Creating Randomness with Games

Jaak Henno
Tallinn University of Technology
Tallinn, Estonia
jaak.henno@ttu.ee

Hannu Jaakkola
Tampere University of Technology, Pori Department
Pori, Finland
hannu.jaakkola@tut.fi

Jukka Mäkelä
University of Lapland,
Rovaniemi, Finland
jukka.makela@ulapland.fi

***Abstract*—In our more and more connected and open World randomness has become an endangered species. We may soon not have anything private, all out communication, interaction with others becomes publicly available. The only method to secure (temporarily) communication is mixing it with randomness – encoding it with random keys. But massive use of the same sources of randomness often reveals, that those sources were not perfectly random, rapid achievements of technology can render some previously secure sources un-secure and in our competition-based world we can never be quite certain with 'given from above' products of their quality – in order to beat each other all producers are 'cutting corners' and this clearly shows in tremendous increase of all kind of security accidents. Thus there is an urgent constant need for new, independent sources of randomness. The common area where we constantly encounter randomness are computer games. When players try to beat others they constantly invent new moves and tactics, i.e. introduce new randomness, which can be captured and used e.g. for generating secret keys in multi-player game communication. Here is presented a method to produce with games of chance m-ary (m>2) random integer sequences utilizing a finite automaton; for assessment of random sequences is introduced a notion of k-randomness. The obtained randomness can be used e.g. for creating secure communication/chat systems in massively multiplayer games.**

Keywords— entropy, randomness, video games, finite-state machines, human behavior, cyclic order, k-random sequences

I. Introduction

Nicholas Negroponte, founder and Chairman Emeritus of the Massachusetts Institute of Technology's Media Lab noticed already in 1995, that humanity is 'moving from a world of atoms to the world of bits' and replacing 'manipulating atoms with manipulating bits' – virtual things [1]. Manipulating atoms, physical things is left for automata and robots, we only create programs which rule these automata and robots. Traditional fields – agriculture, manufacturing and construction are currently producing only 35% of all values, the resti is produced in mental/information sphere, where the input for production of new values is data.

We are irreversibly moving all our human environment into virtual environment. Google, Facebook, Amazon know about us more and more and we do not even know, what all they know about us.

Thus it is becoming increasingly important to keep our privazy, our 'self', our data and our communication and for this we need randomness. Randomness has become a commercial product, several countries are creating public random number generators [2] and with rapidly increasing amount of communications and data we also have growing need for new randomness for encryption. Randomness has also many other applications - in scientific computing, operating systems etc.

Encryption ciphers are based on modifying messages using random data. But they are only temporary measure, when some encrpytion method/cypher is broken it becomes worthless and the randomness used in it is not any more randomness. Data breaches are increasing by more than 20 percent in a year [3] and they have become the most worrying feature of Internet [4]. But with every breach also the randomness used in these encryptions is loosing its value. Thus we constantly need new sources of randomness. New computing environments - the coming era of IoT (Internet of Things), virtual/cloud servers etc. all increase need for randomness.

To satisfy this all the time growing need for randomness are emerging dedicated services to serve entropy, i.e. random data [5]. For delivering provided entropy to users were proposed a special new protocol 'Entropy as a Service' [6]. But for delivery entropy also should be encrypted, thus here is a new source needing 'fresh' entropy. Thus it is not clear, whether this service will reduce the need for entropy or contrary, increase it. Everything is much simpler if the entropy is generated/collect there where it is needed.

II. Types of Randomness

It is impossible to generate random values using computer's basic operations – binary operations conjunction $\wedge$ (and), disjunction $\vee$ (or) and the unary negation $\neg$ (not) – all combinations of these connectives return single determined value (if not, then the computer is severely broken): John von Neumann commented this: "Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin". But he did not elaborate: why?, what are the "non-sinful means" of creating randomness and what actually is "randomness"? .

Computers are deterministic, orderly; randomness is the opposite of order, the absence of any pattern. Current understanding is, that 'true' randomness can be extracted only from physical processes which have rich inner structure – entropy, e.g. thermal fluctuations in computers processor, pixels found by mouse sensor when user makes some rapid random strokes, atmospheric disturbances [7] etc. These sources are 'Pure Randomness Generators' (PRG), but they are often not rich enough e.g. in network servers which do not have means for extracting external randomness.

All operating systems maintain an entropy pool. The first versions of Linux kernel generated entropy from the third derivative of differences in timings of user actions, this information is stored in two files `/dev/random` and `/dev/urandom`. This method turned out to be too slow and currently are used low-order bits (lest significant, i.e. changing most rapidly) of values from timings of user actions on keyboard, mouse movements, IDE requests etc. Tools for extracting entropy from audio [8], video [9] data etc are under development.

