Exploring Differences Between PC Implementation and Popular Consumer Hardware Implementation of an ANSI C Linear Conguential Generator

Darwin Clark
Bainbridge Island High School
Bainbridge Island, WA, United States
darwin.s.clark@gmail.com

***Abstract*—In this work, we ascertain the amount of entropy present in embedded microprocessors. A source of true entropy is often needed to make cryptographically secure connections, both airgapped and within the system. An absence of a sufficient source of entropy can result in insecure connections which can be brute forced, or even have predicted outcomes by an outside attacker.**

I. Background

The explosion of the embedded system world is a computing revolution not even fathomable at the turn of the century . A modern-day consumer can hold in their hand a device with more than 50,000 times the processing power of the top-of-the line processor 30 years ago.

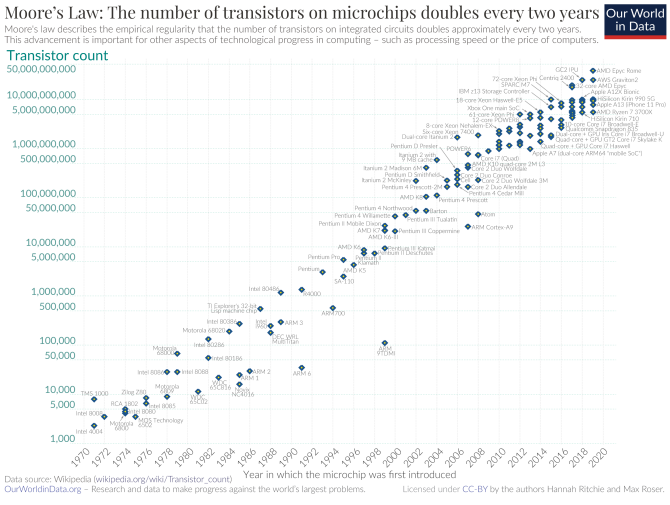

Fig. 1. A diagram showing the growth of mainstream Central Processing Units (CPUs) from 1970 to 2020. Note the non-linear scale on the y-axis

A key impact of this explosion in technology is the miniaturization and widespread distributed of embedded devices. Historically, the need for ”smart” electronic tools has been dominated by analog circuitry. Said circuitry was made with components that were easy to source, cheap, and did exclusively one task. Examples of said devices would be police speed guns, hobbyist HAM radios, or a refrigerator. While modern day versions of these devices are built with digital circuitry, their design would not resemble their inception in any way.

Where this revolution becomes troubling for consumers is the addition of the internet, often shorted to the Internet Of Things (IOT). The fusion of the internet with ’smart’ devices opened them up to the world, but also opened the world up to them. A user now has the ability to access a video feed, the temperature of their freezer, or open their garage door. Unfortunately, it has been shown time and time again that these devices are just as vulnerable as their desktop counterparts.

For better or for worse, modern consumer devices have grown much more complicated from trivial kitchen appliances. Devices such as the Google Home and Amazon Alexa have broadened the liability of the user considerably. These devices have access to financial details, internet usage statistics, and personal photos and correspondence. The consequence for a breach is not limited to just a spoiled chicken.

The vulnerabilities extend much further beyond the digital world. Smart locks and smart garage doors have also become mainstream. These devices allow a user to have automated routines or personalised profiles. Unfortunately, they are filled with the same exploits that have become all too common.

II. Introduction

This paper seeks to explore a very specific niche of consumer device security: The true entropy of Random Number Generators (RNG), Psudo-Random Number Generators (PRNGs), and Hardware Random Number Generators (HRNGs). Finding high-quality sources of entropy for embedded systems can be very challenging considering the limited instruction set, limited budget, and size of the devices. Cryptographically, RNG is chosen as it acts as a backbone for almost all internet-related activity and secure data storage.

A focus will be placed on ensurign crypographic security for the PRNG bitstream on common embedded system microprocessors (Atmel’s AVR328P, STM’s STM32, Expressif’s ESP8266). These Integrated Ciruits (IC) have been chosen for reasons beyond convenience. The AVR line of microprocessors represents a group of cheap, low-power microprocessors often used in research projects or ”low-tech” consumer products. Chips such as the ATTiny45 and ATTiny13 are known for

having a low pin count and current draw. The STM32 represents a mid-range line of chips capable of executing much more complicated instructions (ARM Cortex-M7). Finally, the ESP8266 was chosen as it contains integrated wireless capability (will be used interchangeably with the ESP32), which is common for modern devices.

