Information Sciences 179 (2009) 3442–3450

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

True random number generator based on mouse movement and chaotic hash function

Qing Zhou [a,*], Xiaofeng Liao [a], Kwok-wo Wong [b], Yue Hu [a], Di Xiao [a]

[a] Department of Computer Science and Engineering, Chongqing University, Chongqing 400044, PR China
[b] Department of Electronic Engineering, City University of Hong Kong, Hong Kong, PR China

ARTICLE INFO

Article history:
Received 22 July 2008
Received in revised form 31 May 2009
Accepted 5 June 2009

Keywords:
Chaos
Mouse movement
TRNG

ABSTRACT

A true random number generator (TRNG) makes use of a non-deterministic source to produce randomness. It is considered as more secure than a pseudorandom number generator as the degree of randomness is higher. In this letter, we propose a novel TRNG which generates random bits by moving the mouse casually. It is convenient, universal and low-cost for the personal computer (PC) platform. To eliminate the regular patterns of mouse movements caused by the habit of the same user, we propose three TRNGs based on chaotic hash functions. Experiments are conducted to evaluate the speed, diffusion and randomness performance of these TRNGs. The results show that two of them possess satisfactory performance and can be implemented on common PC platform.

1. Introduction

Secure cryptography requires good random number generator(RNG). For example, secret keys used in symmetric cryptosystems [8,10] and large numbers required in asymmetric cryptosystems [11,16] should better be generated randomly so that they are not predictable. Moreover, RNGs are employed to create challenges, nonces, padding bytes, and blinding values in many cryptographic protocols [3,17,18]. There are in general two types of generators for producing random sequences: true random number generators (TRNGs) and pseudorandom number generators (PRNGs). PRNGs need some input called seeds, along with some deterministic algorithms to generate multiple "pseudorandom" numbers. They are usually faster than TRNGs and are preferable when a lot of random-like numbers are required. TRNGs make use of non-deterministic sources along with some post-processing functions for generating randomness. Such sources include physical phenomena such as thermal noise [13], atmospheric noise [4,6], radioactive decay [14] and even coin-tossing. The random sequences generated are considered as possessing a higher degree of randomness. This is the reason why TRNGs are only considered for one time pad, the only provably secure system. However, all the above-mentioned sources for TRNGs require additional devices when running on a personal computer (PC), the most popular computing platform. This makes those TRNGs inconvenient and expensive for PC applications.

The mouse is a very popular input device equipped in most PCs. It is very convenient and cost-effective to generate random numbers on PC platform by simply moving the mouse. No extra devices need to be purchases. Besides, the users are more confident on the underlying security as the random numbers are generated under their control. From this point of view, this kind of generator is more secure than PRNGs, where all the numbers are predictable once the seed is known.

When the mouse movement is taken as the entropy source, the post-processing function should be chosen carefully. A user usually moves the mouse in a regular pattern, which might make the generated sequences not random at all.

* Corresponding author. Tel./fax: +86 23 65103199.
E-mail address: tzhou@cqu.edu.cn (Q. Zhou).

0020-0255/$ - see front matter Crown Copyright © 2009 Published by Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2009.06.005

Nevertheless, there are no two identical traces created by the same user. If the post-processing function works in a very sensitive way, i.e., even the slightest variation of mouse movement changes the output random number substantially, the patterns in mouse traces are effectively eliminated.

Chaotic systems are well-known as very sensitive to both initial states and system parameters. Furthermore, other characteristics such as ergodicity, pseudo-randomness and mixing properties make chaos-based approaches possess great potential for various cryptographic applications such as hash function [15,20,21], block cipher [8], image encryption [5,22], public-key cryptography [12] and key agreement protocol [18]. For the same reason, chaotic cryptosystems are considered as a good candidate for the post-processing of mouse movement so as to generate the final random bits. In fact, there were already some chaos-based TRNGs proposed [1,2,19]. However, they all require extra hardware and are not suitable for ordinary PC users.

There are at least two cryptographic methods to post-process the data captured from mouse movements. The first one makes use of image encryption algorithms. It is quite natural as the trace of mouse movement can be captured as a 2D image. In [7], we proposed a TRNG based on 'MASK' image encryption algorithm where the random number is generated from the cipher image. Alternatively, the mouse movement can be considered as a message to be processed by a hash function. The output of the hash function, i.e., the hash value, is mapped to the final random bits. This approach is more preferable since hash functions are usually faster than image encryption algorithms. Besides, the diffusion characteristic of hash function is desirable for TRNGs based on mouse movements.

