].

Thus a PRNG can only be random if its seed is truly random and its output is only a function of the seed data, the actual entropy of the output can never exceed that of the seed. However, it is generally computationally infeasible to distinguish between a good PRNG and a perfect RNG. A true random number generator (TRNG) uses a non-deterministic source to produce randomness e.g. measuring chaotic systems in nature like thermal noise, shot noise or flicker noise, which are all present in resistors [6]. Using background radiation and a Geiger counter is an appealing option, but expensive[1] and thus unavailable for the public.

[1]A simple search on Amazon.com reveals that a USB connected Geiger counter costs about $300

Figure 1: Arduino Duemilanove

The Arduino is a free and open-source electronics single-board micro-controller with an Atmel AVR CPU. There are several different versions of the board available[2], but we used the Arduino Duemilanove[3] board (with the ATMega328 [3] micro-controller) for this research. The Arduino toolkit has the `analogRead` function that reads from a given analog pin on the board and returns a 10-bit integer. This function maps input voltages between 5 and 0 volts to integers in the range [0..1023]. This is what we tried to use in order to extract entropy.

Micro-controllers like the Arduino are heavily used in e.g. sensor networks [10] where data integrity is a key issue. It follows that the demand for high quality entropy is rather high in those situations.

The Arduino Reference Manual suggests that reading from an unconnected analog pin gives a “fairly random” number [2], ideal for seeding the `avr-libc` PRNG[4]. We will later show that the numbers are generally not random, and that the reading from an unconnected pin provides very limited entropy. We will also show that building a RNG with the Arduino is infeasible and that if you follow the Arduino Reference Manual, the sheer lack of possible seeds makes it relatively easy for an adversary to guess the seed. We will provide a proof of concept tool for doing such guesswork automatically.

We also attempt to build a random bit generator from the Arduino (without adding extra hardware). We will pose and discuss a few algorithms and discuss how they perform statistically speaking testing and how they fare at extracting entropy. Ultimately, we were unable to identify any such method which rises concerns over the use of Arduino as a TRNG.

[2]See: `http://arduino.cc/en/Main/Boards`

[3]See: `http://arduino.cc/en/Main/ArduinoBoardDuemilanove` for full specifications

[4]Archival of this claim: `http://web.archive.org/web/20110428064453/http://arduino.cc/en/Reference/RandomSeed`

1.1 Contributions

The contributions of this work are the following:

- We implement the monobit, poker and runs statistical tests in the Python programming language, as well as code that exposes an Arduino to these tests. (Section 3.2.)
- We provide a program that given a sequence from the `avr-libc` PRNG seeded with a value from the `analogRead`-function on an Arduino, determines the seed value. It was done by first analyzing data from the Arduino and building a probability distribution of the values. The program either collects data directly from the Arduino first or can be supplied with a data set. We supply a typical data set with the code. This includes an implementation of the `avr-libc random` function. (Section 5.2.)
- We rebut the claim made by the Arduino manufactures that `analogRead` returns "fairly random" integers [2]. (Section 5.1.)

All of the Ardrand code is free software and is maintained at `http://gitorious.org/benediktkr/ardrand`

2 Related Work - Background

Hardware random number generators have been designed with various methods. The search for external entropy has lead researchers down imaginative paths.

Air turbulence in hard drives [4], which is proven to be a chaotic phenomenon, has been used as a source for random numbers. The raw disk times where both structured and correlated. The authors used the Fast Fourier transform algorithm to remove bias and correlation. The worst case observed bitrate was 100 bits/minute.

Intel CPUs contain an on-board RNG [6] chip. It samples thermal noise, shot noise and flicker noise, all of which are present in resistors. The voltage measured across undriven resistors is amplified, but these measurements are correlated to enviromental charasteristics, such as electromagnetic radiation, temperature and power supply fluctuations. The random source used is derived from two free-running oscillators, one fast and one much slower. The thermal noise is used to modulate the frequenciy of the slower clock. The erratic ticks of the slow clock is then used to trigger measurements of the faster one.

Drift between the two clocks is used as a source for binary random digits. The initial measurements are then processed by the von Neumann box. On average, one bit is generated for every 6 raw binary samples.

The limitations of using hard drives as a source of entropy is that not all computers have hard drives, for instance special-purpose hardware like routers or switches. As solid state drives (SSD) become more available, not even all general purpose computers have spinning hard drives any more. Routers and other special purpose hardware do not have exeternal sources for entropy, other than network traffic, which may be observeable or even controllable by an adversary.

One example of this is the OpenWRT router [12]. Since it is based on Linux, it uses the Linux Random Number Generator (LRNG), `/dev/random` and `/dev/urandom`. It provides cryptographic services such as SSH, SSL and wireless encryption. It lacks entropy sources other than network interrupts. It has no mouse or keyboard, so it is impossible to use any user interaction to collect entropy. Although this is not part of the LRNG itself, almost all distributions include a script that saves the state of the LRNG between reboots. This is done so that when the operating system starts, the LRNG has a fresh starting state. The OpenWRT distribution does not do this and thus the LRNG state is reset to a predictable state on every reboot, only determined by time of day and a constant string. This example demonstrates that there is need for external entropy sources.

3 Theoretical Considerations

We begin by making the notions of "fairly random" and "statistically random" more precise by defining statistical tests for sequences of integers. Let us first define a few terms, following the exposition by Menezes et al. [8].