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

Programming language's compilers have methods to create random values, see e.g. [10],[11]. All compiters work under the Operation System (OS) and get their randomness from OS, e.g. in Windows environment randomness to all programming languages comes from the same source as to the Microsoft C/C++ compiler (and the Intel compiler) [12] or newer [13] - they use the random values generated in Common Language Runtime [14], using the entropy produced by processor. But there have been found several problems for Intel processors [15],[16] thus specialists distrust randomness produced by Intel processors [17], e.g. in Linux kernel it is only one of many inputs into the random pool. Researchers have shown that even processors built-in functions (PRG-s) for generating random values can be compromized [18], thus processors and microchips may have built-in hardware trojans [19], which can leak information leading to successful key recovery attacks. After NSA (U.S. National Security Agency) leaks from Snowden many engineers have lost faith in hardware randomness [20].

Hardware entropy pools decrease every time random numbers are generated from it. Requesting a lot of random numbers may starve programs that use the interface; this is a practical issue especially on servers that have no input devices. Other PRG sources also decrease, e.g. the online source of randomness *Random.org* [7] limits its daily available amount of free random bytes and after exceeding your free quota (currently 10^6 bits) you have either to buy new random bits or wait for next increse [21] (hopefully on the next day). But nobody wants to postpone e.g. your ecrypted chat on MMOG (Massively Multi-Player Online Game) to next day.

Thus PRG sources do not suffice, for random number generation are needed also computer algorithms.

Computers are finite devices and after a while 'fell into loop', start to repeat computed values. Thus 'calculated randomness' is pseudo-randomnees produced by pseudo-random number generators (PRNG). All PRNG-s are loops, which after their period repeat produced values.

The first value in the loop is produced by random seed, i.e. derivative from other, ususally PRG source. The next value is calculated from the previous one by some recurrennt function; common method is to use linear (for speed) recurrent functions with reduction by modulus. For these Congruential Generators (CG) is the period (length of the loop) the most important measure of security of such a generator. For the C language it should be at least $2^{32} = 32767$ [22] - a rather small number for current CPU-s and it's use (installing the Microsoft or GNU suite of compilers) requires decent computer skills. A 'high-end' PRNG–s have much bigger period, e.g. period of the 'mersenne twister' is the the Mersenne prime $2^{19937} - 1$, and use of these requires good computer skills and hardware.

Rapid development of computers has renderd obsolate many old methods. Many PRNG-s which at their introduction were considered 'good enough' have later become 'not good enough'. For example, John fon Neumann used for generation of random numbers the 'middle-square' method [23] – for the recurrence step earlier produced number was squared and then the middle digits were sliced out. This mix of number's semantics (squaring) and syntax - use only middle digits in decimal representation - was used already in 13th centyry [24] and seems good, since uncomputability results (e.g. the Rice theorem [25]) indicate, that semantic properties are undecidable from syntax. However, computers revealed that with n-bit starting number (seed) the sequence length is $\leq 8^n$ and with many seeds much shorter, e.g.

$$3792 \to 79^2 = 6241 \to 24^2 = 0576 \to 57^2 = 3249 \to 24^2 \ldots$$

The fate of many other PRNG-s is similar. For instance the RC4 (Rivest Cipher 4) which is/was used in several commercial encryption protocols and standards (e.g. in the TLS - Transport Layer Security – the base of all traffic in WWW), but was prohibited; widely known was periodicity in the random function of Microsoft PHP translator. Already in 1999 were presented general methods for prediction of CG-s [26],[27].

For assessement of quality of new PRNG-s have been constructed several suites of statistical tests – the NIST (the U.S. National Institute of Standards and Technology) suite [28], the Dieharder (Marsaglia) suite [29], Ent [30] etc. These tests check presented samples for some common regularities in everyday data, e.g. the Dieharder 3.20 implements 26 tests.

We tested with the Ent suite several established sources:

1. the first 7 KB part of the 2.1 GB file `/dev/urandom` from Ubuntu 16.04.3 (a three months old installation, used mainly for making music and rarely connected to Internet)
2. 10000 decimal digits (0,...,9) downloaded from the `Random.org` [7] (randomness from atmosphere);
3. 10000 decimal digits created using the function `window.crypto.getRandomValues();`
4. 10000 decimal digits created by Wolfram Mathworld with function `RandomInteger[]` using the default method *Rule30CA*

In the following table are shown three characteristics from the test with Ent: entropy (bits per bit), possible compression (randomness can't be compressed) and serial correlation coefficient.

TABLE I. SOME CHARACTERISTICS OF ESTABLISHED SOURCES OF RANDOMNESS

	Entropy	Compression	Correlation
/dev/urandom	0.988577	1%	0.035161
Random.org	0.919040	8%	0.060193
Windows	0.974450	2%	-0.010378
Wolfram	0.974448	2%	-0.010948

The results are rather similar except a weak preformance of atmosphere processes. But the results of these tests do not tell the whole truth. Although the randomness from Linux performed best, visual inspection (the 'Statistics' tool from free hex editor HxD) reveals, that distribution of frequencies in `/dev/urandom` is rather uneven:

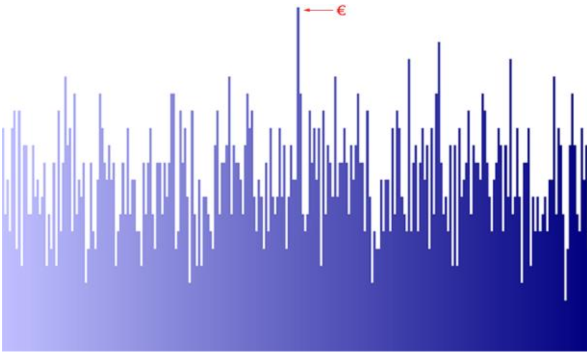

Fig. 1. Distribution of frequencies in the first 7 KB from the /dev/urandom file from Ubuntu 16.04.3; the sharp peak in the middle is the code for the € (Euro) symbol, encoded as 80h (0x80) in the windows-1252 charset. The computer has not been used for handling financial documents.

Thus statistical tests are rather uncertain method for evaluation of randomness sources. There are many surprizing dependencies in data - the above peak in € symbol code come from computer, which is rather new (used for several months) and had never been used for any kind of financial data handling.

III. Random Sequences

In several publications have been proposed methods for producing new, 'derived' randomness using (low-quality) sources as 'parents' [31], [32];[33] etc, (resembes method of gene-engineering to produce 'better' child [34],[35]). The NIST 'randomness beacon' also uses two commercial 'parent' sources' [36], but (currently) still warns: "Do not use beacon generated values as secret cryptographic keys." [37].

Reacently was announced a new breakthrough [38]. The presented method works in polynomial (in input lengths n) time and requires sources with min-entropy (negative logarithm of the maximum of probabilities in sources) $> \log^C n$, where C is a sufficiently large constant. Thus sources should be known and available for inspection beforehand. This is rather difficult to implement if the new randomness, e.g. a secret key should be created 'on the fly', e.g. participants of a multi-player game want to establish secure communication with some other players..

However, there is an area where independent random sequences are constantly produced on-line, in real time – (computer) games.

IV. Randomness in Games

Games use randomness in many ways. Game designers want their games to be not for one-time, but to be playable many times; for this the game should appear different in every play – nobody wants to replay a game where everyting repeats itself, thus randomness is an essential part of repeated, procedurally generated games.

Even more essential is randomness in the infrastructure of multiplayer games. In multiplayer games players communicate with game server(s), but they often want to communicate also with fellow players. In most gaming situations these are unknown persons, thus starting such a communication is from the computer security viewpoint a very dangerous operation, which exposes player to several serious threts – ransomware, password/account theft, credit or debit card information leakage (if player has to made payments), fake game cracs, fake apps etc [39]. Thus every such interaction with unknown players should be from the very beginning encrypted and highest security – with key, generated in players device using the randomness, already generated in the course of the game.

V. Randomness from Games

The whole idea of "game" includes randomness – nobody wants to participate in process, where out-come is (or seems) to be highly pre-determined. Gameplaying is always considered as a process, where outcome depends on randomness.

Produced by players randomness is an essential ingredient in most games and different methods for utizingg this source have been proposed e.g. in [40], [41]. In [42] authors presented a method for producing on-line in real playing time binary random strings from simple repeated games; here the principles of the proposed method are applied to produce from gameplay m-ary ($m \geq 2$) random sequences.

In the (economics theories based) texts on games the game decision mechanism is usually not detailed – it is determined by unpredictable markets. Here we follow computer science tradition (see e.g. [43]) and use for the decision mechanism finite automaton. One player is human or some established source of randomness; computer's next move is computed by an algorithm and this is also the output of the automaton. All games considered here are 'games of luck', where both players have equal chance to win and the best strategy (the Nash equilibrium) for both players is total randomness. Thus if one player is human or some established source of randomness and the other – the computer algorithm, then the (statistical) result of numerous repeated plays is also an assessment for the quality of computer-created randomness. The length of all considered here random sequences/plays is 10000, following the suggestion: "A reasonable estimate (for humanly interesting cases) reckons that some 10,000 digits would suffice" [44], p.16.

Thus in the following will be considered games, satisfying the following description.

Game is a structure $\mathcal{G} = < \mathcal{P}_1, \mathcal{P}_2, M, R, \mathcal{A} >$, where

$\mathcal{P}_1$, $\mathcal{P}_2$ are two players;

$M = \{0,1,...,m-1\}$, $m \geq 2$ - the set of legal moves (actions) of players (same set for both players); in every round both players apply simultaneously one action which initiates some change in automaton $\mathcal{A}$;

$R = [r_1, r_2]$ - player's points; at the start $r_1 = r_2 = 0$;

$\mathcal{A}$ - finite automaton, deciding the computer output (move) and payoff of the game. Here are considered simultaneous (synchroneous) games, where players produce their actions (moves) at the same time, thus the input for the automaton $\mathcal{A}$ are pairs (m_{1i}, m_{2i}), where m_{1i} is the i-s move of the first player, m_{2i} - i-s move of the second player; denote $(m_{1i}, m_{2i})^{-1} = (m_{2i}, m_{1i})$ - actions of players were switched.

