Researcher Daniel HOMOCIANU, PhD
E-mail: daniel.homocianu@feaa.uaic.ro
Professor Dinu AIRINEI, PhD
E-mail: adinu@uaic.ro
"Alexandru Ioan Cuza" University (UAIC)
Faculty of Economics and Business Administration (FEAA)

64B3AC - *SY*MMETRIC *C*RYPTO-*S*YSTEM *FOR B*USINESS USING THE *A*SCII *A*LPHABET AND *A*LGORITHM *C*ODES *C*ALLED IN *C*ASCADE

Abstract. *The paper introduces a cryptographic technique useful to encipher the content of secret business reports and a corresponding prototype called 64B3AC (SYmmetric Crypto-System for Business using the ASCII Alphabet and Algorithm Codes Called in Cascade), developed and tested under a data oriented environment. References to a special interactive tutorial presenting the logic and functionality of this technique are also included. Historical elements starting from initial ideas, challenges and applications up to potential improvements of the current version were considered too. In order to offer an honest image concerning the performance of this technique the paper includes the results of two hundred successful tests focused on processing speed, input and output as occupied space and number of total files generated or used along the encryption or decryption processes (one hundred for each way). Another goal of the paper was to underline the possibilities of the symmetric encryption with many advantages when combined to other techniques.*

Keywords: *symmetric cryptography, multiple / cascade encryption, development environment, a posteriori correction, increased randomness.*

JEL Classification: **D82, D83, L86**

1. Introduction

Nowadays when the approach of security issues is made in terms of standards and proven models, the effort to come with something new and better is considerably harder and must consider the hardware development, the multitude of cryptanalytic techniques and types of attacks but still a compromise between strength and complexity on one side and resource consumption on the other.

The technique we have developed contains a combination of many well-known operations on strings and can be extended to a larger number of basic algorithms (ALGs) easy to adapt for an on-line solution.

The original motivation was to solve in a simple, original and functional way the problem of encrypting passwords to store in a table of a relational data base. That moment we have thought to do the basic operations on DETs (DEcimal digiTs) and ASCs (ASCII characters) and not in binary, considering that cryptography for card readers and ATMs was not our goal. Although we came to the point of operating with vectors/arrays of strings in long processing sessions including hibernation states of the operating system (OS) in some preliminary tests, we did a lot of optimizations in order to reduce the processing time for reasonable limits of input parameters as seen in those two hundreds tests (tables 1, 2 and 3). The cryptographic prototype we have obtained (Fig.3) is currently more suitable for protecting large blocks of text than passwords before storage and can also serve as a benchmark tool for one core of a central processing unit running a Windows type OS for personal computers.

The highlights of the paper are: the algorithms including ASCII compressions and expansions, ballast insertions, transpositions and statistical corrections applied in cascade and currently coded as digits (one digit for each algorithm) in a random symmetric key; a generator of a priori keys and premises of pure randomization support in algorithms; a posteriori keys made of encrypted markers for substitutions also serving as strength levels; useful results for deriving estimation functions; further considerations regarding the use of NVidia's Computer Unified Device Architecture (CUDA) or cloud resources.

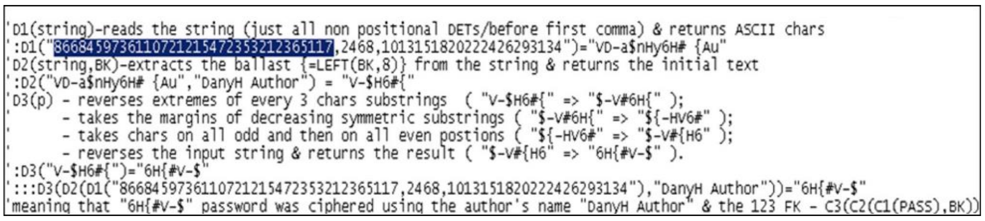
2. Short history of 64B3AC

The first version of this crypto-system was intended to protect the content of the values for the password field in a table of a database about quizzes and their authors (Fig.1) that have full rights granted with password-based access.

An example of Visual Basic (VB) 6.0 source code fragments behind the 1st version of the crypto-system based on three ALGs initially used to cipher (C ALGs) and decrypt (D ALGs) the values of a password field is presented below:

```
...
  Begin VB.TextBox
txtNumCiphPASS
        DataField = “Pass”
        DataSource = “Aut” ‘Authors
  ...
  End
  Begin VB.TextBox txtBallastKEY
        DataField = “Name”
        DataSource = “Aut”
  ...
  End
  Begin VB.Data Aut
        Caption = “Authors”
        Connect = “Access”
        DatabaseName =
“D:\QDB\quizzes.mdb”
        ...
        ReadOnly = 0 ‘False
        RecordsetType = 1 ‘Dynaset
        RecordSource = “Aut”
        ...
  End
...
```

```
Private Sub
cmdButtonBrowseAut_Click(Index As
Integer)
  Dim DeCiphPASS as String
  With Aut.Recordset
     Select Case Index
      Case 0
      If Not (.Bof) Then
         .MoveFirst
      End If
      Case 1
      If Not (.Bof) Then
         .MovePrevious
         If .BOF Then
            .MoveFirst
         End If
      End If
      ...
     End Select
  End With
   ...
  If Not (Aut.RecordSet.EOF) And Not
(Aut.RecordSet.BOF) Then
    DeCiphPASS =
D1(D2(D3(formCreateAut.txtNumCiphPA
SS), formCreateAut.txtBallastKEY))
  End If
End Sub
```



```
'D1(string)-reads the string (just all non positional DETs/before first comma) & returns ASCII chars
':D1("86684597361107212154723532123651117,2468,10131518202224262931 34")="VD-a$nHy6H# {Au"
'D2(string,BK)-extracts the ballast {=LEFT(BK,8)} from the string & returns the initial text
':D2("VD-a$nHy6H# {Au","DanyH Author") = "V-$H6#{"
'D3(p) - reverses extremes of every 3 chars substrings  ( "V-$H6#{" => "$-V#6H{" );
'       - takes the margins of decreasing symmetric substrings ( "$-V#6H{" => "${-HV6#" );
'       - takes chars on all odd and then on all even postions ( "${-HV6#" => "$-V#{H6" );
'       - reverses the input string & returns the result ( "$-V#{H6" => "6H{#V-$" ).
':D3("V-$H6#{")="6H{#V-$"
':::D3(D2(D1("86684597361107212154723532123651117,2468,1013151820222426293134"),"DanyH Author"))="6H{#V-$"
'meaning that "6H{#V-$" password was ciphered using the author's name "DanyH Author" & the 123 FK - C3(C2(C1(PASS),BK))
```