Definition 1. *A random bit generator (RBG) is a device or algorithm that outputs a sequence of statistically independent and unbiased binary digits.*

A random bit generator can easily be used to generate random numbers. To obtain an integer in the interval $[0, n]$ we can simply generate $\lfloor \lg n \rfloor + 1$ bits and cast over to an integer. If the result exceeds n, one option is to discard it and generate a new number.

Definition 2. *A* pseudorandom random bit generator *(PRBG) is a deterministic algorithm or program that given a truly random binary sequence of length k, outputs a binary sequence that appears to be random. The input to the PRGB is called the* seed*, while the output is called a* pseudorandom bit sequence*.*

Note that the output from a PRBG is not random in the colloquial sense of the word. Given the deterministic nature of the algorithm, it will always produce the same sequence for any given seed value.

Definition 3. *Let s be a binary sequence. We say that a* run *in s of length n is a subsequence consisting of either n consecutive 0's or n consecutive 1's. A run is neither preceded or proceeded by the same symbol. We call a run of 1's a block and a run of 0's a gap.*

Definition 4. *Let s be a binary sequence of length n such that $s = s_0, \dots, s_{n-1}$ and let p_i be the probability that $s_i = 1$ for any i. Way say that the generator generating s is biased if $p_i \neq \frac{1}{2}$.*

Determining what is random and what is not is a deep philosophical question — proving mathematically that a generator is indeed generating random bits is impossible [8]. Measuring randomness is as much a philosophical question as it is a mathematical one. There are however statistical tests that allow us to detect certain weaknesses a RBG might have. Note that just because a bit sequence from a generator is accepted by the statistical tests, there is no guarantee that it is indeed random. On the other hand, if it is rejected, we can say with certainty that it is non-random. In other word, when a bit sequence is "accepted" it really is "not rejected".

3.1 Statistical significance

We interpret the results of the statistical tests by means of the χ^2-distributions. It is used in the common χ^2-tests to assess the goodness-of-fit. The χ^2 distribution with k degrees of freedom is given by

$$f(x,k) = \begin{cases} \frac{1}{2^{k/2}\Gamma(k/2)}\, x^{k/2-1}e^{-x/2}, & x \geq 0; \\ 0, & \text{otherwise.} \end{cases}$$

where Γ is the gamma-function, given by

$$\Gamma(n) = (n-1)!.$$

Then we can take our observed data and find an χ^2 statistic, denoted X^2, such that

$$X^2 = \sum_i^k \frac{(O_i - E_i)^2}{E_i}$$

for all i, where E_i denotes the expected number of occurrences and O_i denotes the observed number of occurrences. Then the number X^2 tells us about the significance of the test, given a significance level α. This is usually done by means of a table of percentiles.

The degrees of freedom is the number of variables that are free to vary. It is worth noting that if we have m different values in our calculations, we can often figure out the m^{th} variable from the $m-1$ other values, so then we would have $k = m-1$ degrees of freedom. This is often the case for our tests, such as the Monobit test we will define below.

3.2 Statistical tests

Here we present a few statistical tests we used. We measured against the specifications set forth in FIPS-140-1 [9, 8] rather than selecting the significance levels ourselves. The motivation is that the FIPS document effectively sets a standard for the tests to satisfy and we therefore have something to measure against. There are several others tests available and NIST has published paper [7] that outlines a few tests such as the `DIEHARD`[5] test suite, the tests outlined by Donald Knuth in The Art of Computer Programming and the Universal Statistical Test by Mauer [11]

Let $s = s_0, s_1, \ldots, s_{n-1}$ be a binary sequence of length n. A single bitstring of length $n = 20000$ from our generator is subjected to each of these tests. If any one of the tests fail, we conclude that the output of our generator is non-random.

3.2.1 Monobit test

In a random bit sequence, one would expect that the number of 1's and 0's are about the same. This test gives us a statistic on this distribution. Let n_0 denote the number of 0's and n_1 the number of 1's. We then find the statistic

$$X_1 = \frac{(n_0 - n_1)^2}{2} \tag{1}$$

[5]See `http://stat.fsu.edu/~geo/diehard.html`

which approximately follows a χ^2 distribution with 1 degree of freedom (given n and n_0 we can easily figure out n_1).

3.2.2 Poker test

The poker test tests for certain sequences of five numbers (bits) at a time, similar to a hand in poker. In a random sequence we would expect that each hand would appear approximately the same number of times in s. Let m be a positive integer such that

$$\left\lfloor \frac{n}{m} \right\rfloor \geq 5 \cdot 2^m$$

and let $k = \lfloor \frac{n}{m} \rfloor$. We divide the sequence s into k disjoint parts of length m and let n_i denote the number of sequences of "type" i.

For a binary sequence $s_i \in s$, where $|s_i| = m$, we let n_i be the number of sequences where i equals the decimal representation of s_i. Note that $0 \leq i \leq 2^m$.

The statistic used is then

$$X_3 = \frac{2^m}{k} \left(\sum_{i=1}^{2^m} n_i^2 \right) - k \tag{2}$$

which approximately follows a χ^2 distribution with $2k-2$ degrees of freedom (df).