Automaton's (possible) outputs are "1" ("the player $\mathcal{P}_1$ won", $r_1+=1$), "-1" ("the player $\mathcal{P}_2$ .won, $r_2+=1$), "0" – draw. Thus the automaton has four distinguished states:

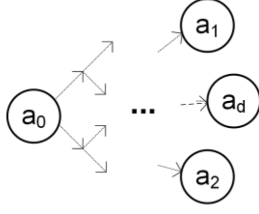

Fig. 2. Game automaton with distinguished states: a_0 – the start state, a_1 – first player won, a_2 – second player won, a_d – draw.

Here a_1 is the game start state, a_1 - here automaton outputs, that the first player won, a_2 - the second player won, a_d - draw; in any other state automaton does not produce output. Denote the set of final (for a round) states $F=\{a_1,a_2,a_d\}$. Other, intermediate states are not fixed beforehand – they emerge in gameplay.

After reaching one of states $a\in F$ automaton goes (without any input) back to state a_0 (a cycle) and game is repeated, may continue with a new round. Automaton does not have other cycles except possible loops at nodes, e.g. for moves (m_i,m_i) - both players have selected the same action. Thus the graph of the automaton is a tree (with possible loops with limited length at some nodes) and all rounds are finite. The length $D(\mathcal{A})$ of the longest round (the depth of the tree) is the depth of the game. Game is repeated and after some fixed number (e.g. 10000) of moves automaton announces if the result of the game is draw or who won.

Automaton is deterministic, i.e. in any state $a\in\mathcal{A}$, $a\notin F$ and for any move (m_i,m_j) there is a single transition $a(m_i,m_j)\rightarrow a'\in\mathcal{A}\setminus\{a_0\}$.

Transitions are in natural way extended to words from $M^{2D(\mathcal{A})}=\{(m_i,m_j)\}^{+}=\{(m_{11}m_{21})...(m_{1t}m_{2t}),t<D(\mathcal{A})\}$

$$a((m_{11},m_{21})(m_{12},m_{22})...(m_{1t},m_{2t}))=$$
$$(a((m_{11},m_{21}))((m_{12},m_{22})...(m_{1t},m_{2t}))$$

Action of words on states of automaton $\mathcal{A}$ creates partition of the set $M^{2D(\mathcal{A})}$ of words into three sub-languages:

$$\mathcal{L}_1=\{w\,|\,a_0w=a_1\},$$
$$\mathcal{L}_2=\{w\,|\,a_0w=a_2\},$$
$$\mathcal{L}_d=\{w\,|\,a_0w=a_d\}$$

Call all words $w\in M^{2D(\mathcal{A})}$ plays.

VI. Desigh of a game from symmetry considerations

In games of chance nobody wants to have worst chances by design of the game and all players actions shuld be significant, i.e. could change the result, thus these games follow the following principle:

in any move, i.e. word (m_i,m_j) from any play both players have equal chances, i.e with all other moves in the play kept the same they can change their action so that frequency of outcomes a_1,a_2 is the same, i.e. 0.5. Since there are k^2 possibly moves it follows that if k is odd, the set $\mathcal{L}_0$ can't be empty – otherwise $|\mathcal{L}_1|=|\mathcal{L}_2|$ is impossible.

Therefore the games with $D(\mathcal{A})=1$ (one round) should satisfy the following conditions.

1. All games are zero-sum, i.e. the involution $\alpha:(m_i,m_j)\rightarrow(m_i,m_j)^{-1}=(m_j,m_i)$ produces an automorphism of automaton $\mathcal{A}$, i.e. $\alpha(\mathcal{L}_1)\subseteq\mathcal{L}_2$, $\alpha(\mathcal{L}_2)\subseteq\mathcal{L}_1$, $\alpha(\mathcal{L}_d)\subseteq\mathcal{L}_d$.

2. Any substitution $\beta:\{m_0,...,m_{k-1}\}\overset{1-1}{\rightarrow}\{m_0,...,m_{k-1}\}$ of actions produces automorphism of automaton $\mathcal{A}$, which does not break the partition $\{\mathcal{L}_1,\mathcal{L}_2,\mathcal{L}_d\}$.

3. The sublanguage $\mathcal{L}_d$ contains all words (m_i,m_i), $m_i\in M$ and is minimal – it should not contain words which could be moved into $\mathcal{L}_1$ or $\mathcal{L}_2$ without breaking conditions 1,2.

Proposition. Conditions 1-3 define for given k unique (up to isomorphism) game payoff function.

Proof. Consider the set $\mathcal{M}_1=\{(m_1,m_i),m_i\in M\setminus\{m_1\}\}$, i.e. moves, where the first player selects action m_1. From the condition 3. it follows, that the set $\mathcal{L}_d$ can contain at most one of them, otherwise we could pairwise move them one to $\mathcal{L}_1$, another to $\mathcal{L}_2$ without breaking conditions 1.2.

