International Journal of Applied Engineering Research ISSN 0973-4562 Volume 10, Number 14 (2015) pp 34068-34071

Dynamic Key Cryptography in Sharing Medicine

A. Manimaran[1,*], V. M. Chandrasekaran[2], Archit[3]

School of Advanced Sciences,VIT University, Vellore-632014[1,2]
School of Computing Sciences and Engineering,VIT University, Vellore-632014[3]
marans2011@gmail.com[1],vmcsn@yahoo.co.in[2], archit5793@gmail.com[3]
Note:-Corresponding Author*

Abstract
In this paper, we are discussing concept of two-key and four layered cryptography. There are many cryptographic methods which are evolving and their need is increasing every day. Always there has been a challenge in designing more robust and reliable techniques. The paper briefly explains the possible use of a more layered and feasible cryptographic technique and the need of two key practice.

Keywords Encryption, decode, key, algorithm, salting, hash, static

Introduction

One of the greatest challenge posing in today's techniques is the static cryptographic code. The static cryptographic code here refers to the encrypted code generated by using several algorithms which is in a way static or non changing. This encryption can be eventually decoded by using various algorithms.

Currently there are various encryption algorithm like 'sha1','md5' which are also known as hash functions as they generate a hash of string of specific lengths. These hashes are mostly static as they contain repeated strings and also in few there may be a common appended string at the beginning. The authors Chandrasekaran et al.[4],discussed about cryptography concept using pair of dice. In this paper they considered sample space of two dice and converted binary to decimal and vice versa for giving the concepts in detail.

To overcome the similar string decoding which can be done using various complex algorithms usually salting is done. Salting refers to adding another hash to the previously generated hash which makes the string even more secure and complex to break. But even by adding this extra salt the final hash remains static and can be guessed if all the strings are similar.

Hence we are here trying to focus on the encryption that is dynamic or that keeps on changing with time. Also by providing it with two keys and four layers of encryption it is more secure and robust. By keeping our pace with the current algorithms we are incorporating simple concepts to visually create a better algorithm of code generation.

Use of Tree for Input

The basic concept behind the input is using a binary tree to store initial values.

Step 1: we take the values or the message.

Step 2: we split this string of message into an array.

Step 3: we generate the binary tree using array inputs.

General Algorithm goes as follows:

Input message: message(m)(type:string)

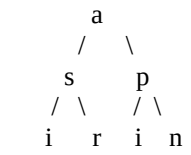
arr(type:array) = split(m) to array

```
split(m){
        index(type:integer)
          for index=0 until m[index] = null
                do
                arr[index]=m[index]
                index = index+1
           end for

      }

toTree(arr){
                step1:insertToRoot(arr[0]) index = index +1
                step2:insertToLeft(arr[index])  index  =
index+1
                step3:insertToRight(arr[index])
                repeat step2 and step3
          }
```

Example:
message(m) = "aspirin"
split(m)
arr = 'a','s','p','i','r','i','n'
toTree(arr)



Using ASCII Table to Assign Values

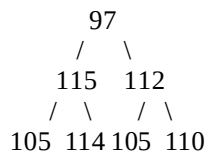
Here we are assigning values corresponding to alphabets in the tree using ASCII table.

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

```
assignValue(arr){
                    for index = 0 to arr[index] = null
                        do
                        arr[index] = asciiValue
                        index = index + 1
                    end for
                }
```

Example:
assignValue(arr)
a=97
s=115
p=112
i=105
r=114
i=105
n=110



Code Generation Using Subtraction

Here all the leftmost nodes and root node values are taken intact and all values to their right on the same level are subtracted to generate the base code.

```
baseCodeGenerator(tree){
                        value[](type:array)
                        value[index]=rootNode index = index+1
                        repeat until null
value[index]=leftmostNode(current) index = index +1
value[index]=leftmostNode(next)-rightNode(next) index = index +1
                        end repeat
                    }
```

Example:value array
97 | 115 | 115-112 | 105 | 105-114 | 105-105 | 105-110

97 | 115 | 3 | 105 | -9 | 0 | -5 |

Binary Conversion and First Key Identification

Here all the values are divided by two to get the remainder and quotient. Concatenate values to get the key and code.

```
binaryConv(value[]){
                    quotient[index](type:array)
                    remainder[index](type:array)
                    index=0
                    repeat
                    quotient[index]=value[index]/2
                    remainder[index]=value[index]%2
                    index = index+1
                    until value[index] = null
                    }
valueConcat(value[index]){
                        if(value[index]!=treeNode && value[index]<0)
                            {
        key=concat(quotient)
        code=concat(remainder)
                            }
else{
if(value[index]<0) append 00 to left of quotient[index] & remainder[index]
if(value[index]=treeNode) append 0000 to right of quotient[index] & remainder[index]
}
                        }
```

