# The Frogbit cipher, a data integrity algorithm

Author: Thierry Moreau

January 1997
Updated, i.e. shortened, April 2005



Author's address: CONNOTECH Experts-conseils Inc.
9130 Place de Montgolfier
Montréal, Qc
Canada H2M 2A1

Tel.: +1-514-385-5691
Fax: +1-514-385-5900
e-mail thierry.moreau@connotech.com

## Abstract

We present a unique approach to data integrity algorithms. Our definition of data integrity protection is inspired from, and suitable for, a stream cipher arrangement. Short of a practical perfect algorithm according to the data integrity definition, we propose a novel heuristic, the Frogbit data integrity algorithm.

Table of contents

## 1. Introduction and prior work

The prior work on MAC and secure hash algorithms use definitions of data integrity based on the fixed-length result of a function (MAC or secure hash) applied to a variable-length message (e.g. [1] and [4], respectively for MAC and secure hash). Contrary to our proposal, almost all data integrity schemes are block algorithms (reference [3] is an exception).

In this paper, we introduce a unique definition of data integrity protection. We propose an arrangement using double "exclusive or" (XOR) encryption and providing this new concept of data integrity protection. Two applications of our arrangement are shown, an impractical one conforming to our data integrity definition, and the Frogbit heuristic that we propose as our main contribution.

## 2. The new arrangement for cryptographic data integrity

A stream cipher operates on cleartext messages serially, applying XOR encryption to each message bit. It uses a pseudo-random bit generator (PR generator) for which the seed is derived from the cipher secret key. Our definition of data integrity protection uses the bit-serial characteristic of a stream cipher. Let us assume that a bit-serial cleartext message is encrypted into ciphertext by a sender, and that a session key is established for this purpose. The *legitimate* receiver decrypts the ciphertext and gets the *recovered cleartext*. An adversary is given the opportunity to modify the ciphertext bits in transit from the sender to the receiver.

**Definition.** We define a cipher that offers data integrity protection as one where the modification of a single bit in the ciphertext by the adversary forces the randomization of the remaining portion of the recovered cleartext.

It is easy to turn a cryptographic scheme strictly conforming to this definition into a scheme where any modification of the ciphertext has at most a $2^{-L}$ probability of remaining undetected by the legitimate receiver: simply append $L$ bits of redundancy to the cleartext message (e.g. $L$ bits set to zero), and have the legitimate receiver check their presence before accepting the recovered cleartext as genuine. We assume that the adversary neither cuts nor extends the ciphertext bit string, but such a concern may be adressed with other forms of simple message redundancy (e.g. a cyclic redundancy check).

We are now ready to describe the arrangement intended to provide the data integrity protection as defined. This is illustrated in figure 1. This figure is concrete with respect to the double XOR encryption applied serially to each cleartext message bit. This double encryption creates the *inner sequence* (the intermediate result in the double XOR encryption), with the useful property that the adversary can not alter the cleartext bit without altering the inner sequence bit.

Using this arrangement, we are now ready to present two designs that provide the data integrity protection, respectively unconditionnally and heuristically.

## 3. Two designs based on the data integrity arrangement

The notation we use for double XOR encryption assigns $m_i$ to the $(i+1)$'th cleartext message bit, $e_i$ to the $(i+1)$'th ciphertext bit, and $s_i$ to the $(i+1)$'th inner sequence bit. Accordingly,

$$e_i = k1_i \oplus m_i \oplus k2_i, \tag{1}$$

$$m_i = k2_i \oplus e_i \oplus k1_i, \tag{2}$$

where $k1_i$ and $k2_i$ are the key bits used respectively for the first and second stage of the double XOR encryption, and $\oplus$ is the "exclusive or" operator. Equations (1) and (2) show, respectively, the double XOR encryption and double XOR decryption procedures. The equation (2) is a simplified notation because the legitimate receiver ignores the actual $e_i$ and has no choice but to use the *received* version of it, which may have been modified by the adversary. This small abuse of notation is applied throughout this paper, including the following equation (3) that shows how the inner sequence is shared between the originator and the legitimate receiver (tacitly in the absence of alterations by the adversary):

$$s_i = k1_i \oplus m_i = k2_i \oplus e_i. \tag{3}$$

### 3.1 The perfect data integrity cipher

The first version of our data integrity arrangement uses an arbitrarily long random bit string as a key store, with an efficient direct access capability, using the notation $R[j]$ for the $j$'th bit pair of this string. After processing a number $i$ of cleartext message bits, the decision machine

state includes the value of $i$ and the inner sequence bits processed so far, that is $s_0, s_1, \ldots s_{i-1}$. The decision machine simply sets the pointer $j$ to an integer whose binary representation is

$$j=1s_0s_1\ldots s_{i-2}s_{i-1}. \quad (4)$$

Then, the processing of $m_i$ uses the bit pair

$$<k1_i,k2_i>=R[j]. \quad (5)$$

This "perfect data integrity cipher" is conformant to our definition of data integrity protection. Indeed, the bit pair $<k1_i,k2_i>$ is dependent on every single bit of the prior inner sequence, and two distinct prior inner sequences can not produce the same pointer $j$ in equations (4) and (5). It should be noted that the decision machine does not use the input $k2_i$, which is available without any computational overhead. In the heuristic described below, the input $k2_i$ is used as a leverage to give more "inertia" to a modification of $s_{i-1}$.

