CHAOS MACHINE: DIFFERENT APPROACH TO THE APPLICATION AND SIGNIFICANCE OF NUMBERS

Maciej A. Czyzewski
mail@maciejczyzewski.me

May 16, 2016

Abstract. In this paper we describe a theoretical model of chaos machine, which combines the benefits of hash function and pseudo-random function, forming flexible *one-way* push-pull interface. It presents the idea to create a universal tool (design pattern) with modular design and customizable parameters, that can be applied where *randomness* and *sensitiveness* is needed (random oracle), and where appropriate construction determines case of application and selection of parameters provides preferred properties and security level. Machine can be used to implement many cryptographic primitives, including cryptographic hashes, message authentication codes and pseudo-random number generators. Additionally, document includes sample implementation of chaos machine named Naive Czyzewski Generator, abbreviated NCG, that passes all the Dieharder, NIST and TestU01 test sets. Algorithm was designed and evaluated to be a cryptographically strong, inasmuch as indistinguishable from a uniform random function. The generator was developed to work as cryptographically secure pseudo-random number generator, collision resistance hash function or a cryptographic module. One can target a given period length by choosing the appropriate space parameter, i.e., for a given parameter m, algorithm is claimed to have period between 2^{8m} to 2^{16m}.

Keywords. chaos machine · dynamical system · chaotic behavior · randomness · control theory · chaotic map · pseudo-random function · push-pull interface

Introduction

A lot of research has gone into chaos and randomness theory. Development in computer software and applications continues to be very dynamic. Each software problem requires different tools and algorithms to solve it effectively and achieve best results. As a consequence, we witness the announcement of new projects in quick succession with multiple updates. The engineer's problem is how to decide which method will suit his needs best.

Random numbers have been one of the most useful objects in statistics, computer science, cryptography, modeling, simulation, and other applications though it is very difficult to construct true randomness. Many applications of randomness have led to the development of several methods for generating random data. The generation of pseudo-random numbers is an important and common task in computer programming. Cryptographers design algorithms such as RC4 and DSA, and protocols such as SET and SSL, with the assumption that random numbers are available.

Hash is the term basically originated from computer science where it means chopping up the arbitrary length message into fixed length output. Hash tables are popular data structures for storing key-value pairs. A hash function is used to map the key value to array index, because it has numerous applications from indexing, with hash tables and bloom filters; to spell-checking, compression, password hashing and cryptography. They are used in many different kinds of settings and accordingly their security requirement changes.

Hash functions were designed for uniqueness, while pseudo-random functions for randomness[1]. There is a tendency for people to avoid learning anything about such subroutines [Knu73]; quite often we find that some old method that is comparatively unsatisfactory has blindly been passed down from one programmer to another, and today's users have no understanding of its limitations. Therefore, appears the idea to create a universal tool (design pattern) with modular design and customizable parameters, that can be applied where *randomness* and *sensitiveness* is needed (random oracle), and where appropriate construction determines case of application and selection of parameters provides preferred properties and security level. It should be so easy to use that an intelligent, careful programmer with no background in cryptography has some reasonable chance of using such tool in secure way.

In this paper we describe a theoretical model of chaos machine, which combines the benefits of hash function and pseudo-random function, forming flexible *one-way*[2] push-pull interface, where the construction and selection of parameters determines usage. It generates sequences of pseudo-random numbers that are unique and sensitive to the initial conditions and inputs. Therefore, machine can be used to implement many cryptographic primitives, including cryptographic hashes, message authentication codes and pseudo-random number generators.

[1] In practice, hash functions are chosen to spread hash values uniformly (pseudo-randomly). However, some techniques of hashing does not require this principle.

[2] One-way functions are easy to compute but it is very difficult to compute their inversed functions.

Part I

Definition and Analysis of the Model

1. Overview

Chaos theory started more than thirty years ago and changed our world view regarding the role of randomness and determinism, these theories present some interesting aspects in cryptography:

- Chaotic systems are highly sensitive to initial conditions and exhibits chaotic behavior. The main characteristics of chaotic systems make them intuitively interesting for their application in cryptography. Edward Lorenz used to say "Chaos: When the present determines the future, but the approximate present does not approximately determine the future.".
- Randomness is the lack of pattern or predictability in events, a phenomenon located at a single point in spacetime. A pseudo-random process is a process that appears to be random but is not. Pseudo-random sequences typically exhibit statistical randomness while being generated by an entirely deterministic causal process.

1.1. Concept

Idea was to create simple model (design pattern) containing several elements, that lets programmer to create own constructions and tools where *randomness* and *sensitiveness* is needed (random oracle). These main assumptions are presented as:

- **Difusion**: a small difference in the input produces a very different output.
- **Deterministic Pseudo-randomness**: a deterministic procedure that produces pseudo-randomness.
- **Algorithmic Complexity**: a simple algorithm that produces highly complex outputs.

Where selection of parameters provides preferred properties and security level. Therefore, machine contains three external variables: *initial secret key*, *time parameter*, and *space parameter*.

- **Initial Secret Key** (Starting Variable). Is a fixed-size input to a chaos machine that is typically required to be random or pseudo-random. Setting the initial secret key is an example of using machine as the MAC[3] algorithm.
- **Time Parameter** (Time Cost). That determines the number of rounds of computation that machine performs. The larger the time parameter, the longer the output computation will take. As computational power increases, users can increase this time parameter to keep the number of wall-clock seconds required to compute each sequence near-constant.
- **Space Parameter** (Memory Cost). Defines the number of dynamical systems to be used in the machine. Concomitantly indicates how many bytes of working space the buffer will require during its computation, because each system needs his own space.

Model contains dynamical system of *push-pull-reset functions* and *buffer space*. Components below are forming machine interface:

- **Push Function** (Input). Is primarily the input function, it absorbs bit string (typically 32 bits or 16 bits value, relatively small) and uses in system evaluation (control theory, it will be discussed later).
- **Pull Function** (Output). It contains chaotic map and construction of pseudo-random functions, which can be freely replaced. The output of pull function is a bit string of fixed length (e.g., 32 or 16 bits).
- **Reset Function** (Reset). This function clears the buffer. After this, operation machine is in the initial state.