Figure 1. Examples / explanations for 1st version of VB ALGs to decrypt (D)
Source: sites.google.com/site/supp4for64b3ac/downloads/fig1.tiff

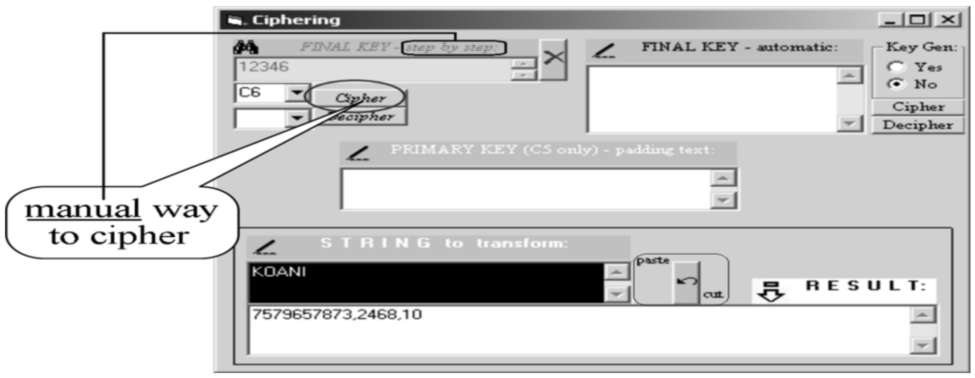
That time the symmetric final key (FK) was set in an incipient inadequate way by the code’s programmer as a predetermined set of consecutive / in cascade references to the corresponding encryption ALGs following a simple principle: the output of the previous ALG (call) is input for the next one. For instance, C3 (C2

(C1 (input))) means “123” as a symmetric FK used both for encryption (as it is) and decryption (in reverse order: 321 - see Fig.1).

In order to get to what we consider to be the 2nd version we have defined six atomic ALGs: the old C3/D3 (Fig.1) was split into four pieces (C1-4/D1-4 - the code sequence before Fig.2), the old C2/D2 became C5/D5 and the old C1/D1 was renamed to C6/D6.

An example of VB6.0 source code sequence behind the 2nd version of the crypto-system based on six ALGs mainly used for encrypting text blocks and implementing the cascade principle by using a symmetric FK (just 1-6 digits) is:

```
Public Function C(IS As String, BK As
String, FK As String)
‘IS   ->  Intermediary String
‘BK   ->  Ballast Key / Padding Text
‘FK   ->  Final Key giving the cascade itself
Dim STBE As String
‘STBE ->  StringToBeExec
Dim i As Long
STBE=IS
For i=1 To Len(FK)
 Select Case Mid(FK,i,1)
  Case 1
   STBE=C1(STBE)
  Case 2
   STBE=C2(STBE)
  Case 3
   STBE=C3(STBE)
  Case 4
   STBE=C4(STBE)
  Case 5
   STBE=C5(STBE, BK)
  Case 6
   STBE=C6(STBE)
  Case Else
   MsgBox “FK Probl.:1-6 required”,
vbCritical + vbOkOnly
   STBE=””
   i=Len(FK)+1 ‘Get out of the For loop
 End Select
Next i
C=STBE
End Function
```



12346

C6

KOANI

7579657873,2468,10

Figure 2. The interface of the 3rd version of the crypto-system – example of step-by step encryption using the cut & paste option (the bottom side button)
Source: sites.google.com/site/supp4for64b3ac/downloads/fig2.tiff

Next (3rd version) we have developed a specialized interface able to automatically translate (fig.2) a succession of manual calls (ten DETs for ten ALGs) into a symmetric FK (DETs). Thus the need for four more transpositions essentially based on reversing extremes of substrings of prime numbers length (5, 7, 11, etc.). This version brought most of the same snags: still limited output alphabet (DETs and commas) because of the only one way substitution defined that time - ASCII Expansion (AEXP) transforming ASCII strings to DECIMAL codes plus commas and positional codes (fig.1) as in the 2nd version; lack of a generator of FKs; no APOKs (A Posteriori/Feed-Back Keys) and consequently a rapidly growing Intermediate Text Block (ITB) as input/output of encryption ALGs running in cascade and a large final encrypted block (Cipher Text - CT); up to 200 DETs FK as maximum number of cascade levels in order to avoid putting the OS in hibernate states and complete the tests.

We consider that the 4th version (Airinei and Homocianu, 2009) was finalized when the prototype was extended offering support for two-way substitutions: both AEXPs and ASCII Compressions (ACOMPs). AEXPs do the same as in the past. ACOMPs do vice versa skipping mostly commas in order to enlarge the output's alphabet and reduce the occupied space. That time we have got encryption errors caused by some untreated ASCII codes in ACOMP generating exceptions - Fig.6.

In its 5th version, the prototype was upgraded to a set of ALGs supporting both a pseudo-random APRK (A Priori Key) generator, and a first generation of APOKs resulting from a single round of statistical corrections just after the final step of the cascade encryption corresponding to the FK's final digit, just before encrypting it by using the Main / Master Key (MK - Fig.7).