In this letter, three TRNGs based on chaotic hash functions are proposed. The random sequences produced by these generators are evaluated using the 15 statistical tests recommended by U.S. NIST [9]. Experimental results show that these TRNGs possess good diffusion and randomness properties.

The rest of the paper is arranged as follows. Section 2 introduces the basic conversions used by our approaches. Section 3 describes two proposed approaches based on hash functions. Test results on speed, diffusion and randomness performance are presented in Section 4. Conclusions are given in Section 5.

2. Basic conversions

In this section, three basic conversions are introduced. They are useful in mapping the mouse movement data to the random bits.

2.1. Analog-to-digital conversion

Mouse movement is essentially a physical motion. In other words, the trace of such a movement is an analog signal while PC can only process digital sequences. Therefore, an analog-to-digital (A/D) conversion needs to be performed. Fortunately, this conversion is supported by almost all operating systems. With a simple calling of application programming interfaces (APIs), a serial of sampled coordinates corresponding to the mouse movement are returned by the operating system. The value of those coordinates usually ranges from 1 to several thousands. Fig. 1a shows the sampled points in a trace of mouse movement.

As stated before, regular mouse movement patterns are usually found for the same user. This means that there is a high similarity among the sampled movements even when the mouse traces are captured at different time. On the other hand, different users usually have distinct movement patterns, as shown in Fig. 1. To get rid of the regular patterns made by the same user, chaos-based hash functions are employed for post-processing.

2.2. Trace-to-number conversion

The input to most chaos-based hash functions is a sequence of real numbers between 0 and 1. Therefore, the coordinates of the sampled points along the mouse trace should first be converted to such a real number sequence. Here we propose the following simple conversion:

Step 1. Get the coordinates of two adjacent points A(x_i, y_i) and B(x_{i+1}, y_{i+1});
Step 2. Compute the angle of line AB using the following equation:

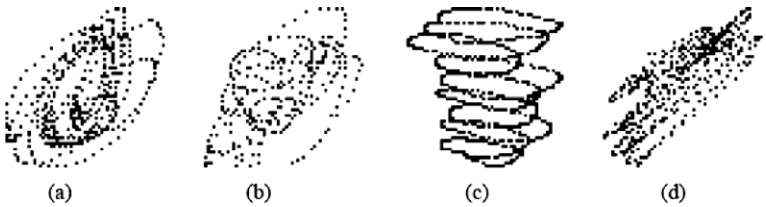

Fig. 1. Sampled points in trace of mouse movements (a) 1st movement by user A, (b) 2nd movement by user A, (c) 1st movement by user B and (d) 1st movement by user C.

$$\theta_i = \begin{cases} \pi/2, & x_{i+1} = x_i \\ \arctan\left(\frac{|y_{i+1}-y_i|}{|x_{i+1}-x_i|}\right), & \text{otherwise} \end{cases} \tag{1}$$

Step 3. Map θ_i to a real number ranging from 0 to 1:

$$r_i = \frac{\theta_i}{\pi/2} \tag{2}$$

Therefore, given n adjacent sampled points, $n-1$ real numbers are obtained using this method. Fig. 2 illustrates the conversions of 3 points. In the following experiments, we use 129 points to get 128 real numbers whose value is between 0 and 1.

2.3. Number-to-bit conversion

As mentioned above, chaos-based hash functions usually deal with real numbers. However, the final hash value is composed of a fixed number of bits. Therefore, the conversion between floating-point numbers and bits is required.

In our implementation, the IEEE 754 double precision floating-point format is adopted to store real numbers. This means that the fraction part contains 52 bits and a real number x ($0 < x < 1$) can be represented as follows:

$$x = \sum_{i=1}^{52} x_i 2^{-i} \tag{3}$$

where $x_i \in \{0,1\}$. Note that the above equation also defines the conversion of a binary sequence to a real number between 0 and 1.

Accordingly, the following operation can be used to obtain x_i or the ith bit of x:

$$x_i = T(x,i) = ((x \ll 52) \wedge (1 \ll i)) \gg i \tag{4}$$

where '$\wedge$' is the logical AND operation, '$\ll$' and '$\gg$' is the left and right shift operation, respectively.