Applying our recommendations, programmer can build effective machine for wide range of applications. An example is described in the second part of this document.

2. Model

2.1. Dynamical System

Chaos is a non-periodic, long-term non-predictive behavior that can be generated by certain nonlinear dynamical systems [KT01]. The chaotic systems are inherently deterministic given the initial state of the system. The chaotic behavior is a result of the exponential sensitivity of the system to the initial state that can not be exactly determined in practice.

2.1.1. Discrete & Continues

There is two main models of evolution law [Via01]. The first one corresponds to transformations $f : M \to M$ on a space

[3] A message-authentication code (MAC) produces a tag t from a message m and a secret key k. The security goal is for an attacker, even after seeing tags for many messages (perhaps selected by the attacker), to be unable to guess tags for any other messages.

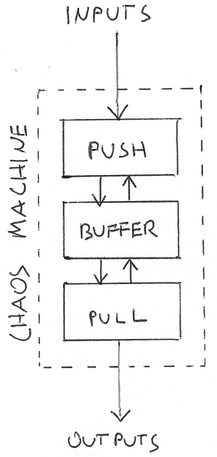

Figure 1: Machine contains 3 main elements: push function, buffer space, pull function. Its initialized by tuple (K, t, m), where K is initial secret key, t is time parameter and m is space parameter.

```
.........................................
.                  BUFFER               .
.          ...                          .
.        . 1 .  2    3    ...    m      .
.     ../  / \ ........                 .
.   /     /    \...     \...............  .
.  / PUSH .       \............  PULL  \ .
.  \...../                      \......./ .
. (chaotic)     hybrid system   (random) .
.........................................
 chaos machine - multiple dynamical systems
```

Figure 2: Buffer space with the representation of systems.

M, the points of which describe the different states of the system. The orbit of each $x_0 \in M$ is the sequence $(x_n)_n$ defined by $x_n = f(x_{n-1})$ for $n > 1$.

Another model are continues-time flows $f^t : M \to M$, $t \in \mathbb{R}$, that is, one-parameter families of transformations satisfying $f^{t+s} = f^t \circ f^s$ for $t, s \in \mathbb{R}$, and $f^0 = id$. The orbit of $x_0 \in M$ is the curve $x_t = f^t(x_0)$, where $t \in \mathbb{R}$. Assuming the flow depends smoothly on time t, there is an associated vector field F on M, defined by:

$$F(x) = \frac{d}{dt} f^t(x)|_{t=0}$$

2.1.2. Computer Limitations

A computer is a finite state machine that may be viewed as a discrete system. As in the discrete-time case, the solutions for the function f are no longer curves, but points that hop in the phase space. The orbits are organized in curves, or fibers, which are collections of points that map into themselves under the action of the map. However, some methods have been developed to represent real-world continuous systems as discrete systems [TLB14].

2.1.3. Transition

In our model, each map f, chaotic or not, will be called a position *transition function* of dynamical system designated as T. To watch out for the multidimensionality of point x, we will write it as a vector $\vec{x}$.

Theorem 2.1 (Coordinate Function)
If $\vec{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ is a point in the n-space then we define the i-th coordinate function $p_i : \mathbb{R}^n \to \mathbb{R}$ as $p_i(\vec{x}) = x_i$

Theorem 2.2 (Transition Function)
More generally, transition function is a map from one position to another. In the theory of dynamical systems, a map denotes an evolution function used to create discrete dynamical systems. Using definition of point from 2.1, transition function $T : \mathbb{R}^n \to \mathbb{R}^n$ is defined:

$$T(\vec{x}) = T([x_1, \ldots, x_n]) = [p_1, \ldots, p_n]$$

Then, transition in dynamical system is described as:

$$\vec{x_i} = T(\vec{x_{i-1}})$$

This map takes a point $\vec{x_{i-1}} = (x_1, \ldots, x_n)$ in the space and maps it to a new point $\vec{x_i} = (p_1(\vec{x_{i-1}}), \ldots, p_n(\vec{x_{i-1}}))$.

In dynamical system's theory, the *Gingerbreadman map* is a chaotic two-dimensional map which was studied by R. Devaney [Dev84] since 1984. It is given by the piecewise linear transformation:

$$\begin{cases} x_{n+1} = 1 - y_n + |x_n| \\ y_{n+1} = x_n \end{cases}$$

Using theorem from 2.1, it can be written as set of coordinate functions:

$$p_1(\vec{x}) = 1 - x_2 + |x_1| \tag{1}$$
$$p_2(\vec{x}) = x_1 \tag{2}$$

Then, *Gingerbreadman map* is equivalent to the transition $T : \mathbb{R}^2 \to \mathbb{R}^2$ given by:

$$\vec{x_n} = T(\vec{x_{n-1}}) = T([x_1, x_2]) = [p_1, p_2] = [1 - x_2 + |x_1|, x_1]$$

This map takes a point $\vec{x_{n-1}} = (x_1, x_2)$ in the plane and maps it to a new point $\vec{x_n} = (p_1(\vec{x_{n-1}}), p_2(\vec{x_{n-1}}))$. Coordinate function (1) and (2) calculates coordinates of the next state in chaotic map.