From the condition 2. it follows, that there should be equal number of elements from the set $\mathcal{M}_1$ inside sets $\mathcal{L}_1$, $\mathcal{L}_2$, otherwise some substitutions which keep m_1 fixed, but move other actions will break the condition 2.

According to condition 2. we could re-arrange actions inside $\mathcal{L}_1$ so that $(m_1,m_2),(m_1,m_3),...,(m_1,m_{k1})\in\mathcal{L}_1$, $k1=\lfloor k/2\rfloor$. Using substitution $\chi:m_i\rightarrow m_{i+k1}$ and the property 2. we get that all sets $\{(m_i,m_{i+1}),(m_i,m_{i+2}),...,(m_i,m_{i+k1})\}$ should belong to $\mathcal{L}_1$. If k is odd, then all moves are now evenly divided between sets $\mathcal{L}_1$ and $\mathcal{L}_2$. If k is even, then from the above discussion it follows that the moves $(m_i,m_{i+k/2})$ should belong both to $\mathcal{L}_1$ and $\mathcal{L}_2$, i.e. the conditions 1.-3. can not decide their placement, thus they should be moved to set $\mathcal{L}_d$.

Thus a game with properties 1.-3.- has an unique (up to involution α) payof function, based on cyclic order [45] on moves: if moves of players are $\mathcal{P}_1(m)=m_i$, $\mathcal{P}_2(m)=m_j$, then output from the automaton $\mathcal{A}$ is:

1, iff $(m_i - m_j + k) \bmod \mathrm{m} \le \lfloor \mathrm{m}/2 \rfloor$, otherwise -1.

TABLE II. DECISION TABLE OF CYCLIC 5-ARY ORDER; E.G. $(0,2),(1,2) \in \mathcal{L}_1$, BUT $(0,3),(2,1) \in \mathcal{L}_2$

	0	1	2	3	4
0	0	+	+	-	-
1	-	0	+	+	-
2	-	-	0	+	+
3	+	-	-	0	+
4	+	+	-	-	0

In case $m = 3$ this is isomorphic to the well-known game rock-paper-scissors, which appeared in China at the beginning of Current Era. Apparently Chinese know how to use symmetry groups for inventing amusing games...

Many video games use variants of this game with greater *m*, e.g. movements of fighters can be *punch, kick, grab, push* (the next one stronger than the previous), in some games even more than ten (the order may be not linear) [46].

VII. THE ALGORITHM

In spite of total randomness of outcome are games of luck very popular. Many humans do not believe in theoretical impossibility of great winnings, but believe in their 'inborn luck'. When playing against human opponents the psychology of other players is also a factor which may change randomness of outcome, thus e.g. in USA is even a league of professional players of the "Rock, Paper, Scissors" game [47], regular tournaments [48] and programming competition [49].

Randomness is an evasive concept to define. The widely accepted definition is the Kolmogorov- Martin-Löf definitions: [50],[51]:

a sequence is random if it can't be expressed by any algorithm or device which can be described using less symbols than what are in the sequence.

This definition and other consequent definitions, e.g. [52] are using infinite sets of concepts ('any algorithm') and apply to infinite sequences, thus useless in practice for evaluating quality of a source of randomness, which produces finite sequences. For finite sequences here is introduced modified definition:

a finite sequence of m-ary integers is k-random iff it can't be created (as sequence of outputs) by any determininstic finite automaton with less than k states and m-ary input alphabet.

When $k \to \infty$ this definition yields the presented above definition. All PRNG-s are interactive (input is seed) deterministic finite automata – with the same seed they produce the same output. PRNG with cycle length *k* produces (maximally) a *k*-random sequence. Thus for designing algorithm for this game was supposed, that our algorithm is playing against finite automaton. Any finite deterministic automaton with *k* states and *m* input symbols produces a periodic sequence [53], i.e. 'goes into loop', if the length of its input is longer than $k \times m$ - there are now new possibilities for the pair (*state, input*), thus this pair has already occured and deterministic automaton produces the same output; thus here the whole sequence of pairs (*input,output*) will repeat some subsequence which already occurred. The evolutionary game theory of bounded rationality [54] of human players also predicts cyclic patterns in playing behavior [55]. Thus for successsful play one has to find when the loop begins, i.e. automaton repeats its moves. Thus the idea of the algorithm is:

scan the sequence of stored moves (input-output pairs) and when you see a situation similar to the current one (the sequence of last moves already appeared earlier) make the move that then (in the previous situation) would make you winning.

Suppose the sequence of moves in a game up to now is

$a_0(m_{11},m_{21})(m_{12},m_{22}),...,(m_{1n},m_{2n}),...(m_{1k},m_{2k})(m_{1k+1},m_{2k+1})...,$
$(m_{1n},m_{2n}),...(m_{1k},m_{2k})$

and $(m_{1n},m_{2n}),...,(m_{1k},m_{2k})$ is the longest repeated subsequence of moves (looking from the current state backwards). Then algorithm should select move, which winns in state (m_{1k+1},m_{2k+1}).

