Performance of Computing Hash-Codes with Chaotically-Trained Artificial Neural Networks

Jacek Tchórzewski[1](✉) and Aleksander Byrski[2]

[1] Cracow University of Technology, Kraków, Poland
jacek.tchorzewski@pk.edu.pl
[2] AGH University of Science and Technology, Kraków, Poland
olekb@agh.edu.pl

Abstract. The main goal of the research presented in this paper was to estimate the performance of applying neural networks trained with the usage of a chaotic model, that may serve as hashing functions. The Lorenz Attractor chaotic model was used for training data preparation, and Scaled Conjugate Gradient was used as a training algorithm. Networks consisted of two layers: a hidden layer with sigmoid neurons and an output layer with linear neurons. The method of bonding the input message with chaotic formula is presented. Created networks could return 256 or 512 bits of hash, however, this parameter can be easily adjusted before the training process. The performance analysis of networks is discussed (that is the time of hash computation) in comparison with popular standards SHA-256 and SHA-512 under the MATLAB environment. Further research may include analysis of networks' training parameters (like mean squared error or gradient) or analysis of results of the statistical tests performed on networks output. The presented solution may be used as a security algorithm complementary to a certificated one (for example for additional data integrity checking).

Keywords: Hashing algorithm · Artificial Neural Networks · Scalable cryptography algorithm · Hashing efficiency

1 Introduction

Hashing functions return a fixed-length bit string from an input bit string, [22]. This functionality may be utilized, for example, in passwords storage, data integrity checking, or digital signatures preparation. Three main features of hashing algorithms are:

1. the process of hash calculation may involve more than one usage of the hashing algorithm,

Supported by the funds assigned by The Polish Ministry of Education and Science to Cracow University of Technology (J.T.) and to AGH University of Science and Technology (A.B.).

2. it should be impossible to retrieve the content of the original message from its hash,
3. probability of returning the same hash value from two different messages should be minimal.

Currently, the most popular hashing standards, which are also certificated, are called SHA-2 and SHA-3 [14,15]. Those functions compute a fixed-length hash from a message. Available hashes lengths vary from 224 bits to 512 bits and the length determines the algorithm – e.g. SHA-256 will always return 256 bits of the hash. Certain certificated standards offer possibilities of choosing two different hashes lengths, for example, a hash equal to 224 bits is created from the truncation of the SHA-256 digest. However, the user has no more possibilities to adjust the hash length. It can be concluded that it is either one fixed length that depends on the algorithm or two possible lengths where the second one comes from the truncation of the hash of size equal to the first one. Hashing algorithms whose outputs are smaller, for example, vary from 80 bits to 160 bits, are called light cryptography hash functions [5].

Since late 1970, researchers proposed many different approaches to hash function construction. Most of the ideas are described in [19], for example, hashing function based on the block ciphers, cellular automata, discrete logarithm problem, or knapsack problem. Testing the strength and effectiveness of this kind of algorithms is still a problem. Existing test suits, like for example SHAVS presented in [13], are dedicated to the tested algorithm. Secondly, the National Institute of Standards and Technology (NIST) in the presented report state that [13]: '*The SHAVS is designed to test conformance to SHA rather than provide a measure of a product's security...*'.

In this paper, an idea of hashing neural networks proposed in [19–21] is further developed, showing the performance of the ANNs used. The discussed networks have two layers: a hidden layer with sigmoid neurons, and an output layer consisting of linear neurons. Two hash lengths were considered, that is 256 and 512 bits (to compare results with certificated standards). For each tested hash length seven networks were generated. Generated networks differed in the number of neurons in the hidden layer. The Lorenz Attractor, which appears to have chaotic behavior under appropriate conditions, was utilized for training data preparation. The length of the returned hash could be potentially set with the precision of one bit before the training process. Furthermore, the performance of the proposed networks is tested in comparison with the performance of the chosen certificated standards (SHA-256 and SHA-512) under the MATLAB environment. The time of hash computation from data that differed in size was considered as the performance measure.

The paper after introducing the idea of the approach and giving state-of-the-art, focuses on the presentation of the performance testing of the ANN-based hashing functions, comparing the obtained models with certificated ones.

2 Related Work

In this section hashing algorithms that are based on chaotic systems are described. In each article not only the chaotic model was considered, but also the core of the hashing algorithm.

In [11] authors proposed their own algorithm, which was based on the Lorenz Attractor. However, they incorporated some functions from the SHA-2 algorithm, for example rotations. In their research, similarly to the presented research, time of computation was considered as a performance measure. The algorithm core consisted of four iterations that were combining intermediate hash results and secret keys. The final results were compared with SHA-1. Even though the proposed algorithm was more efficient, SHA-1 is considered as an outdated function, and there were no comparisons with current standards, like SHA-2 or SHA-3.