Theorem 2.3 (Dynamical System)
In general sense, dynamical system $\phi : t \times \mathbb{R}^n \to \mathbb{R}^n$ where t is time (for discrete $t = \mathbb{Z}$ or $t = \mathbb{N}$), can be described as:

$$\vec{x_i} = T(\vec{x_{i-1}})$$

Using the 2.2 theorem and definition of evolution function ϕ, superscript of function T defines time (or iteration):

$$\phi_i(\vec{x}) := T(\phi_{i-1}(\vec{x})) = T^i(\vec{x})$$

Evolution (trace) of discrete dynamical system from the initial state $\vec{x_0} \in \mathbb{R}^n$:

$$\vec{x_0} = T^0(\vec{x_0}) \to T^1(\vec{x_0}) \to T^2(\vec{x_0}) \to T^3(\vec{x_0}) \to \ldots$$

2.1.4. Vector of Systems

Machine consists of a multiple dynamical systems. Each system has its own evolution function ϕ, hence its own transition function T. The space containing the multiple systems is called *buffer space*. It can be defined as a row vector of m transition functions ($1 \times m$ matrix):

$$S = \begin{bmatrix} T_1 & T_2 & \dots & T_m \end{bmatrix}$$

Variable m determines the number of dynamical systems, known as *space parameter*. Space above has been presented visually on the figure 2. Systems must be initialized with the initial states (points). To do this, we can use initialization vector, known as *initial secret key* defined by the variable K:

$$K = \begin{bmatrix} \vec{k_1} & \vec{k_2} & \dots & \vec{k_m} \end{bmatrix}$$

Then buffer space is described as follows:

$$S = \begin{bmatrix} T_1^0(\vec{k_1}) & T_2^0(\vec{k_2}) & \dots & T_m^0(\vec{k_m}) \end{bmatrix}$$

Where notation S_i means i-th dynamical system. Buffer space can be called as function family of evolution functions. However, it acts as "entropy pool" (seed pool), which evolves in time. Therefore, push function is sometimes called an "entropy collector" (section 2.1.6).

2.1.5. Control Theory

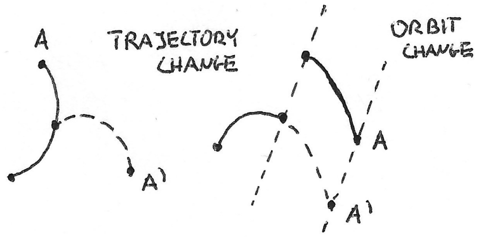

Figure 3: Changes in phase space. Point $A\prime$ represents the original destination, while A chosen.

Control theory is an interdisciplinary branch of engineering and mathematics that deals with the behavior of dynamical systems with inputs, and how their behavior is modified by feedback (figure 3). In our model, it will be a collection of operations which aims to control evolution function on the basis of input [Har06]. Below example of *Henon map*, in general sense i-th coordinate function p_i:

$$\underset{\text{next } i\text{-th coordinate}}{p_i(\vec{x})} = 1 - \overset{\text{parameter}}{a}\,\underbrace{x_1^2 + x_2}_{\text{previous coordinates}}$$

Each p_i function modifies *position* and may have own *parameters*. Changing these variables will change phase space of this specific dynamical system. It can be done in two ways:

- **Orbit Change** (Position Change). This is done by selecting a new position, value $\vec{x}$, consequently changing the orbit. Then i-th dynamical system can be described as follows: $S_i = T_i^0(\vec{x})$.

- **Trajectory Change** (Parameters Change). It involves selecting a new parameter values for i-th coordinate function p_i of selected dynamical system from the buffer. In *Henon map* it would be choosing new parameter a.

  New parameters chosen for single transition function should be stored in *parameters space* (section 3.4). However, chosen parameters can be mutual with all transition functions in buffer. Then additional space is unnecessary.

2.1.6. Hybrid System

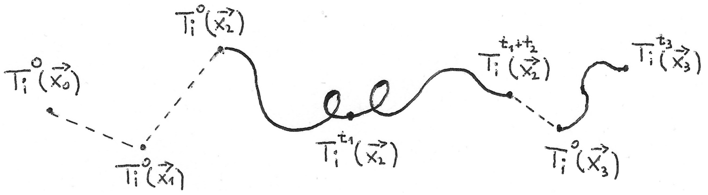

Figure 4: Phase space of S_i. Initial state $\vec{x_0} = \vec{k_i}$. Points $\{\vec{x_1}, \vec{x_2}, \vec{x_3}\}$ are selected by the push function - jumps. Pull function has evolved at $\{t_1, t_2, t_3\}$ time - flow.

A hybrid system is a dynamic system that exhibits both continuous and discrete dynamic behavior - a system that can both "flow" described by a differential equation and "jump" described by a difference equation or control graph. In chaos machine, when performed (figure 4):

- **Push function** (chaotic), *jumps* are governed by the collection of operations which aims to control evolution functions (section 2.1.5). It modifies inflicted systems, based on initial secret key and input.

- **Pull function** (random), *flow* is governed by transition function T that operates on buffer space. Its calculating evolution trace from the current position $\vec{x}$ of S_i:

  $$T_i^j(\vec{x}) \to T_i^{j+1}(\vec{x}) \to \dots \to T_i^{j+t}(\vec{x})$$

  Where j determines current time (2.3 theorem), variable t determines the length of time flow, known as *time parameter*. After simplifications:

  $$S_i = T_i^{j+t}(\vec{x})$$

In addition, pull function may contain theorem of random dynamical system; equations of motion. Chaotic trajectories even look random, and they pass many classic "tests of randomness". This in fact generates the principle of equivalence between chaotic and random systems, as discussed in [Wer13].

2.2. Push Function

Push function is primarily the input function of the machine, it absorbs bit string (typically 32 bits or 16 bits value, relatively small) and uses in system changes. The results are used later by the pull function. The push procedure is a collection of operations which aims to control evolution functions. It modifies inflicted systems, based on initial secret key and input. In summary, these ideas can be presented as:

- **Selection of Group**. Which systems from the buffer space should be changed (section 2.1.4).
- **System Changes**. Controlling orbits and trajectories of selected dynamical systems - *jumps* (section 2.1.6).

Their implementation can be variously interpreted, that will be discussed in sections below. Fully implemented machine in *python* programming language is in section 3.4.

