Human Assisted Randomness Generation Using Video Games

Mohsen Alimomeni, Reihaneh Safavi-Naini

University of Calgary, Department of Computer Science

malimome,rei@ucalgary.ca

January 19, 2014

Abstract

Random number generators have direct applications in information security, online gaming, gambling, and computer science in general. True random number generators need an entropy source which is a physical source with inherent uncertainty, to ensure unpredictability of the output. In this paper we propose a new indirect approach to collecting entropy using human errors in the game play of a user against a computer. We argue that these errors are due to a large set of factors and provide a good source of randomness. To show the viability of this proposal, we design and implement a game, conduct a user study in which we collect user input in the game, and extract randomness from it. We measure the rate and the quality of the resulting randomness that clearly show effectiveness of the approach. Our work opens a new direction for construction of entropy sources that can be incorporated into a large class of video games.

1 Introduction

Randomness has a central role in computer science and in particular information security. Security of cryptographic algorithms and protocols relies on keys that must be random. Random coins used in randomized encryption and authentication algorithms and values such as nonces in protocols, must be unpredictable. In all these cases, unpredictability of random values is crucial for security proofs. There are also applications such as online games, gambling applications and lotteries in which unpredictability is a critical requirement.

Poor choices of randomness in the past have resulted in complete breakdown of security and expected functionality of the system. Early reported examples of bad choices of randomness resulting in security failure include attack on Netscape implementation of the SSL protocol [GW96] and weakness of entropy collection in Linux Pseudo-Random Generator [GPR06]. A more recent high profile reported case was the discovery of collisions among secret (and public) keys generated by individuals around the world [LHA$^+$12, HDWH12]. Further studies attributed the phenomenon partly due to the flaw in Linux kernel randomness generation subsystem.

In computer systems, true randomness is commonly generated by sampling a complex external source such as disk accesses or time intervals between system interrupts, or is from users' inputs. Once some true randomness is generated, one can use *pseudorandom generators* to generate longer sequences whose unpredictability is computational. In all cases, unpredictability of the output of the pseudorandom generator crucially depends on the randomness of the initial true randomness. Importance of true randomness in computer systems has been well recognized and operating systems such as Linux and Windows have dedicated subsystems for entropy collection and randomness generation. These subsystems use internal system interrupts as well as user generated events as entropy source. High demand on entropy pools, for example when a computer runs multiple processes and algorithms that require randomness at the same time, can result in either pseudorandom values instead of truly random values, or stopping the execution until sufficient randomness become available.

The rate of randomness generation can be increased by including new sources of randomness which in many cased requires new hardware. An attractive alternative that does not require additional hardware is to use human assistance in randomness generation. This can be by directly asking human to input random numbers or move the mouse randomly [ZLwW$^+$09]. The process is unintuitive and experiments in psychology have shown that the resulting randomness has bias [Wag72].

In this paper, we propose a novel indirect approach to entropy collection from human input in game play that uses games as a targeted activity that the human engages in, and as a by product generates random values. Video games are one of the most widely used computer applications and embedding entropy collection in a large class of such games provides a rich source of entropy for computer systems.

1.1 Our work

Our main observation is that human, even if highly skilled, would not be able to have perfect game play in video games because of a large set of factors related to human cognitive processes and motor skill and coordination, limitations of computer interface including display, keyboard and mouse, and unpredictability elements in the game. The combination of these factors in well designed games results in the player "missing" the target in the game where although the goal may appear simple, achieving it is not always possible. Games usually come with a scoring system that rewards smaller "misses" of the target and provides incentive for further play.

We propose to use the error resulting from the confluence of the complex factors outlined above, as an entropy source. The unpredictability in the output of this source is inherent in the game design: that is a game that always results in win or loose is not "interesting" and will not be designed. In contrast games in which the user can "loose" a good portion of rounds are considered interesting. In a human game play randomness can be collected from different variables in the game, including the timing of different actions, the size of the "miss" as well as variables recording aspects of the human input such as angle of a shot, and so in each round, even when the user wins, a good amount of entropy can be generated.

Randomness from human game play. Halprin et al. [HN09] proposed to use human input in a game played against a computer as an entropy source. Their work was inspired by Rapport et al.'s [RB92] experiments in psychology that showed a human playing a competitive zero-sum game with uniform optimal strategy, generates better randomness compared to the case that they are directly instructed to supply random inputs. Halprin et al. used an expanded version (larger input set) of the game and replaced one of the users by the computer. The underlying assumption in this approach is that the human sequence of actions, when engaged in a game with uniform optimal strategy, simulates the game optimal strategy and so can be used as a uniform source of randomness. For the choices of human to be close to random, Psychological

results require that human is presented with few choices, otherwise the human will tend to select certain choices with more probability. So Alimomeni et al. [ASNS14] proposed a game design along the above game-theoretic approach in which human had only 3 choices and the randomness extraction was done as part of the game design with no need for seed. They showed that this design still keep the rate of min-entropy high because of the added extractor in the game that needs no randomness.
The above approaches are fundamentally different from the approach in this paper that uses the complexity of the process of generation of human input in the game, as the entropy source. Our approach is more in the spirit of sampling a complex process such as disk access, now using human and computer interaction as the complex process.

To use Halrin et al.'s approach in practice, one needs to design a two-party game with the supporting game-theoretic analysis to show the optimal strategy is uniform. The next step is to convert the game into an interesting game between the human and computer and validate that human would play as expected (is able to simulate the optimal strategy). In contrast our approach can be easily incorporated into video games such as target shooting games and does not need new game designs.

