KIST: A new encryption algorithm based on splay

R. Wei and Z. Zeng
Department of Computer Science
Lakehead University
Thunder Bay, Ontario P7B 5E1, Canada
{rwei,zzeng}@lakeheadu.ca

Abstract

In this paper, we proposed a new encryption algorithm called KIST. This algorithm uses an asynchronous key sequence and a splay tree. It is very efficient in the usage of both space and time. Some elementary security tests have been done.

Key words asynchronous key sequence, splay tree, symmetric key encryption

1   Introduction

In this paper, we propose a new encryption algorithm called KIST (key insertion and splay tree encryption). Some of the characteristics and advantages are as follows.

- An asynchronous key sequence is used, which depends on an initial key and plaintext encrypted.

- A splay tree is used so that the substitution is dynamic.

- The encryption is fast and uses small space.

- Cipher texts are compressed in most cases.

- The block size of the plain text and key size are flexible.

- It is good for massage integrity.

Splay trees were first described in 1983 by Sleator and Tarjan in [7] and the details were presented in [8] in 1985. Splay trees were originally intended as self-balancing binary search trees with the property that recently accessed nodes are quick to access again. This property was applied to data compression by Jones in [3]. The difference between a compression splay tree and a search splay tree is that the compression tree does not require a lexicographic ordering of the nodes, that simplifies the algorithm. An algorithm called semi-rotations was defined and used in [3] for data compression to eliminate the need of treating the zig-zag cases and thus simplified the algorithms. [1] proved that on an n-node splay tree, the amortized cost of an access at distance d from the preceding access is $O(\log(d + 1))$. In addition, there is an $O(n)$ initialization cost. The accesses include searches, insertions, and deletions. More information about splay compression can be found in [2].

The author of [3] noticed that the compression algorithm has a property that the loss of one bit from the stream of compressed data is catastrophic and thus suggested the use

of such compression in encryption. In [4], some brief discussion about the security of the splay tree based encryption was given. [4] also indicated that some ideas of that were used by Lotus in their product Ami Pro.

In general, the method suggested in [4] is not secure. Using splay trees, the more frequent used bytes will be encoded into shorter codes. So the ciphertext definitely will give some information to the attacker. However, the method using splay tree for encryption has some attractive characteristics. It is efficient in regarding both time and space. The splay tree compression operation needs only 2 KB to 4 KB memory. Also the cipher text is compressed under this method.

In this paper, we will propose a new encryption algorithm which applies some techniques of splay trees. We first give a brief review of splay trees.

1.1 Splay trees

In a splay tree for binary search proposed in [8], the tree is splayed when a node is accessed. The idea is that the accessed node becomes the root, and all nodes to the left of it form a new left subtree while all the other nodes form a new right subtree. To reduce the amount of restructuring during the splaying, a method called semi-splay was proposed in [8]. Using semi-splay, only some of the edges along an access path are rotated. That will improve the operations in zig-zag cases.

In [3], splay trees are applied for data compression. To do the data compression, prefix-free codes are used. There are some difference between the binary search trees and data compression trees. As a binary search tree, a lexicographic ordering is kept, that is not necessary for the data compression tree. On the other hand, in a prefix-free code based tree (used for compression), only the leaves can be accessed while all nodes (inner nodes and leaves) in a search tree can be accessed. Considering these differences, Jones in [3] proposed an operation called semi-rotation. Since this method will be used later, we give a little more details in the following.

In a semi-rotation, only two links in the tree are exchanged to move the accessed node up. We will use the following example in [3] to explain the idea. In the example, we use numbers to denote the inner nodes and use alphabetic characters to denote leaves. Only leaves will be accessed. In Figure 1, the node C in the left tree was accessed. So there are two semi-rotations between node C and root 1. First semi-rotation exchanges the links of C and B and the second semi-rotation exchanges the links of nodes E and 3. The resulting tree is shown as at the right of Figure 1. In general, the splay will be performed from the accessed leave up to the root along the path. Every two links are exchanged from the bottom up recursively. When the number of nodes on the path is odd, the last node (the root) is skipped. In the above example, the nodes between C up to the root are $4, 3, 2, 1$. First we go up through 4 and 3, and do a semi-rotation. Then go up through 2 and 1, and do the second semi-rotation. This is an even case.

Following the convention of prefix-free encoding tree, encoding of a leaf is followed the path from the root to the leaf. A left edge is encoded by a bit 0 and a right edge is encoded by a bit 1. In above example, the original code of C is 0110. After splaying, the code is changed to 10. So next time C will be transmitted to a shorter code. Note that each node accessed after splaying will be encoded as a code of half size. This property is good for data compression because more frequency characters will encoded into shorter codes. However, it is not good for security if we use splaying for encryption.

