

SHA-RNN

Recurrent Neural Network with Chaotic System for Hash Functions

Anonymous Authors

[摘要] 在这次作业中我们提出了一种新的 Hash Function —— SHA-RNN。其以海绵结构为基础，融合了混沌系统与循环神经网络的特点，可以接受任意长度的串作为输入，并生成长度为 80 比特的消息摘要。我们会对我们的架构进行详细表述，并给出在时间及安全性上的自评估结果。最后，我们会提供完整的实验框架与使用说明。

GitHub Repo 地址: <https://github.com/Ashitemaru/Sharnn>

1 Introduction

哈希函数 (Hash Functions) 是一种将任意长度的消息生成固定长度摘要的映射，其在消息认证、数字签名、身份验证等方面具有十分重要的应用。一般地，我们要求哈希函数具有有效性与安全性，前者是说哈希函数的运作效率要足够快，而后者则对哈希函数的抗原象攻击、抗第二原象攻击等方面具有一定要求。

混沌系统 (Chaotic System) 的概念最先在物理学中的运动学领域被提出，它是一种非线性系统，其中的值存在着貌似随机的不规则运动，其行为表现为不确定性、不可预测。有部分可以迭代计算的函数可以作为混沌系统的生成函数，比如帐篷函数 (Tent Map)。由于混沌系统具有的给定初值输入后，其后生成序列值不可预测，或者说近似服从均匀分布的特点，其可以被用来作为 Hash 函数的重要组成部分。[1]

循环神经网络 (Recurrent Neural Network) 是一种以序列数据为输入，在序列时序信息的维度上迭代，所有循环单元按链式进行连接的一种递归神经网络。循环神经网络的优势之一，在于其能够在某种程度上“记忆”来自上一个时间状态的信息。如果不同消息分组输入循环神经网络的顺序有先后之别，他们对于同一个“神经元”的作用也有不同的区别。而这种区别，往往是难以用计算追溯到的，也是其相对于简单的全连接层的优势所在。

综合以上三个概念，我们提出了 **SHA-RNN**，一种新的基于密钥的哈希函数。其采用海绵函数作为基本架构，在每次迭代中使用混沌系统产生近似均匀分布的值作为循环神经网络的权重和偏差。

本篇报告将分为以下几部分。在第 2 节，我们会对我们设计哈希函数时所运用到的关键概念进行简要的回放与讲解。在第 3 节，我们会给出我们所设计的哈希函数的具体参数与详细运作流程。在第 4 节，我们会对我们所设计的哈希函数进行分析，给出在时间效率上和安全性检测上的自评估结果。在附录中，我们会对我们编写的实验框架进行详细的介绍，包括代码介绍与使用方法介绍。

2 Related Works

2.1 Sponge Function

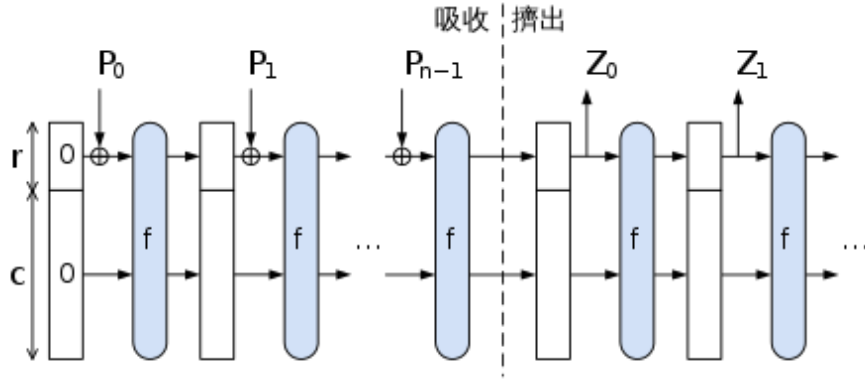


Figure 2.1.1: 海绵函数的示意图

Credit: Wikipedia

海绵函数(Sponge Function) [2] 是一种算法，它可以接受任意长度的输入比特流，得到任意长度的输出。它的参数为单次输出比特长度 r ，隐藏状态长度 $r + c$ ，状态转移函数 f 。其工作流程可以分为 **吸收(Absorb)** 与 **挤出(Squeeze)** 两个阶段，其工作流程如图 2.1.1 所示，描述如下。