Implementation. As a proof of concept we designed and implemented a multilevel archery game, collected user inputs in the game and extracted randomness from the collected inputs. For randomness extraction we used the approach in [BST03] that uses universal hash functions. This allowed us to have provable randomness in the output of the extractor, as long as a good estimate of the input entropy is given. To estimate the min-entropy of the input to the extractor (min-entropy of the user input), we employed a set of min-entropy estimation tests proposed by NIST and used a beta implementation by NIST[1][BK12].

Our results clearly show that error variables, for example the distance between the target and the trajectory of the arrow, provides a very good source of entropy. The experiments show that the game can generate 15 to 21.5 bits of min-entropy per user shot using only the error variable. The variation in the amount of min-entropy is due to the variations in the game level and also varying levels of skill and learnability of users. Our experiments demonstrate that although entropy decreases as players become more experienced, but the entropy of the error variable will stay non-zero and even for the most experienced player at the lowest level of the game, 15 bits entropy per shot can be expected. The details of the game, experiments analysis of the users' input sequences and the extraction algorithm are given in Section 4.

1.2 Applications

Random number generation in game consoles and smart phones. Game consoles require true randomness for secure communication with the game servers, checking the digitally signed games and firmware updates from the server and to provide randomness to the games that is played. Lack of good random generation subsystems in these consoles may result in attacks such as reported in [Hot10]. Incorporating our approach into the video games played in such consoles would provide a method of generating randomness with high rate and verifiable properties. Our approach also provides an ideal entropy source for small devices such as smart phones that are used for gaming and have limited source of entropy.

User support for randomness generation in OS. An immediate application of our proposal is to provide on-demand entropy source for OS random number generation module. In softwares

[1]The software was provided by Tim Hall and John Kelsey from NIST.

such as PGP, Openssl, and GnuPG that need generation of cryptographic keys, using true randomness is critical. Such applications rely on the random number generation of the OS which may not havie true randomness available at the time of the request. Our proposed entropy source can be used for entropy collection from users by asking them to play a simple game. Our experiments showed that producing 100 bits of entropy required 6 runs of the game, making the approach an effective solution in these situations.

Cryptography with unknown computing devices. In many scenarios the user terminal is not well known to the user, or the terminal may not be capable of performing the required cryptographic operations to generate good randomness. For example a terminal used in an Internet Cafe although may wok correctly- but there is no guarantee that it generates good randomness that is needed for secure connection to a remote server. Using our proposed method of extracting random bits from game play, one can generate random bits without relying on the terminal randomness generation subsystem.

Contributory random number generation. In virtualized environments, multiple users share the same hardware (including CPU and RAM) and so the entropy pools of different users share a substantial amount of entropy produced by the system's shared hardware, resulting in the risk of dependence between entropy pools of different users. This is an important concern if the resulting randomness is used for key generation, potentially leading to attacks such as those reported in [HDWH12]. Using users' game play provides a source of randomness that is independent from the underlying hardware and other users' randomness.

1.3 Related works

The experimental psychology studies that suggested human's input in games as a good source of entropy [RB92, AF95], used a two-player game called *matching pennies.* This is a two-party zero-sum game where each user chooses one face of a penny; the penny is thrown; user one wins if the throws have matching faces, and the second if they do not. The game has uniform optimal mixed strategy. The game generates at most one bit per user input. Halprin et al. [HN09] based their work on these studies and used an extended two-dimensional version of the matching penny game as a game between a human and a computer. Using this expanded version the choice set of the user are points on a rectangle, resulting in increased number of random bits per user input. In playing the game, the computer needed the same amount of randomness, which in practice was supplied by a pseudo-random generator. For randomness extraction, a method similar to ours was used which needed an estimate of the min-entropy of the user generated sequence. Authors used a visual representation of the choices of the user on a two dimensional grid to argue the require randomness of the sequence and so applicability of the extractor approach. Our work however uses the best existing tests to quantify the min-entropy of the user sequence and support the intuition behind the approach.

Users' inputs through devices such as mouse and keyboard, has been widely used for background entropy collection in computer systems. An example is Linux based systems [GPR06] in which the operating system continuously runs a background process to collect entropy from users' inputs. Compared to our approach these entropy sources in general are expected to have lower entropy rate when used for on-demand collection of entropy. This is because of the repetitive patterns of mouse movements or key presses.

Organization In Section 2 we provide background and definitions and introduce our approach to randomness extraction. Section 3 describes our designed game and its generated sequence. Section 4 explains the experiments and gives analysis of the sequences collected from users. Finally in Section 5, we summarize our results and give the concluding remarks.