[3] suggested to use a secret initial state of the tree as a encryption key. Then the splay
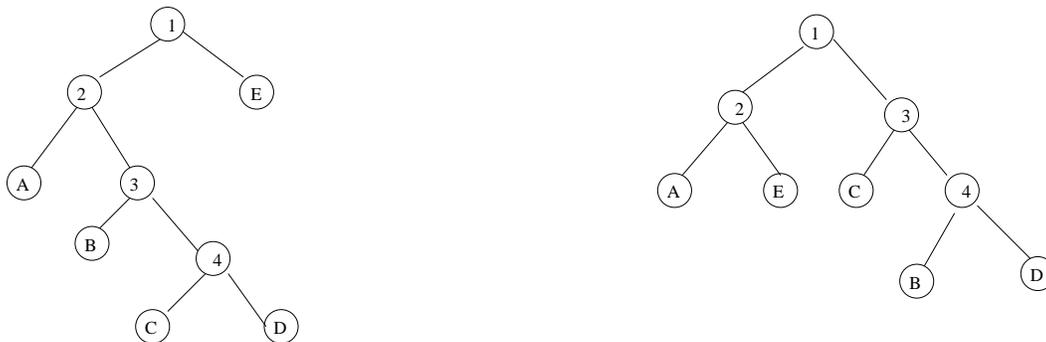
Figure 1: Semi-rotations splay

algorithm is used to encode (encrypt) the inputs. Since the initial state is kept in secret, the attacker will be difficult to find out the original text. This method is not secure. [4] discussed several security weakness of that encryption. Basically, a chosen plain attack (actually, the known plain text attack can also reveal some information) is able to reform the shape of the tree and thus break the encryption. For example, when a string *aaaaaaaa* or a string *abababababababab* is encoded, then the position of node a or a, b is forced to form. (As an example, we will use a tree of 8 layers for the byte encodes in this paper). In this way, one can change the tree in his wish. One method for possible solutions suggested in [4] is using random insertions. The idea is to insert random bytes to the inputs. Since the random bytes are secret, the attacker will be more difficult to figure out the shape of the tree. Another method suggested in [4] is using run-length encoding. Instead of inserting the random bytes at intervals with fixed length, the bytes are inserted at intervals with pseudo random lengths. We will see later that these methods are not good enough to secure the encryption.

The rest of this paper is organized as follows. In Section 2, we analysis the security of the previous splay tree based encryption and indicated its weakness. Section 3 gives the details of our proposed algorithm. Section 4 gives some experimental results of our algorithms. Section 5 gives a brief conclusion.

2   Security considerations

To understand the splay trees, let us look at a small example which contains 8 leaves. We set the original tree as a complete binary tree shown in the top of the Figure 2. Starting from that tree, we encode four letters. Try three times to encode **abcd, beef, adch** respectively. The resulting three trees are displayed in the Figure 2.

These trees look differently, but they have some common characters. For example, the number of leaves in first two layers are two and the number of leaves in first four layers are at least four. All the encoded letters are at the top four layers. This is not strange, because the tree splay has the following properties:

- A new accessed leaf will move up half way.

- Any leaf will not move down more than two layers after a splay.

Therefore, in general, there are always leaves at the first four layers in a splay tree after some time. And most frequent letters will be set at higher level. This property will give

3
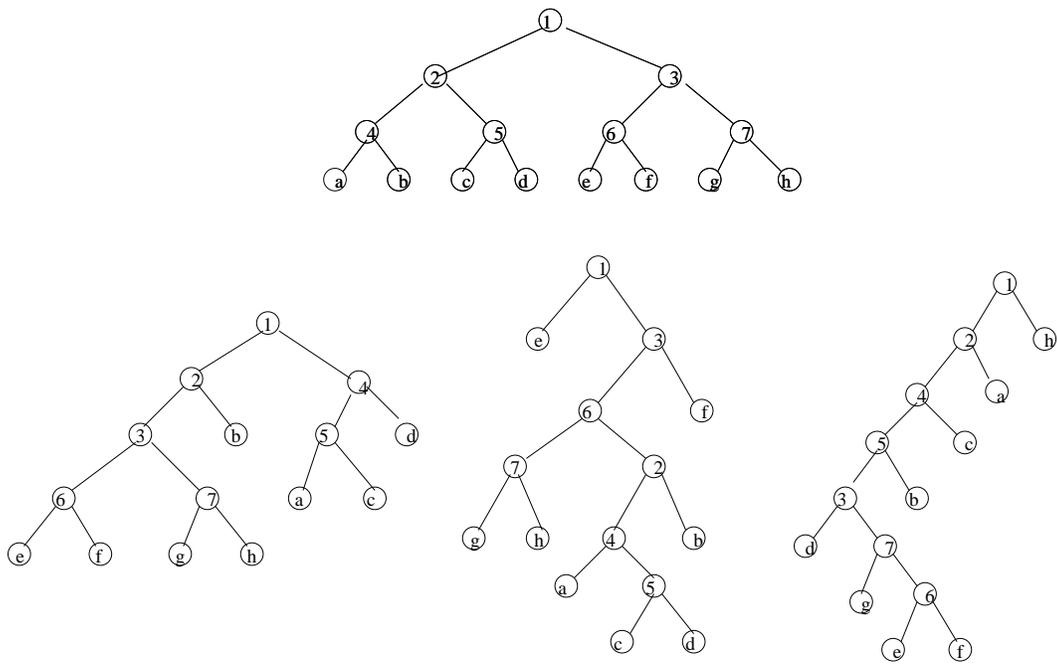
Figure 2: Small examples of splay

attackers information about the plain text. The random insertion to the input will not solve that problem, because it will not change the probability distribution of the plain text. For a secure encryption, we should keep the shape of the tree secret. In other words, we should keep the shape of the tree in random. Note that the tree is dynamic. It will change all the time during the encoding. We need to keep these changes in secret. In next subsection, we introduce our main idea about how to keep the tree in random and the changes in secret.

2.1 Moving inner nodes

From the above discussion we know that to insert random bytes to the inputs is not a very good idea. Because it will not change the fact that the tree is not balance (not random): one subtree of the root is vary small. We propose the following method called *key injection* for the encryption. Instead of insert random bytes (keys) into inputs, we will try to change the link of inner nodes using the keys. The main idea is as below.