在吸收阶段，海绵函数单次接受长度为 r 的消息分组，与上个阶段的隐藏状态 r 比特进行异或，将异或后的隐藏状态经过状态转移函数 f 得到新的隐藏状态。在挤出阶段，我们每次从隐藏状态中提取 r 比特，然后将隐藏状态再经过状态转移函数 f ，如此往复，直到我们得到足够长的消息摘要为止。

2.2 Chaotic Neural Network

引用文献 [2] 提出了两种基于密钥的哈希函数，其采用了海绵函数架构，使用了 **混沌神经网络(Chaotic Neural Network)** 作为海绵函数的状态转移函数 f 。其基本运作模式如图 2.2.1 所示，接下来我们对其运作阶段及状态转移函数 f 进行简单介绍。

首先我们来对其运作的三个阶段进行简单介绍。(1) 在初始阶段，我们假设输入密钥为 K ，将隐藏状态 $HM_0 \leftarrow 0^{r+c}$ ，密钥 $KM_0 \leftarrow K$ ，并对输入做适当填充后分组。(2) 在第 q 个吸收阶段，将本阶段输入消息分组 M_q 与隐藏状态 HM_{q-1} 的前 r 个比特做异或之后，我们将隐藏状态 h_{q-1} 与混沌发生参数 KM_{q-1} 输入混沌神经网络系统之中（包括一个混沌神经网络和一个混沌系统，我们会在后续进行介绍）。(3) 在第 q 个挤出阶段，我们输出隐藏状态 h_{q-1} 的前 r 比特输出，作为当前阶段的消息摘要输出，然后将隐藏状态 h_{q-1} 与上一阶段的生成密钥 KM_{q-1} 输入混沌神经网络中，得到下一阶段的隐藏状态 HM_q 。

这里的 **混沌神经网络** 作为海绵函数 f 的状态转移函数，承担着接受密钥，将隐藏状态混淆的同时使得输出结果尽可能均匀的作用。**混沌系统(Chaotic System)** 接受上个阶段的输出密钥 KM_{q-1} 作为发生参数，产生均匀的输出作为后续 **混沌神经网络(Chaotic Neural Network, CNN)** 各层的权重 (Weights) 与偏差 (Biases)。这里的混沌神经网络采用了简单的全连接层的机制。

2.3 Discrete Skew Tent Map

我们这里首先引入 Skew Tent Map [4] 的概念，它被广泛地应用于混沌系统的生成当中。Skew Tent Map 是一种迭代函数，它接受上一状态输入 $X(n)$ ，将其映射为：

$$X(n+1) = \begin{cases} \frac{\mu}{Q}X(n), & X(n) \leq Q, \\ -\frac{\mu}{1-Q}(X(n) - Q) + \mu, & X(n) > Q. \end{cases}$$

对于大部分输入，通过足够多的迭代步，可以保证这个函数生成的序列是混沌的。而我们这里介绍其离散化的版本，即对于输入 $1 \leq X(n-1) \leq 2^n - 1$ ：

$$\text{DSTMap}_Q X(n+1) = \begin{cases} 2^N \times \frac{X(n)}{Q} & \text{if } 0 < X(n-1) < Q \\ 2^N - 1 & \text{if } X(n-1) = Q \\ 2^N \times \frac{2^N - X(n)}{2^N - Q} & \text{if } Q < X(n-1) < 2^N \end{cases}$$

