Generating True Random Numbers from Battleship

Group 3:
Cole Cathcart, 30043556 (CPSC)
Paruhang Basnet, 30021694 (CPSC)
Nathan Lum, 30064695 (CPSC)
Ryan Smit, 00224908 (CPSC)

April 14, 2023

Introduction

Randomness is a part of a plethora of different disciplines such as Physics, Biology, Mathematics, Statistics, and Computer Science. In Computer Science specifically, we use randomness for many purposes. Machine learning algorithms inherently use randomness in many different steps. In computer games, randomness is used to create fairness between all players, in dynamic procedural generation, to generate chance events, random effects like weather and is also popular in creating gambling-like features. Randomness is also a critical component of computer and information security. Cryptographic primitives often use randomness to generate numbers, which are critical to their ability to ensure security and authenticity.

Randomness generation comes in two forms: pseudorandom number generators (PRNGs) perform a function on a small seed to create a longer random number. On the other hand, true random number generators (TRNGs) take randomness from a direct source of entropy. The two are often used together, and true random number generation is an essential tool in the fields of information security and cryptography. It is important to remember that "true" randomness in the purest sense is not a proven phenomenon. Rather, when we refer to true randomness in this context we merely refer to an entropy source whose behavioral "formula" is currently unknown. One such theoretically suitable source of true randomness is human gameplay. There is a small body of academic work exploring the feasibility of human gameplay from computer games as an effective entropy source, and the data shows that this concept has merit at least in theory[1]. We will be exploring and experimenting with this concept as the basis for our project.

Problem Motivation

As we know, pure, true randomness is impossible to achieve. In order to overcome this lack of true randomness, the most common way to is to make your randomness as unpredictable as possible from the point of view of an adversary. This means that the source of randomness needs to have a high min-entropy. We specifically use min-entropy as the overall measure of entropy when dealing with TRNGs, because overestimating entropy generation in this context is a critical mistake that defeats the entire purpose of the generator. When using human gameplay to generate random numbers, the players' choices among several possible decisions are acting as our entropy source.

Not every game is suitable as an entropy source, however. In order to ensure a game facilitates high min-entropy output, several traits should be carefully considered. Halprin and Naor[1] outline some of these traits and we took them into consideration when choosing an appropriate game for our experiment: This game should allow easy calculation of min-entropy, generate entropy reasonably efficiently, and be fun and quick to play so that the players do not get bored. This last point may seem less important, but consider that if a player is not enthusiastic about the game it is much more likely they will begin making predictable moves. Most importantly, a suitable game should have a best-strategy of being as random as possible. If the best strategy of a game is to take some patterned approach, then min-entropy will be much lower. Assuming that players are invested in the game and want to win, the best way to ensure high entropy is to make randomness truly the best strategy.

With these aspects in mind, we settled on the board game 'Battleship' for our experiment. Battleship is a simple 2-player game often played as a board game, although computerized versions are also widespread. In the game, players secretly place a selection of 5 differently-sized ships onto a 10x10 grid, and then take turns making 'shots' at spaces on the grid with the intent

of hitting the enemy ships. A player wins once they have sunk all 5 of their opponents ships. This game has several desirable traits that we believe make it a good choice for an entropy source:

- Since ‘moves’ consist of coordinate choices on a 10x10 grid, it is easy to calculate min-entropy
- The game is fun and it takes only a few minutes to play on a computer
- The best strategy is to be as random as possible. This is true both for placing ships and for choosing where to place a shot. We argue that a strategic player will always choose randomness as the best option based on these 2 principles:
    - If a player chooses a non-random (aka patterned) method to place their ships, the opponent only needs to figure out the pattern in order to quickly win
    - Choosing a patterned placement of ships will not increase a player’s odds against an opponent choosing random squares to place their shots

  Therefore a player’s best strategy is to make both random ship placements and random shots to have the best probability of success in a given game. Note that we do not declare these principles as a formal proof of the argument, but for the confines of our experiment we will be working with the assumption that this argument is true.

Academic Sources

For our academic sources, we used both articles provided in the project topic selection handout. This included the paper by Halprin and Naor [1] and the paper by Alimomeni, et al [2]. We used these papers both as a guideline on how to conduct our own experiment and as a comparison for our final data. We also referred to 2 documents from NIST: The NIST SP 800-22[3] and NIST SP 800-90B[4]. These documents provide authoritative guidelines for both the creation and testing of random numbers and TRNGs.