3.2.3 Runs test

The runs test determines if the number of runs (see *Definition 3*) in s is what is expected of a random sequences. The expected number of gaps, or blocks, of length i in a sequence of length n is

$$e_i = \frac{n - i + 3}{2^{i+2}}.$$

Let k be equal to the largest integer i for which $e_i \geq 5$, or $k = \max_i e_i \geq 5$. Let B_i, G_i be the number of blocks and gaps, respectively, of length i, for each $1 \leq i \leq k$. The statistic used is then

$$X_4 = \sum_{i=1}^{k} \frac{(B_i - e_i)^2}{e_i} + \sum_{i=1}^{k} \frac{(G_i - e_i)^2}{e_i} \tag{3}$$

which approximately follows a χ^2 distribution with 2k-2 degrees of freedom. We note that this exactly finds the χ^2 statistic since the number of runs is the sum of all gaps and blocks.

3.3 FIPS140-1 bounds

We use the FIPS-140-1 bounds [9] for the tests of our Arduino RBG. Let s be a bit sequence of length 20,000. The documents states explicit bounds as follows:

Monobit test The test is passed if $9.654 < X_1 < 10.346$ and the number n_1 of 1's should satisfy $9654 < n_1 < 10346$. Should follow a χ^2 with 1 degree of freedom.

Poker test The statistic X_3 is computed for $m = 4$ and the test is passed if $1.03 < X_3 < 57.4$. Should follow a χ^2 with 15 df.

Runs test We count the number of blocks and gaps of length i — B_i and G_i respectively — in the sequence s, for each $1 \leq i \leq 6$. For the purpose of this test, runs longer than 6 are truncated to length 6 [9]. The test is passed if the number of runs is each within the corresponding intervals below in table 1. The bounds must hold for both blocks and gaps, all 12 counts must lie within the bounds. The distribution should follow a χ^2 with 16 df.

Length of run	Required Interval
1	2267 - 2733
2	1079 - 1421
3	502 - 748
4	223 - 403
5	90 - 223
6	90 - 223

Table 1: Required intervals for runs test as specified by FIPS-140-1

Long runs test The long runs test is passed if there are no runs of length greater than 34 in the bit sequence s.

3.4 Decorrelation with the von Neumann box

Decorrelation is a term that refers to reducing autocorrelation, the similarity between observations as a function of the time separation between them. This should not be observed in a random sequence, since the very definition of randomness implies differences in the sequence. A source of randomness may be faulty in that the output of it is either biased or correlated.

Suppose that the probability that a RBG generates a 1 with a probability p and a 0 with probability $1 - p$, where p is unknown but fixed. We group the output of the generator into pairs of two bits. The pairs 00 and 11 are discarded, and a 10-pair is transformed to a 1-bit while a 01-pair is transformed into a 0. This procedure is called the von Neumann-corrector [8, 6] or von Neumann-box.

3.5 Algorithms used to try to extract entropy from the Arduino

We implemented several algorithms in our search for entropy. These are descriptions of our algorithms.

The `Mean-RAND` algorithm is implemented by keeping a list of the k last values and their mean. Then we compare the new reading to the mean and evalute to 0 if it is less, otherwise 1. To remove bias and reduce correlation we run it through the von Neumann-box.

Listing 1: The `Mean-RAND` algorithm in Python esque pseudocode

```
def meanrand(n):
  buf = deque([0]*k)
  for i in [0..k]:
    buf.push(analogRead())

  meanval = sum(buf)/len(buf)

  for i in [0..n]:
    meanval -= buf.pop()/k
    buf.push(analogRead())
    meanval += buf[-1]/k
    m = ceil(meanval)

    yield vNbox(1 if analogRead() > m else 0)
```

The `Updown-RAND` algorithm first reads an initial value v_0 which is then used to determine if the next bit value v_1 is 1 if $v_1 > v_0$ and 0 otherwise. We do this twice, i.e. we collect $v_{1,0}$ and $v_{1,1}$ and compare them with the von Neumann box until we obtain a legit bit. This algorithm showed low performance and bandwidth, and has consistently failed the statistical tests.

Listing 2: The `Updown-RAND` algorithm

```
def updownrand(n):
  v0 = analogRead()
  for i in [0..n]:
    yield vNbox(1 if analogRead() > v0 else 0)
```

The `MixMeanUpdown-RAND` algorithm acquires one bit from `Mean-RAND` and one from `Updown-RAND` and XORs them together to produce a new bit. Since this method is dependent on `Updown-RAND` it performs even worse, both in regards to bandwidth and entropy.

Listing 3: The `MixMeanUpdown-RAND` algorithm

```
def mixmeanupdown(n):
  m = meanrand()
  u = updownrand()
  for i in [0..n]:
    yield vNbox(m.next()^u.next())
```

Let $a = a_9 \dots a_1 a_0$ be the binary representation of a 10-bit integer read from the `analogRead`-function on the Arduino. The `Leastsign-RAND` algorithm simply yields the least significant bit a_0. As expected, this algorithm shows greater performance and some promise in regards to randomness. We use the von Neumann-box for decorrelating the output.

Listing 4: The `Leastsign-RAND` algorithm

```
def mixmeanupdown(n):
  for i in [0..n]:
    yield vNbox(analogRead()&1)
```

The `TwoLeastsign-RAND` algorithm works in a very similar fashion. Instead of just using the least significant bit, we use the two least significant bits a_0 and a_1, XOR them together and run through the von Neumann-box. This algorithm has shown the greatest potential for entropy and has also been implemented on the Arduino itself.

