Office de la Propriété Intellectuelle du Canada
Un organisme d'Industrie Canada

Canadian Intellectual Property Office
An agency of Industry Canada

CA 2546148 A1 2007/11/09

(21) 2 546 148

(12) DEMANDE DE BREVET CANADIEN
CANADIAN PATENT APPLICATION

(13) A1

(22) Date de dépôt/Filing Date: 2006/05/09

(41) Mise à la disp. pub./Open to Public Insp.: 2007/11/09

(51) Cl.Int./Int.Cl. *H04L 9/28* (2006.01), *H04L 12/54* (2006.01), *H04L 9/32* (2006.01)

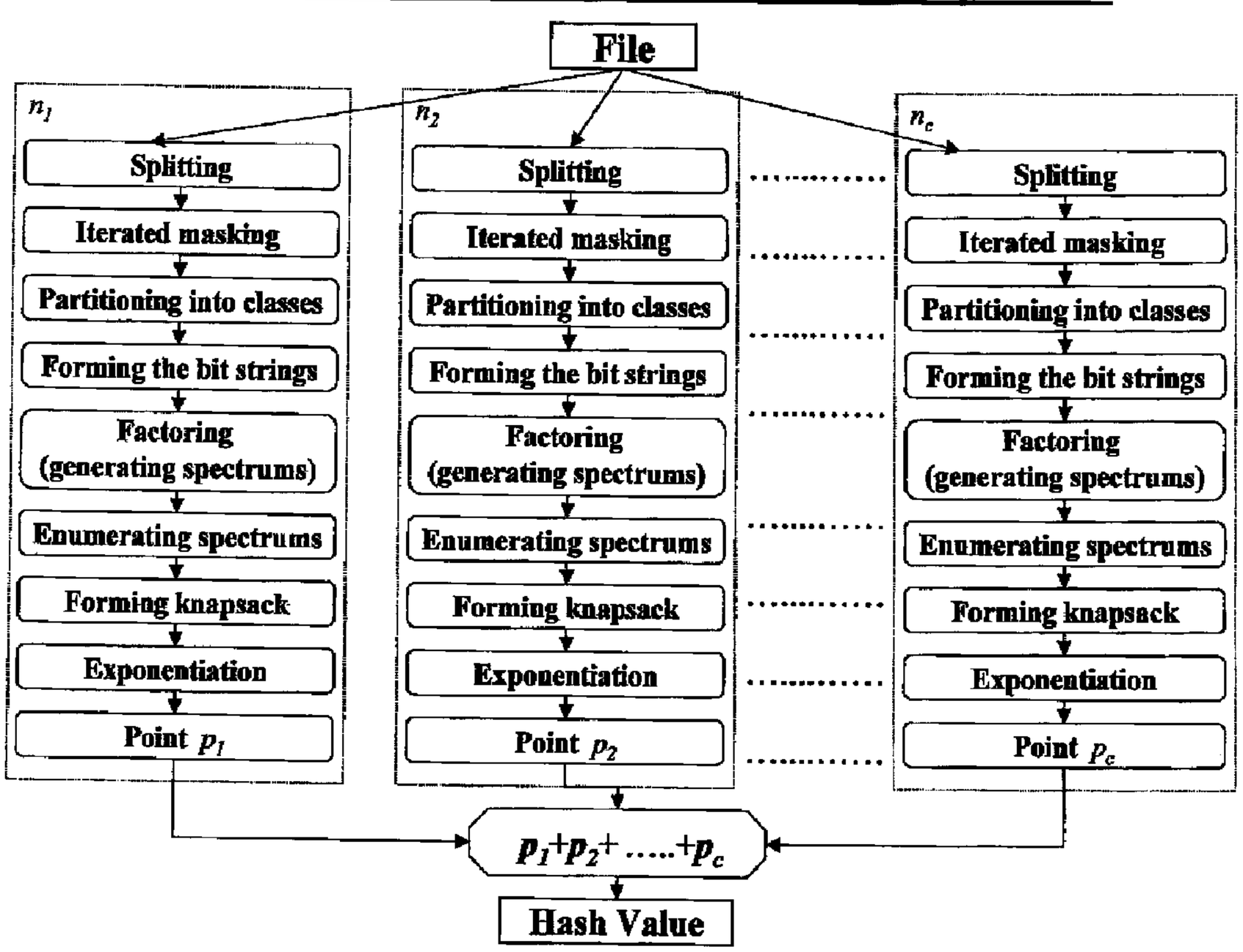
(71) Demandeurs/Applicants:
VOLKOVS, NIKOLAJS, CA;
MURTY, VIJAYA KUMAR, CA

(72) Inventeurs/Inventors:
VOLKOVS, NIKOLAJS, CA;
MURTY, VIJAYA KUMAR, CA

(74) Agent: BORDEN LADNER GERVAIS LLP

(54) Titre : METHODE, SYSTEME ET PROGRAMME D'ORDINATEUR POUR CODAGE PAR ADRESSAGE CALCULE ET AUTHENTIFICATION DE MESSAGES A BASE DE FONCTIONS POLYNOMIALES AVEC PRODUCTION DISTINCTE DE SPECTRES

