Ardrand: The Arduino as a Hardware Random-Number Generator Final Report

Benedikt Kristinsson
Advisor: Ýmir Vigfússon

December 12, 2011

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 analogue pins. This is an enticing application since true true bits of entropy are hard to come by. Unfortunately, we show with 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 in good faith to produce randomness from its analogue pins.

1 Introduction

So much 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. Almost all encryption schemes relay on the notion of secret keys so those keys must be generated in a 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[6, 1].t

Many secure encryption protocols use nonces (numbers used once) to add "noise" in messages[1]. If these numbers are predictable, the nonces do not serve much purpose. Micro-controllers like the Arduino are heavily used in e.g. sensor networks[8] where data integrity is a key issue. It follows that the demand for high quality entropy is rather high in those situations.

Since regular computers are unable to produce truly random numbers, psuedorandom number generators (denoted PRNG) are mostly used. 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 sequcene[6]. 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[4].

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 can be 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).

The Arduino is a free and open-source electronics sigle-board microcontroller with an Atmel AVR CPU. There are several different version of the board available[1], but we used the Arduino Duemilanove[2] board (with the ATMega328[3] microcontroller) 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 is what we tried to use in order to extract entropy. This function maps input voltages.

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[3]. We will later show that this is not true and, and that the reading from an unconnected pin is not very random at all. 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 also attempt to build a random bit generator from the Arduino (without adding extra hardware), but we find that this is infeasible. But we will demonstrate a few algorithms and discuss how they perform exposed to statistical testing and the possibilities of finding some entropy.

1.1 Contributions

The contributions of this work are the following:

- Implementations of the monobit, poker and runs statistical tests in the Python programming language as well as code that exposes and Arduino to these tests. (Section 3.2.)
- A program that given a sequence from the `avr-libc` PRNG seeded with a value from the `analogRead`-function on an Arduino, finds the seed value. It is 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 dataset. We supply a typical dataset with the code. This includes an implementation of the `avr-libc random` function. (Section 5.2.)

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

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

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

- Rebuttal of 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

3 Theoretical Considerations

Let us first define a few terms[6].

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 (turned into a random number generator). If we desire 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 of of length l. 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. 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 1's. Note that 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 $s = s_1, \ldots, s_i, \ldots, s_n$ 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 mathematically what is random and what is not is a very hard task — and proving that a generator is indeed generating random bits is impossible to prove[6]. There are 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, this is not a guarantee that it is indeed random. On the other hand, if it is rejected, we can say that it is non-random. In other word, when a bitsequence is "accepted" it really is "not rejected".

3.1 χ^2-distribution

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 compare 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 and O_i denotes the observed number. 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. One is found in [6, p. 178] and replicating it here is redundant.

The degrees of freedom is the number of values 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} value 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.

3.2 Statistical tests used

Here we present a few statistical tests we used. We measured against the specifications set forth in FIPS-140-1[7, 6] rather than selecting the significance levels ourselves. The motivations for this is that the FIPS document effectively sets a standard for the tests to satisfy and we therefore have something to measure against.

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 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_o denote the number of 0's and n_1 the number of 1's. We then find the statistic

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

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, based on 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

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

and let $k = \lfloor \frac{n}{m} \rfloor$. We divide the sequence s into k non-overlapping 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.

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 is exactly finding 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[7] 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 $9654 < n_1 < 10346$.

Poker test The statistic X_3 is computed for $m = 4$ and the test is passed if $1.03 < X_3 < 57.4$.

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 of length greater than 6 are said to be of length 6[7]. The test is passed if the number of runs is each within the corresponding intervals below in table 1. This must hold for both blocks and gaps, all 12 counts must lie within the bounds.

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). 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 is the von Neumann-corrector[6, 5] or -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 lessen correlation we run it through the von Neumann-box.

Listing 1: The `Mean-RAND` algorithm in Python-ish 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 determins 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 a very 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 one 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 decorrelation purposes as usual.