Listing 5: The `TwoLeastsign-RAND` algorithm

```
def twoleastsign(n):
  for i in [0..n]:
    yield vNbox(analogRead()&1^(analogRead()>>1)&1)
```

3.6 NIST Security Levels

National Institute of Standards and Technology (NIST, America) has defined [9] four basic security levels for cryptographic modules, such as RBGs and RNGs, as well as explicit bounds for statistical tests a RBG must satisfy. The security levels can be outlined as follows

Security level 1 is the lowest level of security that specifies basic requirements for a cryptographic module. No physical mechanisms are required in the module beyond protection-grade equipment. It allows software cryptography functions to be performed by a regular computer. Examples of systems of level 1 include Integrated Circuit Boards and add-on security products.

Security level 2 adds the requirement for tamper-proof coatings and seals, or pick-resistant locks. The coatings or seals would be placed on the module so that it would have to be broken in order to attain physical access to the device. It also adds the requirement that a module must authenticate that an operator is authorized to assume as specific role.

Security level 3 extends the requirements of level 2 to prevent the intruder from gaining access to critical security parameters within the module and if a cover is opened or removed, the critical parameters are erased.

Security level 4 is the highest level of security. It protects the module from compromise of its security by environmental factors, such as voltage or temperature fluctuations. If one attempts to cut through an enclosing of the module, it should detect this attempt and erase all sensitive data. Most existing products do not meet this level of security.

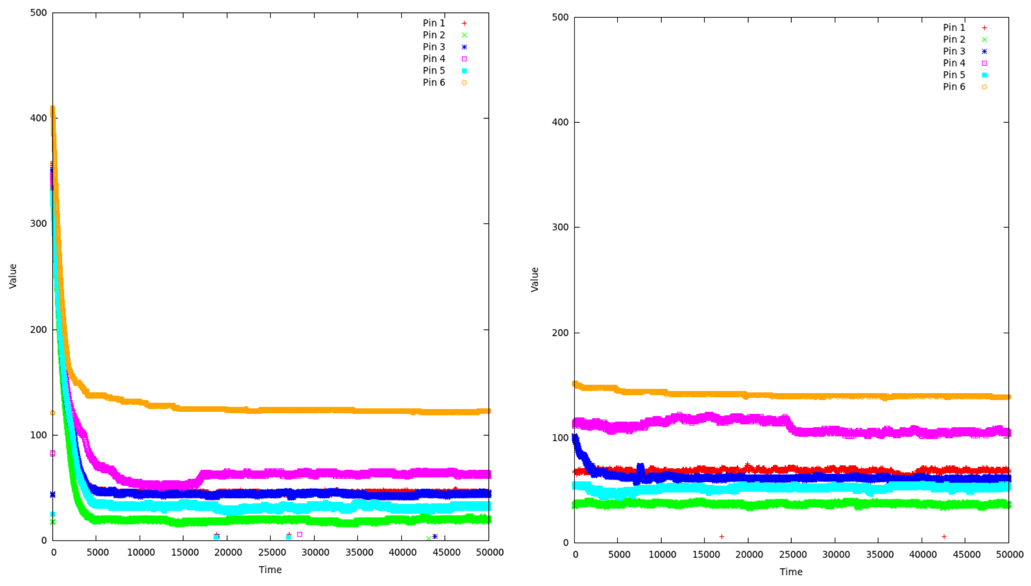
Although we were not aiming for physical security in this scenario, aiming for security level 1 seems like a reasonable decision. Note that in order for a device to conform to any of the security modules it has be able to perform self-tests, both at request and start-up. We implemented the tests in the Python programming language on a general-purpose computer.

FIPS140-1 specifies that the sample must be 20,000 bits, or 2.5KB. But the Arduino Duemilanove only has 2 KB of RAM. Luckily, it has a 32KB Flash memory which could be utilized to implement the statistical tests on the Arduino itself.

4 Experimental Results

We began by analyzing the output of the function `analogRead` on the Arduino in various different settings. We found that the output is dependent on several environmental factors, some of which are unknown to us. Reading from different pins gives different scopes of values, but the behavior is the same. We will show graphs of all pins on one Arduino connected to two computers to back this claim, see Figure 2. The computer to which the Arduino is connected to affects the results. On one computer tested, one of our algorithms produces sequences that were not rejected by the statistical tests.

We want to know if, and by how much, the environment affects the results. We also investigate how and if we can use the `analogRead` output to find entropy and how we test for randomness.



(a) Sample from Ard3 taken on the desktop computer (b) Sample from Ard3 taken on the D620

Figure 2: Readings from Ard3 taken over all pins on both a desktop computer and a D620 laptop.

4.1 Computers and devices used in research

The output of the function `analogRead` on the Arduino is somewhat dependent on the environment in which it resides. We subjected the Arduino boards to different conditions, such as putting it in the freezer or on top of a hot heating element.

These are descriptions of the computers used for the experiments.

- A no-name desktop computer with a Gigabyte GA-MA69GM-S2H motherboard. This machine runs the Debian GNU/Linux testing/wheezer operating system.
- A Dell D505 laptop. This machine runs the Ubuntu GNU/Linux Maverick Meekat 11.04 operating system.
- A Dell D620 laptop. This machine runs the Ubuntu GNU/Linux Maverick Meekat 11.04 operating system as well.

We found that both the laptops showed same or similar behavior, both on raw outputs and statistical testing, but the desktop differed. It is unclear what aspects trigger the deviations but we will discuss this point further in section 4.3. For our research we used three identical Arduino Duemilanove boards with the ATMega 328 micro-controller. To distinguish between them, we call them ard1, ard2 and ard3. These names are also used to distinguish between them in our data samples.