For instance, in the following situation from a real play of 3-ary game (moves follow in pairs, first human then computer, e.g. on the second move human played '1', computer – '2') computer discovered a repeated sequence (underlined), thus its next move will be '2':

1, 1, 1, 2, 2, 2, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 2, 2, 2, 2, 1, 0, 0, 1, 1, 0, 2, 0, 0, 2, 2, 1, 1, 2, 2, 2, 1, 1, 1, 2, 2, 2, 0, 2, 2, 0, 1, 1, 1, 1, 2, 0, 0, 2, 2, 2, 2, 1, 1, 0, 0, 1, 2, 1, 0, 0, 2, 2, 0, 2, 0, 2, 0, 0, 0, 0, 2, 0, 0, 2, 0, 1, 0, 0, 2, 2, 0, 2, 0

The above sequence shows 45 moves (there are 90 symbols); length of the repeated subsequence is 4 moves, thus the all sequence is 41-random.

The algorithm has been implemented in several browser games [56].

VIII. TESTS

We tested the output of algorithm – a new random sequence - in many plays. Against human players (students from the game programming course in the Tallinn University of Technology) computer was in most cases already winning if the length of the game was >30. Humans are not sufficiently random to beat computer, especially if the memory requirements (length of the game) grows; it seems that here works the famous human short-term memory size principle [57].

As opponent players for testing were used established sources of randomness: Javascript's functions *Math.random()*, *window.crypto.getRandomValues()* (the game is implemented in browser [58]), random numbers produced by Wolfram's Mathematica and table table of 10000 random integers downloaded from *https://www.random.org/*.

Tests indicated, that the algorithm plays quite well against all these common sources of 'computed' randomness, i.e. its own randomness is on the same level. Below is a table of results from three tests, each a 10 series of plays, each 10000 rounds with $m = 3$; player $\mathcal{P}_1$ is in the first test random numbers produced by the Javascript function *Math.random()*, in the second – random numbers produced

by the function *RandomInteger[]* of Wolfram's *Mathematica* (using the default rule *Rule30CA*) and in the third – random numbers produced by function *window.crypto.getRandomValues()*; player $\mathcal{P}_2$ is our algorithm; $\mathcal{L}$ is the length of longest cycle. The last row indicates, how many times each player won and length of the longest repeated sequece.

TABLE III. RESULTS OF TESTS

$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{L}$	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{L}$	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{L}$
3350	3365	16	3403	3289	16	3356	3287	18
3396	3237	16	3369	3242	20	3277	3285	16
3328	3332	16	3392	3286	16	3281	3351	18
3428	3209	18	3392	3317	18	3342	3305	18
3310	3377	16	3512	3163	16	3299	3405	16
3369	3365	16	3424	3278	18	3366	3259	16
3360	3345	16	3440	3316	18	3367	3263	16
3315	3402	16	3355	3265	18	3283	3446	20
3322	3412	18	3409	3301	19	3383	3354	16
3294	3364	16	3330	3453	16	3324	3314	16
4	6	18	9	1	20	6	4	20

These results show, that used in tests sequences were (at least) 9980-random according to the above definition – they did not contain repeated sequences longer than 20 moves.

In the following table are discretizised results (showing not actual results, but how many times player was better than the opponent) from 10×10000 series of tests against random numbers table from *Random.org* (the first column in all three subpartions), Javascript function *Math.Random()* (the second column) and against the function *window.crypto.getRandomValues()* (the third column); the last row showss summary results.

TABLE IV. DISCRETIZISED RESULTS OF TESTS

Better $\mathcal{P}_1$			Better $\mathcal{P}_2$			Draw		
4	7	2	4	3	8	2	0	0
6	4	7	4	3	3	0	3	0
5	7	1	3	2	9	2	1	0
6	3	1	3	7	9	1	0	0
2	5	2	7	4	8	1	1	0
3	4	4	5	5	5	2	1	1
4	6	4	4	4	6	2	0	0
4	4	3	5	5	7	1	1	0
7	4	3	3	2	7	0	4	0
4	4	4	4	4	3	2	2	3
85	100	75	88	74	115	27	26	10

As seen from this table, our algorithm was nearly on the same level against *Math.Random()*, slightly outperformed the randomness from Random.org and slightly lost to *window.crypto.getRandomValues()*; tests against other sources of computer randomness produced rather similar results.

IX. CREATED AND ITERATED RANDOMNESS

As output (new randomness) could be used two sequences – the sequence of 'full' moves (pairs of moves from player and computer) or the sequence of only computer-generated moves (twise shorter). We tested both as the source of random sequence against our computer's algorithm. In the following table are results from 10 series of plays, each 10000 rounds with $m = 3$; player $\mathcal{P}_1$ is in the first series (the first three columns of the table) generated in a previous game (10000 moves against Javascript *Random()*) sequence of full moves (pairs), in the second (the last three columns) – sequence of computer moves; player $\mathcal{P}_2$. is our algorithm.