1. Select an inner node n of the tree by using a key $k_i = f(K, P_i)$, where K is an initial symmetric key, P_i is the plain texts encrypted up to current block and f is some function.

2. Find out the node m in layer one, which is at the subtree of the root other than what is the node n at.

3. Exchange the link of n and m.

Here the node m could be an inner node or a leaf. For example, if the inner node 5 is chosen, then three trees at Figure 2 will be changed to trees at Figure 3.

4
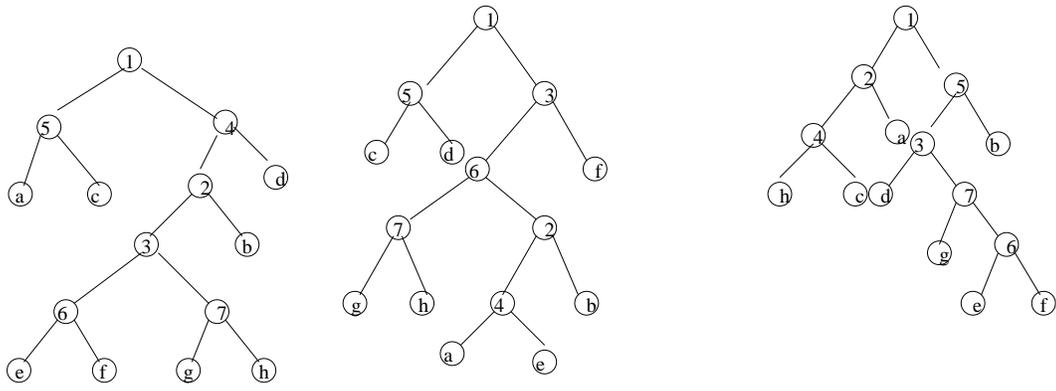
Figure 3: Small examples of key injection

In these examples, the leaves at the layer 1 disappeared after the key injection. The key injection increases the randomness of the tree. Of course, the key injection will increase the size of the cipher text. Therefore there is a trad-off between the security and compression in this algorithm. If we inject more keys, then the tree will be more random and the compression will be worse. To consider this, we will set some parameters and discuss these parameters later.

3   Encryption algorithm

Now we propose our encryption algorithm. Our method consists of tree parts which can be described as following three algorithms:

1. Key generation algorithm.

2. Encryption algorithm.

3. Key injection algorithm.

For simplicity, we will not describe the general algorithm. The following is an example of our encryption. But it is straightforward to give the general algorithm from our example.

Our encryption is based on bytes that enables it to be implemented easily in a 8-bit CPU. We will use a binary tree of 256 leaves for the 256 different bytes. So we need 255 inner nodes in the tree. We use three arrays `left, right, up` to store the nodes' information. Two arrays are of length 255 which records the left and right children of inner nodes. Other array is of length 511 which records the parents of nodes. In this tree, the inner nodes are named $0, 1, 2, \ldots, 254$ and the leaves are named $255, 256, \ldots, 510$. So the index of arrays are the names of the nodes. In the initial stage, the tree is a complete binary tree. So the values of these arrays are initialized as follows:

$\texttt{left}[i] = 2 * i + 1$; for $i = 0, \ldots, 254$.

$\texttt{right}[i] = 2 * i + 2$; for $i = 0, \ldots, 254$.

$\texttt{up}[i] = (i - 1)/2$; for $i = 1, \ldots, 510$, and $\texttt{up}[0]$ is some special symbol.

3.1 Key injection algorithm

As we described before, the key injection means to use a key to move inner nodes. On the other hand, we also want the cipher text compressed a little. After doing some tests, we found that the random injection might cause the algorithm losing the compression property. Actually, in many cases of our test, the cipher text is even larger than the plain text. To avoid that problem, we will only move the inner nodes which are above layer N, where N is some parameter. Since more frequent accessed bytes are usually at higher layers, this method will keep some compression of the cipher text. As we indicated previously, the compression of cipher text means some information about the plain text. But we can let the value of N large enough to eliminate the compression property of the cipher text if that is desired. Our method is as follows. After a key generated, the algorithm will check the layer of the according inner node. If an inner node is below layer N, then we will ignore that key. Using this method, some of the keys will be discarded in random. Therefore the injection is of some randomness, which will increase the security of the algorithm. In the following algorithms, we will use the parameter N.

In our method, we exchange the links of two nodes. So the injection is more efficient than splay. The discarding procedure gives a dynamic length of intervals and it is much more efficient than the run-length encoding mentioned in [4].

The algorithm of *injection function* `injection(byte i, integer N)` is as follows.

In our algorithms, a byte is expressed as an integer between 0 and 255.

Algorithm 3.1: INJECTION(i, N)

comment: i is a byte from input

$j \leftarrow i$
if $(i = 0)$
   then quit
$s \leftarrow 1$
$N \leftarrow n$
while $(up[j] \neq 0)$
$j \leftarrow up[j], s \leftarrow s + 1$
if $(s < N)$
       $\begin{cases} \textbf{if } (j = right[0]) \\ \quad \textbf{then } t \leftarrow left[0] \\ \quad \textbf{else } t \leftarrow right[0] \\ \text{exchange links of } i \text{ and } t \text{ (swap } up[i] \text{ and } up[t]) \end{cases}$
   then
  else quit

The injection function will be called at the beginning of the encryption. In that time, 16 bytes will be injected to the initial tree. After that, the injection function will be called for every byte of plain text encrypted. Note that in our algorithm, we use a byte as a block of plain text. In general, we can injection a key for each block. Since the initial tree is a complete binary tree, most of the first 16 bytes will be injected if we set $N \geq 7$. To avoid discarding any key in the beginning, we can set the threshold N after injecting the 16 keys. After encrypting some bytes, the keys will be discarded in random.