2 Preliminaries

We will use the following notations. Random variables are denoted by capital letters, such as X. A random variable X is defined over a set $\mathcal{X}$ with a probability distribution $\Pr_X$, meaning that X takes the value $x \in \mathcal{X}$ with probability $\Pr_X(x) = \Pr[X = x]$. Uniform distribution over a set $\mathcal{X}$ is denoted by $U_{\mathcal{X}}$ or U_n if $\mathcal{X} = \{0,1\}^n$. The logarithms will be in base 2 throughout the paper. *Shannon entropy of* X is denoted by $H(X)$ and given by, $H(X) = \sum_{x \in \mathcal{X}} \Pr_X(x) \log \Pr(x)$. Shannon entropy, is an average measure of uncertainty. For example, consider a source X that generates 0^n with probability 0.99 and every other n-bit sequence with the same uniform probability. For this source $H(X) \geq 0.01n$ which grows linearly with n. However the source cannot be used as an entropy source because of the predictability of output. Min-entropy of a source is a worst case measure and represents the best chance of the adversary in predicting the output of an entropy source. The *min-entropy* $H_\infty(X)$ of a random variable X is given by, $H_\infty(X) = -\log \max_x \Pr_X(x)$. Note that for the example source X above, $H_\infty(X) \leq 1$ which matches our intuition about the randomness of X. *Statistical distance* measures closeness of distributions and is used to measure closeness of the output of an entropy source to that of a perfect entropy source. The *statistical distance* $\Delta(X, Y)$ between two random variables X and Y over the same range $\mathcal{A}$, is given by $\Delta(X, Y) = \frac{1}{2} \sum_{a \in \mathcal{A}} |\Pr_X(a) - \Pr_Y(a)|$. If $\Delta(X, U_{\mathcal{X}}) \leq \epsilon$, then we say X is ϵ-biased or ϵ-close to uniform. We say X is *almost truly random* if X is ϵ-biased for a sufficiently small ϵ. A sequence of random variables $\{X_i\}_{i=1}^n$ is called an almost truly random sequence, if $\{X_i | X_{i-1} = x_{i-1}, \ldots, X_1 = x_1\}_{i=1}^n$ is ϵ-biased. An entropy source is a generator of sequences of symbols $\{x_i\}_{i=1}^n$ each sampled from a random variable X_i, where all X_i are defined over a finite set $\mathcal{X}$. It is important to note that output symbols of an entropy source may be correlated and not necessarily have the same distribution.

2.1 True Random Number Generators (TRNG)

A TRNG has two components: 1) An *entropy source* that generates possibly biased and dependent sequence of random numbers. This in practice is by reading the output of a physical source such as a lava lamp [C71], sampling a complex process such as disk access in a computer system, or sampling user's input. This sequence can be further sampled and quantized; and 2) A function that is applied to the output of the first step, resulting in an almost truly random sequence that removes the biases and dependencies among symbols of the input. The aim of a TRNG is to generate an *almost truly random sequence.* The closeness to a true random sequence is measured using *statistical distance.* The *rate of a TRNG* is the number of random output bits per time unit (e.g. seconds). A commonly used functions in the last step of a TRNG are *extractors*, briefly recalled below.

Randomness extractor. A randomness extractor [NZ96] is a function that transforms an entropy source with non-uniform distribution to an almost truly random source.

Definition 2.1 *For a family of distributions C, a deterministic* ϵ-extractor *is a function* $\mathsf{ext} : \{0,1\}^n \to \{0,1\}^m$ *that for every distribution $X \in C$, satisfies* $\Delta(\mathsf{ext}(X), U_m) \leq \epsilon$. *A family* *of distributions is called* extractable *if there exists an extractor that satisfies the above property for all distributions in the family.*

Deterministic extractors can be used as long as it can be assumed that the output of the entropy source is one of the distributions in the family C. For example, the family of Bernoulli distributions with unknown probability p, is extractable by the Von Neumann extractor [vN51]. Many families of distributions are not deterministically extractable [Sha11].

Definition 2.2 *For a family of distributions C, an ϵ-*seeded extractor *is a function* $\mathsf{ext} : \{0,1\}^n \times \{0,1\}^s \to \{0,1\}^m$ *that for every distribution $X \in C$ satisfies,* $\Delta(\mathsf{ext}(X,S), U_m) \leq \epsilon$, *where S is the uniform distribution over $\{0,1\}^s$.*

An important family of distributions that are extractable by seeded extractors are k-sources: *a distribution X is a k-*source *if $H_\infty(X) \geq k$.* Although many entropy sources are extractable using seeded extractors, but the the extractor seed needs a source of true randomness that may not be available in practice. In [BST03], Barak et al. proposed a framework for randomness extraction with guaranteed statistical property for the output, that can be seen as a compromise between seeded and deterministic extractors.

Barak et al. framework. The motivation of this work is to extract randomness for cryptographic applications where the adversary may influence the output of the entropy source. The adversary's influence is modelled by a class of 2^t possible distributions generated by the source. They proposed a randomness extractor with provable output properties (in terms of statistical distance) for sources that have sufficient min-entropy while the output source symbols may be correlated. The extraction uses *t-resilient extractor* which can extract from 2^t distributions (selected adversarially), all having min-entropy k. Certain hash functions are good t-resilient extractors.

Theorem 2.1 *[BST03] For every n, k, m and ϵ and $l \geq 2$, an l-wise independent hash function with a seed of length l is a t-resilient extractor, where $t = \frac{l}{2}(k - m - 2\log_2(\frac{1}{\epsilon}) - \log_2(l) + 2) - m - 2 - \log_2(\frac{1}{\epsilon})$.*

An l-wise independent family of hash functions can be constructed using a polynomial $h_s(x) = \sum_{1 \leq i \leq l} a_i x^{i-1}$ of degree l over the finite field $GF(2^n)$, where $s = (a_1, \ldots, a_l)$ is the seed of the extractor and $x \in GF(2^n)$ is the n-bit input.

The t-resilient extractors in Barak et. al's approach reuses a truly random seed that is hardwired into the system (e.g. at manufacturing time) and does not need new randomness for every extraction. Although extractors enjoy sound mathematical foundations, in practice the output of entropy sources are mostly processed using hash functions with computational assumptions and so extractors have not been widely implemented in hardware or software. In this paper we follow the framework of Barak et al.