The algorithm presented in [10] was not a classical hashing scheme but enabled checking the integrity of the data. The proposed procedure was based on huge numbers and their powers under the finite field, which makes the whole idea similar to the RSA ciphering scheme. The authors compared the efficiency of their solution with the Advanced Encryption Standard (AES) and concluded that the performance of their algorithm was slightly worse.

In [9] authors incorporated similar operations as described in the [10], a sponge function that was absorbing input data, and a hyper-chaotic Lorenz system. Because of the complexity of the algorithm, the authors noticed perturbations over time in the function performance. The solution was tested for 256 bit, and 512 bit hash lengths, but enabled returning 1024 bits of hash and more. Authors compared their proposition with SHA-2 and SHA-3 standards, however, did not test it for smaller hashes values.

The innovative idea was presented in [1], where the authors proposed their own equation for input data absorption. Their solution was based on the three-dimensional chaotic map and was excessively tested. The proposed function could return 128, 160, 256, or 512 bits of the hash. Results of the research were compared with SHA-1 and MD5. Both are considered as outdated.

Hashing algorithms may be created in many different ways. For example, in [8] some interesting hashing concepts that are based on evolutionary algorithms and genetic algorithms are presented. More information about hashing strategies is presented in the [17].

Chaotic attractors may be also used in different cryptographic areas. For example, in [6] authors proposed an image encryption scheme based on the Lorenz model. The core of the algorithm was utilizing crossover operations and sequences generated by the attractor. Even though the presented scheme was not a hashing function, research conducted by the authors proved the usefulness of chaotic systems in cryptographic solutions.

In contrast to described solutions, an idea presented in the paper enables the utilization of Artificial Neural Networks (ANNs) trained with the usage of the Lorenz Attractor as hashing models. The main advantage of the proposed scheme is a highly scalable ANNs output. The length of the hash returned by those networks can be adjusted with a precision of one bit (before the training process). Furthermore, the performance of ANNs was tested and results were compared with one of the most popular – and certified by the National Institute of Standards and Technology – Secure Hash Standards, that is SHA-256 and SHA-512. Efficiency comparison performed under MATLAB environment tends to be in favor of the presented networks. Further research will cover the security tests of networks and will also include a comparison with world standards.

3 The Chaotic Model Used

Chaotic equations are non-linear and dynamic systems (models), that are significantly vulnerable to the changes in their initial conditions [4]. The output of such models becomes non-deterministic over time. This phenomenon is also called deterministic chaos. One of the most iconic scientist that was investigating this topic was Edward Lorenz. He once described chaos as a situation [4]: *when the present determines the future, but the approximate present does not approximately determine the future.*

In our work the Lorenz Attractor was used for ANNs training data preparation. Two sets of data were prepared: input data containing binary strings representing messages to be hashed, and output data that contained hashes of those messages obtained with the usage of Lorenz Attractor. The idea was to code a message into attractors' initial conditions and then solve the model. The result was considered as a message hash.

The Lorenz Attractor is defined as presented in Eq. (1):

$$\begin{cases} \frac{dx_1}{dt} = a(x_2 - x_1) \\ \frac{dx_2}{dt} = cx_1 - x_2 - x_1x_3 \\ \frac{dx_3}{dt} = x_1x_2 - bx_3 \end{cases} \tag{1}$$

This model becomes chaotic when: $a = 10$, $b = 8/3$ and $c = 28$ [16]. Each input binary string $M = [m_1, m_2, m_3, ..., m_n]$ (where n denotes the desired hash length), was divided appropriately, converted into two real numbers that were used as first two initial conditions ($x_{1,0}$ and $x_{2,0}$). The algorithm of coding messages into initial parameters is presented below:

1. If n = 256 do Steps 2–4.
2. L11 = ctf($[m_1, ..., m_{64}]$), L21 = ctf($[m_{65}, ..., m_{128}]$).
3. L31 = ctf($[m_{129}, ..., m_{192}]$), L41 = ctf($[m_{193}, ..., m_{256}]$).
4. L1 = ctntf(L11, L21), L2 = ctntf(L31, L41).
5. If n = 512 do Steps 6–11.
6. L11 = ctf($[m_1, ..., m_{64}]$), L21 = ctf($[m_{65}, ..., m_{128}]$).
7. L31 = ctf($[m_{129}, ..., m_{192}]$), L41 = ctf($[m_{193}, ..., m_{256}]$).
8. L51 = ctf($[m_{257}, ..., m_{320}]$), L21 = ctf($[m_{321}, ..., m_{384}]$).
9. L31 = ctf($[m_{385}, ..., m_{448}]$), L41 = ctf($[m_{449}, ..., m_{512}]$).
10. L1 = ctntf(ctntf(L11, L21), ctntf(L31, L41)).
11. L2 = ctntf(ctntf(L51, L61), ctntf(L71, L81)).