Hypothesis

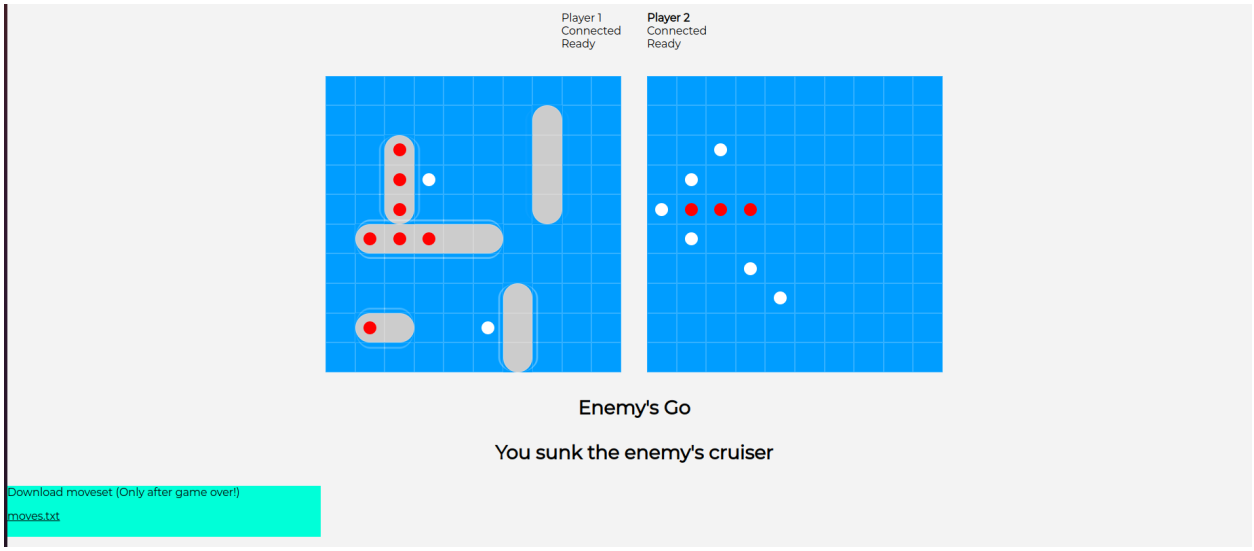
The academic sources we consulted have previously given positive results as to the theoretical viability of gameplay as a TRNG. Our experiment seeks to determine if Battleship is a suitable game for this purpose, not only in terms of randomness but also efficiency in entropy generation compared to real-world TRNGs and other gameplay-based generators. Our hypothesis is that Battleship is indeed suitable for producing random numbers, but will likely prove too inefficient for practical use.

Experiment Framework

In order to test this hypothesis, we created a TRNG using an online version of battleship as an entropy source, and then created programs to test the output of the generator with methods from the NIST statistical test suite[3]. Below, each step of this process is explained.

Data Collection

We were able to find the codebase for a simple online battleship game created using vanilla web developer tools and NodeJS, published under a public MIT license[5]. We further modified this code to suit the needs of our experiment: The most basic unit of entropy generation from this game will be each individual 'shot' taken by each player, so we modified the codebase to record every shot as a 2-digit decimal number representing the coordinate of that shot on the grid (for example, the top-left square would be 00, the bottom right would be 99).



There is an obvious problem with this simple approach: when a player lands a successful hit on an opponent's ship, that shot displays as red, informing the player that a ship is within that area. A strategic player will obviously take this information into account; instead of continuing to make random moves around the board, they will focus on a small area until they have completely sunk that ship. We determined the easiest way to circumvent this occasional decline in entropy was to ignore those moves entirely. As such, we further modified the program to only record the moves of a player when there are no partially-sunk enemy ships on his map, i.e. only moves made when a player is 'searching' for the next ship. Once a game finished, either player could then download the entire list of moves for that game in a .txt file. For example, this is the contents of a file downloaded after a game in which there were 41 'searching' moves. Moves from both players are included in the same sequence:

22625324379668280230805595198314754333304261827555572977792260029239586964878573 16

We generated a dataset by playing this online Battleship game with each other. Note that this is a small and biased sample size of players, which may introduce bias into our results. While playing we did not impose any rules to play as randomly as possible (though we believe this to be the best strategy in any case), but we also did not try to break the system by purposely playing non-randomly either.