3.2 Encryption algorithm

Now we give our encryption algorithm which is essentially the splay tree algorithm of [3] plus key injection algorithm. First we need to define splay function. The algorithm of *Splay function* `Splay(byte i)` is as follows.

Algorithm 3.2: SPLAY(i)

comment: i is a byte from input

$i \leftarrow i + 255$
$j \leftarrow up[i]$
while $(j \neq 0$ and $up[j] \neq 0)$
$\begin{cases} s \leftarrow up[j] \\ \textbf{if } (j = left[s]) \\ \quad \textbf{then } t \leftarrow right[s] \\ \quad \textbf{else } t \leftarrow left[s] \\ \text{exchange links of } i \text{ and } t \\ i \leftarrow up[i] \\ j \leftarrow up[i] \end{cases}$

The encode is proceeded byte by byte. Since encoding is done by following a path from a leaf to the root of the tree, the code bits are produced in the reverse order from the order in which they should be transmitted. Therefore a local stack is used to temporarily store the bits. The algorithm of *encode function* `encode(byte i)` is as follows.

Algorithm 3.3: ENCODE(i)

comment: i is a byte from input

$j \leftarrow i$
$i \leftarrow i + 255$
while $(i \neq 0)$
$\begin{cases} \textbf{if } (i = left[up[i]]) \\ \quad \textbf{then } \text{push bit 0 to stack} \\ \quad \textbf{else } \text{push bit 1 to stack} \\ i \leftarrow up[i] \end{cases}$
while (stack is not empty) pop a bit and output it
Splay(j)

The algorithm first encode the byte. Then the accessed node is splayed.

To do encryption, we need a key sequence created by the key generation algorithm. Assume the key sequence is $(K_1, K_2, \ldots, K_i, \ldots)$, where $0 \leq K_i \leq 255$ for each i. The initial key is $K = (K_1, K_2, \ldots, K_{16})$

The algorithm of encryption is as follows. Suppose the plain text are bytes $P = (p_1, p_2, \ldots, p_m)$.

Algorithm 3.4: ENCRYPTION(P, K)

comment: P is plain text and K is key sequence

for $i = 1$ **to** 16
   do $injection(K_i, N)$
$key \leftarrow 17$
for $j = 1$ **to** m
   do $\begin{cases} encode(p_j) \\ injection(K_{key}, N) \\ key \leftarrow key + 1 \end{cases}$

3.3 Decryption algorithm

In decryption algorithms, the parameter N is same as that in encryption used. Suppose the cipher text is a bitstring $C = (c_1, c_2, \ldots, c_s)$. The decode is proceeded bit by bit following a path from the root to a leaf. A bit will be deleted from the bitstring after it is proceeded. The output of the function is a byte. The algorithm of *decode function* `decode(bitstring C)` is as follows.

Algorithm 3.5: DECODE(C)

comment: C is a bitstring from input

$node \leftarrow 0$
$c \leftarrow$ first bit of C
while $(node \leq 254)$
$\begin{cases} \textbf{if } (c = 0) \\ \quad \textbf{then } node \leftarrow left[node] \\ \quad \textbf{else } node \leftarrow right[node] \\ C = C \setminus \{c\} \\ c \leftarrow \text{ first bit of } C \end{cases}$
output $(node - 255)$
$Splay(node - 255)$

The algorithm of decryption is as follows. Suppose the cipher text is a bitstring C and key sequence is generated by K. The algorithm first injects 16 keys just as the encryption. Then the decode function is called. The key injection is also called for every decoding.

Algorithm 3.6: DECRYPTION(C, K)

comment: C is a bitstring and K is key sequence

for $i = 1$ **to** 16
   do $injection(K_i, N)$
$key \leftarrow 17$
$j \leftarrow 0$
while $(C \neq \emptyset)$ $\begin{cases} decode(C) \\ j \leftarrow j + 1 \\ injection(K_{key}, N) \\ key \leftarrow key + 1 \end{cases}$

3.4 Key generation algorithm

Now we consider the key generation algorithm. The key sequence is generated from initial key and plain text.

In the following, we suppose the symmetric key we used is 16 bytes. It is easy to use different lengths of keys in our algorithm.

We will use a "cyclic" array `key` with length 16 (or the length of the key). Here cyclic means that $key[j] = key[j - 16]$ for $j \geq 16$. These keys are stored in array key. Next, when a byte is encrypted, its parent is Xored to the current key and then used for injection. The key generation algorithm is as follows.

Algorithm 3.7: KEYGENERATION(P, K)

comment: P is plain text and K is initial key

for $i = 1$ **to** 16
 do $key(i) \leftarrow K_i$
$c \leftarrow 17$
for $j = 1$ **to** m
 do $\begin{cases} key(c) = key(c) \oplus up[p_j + 255] \\ \textbf{output } (key(c)) \\ c \leftarrow c + 1 \end{cases}$

The ith key is generated from the initial key and P_j, where P_j contains first j bytes of plain text. Note that because of the splay, the parent of the p_j is not fixed and depends on previous plain text and the initial key. Since one inner node may have two leaves as children, the key change is not determined by a unique plaintext byte.