3 Randomness from user errors

Consider a computer game in which the player aims to hit a target, and wins if the projectile "hits" the target. There are many factors that contribute to the user missing the target even if they play their "best", making the game result uncertain. *We propose to use the uncertainty in the game's result as the entropy source.* Assuming a human is engaged in the game and plays their best, the uncertainty in the result will be due to a large set of factors including, 1) limitations of human cognitive and motor skill to correctly estimate values, and calculate the best response values (e.g. time limitations imposed by the game to calculate the best angle and speed of throw) and perform the required action at the exact time, 2) limitation of input devices for inputing the best values when they are known, for example limitation of a mouse in pointing an arrow in a particular direction, and 3) unknown parameters of the game (e.g. game's hidden constants) and variabilities that can be introduced in different rounds. Other human related factors that would contribute to the unpredictability of the results are, limited attention span, cognitive biases, communication errors, limits of memory and the like. These uncertainties can be amplified by introducing extra uncertainty (pseudo-randomness) in the game: for example allowing the target to have slow movement. As a proof of concept for this proposal, we designed and implemented an archery game, studied user generated sequences and the randomness extracted from them. Below are more details of this work.

3.1 The Game

Our basic game is a single shooter archery game in which the player aims an arrow at a target and wins if the arrow hits the target: the closer to the centre of the target, the higher the score. A screenshot of the game is shown in Figure 1. The arrow path follows the laws of physics and is determined by the direction of the shot, initial velocity of the arrow, and the earth gravity pull force. This results in a parabolic path that the arrow will traverse to hit the target. The player chooses an initial speed and angle of throw to hit the target. The game is available to play at [Ali13].

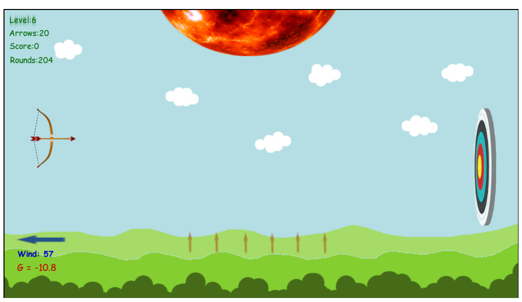


Figure 1: Screenshot of the game

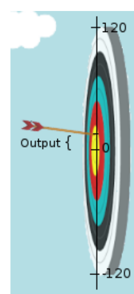


Figure 2: The measurement of output

The target is shown by a circular disk on the screen. The game records the distance between the center of the target and the trajectory (Figure 2). To display the trajectory on the screen, graphic engine translates the location of the arrow into pixel values and show their locations on the display. We however use the actual value of the distance between the center of the target and the trajectory calculated using laws of physics (kinematic equations), and then round it off to a 32 bit floating point number (the effective bits). The advantage of using this approach is not only avoiding entropy loss, but also independence of the implementation and measurements from the screen size and resolution of the end device. For the error variable we use the range $I = [-120, 120]$ with each sample read as a 32 bit floating point number, and represented as [Sign(1bit), Exponent(8bits), Fraction(23 bits)].

We will refer to each shot, as a *round* of the game. After playing the game for a number of rounds, the server will have a sequence of floating point numbers in the interval $I = [-120, 120]$.

The range [-120, 120] can be adjusted depending on factors such as screen size and target shape. One can use multiple seemingly unrelated variables in the game for the source of entropy.

Examples are, the angle and initial velocity of the shot, time that takes for a user to make a shot, and the time between two consecutive shots. We only analyze the entropy source corresponding to the variable that represents the human error in hitting the target. The game was implemented using HTML5 for ease of portability.

3.2 Game parameters and levels

Our initial experiments showed that the game parameters affect the final entropy. We designed game levels with varying sets of parameters. The parameters that we alter in the game are: 1) Location of the target, 2) Speed of the target movement, 3) Gravity force with two possible direction, and 4) Wind speed and direction. These parameters can change for every player shot, or stay constant during the game. There were other possibilities such as adding an obstacle (e.g. a blocker wall) to force the player choose angles from a wider spectrum, putting time limit on each shot to force the player to release the arrow faster, smaller target or farther target in the screen that could be considered in future. The final game has 8 levels, 3 of which were used for experiments labeled as A, B and C respectively. In level A, all parameters were "fixed" with no change during the rounds, and so no input is used from the computer. In level B, a subset of game parameters are changed in each round of the game and the values of the parameters were shown in the interface so the player can decide based on that information. In level C, the values of changing parameters of level B were not shown to the user (except the direction of gravity and wind). The high uncertainty in this level of the game makes the players rely on their intuition to play the game.

We did not perform a user study to show attractiveness of these levels but comments from users indicated level B was the most appealing level.

3.3 Entropy source

The distance between O, the target center, and the trajectory at O', is a 32 bit floating point number. One can use quantization to map these numbers into ℓ bins. A simple quantization is to consider circular areas, centered at O with linearly increasing distances: the first circle will have radius r, the second $2r$, etc. Now if O' for a miss trajectory is in the first circle, it is considered 0, the next one 1 and so on. A good quantization and extraction will ensure that every element of the alphabet is generated "roughly" with the same relative frequency. To have this property, we followed the randomness extraction framework of [BST03] with an extra statistical evaluation step at the end. Our randomness extraction and evaluation has three steps. i) Estimate min-entropy of the sequence; ii) Given the estimate, apply an appropriate extractor (in our case pairwise independent hash function) on the sequence; and iii) Use statistical tests to evaluate quality of the final output sequence.