This paper will explore one PRNG implementation, a Linear Congruential Generator. The reasons for this selection are eplored later in the paper. The generation engines used in this study will act as the baseline for the statistical data measurement. Once selected, they will be implemented on a variety of subsystems (See above list). The raw data will be in the output stream from the engine, stratified by the particular implementation.

While not all embedded systems are connected to the internet, many of them either store, collect, or interpret potentially sensitive data that consumers are not required to have a full understanding of. Such data can include:

- Atmospheric Data
- 802.11 Wi-Fi Credentials
- Personal Computer Credentials
- Video Footage
- Audio Footage
- Location of Homeowner
- Status of Door Locks

III. Prior Work

Given the vast choice of microcontrollers available to any given developer, performing comprehensive PRNG analysis on every individual chip is unreasonable. The topic of cryptographic security outside of the embedded systems field very regularly receives new pieces of literature, including university research papers, amature hacker articles, and NIST standards publications.

Oftentimes research papers are very directly tied to a very niche and singular RNG application. For instance, a 2016 Stanford paper 4 described the research associated with designing a new source of environmental entropy circuit (the Lampert Circuit). A 2017 CMU paper put forth a suite of software called TROMMEL which could be used to analyze an embedded device in its entirety. These papers are examples of the limited scope of security research papers, and while they don't cover the exact implementations implemented here, they contain practices and techniques which will be used to create robust data.

Moving into specifically cryptographic research, there are papers written as recently as 2015 that debunk pseudo-current NIST guidelines for PRNGs.

IV. Methods and Approach (Software)

Part of the reason a LCG was chosen for this experiment was the size that the compiled code occupies. In the world of embedded computing, the amount of available flash space is typically proportional to the computing speed of the processor, and by association battery consumption. With battery life as a common design constraint, it is reasonable for a consumer product designer to select a lower-power (and by extension lower flash size) processor to address cost and battery concerns.

m	a	c	Reference
2^{24}	1140671485	12820163	in Microsoft VisualBasic
$2^{31}-1$	742938285	0	[41]
$2^{31}-1$	950706376	0	[41]
$2^{31}-1$	630360016	0	[70, 134]
$2^{31}-1$	397204094	0	in SAS [149]
$2^{31}-1$	16807	0	[101, 6, 70, 133]
$2^{31}-1$	45991	0	[88]
2^{31}	65539	0	RANDU [58, 70]
2^{31}	134775813	1	in Turbo Pascal
2^{31}	1103515245	12345	`rand()` in BSD ANSI C
2^{31}	452807053	0	[58, URN11]
2^{32}	1099087573	0	[39]
2^{32}	4028795517	0	[39]
2^{32}	663608941	0	[58, URN13]
2^{32}	69069	0	component of original SuperDuper
2^{32}	69069	1	on VAX/VMS [58, URN22]
2^{32}	2147001325	715136305	in BCLP language
2^{35}	5^{13}	0	Apple
2^{35}	5^{15}	7261067085	[64, p.102]
$10^{12}-11$	427419669081	0	`rand()` in Maple 9.5 or earlier
$2^{47}-115$	71971110957370	0	[85]
$2^{47}-115$	−10018789	0	[85]

Fig. 2. A table illustrating common combinations of LCG values and their origin. Sourced from the documentation for `TestU01`

While there are a wide variety of potential values to use within a LCG model, the values selected for this paper correspond to the most popular implementation of a LCG, the BSD ANSI C `rand()` function. These values have the benefit of strong unpredictability, long period length, and commonplace relevance. In figure 2, this is the 10th row. This leaves our mathematical model to be:

$$X_{n+1} = a * X_n + c \mod (m)$$

$$X_{n+1} = 1103515245 * X_n + 2147483648 \cdot \mod (2^{31})$$

Where X_{n+1} represents the next sequential output of the LCG, and X_n represents the current state of the LCG.

External LCG implementation accessible through an API was explored for the software portion of this study (Such as those found in `dieharder` and `TestU01` however they were discarded due to the following reasons:

- Non-Trivial Cross-Compilation Support
- Steep Learning Curve
- To Confirm Researcher Knowledge of the Subject Matter

To ensure ease of use between different microcontrollers, the open-source software PlatformIO was used to trivially transition between different boards. PlatformIO automatically handles serial monitoring, uploading with different protocols (E.G `avrdude` vs. `esptool.py`), and compilation for different frameworks.