TABLE V. TESTS AGAINST RANDOMNESS, CREATED IN GAME

$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{L}$	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{L}$
3298	3344	16	3403	3275	16
3377	3351	16	3328	3337	16
3439	3303	16	3391	3342	16
3419	3284	16	3375	3297	18
3328	3376	16	3490	3272	18
3471	3212	16	3408	3273	18
3360	3294	20	3379	3343	20
3367	3314	16	3342	3370	16
3513	3250	16	3376	3288	16
3416	3362	16	3362	3316	18
7	3	20	8	2	20

According to the above table the created in the game randomness already mostly outperformed our algorithm, its results are better than that of commonly established sources, compare with TABLE III; however, the *k*-randomness is on the same level.

When the generation process was iterated, i.e. generated randomness was used as input for the next play, it become more difficult to predict and our algorithm started to loose. In the following table are results of play against randomness, created on third iteration, i.e. after three rounds of 10x50000 moves; player $\mathcal{P}_1$ is in the first column the table of full moves (pairs), in the second – sequence of computer-generated moves.

TABLE VI. TESTS WITH ITERATED RANDOMNESS

$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{L}$	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{L}$
16811	16740	22	16617	16834	24
16840	16550	18	16636	16759	18
16785	16599	20	16729	16592	20
16779	16701	18	16703	16641	20
16777	16412	18	16601	16720	18
16928	16672	18	16801	16682	20
16904	16610	20	16757	16589	20
16902	16599	18	167877	16547	22
17017	16445	22	16581	16702	20
16680	16655	20	16458	16854	18
10	0	22	5	5	20

X. CONCLUSIONS

Above was presented a method for creating new random sequences with repeated play of a game of chance using as input either a human player or an established source/table of random numbers. Tests show, that produced randomness is quite on level with established sources and can be used e.g. for creating encryption keys for messages in a chat system of a multiplayer online game; such a system is currently under development.

[1] Negroponte, Nicholas. 1995. Being Digital. New York: A. A. Knopf

[2] Sophia Chen. Why are countries creating public random number generators? Science, Jun 28, 2018

[3] BSA. Encryption: Why It Matters. Retrieved May 9, 2018 from http://encryption.bsa.org/

[4] Fortinet 2018. Data Breaches Are A Growing Epidemic. How Do You Ensure You're Not Next? Retrieved May 08,2018 from https://www.fortinet.com/blog/threat-research/data-breaches-are-a-growing-epidemic--how-do-you-ensure-you-re-n.html

[5] Nist 2016. Entropy as a Service. https://csrc.nist.gov/projects/entropy-as-a-service

[6] EaaSP 2018. EaaSP - Entropy as a Service Protocol. https://github.com/usnistgov/EaaS

[7] https://www.random.org

[8] vanheusden.com 2018. audio entropy daemon. https://vanheusden.com/aed/

[9] vanheusden.com 2018. video_entropyd. https://vanheusden.com/ved/

[10] Generating Random Data in Python. https://realpython.com/python-random/

[11] How to generate random numbers, characters, and sequences in Scala. https://alvinalexander.com/scala/how-to-generate-random-numbers-characters-sequences-in-scala

[12] CryptGenRandom. https://docs.microsoft.com/en-us/windows/desktop/api/wincrypt/nf-wincrypt-cryptgenrandom

[13] RtlGenRandom function. https://docs.microsoft.com/en-us/windows/desktop/api/ntsecapi/nf-ntsecapi-rtlgenrandom

[14] Common Language Runtime (CLR) overview. https://docs.microsoft.com/en-us/dotnet/standard/clr

[15] A Provable-Security Analysis of Intel's Secure Key RNG. https://eprint.iacr.org/2014/504.pdf

[16] Gagallium : How I found a bug in Intel Skylake processors. http://gallium.inria.fr/blog/intel-skylake-bug/

[17] The Register. Torvalds shoots down call to yank 'backdoored' Intel RdRand in Linux crypto. Sept 10, 2013, https://www.theregister.co.uk/2013/09/10/torvalds_on_rrrand_nsa_gchq/

[18] G. T. Becker, F. Regazzoni, C. Paar, W. P. Burleson. Stealthy Dopant-Level Hardware Trojans. Journal of Cryptographic Engineering, April 2014, Volume 4:1, pp 19–31

[19] M. Ender, S. Ghandali, A. Moradi, C. Paar. The First Thorough Side-Channel Hardware Trojan. https://eprint.iacr.org/2017/865.pdf

[20] arstechnica. "We cannot trust" Intel and Via's chip-based crypto, FreeBSD developers say. https://arstechnica.com/information-technology/2013/12/we-cannot-trust-intel-and-vias-chip-based-crypto-freebsd-developers-say/

[21] Random.org. Your Quota. https://www.random.org/quota/

[22] ISO/IEC 9899:2011. https://www.iso.org/standard/57853.html

[23] John von Neumann, "Various techniques used in connection with random digits," in A.S. Householder, G.E. Forsythe, and H.H. Germond, eds., Monte Carlo Method, National Bureau of Standards Applied Mathematics Series, vol. 12 (Washington, D.C.: U.S. Government Printing Office, 1951): pp. 36-38