12. $x^l_{1,0} = \mathrm{L1}$, $x^l_{2,0} = \mathrm{L2}$.

Construction of *ctf* and *ctnf* functions is presented in Listing (1.1):

Listing 1.1. Functions used for mapping messages into Lorenz Attractor initial conditions.

```
function res = ctf(temp)
    sum = 0;
    tabSize = size(temp, 2);
    for i = tabSize:-1:1
        sum = sum + (2^(i - 1)) * temp(tabSize + 1 -i);
    end
    while sum > 1
        sum = sum / 10;
    end
    res = sum;
end

function res = ctntf(a, b)
    sum = a + b;
    while sum > 1
        sum = sum / 10;
    end
    res = sum;
end
```

As it can be seen, two hash lengths were considered, namely: $n = 256$ and $n = 512$. These lengths are the most popular ones in SHA-2 and SHA-3 certificated hashing functions families. Third initial condition $x^l_{3,1}$ was a random real number from range $[0, 1]$. The model described in Eq. (1) was solved with the usage of the Runge-Kutta 4th Order method that can be represented as [18]:

$$k_1 = \Delta t * f(t, x_i) \tag{2}$$

$$k_2 = \Delta t * f(t + \frac{\Delta t}{2}, x_i + \frac{k_1}{2}) \tag{3}$$

$$k_3 = \Delta t * f(t + \frac{\Delta t}{2}, x_i + \frac{k_2}{2}) \tag{4}$$

$$k_4 = \Delta t * f(t + \Delta t, x_i + k_3) \tag{5}$$

$$x_{i+1} = x_i + \left(\frac{k_1 + 2k_2 + 2k_3 + k_4}{6} \right) \tag{6}$$

where f is representing equations described in (1), x_i is a vector containing solutions in all three dimensions (that is $x_i = [x_{1,i}, x_{2,i}, x_{3,i}]$) in $i - th$ algorithm iteration, t denotes a time in which calculation is done:

$$t = t_0 + i * \Delta t \tag{7}$$

t_0 is a moment when computation starts and is equal 0, and Δt is denoting a step (assumed time intervals in which computations are done), and was equal to 0.1. The parameter i was an iterator in interval $[0, 39999]$, thus always 40000 elements of solution in all three dimensions were generated. An example solution of a Lorenz Attractor for the following initial parameters: $\Delta t = 0.1$, $x_{1,0} = 0.4$, $x_{2,0} = 0.3$, and $x_{3,0} = 0.5$ is presented in Fig. 1.

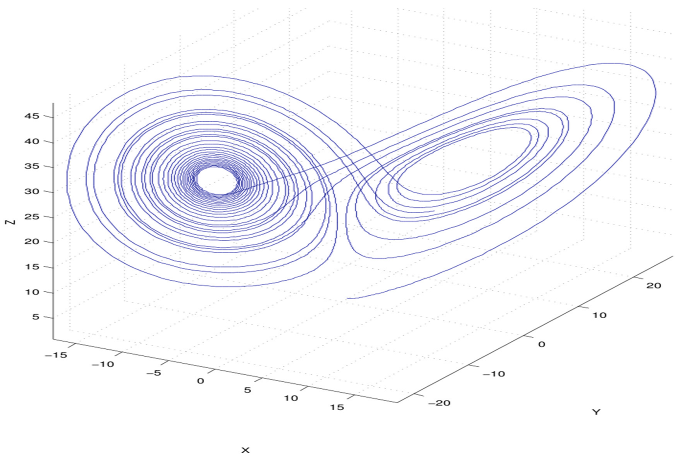

Fig. 1. Example solution of a Lorenz Attractor.

4 ANNs Training and Testing with Usage of Lorenz Attractor

In this section, the process of training and testing of Feed-Forward ANNs is described. Tested ANNs could be divided into two groups: returning 256 bits of hash and returning 512 bits of hash ($n \in \{256, 512\}$). The detailed algorithm of the whole process is presented below (with the assumption, that the value of parameter n was already chosen).

1. Input data preparation. Two sets of data were generated:

$$INPUT^{train}[i] = [b_1, b_2, ..., b_n], i = 1, ..., 10000. \tag{8}$$

$$INPUT^{test}[i] = [b_1, b_2, ..., b_n], i = 1, ..., 5000. \tag{9}$$

   Both $INPUT$ arrays represented bits of messages (denoted as $b \in \{0, 1\}$). In both cases, those bits were generated randomly and all created messages were unique (within the particular matrix as well as between matrices).