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 `Leastsign-RAND` algorithm

```
def mixmeanupdown(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[7] 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

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 zeroized.

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 zeroize all sensitive data. Most existing products do not meet this level of security.

Although we were not aiming for physical security in this research, 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 highly 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 in one setting to back this claim.

4.1 Settings and devices used in research

The output of the function `analogRead` on the Arduino is highly dependent on the environment in which it resides. For this research we mainly used three different locations, as described here. We used subjected the Arduino boards to different conditions in these settings, such as putting it in the freezer or on a hot heating element.

Setting 0 The study of the author. An apartment built by the U.S. Navy with electricity converted to Icelandic and European standards.

Setting 1 Garage furnished as a studio apartment in Kópavogur.

Setting 2 Computer Science lab at Reykjavik University.

For this research we used three distinct but identical Arduino Duemilanove boards with the ATMega 328 microcontroller. To distinguish between them, we call them ard1, ard2 and ard3. These names are also used to distinguish between them in our datasamples.

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 1 we can see that it plays some role and that where you place it definitely has effect on the output. It shows readings taken in various places in setting 0 — in an open space, a closed cupboard, inside a computer case and in a larger open space.

What is more, by looking at the graph it becomes evident that there is not much entropy available. Note the drop at the beginning; it does not appear in all settings, e.g. not in setting 1 (see Figure 3). It should be noted that the data originates from the same device, these are purely environmental factors. It could possibly be linked to the curious property that one of the `RAND` algorithms will pass all statistical tests in that setting.

Our experiments have shown that the output is fairly regular and if we look at Figure 2, 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 somewhat regular patterns in e.g. figures 5 and 4 — they show up more clearly in the case of the temperature experiments since we see a much wider range of values.

4.2.1 Effects of temperature

Temperature is a key environmental factor. We see a much broader range of values when operating in heat or cold. The figures 4, 5 and6 show the output from `analogRead` in various temperature conditions of the extreme kinds. Note that 5 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 microcontroller 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 microcontroller itself.

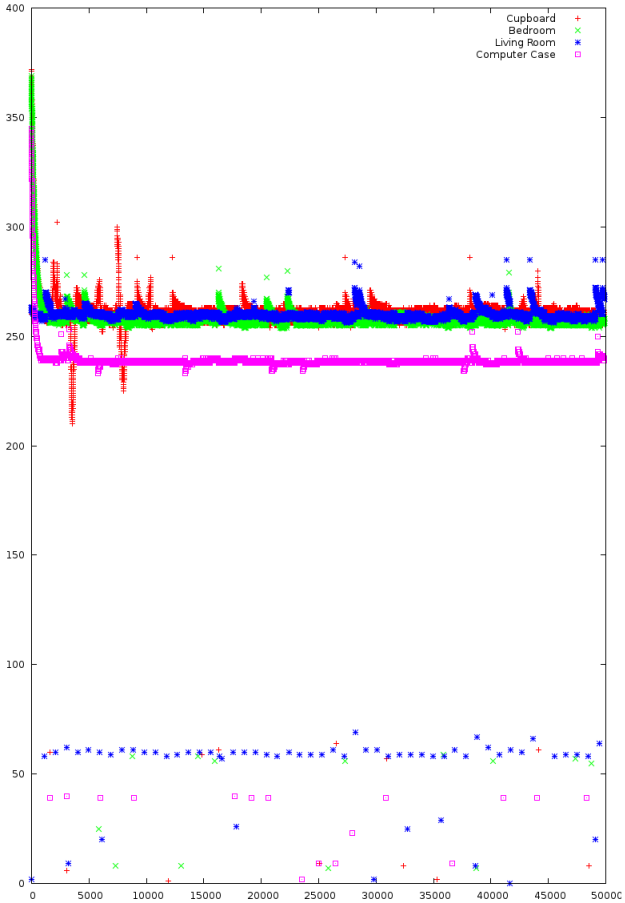
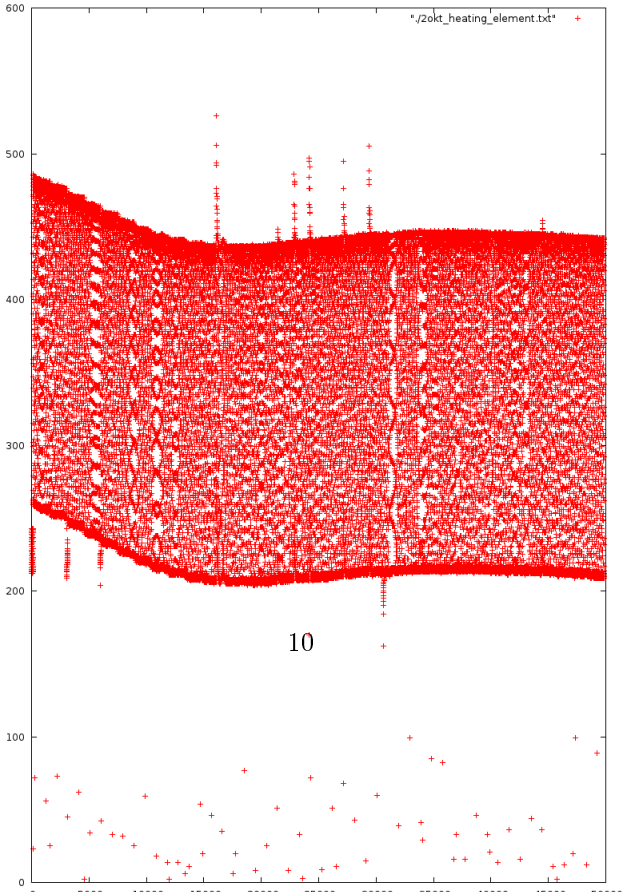
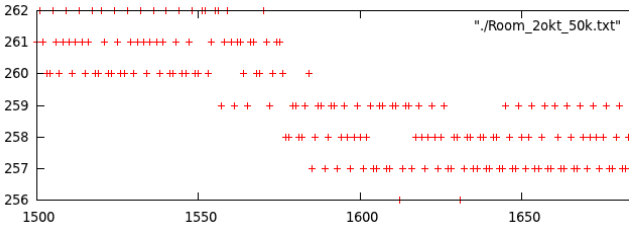
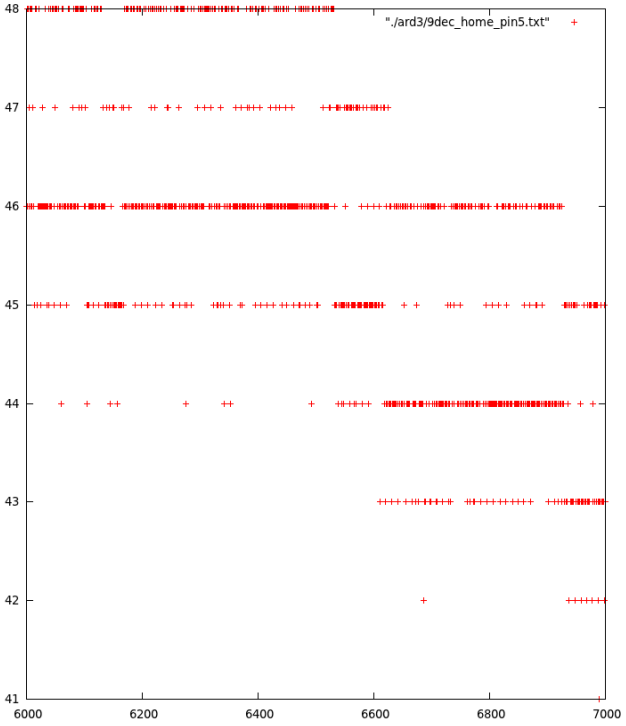


Figure 1: Readings from ard1 at Setting 0, in various places



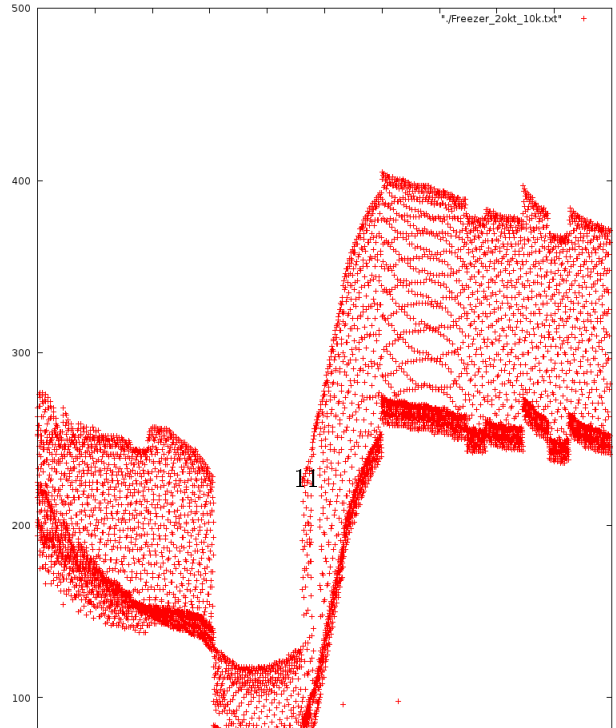


(a) 200 readings from ard1 in setting 0



(b) 1000 readings from ard3 in setting 0

Figure 2: Limited ranges of readings from `analogRead`



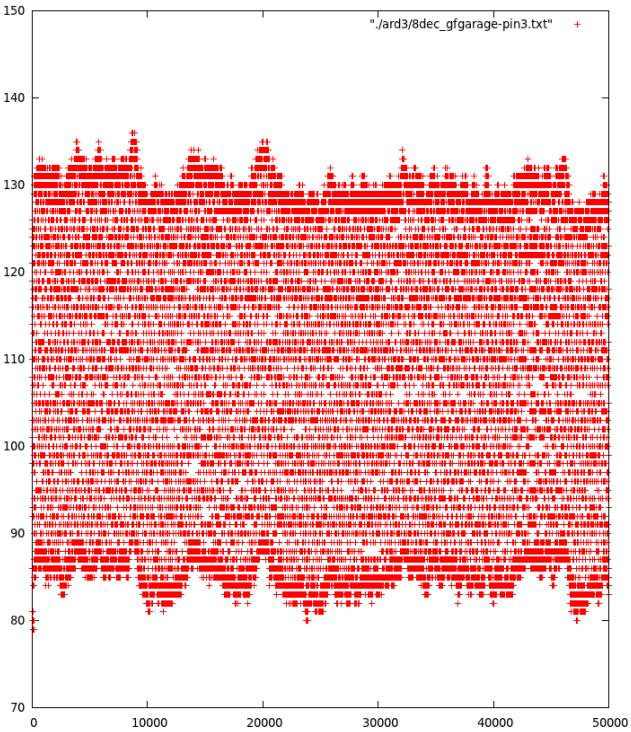