### 3.2 Introducing the Frogbit

The *Frogbit* algorithm is this other and more practical version of our data integrity arrangement. We developed it with by a trial and error process where adversary attacks were simulated under the most defavourable conditions. The name Frogbit comes from a drifting aquatic plant, *Hydrocharis morsus-ranae, Linnaeus* (this specie was inadvertently released in the wilderness of North America by an agency of the Canadian government). Putting aside any allusion to "meandering" crypto policy, the "drifting" characteristic of the Frogbit is found in the key store specification. The Frogbit algorithm can be viewed as a "stream" cipher variant.

As a key store, we use ten independent sequential PR bit generators. Here, “independent” means statistical independence, and additionally that each generator is used at its own stochastic pace. Thus, the pointer produced by the decision machine has two components: 1) a concrete selector $d'(i)$ for one generator out of the group of ten, and 2) an implicit position indicator in the output sequence of the selected generator. The position indicator of any generator advances by two bit locations each time this generator is selected. The number ten is a parameter choice that came naturally during the development process.

The decision machine has two parts. One is an elaborate bit manipulation specification, the *run length process*. This part translates the decision machine inputs
$\ldots, <k2_{i-1},s_{i-1}>, <k2_i,s_i>, <k2_{i+1},s_{i+1}>, \ldots$ into a sequence of function values
$\ldots, d(i-1), d(i), d(i+1), \ldots$ each of which being either “undefined”, or a non-negative integer less than ten. If bit $s_i$ is changed, the minimal consequence is that two defined values of the function $d()$ are either changed, or replaced by a single one different from the original two.

The other part of the decision machine translates $d(i)$ into the selector $d'(i)$. It also applies a permutation to the very index table that translates $d(i)$ into the selector $d'(i)$. Accordingly, we call this part the *index permutation process*.

## 4. The core Frogbit algorithm

The core Frogbit algorithm is defined as iterative functions of the sequence
$<k2_0,s_0>, <k2_1,s_1>, \ldots <k2_{i-1},s_{i-1}>, <k2_i,s_i>$. The following mathematical exposition

may be followed using the numerical example included in table 1. The mathematical formulation bears little relation to the design rationales of the algorithm; it mainly serves as the witness of the algorithm.

To encrypt and to decrypt a message, equations (1) and (2) are applied, respectively. The specification for the bit pair $<K1_i,k2_i>$ is given below. In this process of double XOR encryption or decryption, the bit $s_i$ is defined by equation (3).

The run length process starts with a function $r(i)$ of a portion of the inner sequence, from $s_0$ to $s_i$:

$$r(i)=0 \text{ if } s_i \neq s_{i-1} \tag{6}$$

$$r(i)=r(i-1)+1 \text{ if } s_i=s_{i-1}, \tag{7}$$

where, by convention, $s_{-1}=0$ and $r(-1)=0$. In practice, the run length must be bounded by a finite upper limit, $n$. So, a bounded run length function $r'(i)$ is defined as

$$r'(i)=r(i) \bmod n. \tag{8}$$

When an $r'(i)=0$ is encountered, $r'(i-1)+1$ gives the *bounded run* length. Later, it will be necessary to distinguish between $r'(i)=0$ forced by the $\bmod n$ operation (where $r'(i)=0 \neq r(i)$) from $r'(i)=0$ occurring naturally (where $r'(i)=0=r(i)$).

In the Frogbit cipher, each bounded run is marked with $r'(i)=0$, and is the occasion for to draw a number $d(i)$:

$$d(i) \text{ is undefined for } r'(i) \neq 0, \tag{9}$$

$$d(i) = 2^{r'(i-1)+1} - 2 + \sum_{j=0}^{r'(j-1)} 2^j \times k2_{i-j}, \text{ for } r'(i)=r(i)=0, \text{ and} \quad (10)$$

$$d(i) = 2^{n+1} - 2 + \sum_{j=0}^{n-1} 2^j \times k2_{i-j}, \text{ for } r'(i)=0 \neq r(i). \quad (11)$$

The value $d(i)+2$ is a number whose binary representation is

$$d(i)+2=1k2_{i-r'(i-1)}k2_{i-r'(i-1)+1}...k2_i, \text{ for } r'(i)=r(i)=0, \text{ and} \quad (12)$$

$$d(i)+2=10k2_{i-r'(i-1)}k2_{i-r'(i-1)+1}...k2_i, \text{ for } r'(i)=0 \neq r(i). \quad (13)$$

The run length process is an encoding of the sequence of $<k2_i,s_i>$ into a sequence of function values $d(i)$. If one ignores the undefined values for $d(i)$, then calculates the probability distribution for the remaining values, and feeds them to the Hufman coding algorithm, the result is an encoding scheme which preserves exactly the message length (e.g. for $n=2$, the probability distribution for $d(i)$ values is $1/4$ for $d(i)=0$ or $1$, and $1/16$ for $d(i) \geq 2$).