The operation of extracting the ith to the jth bits is denoted as $T(x,i{:}j)$, which is defined as:

$$T(x, i:j) = [T(x,i), T(x,i+1), \ldots, T(x,j)] = [x_i, x_{i+1}, \ldots, x_j] \tag{5}$$

3. TRNGs based on chaotic hash function

In this section, three TRNGs based on chaotic hash functions are proposed. The test results of these TRNGs are presented in the next section.

3.1. FFNF-based TRNG

Zhang et al. proposed a chaotic key hash function based on FFNF (feedforward-feedback nonlinear digital filter) [21]. It has been proven that this scheme can generate nth-order chaotic signals with uniform distribution. Fig. 3 shows the structure of the chaotic key hash function based on FFNF. The signals in the filter are given by:

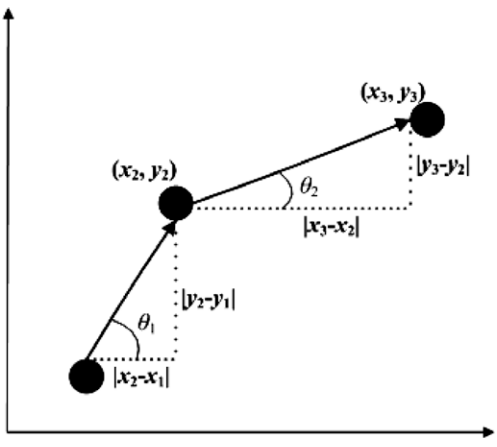

Fig. 2. Illustration of trace-to-number conversion.

$$\begin{cases} y(t) = h \circ \mathrm{mod}\left(\phi + \sum_{i=1}^{2} c_i z_i(t)\right) \\ \sigma(t) = g \circ \mathrm{mod}\left(y(t) + \sum_{i=1}^{2} s_i z_i(t)\right) \\ z_1(t+1) = y(t) \\ z_2(t+1) = z_1(t) \end{cases} \tag{6}$$

where ϕ is the input, c_i and s_i are system parameters, z_i(t) is the state, mod(.) is the modular operation, h and g are piecewise linear chaotic maps (PWLCM) defined as follows:

$$h(w) = g(w) = \begin{cases} w/p, & 0 < w < p \\ (w-p)/(0.5-p), & p \leqslant w < 0.5 \\ (1-w-p)/(0.5-p), & 0.5 \leqslant w < 1-p \\ (1-w)/p, & 1-p \leqslant w < 1 \end{cases} \tag{7}$$

Contrast to common FFNFs, the system parameters c_i and s_i are time-varying and are governed by the input message. Therefore this type of hash function possesses good cryptographic performance. It is briefly described as follows:

Step 1. Divide the input message M into n L-bit blocks, where L is the length of the hash value;
Step 2. Input the secret key SK = $\{\phi_0, z_1(0), z_2(0), p_h, p_g\}$, where ϕ_0 is the initial input, $z_1(0)$, $z_2(0)$ are initial system states, p_h, p_g are the system parameter of the PWLCM $h(w)$ and $g(w)$, respectively. The initiated hash value H_0 is set to $\{0\}_0^L$;
Step 3. For i = 1 to n, process each block repeatedly:
  Step 3.1. $\phi = \phi_{i-1}$;
  Step 3.2. For j = 1 to L, process each bit in block M_i repeatedly:
    (1) $r_j = H_{i-1}(j) \oplus M_i(j)$, where $H_{i-1}(j)$ and $M_i(j)$ is the jth bits of H_{i-1} and M_i, respectively, $\oplus$ is the exclusive OR operation.
    (2) If $r_j = 0$, then $c_1 = 3.57$, $c_2 = 4$, $s_1 = -2.3$, $s_2 = 3$.
     Otherwise $c_1 = 5.7$, $c_2 = 7$, $s_1 = 3.7$, $s_2 = 5$.
    (3) Iterate the proposed FFNF for one step to obtain the output signal $\sigma(t)$.
    (4) $H_i(j) = T(\sigma(t), 52)$, where function T(.) is defined by Eq. (4).
  Step 3.3. $\phi_i = \sigma(t)$;
Step 4. Output the final hash value H_n.