我们将在混沌系统中引入这个函数以保证生成序列的随机性。

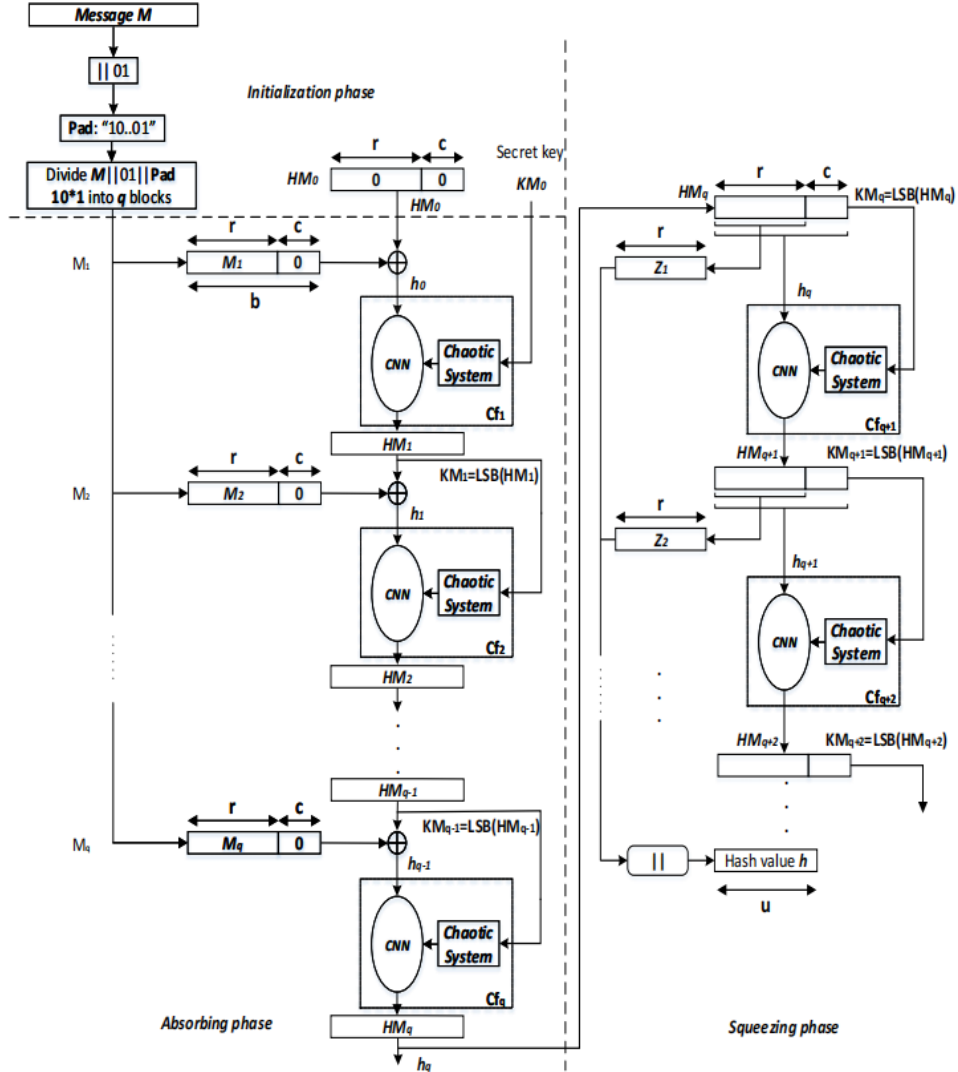


Figure 2.2.1 Keyed-Sponge CNN hash functions

Credit: Citation [3], Figure 3.

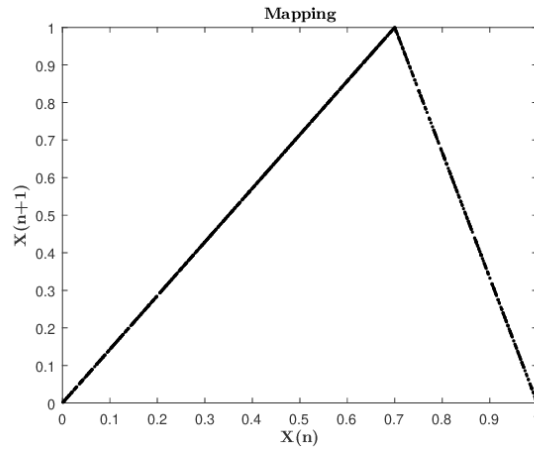


Figure 2.3.1 Skew Tent Mapping

Credit: Research Gate

3 SHA-RNN

我们所提出的哈希函数架构采用海绵函数作为基础，采用的基本架构与图片 2.2.1 类似。不过，我们会详细介绍我们针对其中的参数配置。

3.1 Parameters

在这里我们定义我们海绵函数以及其他部件所使用的常数。

- RNN 层属性
 - 单次输入长度 r : 136 字节
 - 隐藏状态大小 $r + c$: $136 + 64$ 字节
 - 单次输出大小: 32 字节
- 哈希函数属性
 - 哈希函数输出长度: 80 比特（对单次输出做高位截断）
- 用户可以设置的参数，以及我们的预置配置，用户可以通过修改源码重新编译得到新的配置
 - 混沌系统偏移量 Q : 0x789ABCDE
 - 输入密钥: K_m
 - 混沌系统生成种子 K_s : 0x10
 - 混沌系统构建迭代次数 U_s : 10
 - 非线性层迭代次数 N_r : 8