(54) Title: METHOD, SYSTEM AND COMPUTER PROGRAM FOR POLYNOMIAL BASED HASHING AND MESSAGE AUTHENTICATION CODING WITH SEPARATE GENERATION OF SPECTRUMS



(57) Abrégé/Abstract:
A secure hashing method is provided consisting of: (1) representing an initial sequence of bits as a specially constructed set of polynomials as described herein, (2) transformation of this set by masking, (3) partitioning the transformed set of polynomials into a

(21) 2 546 148

(13) A1

(57) Abrégé(suite)/Abstract(continued):
plurality of classes, (4) forming the bit string during the partitioning, (5) for each of the plurality of classes, factoring each of the polynomials and, so as to define a set of irreducible polynomials, collecting these factors in registers defined for each of the plurality of classes, (6) wrapping the values of the registers from the plurality of classes by means of an enumeration, (7) organizing the enumerations and the bit strings into a knapsack, and, finally, (8) performing an exponentiation in a group to obtain the hash value or the MAC value.

ABSTRACT

A secure hashing method is provided consisting of: (1) representing an initial sequence of bits as a specially constructed set of polynomials as described herein, (2) transformation of this set by masking, (3) partitioning the transformed set of polynomials into a plurality of classes, (4) forming the bit string during the partitioning, (5) for each of the plurality of classes, factoring each of the polynomials and, so as to define a set of irreducible polynomials, collecting these factors in registers defined for each of the plurality of classes, (6) wrapping the values of the registers from the plurality of classes by means of an enumeration, (7) organizing the enumerations and the bit strings into a knapsack, and, finally, (8) performing an exponentiation in a group to obtain the hash value or the MAC value.

METHOD, SYSTEM AND COMPUTER PROGRAM FOR POLYNOMIAL BASED HASHING AND MESSAGE AUTHENTICATION CODING WITH SEPARATE GENERATION OF SPECTRUMS

Field of Invention

The present invention relates generally to data communication and storage. More specifically, the present invention relates to systems and method for improving the security associated with data integrity and authentication.

Background of Invention

A hash function is best understood as a map that sends binary strings of arbitrary length to binary strings of length τ (or hash length).

$$H: \{0,1\}^* \rightarrow \{0,1\}^\tau$$

This τ value is fixed in advance. Commonly used hash functions have a τ value that varies from 32 to 512 [5].

A Message Authentication Code (or MAC) is a hash function that incorporates a key, namely:

$$H: \{0,1\}^* \times \mathbf{K} \rightarrow \{0,1\}^\tau$$

where **K** is a key space. When a user sends data, the hash value, or MAC value of the data is also calculated and appended to the message. The recipient can then verify the integrity of the data by recomputing the hash value or MAC value and comparing it with the one that was appended to the message, thereby enabling the recipient, for example, to authenticate the message.

One of the challenges in providing hash-based data integrity solutions is that the hash value needs to be efficiently computable, and collisions should be improbable. Specifically, given a binary string M, it should be computationally infeasible to find another string M', satisfying the following equation:

$$H(M) = H(M')$$

Hash and MAC algorithms are extremely important and at the same time the most vulnerable systems of network security [5]. If a hash or MAC value can be determined by an external agency, a "collision" has occurred. If hash or MAC values are subject to collisions, the recipient of data can never be certain that the data in question has not bee tampered with or corrupted. Its collision resistance measures the value of a hash (MAC) algorithm. Since the algorithm produces a string of fixed length from a message of any length, it is clear that there will be collisions. However, a good hash algorithm is one for which it is computationally infeasible to create a collision.

In a recent dramatic development, all the main hash algorithms MD-5, RIPEMD, and the MAC algorithm SHA-1 were compromised. Collisions were created for MD-5 and RIPEMD, and a group of Chinese mathematicians managed to reduce the number of operations needed to realise the brute-force attack on SHA-1 to a danger level. It should be noted that SHA-1 is the hash algorithm currently recommended by the US government. Keeping in mind that a lot of different security applications (Kerberos, MIME, IpSec, Digital Signatures and so forth) are using hash algorithms (mainly SHA-1), there is an urgent need to construct new hash algorithms.

All the main hash functions and secure functions, including those mentioned above, are referred to as iterated hash functions. They are based on an idea proposed by Damgard-Merkle [10]. According to this idea, the hash function takes an input string of bits, and partitions it into fixed-sized blocks of a certain size q. Then a compression function takes q bits of i-th partition and m bits from the previous calculation and calculates m bits of i+1 iteration. The output value of the last iteration (of size m) is the hash value.

Since it now appears to be easier to create a collision in existing main hash functions and secure functions, the development of new hashing algorithms that would not be based on the Damgard-Merkle approach, is extremely desirable.