4 Applications

We implemented the KIST as an Eclipse plug-in application (see [9]). The following methods are used.

- The hash function MD5 is used for the initial key so that a user can use a pass phrase as a key instead of a random key.

- Since ASCII codes used for Java source file only uses 127 nodes (bytes), we use an correspondence table for some frequent Java key words or phrases and the unused bytes. In this way, the Java source file can be compressed better.

- A user can choose options for encryption or compression only.

We also suggest to use KIST for wireless sensor networks (WSN) and other mobile devices, where space limitation is an important issue. A sensor typically has 8 - 120 KB of code memory and 512 - 4096 bytes of data memory (see [5]). KIST needs only 2 - 4 KB memory and the cipher text is compressed, that is good for the space limitation of nodes in WSN. By the same reason, the KIST is also suitable for small mobile devices.

5 Conclusion

In this paper, we introduce an encryption system KIST which is based on splay trees. An asynchronous key sequence is used to change the tree dynamically and secretly. Since a tree can be viewed as a substitution, the main encryption uses dynamic substitutions. The main operations of the encryption are byte xor and bytes swap. To decide some parameters, we did some tests and suggested better settings. This system has several advantages as listed in Section 1.

Some limitations of the system are:

- It has a bad error propagation. Even a bit error will cause catastrophic results. Although this is good for message integrity, it can not used in a noisy channel.

- The key sequence dose not have a resynchronizing method.

To handle above problems, we suggest that the tree refreshes after certain number of bytes has been encrypted and applies the initial key again.

We note that since this encryption is different from most existing stream ciphers and block ciphers, more new kind of security analysis should be developed for it. Although we did some chosen ciphertext attack to the algorithm, more serious security analysis should be done in the future. However, we think it is a good idea to publish this algorithm so that researchers may do more analysis and further improvements.

References

[1] R. Cole, On the Dynamic Finger Conjecture for Splay Trees. Part II: The Proof, SIAM Journal on Computing, 30(2000), 44-85.

[2] D. Grinberg, S. Rajagopalan, R. Venkatesan and V.K. Wei, Splay trees for data compression, Proc. 6th annual ACM-SIAM sympsium on Discrete algorithms (1995), 522-530.

[3] D.W. Jones, Application of splay trees to data compression, Communications of the ACM, 31(1998), 996-1007.

[4] D.W. Jone, Data compression and encryption algorithms, http://www.cs.uiowa.edu/~jones/compress/.

[5] Y.W. Law, J. Doumen and P. Hartel, Survey and benchmark of block ciphers for wireless sensor networks, ACM Transactions on Sensor Networks, 2(2006), 65-93.

[6] A.J. Menezes, P.C. van Oorschot and S.A. Vanstone, Handbook of Applied Cryptography, CRC Press, 1997.

[7] D.D. Sleator and R.E. Tarjan, Self-adjusting binary trees, Proc. ACM SIGACT Symposium on Theory of Computing, ACM New York (1983) 235-245.

[8] R.E. Tarjan and D.D. Sleator, Self-adjusting binary search trees, Journal of the ACM, 32(1985), 652-686.

[9] http://peace.lakeheadu.ca/kist.html

Appendix

Experimental results

Since our algorithm does not use iterations, the normal difference attack may not be suitable security analysis for it. In this paper, we just show some elementary analysis for the algorithm. We will do more security analysis in the future.

Parameter N

In the previous subsections we have given general algorithms of encryption and decryption which have a parameter N. To determine the value of N, we did some tests and the selected results are listed in Table 1.

FileType	N	size of P	size of C	compression rate	# of keys used	key rate
.java	7	1321	1093	82.74	226	17.11
.java	8	1321	1161	87.89	313	23.69
.java	9	1321	1291	97.73	467	35.35
.java	10	1321	1458	110.37	613	46.40
.java	8	9265	8365	90.29	2147	23.17
.java	9	9265	9470	102.21	3208	34.62
.tex	7	14636	11920	81.44	2185	14.93
.tex	8	14636	13113	89.59	3333	22.77
.tex	8	15744	14042	89.19	3714	23.59
.tex	9	15744	15343	97.45	5028	31.94
.doc	8	54784	26938	49.17	10963	20.01
.doc	9	54784	30643	55.93	14951	27.29
.doc	10	54784	34506	62.99	19466	35.53
.doc	11	54784	38041	69.44	24167	44.11
.doc	12	54784	40867	74.60	28279	51.62
.doc	8	55296	43949	79.48	13938	25.21
.doc	9	55296	47586	86.06	18580	33.60
.doc	10	55296	51025	92.28	23351	42.23
.tif	8	124644	58399	46.85	23724	19.03
.tif	9	124644	66142	53.06	32217	25.85
.tif	10	124644	73635	59.08	41779	33.52
.tif	11	124644	81121	65.08	51730	41.50
.tif	12	124644	87678	70.34	60896	48.86
.tif	13	124644	93087	74.68	69126	55.46
.tif	8	124644	50980	40.90	11464	9.20
.tif	9	124644	56805	45.57	16450	13.20

Table 1: Parameter test

In our tests, we tried to encrypt several text files such as java source codes, latex source codes. We then encrypted some Microsoft words files. We also tried some picture with .tif format. In Table 1, we listed some results for these different types of files. For different values of N, we listed the sizes of plain texts and cipher texts (in bytes). The column "compression rate" is the percentage value of (size of cipher text) /(size of plain text), the

column "key rate" is the percentage value of (number of keys injected) /(size of plain text). We want the compression rate smaller so that the cipher text is compressed. We also do not want the key rate too small so that too many keys are discarded.