With its sophisticated design, this hash function is very effective in generating a good hash value [21]. However, the speed performance is not satisfactory. To process one bit of the input message, the FFNF needs to be iterated for one step. A single iteration requires a lot of multiplication/division operations and many other operations such as addition, logical branching and number-to-bit conversion.

To generate a random number, the 128 real numbers obtained from the mouse trace are first converted to 6656 bits. Then the above-mentioned method is utilized to generate a 256-bit hash value. In our experiments, the secret key is chosen as follows: $\phi_0 = 0.564$, $z_1(0) = 0.689$, $z_2(0) = -0.548$, $p_h = 0.35$ and $p_g = 0.25$.

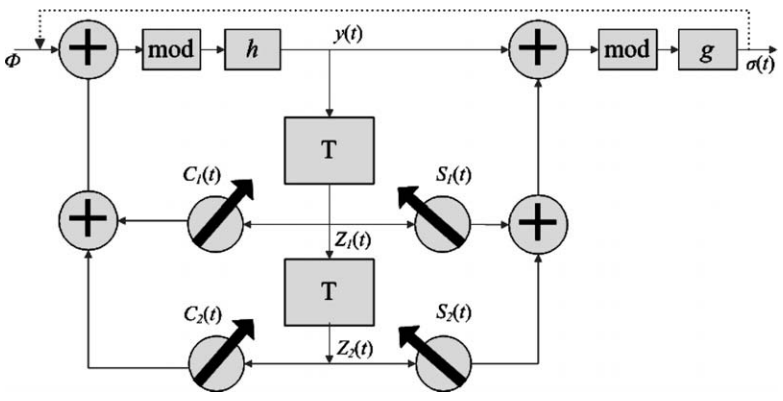

Fig. 3. Structure of hash function based on FFNF.

3.2. Tent map based approach

Xun proposed a hash function based on chaotic tent maps which is simple, effective and efficient [20]. Its architecture is shown in Fig. 4. The rectangle denotes function $G^n(a,x_0)$ which stands for n-time iteration of the tent map $G(a,x_0)$ defined as follows:

$$G(a,x_0)=\begin{cases} x_0/a & 0<x_0\leqslant a \\ (1-x_0)/(1-a) & a<x_0<1 \end{cases} \tag{8}$$

The states s_i and t_i in Fig. 4 are computed as follows:

$$\begin{cases} g_i = G^n(s_i \oplus M_i, t_i \oplus M_i') \\ t_{i+1} = g_i \oplus s_i \\ s_{i+1} = (g_i \oplus M_i') \otimes t_i \end{cases} \tag{9}$$

where M_i is a 52-bit floating-point number between 0 and 1, M_i' has the same set of bits as M_i, but all the bits are reversed in position, i.e., $M_i = \sum_{i=1}^{52} x_i 2^{-i}$ and $M_i' = \sum_{i=1}^{52} x_i 2^{i-53}$.

The $\oplus$ and $\otimes$ operations are defined as:

$$a \oplus b = (a+b) \bmod 1 \tag{10}$$

$$a \otimes b = G(\min(a,b), \max(a,b)) \tag{11}$$

The hash function can be briefly described as follows:

Step 1: The input message is first converted to n floating-point numbers M_i, $(0 \leqslant M_i < 1)$. Both s_0 and t_0 are set to zero;
Step 2: For each number M_i, compute the corresponding M_i', s_{i+1} and t_{i+1} using Eq. (9);
Step 3: Convert (s_n, t_n) to a 104-bit hash value using the method introduced in Section 2.3.

This hash function is faster than the FFNF-based approach since it requires only n ($n = 75$ as recommended in Ref. [20]) iterations of tent map to process an input real number. The output hash value is also sensitive to the input messages since both the initial state and the system parameter of the tent map are determined by the current message m_i and the two states s_i and t_i. Moreover, the states s_i and t_i are in turn sensitive to the previous messages.

To generate a random number, we input the first 128 real numbers to the hash function based on tent map. It then returns a 104-bit hash value as the generated random bits.