4.2 Analysis of `analogRead`

Our first hypothesis was the space and volume of the area that the Arduino resided in affected the values. If we look at Figure 3 we can see that the environment lays some role and that where you place it definitely has effect on the output. The figure shows readings taken in various different places — in an open space, a closed cupboard, inside a computer case and in a larger open space.

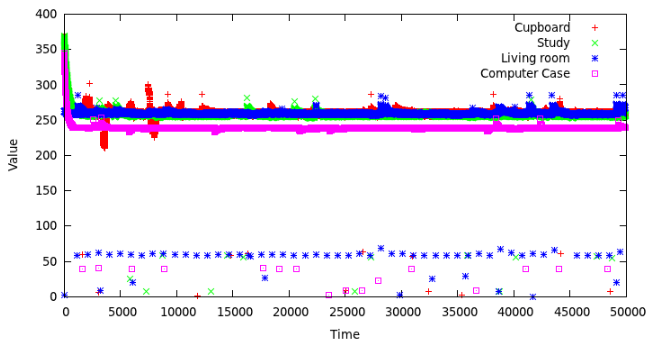


Figure 3: Readings from ard1 connected to the desktop. Samples are taken inside a small cupboard, in a fairly large room, a large living room and inside the desktop computer case itself.

Moreover, by looking at the graph it becomes evident that there is limited entropy available. Note the drop at the beginning; it does not appear for all computers, e.g. specifically when connected to the D620 laptop (see Figure 4). It should be noted that the data originates from the same Arduino device, in the same setting. The only factor is the computer used.

Our experiments have shown that the output is fairly regular and if we look at Figure 5, showing more limited ranges of readings, we see that the structure and apparent lack of entropy. The readings should have been heavily influenced by analog noise [2]. This is further investigated in section 5.1.

Note the interference patterns in e.g. figures 7b and 6 — they show up more clearly in the case of the temperature experiments since we see a much wider range of values. Although we are not sure what causes these patterns, electrical fluctuations are a potential candidate. These patterns might also be a product of the analog pins themselves, or their manufacturing process. The exact physical causes for this phenomenon appear complex and are beyond the scope of this paper.

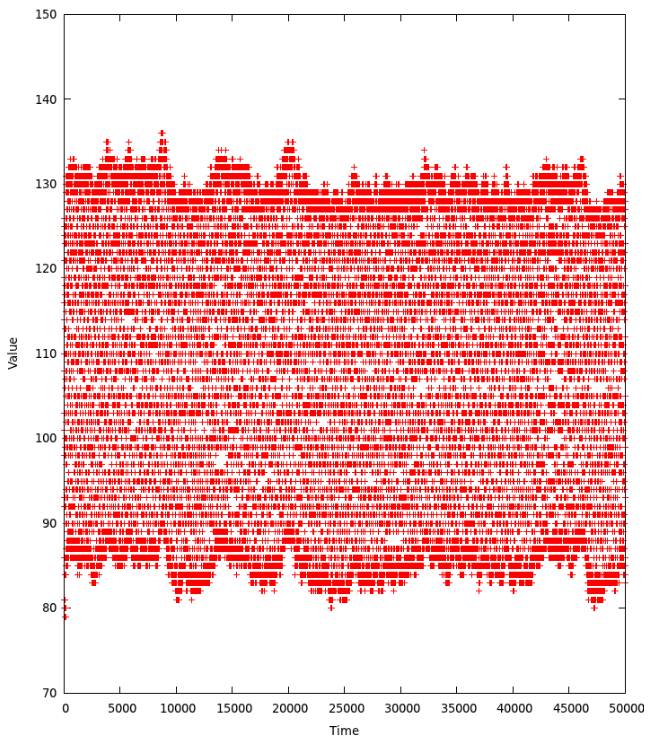


Figure 4: Readings from ard3 connected to the D620 laptop

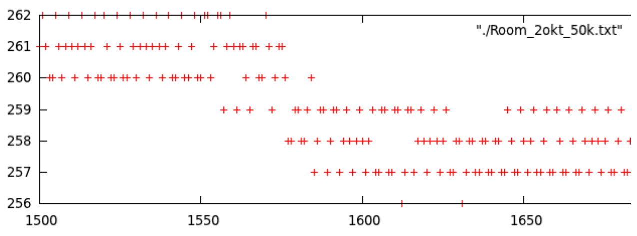


Figure 5: 200 readings from ard1 connected to the desktop computer

4.2.1 Effects of temperature

Temperature is a key environmental factor. We see a much broader range of values when the Arduino is operating in heat or cold. The figures 6, 7b and 7a show the output from `analogRead` in various temperature conditions of the extreme kinds. Note that Figure 7b only has 10000 values, as opposed to the 50000 values in all the other figures. This is because the Arduino simply stops working after a few minutes at $-11°C$. Arduino have not released any information regarding operating temperatures but according to AVR the operating temperatures for the ATMega 328 micro-controller is $-40°C$ to $85°C$ [3]. One of our Arduino boards (ard1) broke after spending 4 hours in the freezer at $-12°C$, so we conclude that some other component(s) on the board survive less cold than the micro-controller itself.