3.2 Initialization

在初始化阶段，我们对原始消息进行填充。对于一条原始消息 M ，我们首先在其后填充比特串 01 得到 $M' = M || 01$ 。然后，我们计算 M' 与最近的消息分组倍数长度的距离，然后对其填充比特串 10^*1 ，这里 a^* 表示接受类似于集合 $\{a^i | i \geq 0\}$ 的比特串。

然后，我们将填充后的消息 $M'' = M' || 10^*1$ 进行分组，使得 $M_1 || M_2 || \dots || M_q = M''$ ，且每个分组 M_i 的长度为 r 比特，其中 $1 \leq i \leq q$ 。

3.3 Chaotic Neural System

3.3.1 Chaotic System

3.3.1.1 DSTMap

可以将一个混沌系统看成一个混沌状态发生器，或者将其视为是一个满足如下条件的映射：

$$\text{DSTMap}_Q X(n+1) = \begin{cases} 2^N \cdot \frac{X(n)}{Q} & 0 < X(n-1) < Q \\ 2^N - 1 & X(n-1) = Q \\ 2^N \cdot \frac{2^N - X(n)}{2^N - Q} & Q < X(n-1) < 2^N \end{cases}$$

对于给定的 32 比特数 N 与上一状态 $X(n)$ ，我们通过上述函数将其转化为 $X(n+1)$ ，该映射满足点列 $\{X(i)\}_{i=0}^{\infty}$ 近似均匀随机。

此外，我们还可以定义类似的运算：

$$\text{DPWLMap}_Q(X(n)) = \begin{cases} 2^N \cdot \frac{X(n-1)}{Q} & 0 < X(n-1) \leq Q \\ 2^N \cdot \frac{X(n-1) - Q}{2^{N-1} - Q} & Q < X(n-1) \leq 2^{N-1} \\ 2^N \cdot \frac{2^N - X(n-1) - Q}{2^{N-1} - Q} & 2^{N-1} < X(n-1) \leq 2^N - Q \\ 2^N \cdot \frac{2^N - X(n-1)}{Q} & 2^N - Q < X(n-1) \leq 2^N - 1 \\ 2^N - 1 - Q & \text{otherwise;} \end{cases}$$

这个函数将被用于我们非线性的激活函数。

3.3.1.2 Chaotic System

系统初始化时，需要给定参数 Q, K_s, K_0, U_s 。然后，我们将 K_0 作为初始条件 $CS \leftarrow K_0$ ，进行 U_s 次如下迭代 $CS \leftarrow \text{DSTMap}_Q(CS \times K_s)$ 得出结果作为 $X(0)$ 。之后的每次迭代均通过 $X(i+1) \leftarrow \text{DSTMap}_Q(X(i))$ 并返回 $X(i+1)$ 来实现。

3.3.2 Chaotic Neural Network

我们的 Chaotic Neural Network 采用 RNN 架构。下面对其进行介绍。每个 RNN 架构配有一个混沌状态发生器 CS ，我们下面用记号 $X \sim CS$ 表示进行对混沌状态映射函数进行一次迭代后得到的新的结果，或者说对混沌状态发生器做一次采样后得到的结果 X 。

由于我们指定了 RNN 架构的输入为 $b = 200$ 字节 = 1600 比特，所以我们将输入的 1600 比特切分成 10 份，分别记为 I_1, I_2, \dots, I_{10} ，每份 160 比特，记 $I_i = J_{i1}J_{i2}J_{i3}J_{i4}J_{i5}$ ，每个 J_{ij} 为一个小消息分组，长度为 32 比特，视为无符号整数参与运算。

我们定义两种神经元，一种是 RNN Cell，另一种是 Non-linear Cell。RNN Cell 共有 10 个，每个可以接受 160 比特的消息输入和 32 比特的上一状态输入，输出为 32 个比特的当前状态。Non-linear Cell 的输入为 320 比特，输出为 1600 比特。