Various authors have considered hash algorithms when a message to be hashed is presented as a class of various algebraic objects - elements of fields, groups, etc. Hashing based on polynomial representation is known. One of the most famous approaches is to present data as a collection of polynomials over the field of a certain characteristic. Carter and Wegman [11] proposed the

method of presenting a message as coefficients of a polynomial, and a certain point of evaluation of the polynomial is used as a key.

Krovetz and Rogaway [7] considered a message as a collection of elements (m_0, m_1, ..., m_n) of a certain field F. A hash value of the message is a point $y \in F$, which is computed by calculation in the field F of the value $m_0k^n + m_1k^{n-1} + ... + m_nk^0$ for some key $k \in F$.

However, these approaches do not represent the data in terms of polynomials, nor do they compute the hash value using the factorization of these polynomials.

There is a need therefore for methods, computer programs and computer systems that while utilizing hash or MAC algorithms (in particular algorithms of the SHA family) are operable to provide an improved level of security over existing methods, computer programs and computer systems that implement SHA type hash and MAC algorithms. There is a further need for the methods, computer programs and computer systems that meet the aforesaid criteria and are further easy to implement with existing technologies, and are computationally feasible.

Summary of Invention

In one aspect of the present invention, a secure hashing method is provided consisting of: (1) representing an initial sequence of bits as a specially constructed set of polynomials as described herein, (2) transformation of this set by masking, (3) partitioning the transformed set of polynomials into a plurality of classes, (4) forming the bit string during the partitioning, (5) for each of the plurality of classes, factoring each of the polynomials and, so as to define a set of irreducible polynomials, collecting these factors in registers defined for each of the plurality of classes, (6) wrapping the values of the registers from the plurality of classes by means of an enumeration, (7) organizing the enumerations and the bit strings into a knapsack, and, finally, (8) performing an exponentiation in a group to obtain the hash value or the MAC value.

Step (8) above may consist of the calculation of α^V in the field $\mathbf{F}_{2^\tau}$, where α is a generator of the multiplicative group $\mathbf{F}_{2^\tau}^{\times}$. Alternatively, one may consider $V\gamma$, where E is an elliptic curve over $\mathbf{F}_{2^\tau}$ and γ is a generator of the group $E(\mathbf{F}_{2^\tau})$. In fact, both of these methods are special cases of a more general scheme in which we consider any abstract cyclic group G with a generator g. We assume that the numeration of the elements of G requires a binary string of the size τ and that the discrete logarithm problem in G is difficult.

Because of the polynomial representation described above, in order to create a collision in accordance with the secure hash function described above, an attacker would be required to solve a collection of systems of non-linear iterated exponential equations over a finite field having specific constraints. In the case of a MAC, this difficulty is combined with the difficulty of opening the knapsack, and the difficulty of (a) solving the elliptic curve discrete logarithm referred to below, or (b) the discrete logarithm problem in the finite field, and this further contributes to the security of the method of the present invention.

As a result of the structure of the procedure, the resulting hash or MAC value has the following important attributes:

a) the length of the output can be changed simply by changing the final step;

b) the computation is a bit-stream procedure as opposed to a block procedure;

c) creating a collision requires the solution to several difficult mathematical problems; and

d) varying some parameters (the number of the bit strings, or the length of the bit strings, for example) allows easy variation of the difficulty of creating a collision.

Brief Description of Drawings

A detailed description of the preferred embodiment(s) is(are) provided herein below by way of example only and with reference to the following drawings, in which:

Figure 1 "Scheme of hashing with separate generation of spectrums" is a flowchart diagram illustrating the steps involved in the present invention in connection with secure hashing.

Figure 2 "Hashing with iterations" is a flowchart diagram illustrating the steps involved in the present invention with consideration of rounds (iterations) in the process of the calculation of a hash value.

Figure 3 "Message authentication coding with iterations" is a flowchart diagram illustrating the steps involved in the present invention with consideration of rounds (iterations) in the process of the calculation of a MAC value.

In the drawings, preferred embodiments of the invention are illustrated by way of example. It is to be expressly understood that the description and drawings are only for the purpose of illustration and as an aid to understanding, and are not intended as a definition of the limits of the invention.

Detailed Description of the Invention

The secure hashing method more particularly consists of: (1) representing an initial sequence of bits as a specially constructed set of polynomials as described herein, (2) transformation of this set by masking, (3) partitioning the transformed set of polynomials into a plurality of classes, (4) forming the bit string during the partitioning, (5) for each of the plurality of classes, factoring each of the polynomials and, so as to define a set of irreducible polynomials, collecting these factors in registers defined for each of the plurality of classes, (6) wrapping the values of the registers from the plurality of classes by means of an enumeration, (7) organizing the enumerations and the bit strings into a knapsack, and finally, (8) performing an exponentiation in a group to obtain the hash value or the MAC value.