Figure 3: Readings from ard3 taken in setting 1

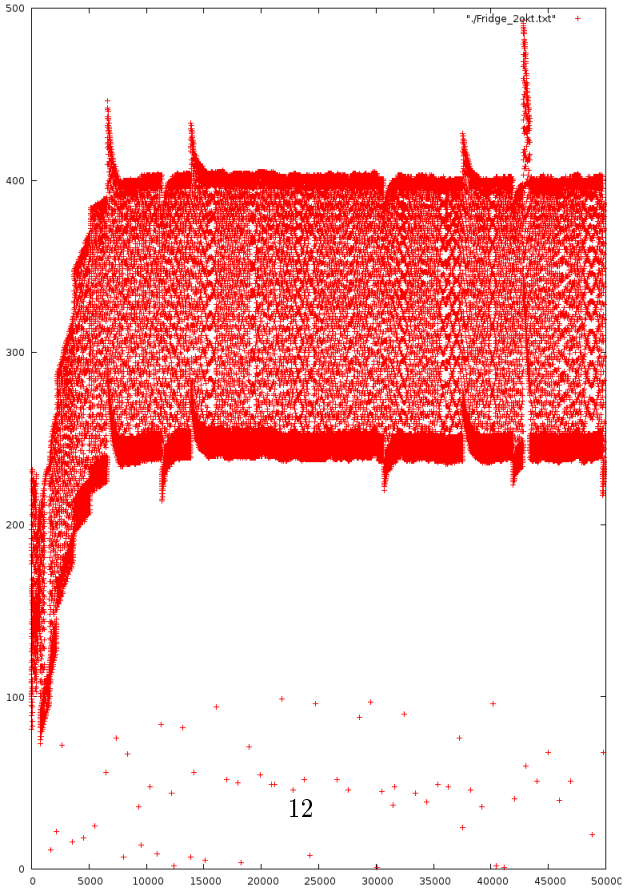


Figure 6: Reading inside a fridge (approx. 1C) at setting 0

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 is a very bad idea 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[4] at `http://web.archive.org/web/20110428064453/http://arduino.cc/en/Reference/RandomSeed`. 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.

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[5]. 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 7 there are only roughly 100 values that show up, but it is worth noting that using different pins give us different scopes of values.

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

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

and show that the results are statistically significant and we can reject it as non-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 2: 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".

[4] http://www.archive.org
[5] See `http://arduino.cc/en/Reference/AnalogRead`

We note that the observed bitrate for reading values directly from `analogRead` is 17531 bps[6].

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[7] that given a sequence from the `avr-libc` (Arduino) PRNG, finds the seed — given that is was chosen by use of `analogRead`.

The `avr-libc` PRNG is a Linear congruential generator defined by the recurrence relation[8]

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

In order to account for the diversity in values that the Arduino outputs 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 distributed values.

We then create one deque 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:
```