Following code examples from open-source implementations of PRNGs, the LCG initially defined a 10,000-integer array to keep track of the state, however this was later removed as it both overran the flash requirements of several

Fig. 3. The entirety of the PlatformIO configuration file used in this study. Note the short length. The only other file that required maintaince was the code loaded onto the microcontroller

```
[env:esp]
platform = espressif8266
board = esp12e
framework = arduino

[env:ard]
board = uno
framework = arduino
platform = atmelavr

[env:seeduino]
board = seeed_xiao
platform = atmelsam
framework = arduino
```

microcontrollers, and was not nearly large enough to maintain a record of reasonable size. The initial goal was to write the state of the array to a file and use that to record data. Later on however, the coed was modified to only store the current and last state of the LCG (X_n and X_{n-1}). This design decision is unique to LCGs, as it is only necessary to know the previous state in order to compute the next state. Below are the two most important code segments from the LCG: `loop()` / `setup()` and `stepLCG()`.

After retrieval of the output data from the microcontroller, it was analyzed using two pieces of software: RStudio and the `dieharder` suite of tests. The `dieharder` group of tests were not the primary analysis tool, due to lack of applicable analysis. This discontinuity will be discussed later in the paper. RStudio is used to provide a graphical analysis of the data.

V. Methods and Approach (Hardware)

Given that the sole purpose of this study is solely to access the computation mechanisms of the aforementioned microprocessors, the physical data collection was not relatively complicated. Shown in figures 5, 6, and 7 are photographs of the experimental setups. Note the linear power supply in the background has been turned off, in order to reduce sources of noise. The reasoning behind the processor selection was discussed in section (II).

You will note that there is a USB amp-meter between the computer and the microcontroller being studied. This was done intentionally to detect any difference In power consumption of the three micocontrollers during data collection, however it was determined that this information is both not relevent to the study and not statistically different between microcontrollers.

VI. Results (Numerical)

The results will be given in two sections: numerically, and graphically. There will be a paragraph discussing the numerical characteristics of the data for each micocontroller, then associated graphs with little or no interpretation.

Seedunio Xiao - Approximately 458000 LCG samples were taken from the ATMEL SAMD processor. The samples

Fig. 4. The LCG Arduino-framework code used in this study.

```
double stepLCG(double in) {
   double output;

   // fmod()
   double temp = (a*in + c);
   // printf("%f \n Pre-Step 1: ", temp);
   output = fmod(temp,m);

   return output;
}

void setup() {
  // put your setup code here, to run once:
  a = 1103515245;
  m = 2147483648; // 2^31
  c = 12345;
  Xn = 2;

  Serial.begin(115200);
  delay(10);
  Serial.println('\n---Starting ANSI C
      LCG---');
  delay(5);
}

void loop() {
  // put your main code here, to run
      repeatedly:
  delay(10);

  XnOld = Xn;
  Xn = stepLCG(XnOld);

  Serial.println(Xn);
}
```

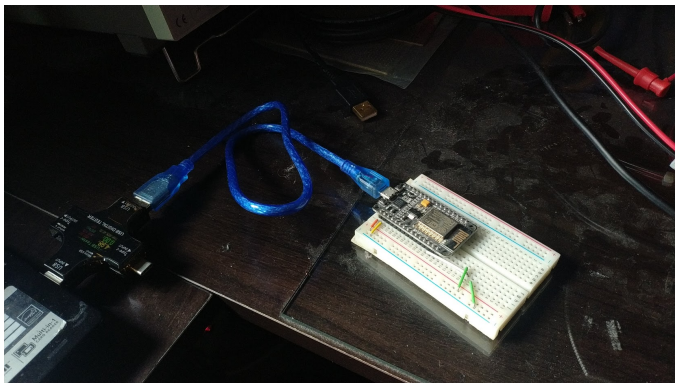

Fig. 5. Experimental Setup for the NodeMCU ESP8266

occupied a filesize of about half a megabyte, 511 Kb. The data collection took approximately 2 minutes, and the firmware occupied 17396 bytes of flash.

NodeMCU ESP8266 - Approximately 10125 LCG samples were taken from the Expressif ESP processor. The samples occupied a filesize of about 0.1 megabytes, 114 Kb. The data collection took approximately 1.5 minutes, and the firmware