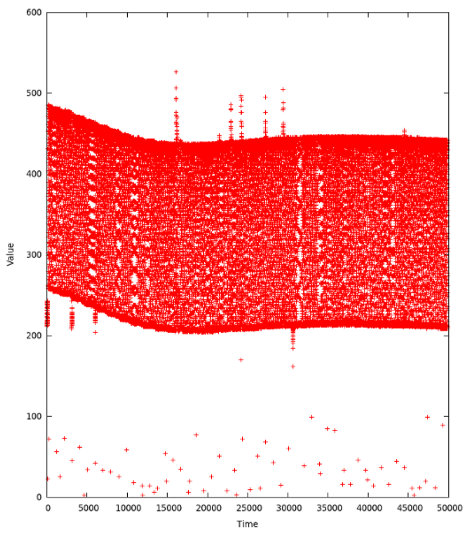
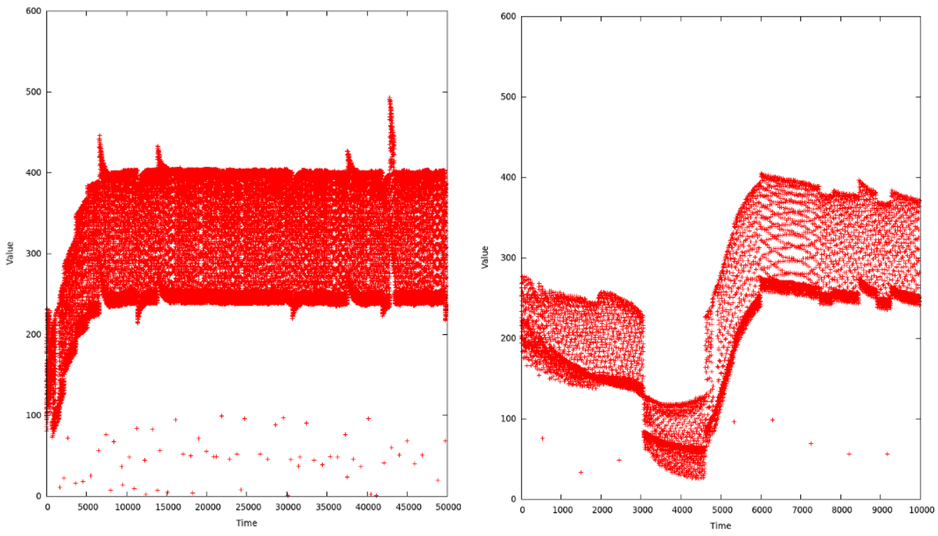


Figure 6: Readings on top of a hot heating element (approx. 40C) connected to a Dell D505



(a) Fridge (approx. 1C) (b) Freexer (approx. -11C)

Figure 7: Readings in cold temperatures with a Dell D505 laptop

4.3 Harvesting entropy and statistical testing

We subject the output of the `RAND` algorithms described in section 3.5 to the statistical tests described in section 3.2. These are our results using the three different computers used in our experiments.

We used a baudrate of 115200 bps, fast available for the Arduino over the FTDI RS232-to-USB connection, in these experiments for maximum bandwidth. The maximum reading rate of `analogRead` is about 10000 times per second

[2]. When we read[6] iteratively from `analogRead` over USB, `pyserial` will raise either an `OSError` or `SerialException`. This happens approximately once every 500 times and reading the `pyserial` source code[7] we find a comment stating that "disconnected devices, at least on Linux, show the behavior that they are always ready to read immediately but reading returns nothing". We therefore conclude that these exceptions are the result of `analogRead` blocking for moment.

These are the results of our `RAND` algorithms using all three computers, subjected to the FIPS boundaries. Each bit sequence tested is $n = 20000$ bits long and each recorded result is the average of three consecutive runs. Green means Accepted and red Rejected.

4.3.1 Results with Desktop computer

Algorithm	Monobit	Poker	Runs	Long runs	Bandwidth
`Leastsign`	$n_1 = 9947$	$X_3 = 869.44$	**Rejected**	**Accepted**	290.55 bps
`Twoleastsign`	$n_1 = 10027$	$X_3 = 1290.05$	**Rejected**	**Accepted**	133.6 bps
`Mean`	$n_1 = 9979$	$X_3 = 149.87$	**Rejected**	**Accepted**	85.34 bps
`Updown`	$n_1 = 8352$	$X_3 = 1959.2$	**Rejected**	**Accepted**	3.87 bps

Table 2: Statistical tests on desktop

4.3.2 Results with the D620 laptop

Algorithm	Monobit	Poker	Runs	Long runs	Bandwidth
`Leastsign`	$n_1 = 10006$	$X_3 = 34.59$	**(Rejected)**	**Accepted**	290.55 bps
`Twoleastsign`	$n_1 = 10027$	$X_3 = 10.36$	**Accepted**	**Accepted**	172.0 bps
`Mean`	$n_1 = 10030$	$X_3 = 4743, 17$	**Rejected**	**Accepted**	25.32 bps

Table 3: Statistical test on D620 laptop

We can see that the `Twoleastsign-RAND` algorithm here produces sequences that are not rejected as being non-random. The `Leastsign-RAND` algorithm is a little less consistent since it is rejected by the tuns tests some of the time.

We have exposed sequences that pass our statistical tests to a statistical test suite made available by NIST[8]. This suite consists of 15 tests, some of which are also implemented by us. We found that when a sequences passes our tests, it will also pass all the NIST tests. This implies that it is possible to generate random bits on the Arduino, but it relies on external factors that are not fully understood by us. Our best guess is that the voltage from the USB connection on the computer influences the regularity of the `analogRead` readings.