We used the NIST tests [BK12] outlined in Appendix A, to estimate the min-entropy of our sequences. Our experimental results showed that our entropy sources were not independent and identically distributed (i.i.d). This was because for each data set, either the Directional run or Covariance tests (part of shuffling tests) failed. We estimated the min-entropy of our sequences assuming non-i.i.d sources. In the post processing step, sequences were converted to truly random sequences, using extractors. We used a t-resilient extractor defined over a finite field, and so floating point numbers needed to be translated into numbers in that field. One naive approach was to cast the floating point numbers into an integer value corresponding to the same bit representation of the floating point number. This method however will affect the structure of the sequence. For example, the sequence of differences between two floating point numbers (which represents the distance of the arrow from the target centre) will have a different structure from the sequence of differences between their corresponding integer values if simply casted. In order to maintain the structure of the entropy sequences in our experiments in Section 4, we added a processing step to convert the output sequence into a sequence of integers while keeping the structure of the source as explained in Appendix C. The final output string (after application of extractor) was evaluated using statistical tests. We used the TestU01 framework [LS07b] with an implementation available at [LS07a]. This framework has implemented an extensive sets of statistical tests including [BRS+10, Mar98].

3.4 From the entropy source to the output

We read 32 bit floating point numbers as the output of the entropy source and interpreted each sample as a 32 bit integer as described in Appendix C. To use a min-entropy test, we needed a sufficiently long sequence over an alphabet. We interpreted each 32 bits block as a collection of subblocks of different lengths. We were limited by available user generated data and so the size of the subblock depended on the experiment to ensure that a sufficiently long sequence was available. We used the min-entropy test outlined in Appendix A and considered each sample as 32 1bit subblocks, and obtained an estimation of min-entropy per bit. We considered all bits of the input, even those with low min-entropy in the estimation. Given the estimate of k bit min-entropy for a single bit, we obtained an approximate value for min-entropy of each sample as $32k$. Here we effectively assumed bits have similar min-entropy which is reasonable since our per-bit min-entropy estimation considered all bits. We performed the above calculations for data from each player including all levels, resulting in minimum estimated min-entropy of 0.626 per bit. For 32 bits, we estimated $32 \times 0.626 \approx 20$ as the minimum min-entropy of the source per 32 bits. Note that this minimum is over the data from all levels for each user, and the minimum we reported earlier (15 bits) is the measured min-entropy for the most skilled user in the simplest level.
We closely followed the framework explained in Section 2.1 by using a 32-wise independent hash function, with $\epsilon = 2^{-2}$, and $m = 11$. Using theorem 2.1, the extractor was chosen to be t-resilient with $t = 16$. Here 2^t is the number of possible distributions chosen by the adversary. Variations of the distribution due to the players experience could be modeled similarly. The random seed for the extractor was generated from /dev/random in Linux. To examine the properties of the final output sequence, we used the statistical tests Rabbit [LS07b]. Rabbit set of tests includes tests from NIST [ea10] and DIEHARD [Mar98] with modified parameters tuned for smaller sequences and hardware generators. We used an implementation of these tests in [LS07a]. All tests were successfully passed.

4 Experimental setup and results

In this section, we present our experimental results. We asked a set of 9 players to play each of the three levels at least 400 rounds. The rest of the levels were played for learning. Our objective was to answer to the following:
1- *The minimum entropy that can be expected in a single round:* As noted earlier factors such as user's skill level before the game starts, and learning through playing the game, and the match between the skill level and difficulty of the game will affect the final entropy of each round.

2- *The change in min-entropy of a player over time:* We examine how more experience and familiarity with the game would affect the amount of entropy generated in a round.
3- *The effect of the game level on min-entropy:* In this experiment, we determine the best game parameters that maximize the rate of the output entropy.

4.1 Entropy per round

We performed two sets of experiments to estimate the minimum entropy per round that can be expected from the game.

4.1.1 Entropy of generated sequences for one player

In this experiment we measured the min-entropy of the sequences generated by each player. We partitioned a player's sequence of shots into 20 parts and measured the min-entropy for each part per bit, i.e. considering each bit of a floating point number as one sample which gives us 32 samples per round. The graph in Figure 3, demonstrates the maximum, minimum and average min-entropy for each player, here a set of 9 players. We also repeated the experiment in Appendix B for all players and Figure 4 illustrates the result of min-entropy in each bit for one sample user, which is consistent with the experiment on data from all users.

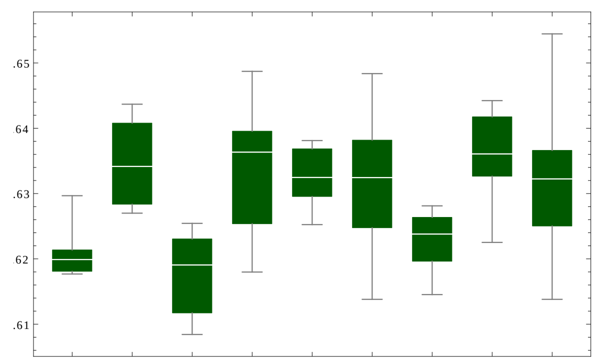


Figure 3: Min-entropy for players

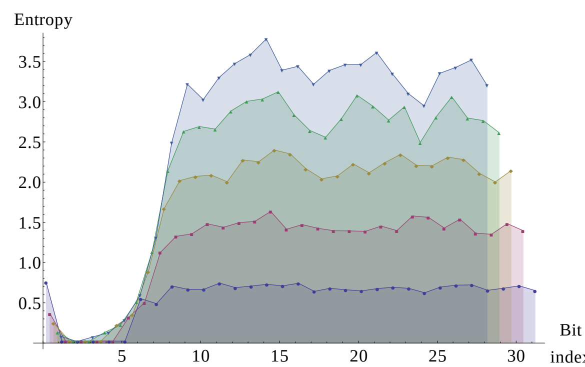


Figure 4: Min-entropy in blocks of bits (One user)

4.1.2 Sequence generated by the population

In this experiment, data from all users were considered as one. We then measured the min-entropy (per bit) for this data set. Our estimation of min-entropy for the population shows that the average min-entropy in the output is 0.643 per bit, so on average, with 5 shots (5×32 bits) one can generate 103 bits of min-entropy. The average time for each shot (over all players) was approximately 2 seconds. Note that the estimation was higher than the average min-entropy of all users (when min-entropy was measured separately) which is 0.61 because of higher estimation of min-entropy by NIST tests for larger datasets as noted at the end of Appendix A.