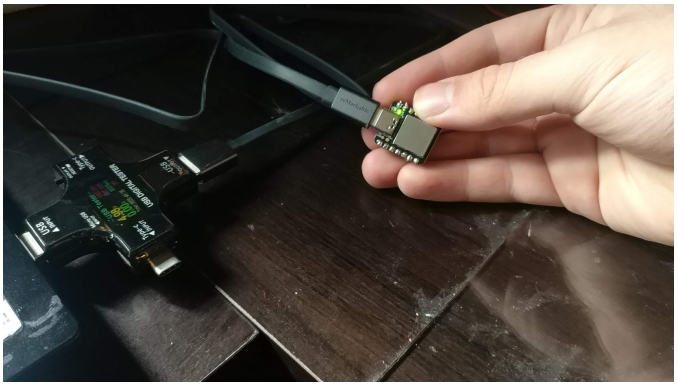

Fig. 6. Experimental Setup for the Seeduino Xaio

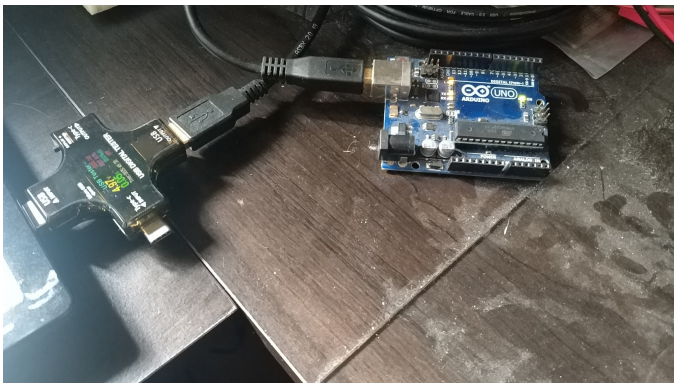

Fig. 7. Experimental Setup for the Arduino Uno

occupied 263925 bytes of flash.

Arduino Uno - Approximately 53900 LCG samples were taken from the Arduino AVR processor. The samples occupied a filesize of about 0.2 megabytes, 207 Kb. The data collection took approximately 5 minutes, and the firmware occupied 3180 bytes of flash.

VII. Results (Graphical)

Now we will present the graphical representations of the data. The graphical forms will include: A scatter plot (in both 2D and 3D) comparing the previous state of the LCG to the next state, and a histogram of the results distribution. The data is located in figures 8 to 16.

The raw `.csv` data files will be accessible in the bibliography.

VIII. Analysis and Interpretation

To begin, we will describe both a PRNG that is considered "poor", and a PRNG that is considered "good" to illustrate how our conclusions about the output of a particular PRNG are reached.

> "Passing many tests improves one's confidence in the RNG, although it never proves that the RNG is foolproof. In fact, no RNG can pass every conceivable statistical test. One could say that a **bad** RNG is one that fails simple tests, and a good RNG is one that fails only very complicated tests that are extremely hard to find or impractical to run." — L'Ecuyer (2)