2.3. Pull Function

The output of pull function is a bit string of fixed length (e.g., 16 or 32 bits), sequences of pseudo-random numbers that are unique and sensitive to the initial conditions. Push procedure contains 3 main tasks:

- **Selection of Group**. Which systems from the buffer space should be changed (section 2.1.4).
- **System Evolution**. Calculating evolution trace of selected dynamical systems - *flow* (section 2.1.6).
- **Randomizing**. Producing pseudo-randomness from chaotic data using pseudo-random functions.

3. Interface

Goal for chaos machine is to make design that system designers can fairly easily incorporate into their own systems, and that is better at resisting the attacks we know about than the existing, widely-used alternatives. It poses the following constraints on the design of machine:

- Everything is reasonably efficient. There is no point in designing a module that nobody will use, because it slows down the application too much.
- Machine is so easy to use that an intelligent, careful programmer with no background in cryptography has some reasonable chance of using it in secure way.
- Interface should have simple design, to be able to implement many cryptographic primitives (figure 5), including cryptographic hashes, message authentication codes, stream ciphers and pseudo-random number generators.

We found that the chaos machine can be compromised by exploiting some *implementation error*. The only preventative measures we found for machine was to try to make the interface reasonably simple so that the programmer trying to use this tool in a real-world product can use it securely without understanding much about how the chaos machine works.

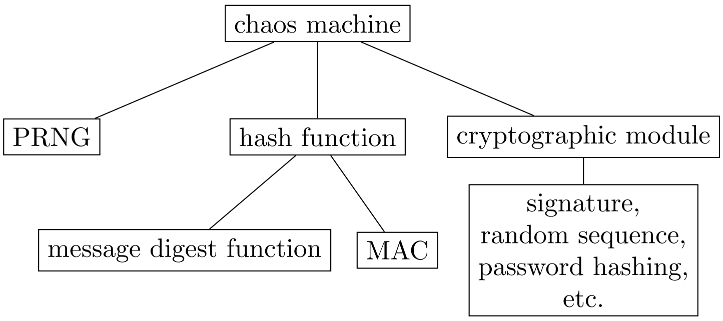

Figure 5: Specially, numbers generated by chaos machine are useful in the following kinds of applications (interface and the parameters allow implementing).

3.1. Input/Output

As it was already mention (section 1.1), chaos machine contains dynamical system of *push-pull-reset functions* and *buffer space*. These components are forming *one-way* push-pull interface, making machine intuitively simple to use.

```
// reset machine
reset();

// push to machine
push(0xDEADBEAF);
push(0xBADDCAFE);

// pull a number
uint32_t x = pull();

// x = 0xB105F00D
```

An important point which is worth to mention are the parameters, that regulates machine needs (section 1.1). Number of iterations of the push and pull function is defined as the *machine time*. In example above machine time amounts three (two push and one pull operation). A *machine timeline* is a graphical representation of a period of machine time, on which push and pull events are marked (figure 7, 8).

3.2. Structure

Appropriate order of push and pull functions in time is called a construction or design. Programmer that is using the chaos machine can freely create his own designs. Often, it will be related to the application. If we have a message, we will have to perform some operations like push to add them to the machine. However, it may be done in many ways.

3.2.1. Algorithmic Complexity

Typically, if someone uses machine as hash function, he will push all data to machine and then pull message digest at the end. But he can pull after each push or according to another rule, making chaos machine applicable for *stream hashing* or *dynamical reseeding* (figure 7). For example let's imagine:

> We have an input 256 bits message and want to generate 512 bits of output. If machine operates on 32 bits input/output system, we must provide 256/32 = 8 push and 512/32 = 16 pull functions. It's worth to mention that attacker having an output, didn't know when at the timeline pull functions have been called, because he doesn't know the construction.

Computational complexity increase dynamically with the machine time. Number of possible combinations can be written as:

$$\frac{a+1}{1} * \frac{a+2}{2} * \frac{a+3}{3} \dots \frac{a+b-1}{b-1} = \frac{(a+1)_{(b-1)}}{(b-1)!}$$

Where variables are defined as: a is as number of pull functions, b is a number of push functions. In result, $(a)_n$ is a *Pochhammer symbol.*

3.2.2. Types

Type	Relationship
Involved	Input Data, Push & Pull Sequence
Individual	Input Data, Push Sequence

Figure 6: The table presents what influence the process of output generation.

The *butterfly effect*[4] refers to a concept that small causes can have large effects. Consequently, pull function can be an event affecting future events and states. Therefore, there are two types of machines (figure 6):

- **Involved**: The push function uses and modifies the states in the buffer. Each pull leans slightly[5] trajectories or orbits from buffer (butterfly effect in each event).
- **Individual**: Push only reads states from the buffer or operates on copy. Therefore, after the same combination of push functions, pull gives the same sequence.

Visualization of difference between these types was presented at the figure 8. Sample example of involved type will be present in the part two of this document.

3.3. Application

Chaos machine lets programmer to create own constructions and tools where *randomness* and *sensitiveness* is needed (figure 5). Its presents modular design with customizable parameters. Where appropriate construction determines case of application, and where selection of parameters provides preferred properties and security level.

Practical example is provided in second part of this document. It includes sample implementation of chaos machine named Naive Czyzewski Generator, abbreviated NCG, that passes all the Dieharder, NIST and TestU01 test sets.

Below there will be analyzed the most typical applications. However, its recommend to use machine as an independent tool, due to the versatility and possibilities of use.

3.3.1. Initialization Tuple

The aim was to create a tool where user defines his needs by choosing appropriate parameters. For example, by selecting huge space parameter, machine produces long-period pseudo-random sequences[6]. Likewise, by increasing the time parameter, security level increases. Therefore, if security is a concern, chaos machine should fulfill these principles:

- Initial secret key should be provided or chosen uniformly.
- Space parameter should be big enough to provide space-hardness and long-period output sequences.
- Time parameter should be carefully selected to the appropriate level of security.