Min-entropy Estimation

In order to estimate how much entropy our generator was making, we needed to calculate the min-entropy of our game, more precisely the min-entropy per 'shot.' In theory, since there are 100 squares on the board, the min entropy should be equal to $-log_2(\frac{1}{100})$ or 6.6 bits per move. This assumes, however, that each square has an equal likelihood of being clicked, since min entropy is calculated by taking the entropy of the *most likely* or highest probability outcome. In reality some squares may show a bias, which could reduce the actual min-entropy by a significant factor.

In order to get a more accurate min-entropy estimation, we used a NIST-recommended method called the *most common value estimate*[4]. This method simply collects the observed probability distributions of all possible moves and takes the entropy of the most commonly observed move to be overall min-entropy. We collected all recorded moves from all games in our dataset and mapped the frequencies of each of the 100 grid squares, taking the entropy of the most commonly clicked square. The result was indeed a decrease in min entropy from our theoretical entropy of 6.6, to 5.7 bits per click (see results section).

An important caveat to this method of estimation is that NIST specifies it should only be used for random variables that are *independent and identically distributed*, or IID[4]. Since Battleship only allows 1 move per square, the probability distribution of each successive move from the first decreases, akin to drawing marbles from a bag without replacement. The first move has 1/100 choices, but the second has 1/99, etc. We argue, however, that Battleship still produces IID results, because each game starts with an equal 1/100 chance of choosing any tile. Therefore, there is an equal probability of every subsequent subset of moves at 1/99, and so on. This augment seems to be confirmed by the fairly even distribution of observed results from our dataset (see results section).

Data Processing

After generating a raw string of output coordinates and calculating min-entropy, we needed some kind of program to transform this output into bit-string random numbers that are useful for testing or cryptographic purposes. We began by concatenating the movesets from our entire dataset of games into a single string (this step is optional; the generator will also work with a single game moveset). We then wrote an implementation in Python to process this data. Theoretically, each click has 6.6 bits of entropy. So, to get a final output bit-string with 128-bits of entropy (we chose this number as it is a common requirement for cryptographic purposes), we would need to combine at least 20 clicks to process into a single number . In practice, with our dataset we found the actual min-entropy to be 5.7 bits per click (see results section). With this more conservative min-entropy, we instead used 23 clicks for a single number. As such, the data

is processed 23 clicks at a time (46-digit numbers). These 23-click sequences are then directly converted into binary strings.

Another problem arises here which is not unique to our TRNG, that being that the bit-strings produced directly from the entropy source are often not uniform enough for cryptographic use, showing a bias for 1s or 0s. In order to remedy this, NIST recommends the optional use of an *extractor* function to be performed on the raw bit strings. An extractor is essentially a function that performs "whitening" on a bit string, in order to make the result more uniform[4]. NIST recommends many different types of extractors, but in our case we will take advantage of the *leftover hash lemma* to simply use a hashing algorithm as our extractor[4]. We chose MD5 for this purpose as it produces a 128-bit number as output. It is important to note that hashing our bit-string does not add any entropy into the output; that means if we want a final output with 128-bits of entropy, the input to the hashing algorithm must contain 128-bits of entropy. Our raw bit-strings should already contain this much entropy as per the method discussed above, but their length may be much greater than 128 bits long. Regardless of the raw bit-string length, the extracted number will always be exactly 128-bits long.

After the extracting step, we should now have a 128-bit number suitable for cryptographic use. We collected both the raw and extracted bit strings in order to perform NIST randomness testing on each set and compare.

Randomness Testing

We wanted to test the outputs of our generator to ensure their randomness and useability for cryptographic purposes. We used 8 of the tests outlined in the NIST Special Publication 800-22[3] as our test battery and implemented them using Python. It is important to use a variety of statistical tests instead of just one, because each test looks for a different indicator of randomness and some tests may be easily fooled on some aspects that other tests may easily catch. Both the extracted and raw bit strings were fed into these functions, using a p-value decision rule of 0.05 for all tests. The results of the testing on both sets of numbers were then collected for analysis (see results section). The 8 tests and their descriptions are as follows:

- Frequency (Monobit) Test: Compares the distribution of 1s and 0s in the entire sequence, assessing how close the fraction of 1s is to 1/2
- Runs Test: Assesses the number, size and frequency of 'runs' in the sequence, which are defined as uninterrupted subsequences of identical bits
- Linear Complexity Test: Determines if the sequence is complex enough to be considered random by assessing the length of a Linear Feedback Shift Register (LFSR) created from the sequence
- Serial Test: Determines if the number of occurrences of all possible m-bit patterns in the sequence is similar to what would be expected from a random sequence