As we can see in Figure 4 and 2b, `analogRead` is producing different patterns (and no initial drop) on this computer, compared to the desktop computer.

[6]See `poll.py` in the Ardrand codebase

[7]See `http://www.java2s.com/Open-Source/Python/Development/pySerial/pyserial-2.5-rc2/serial/serialposix.py.htm`

[8]See `http://csrc.nist.gov/groups/ST/toolkit/rng/index.html`

Note the drop on pin 3 in the beginning. Curiously, the `Twoleastsign-RAND` algorithm will fail when we choose pin 3. There is no guarantee that these sequences are truly random, they have just not been rejected as non-random. This shows that using some computers, the Arduino could possibly work as a RBG. But for a device to be a RBG, it has to work using all PC hardware, in all settings.

4.3.3 Results with the D505 laptop

Algorithm	Monobit	Poker	Runs	Long runs	Bandwidth
`Leastsign`	$n_1 = 10033$	$X_3 = 27.4$	**Failed**	**Accepted**	473.32 bps
`Twoleastsign`	$n_1 = 10080$	$X_3 = 18.68$	**(Accepted)**	**Accepted**	240.0 bps
`Mean`	$n_1 = 9980$	$X_3 = 3365.65$	**Rejected**	**Accepted**	27.1 bps

Table 4: Statistical with the D505 laptop

This computer has shown inconsistent test results and sequences produced by it will sometimes be accepted by the poker and runs tests, while sometimes they are not. As before, the causes for this are unknown to us and open for speculations.

5 Breaking the Arduino as a RNG

This section is twofold. We will both show that using the `analogRead` function to seed the `avr-libc` PRNG does not give adequate security and we also exhibit proof-of-concept code that finds such a seed value, given a sequence from the PRNG.

5.1 Refuting the claims made by Arduino

The Arduino Reference Manual [2] states the following in the section about the `randomSeed` function. This claim is at the time of writing found in the manual, and is available via The Internet Archive[9]. The reference manual is only available online.

> "If it is important for a sequence of values generated by random() to differ, on subsequent executions of a sketch, use randomSeed() to initialize the random number generator with a fairly random input, such as analogRead() on an unconnected pin."

After having visually examined the raw output with the graphs in the section above, we clearly saw that the output is very likely non-random and not even "fairly random" as claimed. This would also explain the troubles we had in devising an algorithm that produces random bits.

[9]See `http://web.archive.org/web/20110428064453/http://arduino.cc/en/Reference/RandomSeed`

The first issue with `analogRead` is that it only returns 10-bit integers, since it reads from the 10-bit analog-to-digital converter on the Arduino board[10]. It then follows trivially that if you use `analogRead` to seed the PRNG, there are only $2^{10} = 1024$ seed values for an adversary to explore.

As we can see from Figure 2 there are only roughly 100-400 values that show up, but it is worth noting that using different pins give us different scopes of values. Note the drop when using the desktop but not the D620 laptop.

We exposed the output from `analogRead` to the same statistical tests as our `RAND` algorithms. In order to use the FIPS-bounds to measure against we needed 20 000 bits, or 2000 10-bit integers that we converted to binary. We state the null hypothesis as follows,

$$H_0 = \text{The output from } \texttt{analogRead} \text{ is "statistically random"}$$

and show that the results are statistically significant and we can reject it as non-random. Note that the Arduino developers claim that the output is "fairly random" and not "statistically random". These are the average over three consecutive runs in setting 0.

Statistical test	X statistic	Accepted	Required X interval
Monobit	903.847	**Rejected**	$9.964 < X_1 < 10.346$
Poker	3211.45	**Rejected**	$1.03 < X_3 < 57.4$
Runs	2812.81	**Rejected**	Lengths of runs used

Table 5: Results of the statistical test applied to `analogRead` output

We see that all of the statistics are far off from the FIPS requirements so we can safely conclude that the null hypothesis H_0 is false and `analogRead` is not even "fairly random".

We note that the observed bitrate for reading values directly from `analogRead` is 17531 bps[11].

5.2 Finding the seed

This limited range of possible values from `analogRead` cuts down on search time for the seed. As we have seen, there are only a few hundred values that show up most of the time, although these values may vary. We have designed proof-of-concept code[12] that given a sequence from the `avr-libc` (Arduino) PRNG, finds the seed — assuming it was generated by `analogRead`.

The `avr-libc` PRNG is a Linear congruential generator defined by the recurrence relation[13]

$$X_{n+1} \equiv 7^5 \cdot X_n \pmod{2^{31} - 1}.$$

[10]See `http://arduino.cc/en/Reference/AnalogRead`

[11]Baudrate 115200 bps

[12]Implemented in the file `seedfind.py` in the Ardrand codebase

[13]Resides in `libc/stdlib/random.c` in avr-libc-1.7.1 and a Python implementation is found at `avrlibcrandom.py` in the Ardrand codebase

In order to account for the diversity in values that the Arduino returns in various settings, our implementation inputs either a text file of samples or can connect to an Arduino board and collect fresh samples. To provide the same interface, this is implemented by means of inherited classes in the code.

Let C denote the number of calls a program has made to the PRNG in order to generate a sequence s. Let m be our best-guess or estimation of the unknown C. Our program inputs s and m, as well as a sample source. It then builds a list of values in the range $[0, 1023]$, sorted by the frequency by which they appear in the given sample. Let P denote this list of probability distribution values.