2. Target data preparation. To use the Lorenz Attractor, the formula presented in Eq. (1), messages from the training set ($INPUT^{train}$) had to be encoded as its initial conditions. Details related to the process of messages compression are described in Sect. 3. As a result, each message could have been represented as two real numbers from interval $[0, 1]$:

$$IC[i] = [XL_i, YL_i], i = 1, ..., 10000; XL_i, YL_i \in \mathbb{R} \wedge XL_i, YL_i \in [0, 1]. \quad (10)$$

IC is denoting an initial condition array. The main advantage of such solution is the fact, that it enabled to algorithmically bond each message with attractor's formula via its first two initial conditions. With IC array prepared, Lorenz Attractor formula was solved for each message from $INPUT^{train}$ separately. That is, for $i - th$ message, the initial conditions were set to: $[x_{1,0} = IC[i][0], x_{2,0} = IC[i][1], x_{3,0} = rand()]$, where $rand()$ was a function returning random real number from range $[0, 1]$. Then, the attractor was solved with usage of Runge-Kutta 4th Order method (see Eqs. (2)–(6)). The solution array for $i - th$ message can be represented as:

$$VS'_k[i] = [x_{k,0}, x_{k,1}, ..., x_{k,39999}], k \in \{1, 2, 3\}. \quad (11)$$

In all three dimensions exactly 40000 elements of solution were generated. To form a hash of $i - th$ message, results had to be truncated. Truncated vectors are presented in Eq. (12).

$$VS_k[i] = [x_{k,1*step+1000}, x_{k,2*step+1000}, ..., x_{k,n*step+1000}], k \in \{1, 2, 3\}, \quad (12)$$

where:

$$step = \lfloor \frac{40000 - 1000}{n} \rfloor. \quad (13)$$

The first 1000 elements were skipped in all cases to avoid small distances (in the Euclidean sense) between solutions in 3D space. The step parameter was calculated to cover the whole solution space, and to avoid situations when two neighboring samples in a particular dimension are too close to each other. Neighboring samples might also form an ascending or descending slope, which was undesirable. After the truncation process, all vectors had to be binarized to form hashes. The general binarization formula is presented in Eq. (14).

$$BS[i][j]_k = \begin{cases} 1 \text{ if } VS[i][j]_k \geq AVG_k[j] \\ 0 \text{ otherwise} \end{cases} \quad (14)$$

Where $AVG_k[j]$ is the average value calculated from VS_k array ($k \in \{1, 2, 3\}$), for each column $j \in [1, 2, ..., n]$. At this stage of computation, every message from $INPUT^{train}$ array had three hashes candidates (each of length equal to n) stored in arrays BS_1, BS_2 and BS_3. To complete the target data preparation only one array of hashes had to be chosen. To determine which exactly should it be, statistical tests described in [19] were performed on arrays $BS_{1,2,3}$, and after analysis of results, only one was chosen.

3. ANNs training process. For every $n \in \{256, 512\}$ value, 7 networks were created. The structure of each network was the same and could be represented as: I-HL-OL-O. I was an input layer of size n where input messages were given. HL was a hidden layer containing sigmoid neurons. The number of neurons in this layer varied in networks within one group. OL was an output layer with n linear output neurons. O was a network output that form a hash. All ANNs were trained with the usage of the Scaled Conjugate Gradient method (SCG) [12]. The training set consisted of two arrays: $INPUT^{train}$ and $TARGET^{train}$. The first one, $INPUT^{train}$, was described in the step 1. The target set was an array containing results obtained from Lorenz Attractor that were truncated but not binarized (see Eq. (12)). Target array was selected in the process of statistical analysis of the three binarized and truncated versions of arrays (see Eq. (14), that is B.P.T., S.T., and C.T. tests described in [19] were performed. Selection of BS_i array in the statistical analysis process determined that VS_i was considered as a target set ($TARGET^{train} = VS_i$). Example results of such tests performed on a different set of networks is presented in [21].
4. Output data generation. Each trained network was used to generate output test data from $INPUT^{test}$ data. $OUTPUT^{test}$ data were used to prepare binarization vectors used in the performance analysis process (see Sect. 5).
5. ANNs evaluation process. The performance of hashing network was tested (in comparison with certificated standards). This was an independent stage in which different sets of data were used. All details and results are described in Sect. 5.