With the recommended parameter $n=2$, the Frogbit specification so far can be implemented as a table lookup operation, as shown in table 2.

The index permutation process maintains a table of permuted indexes, of a dimension $N=3\times 2^n-2$ (hence the number ten when $n=2$), using the following mathematical notation:

- the function $T(i)$ represents the table at the end of processing for bit $i$,
- the table contents is represented by a curly bracketed list of numbers, for instance $T(-1)=\{0,1,2, \ldots N-1\}$, and
- the notation $T(i)[j]$ represents a query of the $(j+1)$'th entry of the table $T(i)$.

Formally, let $i'(i)$ be a function that "points to" the beginning of a bounded run that finishes at bit position $i$:

$$i'(i) \text{ is undefined if } r'(i) \neq 0, \tag{14}$$

$$i'(i) = i - 1 - r'(i-1) \text{ if } r'(i) = 0. \tag{15}$$

When converted to a programmatic specification, the function $i'(i)$ becomes the mere designation for the previous value of the permuted index table, which is updated only when $r'(i)=0$.

The generator index function, $d'(i)$ is defined as

$$d'(-1) = 0, \tag{16}$$

$$d'(i) = d'(i-1) \text{ if } r'(i) \neq 0, \tag{17}$$

$$d'(i) = T(i'(i))[d(i)] \text{ if } r'(i) = 0. \tag{18}$$

Then, for the bit $i+1$ to be processed by the Frogbit cipher (that is, $m_{i+1}$ for encryption, or $e_{i+1}$ for decryption), the selector function $d'(i)$ (where $0 \leq d'(i) < N$) indicates the PR sequence to utilize for the pair of bits $<k1_{i+1}, k2_{i+1}>$.

Finally, the index table function is defined as

$$T(-1) = \{0, 1, 2, \ldots N-1\}, \tag{19}$$

$$T(i) \text{ is undefined if } r'(i) \neq 0, \tag{20}$$

$$T(i) = \{T(i'(i))[P[j,0]], T(i'(i))[P[j,1]], \ldots, T(i'(i))[P[j,N-1]]\},$$

$$\text{where } j = 2 \times d'(i-1) + s_{i-1}, \text{ if } r'(i) = 0. \tag{21}$$

The notation $P[\alpha,\beta]$ represents a fixed two-dimensional permutation table, with a first index in the range $0 \le \alpha < 2N$, a second index in the range $0 \le \beta < N$, and each entry in the table in the range $0 \le P[\alpha,\beta] < N$.

For instance, with $n=2$, the table $P[\alpha,\beta]$ has 20 rows and 10 columns. A fixed permutation table is suggested, see table 3. It is necessary for each row in the table to be a permutation. A nice property for a permutation table is to allow the function $T(i)$ to take 10 factorial ($10!=3,628,800$) different values, which is the mathematical maximum. A large number of permutation tables satisfy this condition.

The suggested permutation table, table 1.1, was selected by a “fair” PR program that enforced a number of criteria. It limited the frequency of a given number in any column of the table (a limit of three occurrences per column). It forced the permutation of each row to contain exactly two cycles. It didn't allow any cycle of length one. Additional care was taken to avoid cycles made of the same positions in two different rows.

With regard to the key store organization, the generic requirements for a Frogbit cipher implementation is to have $N$ independent PR sequences securely derived from a secret key. The PR sequences are independently utilized by the Frogbit cipher. The Frogbit cipher accomodates any PR generator that is acceptable as a stream cipher. Moreover, it might be secure even with cryptographically weak generators.

The following notation will be useful to discuss the various application scenarios for the core Frogbit algorithm: let

$$<e_i,G_{i+1}>=F(<m_i,G_i>,K), \text{ for } 0 \le i<n, \tag{22}$$

where $n$ is the message length and $K$ is the contents of the secret key store. Equation (22) represents the simultaneous 1) encryption of cleartext message bit $m_i$, into the ciphertext bit $e_i$ using the Frogbit state $G_i$, and 2) internal Frogbit state transformation from $G_i$ to $G_{i+1}$. Equation (22) reconciles the mathematical specification of the core Frogbit algorithm with figure 1, where the state information is explicitly represented.