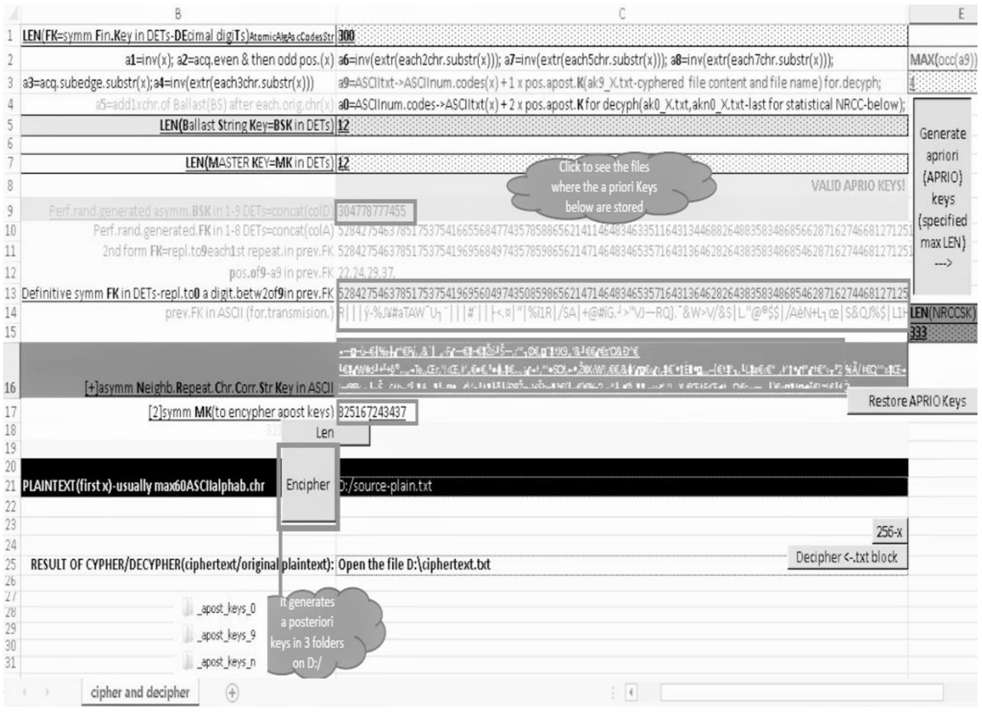
The current version (6th) adds a second system of APOKs generators based on a multi-step simple statistical correction, upgraded ASCII substitution ALGs that support a perfectly random APRK generator and, of course, the 64B3AC acronym. This 6th version of the prototype was tested with cascades up to 1000 (1K) levels. The primary validation of the keys and CT was designed to be done based on their dimension in bytes currently computed together with the number of seconds of processing before finishing the encryption. We intend to add at least simple checks of the digits obtained after converting ASCs of keys and CT to DETs (reduce KB of data to a number) if not hash functions derived from compression ALGs (Alshaikhli and Al Ahmad, 2015). By simply combining permutations on substrings of a prime number length (no APOKs), ballast insertions, AEXPs & ACOMPs as substitutions and corrective ALGs (non-random APOKs) all coded as 0 to 9, we have created the support for a perfectly random APRKs generator used before launching the encryption: FK - to store the cascade of ALGs, MK - to encode the content of files storing APOKs, Neighbor Repetitive Char Correction String Key (NRCCSK) - to do intermediate statistical corrections, Ballast / Padding String / Key (BSK) for random ballast.

The common part of these versions mentioned above is the cascade technique easy to be understood by simply referring to the Triple DES (Stamp, 2006) as evolution from the criticized DES that had limitations imposed by NSA.

The One-Time Pad inspired us to improve the crypto-system by enabling it to ensure the uniqueness of the CT at different runtime moments even for the same input parameters by simply reconsidering just the ALG that generates random chars for corrections and ballast insertions.

3. Some advantages of the development environment

The speed and easiness of programming by using Beginners All-Purpose Symbolic Instruction Code (B.A.S.I.C.) and later Q-Basic, Visual Basic (VB) as part of the Microsoft Visual Studio, Visual Basic for Applications (VBA) and so one is something undeniable since the age of Z80 CPUs (Mann, 1983).



	B	C	E
1	LEN(FK=symm Fin.Key in DETs-DEcimal digiTs)AtomicAlgAscCodesStr	300	
2	a1=inv(x); a2=acq.even & then odd pos.(x)	a6=inv(extr(each2chr.substr(x))); a7=inv(extr(each5chr.substr(x))); a8=inv(extr(each7chr.substr(x)));	MAX(occ(a9))
3	a3=acq.subedge.substr(x);a4=inv(extr(each3chr.substr(x)))	a9=ASCIItxt->ASCIInum.codes(x) + 1 x pos.apost.K(ak9_X.txt-cyphered file content and file name) for.decyph;	4
4	a5=add1xchr.of Ballast(BS) after each.orig.chr(x)	a0=ASCIInum.codes->ASCIItxt(x) + 2 x pos.apost.K for decyph(ak0_X.txt,akn0_X.txt-last for statistical NRCC-below);	
5	LEN(Ballast String Key=BSK in DETs)	12	Generate apriori {APRIO} keys {specified max LEN} --->
6			
7	LEN(MASTER KEY=MK in DETs)	12	
8		VALID APRIO KEYS!	
9	Perf.rand.generated asymm.BSK in 1-9 DETs=concat(colD)	304778777455	
10	Perf.rand.generated.FK in 1-8 DETs=concat(colA)	528427546378517537541665568477435785886562141146483463351164313446882648835834868566287162746681271251	
11	2nd form FK=repl.to9each1st repeat.in prev.FK	528427546378517537541969568497435785986562147146483465357164313646282643835834868546287162744681271251	
12	pos.of9-a9 in prev.FK	22,24,29,37,	
13	Definitive symm FK in DETs-repl.to0 a digit.betw2of9in prev.FK	528427546378517537541969560497435085986562147146483465357164313646282643835834868546287162744681271251	
14	prev.FK in ASCII (for.transmision.)	R|||ý-%J¥#aTAW¯U┐¯|||#`|||├<.¤|"|%i1R|/SA|+@#iG.┘>"VJ—RQ}.¯&W>V/&S|L."@®$$|/AèN+L┐œ|S&QJ%$|L1H	LEN(NRCCSK)
15			333
16	[+]asymm Neighb.Repeat.Chr.Corr.Str Key in ASCII	[illegible]	
17	[2]symm MK(to encypher apost keys)	325167243437	Restore APRIO Keys
18			
19			
20			
21	PLAINTEXT(first x)-usually max60ASCIIalphab.chr	D:/source-plain.txt	
22			
23			256-x
24			
25	RESULT OF CYPHER/DECYPHER(ciphertext/original plaintext):	Open the file D:\ciphertext.txt	Decipher <-.txt block

Len
Encipher
Click to see the files where the a priori Keys below are stored
_apost_keys_0
_apost_keys_9
_apost_keys_n
It generates a posteriori keys in 3 folders on D:/
cipher and decipher

Figure 3. 64B3AC's interface (Excel form: cells, rules, buttons and macros)
Sources: sites.google.com/site/supp4for64B3AC/downloads/64B3AC-interactive-supp-tutorial.pdf and sites.google.com/site/supp4for64b3ac/downloads/fig3.tiff

Our first reasons were related to speed of development and continuity of a work began under VB6.0. But VBA with Microsoft Office Excel (fig.3) was our choice for many other motivations:

- High code compatibility between VBA in different versions of Microsoft Excel (Office 2007, 2010 or 2013) and VB6.0 or .NET;
- Notable speed differences (up to 1:2) when testing this crypto-system on a AMD's x64 based architecture by using Office on 32 and on 64 bits;

- Ease of generating and managing the encipher/decipher parameters and those hundreds random atomic digits as source values for APRKs, APOKs, blocks for ballast insertion and corrections by using cells; the values from those cells are objects of concatenations, exception treatments (validations, formatting rules, IF tests) or dynamic position based searches able to quickly identify custom separated blocks and easily generate derivate cells - e.g. MID(String, SEARCH (subString1, separator1), SEARCH (subString2, separator2));
- Ease of using intuitive cells instead of variables in order to manage the progress of overall processing, the execution time and the output's measurement;
- Ease of automatic recalculation (update) of cells containing randomization functions for each modification of any other cell compared to the complexity of doing that directly from code;
- Additional visual feed-back when debugging a VBA script by using the extended desktop which allows us to continue to see the interface of the application and the exact state of the values in cells, having a considerable impact on decrease of the development effort;
- Theoretical zero exposure of the VBA ALGs configured to work when clicking on the buttons of the custom form (fig.3) defined in Excel if the host sheet is password protected and all possible exceptions are captured. Anyway, the 64B3AC's strength identifies with the power of the system of keys and not with the secrecy of the ALGs. Moreover, for certain exceptions, we can trigger some confusion making ALGs able to proportionally load the CPU at decryption and lead to untraceable nonsenses as apparently meaningful content. Hence the need to measure the output (Fig.5) and the time needed to encrypt and most importantly decrypt with certain parameters and even derive some corresponding estimation functions.

4. Assumptions and additional arguments

Let us assume that, starting from an intercepted cipher text, a common transistor based processor (operations in binary) doing brute force decryption tests (no cryptanalysis) would be able to detect an untreated error/exception or a meaningless result (dead-ends) corresponding to a wrong combination of those at least 10^100 theoretical possibilities in ~ 1/10000 seconds, where 100 is the minimum tested length of 64B3AC's FK. Even so, to test all the combinations of the FK we would need 10^100 / (10000*3600*24*365) $\underset{\sim}{}$3.17*10^88 years (one year usually equals 365*24*3600 seconds).

The blind and time consuming brute force attack scenario above being obviously far from enough on can think of some optimizations. If we imagine the 64B3AC crypto-system simply as a tree with ten branches per level and at least 100 levels (FK's minimum length), the intercepted cipher text acts as a leaf from all those minimum 10^100 possible ones for the same plaintext. Thus an efficient tree search based attack (Giribet, 2007) will most probably try to find out the plaintext (root) by both trying to estimate the number of levels and generating the

reverse route of “sap” thru ramifications from the leaf back to the root by isolating the dead-ends – back-tracking (Stamp and Low 2007).
In order to strengthen the 64B3AC against modern cryptanalytic attacks (Swenson, 2008) we have considered several methods:

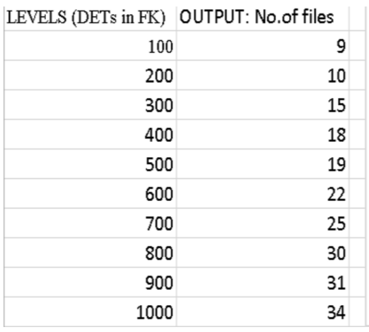
(A) At the end of executing the ACOMP ALG (coded as 0 in FK) we have used a very simple statistical correction of the ITB. This correction was meant to replace the first character in each sequence of two neighbors with the same value by another character from another block randomly generated (once at initialization - currently for APRKs or many times). And there are chances for the random block to absolutely arbitrary provide at a certain point the same character as the one meant to be replaced. The sequence generating this random block can be used any time we want with minimum costs in terms of processing time and not just once, as we did in order to reduce the waiting time and with the compromise of having identical results on different moments for the same input parameters. Thus the neighbor correction becomes a great source of entropy. In addition, it creates an additional level of safety by generating APOKs. Each of these keys indicates a set of characters in ITB that were replaced at a certain point, their positions and has the content encrypted by using the MK. The entire set is generated (after encryption) and must be copied (before decryption) at “D:/_apost_keys_n” (APOK for neighbor correction).

(B) We have also reviewed the first part of the ACOMP ALG and now it operates on any kind of ASCII text and not just on DETs as in previous versions. This creates the perfect support for generating pure random final APRKs including the FK which indicates the sequencing of 0-9 ALGs on each execution level (call) of those up to 1000 tested. In other words, in previous versions we could not have a sequence as 2579430560 because the AEXP ALG (coded as 9 in the FK) tried to transform characters into numerical codes between 0 and 255 (DETs) and later the ALG coded as 0 tried to make ASCII characters from 2/3 digit codes (DETs) reporting failures when encounters an ASCs. And that happened because of those two consecutive occurrences of 0 with no 9 between them. Therefore, in order to avoid impossibilities of transformation and corresponding fatal errors, the first version of ALGs for generating keys were designed consequently with no full random support and unfortunately able to cause sets of weak keys, vulnerabilities and lower entropy. Now things are not the same because the ALG coded as 0 also generates a set of feed-back keys / APOKs before making the statistical correction. These keys (“D:/_apost_keys_0”) have a content encrypted with the MK and store positions (support for decryption) of those non-DETs chars found when trying to compress (ACOMP) and remaining as such.

(C) The AEXP ALG behaves similarly in terms of feed-back. But in the previous versions, the subsequence of positions created by this one was glued to the ITB (Fig.1 - D1), making it harder and harder to further cascade and finally generating a huge ciphertext.txt (tens to hundreds of MB). Now this procedure was dropped and we have decided to do the separate storage of the substring of positions as another set of generated APOKs (_apost_keys_9).

(D) The ALG coded as 5 inserts ballast depending on the BSK. But that can be easily modified the way that for each request for ballast, the ballast will be different. This is also a good source of entropy and ensures that with the same input parameters we will get to different CTs at different moments.

In this 6th version of the 64B3AC crypto-system the ALGs coded as 1, 2, 3, 4, 6, 7 & 8 make transpositions on the ITB. The entire set of ALGs (0-9) is put to work in a symmetrical cascade offering support for perfectly random generators of APRK FK (the cascade itself) and creates several files (APOKs) in addition to the CT.