5 Analysis of Performance of the Hashing ANNs

In this section, analysis of the performance of hashing ANNs is presented as well as a comparison of the performance of hashing ANNs with MATLAB implementation of SHA-256 and SHA-512 functions. The performance measure was a time of hash computation. Hashes were calculated from data that differed in size. Accordingly to [3], the computational cost of one feed-forward network pass is $O(W)$, where W is the total number of weights. The training cost is equal to $O(W^2)$. Hashing algorithms have cost $O(1)$ for small messages, and $O(m)$ for longer messages where m is the message length. This implicates directly from their construction. Experiments' details are presented below:

– MATLAB implementation of SHA-512 and SHA-256 presented in [7] was used. This is the only implementation of these certificated hashing functions officially published on the MathWorks website (which is an official file exchange platform dedicated to MATLAB users).
– MATLAB functions were tested for the following data sizes: 512 b, 100 B, 1 kB, 10 kB, 50 kB, 100 kB, and 500 kB. As a *data* here strings of appropriate length were considered. Data representing 100 kB and 500 kB were not created directly as strings, but multiple hashing operations were performed on 50 kB data strings (2 times and 4 times, respectively).

- Hashing networks were tested for the following data sizes: 512 b, 128 B, 1 kB, 10 kB, 50 kB, 100 kB, 500 kB, 1000 kB, 5000 kB, and 50000 kB. As a *data* in this scenario, arrays of bits representing messages were considered. Every array had exactly n random bits in one row, and the appropriate number of rows. Hashing 1000 kB of data (and more) was tested as a multiple hashing of 500kB array. Performance measurements included the binarization process, but in this scenario, a binarization vector was used, which can be represented as:
$$BV = [AVG_1, AVG_2, ..., AVG_n], \tag{15}$$
where AVG_i is the average value calculated from the i-th column of the ANNs $OUTPUT^{test}$ array (for each network vector BV was generated separately).
- Experiments were conducted in the MATLAB2020 environment on the Personal Computer with 16 GB RAM, and AMD Ryzen 5 2600 Six-Core Processor (3.4 GHz).
- Notation $L\{n\}N\{HL\}$ denotes hashing network train with usage of the Lorenz Attractor, returning n bits of hash, and having HL neurons in the hidden layer.

Results of the experiments are presented in Tables 1, 2, and 3.

Table 1. Performance of SHA-256 and SHA-512 implemented in MATLAB [7].

	Data size (kB)						
	0.0625	0.098	1	10	50	100	500
	Time of computation (s)						
SHA-256	0.659	0.527	4.303	41.692	245.496	498.848	2436.608
SHA-512	0.926	0.832	6.999	64.525	358.101	716.745	3570.202

Table 2. Performance of Lorenz hashing networks returning 256 bits of hash.

	Data size (kB)									
	0.0625	0.125	1	10	50	100	500	1000	5000	50000
	Time of computation (s)									
L256N128	0.012	0.009	0.008	0.012	0.024	0.043	0.179	0.352	1.825	18.667
L256N192	0.013	0.008	0.009	0.012	0.026	0.044	0.202	0.433	2.099	20.883
L256N256	0.013	0.010	0.010	0.014	0.031	0.058	0.270	0.505	2.436	24.665
L256N320	0.014	0.009	0.009	0.013	0.033	0.051	0.247	0.510	2.719	26.020
L256N384	0.015	0.011	0.011	0.014	0.036	0.054	0.280	0.580	2.864	28.548
L256N448	0.014	0.011	0.011	0.015	0.034	0.071	0.326	0.598	3,149	30.699
L256N512	0.019	0.013	0.013	0.018	0.040	0.078	0.342	0.705	3.795	37.023

Table 3. Performance of Lorenz hashing networks returning 512 bits of hash.

	Data size (kB)									
	0.0625	0.125	1	10	50	100	500	1000	5000	50000
	Time of computation (s)									
L512N256	0.016	0.015	0.013	0.018	0.036	0.054	0.239	0.456	2.297	23.585
L512N384	0.020	0.016	0.016	0.019	0.037	0.060	0.249	0.493	2.688	26.531
L512N512	0.022	0.019	0.020	0.026	0.045	0.068	0.323	0.649	3.564	34.054
L512N640	0.024	0.032	0.026	0.032	0.059	0.099	0.392	0.787	3.919	35.527
L512N768	0.024	0.023	0.025	0.029	0.063	0.106	0.464	0.854	4.259	42.69
L512N896	0.028	0.026	0.026	0.032	0.062	0.115	0.491	0.872	4.259	44.082
L512N1024	0.031	0.029	0.029	0.036	0.073	0.134	0.578	1.132	5.735	56.269

Results from the Table 2 are visualized in Fig. 2, and from Table 3 in Fig. 3. Measurements were also approximated with the usage of a Bezier curve to make them more readable. In both figures and for each network, it can be observed that until about 50 kB data size threshold, the time of computation is growing slightly, and after this threshold, the growth takes the form of a linear function. In all cases, the time of computation for networks with a bigger number of neurons in the hidden layer is growing faster than for networks with a smaller number of neurons (note that the logarithmic scale is used for both: OX and OY axes). Networks returning 512 bits of the hash are also slower than networks returning 256 bits of the hash.