We then create one bidirectional queue for each of the 1024 values in P. Let k be the length of the sequence s. For all $i \in P$ we create a deque with k pseudo random integers derived from i as a seed. Then we iterate over P and generate $k + m$ integers for each deque (holding k values at a time) until we find the sequence.

Here is pseudo-code for our program. The functions `srandom` and `random` are the seeding function for the `avr-libc` PRNG and random function, respectively.

Listing 6: Finding the seed

```
def findseed(s, m, samplesource):
  k = len(s)
  lastk = [deque()]*1024

  P = buildProbDist(samplesource)

  # Expand all the deques by k elements from P[i] as seed
  for i in P:
    srandom(i)
    lastk[i] = deque([random() for _ in range(k)])
    # Did we receive a sequence derived directly from the seed?
    if lastk[i] == s:
      return i

  while True:
    for i in P:
      for _ in range(m+k):
        srandom(lastk[i][-1])
        v = random()
        lastk[i].popfront()
        lastk[i].append(v)
        if lastk[i] == s:
          return i
```

This program has running bounds given by $\mathcal{O}(C)$, since it has a endless `while`-loop, only bounded by C. The running time is thus not bounded by the $\mathcal{O}(m + k)$ loop, since that is only an estimation or best-guess of how long it takes to find the correct seed.

5.2.1 Possible optimizations

While this program runs reasonably fast in practice, one can think of optimizations of the code. Let G a sorted list of observed values in the sample source. Then one variation of the `findseed` program might generate sequences from each value $g \in G$ as seed for some constant time t before moving on to the less likelier values that are in $P - G$, the unobserved values. Thus we would spend t times more time on the more probable values. The while loop of this variation would look like

Listing 7: One possible optimization of the `findseed` program

```
...
  while True:
    # Iterate over a sorted list of the observed values
    for i in G:
      for _ in range(t*(m+k)):
        expand(lastk[i])
        if lastk[i] == s:
          return i
    # Check the unobserved values, but spend less time there
    for i in P-G:
      for _ i range(m+k):
        expand(lastk[i])
         if lastk[i] == s:
           return i

def expand(que):
  srandom(que[-1])
  v = random()
  que.popfront()
  lastk[i].append(v)
```

In order to determine the efficiency of our program, we first pseudo-randomly select an integer d such that $1 \leq d \leq 1000$ with the Python PRNG. Then we generate a sequence $s = s_1, \ldots, s_{d+100}$ with the `avr-libc` and pass the subsequence $s_{d+1}, \ldots, s_{d+100}$ to our program. To seed the PRNG, we use the Python PRNG to select a `analogRead` value from a file of samples collected from an Arduino. We did this 5000 times and our program found the seed in every case, with a mean time of 1.6 seconds[14] spent on each sequence.

6 Conclusions

The primary goal of this project was to investigate the feasibility of using vanilla Arduino Duemilanove boards without add-on hardware as cheap RBG devices, in hopes of building a true hardware random number generator. Early on, we realized that there simply wasn't enough entropy available to build a RBG. Most to our surprise, we found that using some of our computers, the Arduino seems to work as a RBG when using the `Twoleastsign-RAND` algorithm. The resulting strings were not rejected by our implemented statistical tests that had rejected all other sequences we tried. It should be noted that due to the very nature of randomness, we cannot say for sure that it is indeed producing random bits — we can only state that it was not rejected as non-random. But since this is only the case using specific hardware, we must reject the Arduino as a RBG since it is not universal.

[14]Using a computer with an AMD Athlon 64 X2 4000+ Dual Core Processor

Since it is only not rejected using a certain hardware, we have ultimately shown that these boards are not ideal devices for this task. Instead we have refuted the claim made by Arduino themselves and devised a program to find the seed when the onboard PRNG is seeded with `analogRead`, as recommended by Arduino.

In future work, the exact nature of the ports read by`analogRead` should be investigated with respect to the environment or USB connectors. It also remains open to find a cheap, readily available hardware that produce random statistically unbiased random numbers quickly.

7 *

[1] Gary Anthes. The Quest for Randomness. *CACM*, 54(4):13–15, 2011.

[2] Arduino.cc. Arduino Reference Manual, 2011.

[3] ATMEL. 8-bit Atmel microcontroller with 4/8/16/32K Bytes In-System Programmable Flash, Datasheet, 2011.

[4] Don Davis, Ross Ihaka, Philip Fenstermacher. Cryptographic Randomness from Air Turbulence in Disk Drives. *Advances in Cryptography*, 839:114–120, 1994.

[5] Ian Goldberg and David Wagner. Randomness and the Netscape Browser. *Dr. Dobbs Journal*, 21:66–106, 1996.

[6] Benjamin Juan and Paul Kocher. The Intel Random Number Generator, 1999.

[7] National Institute of Standards & Technology Juan Soto. Statistical testing of random number generators, 1999.

[8] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[9] National Institute of Standards and Technology. FIPS PUB 140-1: Security Requierments for Cryptography Modules, 1994.

[10] Rúnarsson, Kristinsson & Jónsson. TSense: Trusted Sensors and Supported Infrastructure, 2010.

[11] Ueli M. Mauer. A universal statistical test for random bit generators. *Journal of Cryptography*, 5:89–105, 1992.

[12] Zvi Gutterman, Benny Pinkas, Tzachy Reinman. Analysis of the Linux Random Number Generator. *Security and Privacy*, pages 385–400, 2006.