接下来我们描述一次 RNN 架构的运作流程。

首先设 $\text{prev} \leftarrow 0$ 作为 RNN 的隐藏状态，然后对于 10 份消息分组，我们依次将 I_i 与 prev 输入 RNN Cell，得到 160 比特的消息输出 O_i 与新的状态 prev 。最后，我们将 10 份消息的输出 $\{O_i\}_{i=1}^{10}$ 连接起来，就得到了 RNN Cell Layer 的输出 320 比特。

然后将 RNN Cell Layer 输出的 320 比特经过非线性激活层，便得到这次 RNN 层海绵函数状态转移的终态的 1600 比特。

3.3.3 RNN Cell

每一个 RNN 神经元接受 160 个比特的串作为输入，这里将其记为 $I_i = J_{i1}J_{i2}J_{i3}J_{i4}J_{i5}$ ，其中 J_{ij} 是一个 32 比特的串。我们令 $Q_1 \sim CS, Q_2 \sim CS$ ，然后令 $W_i \sim CS, 1 \leq i \leq 5$ ，定义 32 位乘法为将乘法结果的高 32 位与低 32 位进行异或运算：

$$\begin{aligned} F_1 &\leftarrow J_{i1} \times W_1 + J_{i2} \times W_2 + J_{i3} \times W_3 + \text{prev} \\ F_2 &\leftarrow J_{i4} \times W_4 + J_{i5} \times W_5 \end{aligned}$$

接着，分别迭代 20 次 $F_1 \leftarrow \phi_{Q_2}(F_1)$ ， $F_2 \leftarrow \rho_{Q_1}(F_2)$ ，这里 $\phi = \text{DSTMap}$ ， $\rho = \text{DPWLMap}$ ，最后我们返回 $F_1 \oplus F_2$ ，这里返回长度为 32 比特。

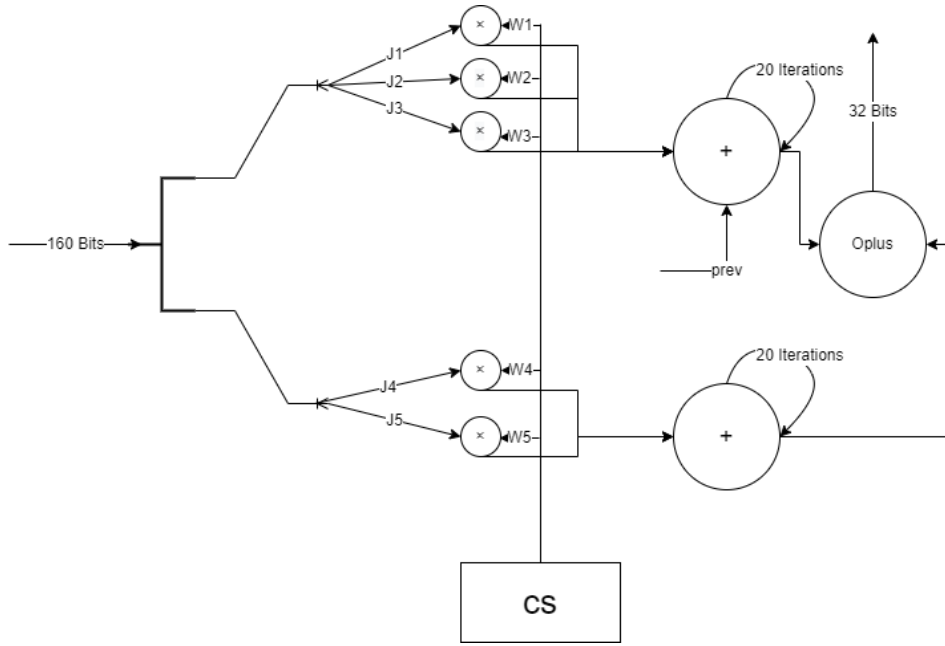


Figure 3.2.3.1: RNN Cell 计算流程

3.3.4 Non-linear Cell

非线性层负责将 320 比特的输出升采样成 1600 比特。其包含的 Cell，我们将其记为 Cell_{5-8} 。

我们首先定义一些基本运算，这里借鉴了参考资料 [2]：

$$\begin{cases} \text{Maj}(D_1, D_2, D_3) = (D_1 \wedge D_2) \oplus (D_1 \wedge D_3) \oplus (D_2 \wedge D_3) \\ \text{Ch}(D_1, D_2, D_3) = (D_1 \wedge D_2) \oplus (\neg D_1 \wedge D_3) \\ \Sigma 0(D_1) = \text{ROTR}^2(D_1) \oplus \text{ROTR}^{13}(D_1) \oplus \text{ROTR}^{22}(D_1) \\ \Sigma 1(D_3) = \text{ROTR}^6(D_3) \oplus \text{ROTR}^{11}(D_3) \oplus \text{ROTR}^{25}(D_3) \\ \text{ROTR}^n(x) = (x \gg n) \vee (x \ll (32 - n)) \end{cases}$$