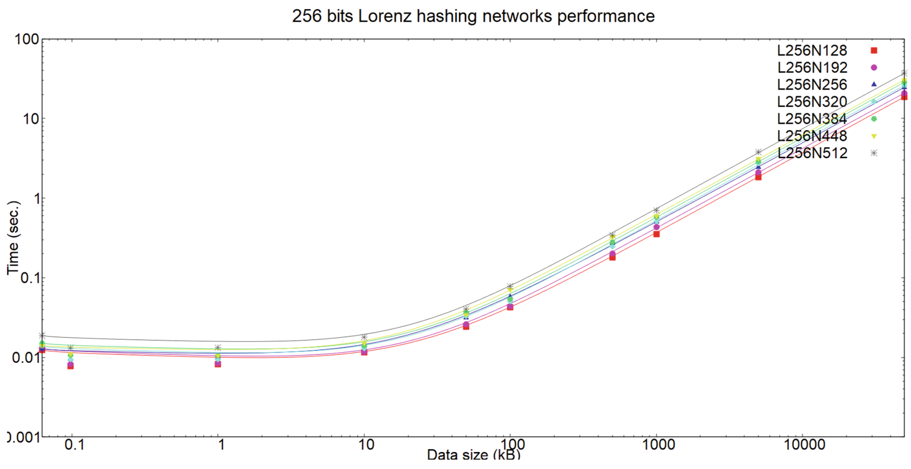

Fig. 2. Performance of hashing networks returning 256 bits of hash.

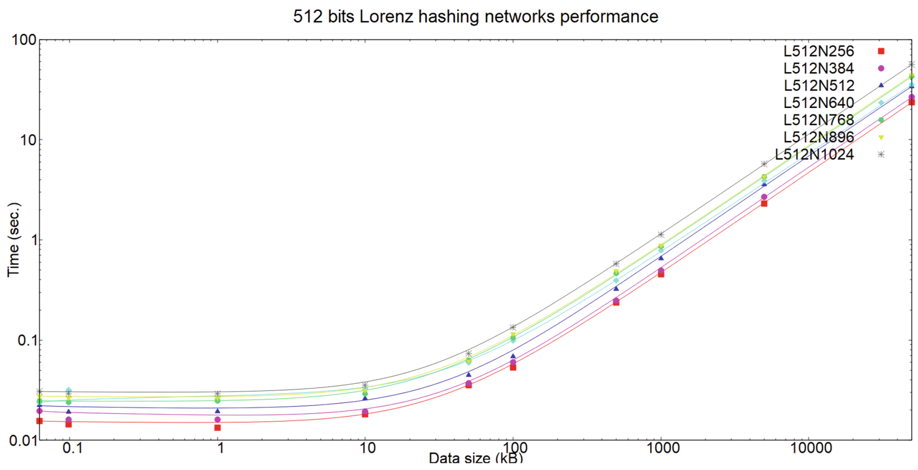

Fig. 3. Performance of hashing networks returning 512 bits of hash.

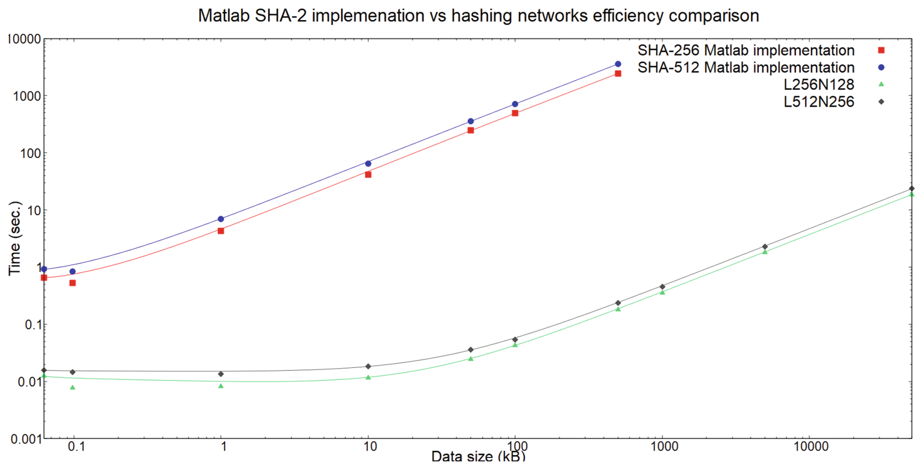

Fig. 4. Comparison of performance of the fastest hashing networks returning 256 bits, and 512 bits of hash, and MATLAB implementation of SHA-256, and SHA-512.