A number of notations used in this disclosure should be understood. $M \in \{0,1\}^*$ shall represent a sequence of bits, where $\{0,1\}^*$ is the set of all possible finite strings. $|M|$ shall represent the length of M in bits. As usual, $M[k]M[l]$ is the concatenation of two bits from M, namely the k-th and l-th bits. We will write $M(i, j)$ for the sequence of bits $M[i]M[i + 1] \ldots M[j]$, if $i \leq j$, $j \leq |M|$ or $M[j]M[j\text{-}1] \ldots M[i]$ if $j \leq i$, $i \leq |M|$. By $M[k, i]M[l, j]$ we denote the concatenation of two strings $M[k, i]$ and $M[l, j]$. If M and M_1 are two strings such that $|M| = |M_1|$ then $M \oplus M_1$ is their bitwise *xor*. For a sequence of bits $M(i, j)$ denote by $\overline{M(i, j)}$ the complement string. Where there is no danger of confusion, we will denote $M(i, j)$ by M_i.

The method of the present invention uses polynomials of some fixed degree, say n. To make this computationally efficient, we choose an n satisfying $4 \leq n \leq 10$. In this range, we can store a lookup table to make factorization of polynomials very fast. We could choose a larger value of n if we wished, provided we had more memory and computational power.

Irreducible polynomials of degree m are obtained as follows. For $m \geq 1$, the number of irreducible polynomials of degree m over $\mathbf{F}_2$ is given by the well-known formula

$$\frac{1}{m}\sum_{d/m}\mu(m/d)2^d$$

where μ is the Möbius function. For $m = 0$, we have 2 "polynomials", namely the two elements of the field. In particular, we have the following table of values for small values of m:

degree	Number of irreducible polynomials
0	2
1	2
2	1
3	2
4	3

Denote by $J(n)$ the lexicographically ordered set of all irreducible polynomials of degree less than n with coefficients in $\mathbf{F}_2$, the finite field of two elements. Denote by $irr(n)$ the cardinality of $J(n)$. Thus, $irr(4) = 7$ and $J(4)$ is the following ordered list:

number	Polynomial
1	0
2	1
3	X
4	x+1
5	$x^2 + x + 1$
6	$x^3 + x + 1$
7	$X^3 + x^2 + 1$

For $h \in \mathbf{F}_{2^n}$, $FC_n(h, i)$ indicates the multiplicity of the i-th irreducible polynomial from $J(n)$ in the factorization of h. For example, for $n = 4$, noting that the third element in $J(4)$ is the polynomial x and the fourth element is $x + 1$, we have

$$FC_4(x^2 + x, 1) = 0, \quad FC_4(x^2 + x, 3) = 1, \quad FC_4(x^3 + x^2, 4) = 1.$$

It is evident that $FC_n(h, 1) = 1$ if and only if $h = 0$ and $FC_n(h,2) = 1$ if and only if $h = 1$. Later, when it is clear from the context, we will omit n and simply write J and $FC(h, i)$.

It should be understood that for any polynomial h over $\mathbf{F}_2$ of degree less than n denote by $int(h)$ an integer number represented by h as a binary number. For instance, $int(x^3+x^2+x) = 14$ and $int(x^3 + 1) = 9$. In general, if

$$h = \alpha_0 x^{n-1} + \alpha_1 x^{n-2} + \ldots + \alpha_{n-1}$$

is a polynomial with coefficients $a_0,\ldots, a_{n-1}$ from $\mathbf{F}_2$, then let us define the integer $\tilde{a}$ by

$$\tilde{a}_i = \begin{cases} 0 & \text{if } a_i = 0 \text{ in } \mathbf{F}_2 \\ 1 & \text{If } a_i = 1 \text{ in } \mathbf{F}_2. \end{cases}$$

Then,

$$int(h) = \tilde{a}_0 2^{n-1} + \tilde{a}_1 2^{n-2} + \ldots + \tilde{a}_{n-1} 2^0.$$

Padding

It should be understood that a sequence of bits could be empty or could be of a small size, and therefore there may be a need for padding. Let M be any sequence of bits. Denote $k = |M|$. As M can be empty we describe the padding procedure. If k is greater than or equal to 4096 bits, the sequence is not changed. Otherwise we pad M to 4096 bits sequence as follows. We choose the smallest m such that the total number of the elements of a Gray code of size m, is bigger than $(4096-k)/m$ and, after that, we pad $(4096-k)$ bits of M by the elements of a Gray code of size m.

Recall that a Gray code of size m is an ordering of the elements of $\{0, 1\}^m$ in such a way that only one bit changes from one entry to the next. Thus,

00, 01, 11, 10

is a Gray code of size 2,

000, 001, 011, 010, 110, 111, 101,100

is a Gray code of size 3 and

0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101,
1111, 1110, 1010, 1011, 1001, 1000

is a Gray code of size 4.

Note that such padding requires knowing the size of a file beforehand. If we do not know the size of a file, then by repeating the elements of a Gray code of such a size *m* so that the total number of the elements of a Gray code is 4096, we can do the padding.