Slow one-way functions are useful as so-called password-based key derivation functions, where the relative high computation time protects against password guessing. The function can be made arbitrarily slow by increasing time parameter. In order to compete with fast pseudo-random number generators or speedy hash functions, machine should:

- Space parameter should be relatively small to the input data.
- Time parameter must be very small or even equal to one iteration.

3.4. Example of Machine

In *python*, initialization tuple (K, t, m) can be described as:

```
# Chaos Machine (K, t, m)
K = [0.33, 0.44, 0.55, 0.44, 0.33]; t = 3; m = 5
```

Below example of chaos machine that is using *Logistic map*[7]:

$$p_1(\vec{x}) = rx_1(1-x_1) \tag{3}$$

$$T(\vec{x}) = T([x_1]) = [p_1] \tag{4}$$

$$S = \begin{bmatrix} T_1^0(\vec{k_1}) & T_2^0(\vec{k_2}) & T_3^0(\vec{k_3}) & T_4^0(\vec{k_4}) & T_5^0(\vec{k_5}) \end{bmatrix} \tag{5}$$

This map is one-dimensional, so it need only one coordinate function (3). Therefore, transition function T looks like (4) and buffer space have 5 transition functions (5).

[4] It has similar properties to the *avalanche effect.*

[5] May contain theorem of random dynamical system, read more about *pullback attractor.*

[6] If period is calculable or can be approximate.

[7] Its bad example, because its period is relatively small. The map itself has the characteristics unsuitable in cryptography.

```
        Case: #1
time                   1       2       3       4       5       6       7
interface           push() | push() | pull() | push() | pull() | pull() | pull()
output                                 X               X       X       X

        Case: #2
time                   1       2       3       4       5       6       7
interface           push() | pull() | pull() | push() | push() | pull() | pull()
output                         X       X                       X       X
```

Figure 7: Sequence of events plays a major role in output generation. Its management is held by the push-pull interface which creates a timeline of events. Input data pushed into machine in both cases is the same, parameters are the same, but output is different (same length, different sequences).

```
       Type: Involved
time               1           2              3              4             5
interface     push(X_1) | push(X_2) | pull() -> Y_1 | push(X_3) | pull() -> Y_3

time               1           2                             3             4
interface     push(X_1) | push(X_2) |    skipped    | push(X_3) | pull() -> Y_2

       Type: Individual
time               1           2              3              4             5
interface     push(X_1) | push(X_2) | pull() -> Y_1 | push(X_3) | pull() -> Y_2

time               1           2                             3             4
interface     push(X_1) | push(X_2) |    skipped    | push(X_3) | pull() -> Y_2
```

Figure 8: Timeline above shows the difference between “Involved” and “Individual” type. Word “skipped” mean that none action was provided. In involved machine pull affects chaotic system (why value $Y_2 \neq Y_3$).

3.4.1. Push Function

In example below, push function is changing orbits and trajectories of all dynamical systems from buffer. Each system have own column in parameters space (for independent trajectory). In case above we are intrested in parameter r that should be between 3 and 4 (chaotic behavior). Evolution parameter is variable that helps in controling evolution functions on the basis of input.

```
# Buffer Space (with Parameters Space)
buffer_space, params_space = [], []

# Machine Time
machine_time = 0

def push(seed):
  global buffer_space, params_space, machine_time, \
    K, m, t

  # Choosing Dynamical Systems (All)
  for key, value in enumerate(buffer_space):
    # Evolution Parameter
    e = float(seed / value)

    # Control Theory: Orbit Change
    value = (buffer_space[(key + 1) % m] + e) % 1

    # Control Theory: Trajectory Change
    r = (params_space[key] + e) % 1 + 3

    # Modification (Transition Function) - Jumps
    buffer_space[key] =
      round(float(r * value * (1 - value)), 10)
    params_space[key] =
      r # Saving to Parameters Space

  # Logistic Map
  assert(max(buffer_space) < 1)
  assert(max(params_space) < 4)

  # Machine Time
  machine_time += 1
```

Pay attention, that buffer space is implemented as matrix of coordinate spaces ($n \times m$, where n is number of space dimensions of transition function, or simpler, number of p functions).

3.4.2. Pull Function

```
def pull():
  global buffer_space, params_space, machine_time, \
    K, m, t

  # PRNG (Xorshift by George Marsaglia)
  def xorshift(X, Y):
    X ^= Y >> 13
    Y ^= X << 17
    X ^= Y >> 5
    return X

  # Choosing Dynamical Systems (Increment)
  key = machine_time % m

  # Evolution (Time Length)
  for i in range(0, t):
    # Variables (Position + Parameters)
    r     = params_space[key]
    value = buffer_space[key]

    # Modification (Transition Function) - Flow
    buffer_space[key] =
      round(float(r * value * (1 - value)), 10)
    params_space[key] =
      (machine_time * 0.01 + r * 1.01) % 1 + 3

  # Choosing Chaotic Data
  X = int(buffer_space[(key + 2) % m] * (10 ** 10))
  Y = int(buffer_space[(key - 2) % m] * (10 ** 10))

  # Machine Time
  machine_time += 1

  return xorshift(X, Y) % 0xFFFFFFFF
```

To generate pseudo-randomness, pull function use *xorshift* algorithm with chaotic data. Additionaly, it includes *pullback attractor* that leans trajectory of selected dynamical system.

3.4.3. Reset Function

```
def reset():
  global buffer_space, params_space, machine_time, \
    K, m, t

  buffer_space = K; params_space = [0] * m
  machine_time = 0
```

Function above clears buffer, parameters space and resets machine time. After this, operation machine is in the initial state. It should be executed at the beginning of usage.