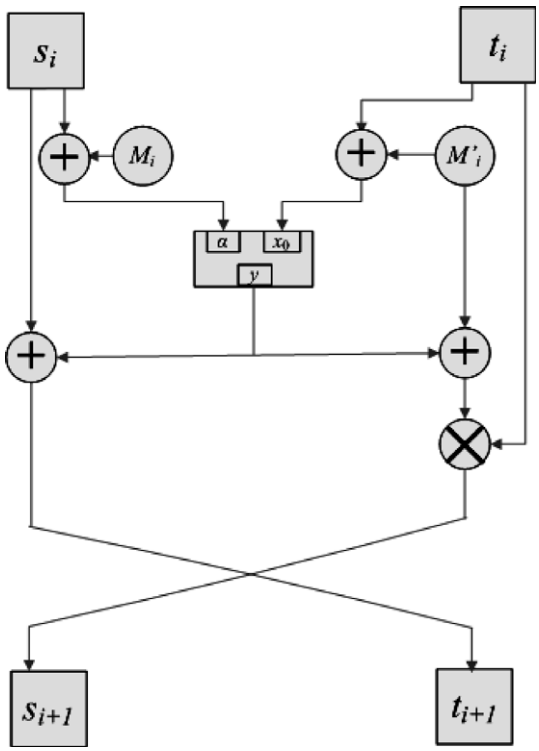

Fig. 4. Structure of hash round function based on tent map.

3.3. A new approach based on tent map

Experiment results show that both TRNGs based on FFNF and tent map can generate random numbers with good randomness (see Section 4 for details). However, the speed of the FFNF-based approach is not competitive. On the contrary, the speed of the TRNG based on tent map is faster, but it can only produce a 104-bit random number with a single mouse movement. Such a random number may be adequate for common applications, but is considered short for cryptographic purpose. For example, the key length of AES (Advanced Encryption Standard) should be 128, 196 or 256 bits. To acquire adequate random bits as required, the user has to move the mouse several times, which is quite troublesome. To overcome this weakness, we revise Xun's hash function to produce 256 bits with a single mouse movement. The structure of the new algorithm can be found in Fig. 5.

The hash function proposed by Xun, i.e., 'T-hash' function, is adopted as the core component of the new algorithm. Three 'T-hash' functions work together to increase the number of output bits. Note that each 'T-hash' function interacts with the other two functions, since the tent maps in three 'T-hash' functions work in the coupled map lattice (CML) mode:

$$\begin{cases} x_{n+1,1} = (1-e)f(x_{n,1}) + \frac{e}{2}[f(x_{n,2}) + f(x_{n,3})] \\ x_{n+1,2} = (1-e)f(x_{n,2}) + \frac{e}{2}[f(x_{n,3}) + f(x_{n,1})] \\ x_{n+1,3} = (1-e)f(x_{n,3}) + \frac{e}{2}[f(x_{n,1}) + f(x_{n,2})] \end{cases} \tag{12}$$

where $x_{n,i}$ denotes n iterations of the ith ($i = 1, 2, 3$) site, f denotes the tent map and $e = 0.05$ is a coupling constant. Except for the CML mode, other components of the 'T-hash' function remain unchanged: the current message block M_i and two registers S_i and T_i together determine the initial state and system parameter for tent map, and the chaotic states after 75 iterations determine the values of S_{i+1} and T_{i+1} in turn.

The steps to obtain a 256-bit random number are described as follows:

Step 1. Input 128 real numbers generated from mouse movement.
Step 2. Initialize all the registers used, $S_1 = 0.1$, $T_1 = 0.2$, $S_2 = 0.3$, $T_2 = 0.4$, $S_3 = 0.5$, $T_3 = 0.6$.
Step 3. For $i = 1$ to 128, process each input number and update S_1, T_1, S_2, T_2, S_3 and T_3 according to Fig. 5 and Eq. (12).
Step 4. Convert S_1, T_1, S_2, T_2, S_3 and T_3 to 40, 40, 48, 48, 40, 40 bits, respectively, using the method introduced in Section 2.3 to obtain the 256-bit random number.

4. Experiment results

The following experimental procedures are taken to test the performance of the three approaches described in Section 3. First, three users are invited to produce their own mouse movements, each for 4096 trials. Then the captured images are processed by the three approaches individually. The random numbers generated are stored in nine different files. The file size is 425,984 bits for the second approach and exactly 1M bits for the other two approaches. The speed and diffusion properties of each approach are first analyzed, followed by the randomness evaluation using U.S. NIST statistical test suite.