Splitting

Splitting is best understood as a stage in accordance with the present invention whereby the string of bits is decomposed into smaller, overlaid bit segments or "classes". This decomposition can be achieved in a number of different ways: for example by means of random distribution or other methods.

Let M be any sequence of bits. We assume that the size of M is at least 4096 bits, so the padding of the initial string of bits has already been done if its size was less than 4096 bits. Denote $k = |M|$ and choose some $n_1, n_2, \ldots, n_c$, where $4 \le n_i \le 10$. We split M into the set of the following sequences:

(1) $$M(1, n_1), M(2, n_1 + 1), \ldots, M(k - n_1 + 1, k),$$

$$M(k - n_1 + 2, k)\, M[1], M(k - n_1 + 3, k)M(1,2), \ldots, M[k]M(1, n_1 - 1).$$

$$M(1, n_2), M(2, n_2 + 1), \ldots, M(k - n_2 + 1, k),$$

$$M(k - n_2 + 2, k)\, M[1], M(k - n_2 + 3, k)M(1,2), \ldots, M[k]M(1, n_2 - 1).$$

.....

$$M(1, n_c), M(2, n_c + 1), \ldots, M(k - n_c + 1, k),$$

$$M(k - n_c + 2, k)\, M[1], M(k - n_c + 3, k)M(1,2), \ldots, M[k]M(1, n_c - 1).$$

Note that any sequence $M(i, i + n_j - 1)$ can be considered as a polynomial of degree less than n_j over $\mathbf{F}_2$, that is, if for some $i \le k - n_j + 1$ sequence $M(i, i + n_j - 1)$ is $a_0, a_1, \ldots, a_{n_j-1}$, then the corresponding polynomial is

(2) $$a_0 x^{n_j-1} + a_1 x^{n_j-2} + \ldots + a_{n_j-1}$$

Denote by $S(M, n_1)$, $S(M, n_2)$, ..., $S(M, n_C,)$ the given ordered sets (1) and note that any $S(M, n_j)$ contains k polynomials. Remark also that the just described procedure of forming the sequences $S(M, n_j)$, $j = 1, \ldots c$ is a stream procedure. Of course, in practise, there is no need to consider all sets $S(M, n_j)$ for $j = 1, ..., c$, as the time of the calculation of a hash (or MAC) value in that case will be increased, which will be discussed below.

Example I. As an example, consider the following sequence of bits M without padding:

011010

Let $n_1 = 4$ and $n_2 = 6$. Let the procedure of forming sequences be $S(M, 4)$ and $S(M, 6)$. So, the elements of $S(M, 4)$ are

$$x^2 + x, x^3 + x^2 + 1, x^3 + x, x^2, x^3 + 1, x + 1$$

while $S(M, 6)$ contains elements

$$x^4 + x^3 + x, x^5 + x^4 + x^2, x^5 + x^3 + 1, x^4 + x + 1, x^5 + x^2 + x, x^3 + x^2 + 1.$$

Now, we want to express relations between the elements of $S(M, n_j)$ for different j.

Consider some n_j, $j = 1, \ldots, c$ and the corresponding sequence of polynomials $S(M, n_j)$. It is not hard to see that for any $i = 1, \ldots, k$ we get

$$M_{i+1} = AM_i^T \oplus BM_{i+2}^T, \tag{3}$$

where

$$A = \begin{pmatrix} 0 & 1 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 1 & \ldots & 0 & 0 \\ 0 & 0 & 0 & \ldots & 0 & 0 \\ \ldots\ldots\ldots & & & & & \\ 0 & 0 & 0 & \ldots & 0 & 1 \\ 0 & 0 & 0 & \ldots & 0 & 0 \end{pmatrix},$$

and

$$B = \begin{pmatrix} 0 & 0 & 0 & \dots & 0\,0 \\ 0 & 0 & 0 & \dots & 0\,0 \\ 0 & 0 & 0 & \dots & 0\,0 \\ & \dots\dots & & & \\ 0 & 0 & 0 & \dots & 0\,0 \\ 0 & 0 & 0 & \dots & 1\,0 \end{pmatrix},$$

are n_j x n_j matrices, M_i^T and M_{i+2}^T are transposed vectors.