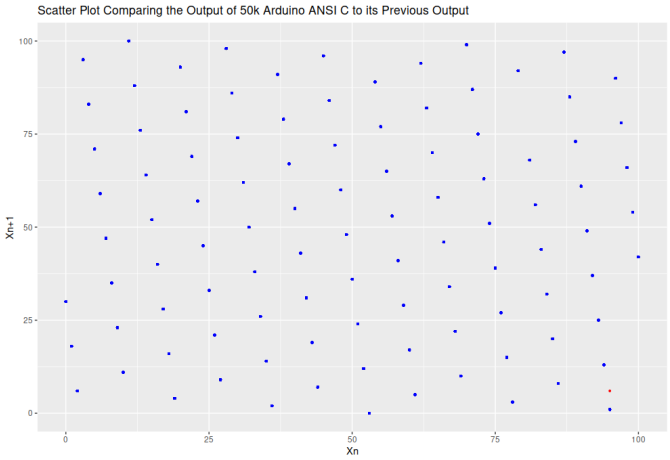

Fig. 8. 2D LCG state scatter plot for the Arduino Uno

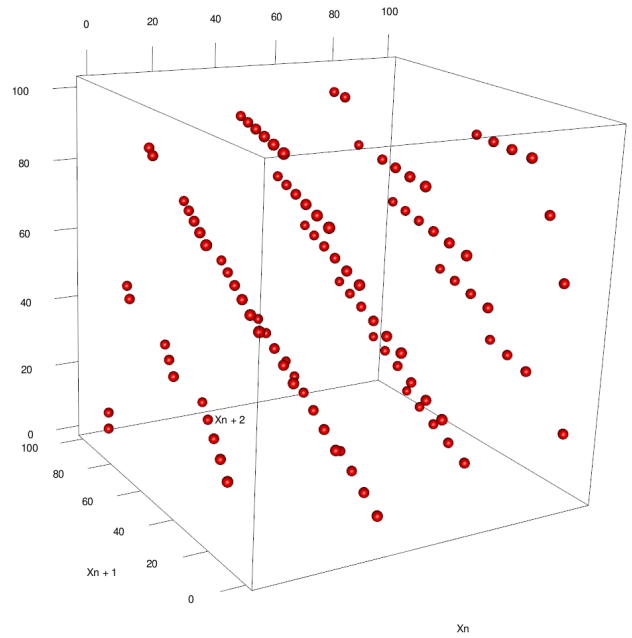

Fig. 9. 3D LCG state scatter plot for the Arduino Uno

Following this doctrine, one of the ways a PRNG will be judged in this paper is the shape of both 2 dimensional and 3 dimensional plots of the state of the PRNG. This method of representation accurately follows the scope of this paper, which is entropy idiosyncrasy – The predictability of the PRNG is a common vector for exploitation. Recall that for other applications, speed and efficiency are alternative interests to the researcher (such as a simulation environment).

Figure 17 ilistrates low-sample size outputs of the American National Standards Institute (ANSI) C programming language rand() function (In the exact same graphing technique used to