For the security reason, the key rate cannot be too small. We want the key rate bigger than 0.20. In this way, in average a key will be injected for each 5 bytes of plain text. So we first look at the key rate, then consider the value of compression rate. By our tests, for the text files (.java and .tex) $N = 8$ is always better. The cipher text will be compressed about 10%.

For the word files, $N = 8$ is still fine for the key rate while the compression is better. Using $N = 9$ will inject more keys and the cipher text can still be compressed about $10\% - 40\%$.

For the .tif file, we can use $N = 9$ and the cipher text can be compressed $40\% - 50\%$, although using $N = 8$ is still having the key rate around 20%.

In summery, we suggest to use $N = 8$ in general. But we can leave N as a parameter of the scheme. So if we want the scheme more secure then we can use bigger N and if compression is more important, then we can use smaller N.

We should indicate that this algorithm will not compress for some kind of files, such as .gif files, MP3 files etc. For these files, the cipher text is always larger than plain text. Basically, if a file is not good for splay tree compression, then it is not good for this encryption in a view of the size of the cipher text.

Randomness of cipher test

We did some randomness tests for the various cipher texts of the proposed encryption. Some basic statistical tests of FIPS 140-1 (see [6]) are done. The results are as follows.

All the cipher texts passed the frequency test (monobit test) and the Porker test. All of the cipher texts passed the runs test and the long run test.

Randomness of key sequence

For testing the key generation algorithm, we recorded the key sequences when we encrypt several plain texts. We use integers from 0 to 255 to denote the bytes. In the following records, $a : b$ means byte a appears b times in the key sequence.

0:4 1:4 2:4 3:2 4:5 5:8 6:7 7:7 8:5 9:3 10:3 11:7 12:0 13:3 14:1 15:6 16:5 17:8 18:1 19:5 20:3 21:6 22:4 23:8 24:4 25:8 26:9 27:3 28:8 29:5 30:5 31:5 32:1 33:5 34:2 35:2 36:6 37:4 38:4 39:4 40:7 41:5 42:4 43:2 44:3 45:7 46:4 47:12 48:3 49:3 50:6 51:7 52:3 53:5 54:5 55:6 56:7 57:5 58:6 59:3 60:7 61:5 62:7 63:4 64:6 65:7 66:9 67:2 68:5 69:6 70:6 71:5 72:5 73:3 74:4 75:4 76:2 77:7 78:2 79:7 80:5 81:5 82:4 83:8 84:8 85:4 86:5 87:7 88:5 89:5 90:3 91:6 92:2 93:6 94:6 95:2 96:7 97:11 98:4 99:9 100:11 101:9 102:3 103:3 104:7 105:7 106:7 107:3 108:7 109:5 110:8 111:2 112:3 113:5 114:3 115:5 116:7 117:5 118:2 119:6 120:3 121:4 122:3 123:7 124:5 125:7 126:8 127:4 128:4 129:2 130:8 131:2 132:5 133:4 134:3 135:6 136:2 137:5 138:1 139:5 140:8 141:6 142:2 143:2 144:7 145:4 146:4 147:6 148:5 149:8 150:8 151:8 152:4 153:9 154:6 155:4 156:3 157:3 158:10 159:5 160:4 161:4 162:7 163:4 164:4 165:8 166:4 167:3 168:5 169:4 170:5 171:7 172:6 173:10 174:4 175:4 176:3 177:6 178:3 179:4 180:3 181:4 182:5 183:9 184:7 185:5 186:6 187:4 188:6 189:2 190:5 191:6 192:4 193:8 194:5 195:8 196:2 197:6 198:7 199:11 200:6 201:1 202:5 203:5 204:6 205:5 206:6 207:4 208:6 209:2 210:1 211:6 212:5 213:5 214:1 215:5 216:4 217:6 218:3 219:3 220:5 221:4 222:4 223:5 224:6 225:9 226:4 227:8 228:8 229:7 230:7

231:6 232:4 233:7 234:7 235:5 236:7 237:6 238:8 239:5 240:5 241:9 242:4 243:6 244:8 245:5 246:9 247:6 248:6 249:4 250:5 251:6 252:5 253:2 254:4 255:8

This key sequence is of 1321 keys. Only one byte (12) appears 0 times. one bytes appear 12 times, 2 bytes appear 10 times, 3 bytes appears 11 times. Others appear 1 to 9 times. The average is $\frac{1321}{256} = 5.16$

Another example of key sequence is of 54784 keys displayed below.