Consider two sequences $S(M, n_{j_1})$, and $S(M, n_{j_2})$, where $n_{j_1} < n_{j_2}$. Denote elements from $S(M, n_{j_1})$ and $S(M, n_{j_2})$ by $M_i^{n_{j_1}}$ and $M_i^{n_{j_2}}$, correspondingly. Then the following hold:

(4)
$$M_i^{n_{j_1}} = D_1 (M_i^{n_{j_2}})^T,$$
$$M_{i+1}^{n_{j_1}} = D_2 (M_i^{n_{j_2}})^T,$$
$$\dots\dots\dots\dots\dots\dots\dots\dots,$$
$$M_{i+q}^{n_{j_1}} = D_q (M_i^{n_{j_2}})^T,$$

where, $q = n_{j_2} - n_{j_1}$, and

$$D_1 = \begin{pmatrix} 1 & 0 & 0 & \dots & 0\,0 \\ 0 & 1 & 0 & \dots & 0\,0 \\ 0 & 0 & 1 & \dots & 0\,0 \\ & \dots\dots & & & \\ 0 & 0 & 0 & \dots & 0\,0 \\ 0 & 0 & 0 \dots & 1 \dots & 0\,0 \end{pmatrix},$$

$$D_2 = \begin{pmatrix} 0 & 1 & 0 & \dots & 0\,0 \\ 0 & 0 & 1 & \dots & 0\,0 \\ 0 & 0 & 0 & \dots & 0\,0 \\ & \dots\dots & & & \\ 0 & 0 & 0 & \dots & 0\,0 \\ 0 & 0 & 0 \dots & 01 \dots & 0\,0 \end{pmatrix},$$

$$\dots\dots\dots\dots\dots\dots\dots,$$

$$D_q = \begin{pmatrix} 0 & 0 & 0 \dots & 1 \dots & 0\,0 \\ 0 & 0 & 0 & \dots & 0\,0 \\ 0 & 0 & 0 & \dots & 0\,0 \\ & \dots\dots & & & \\ 0 & 0 & 0 & \dots & 1\,0 \\ 0 & 0 & 0 & \dots & 0\,1 \end{pmatrix},$$

are $n_{j_1} \times n_{j_2}$ matrices and $(M_i^{n_{j_2}})^T$ is a transposed vector $M_i^{n_{j_2}}$.

Present one more way of splitting. We will refer it to as a contrary splitting.

Again, let M be any sequence of bits and we assume that it has the size at least 4096 bits. Choose some $n_1, n_2, \dots, n_c$, where $4 \le n_i \le 10$, $c \ge 2$. Let us divide the set of the elements $\{n_1, n_2, \dots, n_c\}$ into two parts, N_1 and N_2. As before, we split M into the set of the sequences in a form (1) for the elements, say, from N_1. At the same time, for the elements from N_2, the splitting will be different. Let us fix some n from N_2. Without loosing generality, we show how to organize the splitting for the fixed n, the index of which will be omitted. So we form:

(1`) $M[k]M[k-1]\dots M[k-n+1], M[k-n]M[k-n-1]\dots M[k-2n+1], \dots, M[n-1]\dots M[2]M[1],$

$M[n-2]M[n-3]\dots M[1]M[k], M[n-3]\dots M[1]M[k]M[k+1], \dots, M[1]M[k]\dots M[k-n+2].$

Again, we generate the sequences (1`) for all the elements from $N_2$. In contrary splitting, we need to have at least one sequence in form (1`). Note that the division of the set $\{n_1, n_2, \dots, n_c\}$ into subsets N_1 and N_2 can be done arbitrarily.

Multiple Masking

The elements of the sets $S(M, n_1)$, $S(M, n_2)$, ..., $S(M, n_C)$ will be further transformed in accordance with a multiple masking procedure. Masking is a transformation of the elements of sequence (1). In fact, masking is a non-linear function that generates polynomials, having as input polynomials that are obtained during the splitting. The main aim of masking is to provide a non-linear transformation of the polynomials from 1. Such a transformation forces an attacker to solve the system of non-linear equations in order to obtain the elements of sequence (1), based on transformed polynomials. So, a non-linear function can be used for masking if the calculation of values of the function is fast and results in a system of non-linear equations over the finite fields. We have chosen a function that is quite fast and that results in a system of exponential iterated equations. Below, we describe the construction of the function.

Based on the sequence of sets $S(M, n_1)$, $S(M, n_2)$, ..., $S(M, n_c)$ we prepare c registers CUR, so that, for any sequence $S(M, n_j)$, we associate sequence CUR^{n_j}. We need to show how to calculate the values of the registers and we describe the procedure for a fixed n_j. Therefore, we shall omit the upper index n_j for $CU\,R^{n_j}$ and instead of n_j, we sometimes will use n.

We describe the calculation of the values of CUR iteratively. Let δ and β be generators of $GF(2^n)$. We set

$$CUR_1 = M_1 \oplus \delta \oplus \beta,$$

$$CUR_2 = M_2 \oplus \delta^{\mathrm{int}(M_1)} \oplus \beta^{\mathrm{int}(CUR_1)},$$

$$CUR_3 = M_3 \oplus \delta^{(\mathrm{int}(M_2)+\mathrm{int}(M_1)) \bmod 2^n} \oplus \beta^{(\mathrm{int}(CUR_2)+\mathrm{int}(CUR_1)) \bmod 2^n},$$

.....

$$CUR_i = M_i \oplus \delta^{(\mathrm{int}(M_{i-1})+\mathrm{int}(M_{i-2})) \bmod 2^n} \oplus \beta^{(\mathrm{int}(CUR_{i-1})+\mathrm{int}(CUR_{i-2})) \bmod 2^n}$$

for $i = 1,\ldots,2^n+1$.