4.1. Speed

Table 1 lists the average time required to generate the random numbers using different approaches. The time consumed for a TRNG based on 'MASK' image encryption algorithm proposed in [7] is also listed for comparison. All approaches were implemented with non-optimized Matlab codes, running on an ordinary PC with a 1.5 GHz Intel Celeron CPU. It is observed

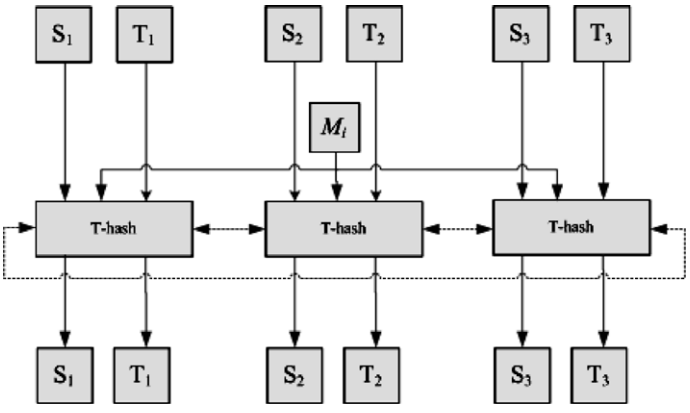

Fig. 5. Structure of the modified TRNG based on tent map.

Table 1
Average time required to generate a random number using different approaches.

Approach	MASK	FFNF	Tent map	New approach
Total time (milliseconds)	535	393	92	282
Number of random bits	256	256	104	256
Time to generate a bit (milliseconds)	2.09	1.54	0.88	1.10

from Table 1 that the three TRNGs based on chaotic hash function are much faster than the one based on image encryption algorithm. This is mainly because the data volume processed by hash functions is much lower than that by the image encryption algorithm.

4.2. Diffusion property

Diffusion is one of the important properties for hash function. It is also a critical characteristic for our proposed TRNGs because the traces of mouse movement made by the same user usually possess a high similarity. In the experiment, the last sample point, i.e., the 129th point, of the mouse movement is slightly moved to its left neighbouring position. The hash value generated is then compared with the original one. The ideal change rate should be 0.5. The experiment has been performed for 1000 traces of mouse movements made by user A. Table 2 lists the average change rate for each approach. From the table, it is clear that the TRNG based on FFNF possesses the best diffusion property. The diffusion properties of the other two TRNGs are also satisfactory.

4.3. Randomness property

The U.S. NIST statistical test suite is used to test the randomness of the generated bits. It includes 15 statistical tests and each of them is formulated to test a null hypothesis that the sequence being tested is random. There is also an alternative hypothesis which states that the sequence is not random. For each test, there is an associated reference distribution (typically normal distribution or χ^2 distribution), based on which a *P_value* is computed from the binary sequence. If this value is greater than a pre-defined threshold α (0.01 in default), the sequence passes the test. The two approaches that NIST has adopted include the examination of (1) proportion of sequences that pass a statistical test, and (2) uniformity of the distribution of those *P_values*.

According to [9], if m sequences were tested, the proportion of sequences that passed a specific statistical test should lie above p_α:

$$p_\alpha = (1-\alpha) - 3\sqrt{\frac{\alpha(1-\alpha)}{m}} \tag{13}$$

In our experiment, $m = 1000$, $\alpha = 0.01$, and $P_a = 98.05\%$.

To check the distribution of *P_values*, the interval between 0 and 1 is divided into 10 sub-intervals. The number of *P_values* in each sub-interval is counted, based on which a *P_valueT* is calculated. If *P_valueT* > 0.0001, the sequences are considered to be uniformly distributed.

The NIST statistical test suite contains 15 tests (the Lempel-Ziv complexity test is removed from the test suite since Version 1.7). For the details of those tests, please refer to [9]. Some tests such as FT, FBT, RT, ST, AET and CST require only 100 bits for each sequence. Other tests, however, require more bits. Specially, the PTMT, LZCT, RET, REVT tests need about 1 M for each sequence. As 1000 sequences are used in the experiment, the total number of bits required for these tests is huge.