4.2 Effect of players' experience on min-entropy

An important factor in game design is the effect of players' experience on the generated entropy. Intuitively, we expect the entropy to decrease as players play more. In our game, one expects more experienced players to hit closer to the target center more often and so less observable error, while an inexperienced player's shot to appear more random. We estimated the min-entropy of each of the game levels for 3 different players. Our results confirms this expectation. However it shows that even the most experienced user at the lowest game level can generate good level of entropy.

Figure 5 illustrates the change in min-entropy in each of the three game levels A, B and C, as players gain more experience. Figure 5 also shows how the design of the game neutralizes the affect of player's experience to keep the average min-entropy high enough for randomness extraction.

The graphs 5 and 6 are divided into three parts, each consisting of 3 graphs corresponding to the 3 players. The three parts, left (from 0 to 20), middle (21 to 40) and right (41 to 60), correspond to the levels A, B and C, respectively. We used 3 players with the highest (the blue curves marked with letter H), average (the red curves with letter A) and lowest (the yellow curves with letter L) scores for this experiment. An interesting observation about level C is that the min-entropy does not necessarily decrease for a user which is expected from the fact that game parameters are randomly changed and not known by the players.

4.3 Min-entropy and game levels

We considered the change in min-entropy over time for a level. That is reduction in entropy as users become more skilled. We used the min-entropy estimation for all player's data, when partitioned into 20 sections as in previous experiment. The data corresponds to the sequence of shots over time and so the first section of the data comes first -that is users starting the game- then the second and the third sections as they get more experienced. We did this experiment for data for all users to find the average trend of min-entropy.

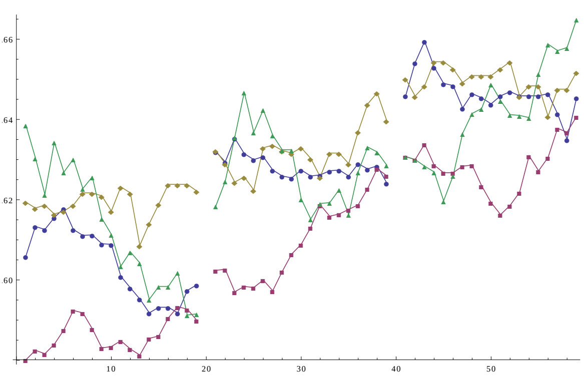


Figure 5: Min-entropy during level A, B, C for 3 users

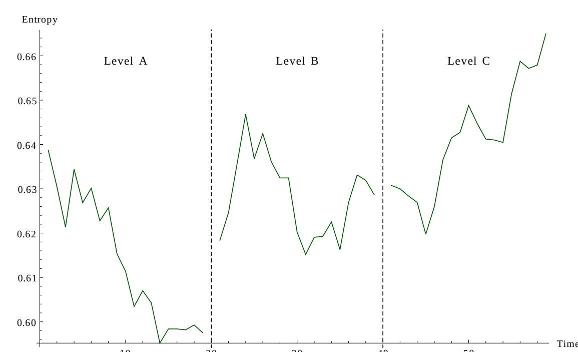


Figure 6: Average min-entropy change during levels over all users

The graph is divided into three parts corresponding to the three levels as in previous section. Figure 6 shows the results of all measurements in the left (level A), middle (level B), and right (level C) parts. Level A shows a reduction in the min-entropy as the players become more experienced, and it has the highest min-entropy decrease among the three levels. In level B, the min-entropy fluctuates around the value 0.625 and is relatively stable. For level C however, there is no clear trend in the data and this is true in general for all players, but the average min-entropy is higher than levels A and B. One reason for the increase of min-entropy in level C is probably the reluctance of players to play well over time due to the many unknown variables of the game that makes it hard to win. This confirms the effect of non-deterministic and unknown values of parameters which makes the skill level somewhat irrelevant.

4.4 Randomness required by computer

As noted earlier, the least significant bits of the output corresponds to cumulation of small errors in different part of the system and contribute most to the min-entropy. Thus even the sequence collected from level A *without any random input from the computer,* could be used for entropy generation. To confirm this observation we asked the most experienced player (with highest score) to play level A again, after they had played levels A, B and C more than 1200 rounds. We measured the min-entropy for this data. The player had 20 arrows to hit the target and with each shot to the center, a bonus arrow was given. The user played for 3 games, totaling 331 shots to the target. With 83% of the shots to the center, the estimated min-entropy of the player in this 331 shots was roughly 0.521 per bit.

This suggests that the sequence generated by the game has a minimum min-entropy independent of the randomness used by the game (computer). For higher levels of game that require randomness, one can use pseudorandom generators in real-time or generate the sequences beforehand and use them as needed.

5 Concluding remarks

We proposed and analyzed a new approach to entropy collection using human errors in video games. We verified the approach by developing a basic intuitive game and studied the sequences generated by users.