For $i = 2^n + 2,..., k$ we calculate CUR_i in accordance with

$$CUR_i = M_i \oplus \delta^{(\mathrm{int}(M_{i-1})+\mathrm{int}(M_{d_1}))\bmod 2^n} \oplus \beta^{(\mathrm{int}(CUR_{i-1})+\mathrm{int}(CUR_{d_2}))\bmod 2^n},$$

where

$$d_1 = i-2-\mathrm{int}(M_{i-1}), \quad d_2 = i-2-\mathrm{int}(CUR_{i-1}).$$

We stress that the procedure just described for calculating the values CUR_i is, in fact, a stream procedure.

We note that for the manipulations above, we are viewing F_{2^n} as a field and also as a vector space over F_2. Let $f(x)$ be a polynomial so that there is an isomorphism of fields.

Denote by ϕ_f the isomorphism of vector spaces

$$F_2[x]/(f(x)) \to F_2^n.$$

Let δ and β be generators of $F_{2^n}^{\times}$ corresponding to polynomials $f(x)$ and $g(x)$. We set

$$CUR_1 = M_1 \oplus \phi_f(\delta) \oplus \phi_g(\beta),$$

$$CUR_i = M_i \oplus \phi_f(\delta^{(\mathrm{int}(M_{i-1})+\mathrm{int}(M_{i-2}))\bmod 2^n}) \oplus \phi_g(\beta^{(\mathrm{int}(CUR_{i-1})+\mathrm{int}(CUR_{i-2}))\bmod 2^n})$$

for $i = 1,...,2^n+1$. For $i = 2^n + 2,..., k$ we calculate CUR_i in accordance with

$$CUR_i = M_i \oplus \phi_f(\delta^{(\mathrm{int}(M_{i-1})+\mathrm{int}(M_{d_1}))\bmod 2^n}) \oplus \phi_g(\beta^{(\mathrm{int}(CUR_{i-1})+\mathrm{int}(CUR_{d_2}))\bmod 2^n}),$$

where

(5) $$d_1 = i-2-\mathrm{int}(M_{i-1}), \quad d_2 = i-2-\mathrm{int}(CUR_{i-1})$$

As it was mentioned above, we calculate the values of registers CUR^{n_j} for all the elements of the sequence $S(M,n_1),S(M,n_2),...,S(M,n_c)$. Of course, for different n_j we use different fields $F_{2^{n_j}}$ with the corresponding generators. Thus, we have prepared c sets of collections of polynomials

(6) $$CUR_i^{n_1},CUR_i^{n_2},...,CUR_i^{n_c},$$

where $i=1,...,k$. It is clear that the combination of splitting and masking is a stream procedure.

Partitioning

So, c sets of the sequences of polynomials (6) for i = 1, ..., k of the corresponding degrees are prepared. We choose some $r_j \geq 200$. The value of r_j will define the number of classes, on which we will split each sequence. On the other hand, r_j will also define the number of elements of a knapsack that we are going to form. In order to construct the knapsack of a cryptographic significance (for constructing, say MAC) we choose r_j not less than 200. Now we need to distribute the elements of all the sequences (6) between the partitions. It can be done in many ways.

One of the simplest ways of such a distribution is to assign the first k/r_j elements to the first partition, then next k/r_j elements to the second partition and so forth. The last partition in that case will contain $k \bmod r_j$ polynomials.

Let us present another possibility of the partitioning. Firstly, we fix some number nm_1 of polynomials from CUR^{n_1} and associate them with the first partition. The next nm_1 polynomials are associated with the second partition and so forth. In that case, we will have $k/nm_1 + 1$ partitions for sequence CUR^{n_1}. Then, we consider nm_2 polynomials per partition for sequence CUR^{n_2}, nm_3 polynomials in one partition for CUR^{n_2} and so forth. In general, for a sequence CUR^{n_j}, we choose nm_j polynomials in one partition, which means that the sequences of polynomials (11) will be split into

$$k/nm_1 + 1,\ k/nm_2 + 1,\ ...,\ k/nm_c + 1$$

partitions, correspondingly.

For hardware implementation of the presented hash and MAC system, an efficient way of distributing the elements of sequences (6) between partitions can be defined as follows. We choose some number z greater than or equal to 1 and assign the first z elements of the sequences to the first partition, the next z elements to the second partition and so forth. When the last r_j-th partition is considered we return to the very first partition and the next z elements we assign to it, then we use the second partition and so on. In the case of MAC number z can be defined by (part of) a key.

Note that all the methods presented above of partitioning the polynomials from the sequences (6) have one specific feature, namely, a few (neighbouring) polynomials can be assigned to one and the same partition, say, in a framework of one iteration.

Now we want to present a few methods of the partitioning, usage of which will guarantee that any two neighbouring elements cannot be sent to one and the same partition, during one iteration.