In this experiment, three users were invited to produce 4096 mouse movements each. After that, the captured images were post-processed by the three approaches, resulting in about 1 M, 425 K and 1 M bits, respectively. As for FT, FBT, RT, ST, AET and CST tests, each sequence is just the hash value produced by the corresponding approach, and we use the first 1000 images to obtain the 1000 sequences. The passing proportion and *P_valueT* of the sequences generated by user A using the three approaches are listed in Table 3. In the test of *P_value* uniformity for FT and CST, each sequence is set to 1024 bits by concatenating a couple of consecutive hash values. Otherwise, the number of different *P_value* would not be sufficient to carry out the uniformity test.

As for other tests which require more than 100 bits for each sequence, we produce 1000 different sequences by setting 1000 different initial states of hash functions. Specifically, for the FFNF based approach, ϕ_0 varies from 0.564000 to 0.564999; for the new approach based on tent map, S_1 ranges 0.1000 to 0.1999. The test results can also be found in Table 3. Note that these tests

Table 2
Change rate for the three approaches.

Approach	FFNF	Tent map	New approach
Change rate (%)	50.1	49.4	49.6

Table 3
Test results for the three approaches (user A).

Test name	FFNF		Tent map		New approach	
	Proportion	*P_valueT*	Proportion	*P_valueT*	Proportion	*P_valueT*
FT	0.9880	0.0877	0.9920	0.0076	0.9930	0.1223
FBT	0.9900	0.4540	0.9960	0.0529	0.9910	0.2236
CST*	0.9890	0.1503	0.9920	0.0468	0.9940	0.1875
RT	0.9890	0.1766	0.9900	0.0780	0.9880	0.0391
LROBT	0.9980	0.1855	0.9970	0.5810	0.9920	0.0616
AET	0.9910	0.0018	0.9930	0.7868	0.9890	0.9737
ST*	0.9890	0.5121	0.9870	0.7136	0.9920	0.6434
RBMRT	0.9870	0.0103	–	–	0.9930	0.3221
DFTT	0.9930	0.3041	–	–	0.9850	0.2622
ATMT*	0.9830	0.0698	–	–	0.9810	0.0012
PTMT	0.9840	0.3702	–	–	0.9840	0.4118
MUST	0.9900	0.0835	–	–	0.9820	0.0640
RET*	0.9841	0.3139	–	–	0.9894	0.6352
REVT*	0.9872	0.2470	–	–	0.9877	0.9216
LCT	0.9850	0.3488	–	–	0.9930	0.3267

Table 4
Passing proportion of the new approach for the three users.

Test name	User A	User B	User C
FT	0.9880	0.9910	0.9900
FBT	0.9900	0.9940	0.9880
CST*	0.9890	0.9920	0.9900
RT	0.9890	0.9940	0.9930
LROBT	0.9980	0.9920	0.9880
AET	0.9910	0.9930	0.9860
ST*	0.9890	0.9900	0.9850
RBMRT	0.9870	0.9930	0.9900
DFTT	0.9930	0.9860	0.9910
ATMT*	0.9830	0.9820	0.9830
PTMT	0.9840	0.9820	0.9860
MUST	0.9900	0.9850	0.9890
RET*	0.9841	0.9878	0.9861
REVT*	0.9872	0.9913	0.9896
LCT	0.9850	0.9890	0.9880

are not carried out for the second approach. Tests ATMT, ST, CST, RET and REVT, whose names are marked by asterisks in Table 3, include a series of sub-tests, in such case only the minimum passing proportion of all the sub-tests is given. As for RET and REVT, only a portion of sequences are tested because of the limitation of the number of zero crossings [9].

It is observed from Table 3 that both the FFNF approach and the new approach based on tent map pass all the statistical tests, i.e., the passing proportions are greater than 98.05% and *P_valueT* greater than 0.0001. The test results for the other two users are similar to those of user A. Table 4 lists the test results of the new approach based on tent map for the three users. According to [9], we can conclude that the data generated by these two approaches are random.

5. Conclusion

Random numbers are essential elements in many cryptographic applications. An approach to generate random numbers based on mouse movement is proposed. The procedures of converting the physical mouse movement to random bits are described. However, there usually exist patterns among the mouse movements produced by the same user which would affect the randomness of the generated numbers. To solve this problem, the mouse movement is post-processed by chaotic hash functions. Three TRNGs based on hash functions are investigated. Experiments have been performed to test their speed, diffusion and randomness properties. Both the FFNF approach and the new approach based on tent map show excellent performance in these aspects. Therefore, they are convenient, cheap, universal, fast and secure, and can be used on common PC platform.