Our experiments showed that with this simple design and considering the "worst" case where the user was experienced and made the least error, the rate of entropy generation is at least 15 bits per shot. This rate can be increased by adding variability to the game and also using multiple measurable variables instead of only one. Adding variability to the game increased the min-entropy by 7 bits per round. In choosing parameters one needs to consider attractiveness of the game: increase in entropy can be immediately obtained if game constants such as gravitational force in our case are changed without user's knowledge. However this would decrease the entertainment factor of the game. Studying these factors and in general the randomness generated by users needs a larger user study which is part of our future work. For the randomness extraction we implemented and used t-resilient extractors. The output from extractors passed all statistical tests.

Our work opens a new direction for randomness generation in environments without computational capability or randomness generation subsystems, and provides an attractive solution in a number of applications.

References

[AF95] P. Ayton and R. Falk. Subjective randomness in hide-and-seek games. In *BOOK of Abstracts of the 15th Bi-annual Conference on Subjective Probability, Utility and Decision-Making*, page 37, 1995.

[Ali13] Mohsen Alimomeni. Archery game, 2013. `http://pages.cpsc.ucalgary.ca/~malimome/game/`.

[ASNS14] Mohsen Alimomeni, Reihaneh Safavi-Naini, and Setareh Sharifian. A true random generator using human gameplay. In *GameSec*, pages 10–28, 2014.

[BK12] Elaine Barker and John Kelsey. Recommendation for the entropy sources used for random bit generation, August 2012. http://csrc.nist.gov/publications/drafts/800-90/draft-sp800-90b.pdf.

[BRS+10] Lawrence E. Bassham, III, Andrew L. Rukhin, Juan Soto, James R. Nechvatal, Miles E. Smid, Elaine B. Barker, Stefan D. Leigh, Mark Levenson, Mark Vangel, David L. Banks, Nathanael Alan Heckert, James F. Dray, and San Vo. Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, Gaithersburg, MD, United States, 2010.

[BST03] Boaz Barak, Ronen Shaltiel, and Eran Tromer. True random number generators secure in a changing environment. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 2779 in LNCS, pages 166–180, 2003.

[C71] W.E. C. Display device, March 16 1971. US Patent 3,570,156.

[CT91] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications. John Wiley & Sons, Inc., 2nd edition, 1991.

[ea10] Rukhin et al. A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications, 2010. http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf.

[GPR06] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the linux random number generator. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[GW96] Ian Goldberg and David Wagner. Randomness and the netscape browser. *Dr Dobb's Journal-Software Tools for the Professional Programmer*, 21(1):66–71, 1996.

[HDWH12] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 35–35, Berkeley, CA, USA, 2012. USENIX Association.

[HN09] R. Halprin and M. Naor. Games for extracting randomness. In *Proceedings of the 5th Symposium on Usable Privacy and Security*, page 12. ACM, 2009.

[Hot10] George Hotz. Console hacking 2010-ps3 epic fail. In *27th Chaos Communications Congress*, 2010.

[LHA+12] ArjenK. Lenstra, JamesP. Hughes, Maxime Augier, JoppeW. Bos, Thorsten Kleinjung, and Christophe Wachter. Public keys. In *Advances in Cryptology CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 626–642. Springer Berlin Heidelberg, 2012.

[LS07a] Pierre L'Ecuyer and Richard Simard. Testu01, August 2007. http://www.iro.umontreal.ca/~simardr/testu01/tu01.html.

[LS07b] Pierre L'Ecuyer and Richard Simard. Testu01: A c library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33(4), August 2007.

[Mar98] George Marsaglia. Diehard, 1998. http://www.stat.fsu.edu/pub/diehard/.

[NZ96] Noam Nisan and David Zuckerman. Randomness is linear in space. *Journal of Computer and System Sciences*, 52(1):43–52, February 1996.

[RB92] A. Rapoport and D.V. Budescu. Generation of random series in two-person strictly competitive games. *Journal of Experimental Psychology: General*, 121(3):352, 1992.

[Sha11] R. Shaltiel. An introduction to randomness extractors. *Automata, Languages and Programming*, pages 21–41, 2011.

[vN51] John von Neumann. Various techniques used in connection with random digits. *J. Research Nat. Bur. Stand., Appl. Math. Series*, 12:36–38, 1951.

[Wag72] W.A. Wagenaar. Generation of random sequences by human subjects: A critical survey of literature. *Psychological Bulletin*, 77(1):65, 1972.

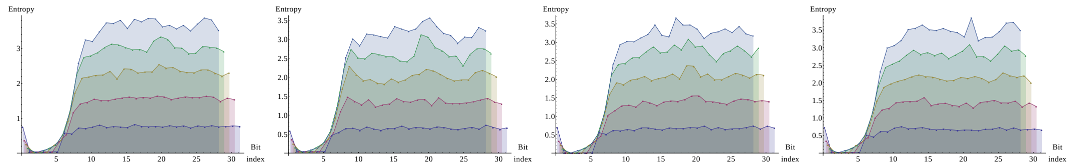
[ZLwW+09] Qing Zhou, Xiaofeng Liao, Kwok wo Wong, Yue Hu, and Di Xiao. True random number generator based on mouse movement and chaotic hash function. *Information Sciences*, 179(19):3442 – 3450, 2009.

A Entropy estimation of the source using NIST tests

To measure the min-entropy (or Shannon entropy) of a source one needs to assume certain structure in the source distribution. For a list of n samples $\{s_i\}_{i=1}^n$ from a source S over the finite set $\mathbb{S}$, if we assume that the source S is i.i.d., that is samples are independently and identically distributed, then having enough samples from the source allows us to estimate the probability distribution of the source with high confidence and find the entropy as in [BK12] (Section 9.2).