LEVELS (DETs in FK)	OUTPUT: No.of files
100	9
200	10
300	15
400	18
500	19
600	22
700	25
800	30
900	31
1000	34

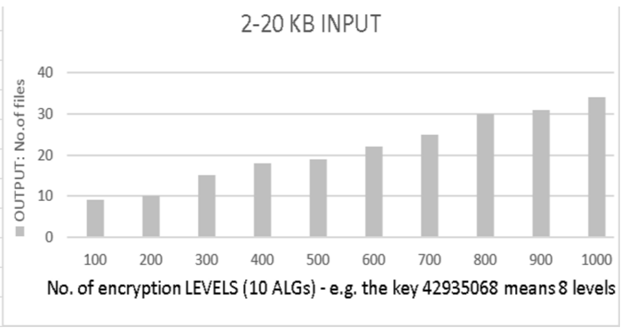


Figure 4. No. of files (output) vs. No. of encryption levels (FK) of 64B3AC
Sources: sites.google.com/site/supp4for64b3ac/downloads/fig4.tiff and the .zip archives (folders, source code, data samples and encryption and decryption results) available at: sites.google.com/site/supp4for64B3AC/downloads

Because the random input blocks (APRKs) for both ballast insertions (ALG code 5) and statistical corrections (end of ALGs code 0 & 9) are easy to refresh/regenerate, they can easily induce a lot of entropy both in CT and APOKs. In addition the last ones can be sent on many channels decreasing the risk of interception. Thus the need to develop and use a protocol able to automatically handle the parallel transmission of CT and APOKs (many files - Fig.4) on different channels might lead to a consistent improvement.

5. Experimental results

In addition to references to code sequences (Barnes, 2010) and screen captures of applications, the 64B3AC crypto-system comes together with an interactive tutorial (Homocianu, 2016) containing both a limited emulation of the functional application and a text-based ad-hoc documentation integrated in an all-in-one portable document format (.pdf) file explaining its entire functionality.

The final tests were made using VBA on Office 2013 64bits and Windows 7 Ultimate 64 bits running on a machine made around the 3rd generation Core i5 3470 CPU (3.2 GHz, x64) and a DDR3 memory module (1.6 GHz, 4096MB).

When running final tests we had no Internet connection, no graphic drivers & no antivirus software installed and no additional processes running.

The preliminary tests were made using various CPUs: AMD Athlon XP 3200+ 2.2GHz single core-1 thread, AMD Athlon 64 X2 4200+ 2.2GHz dual core-2 threads, Intel Pentium E2140 1.6GHz dual core-2 threads, Intel Core2 Duo P8600 2.4 GHz dual core-2 threads, Intel Atom N550 1.5 GHz dual core-4 threads, and the Core i5 quad core-4 threads mentioned above. These tests clearly show that a single core of CPU/single thread was actually used for encryption and decryption using the cascade-based technique chosen for 64B3AC. That conclusion was clear after analyzing the CPU Usage and CPU Usage History values and graphs on the Performance tab in Windows Task Manger: ~100% load for single core and one thread CPU, ~50% load for dual core and two threads CPUs, ~ 25% load for dual/quad core and four threads ones.

As a benchmark tool, 64B3AC shows expected differences between different architectures/generations of CPU units (e.g. Athlon 64 X2-2005 vs. Core i5: 1X-2012 means in terms of speed 1 vs. 2-8X from the simplest to the most complex tests of those 200 made with Core i5).

As a result of the final tests, we have generated three matrices after those 200 tests we have made on the current version of the prototype (100 for encryption, 100 for decryption with the same parameters). The intersection of the axes is meant to represent either the result as processing time (tables 1 and 2) or the Kilobytes (KB - table 3) of output (CT + APRKs + APOKs). On the axes we indicate 10 samples of plaintext with different length of chars (k meaning KB) and other 10 samples (100 -1000 DETs or levels) depending on the length of final semi random FK APRK which describe the order in which the ALGs are processed.

Table 1. Encryption time (sec.) depending on input plaintext (KB) and FK's length

INPUT (KB) Encryption time: LEVELS (DETs in FK)	2	4	6	8	10	12	14	16	18	20
100	2	3	3	4	6	7	11	14	16	21
200	4	7	18	27	43	59	84	136	124	168
300	5	7	9	14	19	30	46	66	63	88
400	5	8	11	18	26	38	50	71	86	125
500	8	18	35	63	111	151	278	261	341	452
600	11	28	61	112	208	293	400	493	601	886
700	13	32	70	122	205	313	460	546	755	1086
800	10	15	85	41	61	95	127	163	268	262
900	18	48	100	182	310	438	817	837	1218	1783
1000	19	48	104	197	308	482	604	805	1131	1638

Source: The .zip archives: sites.google.com/site/supp4for64B3AC/downloads

Table 2. Decryption time (sec.) depending on input plaintext and FK's length

INPUT (KB) Decryption time: LEVELS (DETs in FK)	2	4	6	8	10	12	14	16	18	20
100	2	2	2	3	4	5	7	8	12	14
200	2	6	12	18	28	45	63	97	101	135
300	3	4	7	11	15	22	30	99	49	63
400	3	6	9	14	22	30	40	56	67	83
500	6	15	30	50	87	126	206	217	283	372
600	8	23	48	87	143	225	315	446	489	679
700	10	26	56	101	161	241	365	442	577	892
800	6	11	20	94	49	70	94	127	162	190
900	14	42	85	148	239	359	636	700	959	1434
1000	15	41	84	168	285	361	623	654	916	1372

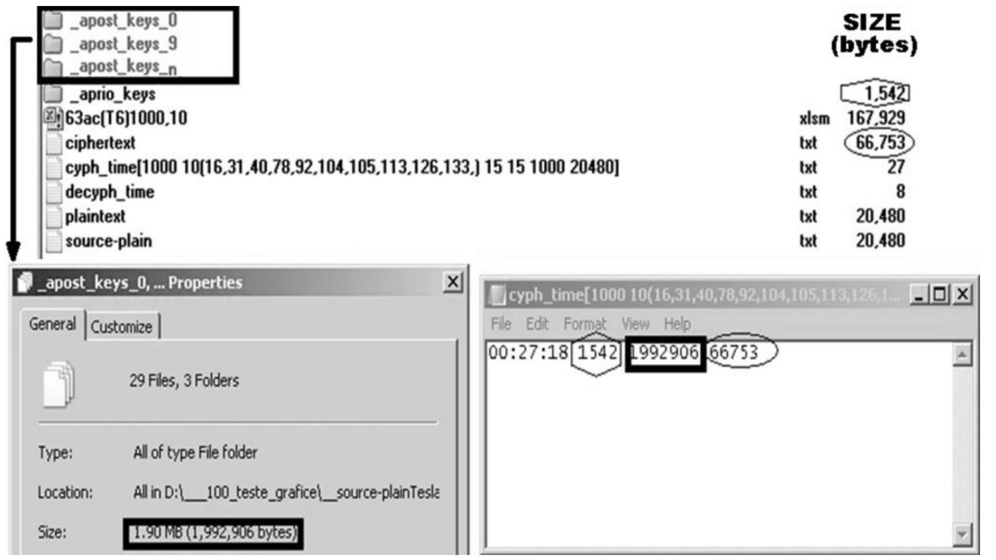
Source: The .zip archives: sites.google.com/site/supp4for64B3AC/downloads