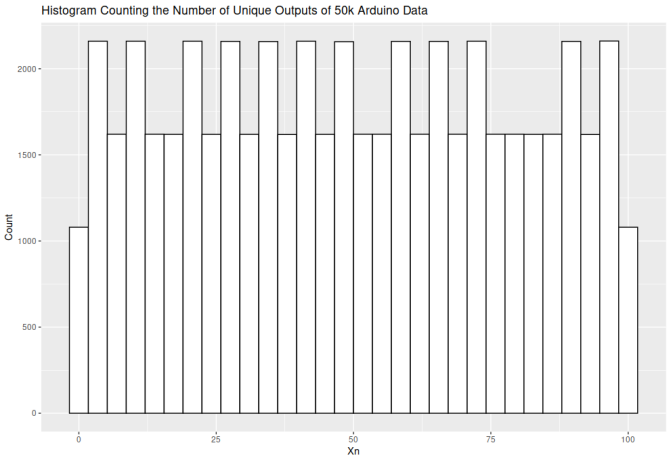

Fig. 10. Histogram for the Arduino Uno data

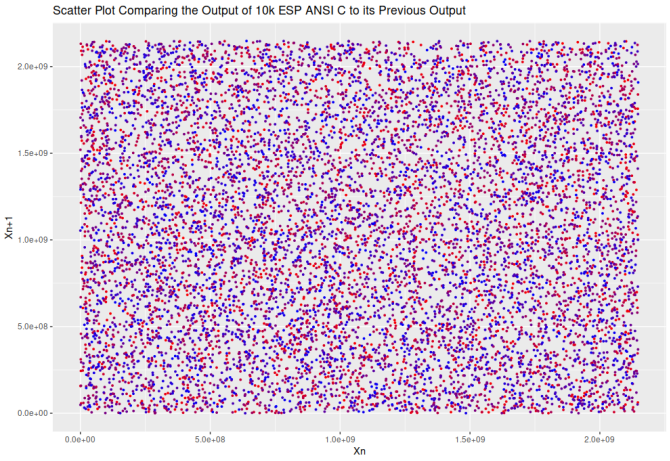

Fig. 11. 2D LCG state scatter plot for the ESP8266

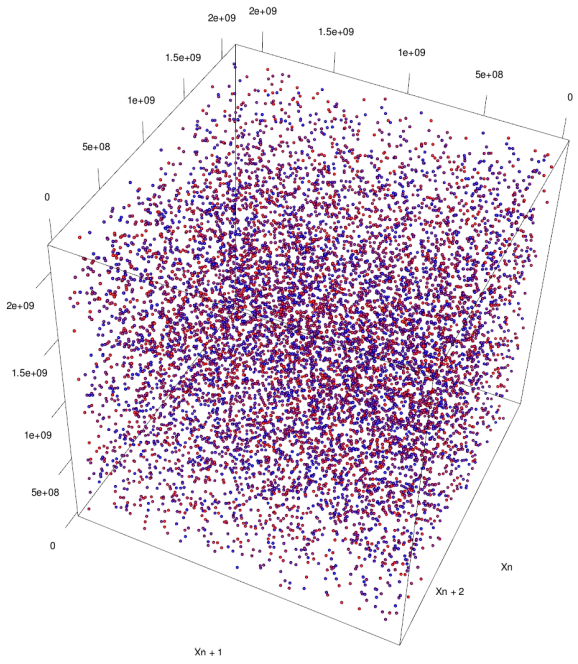

Fig. 12. 3D LCG state scatter plot for the ESP8266

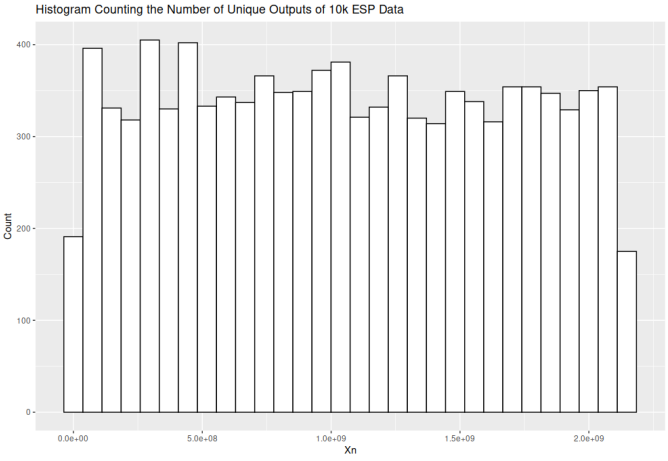

Fig. 13. Histogram for the ESP8266 data

display the output of the study) . While not the default PRNG for C++, this LCG is readily accessible in the C++ standard library (through `minstd_rand()`). A key characteric of these figures is: Entropy. It is impossible to tell (numerically or graphically) what the next state of the generator will be given two values. Our suspicions can further be confirmed looking at a 3-Dimensional graph of the output, shown in figure 18. Resembling a swam of locusts, the data appears randomly distributed.‘

To illustrate a counterexample, we now introduce an example of a bad LCG. Shown 2-dimensional in figure 19, the most telling difference is the appearance of visual lines. These artifacts represent a clear vulnerability in this LCG – The predictability of the next state or value to be outputted. With 5 or 6 values, an attacker can make an accurate guess as to the next state of the LCG, compositing its cryptographic integrity. For the purposes of visualization, plotting this data in 3-Dimensions (Figure 20) produces a predictable output: a series of vertical planes.‘

Returning to the data revealed through out study, we must first begin by providing a caviat to the data obtained for the ”Arduino” microcontroller. The Arduino Uni is a board that is based around the Atmega328p, which is a Low-power microprocessor with an 8-bit word size. This means that any piece of information transferred within the microcontroller must be 8-bits or less in size – Equal to $2^8 = 256$. While the Arduino can store numbers greater than this size in flash (I.E in the compiled code), during operation any information that enters RAM cannot exceed the 8-bit limitation. This presents a problem with the large modulus required by an ANSI C LCG. In effect, the data retrieved from the Arduino must be understood to **not** be from an ANSI C LCG. The defines were taken from an introductory novel from an example LCG.