This state information $G_i$ comprises the following elements, listed below as they appear just before the computation of function $r(i)$:

for the run length processing:

1) the previous bit for the run length processing, that is $s_{i-1}$,

2) the current bounded run count, that is $r'(i-1)$,

3) a partial sum, that is the accumulated binary representation $k2_{i-r'(i-1)}$, $k2_{i-r'(i-1)+1}$, ... $k2_{i-1}$ (actually, when $n=2$, this is at most a single bit of information);

for the index premutation process:

4) the current table of permuted indexes, that is $T(i'(i))$;

for the key source selection:

5) the current key stream number, that is $d'(i-1)$,

6) the $N$ distinct key stream positions.

Up to now, the secret key was assigned to the contents of the key store. The initial state

$G_0$ was explicitly defined as $s_{-1}=0$, $r(-1)=0$, an empty partial sum, $T(-1)=\{0,1,2, \ldots 9\}$, $d'(i-1)=0$, and every key stream positions at the starting position. As an alternate implementation strategy, the components 1) to 5) of initial state $G_0$ could be derived from the secret key as well. Also, an implementation may publicly define the key store contents and assign the secret key to the key stream positions.

With the proposed data integrity arrangement, we suggested that redundancy inserted into the cleartext by the sender and checked by the legitimate receiver would provide an adequate mechanism for data integrity protection.

## 5. Design rationales

The design rationales for the core Frogbit algorithm are better described as an ad-hoc heuristic development process than as the refinement of prior research in the field of cryptographic data integrity. Yet, the original incentive for our work was the sparing published work on practical data integrity algorithms in the context of a stream cipher.

The actual heuristic development process started with the vague idea that the run length encoding could provide the projection of a single bit modification to adjacent bits in a stream cipher. Every other feature or component was added by a sequential trial and error process, with some empirical measure of progress towards the Frogbit primary objective. In this random search, we did not follow routes with very bad efficiency characteristics. In addition, we stopped when the algorithm on the drawing board appeared to fulfill the requirements.

For the record, the important aspects of the Frogbit algorithm were put into place in the following sequence:

1) the inner sequence, and the utilization of independent and secret bits $s_i$ and $k2_i$,
2) the key store organization where no PR bit is ever discarded (we considered schemes similar to [2]), and finally
3) the full permutation of the index table.

As we headed towards a better looking algorithm, we had to refine our security measurement approach. The obvious weaknesses of early attempts were easily found with manually entered test data. Then, exhaustive search for collisions was used, and we obviously couldn't avoid making use of the birthday paradox of data integrity algorithms (although we were looking for data integrity of encrypted messages, we attempted to observe collisions in the clear). When the residual collisions of our algorithm were not visible with the "resolution" of direct birthday attacks, we reverted to *component collision* search. We will discuss the concept of component collision in section 6.

There are a couple of ad-hoc verifications which may be applied to the final Frogbit algorithm, but which can only give negative indications about its validity. The Frogbit state is a structured piece of information, something radically different from the familiar set $\{0,1\}^L$ (binary strings of length $L$, to which current data integrity algorithms maps the message space, e.g. $L=160$ for the SHA, [5], [6]). Whatever the set of possible Frogbit state is, its probability distribution is definitely not uniform. We refer to the probability distribution induced by uniformly sampling messages from the message space and taking the corresponding final Frogbit

state (assuming a fixed secret key). So, we felt it was necessary to assess the *Frogbit state entropy* associated with the final Frogbit state probability distribution. Short of a mathematical analysis of our algorithm, we created an ad-hoc data compression program to assess the typical bit length of *compressed* Frogbit final state. At best, this gives an upper limit on the actual Frogbit state entropy. These experimental results showed bit counts usually between 108 and 160.

At the end of the journey, the heuristic development process gave the core Frogbit algorithm, which fits into the data integrity arrangement shown in figure 1. The algorithm and the arrangement are radically different from prior design principles for cryptographic data integrity solutions. For this reason, the notion of modes of operation has to be revisited in the context of the Frogbit algorithm.

## 6. Component collision analysis

Our strongest validation of the Frogbit algorithm security is component collision search. Let us use the notation $G_i$ for the Frogbit state just before the processing of bit $i$. A full Frogbit collision is two inner sequence substrings $s_c$, $s_{c+1}$, …, $s_{c+k-1}$ and $s'_c$, $s'_{c+1}$, …, $s'_{c+k-1}$ that meets two requirements:

1) the two inner sequences start and end with the same pair of states $G_c$ and $G_{c+k}$, and

2) they use identical sequences from the key store for its $k2_i$ components (the $k1_i$ components are embedded in the inner sequence bits, the latter being alterable externally through the ciphertext).

The Frogbit state contains structured data. It is thus possible to isolate one or more component of the Frogbit state and search collisions for this partial state information.

For a collision search, we used the table of permuted indexes as the core component of the Frogbit state. Indeed, since this table has only $10! \approx 20^5$ states and 20 different transitions at each update operation, so collisions are deemed to occur with small number of transitions. For instance, the permutation rows 14, 9, and 7 lead to the same state of the index table as the permutation rows 3, 17, and 0. Sample research work that could relate to the Frogbit algorithm includes [7].