The data in first two tables (1 and 2) suggests most probably an exponential growth of the time we need in order to encrypt/decrypt depending on both the plaintext's size (KB) and FK's length. The data in table 3 indicates a linear relation between the size of input and output and most probably an exponential one between FK's length and the size of output.

Table 3. Size of output (KB) depending on input plaintext and FK's length

INPUT (KB) Size of output: LEVELS (DETs in FK)	2	4	6	8	10	12	14	16	18	20
100	12	23	35	46	58	71	84	96	108	121
200	34	68	103	139	176	216	256	295	336	377
300	30	63	96	128	163	199	236	273	308	346
400	41	85	129	173	220	271	319	368	419	469
500	64	130	200	269	345	422	499	578	656	736
600	105	219	337	456	589	714	845	979	1106	1260
700	142	290	453	609	782	948	1126	1301	1485	1669
800	85	176	268	359	460	565	668	769	872	982
900	189	388	603	816	1045	1270	1505	1741	1987	2232
1000	175	357	546	743	948	1147	1362	1577	1792	2007

Source: The .zip archives: sites.google.com/site/supp4for64B3AC/downloads



```
                                                              SIZE
                                                             (bytes)
_apost_keys_0
_apost_keys_9
_apost_keys_n
_aprio_keys                                                     1,542
63ac(T6)1000,10                                          xlsm 167,929
ciphertext                                               txt   66,753
cyph_time[1000 10(16,31,40,78,92,104,105,113,126,133,) 15 15 1000 20480]   txt   27
decyph_time                                              txt        8
plaintext                                                txt   20,480
source-plain                                             txt   20,480
```

_apost_keys_0, ... Properties

General | Customize

29 Files, 3 Folders

Type: All of type File folder

Location: All in D:\___100_teste_grafice__source-plainTesla

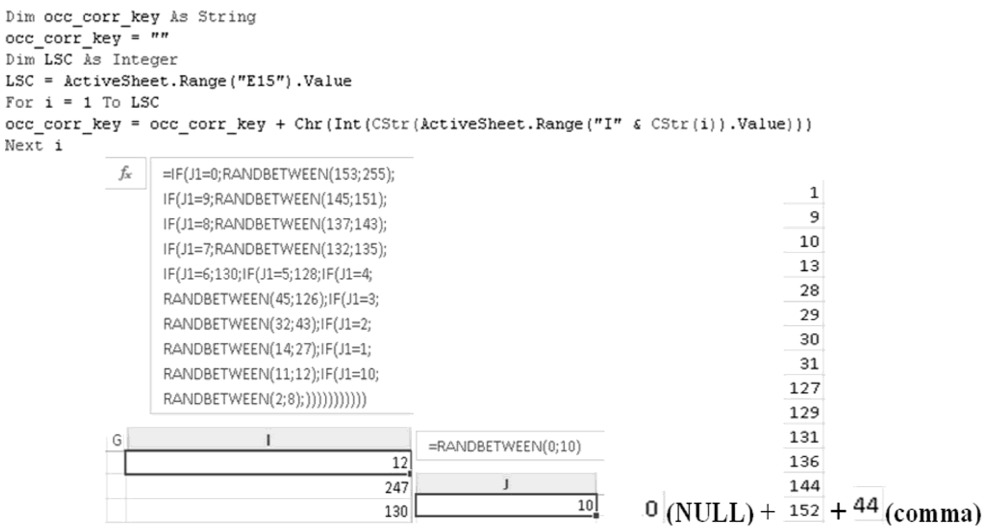
Size: 1.90 MB (1,992,906 bytes)

cyph_time[1000 10(16,31,40,78,92,104,105,113,126,1...

```
00:27:18 1542 1992906 66753
```

Figure 5. The output's size and structure after running a 64B3AC encryption
Source: sites.google.com/site/supp4for64b3ac/downloads/fig5.tiff

The fluctuations noticeable in all the tables above (200L-peak, 800L and1000L-gaps) and happening for all 10 samples of input text (plaintext) are due to some differences concerning the distribution of 9 (cyph_time[..].txt filename - Fig.5) and 0 values (associated to the most time consuming ALGs: AEXP and ACOMP substitutions) in FK.



```
Dim occ_corr_key As String
occ_corr_key = ""
Dim LSC As Integer
LSC = ActiveSheet.Range("E15").Value
For i = 1 To LSC
occ_corr_key = occ_corr_key + Chr(Int(CStr(ActiveSheet.Range("I" & CStr(i)).Value)))
Next i
```

```
=IF(J1=0;RANDBETWEEN(153;255);
IF(J1=9;RANDBETWEEN(145;151);
IF(J1=8;RANDBETWEEN(137;143);
IF(J1=7;RANDBETWEEN(132;135);
IF(J1=6;130;IF(J1=5;128;IF(J1=4;
RANDBETWEEN(45;126);IF(J1=3;
RANDBETWEEN(32;43);IF(J1=2;
RANDBETWEEN(14;27);IF(J1=1;
RANDBETWEEN(11;12);IF(J1=10;
RANDBETWEEN(2;8);)))))))))))
```