3.4.4. Testing

```
# Initialization
reset()

# Pushing Data (Input)
import random
message = random.sample(range(0xFFFFFFFF), 100)
for chunk in message:
  push(chunk)

# Pulling Data (Output)
while True:
  print("%s" % format(pull(), '#04x'))
  print(buffer_space); print(params_space)
```

Implemented algorithm above is fully functional example of (involved) chaos machine, which has the characteristics of *randomness* and *sensitiveness*. This construction may not be producing a high-quality outputs. However, this example is using all theoretical knowledge about the chaos machine.

Part II

Naive Czyzewski Generator: Implementation of Chaos Machine

4. Summary

The algorithm[8] is a sample[9] implementation of (involved) chaos machine. Emphasis has been placed on period that is calculable, but also on high sensitivity to initial conditions and quality of output. Algorithm passes all the Dieharder, NIST and TestU01 test sets. In addition, it shows resistance to common cryptographic attacks[10].

4.0.1. Disadvantages

The drawback is the limited quantity of the machine parameters. On each push action, it engages all possible states from buffer space. Therefore, hashing for huge buffers does not make sense (computation complexity increase). When the buffer space is huge, the algorithm is suitable only for a use as the pseudo-random number generator. Therefore, it's has prefixed "naive" and instead of the word "machine" occurs "generator". However, for small values of parameters algorithm works as fully functioning chaos machine.

4.0.2. Construction

Transition function was constructed on modified logistic map. Buffer space is formed by the tape, which cells are grouped into two types. Pull function uses half of the cells by the algorithm with known period length (e.g. linear feedback shift register). The second half belongs to the set of dynamical systems. The only requirement is an even number of cells, in order to divide into equal halves. Therefore, variants are distinguished in the following way:

Name	Algorithm
NCG	LFSR
NCG$_{Xorshift}$	Xorshift
NCG$_{KISS}$	KISS

The examples above have been tested, in particular the basic version. However, they can be freely replaced.

4.0.3. Security

Statistical tests affirm that NCG can compete with ISAAC (as CSPRNG), Blum Blum Shub or Fortuna algorithm (figure 9, 10, 11). The NCG was designed and seems to be cryptographically secure, however there is no security proof.

Ten USA dollars prize to whoever sends me the deterministic algorithm reconstructing seed/buffer space states from generated output sequence of pseudo-random numbers.

Name of Battery	Total CPU Time	Result
Alphabit	00:00:00.03	Passed
SmallCrush	00:00:07.25	Passed
Crush	00:51:15.84	Passed
BigCrush	05:30:14.75	Passed
pseudoDIEHARD	00:00:22.12	Passed
Rabbit	00:00:06.54	Passed
FIPS-140-2	00:00:12.24	Passed

Figure 9: The result obtained using TestU01 (2.6 GHz Intel Core i7/clang v3.6.0). They were made for the algorithm presented in the "Appendix" section.

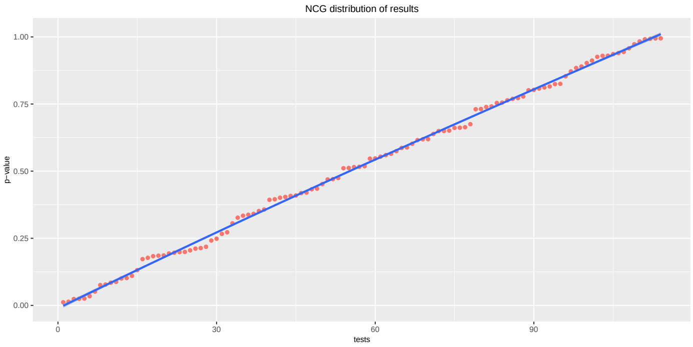

Figure 10: Normal probability plot (rankit), sorted p-values from statistical tests.

5. Parameters

The same parameters occur in the theoretical model of chaos machine. Additionally, in algorithm is located auxiliary function, it operates on the half of cells in the tape. Its task is to introduce cyclical fluctuations, in reference implementation

[8] This part was presented at the *June 5, 2015*.

[9] The algorithm is not thoroughly discussed. If you want to know precisely how it works, please go to the section "Appendix" where is source code in ANSI C.

[10] Although it is "designed to be cryptographically secure", no security proof is given, and only statistical tests argue for its security.

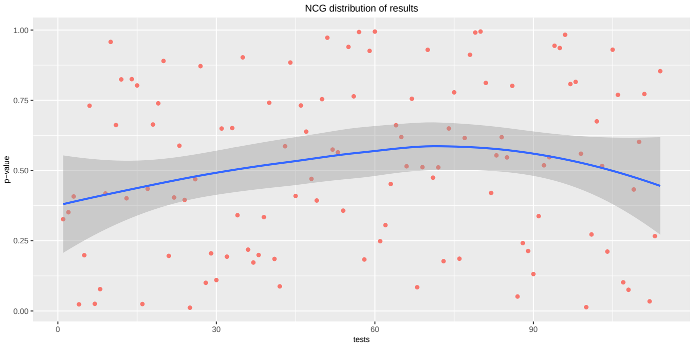

Figure 11: Results of the Dieharder test sets in chronological order. An average p-value is 0.49 (blue line).

it's LFSR[11] because its period is determined, but it can be changed to any other algorithm (table with variants).

Parameter	Value	Description
K	π digits	Initial secret key
t (TIME)	1	Time parameter
m (SPACE)	16	Space parameter

Figure 12: Above table of default parameters in algorithm. The requirement for m is that it must be an even number.

5.1. Time Parameter

This parameter determines the number of rounds of computation that machine performs. The larger the time parameter, the longer the output computation will take. We make the exact time-space trade-offs precise on the figure 13.