To find collisions when more components of the Frogbit state are taken into account, we programmed a simulation with exhaustive search using a birthday attack strategy. It took about 12 hours of computing time on a Pentium 75MHz processor to find a collision involving all of the Frogbit state, except for the sequences of $K2_i$. In the theoretical perspective where the Frogbit state implicitly defines the sequences of $K2_i$, the collision found has identical *relative* keystream positions, in the final state $G_{c+k}$, but the *absolute* keystream positions in the initial state $G_c$ do not agree. The specific collision occurs from the (randomly chosen) initial state $G_c$ where $s_{c-1}=1$, $r(c-1)=0$, $T(i'(c))=\{4,1,5,2,0,9,7,8,3,6\}$, and $d'(c-1)=7$; the two alternative sequences of $d(c+i)$'s leading to the same final state are ƀ, 9, ƀ, 4, ƀ, 6, ƀ, 9, ƀ, 6 and ƀ, 5, ƀ, 8, ƀ, 6, ƀ, 3, ƀ, 2 (undefined values for $d(c+i)$ are indicated by ƀ). Exactly two bit pairs are extracted from the PR generators 5 to 9, and none from generators 0 to 4.

Coming back to our initial definition of data integrity, if we were able to find a complete

collision of the core Frogbit algorithm, we could hope to estimate the very small probability that a random alteration of the ciphertext has a limited impact on the recovered plaintext. Our inalility to find such collisions is the strongest validation of the Frogbit algorithm so far. It is as if the PR generators embedded in the Frogbit algorithm were very important for its validity.

If the core Frogbit algorithm had to be strengthened, there are at least three avenues.

1) Increasing the base two from sequences (12), and (13). This requires taking more than two bits from the key streams, e.g. a triple $<k1_i,k2_i,k3_i>$ for base $B=4$. Using the additional bit(s), the following digits are formed

$$d_i=k2_i\times 2+k3_i,$$

and are inserted in sequences (12), and (13) which now would read "The value $d(i)+B$ is a number whose base $B=4$ representation is

$$d(i)+B=1d_{i-r'(i-1)}d_{i-r'(i-1)+1}\ldots d_i, \text{ for } r'(i)=r(i)=0, \text{ and} \quad (23)$$

$$d(i)+B=2d_{i-r'(i-1)}d_{i-r'(i-1)+1}\ldots d_i, \text{ for } r'(i)=0\neq r(i)\text{"}. \quad (24)$$

The extra bit $k3_i$ does not participate in the double XOR encryption.

2) Increasing the vertical dimension of the permutation table. With the current Frogbit definition, a single bit, $s_i$, "joins" $d(i)$ in equation (21) to select a permutation row. The least significant bit of the indice $i$, or even an extra key stream bit $k4_i$ could be considered in equation (21).

3) Increasing the parameter $n=2$ to $n=3$.

## 7. Practical proposals

In order to encourage cryptanalysis attempts at the core Frogbit algorithm, we publish [[electronically]] a challenge program written in the C programming language. This program encrypts and decrypts ASCII or binary files, but it is meant to offer a base for cryptanalysts who would attempt to find collisions in the core data integrity mechanism. This source code includes the compression function that we used to assess the Frogbit state entropy. The key store specification uses very simple PR generators and allows their parameterization to a great extent. The complete Frogbit initial state, including the generators' seed (and their parameters if desired) is provided to the challenge program in readable ASCII format. The Frogbit final state is output by the program in this same format. The challenge is to find two different files starting with the same the initial state and producing the same final state.

## 8. Conclusion

In this paper, a comprehensive cryptographic algorithm proposal was described. We believe the main novel aspects of our work to be:

1) our definition of data integrity (section 2),
2) the abstract data integrity arrangement (figure 1),
3) the core Frogbit algorithm that we presented as a data integrity scheme based on pseudo-random number generators,
4) the use of cleartext message salting to overcome the usual restriction on secret key re-use in the case of the stream cipher (section 6.1), and
5) a practical scheme for a “semi-proprietary” algorithms, that is a published cipher

*structure* with a significant portion of the *implementation details* remaining secret. We used an heurisitc development approach to venture into an unprecedented approach to cryptographic data integrity. The verifications we applied to the Frogbit proposal support its validity as a secret key data integrity mechanism. To reconcile our new approcah with the current practices in the field of cryptography, we had to review the concept of a mode of operation used with a block algorithm. The theoretical support for the suggested modes of operation is lacking bcause previous work in this area focused on block algorithms. This may trigger concerns from an application perspective.

For a working implementation of our algorithms, the specifications given in this paper are incomplete. Two major areas are the design of ten pseudo-random number generators and the derivation of the complete internal state of the algorithm from a fixed length secret key. We presented a hash function specification which is admittedly beyound our stated security target (that is, data integrity protection based on a secret key). Consequently, even if we don't know it, cryptographers are advised to suspect a flaw in our hash function proposal. The hash function specification is thus given only as a challenge (finding a flaw could lead to a design improvement proposal), or as a blueprint for a MAC function design.