Cell₅₋₈ 负责完成维度上的升采样过程，其接受 160 比特的串作为输入，输出为 256 比特，其计算方式为：

$$\begin{cases} H_0 = D_0 \oplus t_1 \oplus \text{Maj}(D_1, D_2, D_3) \oplus \Sigma 0(D_1) \\ H_1 = t_1 \oplus D_0 \\ H_2 = D_0 \oplus D_1, H_3 = D_1 \oplus D_2, H_4 = D_2 \oplus D_3 \\ H_5 = D_0 \oplus D_1 \oplus t_1 \\ H_6 = D_1 \oplus D_2 \oplus t_1 \\ H_7 = D_2 \oplus D_3 \oplus t_1 \\ \text{where } t_1 = \text{Ch}(D_1, D_2, D_3) \oplus D_4 \oplus \Sigma 1(D_3) \end{cases}$$

非线性激活层首先定义 $h_i = J_i \oplus J_{i+5}, 1 \leq i \leq 5$ ，其中 $J_i = W_i \times I_i$ ， $W_i \sim CS$ ， I_i 是将输入的 320 比特等分成 10 份中的结果，长度为 32 比特，这里 $1 \leq i \leq 10$ 。32 位乘法定义为两数相乘结果中高 32 位与低 32 位的异或运算。

然后，对这个结果的前 5 个 32 比特迭代重复 N_r 次 Cell₅₋₈，

对输入的每个 160 比特经过 Cell₅₋₈ 升采样为 256 比特，然后将结果经过 N_r 次迭代后，作为结果输出后级联得到 512 个比特。然后再将该结果经过 Cell₈₋₈ 每迭代一次，生成 512 比特后级联，直到结果超过 1600 比特后取前 1600 比特。之后，每次从结果中提取 64 比特，然后再进行迭代，直到满足输出条件长度后截断。

4 Evalutaion

4.1 Efficiency

我们对于大小分别为 1 MB, 2 MB, 4 MB, 8 MB, \dots , 1 GB 的数据做了效率检验（详见单元测试的 test3 程序），得出的结果如下：

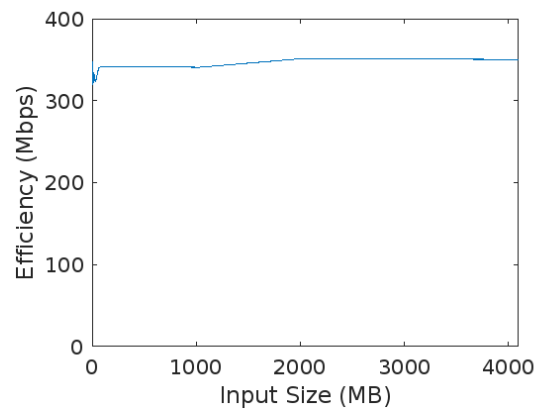


Figure 4.1.1 效率测试结果

我们的实验环境为：

- 处理器 Intel64 Family 6 Model 141 Stepping 1 GenuineIntel ~2688 Mhz (Intel-i5)
- 内存：16 GB
- 操作系统：Windows WSL2 (Ubuntu 20.04)

4.2 Security

4.2.1 Random Number Testing

我们使用NIST提供的 [Statistical Test Suite](#) 对哈希结果的随机性进行检测。

首先生成哈希序列。在 $[0, 2^{31} - 1)$ 中从零开始均匀选取 2^{21} 个数字，将数字对应的 32 位整数作为输入进行哈希，得到 20MB 输出。使用 NIST Statistical Test Suite 基于多种度量进行随机性测试。默认参数选择 $n = 1,500,000$ 可符合大部分测试的要求，按文档 [6] 要求使用其他参数的，在下表最后一列注明。