[24] I. Ekeland. The Broken Dice, and Other Mathematical Tales of Chance. University of Chicago Press 1993, pp 1-190 ISBN: 9780226199924

[25] H. G. Rice. "Classes of Recursively Enumerable Sets and Their Decision Problems". Trans. Amer. Math. Soc. 74 1953, pp 358–366

[26] H. Krawczyk. How to predict congruential generators. Journal of Algorithms, vol. 13:4, December 1992, pp 527-545

[27] J. Stern. Secret linear congruential generators are not cryptographically secure. 28th Annual Symposium on Foundations of Computer Science (sfcs 1987), DOI: 10.1109/SFCS.1987.51

[28] NIST SP 800-22. A Statistical Test Suite for Random. https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-22r1a.pdf

[29] R.G.Brown. Dieharder: A Random Number Test Suite. https://webhome.phy.duke.edu/~rgb/General/dieharder.php

[30] ENT. A Pseudorandom Number Sequence Test Program. http://www.fourmilab.ch/random/

[31] J. Bourgain. More on the sum-product phenomenon in prime fields and its applications. International Journal of Number Theory, 01(01):1–32, 2005

[32] Y. Dodis, D. Wichs. Non-malleable extractors and symmetric key cryptography from weak secrets. STOC 2009, pp 601–610

[33] Xin Li. Improved non-malleable extractors, non-malleable codes and independent source extractors. Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing 2017, pp 1144–1156

[34] Engineering the Perfect Baby. https://www.technologyreview.com/s/535661/engineering-the-perfect-baby/

[35] A Chinese scientist says he edited babies' genes. What are the rights of the genetically modified child? https://www.washingtonpost.com/news/monkey-cage/wp/2018/12/06/a-chinese-scientist-says-hes-edited-babies-genes-what-are-the-rights-of-the-genetically-modified-child

[36] NIST Randomness Beacon. https://www.nist.gov/programs-projects/nist-randomness-beacon

[37] NIST. Computer Security Division. https://beacon.nist.gov/home

[38] E. Chattopadhyay, D. Zuckerman. Explicit Two-Source Extractors and Resilient Functions. https://eccc.weizmann.ac.il//report/2015/119/

[39] Top 5 threats for online gamers and how to avoid them. https://www.welivesecurity.com/2016/08/31/top-5-threats-online-gamers-avoid/

[40] R. Halprin, M. Naor. Games for Extracting Randomnes. SOUPS '09 Proceedings of the 5th Symposium on Usable Privacy and Security 2009,

[41] M. Alimomeni, R. Safavi-Naini. Human Assisted Randomness Generation Using Video Games. https://eprint.iacr.org/2014/045.pdf

[42] Henno, J., Jaakkola, H., Mäkelä, J. Using games to understand and create randomness. SQAMIA2018 - Proceedings of the 7th Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, Vol. 2217, CEUR-WS, http://ceur-ws.org/Vol-2217/

[43] E. Ben-Porath. Repeated Games with Finite Automata. Journal of Economic Theory 59:1, 1993, pp 17-32

[44] Sergio B. Volchan. What Is a Random Sequence? https://www.maa.org/sites/default/files/pdf/upload_library/22/.../Volchan46-63.pdf

[45] A Quilliot. Cyclic Orders. European Journal of Combinatorics 10:5, 1989, pp 477-488

[46] Gang Beasts Controls Guide. https://www.gameskinny.com/ly5jv/gang-beasts-controls-guide

[47] USA Rock Paper Scissors League. https://myspace.com/usarps

[48] Rock Paper Scissors tournament rules. https://do317.com/p/rpsrules

[49] Rock Paper Scissors Programming Competition. http://www.rpscontest.com/

[50] Kolmogorov, A.N. (1965). Three Approaches to the Quantitative Definition of Information. Problems Inform. Transmission. 1(1), pp 1–7

[51] Martin-Löf, P. (1966). The definition of random sequences. Information and Control. 9 (6): 602–619

[52] Schnorr, C. P. (1971). A unified approach to the definition of a random sequence. Mathematical Systems Theory. 5 (3), pp 246–258

[53] A. Cobham, Uniform tag sequences, Math. Systems Theory, 6 (1972), pp 164–192

[54] H. Matsushima. Bounded Rationality in Economics: A Game Theorist's View. The Japanese Economic Review (1997), 48:3, pp 293-306

[55] Z. Wang, B. Xu, H. Zhou. Social cycling and conditional responses in the Rock-Paper-Scissors game. eprint arXiv:1404.5199, https://ui.adsabs.harvard.edu/#abs/arXiv:1404.5199

[56] http://staff.ttu.ee/~jaak/games/

[57] GA. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. Psychological Review. (1956) 63, pp 81–97

[58] Rock, Paper, Scissors – m-ary. http://staff.ttu.ee/~jaak/games/paber_kivi_m_ary_js.htm