0:330 1:228 2:182 3:136 4:489 5:209 6:162 7:221 8:302 9:167 10:211 11:193 12:180 13:640 14:176 15:135 16:188 17:188 18:176 19:280 20:151 21:496 22:209 23:164 24:125 25:168 26:193 27:541 28:160 29:180 30:174 31:142 32:400 33:187 34:261 35:215 36:165 37:260 38:404 39:221 40:395 41:117 42:171 43:162 44:188 45:121 46:166 47:189 48:224 49:187 50:219 51:181 52:173 53:172 54:154 55:134 56:175 57:198 58:202 59:158 60:215 61:168 62:195 63:222 64:180 65:154 66:178 67:185 68:217 69:143 70:200 71:170 72:191 73:148 74:195 75:222 76:198 77:197 78:158 79:318 80:189 81:158 82:192 83:260 84:326 85:191 86:129 87:425 88:263 89:161 90:158 91:186 92:158 93:201 94:174 95:225 96:169 97:170 98:214 99:182 100:194 101:193 102:198 103:158 104:174 105:180 106:439 107:394 108:192 109:157 110:155 111:257 112:222 113:154 114:124 115:158 116:218 117:171 118:143 119:178 120:190 121:161 122:239 123:170 124:176 125:170 126:178 127:174 128:149 129:214 130:164 131:178 132:272 133:223 134:173 135:181 136:167 137:214 138:380 139:162 140:187 141:243 142:216 143:270 144:202 145:147 146:132 147:208 148:178 149:169 150:219 151:430 152:167 153:225 154:186 155:192 156:147 157:202 158:239 159:202 160:178 161:207 162:172 163:176 164:426 165:200 166:174 167:191 168:204 169:258 170:210 171:393 172:235 173:153 174:172 175:159 176:291 177:166 178:449 179:131 180:399 181:292 182:182 183:173 184:158 185:202 186:220 187:644 188:185 189:231 190:245 191:184 192:271 193:510 194:139 195:227 196:303 197:200 198:173 199:183 200:409 201:162 202:285 203:148 204:144 205:159 206:187 207:181 208:148 209:518 210:243 211:174 212:177 213:198 214:188 215:229 216:419 217:149 218:385 219:156 220:457 221:181 222:147 223:176 224:193 225:181 226:197 227:195 228:189 229:214 230:165 231:168 232:200 233:199 234:205 235:181 236:137 237:211 238:180 239:200 240:160 241:260 242:143 243:162 244:203 245:188 246:172 247:155 248:204 249:181 250:333 251:184 252:188 253:177 254:168 255:170

When we use different keys for the same file, the key sequences are different. But their distributions are similar. One example of the same plain text but different keys is as follows. The same plain text (54789 bytes) using different 16 keys. The key sequence is

0:278 1:173 2:178 3:230 4:178 5:158 6:180 7:183 8:168 9:238 10:180 11:221 12:192 13:661 14:156 15:197 16:177 17:158 18:277 19:190 20:263 21:276 22:188 23:189 24:205 25:437 26:186 27:269 28:202 29:209 30:253 31:213 32:435 33:280 34:149 35:217 36:152 37:157 38:166 39:194 40:159 41:420 42:160 43:253 44:150 45:404 46:429 47:262 48:156 49:190 50:180 51:197 52:169 53:200 54:203 55:144 56:211 57:251 58:199 59:185 60:185 61:205 62:193 63:166 64:339 65:190 66:171 67:169 68:228 69:195 70:437 71:432 72:179 73:188 74:175 75:206 76:147 77:138 78:417 79:145 80:159 81:203 82:292 83:159 84:197 85:166 86:661 87:238 88:167 89:225 90:166 91:131 92:147 93:203 94:195 95:156 96:180 97:313 98:255 99:436 100:216 101:215 102:168 103:186 104:199 105:256 106:235 107:210 108:158 109:219 110:161 111:329 112:189 113:206 114:220 115:236 116:268 117:181 118:188 119:170 120:176 121:173 122:236 123:133 124:173 125:207 126:241 127:303 128:239 129:187 130:164 131:145 132:129 133:167 134:180 135:204 136:153 137:212 138:155 139:216 140:166 141:233 142:176 143:231 144:223 145:133 146:201 147:154 148:181 149:189 150:148 151:174 152:164 153:179 154:352 155:291 156:176 157:205 158:225 159:178 160:295 161:157 162:246 163:133 164:143 165:156 166:432 167:229 168:211 169:195

13

170:157 171:207 172:193 173:172 174:214 175:198 176:181 177:171 178:181 179:135 180:138
181:196 182:426 183:161 184:164 185:196 186:394 187:170 188:216 189:174 190:187 191:145
192:181 193:177 194:205 195:183 196:368 197:369 198:151 199:203 200:201 201:181 202:152
203:206 204:150 205:155 206:163 207:165 208:121 209:199 210:162 211:262 212:145 213:166
214:153 215:140 216:202 217:168 218:140 219:164 220:259 221:178 222:219 223:171 224:142
225:167 226:315 227:178 228:416 229:180 230:196 231:412 232:142 233:305 234:282 235:169
236:173 237:175 238:258 239:187 240:200 241:166 242:377 243:384 244:149 245:154 246:179
247:243 248:157 249:157 250:413 251:223 252:392 253:267 254:168 255:155

Key discarding

We also tested the behavior of the key discarding. We define the key discarding sequence as
follows. A number 0 means a key is injected. The positive integers in the sequence denotes
the number of successive keys discarded. The sequence {0 1 0 0 3 . . .} means that after first
key injected, one key is discarded. Then two keys are injected and 3 keys are discarded,
etc.

We give two examples below using $N = 8$. One example is for a Java source file (1321
bytes). Part of the sequence is:

0 3 0 2 0 4 0 1 0 0 1 0 3 0 5 0 5 0 1 0 2 0 2 0 14 0 0 1 0 4 0 6 0 0 3 0 2 0 2 0 1 0 0 4 0 13 0
4 0 2 0 1 0 5 0 7 0 2 0 11 0 1 0 0 2 0 0 6 0 0 3 0 1 0 1 0 0 4 0 1 0 0 12 0 11 0 3 0 0 6 0 6 0 4
0 0 15 0 2 0 3 0 4 0 0 4 0 0 0 11 0 0 0 1 0 4 0 2 0 7 0 1 0 2 0 3 0 1 0 2 0 4 0 2 0 3 0 7 0 0 2
0 0 5 0 3 0 7 0 4 0 6 0 0 0 7 0 1 0 1 0 2 0 1 0 5 0 4 0 0 0 4 0 0 11 0 1 0 0 3 0 0 0 1 0 4 0 4 0
0 0 8 0 6 0 0 3 0 3 0 3 0 3 0 0 0 5 0 0 2 0 5 0 7 0 1 0 1 0 3 0 0 0 2 0 1 0 2 0 4 0 2 0 5 0 0 1 0
2 0 0 12 0 6 0 0 2 0 14 0 0 1 0 11 0 1 0 6 0 6 0 9 0 7 0 1 0 0 10 0 2 0 1 0 0 1 0 5 0 2 0 3 0 4
0 1 0 10 0 0 3 0 3 0 1 0 4 0 4 0 6 0 8 0 11 0 0 0 1 0 7 0 1 0 1 0 5 0 4 0 1 0 3 0 4 0 8 0 2 0 13
0 2 0 7 0 6 0 2 0 2 0 5 0 4 0 4 0 0 2 0 6 0 6 0 3 0 6 0 5 0 1 0 1 0 0 4 0 0 9 0 8 0 0 2 0 2 0 8 0
0 5 0 1 0 3 0 2 0 2 0 0 4 0 1 0 9 0 5 0 2 0 18 0 1 0 5 0 9 0 1 0 15 0 2 0 4 0 3 0 2 0 1 0 4 0 2
0 15 0 3 0 3 0 0 3 0 10 0 2 0 1 0 1 0 . . . . . .

The maximum number of successive 0's in the file is 3. If we use $a : b$ to denote that
number of occurrence of a is b, then we have:

1:53 2:43 3:32 4:33 5:23 6:16 7:12 8:7 9:4 10:3 11:6 12:3 13:2 14:2 15:3 17:1 18:1

Another example is for a Microsoft word file (54784 bytes) displayed below.

Part of the sequence is:

0 0 1 0 2 0 6 0 0 6 0 14 0 1 0 4 0 0 1 0 2 0 2 0 4 0 0 0 3 0 0 3 0 0 5 0 2 0 0 4 0 2 0 11 0 15
0 11 0 13 0 1 0 12 0 1 0 0 5 0 11 0 0 9 0 5 0 0 7 0 5 0 12 0 4 0 6 0 1 0 8 0 7 0 40 0 22 0 8 0
0 0 0 0 6 0 0 0 11 0 0 1 0 3 0 8 0 2 0 0 2 0 0 0 4 0 28 0 10 0 0 13 0 11 0 0 3 0 0 15 0 2 0
19 0 3 0 0 4 0 9 0 1 0 2 0 0 0 2 0 15 0 3 0 1 0 2 0 0 1 0 7 0 7 0 3 0 1 0 1 0 2 0 0 0 0 3 0 1 0
3 0 1 0 4 0 3 0 8 0 2 0 6 0 4 0 6 0 1 0 1 0 6 0 0 1 0 4 0 0 4 0 2 0 0 0 0 5 0 4 0 11 0 3 0 12 0
6 0 2 0 7 0 1 0 1 0 2 0 3 0 10 0 5 0 3 0 1 0 8 0 3 0 4 0 0 1 0 4 0 3 0 0 2 0 1 0 2 0 1 0 3 0 3 0
8 0 1 0 0 3 0 2 0 6 0 0 2 0 10 0 15 0 0 3 0 1 0 2 0 1 0 7 0 0 2 0 10 0 3 0 4 0 4 0 4 0 1 0 0 1 0
5 0 10 0 4 0 5 0 0 2 0 2 0 21 0 0 0 4 0 0 2 0 0 5 0 7 0 4 0 0 26 0 0 3 0 3 0 0 3 0 0 0 4 0 7 0 6
0 2 0 1 0 2 0 1 0 11 0 0 0 0 0 3 0 0 1 0 1 0 4 0 0 3 0 7 0 7 0 3 0 1 0 5 0 0 1 0 7 0 3 0 3 0 10
0 0 2 . . . . . .

The maximum number of successive 0's in the sequence is 7. And we have:

1:1818 2:1345 3:1065 4:802 5:562 6:543 7:395 8:339 9:256 10:198 11:150 12:123 13:111 14:107 15:73 16:63 17:52 18:56 19:31 20:36 21:31 22:22 23:25 24:20 25:11 26:14 27:8 28:11 29:7 30:3 31:6 32:12 33:5 34:4 35:4 36:1 37:7 38:3 39:2 40:2 41:2 42:1 43:3 44:1 46:1 48:1 49:1 50:1 53:1 54:1 55:1 58:1

The tests have shown some randomness of the key discarding. The key discarding depends on the value of N, key sequence and the plain text. We need to do more testing on this direction in the future.

Chosen Plaintext Attacks

We did some tests for simple chosen plaintext attacks. We give three examples when $N = 4$ using the same key.

Plaintext	Output stream
aaaa	38 94 80
aaaaa	3c 4a 40
aaaaaaaaaaaaaaaaaaaa	3f ff f8 94 80

The output steams are totally different. We note that if N were not introduced, these output streams will be similar except the last couple of bits.

In another test, we let the plaintext consists of "bbbb···". Two different keys are used and the first 20 bytes of ciphertext are as follows:

e1 d9 41 c7 21 f9 7f 94 0b 80 98 26 02 60 98 09 82 60 26 09
51 8a e0 ff 87 47 ff 65 a0 01 35 c0 05 d7 00 17 5c 00 5d 70