With the graphical ”tells” layed out already, the conclusions

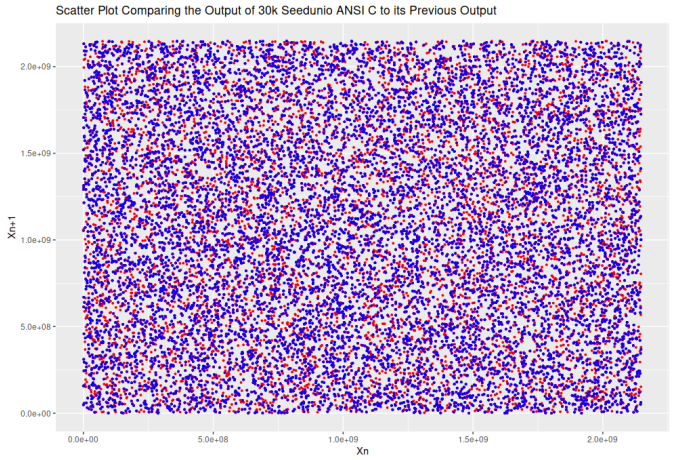

Fig. 14. 2D LCG state scatter plot for the Seeduino Xiao

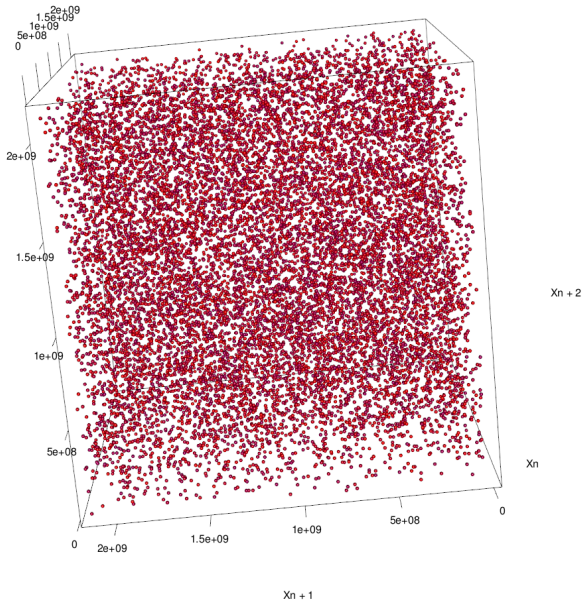

Fig. 15. 3D LCG state scatter plot for the Seeduino Xiao

we can draw from our graphics are quite obvious at this point. For both the ESP8266 and the Seeduino the data occupies the same scope, from 0 to $(2 * 10^9$. A second noticable visual differnce is the density of the dots. The size of the dots in both samples is exactly the same, however the data from the Seeduino appears much more solid, which is a testament to the differnece in amount of data collected. The histograms for both microcontrollers was remarkably similar as well, both graphs had a relatively low first column, followed by a spike then two decreasing high columns. This suggests that for both microcontrollers the LCG was running not only as interdependent, but they were in the same state at the same time.

The Arduino data was much more interesting however, as it presented idiosyncrasies that not present visually in the other

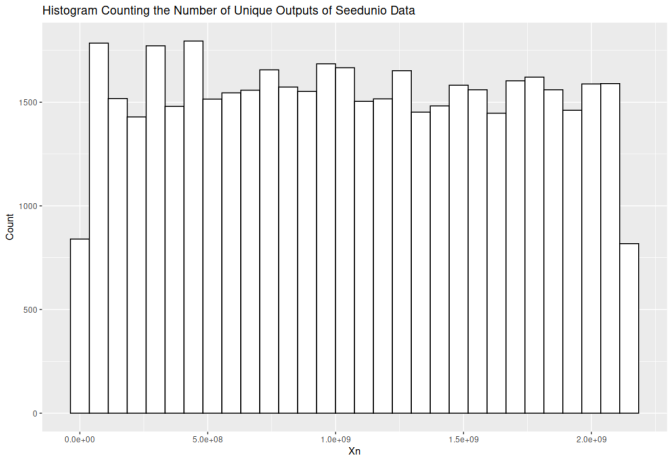

Fig. 16. Histogram for the Seeduino Xiao data

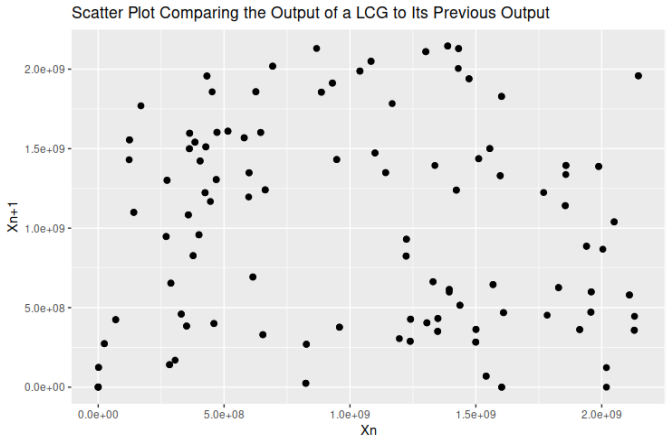

Fig. 17. Low-Sample ANSI C rand() function 2D output

two data sets. Firstly, the histogram alternated between only 3 different counts of numbers. Notice in figures 13 and 16, there was a large variation in the counts for each number. This variance was not present in the Arduino histogram. This effect shows that the same numbers are being cycled through at the same rate, which is an undesirable quality for a PRNG. The second undesirable element is the horizontal lines formed in the 3D graph. In this type of graph, a completely random assortment of dots is desirable, as any shown pattern indicates that the next state can be predictable from the previous state.

IX. Conclusion

A key weak point of this study is the reliance on graphical conclusions as opposed to more traditional statistical tests. Several testing batteries were explored (`TestU01`,`diehard`, and `dieharder`) however none were deemed appropriate for this study. The vast majority of testing suites are designed based off of two principles which invalidate their use in this study:

- Cryptographic Security
- Cutting Edge Pseudo Random Number Generators

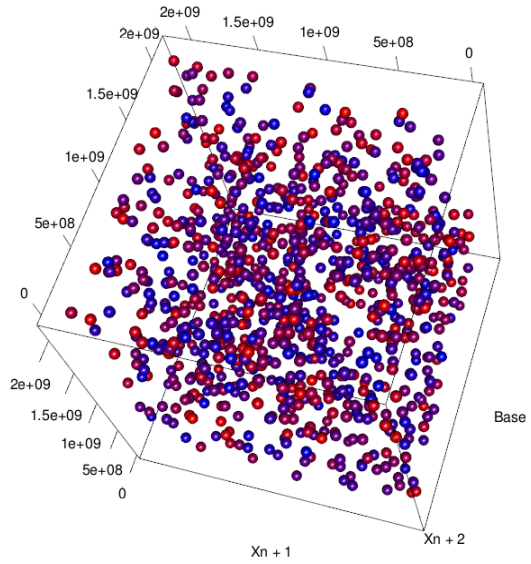

Fig. 18. Low-Sample ANSI C rand() function 3D output

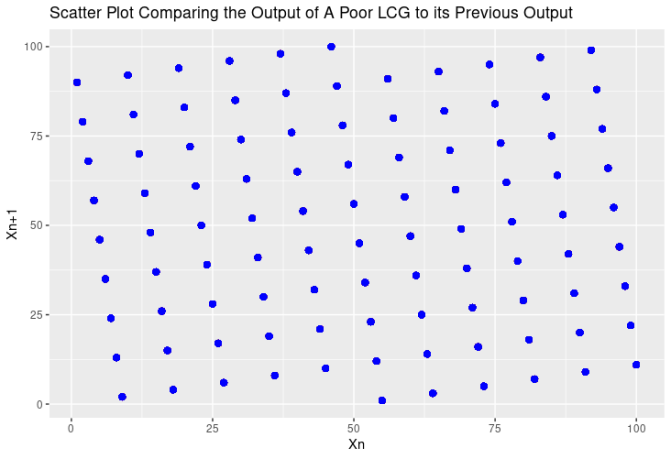

Fig. 19. Low-Sample ”bad” LCG 2D output

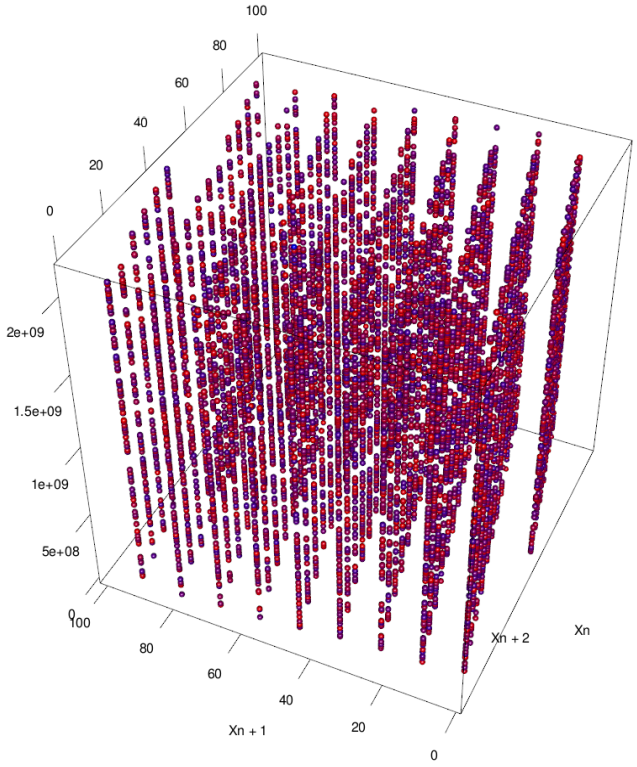

Fig. 20. Low-Sample ”bad” LCG 3D output

It is not a topic of question weather or not the ANSI C LCG is crypographically secure. The topic of crypography is brought up as an attempt to illustrate the importance behind random number generator entropy. This unique combinations of facts makes industry-standard testing suites useless for this application. Every piece of data was tested through popular (and some simple simple) `dieharder` tests including `birthdays`, `operm5`, and `diehard_rank_6x8` and failed all of them. This is a predictable outcome with **any** LCG, which would fail almost all tests in `dieharder`.

This is a fundamental flaw in the study design. For future iterations of this study, it would be prudent to make the following improvements:

- Implementing a more modern PRNG, such as a Mersenne Twister.
- Implementing a form of hardware entropy (such as a floating analog pin) and comparing the difference in statistical test performance
- Selecting a wider group of microcontrollers
- Exploring the impact of wireless connection

In conclusion, we were able to compare predictive graphs in both 2D and 3D of a ANSI C Linear Congruential Generator and find that when implemented at its most basic form, there is not a graphical difference in the predictable state of the LCG.