In Fig. 4 the performance of MATLAB implementation of SHA-256 and SHA-512 (presented in [7]), and the performance of two fastest hashing networks (one returning 256 bits of hash, and the second one returning 512 bits of hash) is compared. As it can be seen, networks are more efficient in this scenario. The time of computation for MATLAB SHA functions is also linear, however, SHA-512 algorithm is less efficient than SHA-256.

Table 4 shows a comparison of hashing efficiency of all networks and both MATLAB functions for popular data types. Assumed data sizes were [2]: 10 kB for a JPG image (JPG row), 19 kB for a PDF file (PDF row), 3.5 MB for an MP3

Table 4. Efficiency of popular data files hashing.

	Time (s)						
	L256N128	L256N192	L256N256	L256N320	L256N384	L256N448	L256N512
JPG	0.002	0.011	0.012	0.020	0.012	0.021	0.017
PDF	0.005	0.015	0.017	0.024	0.018	0.027	0.023
MP3	1.336	1.503	1.774	1.879	2.053	2.215	2.663
DVD	1565.721	1751.245	2068.131	2182.298	2394.243	2574.439	3105.545
BRM	9002.903	10069.623	11891.720	12548.144	13766.865	14802.953	17856.842
	L512N256	L512N384	L512N512	L512N640	L512N768	L512N896	L512N1024
JPG	0.006	0.012	0.028	0.075	0.027	0.015	0.042
PDF	0.010	0.017	0.034	0.081	0.035	0.023	0.052
MP3	1.691	1.908	2.461	2.612	3.077	3.164	4.063
DVD	1977.860	2225.143	2855.949	2977.090	3579.241	3696.065	4718.253
BRM	11372.694	12794.540	16421.609	17117.942	20580.548	21252.342	27129.811
	Time (different measure units)						
	MATLAB SHA-256			MATLAB SHA-512			
JPG	49.362 s			70.817 s			
PDF	93.248 s			135.104 s			
MP3	4.855 h			7.111 h			
DVD	236.715 days			346.754 days			
BRM	1361.110 days			1993.838 days			

file or an Ebook (MP3 row), 4 GB for a DVD movie (DVD row), and 23 GB for a Blue-Ray Movie (BRM row). Times presented in this table were not directly calculated, but interpolated with the usage of the linear interpolation performed on data presented in previous tables in this section. The obtained results show clearly that the cost of hashing using the ANN-based algorithms seems feasible even for very large commonly used files while comparing the obtained results with the classic implementation makes the latter much slower or actually unacceptable (even counted in days).

6 Conclusions

In this article, the concept of hashing artificial neural networks trained with the usage of the Lorenz Attractor was presented. The research was focused on the assessment of the performance of the proposed hash generators.

All the discussed networks had one hidden layer with sigmoid neurons and an output layer with n linear neurons (where n was denoting a hash length). Two values of the n parameter were considered, that is 256 and 512. For both values of n, seven networks were created that differed in the number of neurons in the hidden layer. All networks were trained with the usage of the Scaled Conjugate

Gradient method. The training set consisted of random messages (input data) and an appropriately prepared target set (hashes created from input messages with usage of the Lorenz Attractor).

ANNs presented significantly better time efficiency than SHA-256, and SHA-512 implemented in MATLAB. One of the biggest advantages of the proposed solution is a potentially scalable output of networks. The length of the returned hash can be established during the training process with a one-bit precision. Networks are also easy to replace, thus compromising one network can be easily fixed via training a new one.

Comparing the efficiency of our algorithms vs. the efficiency of MATLAB implementations of classic hashing algorithms showed, that for large, commonly used files, hashing times are still feasible for the ANN-based approach (reaching hours), while classic SHA implementations runtimes became unacceptable (reaching thousands of days).

Future work is aimed at including:

- testing different networks structures,
- testing different chaotic series,
- testing more values of n parameter,
- performing statistical tests as presented in [19]. Results of these tests will be also compared with results of the same tests performed on the certificated standards. A suite of tests presented in [19] may be also extended, for example by the Floyd-Marshall algorithm.

The example use case is presented in [21]. In the scenario described in [21], hashing networks are used to perform additional data integrity checking operations in the cloud environment. Because of variable output hash length and virtual machines' idle time, hash generation in such system has a very small computational overhead.