- Approximate Entropy Test: Similar to the serial test, but deals with frequencies of consecutive m and m+1 length patterns
- Cumulative Sums Test: Assesses the maximal excursion from zero of the random walk defined by the 'adjusted' sum of all digits in the sequence (0 becomes -1 when calculating the sum)
- Random Excursions Test: Determines if the number of cycles having *K* visits in a cumulative-sum random walk deviates from the expected results from a random number
- Random Excursions Variant Test: Similar to the Random Excursions Test, but uses more random-walks
  [3]

Instructions for Running the Code

All the above steps combine to create our TRNG, and the code we have submitted can be used to recreate our experiment. The code for this project consists of 2 parts; the Battleship game implemented in JavaScript and the data processing/evaluation scripts implemented in Python. We have submitted a .zip file containing all the code needed to use the TRNG. The README file within that folder contains instructions for launching the Battleship game on a local network, however, we have also hosted the game on Heroku and you should be able to play the game by simply going to this link: https://battleshipg3.herokuapp.com/ (note that only the multiplayer mode works, so you will need 2 players). After playing a game, click the "download moves" link in the bottom corner to download a .txt file containing that game's moveset coordinate list. Any number of game movesets may be copied *manually* into a single file for use with the testing scripts, but a single game moveset should be enough to generate at least 1 random number.

Once you have generated a moveset .txt file, it must be renamed/pasted to "allmoves.txt" (this name is hard-coded) and placed in the '/games' directory in the '/tests' directory in the codebase. There are 2 Python scripts in this directory, 'generator.py' and 'GameRandTest.py'. Running 'generator.py' will print a 2d array of the Battleship board representing the frequencies of all clicks, and then generate as many 128-bit whitened numbers as possible from the input and print them as well. Running 'GameRandTest.py' will also generate as many whitened numbers as possible, and also run 8 NIST tests on all whitened and un-whitened binary numbers, printing the results (see the results section for more into). Please note that 'GameRandTest.py' requires both numpy and scipy to be installed.

Results

In total, we played 50 games of Battleship for our dataset, which contained altogether enough entropy to generare 88 random numbers assuming our calculated min-entropy of 5.7 bits per click. We felt that this was an adequately sized dataset with which to perform analysis and randomness testing.

Min-entropy

Our theoretical min-entropy was 6.6 bits of entropy per click. As discussed previously, we used the NIST-reccomended *most common value* estimate to calculate a more conservative min-entropy. Our 50 games contained 2025 recorded moves in total, from which we created a heat-map of the board:

16	13	21	23	24	11	21	20	9	10
8	33	21	18	21	21	23	21	21	14
19	27	23	13	24	15	21	26	19	16
18	21	18	24	23	27	19	26	13	20
18	27	21	28	23	20	19	17	18	18
19	19	22	30	24	29	12	22	14	17
21	23	21	12	26	22	19	19	20	18
19	22	32	19	26	17	22	24	18	26
10	32	21	20	21	27	16	28	17	11
14	17	37	17	23	17	24	21	14	13

From our click map, our highest clicked square had 37 clicks out of our 2025 total clicks. Our calculated min-entropy was therefore $-\log_2(37/2025) = 5.7$ bits of entropy per click. This coincides with our expectation of the bits of entropy being lower in practice compared to our theoretical entropy calculated previously. Apart from a slight bias against spaces near the corners, this heatmap shows a relatively even distribution across the board (the expected uniform distribution would be 25/2025 clicks per square), which seems to confirm our earlier arguments that our results are in fact IID and that random clicks seem to be the best strategy.

Randomness

We tested our 88 random numbers using the NIST statistical test suite[3] with a p-value of 0.05. These tests were run on all numbers (sequences of 23 moves) both before and after the use of the extractor in order to compare. Our assumption was that the un-extracted numbers would have trouble passing many of the tests and that the extracted numbers would perform much better. In reality this did prove to be true, although the discrepancy was not as high as we had expected. The raw numbers had an overall passing rate of ~90% compared with a ~99% rate from the extracted numbers, but only 40/88 raw numbers passed all 8 tests compared to 83/88 extracted numbers. Overall we are happy with the performance of our dataset against these tests and conclude that they are suitably random to consider our experiment a successful TRNG, at least theoretically.

```
--------------------------------------------------------------------------------
Percentage of NIST tests passed: 99.28977272727273%
Individual test pass counts:
Frequency Test: 88
Runs Test: 88
Serial Test: 88
Random Excursion Test: 88
Random Excursion Variant Test: 88
Cumulative Sums Test: 88
Approximate Entropy Test: 88
Linear Complexity Test: 83
Number of blocks that passed all tests: 83
--------------------------------------------------------------------------------
Percentage of NIST tests passed (Un-Hashed): 90.19886363636364%
Individual un-hashed test pass counts:
Frequency Test: 77
Runs Test: 73
Serial Test: 87
Random Excursion Test: 88
Random Excursion Variant Test: 88
Cumulative Sums Test: 88
Approximate Entropy Test: 88
Linear Complexity Test: 46
Number of blocks that passed all tests: 40
```