编号	测试类型	通过率	p 值均匀性 ³	非默认参数
01	Frequency	111/111	0.580520	-
02	Block Frequency	20758/20971	0.273558	$n = 8000, M = 80$
03	Cumulative Sums	$2/2^1$	通过 ¹	-
04	Runs	111/111	0.263452	-
05	Longest Run of Ones	109/111	0.656043	-
06	Rank	109/111	0.328861	-
07	Discrete Fourier Transform	111/111	0.674920	-
08	Nonperiodic Template Matchings	N/A ⁴	N/A ⁴	N/A ⁴
09	Overlapping Template Matchings	111/111	0.003401	$m = 10$
10	Universal Statistical	54/55	0.719747	$n = 3,000,000$, 此时 $L = 8$
11	Approximate Entropy	110/111	0.000086 ²	$m = \lfloor \log_2 n \rfloor - 6$
12	Random Excursions	$8/8^1$	通过 ¹	-
13	Random Excursions Variant	$18/18^1$	通过 ¹	-
14	Serial	$2/2^1$	通过 ¹	$m = \lfloor \log_2 n \rfloor - 3$
15	Linear Complexity	110/111	0.818179	-

注:

1. 表示该检测度量有多个指标
2. 表示该指标的 p 值过小，表示 p 值分布不提示均匀分布
3. p 值大于 10^{-4} 表示通过
4. 由于本次作业要求哈希结果的位数过少，本测试我们未能找到合适的测试参数

4.2.2 Diffusion Test

为了度量哈希的扩散效果，我们对 SHA-RNN 算法进行了扩散测试

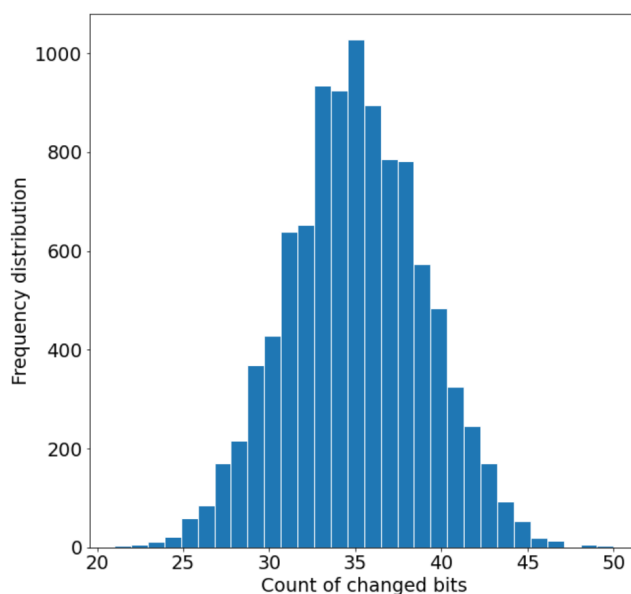
- 输入串 M ，计算哈希值 H_1 ，对于 M 我们选取 Wikipedia 中对海绵结构的介绍

In cryptography, a sponge function or sponge construction is any of a class of algorithms with finite internal state that take an input bit stream of any length and produce an output bit stream of any desired length. Sponge functions have both theoretical and practical uses. They can be used to model or implement many cryptographic primitives, including cryptographic hashes, message authentication codes, mask generation functions, stream ciphers, pseudo-random number generators, and authenticated encryption.

Sponge functions have both theoretical and practical uses. In theoretical cryptanalysis, a random sponge function is a sponge construction where f is a random permutation or transformation, as appropriate. Random sponge functions capture more of the practical limitations of cryptographic primitives than does the widely used random oracle model, in particular the finite internal state.