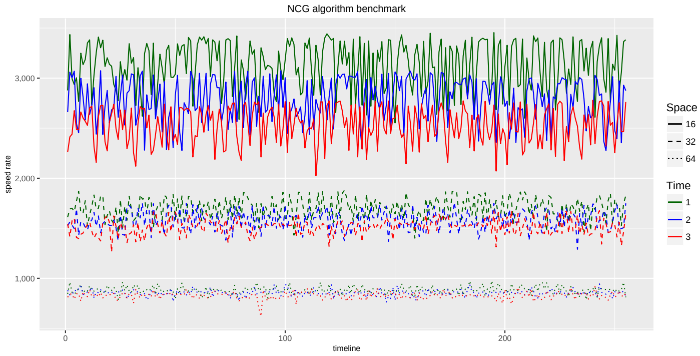

Figure 13: Plot that shows the impact of parameters to the speed rate. For each case, benchmark has performed 100,000 times the push and pull function with different input data. As we can see the algorithm slows 3 times for the 4 times bigger space parameter.

5.2. Space Parameter

In this machine, space parameter determines the length of the tape (number of cells). Let's assume that each cell has 16 bits capacity. Accordingly, half of the cells has a known[12] period 2^{16}, half unknown. It can be written as:

$$\prod^{m/2} 2^{16} \leq x \leq \prod^{m} 2^{16}$$

$$2^{8m} \leq x \leq 2^{16m}$$

$$2^{n/2} \leq x \leq 2^{n}$$

In the reference implementation we have m = 16, which gives the period 2^{128}, which can be up to 2^{256}. Hence, an implemented algorithm has a period between $2^{n/2}$ to 2^{n}. Theoretically, if we have performed at every $2^{n/2}$ push function with truly random value, it will be possible to produce infinite period length.

6. Application

Main aim of creating this algorithm was to present implemented model of chaos machine. As it was already mentioned before, the drawback is the limited quantity of the machine parameters. When the space parameter is huge, the algorithm is suitable only for a use as the PRNG.

At the beginning of the use of NCG, we must call the reset function. Initially, buffer space (entropy pool) is empty, but in accordance with the chaos machine model it must have an initial secret key. Code in "Appendix" is not thread safe.

6.1. Pseudo-random Number Generator

To implement generator we need to push seed into machine, and then pull out the pseudo-random numbers. Period is counted from the last execution of the push function, because there is no data from the outside (closed circuit). The same construction, input data and parameters will produce the same sequence of numbers, because the process is fully deterministic. However, with sufficient care, a system can be designed to produce cryptographically secure random numbers from the sources of randomness available in a modern computer. The basic design is to maintain an "entropy pool" of random bits that are assumed to be unknown to an attacker [ÖP14]. New randomness is added through push function to the buffer that evolves states of the machine.

6.2. Hash Function

The algorithm cannot compete with the speed of algorithms such as murmurhash3, farmhash or cityhash. However, the algorithm can be effectively used as a message authentication code where speed is not so important. Calculable period and excellent quality create narrow range of applications as hash function.

[11] A linear-feedback shift register (LFSR) is a shift register whose input bit is a linear function of its previous state. The taps are XOR-ed sequentially with the rightmost bit and then fed back into the leftmost bit.

[12] If a PRNG's internal state contains n bits, its period can be no longer than 2^n results, and may be much shorter. LFSR have period 2^n, but in this case cells have 16-bit, which gives period 2^{16}.

Appendix

6.3. Chaos Machine

```c
/*  NCG written in 2015 by Maciej A. Czyzewski

To the extent possible under law, the author has dedicated all copyright
and related and neighboring rights to this software to the public domain
worldwide. This software is distributed without any warranty.

See <http://creativecommons.org/publicdomain/zero/1.0/>.  */

#include <stdint.h>
#include <string.h>

// S - seed, I - increment, t - mask, i - temporary
uint32_t S, I, t, i;

// The number of rounds in algorithm (time parameter)
#define TIME 1

// The length of the initial states (space parameter)
#define SPACE 16

// Abbreviation for getting values from the tape
#define M(i) ((i) % SPACE)

// Bits rotation formula
#define R(x, y) (((x) << (y)) | ((x) >> (16 - (y))))

// Variables in the algorithm
uint16_t a, b, c, d, e;

// Initial secret key - pi digits (buffer space)
uint16_t G[SPACE], K[SPACE] = { 1, 4, 1, 5, 9, 2, 6, 5,
                                3, 5, 8, 9, 7, 9, 3, 2 };

void push(uint32_t seed) {
  // Preparation
  I = seed * 0x3C6EF35F;

  for (S = seed, i = 0; i < SPACE; i++) {
    // Reinforcement
    G[M(i)] ^= ((I * (S + 1)) ^ S) >> 16;
    G[M(i)] ^= ((I * (S - 1)) ^ S) >> 00;

    // Finalization
    I ^= ((G[M(I - 1)] + G[M(i)]) << 16)
      ^  ((G[M(I + 1)] - G[M(i)]) << 00);
  }
}

uint32_t pull(void) {
  // Variables
  a = G[M(I + 0)]; b = G[M(I + 1)];

  // Initialization
  t = (a + I) * (b - S);

  // Chaos
  e = (G[M(t - b)] << (a % 9)) ^ (G[M(t + a)] >> (b % 9));

  // Rounds
  for (i = 0; i < TIME * 2; i += 2) {
    // Absorption
    c ^= G[M(I + i - 2)]; d ^= G[M(I + i + 2)];

    // Mixing                 Modification
    c ^= (d ^= R(e, c % 17)); G[M(I + i - 2)] -= (d += (t & c));
    d += (c += R(t, d % 17)); G[M(I + i + 2)] += (c += (e & d));
  }

  // Transition
  G[M(I + 0)] = R(c, t % 17) ^ R(d, t % 17) ^ (t & a) ^ (e & b);
  G[M(I + 1)] = (b >> 1) ^ (-(b & 1u) & 0xB400u); // LFSR

  // Finalization
  t += (c ^ (b << 8) ^ (d << 16) ^ (a & 0xFF) ^ ((a >> 8) << 24));

  // Cleaning
  c = d = 0xFFFF;

  // Increment
  I += 2;

  return t;
}

void reset(void) {
  // Copying defaults
  memcpy(G, K, 2 * SPACE);
}
```