Efficiency

We kept track of the length of our games and found the average time to be ~2m36s, with an average of ~40 moves per game. At this rate, we are generating entropy at ~1.6 bits per second and about 1.8 numbers per game. This is by no means terrible, at least compared to similar experiments[1][2], but when we consider that real-world TRNGs are often generating several megabits of entropy per second it becomes clear that this method is not even close to being fast enough. This aligns with our hypothesis: Although our TRNG seems to succeed at producing random numbers, it is unlikely that this efficiency makes it practical for any real-world use.

Evaluation

Feasibility

Our TRNG was overall successful at creating suitably random numbers, and we conclude that Battleship can indeed be used as an entropy source for a TRNG. In line with our hypothesis, however, the relative inefficiency of the model is a major weakness. Not only is the speed of entropy generation relatively slow compared to many practical TRNGs in use today, but it also

requires 2 humans to play the game. Even if we extrapolate to a best-case theoretical system wherein thousands and thousands of players are playing online Battleship for fun at any given time, we are only getting roughly as many random numbers as there are players every few minutes. The manpower cost of this method alone is enough to conclude that Battleship is not efficient enough for practical use as a TRNG. Another point against this system is its potential weakness to attack from an adversary; a group of attackers, or even a single attacker with an army of bots, could easily swarm an online Battleship game and purposely generate very low entropy gameplay, say by choosing grid squares in ascending order, to effectively drop the entropy of all output numbers to zero. Even if this were only feasible for a short amount of time or could only drop the entropy somewhat, it would pose a major security threat. A possible counter to this vulnerability would be to further modify the program to try and detect very-low entropy gameplay. To answer our hypothesis, Battleship is suitable as an entropy source for a TRNG in terms of randomness of the output, but not in terms of efficiency for any realistic scenario.

Comparisons to Other Experiments

Our results were similar to those obtained by the two academic papers on which we based our experiment. Like them, our experiment concludes that human gameplay is at least capable of being used as an entropy source, though we believe the practicality of these systems does not extend outside an academic setting. At 5.77 bits per click, Battleship proved to be slightly better at entropy generation than the game discussed by Alimomeni, et al[2] but worse than Halprin & Naor's game[1]. The gap between our game's 5.77 and Halprin and Naor's 9.9 is not as large as we had thought, we had worried that restricting our game to only 100 move choices as opposed to using the pixel coordinates of the entire game screen would result in vastly less entropy per click, but it appears there are diminishing returns for entropy generation as move choices grow larger. Overall we believe the results and conclusions of our experiment align with the results of these papers.

Planning Execution

All of the work proposed in our Project Proposal was successfully completed. Although we did not create our own version of Battleship from scratch, we did have to significantly modify the Open Source version for our needs, including modifications for online playing, saving player moves (while ignoring some) and generating an output file of the moves. Within the limited time frame of the project and because of their complexity, there are seven statistical tests in the NIST framework that we did not implement.

Cole Cathcart was responsible for implementation of the Battleship game. Paruhang Basnet wrote the script to run the NIST tests on the numbers. Most of the presentation was

created by Ryan Smit. All group members participated in playing games to create the dataset, and writing the final report.

References

[1] Halprin, R., Naor, M. (2010). Games for Extracting Randomness http://www.wisdom.weizmann.ac.il/~naor/PAPERS/games_for_extracting_randomness_abs.html

[2] Alimomeni, M., Safavi-Naini, R., Sharifian, S. (2013). A True Random Generator Using Human Gameplay https://link.springer.com/chapter/10.1007/978-3-319-02786-9_2

[3] Rukhin, et al. (2010). A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-22r1a.pdf

[4] Turan, et al. (2018). Recommendation for the Entropy Sources used for Random Bit Generation https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90B.pdf

[5] Original Battleship Game GitHub Repo (several authors) https://github.com/kubowania/battleships