References

1. Akhavan Masoumi, A., Samsudin, A., Akhshani, A.: A novel parallel hash function based on a 3D chaotic map. EURASIP J. Adv. Signal Process. **2013**, 126 (2013). https://doi.org/10.1186/1687-6180-2013-126
2. Angela. Average file sizes. https://blog.online-convert.com/average-file-sizes/. Accessed Feb 2022
3. Bishop, C.M., Nasrabadi, N.M.: Pattern recognition and machine learning. Springer **4**(4), 246–249 (2006)
4. Boeing, G.: Visual analysis of nonlinear dynamical systems: chaos, fractals, self-similarity and the limits of prediction. Systems **4**(4) (2016). https://doi.org/10.3390/systems4040037
5. Gong, Z.: Survey on lightweight hash functions. J. Cryptol. Res. **3**(1), 1–11 (2016)
6. Guesmi, R., Ben Farah, M.A., Kachouri, A., Samet, M.: Hash key-based image encryption using crossover operator and chaos. Multim. Tools Appl. **75**(8), 4753–4769 (2015). https://doi.org/10.1007/s11042-015-2501-0

7. Khitish. Sha algorithms 160,224,256,384,512. https://nl.mathworks.com/matlabcentral/fileexchange/31795-sha-algorithms-160-224-256-384-512. Accessed Feb 2022
8. Kidoň, M., Dobai, R.: Evolutionary design of hash functions for IP address hashing using genetic programming. In: 2017 IEEE Congress on Evolutionary Computation (CEC), pp. 1720–1727 (2017). https://doi.org/10.1109/CEC.2017.7969509
9. Liu, H., Kadir, A., Liu, J.: Keyed hash function using hyper chaotic system with time-varying parameters perturbation. IEEE Access **7**, 37211–37219 (2019). https://doi.org/10.1109/ACCESS.2019.2896661
10. Marco, A., Martinez, A., Bruno, O.: Fast, parallel and secure cryptography algorithm using lorenz's attractor. Int. J. Mod. Phys. C **21** (2012). https://doi.org/10.1142/S0129183110015166
11. Medini, H., Sheikh, M., Murthy, D., Sathyanarayana, S., Patra, G.: Identical chaotic synchronization for hash generation. ACCENTS Trans. Inf. Secur. **2**, 16–21 (2016). https://doi.org/10.19101/TIS.2017.25002
12. Møller, M.F.: A scaled conjugate gradient algorithm for fast supervised learning. Neural Netw. **6**(4), 525–533 (1993). https://doi.org/10.1016/S0893-6080(05)80056-5
13. NIST. The secure hash algorithm validation system (SHAVS). Tech. rep. (2012). https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Algorithm-Validation-Program/documents/shs/SHAVS.pdf
14. NIST. Fips 180-4: Secure hash standard (SHS). Tech. rep. (2015). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf
15. NIST: Fips pub 202: Sha-3 standard: permutation-based hash and extendable-output functions. Tech. rep. (2015). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf
16. Peng, J., Jin, S.Z., Liu, H.I., Zhang, W.: A novel hash function based on hyperchaotic lorenz system. In: Cao, B., Li, T.F., Zhang, C.Y. (eds.) Fuzzy Information and Engineering, vol. 2, pp. 1529–1536. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03664-4_162
17. Singh, M., Garg, D.: Choosing best hashing strategies and hash functions. In: 2009 IEEE International Advance Computing Conference, pp. 50–55 (2009). https://doi.org/10.1109/IADCC.2009.4808979
18. Süli, E., Mayers, D.F.: An Introduction to Numerical Analysis, pp. 328–329. Cambridge University Press, Cambridge (2003)
19. Tchórzewski, J., Jakóbik, A.: Theoretical and experimental analysis of cryptographic hash functions. J. Telecommun. Inf. Technol. (2019)
20. Tchórzewski, J., Jakóbik, A., Grzonka, D.: Towards ANN-based scalable hashing algorithm for secure task processing in computational clouds. In: 33rd European Conference on Modelling and Simulation, pp. 421–427 (2019)
21. Tchórzewski, J., Jakóbik, A., Iacono, M.: An ANN-based scalable hashing algorithm for computational clouds with schedulers. Int. J. Appl. Math. Comput. Sci. **31**(4), 697–712 (2021)
22. Wang, J., Zhang, T., Song, J., Sebe, N., Shen, H.T.: A survey on learning to hash. IEEE Trans. Pattern Anal. Mach. Intell. **40**(4), 769–790 (2018). https://doi.org/10.1109/TPAMI.2017.2699960