G	I
	12
	247
	130

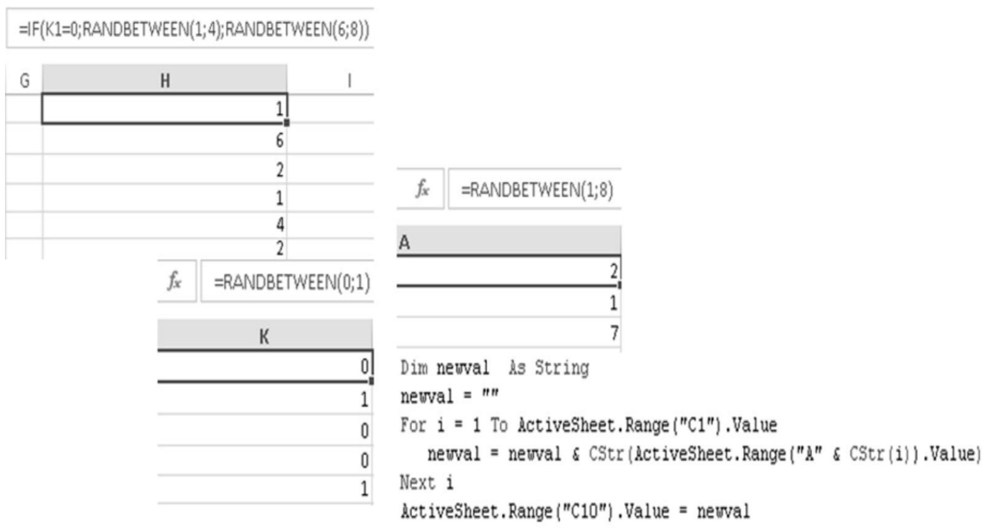
=RANDBETWEEN(0;10)

J
10

1, 9, 10, 13, 28, 29, 30, 31, 127, 129, 131, 136, 144, 152 + 44 (comma)

0 (NULL) +

Figure 6. ASCII codes of characters to avoid in 64B3AC's substitutions (VBA exceptions / errors)
Source: sites.google.com/site/supp4for64b3ac/downloads/fig6.tiff



```
=IF(K1=0;RANDBETWEEN(1;4);RANDBETWEEN(6;8))
```

G	H	I
	1	
	6	
	2	
	1	
	4	
	2	

```
=RANDBETWEEN(0;1)
```

K
0
1
0
0
1

```
=RANDBETWEEN(1;8)
```

A
2
1
7

```
Dim newval  As String
newval = ""
For i = 1 To ActiveSheet.Range("C1").Value
   newval = newval & CStr(ActiveSheet.Range("A" & CStr(i)).Value)
Next i
ActiveSheet.Range("C10").Value = newval
```

Figure 7. Spreadsheet functions returning random digits used by the 64B3AC's MK (left) and FK (right)
Source: sites.google.com/site/supp4for64b3ac/downloads/fig7.tiff

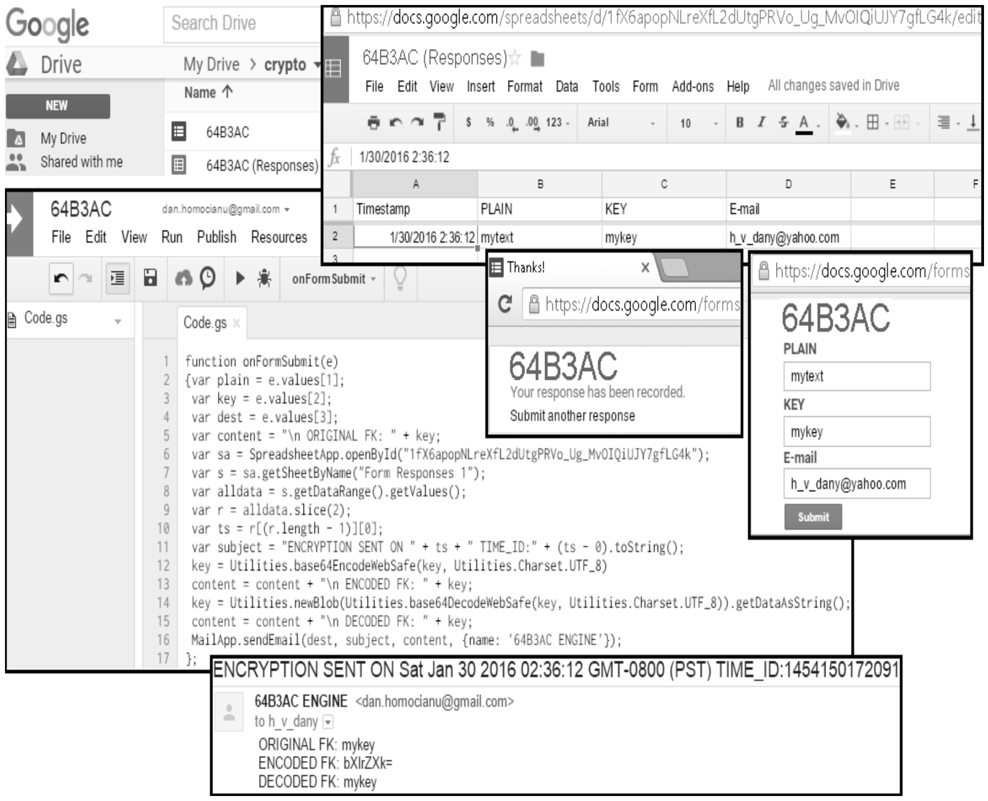
The initial exceptions (16 - Fig.6) making us to reduce the actual output alphabet from 256 to 240 characters in the 6th (current) version of the crypto-system are caused by the character with code 0 (NULL), by some other 14 characters acting as NULL in VBA (right of Fig.6) and also by the character with code 44 (comma - still reserved for AEXPs).

6. Further approaches

In order to improve the current solution, first of all we intend to define and implement a perfectly random key generator exploiting the already existent support for it in the architecture of all ALGs.

Next we have plans to offer support for a runtime dynamic definition of a greater number of faster ALGs and corresponding parameters randomly associated to characters of a larger alphabet (e.g. ASCII instead of DETs). Thus we will deal with native ASCII FKs and not DECIMAL compressed to ASCII ones. This support can be ensured by using various substitution matrices, corresponding checksums also serving as positional markers for random ballast insertions, and the infinite collection of prime numbers useful for indicating the length of substrings as objects of permutations. We also intend to define an extended domain for the CTs (UNICODE - fig.8).

In addition to those above we will use two main approaches of exploiting the physical processing resources.