Acknowledgements

The work is supported by the National Nature Science Foundation of China under Grant 60573047 and 60703035, the Natural Science Foundation Project of CQ CSTC under Grant 2009BA 2024 and Program for New Century Excellent Talents in University of China (Grant No. NCET-08-0603).

References

[1] G.M. Bernstein, M.A. Lieberman, Random number generation using chaotic circuits, IEEE Transactions on Circuits and Systems 37 (1990) 1157–1164.
[2] S. Callegari, R. Rovatti, G. Setti, Embeddable ADC-based true random number generator for cryptographic applications exploiting nonlinear signal processing and chaos, IEEE Transactions on Signal Processing 53 (2005) 793–805.
[3] F. Cao, Z. Cao, A secure identity-based proxy multi-signature scheme, Information Sciences 179 (3) (2009) 292–302.
[4] D. Davis, R. Ihaka, P. Philip Fenstermacher, Cryptographic randomness from air turbulence in disk drives, Advances in Cryptology (1994) 114–120.
[5] J. Fridrich, Symmetric ciphers based on two-dimensional chaotic maps, International Journal of Bifurcation and Chaos 8 (1998) 1259–1264.
[6] W.T. Holman, J.A. Connelly, A.B. Downlatabadi, An integrated analog/digital random noise source, IEEE Transaction on Circuits and System I 44 (1997) 521–528.
[7] Y. Hu, X. Liao, K.W. Wong, Q. Zhou, A true random number generator based on mouse movement and chaotic cryptography, Chaos Solitons Fractals (2007), doi:10.1016/ j.chaos. 2007.10.022.
[8] G. Jakimoski, L. Kocarev, Block encryption ciphers based on chaotic maps, IEEE Transaction on Circuits System I 48 (2002) 163–169.
[9] NIST, A Statistical Test Suite for Random and Pseudo-random Number Generators for Cryptographic Applications, <http://csrc.nist.gov/rng/rng2.html>, 2001.
[10] NIST, Announcing the ADVANCED ENCRYPTION STANDARD (AES), <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>, 2001.
[11] R.L. Rivest, A. Shamir, L.M. Adleman, A method for obtaining digital signatures and public-key cryptosystems, Communications of the ACM 21 (2) (1978) 120–126.
[12] R. Tenny, L.S. Tsimring, Additive mixing modulation for public key encryption based on distributed dynamics, IEEE Transactions on Circuits and Systems I 52 (2005) 672–679.
[13] C. Tokunaga, D. Blaauw, T. Mudge, True random number generator with a metastability-based quality control, IEEE Journal of Solid-state Circuits 43 (1) (2008) 78–85.
[14] J. Walker, HotBits: Genuine Random Numbers Generated by Radioactive Decay, <http://www.fourmilab.ch/hotbits>, 2002.
[15] Y. Wang, X. Liao, K. Wong, One-way hash function construction based on 2D coupled map lattices, Information Sciences 178 (5) (2008) 1391–1406.
[16] B. Wang, Q. Wu, Y. Hu, A knapsack-based probabilistic encryption scheme, Information Sciences 177 (19) (2007) 3981–3994.
[17] D. Xiao, X. Liao, S. Deng, A novel key agreement protocol based on chaotic maps, Information Sciences 177 (4) (2007) 1136–1142.
[18] D. Xiao, X. Liao, S. Deng, Using time-stamp to improve the security of a chaotic maps-based key agreement protocol, Information Sciences 178 (6) (2008) 1598–1602.
[19] M.E. Yalcn, J.A.K. Suykens, J. Vandewalle, True random bit generation from a double-scroll attractor, IEEE Transactions on Circuits and Systems I 51 (2004) 1395–1404.
[20] X. Yi, Hash function based on chaotic tent maps, IEEE Transactions on Circuits and Systems II 52 (2005) 354–357.
[21] J. Zhang, X. Wang, W. Zhang, Chaotic keyed hash function based on feedforward-feedback nonlinear digital filter, Physics Letters A 362 (2007) 439–448.
[22] Q. Zhou, K.W. Wong, X. Liao, T. Xiang, Y. Hu, Parallel image encryption algorithm based on discretized chaotic map, Chaos Solitons Fractals 38 (4) (2008) 1081–1092.