Example:
97/2 | 115/2 | 3/2 | 105/2 | -9/2 | 0/2 | -5/2 quotient array only integral calculation

48 | 57 | 1 | 52 | -4 | 0 | -2 quotient array

97%2 | 115%2 | 3%2 | 105%2 | -9%2 | 0%2 | -5%2 remainder array

1 | 1 | 1 | 1 | -1 | 0 | -1 remainder array

key = 48000057000010000520000004000000002

code = 10000100001000010000001000000000001

Adding Time Stamps

Here we will add the time stamps to the code and take the current timestamp as another key.

```
addTime(){
        length= getLength(code)
        time=
concat(currentTimeStamp,currentTimeStamp) until

    getLength(time)==getLength(code)
        newTime = code + time
    }
```

Example:

code= 10000100001000010000001000000000001

currentTimeStamp=25:50:18:19:2:2015(sec:min:hrs:date:month:year)

length=35

time=25501819220152550181922015255018192

addTime()

/**********adding time**********

/ 10000100001000010000001000000000001

/ + 25501819220152550181922015255018192

/————————————————————————————

/ 35501919221152560181923015255018193

**********/

Dynamic Code Generation

Here we add timeStamp for every second and subtract the previous timeStamp every second.

```
for every second do
dynamicTime(newTime){

        length= getLength(code)
        time=
concat(currentTimeStamp,currentTimeStamp) until

    getLength(time)==getLength(code)
        prevTimeStamp=time
        newtTme = code + time -
prevTimeStamp

        }
end for
```

Example:

```
dynamicTime()
newCode=    35501919221152560181923015255018193
```

say after one min:

```
 25511819220152551181922015255118192
+35501919221152560181923015255018193
newCode=610137384413051113638450305101363 85
```

Hence the message keeps changing every second.

Decoding to Original Message

```
 decode(){
        baseCode=codeCurrent - timeKey -
currentTimeStamp
        baseCodeArray=split(baseCode,0000) //split
using 0000 as pivot
        baseCodeArray=replace(00,-)//replace array
items 00 with -
        keyArray=split(key,0000)//split using 0000
as pivot
        keyArray=replace(00,-)//replace array items
00 with -
        keyArray=keyArray*2
        array=keyArray+baseCodeArray
        toTree()
        baseCodeGeneratorAdd(tree){
            value[](type:array)
            value[index]=rootNode index =
index+1
            repeat until null

    value[index]=leftmostNode(current) index = index
+1

    value[index]=leftmostNode(next)+rightNode(next)
index = index +1
            end repeat

        }
        message=takeTreeValues()=assign ASCII
value

    }
```

Example:

key = 48000057000010000520000004000000002

timeKey= 25501819220152550181922015255018192

codeCurrent=610137384413051113638450305101363 85

currentTimeStamp=25511819220152551181922015255118192

baseCode = codeCurrent - timeKey - currentTimeStamp

baseCode = 10000100001000010000001000000000001

baseCode=split(baseCode,0000)

baseCodeArray=1,1,1,1,001,0,001

baseCodeArray=replace(00,-)

baseCodeArray=1,1,1,1,-1,0,-1

keyArray=split(key,0000)

keyArray=48,57,1,52,004,0,002

keyArray=replace(00,-)

keyArray=48,57,1,52,-4,0,-2

keyArray=keyArray*2

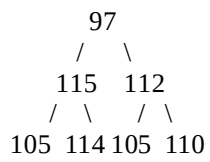
keyArray=96,114,2,104,-8,0,-4

array=keyArray+baseCodeArray

array=97,115,3,105,-9,1,-5

toTree()

baseCodeGeneratorAdd()



takeTreeValues()=assign ASCII value

message=a,s,p,i,r,i,n

Conclusion

In this paper we have devised a new encryption method. This method consists of dynamic code generation using tree and other simple concepts of timestamps addition. We here used binary tree in its most primitive form and the timestamps are taken in generic UTC format. The robustness of the encrypted code depends upon the message size. To keep the code generation simple we used ASCII values for assignment and simple binary implementation. We tried to keep the algorithm simple and easy to implement by not giving more complexity to it and keeping in mind that the final code generated should not be static.

References

1. Wikipedia-Binary Tree,ASCII Values
2. ASCII- http://upload.wikimedia.org/wikipedia/commons/thumb/7/7b/Ascii_Table-nocolor.svg/1000px-Ascii_Table-nocolor.svg.png
3. Data Structure and Algorithms By Weiss,Pearson Publications
4. V.M.Chandrasekaran,A. Manimaran,Akhil Ranjan,Cryptography Using a Pair of Dice,International Journal of PharmTech Research,7(1)(2015),85-89.