6.4. Pseudo-random Number Generator

6.4.1. Implementation

```
void ncg(const uint32_t seed) {
  // Cleaning tape
  reset();

  // Push to NCG structure
  push(seed);
}
```

6.4.2. Usage

```
#include <stdio.h>
#include <stdlib.h>

#include "../src/ncg.c"
#include "../src/include/random.c"

int main (int argc, char const *argv[])
{
  if (argc < 2) {
    printf("usage: %s <number> \n", argv[0]);
    return 1;
  }

  ncg((uint32_t) atoi(argv[1]));

  while (1) {
    putc_unlocked(pull(), stdout);
  }

  return 0;
}
```

6.5. Hash Function

6.5.1. Implementation

```
#define GET_32_INT(n, b, i)                           \
{                                                     \
  (n) = ( (unsigned long) (b)[(i)    ]       )        \
      | ( (unsigned long) (b)[(i) + 1] << 8  )        \
      | ( (unsigned long) (b)[(i) + 2] << 16 )        \
      | ( (unsigned long) (b)[(i) + 3] << 24 );        \
}

#define PUT_32_INT(n, b, i)                            \
{                                                      \
  (b)[(i)    ] = (unsigned char) ( (n)       );        \
  (b)[(i) + 1] = (unsigned char) ( (n) >> 8  );        \
  (b)[(i) + 2] = (unsigned char) ( (n) >> 16 );        \
  (b)[(i) + 3] = (unsigned char) ( (n) >> 24 );        \
}

void ncg(const uint8_t *initial_message, size_t initial_length,
               uint8_t *result, size_t result_length) {
  // Cleaning tape
  reset();

  // Declaration of variables
  size_t length, offset;

  // Declaration of message
  uint8_t *message = NULL, *buffer = NULL;

  // Declaration of message chunk
  uint32_t chunk;

  // Calculate new length
  for (length = initial_length;
       length % 4 != 0; length++);

  // Prepare message
  message = (uint8_t*) malloc(length * 8);

  // Copy block of memory
  memcpy(message, initial_message, initial_length);

  // Complement to the full blocks
  if (length - initial_length > 0) {
    // Append "1" bit
    message[initial_length] = 0x80;

    // Append "0" bits
    for (offset = initial_length + 1; offset < length; offset++)
      message[offset] = 0;
  }

  // Append the len in bits at the end of the buffer
  PUT_32_INT(initial_length * 8, message + length, 0);

  // Initial_len >> 29 == initial_len * 8 >> 32, but avoids overflow
  PUT_32_INT(initial_length >> 29, message + length + 4, 0);

  // Process the message in successive 32-bit chunks
  for (int i = 0; i < length; i += 4) {
    // Get little endian
    GET_32_INT(chunk, message + i, 0);

    // Push to NCG structure
    push(chunk);
  }

  // Releasing memory
  free(message);

  // Allocate memory for result
  buffer = (uint8_t*) malloc(result_length * 8);

  // Process the result in successive 32-bit chunks
  for (int i = 0; i < result_length / 4 + 1; i++)
    PUT_32_INT(pull(), buffer, i * 4);

  // Save it on the pointer
  for (int i = 0; i < result_length; i++)
    result[i] = buffer[i];
```

```c

  // Releasing memory
  free(buffer);
}
```

6.5.2. Usage

```c
#include <stdio.h>
#include <stdlib.h>

#include "../src/ncg.c"
#include "../src/include/hash.c"

int main (int argc, char const *argv[])
{
  const char *initial_message = argv[1];
  size_t initial_length = strlen(initial_message);

  if (argc < 3) {
    printf("usage: %s 'string' <number> \n", argv[0]);
    return 1;
  }

  size_t result_length = atoi(argv[2]);
  uint8_t *result = (uint8_t*) malloc(result_length * 8);

  printf(">> ");
  for (int i = 0; i < initial_length; i++)
    printf("%p ", initial_message[i]);
  printf("\n");

  ncg((uint8_t*) initial_message, initial_length, result, result_length);

  printf("<< ");
  for (int i = 0; i < result_length; i++)
    printf("%p ", result[i]);
  printf("\n");

  return 0;
}
```

References

[Knu73] Donald E. Knuth. *Seminumerical Algorithms*. Second. Vol. 2. The Art of Computer Programming. 1973. Chap. 3.

[Dev84] Robert L. Devaney. “A Piecewise Linear Model for the Zones of Instability of an Area Preserving Map.” In: *Physica*. Vol. 10. 1984, pp. 387–393.

[KT01] Kunihiko Kaneko and Ichiro Tsuda. *Complex systems : chaos and beyond a constructive approach with applications in life sciences*. Physics and astronomy online library. Translated from the Japanese. Berlin, London: Springer, 2001. ISBN: 3-540-67202-8. URL: `http://opac.inria.fr/record=b1101628`.

[Via01] Marcelo Viana. “Dynamical Systems: Moving into the Next Century”. In: *Mathematics Unlimited: 2001 and Beyond*. Springer, 2001, pp. 116–7.

[Har06] D. Harrivel. “Butcher series and control theory”. In: *ArXiv Mathematics e-prints* (Mar. 2006). eprint: `math/0603133`.

[Wer13] C. Werndl. “Are Deterministic Descriptions And Indeterministic Descriptions Observationally Equivalent?” In: *ArXiv e-prints* (Oct. 2013). arXiv: `1310.1615 [math.DS]`.

[ÖP14] B. Ömer and C. Pacher. “Saving fractional bits: A practical entropy efficient code for fair die rolls”. In: *ArXiv e-prints* (Dec. 2014). arXiv: `1412.7407 [cs.IT]`.

[TLB14] C. Thomson, L. Lue, and M. N. Bannerman. “Mapping continuous potentials to discrete forms”. In: 140.3, 034105 (Jan. 2014), p. 034105. DOI: `10.1063/1.4861669`. arXiv: `1309.7292 [cond-mat.soft]`.