- 对串 M 随机选取一位进行翻转，然后再计算哈希值 H'
- 计算 H 和 H' 中不同有位数的数量 B_i

将上述过程在 SHA-RNN 算法中重复 $N = 10,000$ 次，结果如下图所示



定量计算

1. 改变位数的最小值 $B_{\min} = \min(\{B_i\}_{i=1, \dots, N})$
2. 改变位数的最大值 $B_{\max} = \max(\{B_i\}_{i=1, \dots, N})$
3. 改变位数的标准差 $\Delta B = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (B_i - \bar{B})^2}$
4. 改变位数的平均值 $\bar{B} = \frac{1}{N} \sum_{i=1}^N B_i$
5. 每位的改变概率 $P = \left(\frac{\bar{B}}{80}\right) \times 100\%$

最小值	最大值	标准差	平均值	每位变化概率
21	50	4.14	35.00	43.75%

4.2.3 DSTMap

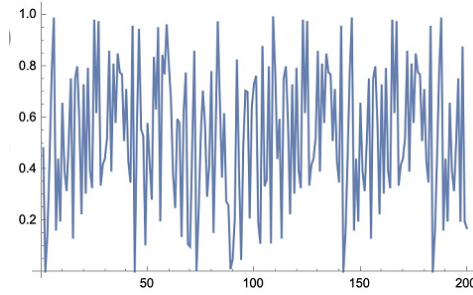


Figure 4.2.3.1 DSTMap Result

我们混沌系统采用的发生函数 DSTMap_Q 可以很好地起到生成均匀分布的数据的目的。我们这里对 STMap 做了实验，取 $\mu = 1, Q = 0.3$ ，然后对输出函数 $\text{STMap}_{0.3}$ 做傅里叶变换后得到上图，可以看出数据具有较强的随机性，满足均匀分布的条件。

5 References

- [1] LIU J, FU X. Spatiotemporal chaotic one-way hash function construction based on coupled tent maps[J]. Journal on communications, 2007, 28(6): 34.
- [2] Bertoni G, Daemen J, Peeters M, et al. Sponge functions[C]//ECRYPT hash workshop. 2007, 2007(9).
- [3] Abdoun N, El Assad S, Manh Hoang T, et al. Designing two secure keyed hash functions based on sponge construction and the chaotic neural network[J]. Entropy, 2020, 22(9): 1012.
- [4] Hasler M, Maistrenko Y L. An introduction to the synchronization of chaotic systems: coupled skew tent maps[J]. IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, 1997, 44(10): 856-866.
- [5] Penard W, van Werkhoven T. On the secure hash algorithm family[J]. Cryptography in context, 2008: 1-18.
- [6] Smid E B, Leigh S, Levenson M, et al. A statistical test suite for random and pseudorandom number generators for cryptographic applications[J]. Her research interest includes Computer security, secure operating systems, Access control, Distributed systems, Intrusion detection systems, 2010.

6 Appendix: Introduction to Codebase

实验框架使用方法介绍如下：

```

1  # Build
2  ./run.sh
3
4  # Usage
5  bin/SHA-RNN # (1) Input Mode
6  bin/SHA-RNN -f <file_path> # (2) Hash file
7  bin/SHA-RNN -s <string> # (3) Hash string

```

实验框架介绍如下:

```

1  .
2  └─ CMakeLists.txt
3  └─ bin
4  │   └─ SHA-RNN # 主程序
5  │   └─ test1 # Basic Usage Test
6  │   └─ test2 # Consistency Test
7  │   └─ test3 # Perf Test
8  │   └─ test4 # Diffusion Test
9  └─ include
10 └─ define.h
11 └─ main.cpp
12 └─ readme.md
13 └─ readme.pdf
14 └─ run.sh
15 └─ src
16 │   └─ Bitset.hpp # 比特流
17 │   └─ ChaoticMap.hpp # DSTMap / DPWLMap
18 │   └─ ChaoticSystem.hpp # 混沌系统
19 │   └─ NonLinear.hpp # 非线性激活层
20 │   └─ PaddedStream.hpp # 对输入串填充
21 │   └─ RNN.hpp # RNN 架构
22 │   └─ RNNHash.hpp # 派生自 SpongeHash, 负责调用 RNN 架构的逻辑
23 │   └─ SpongeHash.hpp # 海绵结构
24 └─ statistics
25 └─ Makefile
26 └─ analyze.ipynb # 扩散数据分析
27 └─ diffusion.cpp # 扩散数据生成
28 └─ eval.py # NIST 随机性测试脚本
29 └─ sample.cpp # 随机性测试采样
30 └─ text.in # 扩散测试样本数据
31 └─ test
32     └─ test1.cpp # Basic Usage Test

```

```
33 | └─ test2.cpp # Consistency Test
34 | └─ test3.cpp # Perf Test
35 | └─ test4.cpp # Diffusion Test
```