NIST draft [BK12] gives the requirements of entropy sources as well as proposing a number of tests to estimate the min-entropy of the source in i.i.d. and non-i.i.d. case, both. The testing method first checks whether the source can be considered i.i.d. NIST suggests the following set of tests for i.i.d. sources (Section 9.1 of [BK12]): 6 shuffling tests, Chi-square test, test for independence and goodness of fit. If all of the test are passed, the source is assumed to be i.i.d. and then *a conservative method* is used to estimate the entropy of i.i.d. source. If any of the tests are not passed however, the source is considered to be non-i.i.d., and a number of tests are used to estimate the min-entropy. These second group of tests are collision test, partial collection test, Markov test, compression test and the frequency test, each outputting a value as the estimation of the min-entropy. The final min-entropy will be the minimum over all these estimated values. While i.i.d. and non-i.i.d. tests provide an estimation of the entropy, they may fail to detect anomalies in the source. Therefore, NIST defines a set of sanity checks that will make sure this does not happen. The sanity checks contain two tests: Compression test and collision test. If the sanity checks fail, no estimation will be given.

For our experiments, we obtained an unofficial version of the code (the code is not released yet) and used it to estimate the min-entropy of our source. We ran tests to verify whether our estimations are meaningful (our sanity checks), and also check consistency in the min-entropy estimation for a data set from /dev/random in Linux. Our analysis showed that the estimation from NIST set of tests are sound, but are very conservative (admitted in Section 9.3.1 of [BK12].



(a) Min-entropy in blocks of bits (all players) **(b)** Min-entropy in blocks of bits (level A) **(c)** Min-entropy in blocks of bits (level B) **(d)** Min-entropy in blocks of bits (level C)

Figure 7: Min-entropy of bits

For example, we expect to have roughly the same approximation of min-entropy for the data in /dev/random. But the approximation from the NIST tests depended very much on the number of samples given to the tests (which is quite intuitive and acceptable). This caused very low estimates for a subset of our users with smaller sample size and in general, *min-entropy estimation in our experiments were conservative.*

B Most important bits in the output

Different bits of the 32 bit representation of the error variable may have different amount of entropy. In this experiment, we run the min-entropy estimation test individually on each bit of the output (one bit per 32 bit sample). We also run the same min-entropy estimation test on t consecutive bits of this 32-bit sample. We used a sliding window of t consecutive bits, shifting one bit at a time, for $t = 1, 2, 3, 4, 5$. Figure 7 shows the result of this experiment.

Each point on the X-axis of each curve corresponds to the starting of a t-block. For $t = 2$ (second curve from below) for example, the first point corresponds to the block consisting of the first and the second bits, the second point corresponds to the block corresponding to the second and the third bits and so on. The Y-axis shows the min-entropy of the block. The graphs in Figure 7a are for all the data from all users in all levels. The Figures 7b, 7c and 7d show the result for for level A, level B and level C, individually (respectively).

The graph shows, the min-entropy of the most significant bit (MSB) is high and then the following 5 bits have min-entropy close to zero. The MSB corresponds to the sign bit of the floating point number as described in IEEE 754 for single precision floating point format. This sign bit in the number shows if the arrow hits the target below or above the center. The next 5 bits are the first bits of the 8 bit exponent in this representation. Since the output of the game is in the interval $[-120, 120]$, the exponent part of the output numbers is less than 8 bits and so these values have almost zero entropy. The rest of the bits in the output have high min-entropy. This is specifically true for bits in locations 20 to 32. The graphs show that the higher entropies are contributed by the less significant bits of the output, which correspond to small errors of the players. These small errors are independent of the user and level of the game. This suggests that *the min-entropy contributed from these bits are present for all users and levels of the game.* Thus, even Level A of the game expects to generate good amount of min-entropy from these bits. This conclusion is also confirmed by other experiments in section 4.

C Converting floating point numbers to integers

In this step, we apply a function $fi : I \to \mathbb{Z}$ to the sequence of floating point numbers generated by the entropy source.

1. Divide I into 2^{32} partitions.
2. Index each partition from -2^{16} to 2^{16}.
3. For each number generated, return the index of the partition containing the number.

This additional step applied on the source, does not decrease the entropy.

Theorem C.1 *The conversion function fi does not decrease the entropy in terms of Shannon and min-entropy.*

Proof. Consider X as the distribution of the entropy source when generating one symbol. The distribution of the source after applying the conversion function, would become $fi(X)$. It is simple to show that $H(fi(X)) \leq H(X)$ [CT91], with equality if and only if fi is an injective function. Since the function fi is injective from the 32 bit floating point numbers to 32 bit integers, for Shannon entropy, we have $H(fi(X)) = H(X)$. For min-entropy we have the same result:

$$
\begin{aligned}
H_\infty(fi(X)) &= -\log \max_y \Pr[fi(X) = y] \\
&= -\log \max_y \Pr[X \in fi^{-1}(y)] \\
&= -\log \max_x \Pr[X = x] \\
&= H_\infty(X)
\end{aligned}
$$

where $y \in \{2^{-16}, \ldots, 2^{16}\}$ and $x = fi^{-1}(y)$ for all y. □

In general, applying a function on an entropy source will decrease the Shannon and min-entropy unless the function has certain properties. For Shannon entropy being injective is the necessary and sufficient condition to preserve the entropy. For min-entropy however, being injective is sufficient but not necessary to preserve the entropy.