Finally, we suggested directions for the strengthening of our algorithm, but we couldn't find specific reasons to pursue them. We suspect our research and the field of graph theory to be related.

## References

[1] Bakhtiari, S., Safari-Naini, R.S., Pieprzyk, J., ***Keyed Hash Functions***, in ***Cryptography - Policy & Algorithms***, LNCS 1029, Springer, 1995, pp 201-214

[2] Coppersmith, Don, Krawczyk, Hugo, and Mansour, Yishay, ***The Shrinking Generator***, in Advances in Cryptology, CRYPTO'93, Springer-Verlag, 1994, pp 22-39

[3] Desmedt, Yvo, ***Unconditionally Secure Authentication Schemes and Practical and Theoretical Consequences***, Advances in Cryptology - Crypto'85 Proceedings, Springer-Verlag, 1986, pp 42-45

[4] Dobbertin, Hans, Bossalaers, Antoon, Preneel, Bart, ***RIPEMD-160: A Strengthened Version of RIPEMD***, in ***Fast Software Encryption, 3rd international workshop***, Cambridge, UK, February 1996, LNCS 1039, Springer, 1996, pp 71-82

[5] National Institute of Standards and Technology, ***Secure Hash Standard***, Federal Information Processing Standards Publication FIPS PUB 180, U.S. Department of Commerce, 1993 May 11 (also in Federal Register, January 31, 1992)

[6] National Institute of Standards and Technology, ***Proposed Revision of Federal Information Processing Standards (FIPS) 180, Secure Hash Standard***, Federal Register, V. 59, n. 131, 11 July 1994, pp 35317-35318

[7] Suen, W.C. Stephen, ***On Large Induced Trees and Long Induced Paths in Sparse Random Graphs***, Journal of Combinatorial Theory, Series B, Vol. 56 (1992), pp 250-262