[6] Baudrate 115200 bps

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

[8] 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

```
    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, its running time is not bounded by the $\mathcal{O}(m+k)$ loop, 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 unlikelier values that are in $P - G$, the unobserved values. Thus we would spend t times more time on the likelier values. The while loop of this variation would look like

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

```
...
  while True:
    for i in G:
      for _ in range(t*(m+k)):
        expand(lastk[i])
        if lastk[i] == s:
          return i
    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)
```

6 Conclusions

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.

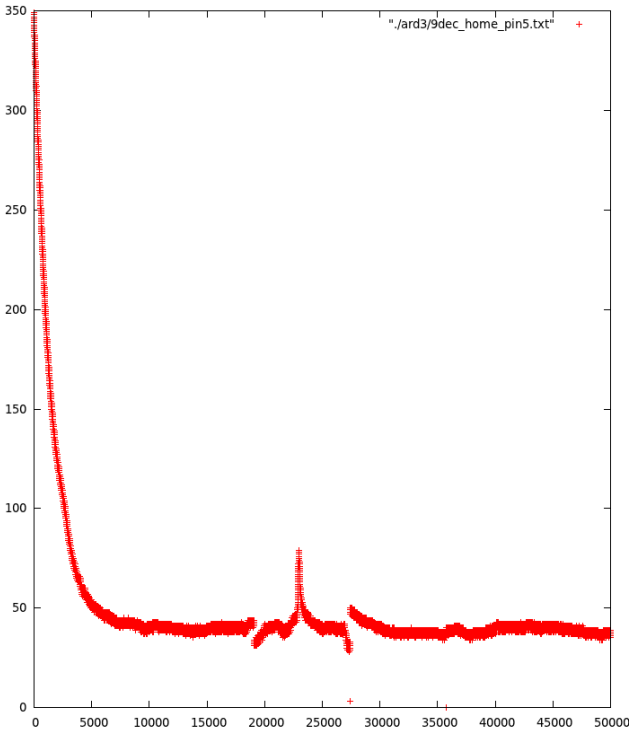
[4] Ian Goldberg and David Wagner. Randomness and the Netscape Browser, 1996.

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

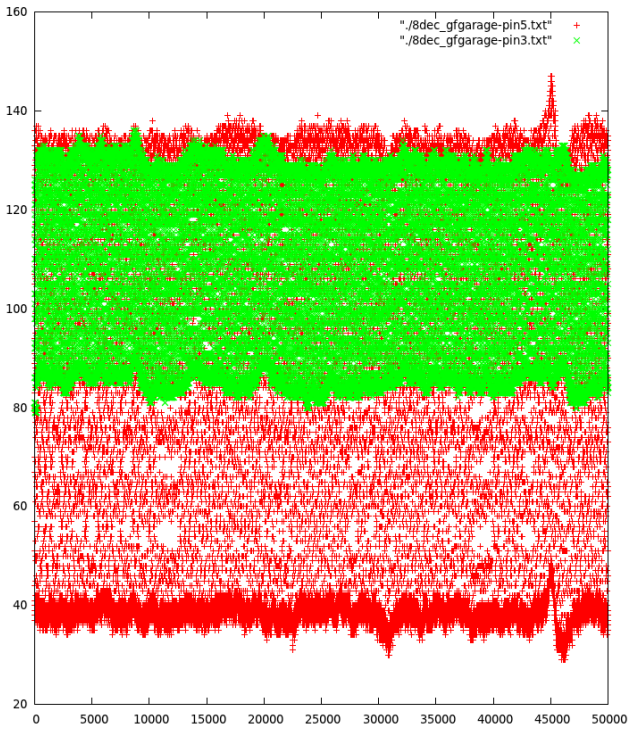
[6] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

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

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



(a) Sample from Ard3 taken in setting 0



(b) Sample from Ard3 taken in setting 1

Figure 7: Readings from our Arduino no. 3 on pins 3 (green) and pin 5 (red). NOTE: home pin3, show same scales