Google
Search Drive
Drive
My Drive > crypto
Name
NEW
My Drive
Shared with me
64B3AC
64B3AC (Responses)
https://docs.google.com/spreadsheets/d/1fX6apopNLreXfL2dUtgPRVo_Ug_MvOIQiUJY7gfLG4k/edit
64B3AC (Responses)
File Edit View Insert Format Data Tools Form Add-ons Help All changes saved in Drive
64B3AC
dan.homocianu@gmail.com
File Edit View Run Publish Resources
onFormSubmit
Code.gs
Thanks!
https://docs.google.com/forms
Submit

	A	B	C	D
1	Timestamp	PLAIN	KEY	E-mail
2	1/30/2016 2:36:12	mytext	mykey	h_v_dany@yahoo.com

```
function onFormSubmit(e)
{var plain = e.values[1];
 var key = e.values[2];
 var dest = e.values[3];
 var content = "\n ORIGINAL FK: " + key;
 var sa = SpreadsheetApp.openById("1fX6apopNLreXfL2dUtgPRVo_Ug_MvOIQiUJY7gfLG4k");
 var s = sa.getSheetByName("Form Responses 1");
 var alldata = s.getDataRange().getValues();
 var r = alldata.slice(2);
 var ts = r[(r.length - 1)][0];
 var subject = "ENCRYPTION SENT ON " + ts + " TIME_ID:" + (ts - 0).toString();
 key = Utilities.base64EncodeWebSafe(key, Utilities.Charset.UTF_8)
 content = content + "\n ENCODED FK: " + key;
 key = Utilities.newBlob(Utilities.base64DecodeWebSafe(key, Utilities.Charset.UTF_8)).getDataAsString();
 content = content + "\n DECODED FK: " + key;
 MailApp.sendEmail(dest, subject, content, {name: '64B3AC ENGINE'});
};
```

64B3AC
Your response has been recorded.
Submit another response

64B3AC
PLAIN: mytext
KEY: mykey
E-mail: h_v_dany@yahoo.com

ENCRYPTION SENT ON Sat Jan 30 2016 02:36:12 GMT-0800 (PST) TIME_ID:1454150172091

64B3AC ENGINE <dan.homocianu@gmail.com>
to h_v_dany

ORIGINAL FK: mykey
ENCODED FK: bXlrZXk=
DECODED FK: mykey

Figure 8. Example of using Google Script - *Code.gs*, Google Forms - *64B3AC* & Google Sheets - *64B3AC (Responses)* to automatically send UNICODE encryption parameters via e-mail
Source: sites.google.com/site/supp4for64b3ac/downloads/fig8.tiff

The first one is based on working with a cloud scripting language (e.g. Google Apps Script / Google Script derived from Java Script) able to shorten the development (on-line prototype). Such a prototype should be able to automatically send both an e-mail message (1st channel - fig.8) to the receiver having the Time Id as subject and the CT (content or link to it) and short messages (SMSs) to the receiver's phones / devices (another channels) having the same Time Id and parts of the set of decryption keys (content or link).

The second approach will benefit from working with a large number of substitution matrices simultaneously defined at runtime by using parallel algorithm specifications (Hummel, 1994) and parallel computing platforms - e.g. Nvidia CUDA C/C++.

7. Conclusions

The general conclusion of the paper underlines the importance of the security supported by methods, techniques and tools able to provide an increased level of complexity and accompanied by instruments able to explain the application's logic together with the corresponding requirements.

The chosen acronym is able to synthesize the logic of the whole approach and also refers to a hexadecimal text (Color-Hex, 2016) which encodes (in many color spaces) a certain color related to that present in the symbols of the chosen development environments.

The examples describe the components of a functional model primarily based on substitutions, permutations, ballast insertions and statistical corrections. These are used one after another in an arbitrary cascade type order on ASCII strings. Their implementation involves a user friendly interface and a productivity oriented software development application.

The tables with test data made on the final version serve as a preliminary support for processing time and output size estimations very useful when developing further improvements of the crypto-system.

The paper does not claim completeness although the approaches were defined after many increments on the initial ideas starting from storage needs up to message sending requirements and promises a lot in terms of speed of processing although the cascade itself (Lin at al., 2010) usually won't benefit from the advantages of parallel computation.

ACKNOWLEDGMENT:

The paper was funded by the Department of Research, Faculty of Economics and Business Administration, "Alexandru Ioan Cuza" University of Iasi.

REFERENCES

[1] **Airinei, D., Homocianu, D. (2009), *An Optimized Cryptographic Way to Secure DSS Spreadsheet Reports.*** *Proceedings of the Ninth International Conference on Informatics in Economy - Education, Research & Business Technologies*, 903-908, ASE Publishing House, Bucharest;

[2] **Alshaikhli, I. F., AlAhmad, M. A. (2015), *Cryptographic Hash Function: A High Level View.*** In: *Al-Hamami, A. H., Al-Saadoon, G. M. W.* (eds.): *Handbook of Research on Threat Detection and Countermeasures in Network Security*, 80-94, IGI Global, Hershey;

[3] **Barnes, N. (2010)*, Publish your Computer Code: It Is Good Enough*.** *Nature*, 467, 753-757;

[4] **Color-Hex (2016). *#64b3ac Color Hex*.** [Online]. Available at: <color-hex.com/color/64b3ac>;

[5] **Giribet, G. (2007)*, Efficient Tree Searches with Available Algorithms*.** *Evolutionary Bioinformatics*, 3, 341-356;

[6] **Homocianu, D. (2016)*, Interactive Support Tutorial for the 64B3AC Crypto-system*.** [Online]. Available at: <sites.google.com/site/supp4for64B3AC /downloads/64B3AC-interactive-supp-tutorial.pdf>;

[7] **Hummel, S. F. (1994)*, On the Implementation of Set-based Parallel Algorithms*.** In: *Blelloch, G. E., Chandy, K. M., Jagannathan, S.* (eds.): *DIMACS series in discrete mathematics and theoretical computer science*, Vol. 8., 101-114, The American Mathematical Society, New Jersey;

[8] **Lin, M., Cheng, S., Wawrzynek, J. (2010)*, Cascading Deep Pipelines to Achieve High Throughput in Numerical Reduction Operations*.** *Proceedings of the International Conference on Reconfigurable Computing*, IEEE Publishing, Cancun;

[9] **Mann, S. (1983)*, Review: Kaypro 10*.** *InfoWorld*, 44, 109-110;

[10] **Stamp, M. (2006)*, Information Security. Principles and Practice***, *Wiley*, New Jersey;

[11] **Stamp, M., Low, R. M. (2007)*, Applied Cryptanalysis: Breaking Ciphers in the Real World***, 152-158, *John Wiley & Sons*, New Jersey;

[12] **Swenson, C. (2008)*, Modern Cryptanalysis: Techniques for Advanced Code Breaking***, Wiley, Indianapolis.