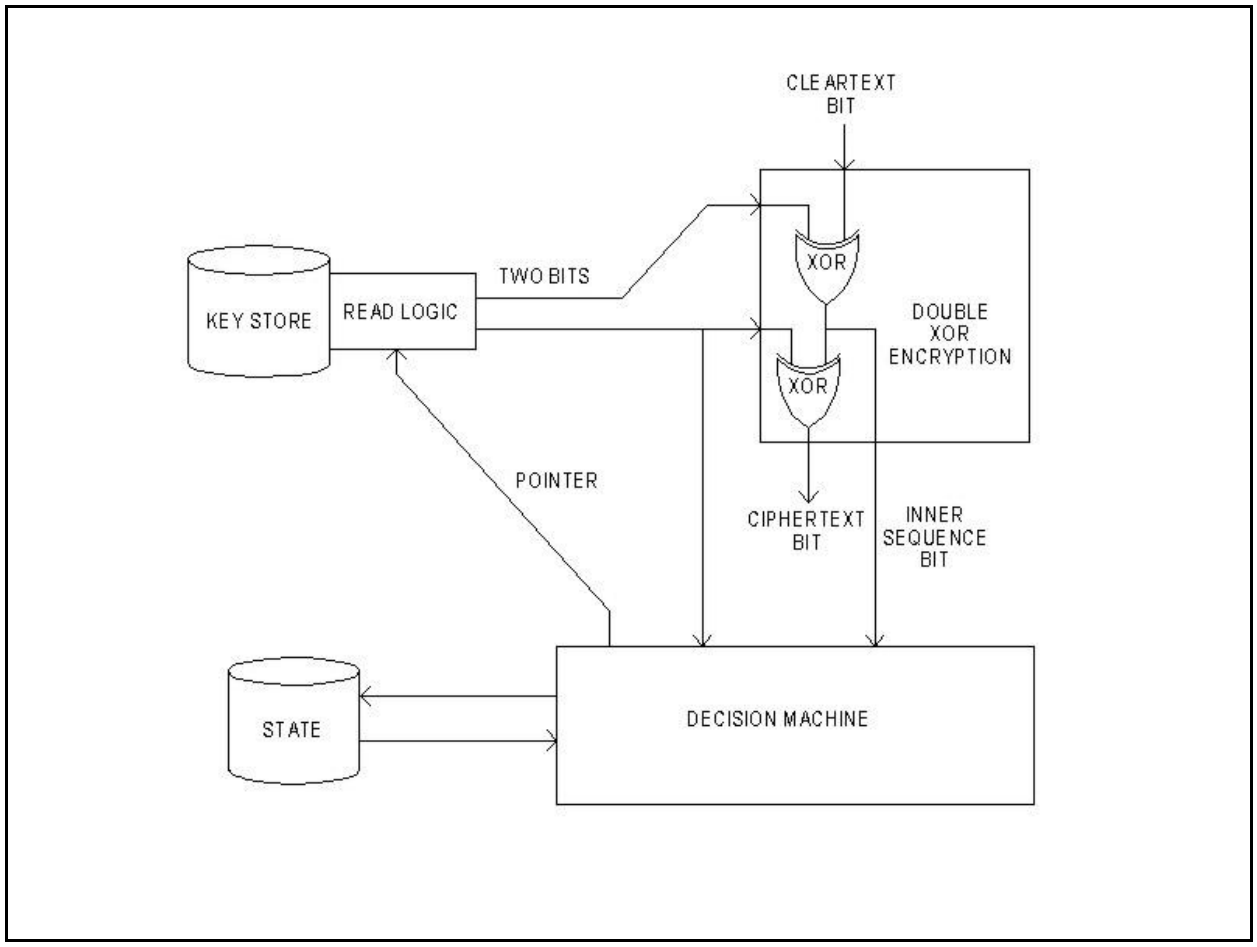


Figure 1) The proposed data integrity arrangement

| $i$ | −1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $m_i$ | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $k1_i$ | | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| $s_i=m_i \oplus k1_i$ | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| $k2_i$ | | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| $e_i=s_i \oplus k2_i$ | | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| r(i) | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 0 | 0 | 0 | 1 | 2 | 0 | 0 |
| r′(i)=r(i) mod 2 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| run length | | 1 | 1 | 1 | | 2 | | >2 | | >2 | | 2 | 1 | 1 | | >2 | 1 | 1 |
| d(i) | | 1 | 0 | 1 | | 4 | | 7 | | 9 | | 5 | 1 | 1 | | 8 | 0 | 0 |
| d′(i) | 0 | 1 | 5 | 2 | 2 | 2 | 2 | 0 | 0 | 8 | 8 | 9 | 8 | 0 | 0 | 8 | 1 | 8 |

| $i$ | −1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $i$ | | −1 | 0 | 1 | | 2 | | 4 | | 6 | | 8 | 10 | 11 | | 12 | 14 | 15 |
| T(i)[0] | 0 | 5 | 4 | 1 | | 9 | | 5 | | 7 | | 5 | 8 | 4 | | 1 | 8 | 6 |
| T(i)[1] | 1 | 8 | 2 | 5 | | 7 | | 4 | | 0 | | 8 | 0 | 9 | | 7 | 4 | 8 |
| T(i)[2] | 2 | 7 | 6 | 7 | | 4 | | 9 | | 1 | | 2 | 6 | 1 | | 6 | 3 | 7 |
| T(i)[3] | 3 | 9 | 3 | 0 | | 5 | | 6 | | 8 | | 4 | 9 | 7 | | 0 | 9 | 3 |
| T(i)[4] | 4 | 3 | 7 | 2 | | 8 | | 2 | | 6 | | 9 | 5 | 3 | | 8 | 7 | 9 |
| T(i)[5] | 5 | 2 | 9 | 8 | | 3 | | 7 | | 9 | | 6 | 3 | 5 | | 9 | 2 | 5 |
| T(i)[6] | 6 | 4 | 5 | 9 | | 6 | | 3 | | 2 | | 0 | 1 | 0 | | 2 | 6 | 1 |
| T(i)[7] | 7 | 0 | 1 | 6 | | 0 | | 1 | | 5 | | 1 | 4 | 6 | | 4 | 5 | 4 |
| T(i)[8] | 8 | 6 | 0 | 3 | | 2 | | 0 | | 3 | | 7 | 2 | 8 | | 5 | 1 | 0 |
| T(i)[9] | 9 | 1 | 8 | 4 | | 1 | | 8 | | 4 | | 3 | 7 | 2 | | 3 | 0 | 2 |
| $<k1_i,k2_i>$, gen. 0 | | 11 | 01 | 01 | 11 | 10 | 11 | 10 | 10 | 01 | 00 | 00 | 11 | 10 | 01 | 01 | 11 | 10 |
| $<k1_i,k2_i>$, gen. 1 | | 00 | 10 | 01 | 00 | 10 | 00 | 11 | 11 | 01 | 11 | 11 | 00 | 00 | 00 | 10 | 11 | 01 |
| $<k1_i,k2_i>$, gen. 2 | | 11 | 00 | 00 | 01 | 10 | 10 | 00 | 10 | 11 | 11 | 11 | 10 | 10 | 01 | 11 | 00 | 10 |
| $<k1_i,k2_i>$, gen. 3 | | 11 | 00 | 11 | 10 | 01 | 01 | 01 | 10 | 01 | 10 | 01 | 01 | 00 | 10 | 01 | 10 | 11 |
| $<k1_i,k2_i>$, gen. 4 | | 11 | 10 | 01 | 11 | 10 | 01 | 01 | 00 | 11 | 10 | 10 | 11 | 01 | 10 | 01 | 00 | 00 |
| $<k1_i,k2_i>$, gen. 5 | | 11 | 00 | 10 | 10 | 01 | 01 | 00 | 10 | 10 | 00 | 01 | 01 | 01 | 10 | 11 | 00 | 11 |
| $<k1_i,k2_i>$, gen. 6 | | 11 | 11 | 01 | 10 | 00 | 00 | 10 | 10 | 01 | 10 | 00 | 01 | 11 | 10 | 11 | 11 | 11 |
| $<k1_i,k2_i>$, gen. 7 | | 00 | 11 | 01 | 01 | 00 | 00 | 01 | 00 | 11 | 00 | 00 | 01 | 10 | 10 | 11 | 00 | 10 |
| $<k1_i,k2_i>$, gen. 8 | | 01 | 11 | 11 | 00 | 11 | 10 | 11 | 00 | 01 | 01 | 01 | 01 | 10 | 11 | 11 | 11 | 01 |
| $<k1_i,k2_i>$, gen. 9 | | 01 | 00 | 01 | 10 | 10 | 11 | 01 | 11 | 11 | 11 | 11 | 11 | 00 | 00 | 11 | 00 | 01 |
| position, gen. 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 3 | 3 | 3 | 3 | 3 | 4 | 5 | 5 | 5 |
| position, gen. 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| position, gen. 2 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| position, gen. 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| position, gen. 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| position, gen. 5 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| position, gen. 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| position, gen. 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| position, gen. 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 4 | 4 |
| position, gen. 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |

Table 1) A numerical example of the Frogbit cipher

| Inputs | | | | | Outputs | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| External | | Internal memory | | | | | | Internal memory | | |
| | | | | | Externally used | | | | | |
| $k2_i$ | $s_i$ | $k2_{i-1}$ | $r'(i-1)$ | $s_{i-1}$ | $d_3(i)$ | $d_2(i)$ | $d_1(i)$ | $d_0(i)$, $k2_i$ | $r'(i)$ | $s_i$ |
| 0 | 0 | 0/1 | 0 | 0 | ? | ? | ? | 0 | 1 | 0 |
| 0 | 0 | 0/1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0/1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0/1 | 0 | 1 | ? | ? | ? | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0/1 | 0 | 0 | ? | ? | ? | 1 | 1 | 0 |
| 1 | 0 | 0/1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0/1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0/1 | 0 | 1 | ? | ? | ? | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |

Table 2) The logic table for the Frogbit run length encoding

| d′(i-1) | $s_{i-1}$ | P[j,0] | P[j,1] | P[j,2] | P[j,3] | P[j,4] | P[j,5] | P[j,6] | P[j,7] | P[j,8] | P[j,9] |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 5 | 8 | 7 | 9 | 3 | 2 | 4 | 0 | 6 | 1 |
| 0 | 1 | 2 | 3 | 7 | 6 | 8 | 1 | 9 | 0 | 5 | 4 |
| 1 | 0 | 6 | 0 | 4 | 2 | 3 | 7 | 8 | 1 | 9 | 5 |
| 1 | 1 | 6 | 5 | 8 | 4 | 2 | 3 | 0 | 9 | 7 | 1 |
| 2 | 0 | 3 | 2 | 0 | 6 | 8 | 1 | 5 | 9 | 7 | 4 |
| 2 | 1 | 6 | 2 | 9 | 1 | 5 | 8 | 7 | 3 | 4 | 0 |
| 3 | 0 | 4 | 8 | 9 | 7 | 3 | 0 | 1 | 6 | 5 | 2 |
| 3 | 1 | 5 | 9 | 3 | 7 | 0 | 4 | 2 | 1 | 6 | 8 |
| 4 | 0 | 8 | 0 | 1 | 2 | 6 | 3 | 7 | 4 | 9 | 5 |
| 4 | 1 | 9 | 4 | 6 | 5 | 2 | 3 | 0 | 8 | 1 | 7 |
| 5 | 0 | 7 | 6 | 4 | 8 | 1 | 9 | 5 | 2 | 3 | 0 |
| 5 | 1 | 1 | 4 | 8 | 0 | 7 | 2 | 9 | 5 | 3 | 6 |
| 6 | 0 | 7 | 4 | 5 | 0 | 9 | 8 | 3 | 6 | 1 | 2 |
| 6 | 1 | 3 | 9 | 0 | 5 | 6 | 2 | 8 | 4 | 1 | 7 |
| 7 | 0 | 8 | 9 | 1 | 0 | 2 | 7 | 5 | 6 | 4 | 3 |
| 7 | 1 | 9 | 5 | 4 | 1 | 7 | 0 | 3 | 8 | 2 | 6 |
| 8 | 0 | 7 | 3 | 6 | 9 | 5 | 4 | 1 | 2 | 0 | 8 |
| 8 | 1 | 4 | 7 | 9 | 5 | 1 | 6 | 2 | 8 | 0 | 3 |
| 9 | 0 | 2 | 7 | 3 | 8 | 9 | 6 | 4 | 5 | 0 | 1 |
| 9 | 1 | 1 | 6 | 5 | 4 | 0 | 9 | 7 | 3 | 2 | 8 |

Table 3) The permutation table for the Frogbit cipher