The next few possibilities of partitioning are based on the calculation of special indices. We calculate c sets of indices in accordance with

$$ind_1^{n_j}(1) = \mathrm{int}(CUR_1^{n_j} + 1) \bmod r_j + 1, \tag{7}$$

$$ind_1^{n_j}(2) = (ind_1^{n_j}(1) + \mathrm{int}(CUR_2^{n_j}) + 1) \bmod r_j + 1$$

.....

$$ind_1^{n_j}(i) = (ind_1^{n_j}(i-1) + \mathrm{int}(CUR_i^{n_j}) + 1) \bmod r_j + 1$$

for $j = 1,...,c, i = 3,...,k$. We need to assign to each value $CUR_i^{n_j}$ one of r_j partitions, where $j = 1, ...,c$, $i = 1, ..., k$. For $CUR_i^{n_j}$ we calculate the value $ind_1^{n_j}(i)$ in accordance with (7), and the value will define the partition, to which $CUR_i^{n_j}$ will be assigned.

Consider a few more possibilities to construct the index of the partitioning

(8) $$ind_2^{n_j}(1) = (\mathrm{int}(CUR_1^{n_j} \oplus M_1) + 1) \bmod r_j + 1 ,$$

$$ind_{n_j}(2) = (ind_{n_j}(1) + \mathrm{int}(CUR_2^{n_j} \oplus M_2) + 1) \bmod r_{ij} + 1 ,$$

.....

$$ind_2^{n_j}(i) = (ind_2^{n_j}(i-1) + \mathrm{int}(CUR_i^{n_j} \oplus M_i) + 1) \bmod r_j + 1 ,$$

or

(9) $$ind_3^{n_j}(1) = (\mathrm{int}(CUR_1^{n_j}) + \mathrm{int}(M_1) + 1) \bmod r_j + 1 ,$$

$$ind_3^{n_j}(2) = (ind_3^{n_j}(1) + \mathrm{int}(CUR_2^{n_j}) + \mathrm{int}(M_2) + 1) \bmod r_j + 1 ,$$

.....

$$ind_3^{n_j}(i) = (ind_3^{n_j}(i-1) + \mathrm{int}(CUR_i^{n_j}) + \mathrm{int}(M_i) + 1) \bmod r_j + 1 ;$$

(10) $$ind_4^{n_j}(1) = (\mathrm{int}(M_1) + 1) \bmod r_j + 1 ,$$

$$ind_4^{n_j}(2) = (ind_4^{n_j}(1) + \mathrm{int}(M_2) + 1) \bmod r_j + 1 ,$$

.....

$$ind_4^{n_j}(i) = (ind_4^{n_j}(i-1) + \mathrm{int}(M_i) + 1) \bmod r_j + 1$$

and so on. Thus, any function of calculating the indices that involves in one way or another the elements of the corresponding $S(M,n_j)$ or CUR^{n_j} can be used for defining the indices. When we consider MAC, such a function can be dependent on a key.

We present one more method of partitioning the polynomials. We will refer to it as the conditional distribution.

To every n_j, associate two numbers $q_j > 0$ and $g_j > 0$. Then, for any j, if $M[1]$ is 1, we assign $CU\ R_1^{n_j}$ to partition q_j, otherwise, to partition $q_j + g_j$, and consider the calculated partition as the current partition. Assume that the index of the current partition is h_j. If, then, $M[2]$ is 1, we assign $CU\ R_2^{n_j}$ to the partition $h_j + q_j$, otherwise, polynomial $CU\ R_2^{n_j}$ will be sent to partition $h_j + q_j + g_j$, and so on. So, if on the i-th steps of the consideration the current partition partition ω_j, then if $M[i+1]$ is 1, we assign polynomial $CU\ R_{i+1}^{n_j}$ to partition $\omega_j + q_j$, otherwise, we send it to partition $\omega_j + q_j + g_j$. If $\omega_j + q_j + g_j$ (or $\omega_j + q_j$) is greater than r_j, where r_j is the number of partitions prepared for the sequence $CU\ R^{n_j}$, we start counting the indices of the partitions from the very beginning. In the simplest case we set $q_j = g_j = 1$.

Forming the bit strings

In a framework of the current subsection we will refer to tables, instead of partitions. We want to show now how to construct special row matrices, which we will name the bit strings. Note that the bit strings are constructed so as not to allow sending any two neighbouring polynomials from any sequence CUR^{n_j} to one and the same table.

Denote by $Tb_i^{n_j}$ the $i-th$ table, related to sequence CUR^{n_j} and by $Tb_{is}^{n_j}$ let us denote the $s-th$, element (entry) of the table, where $s = 0,...,2^{n_j} - 1$, $j = 1,...,c$. In fact a table (or a partition) is a row matrix containing 2^{n_j} elements.

To each table $Tb_i^{n_j}$ we associate, then, in general, 2^{n_j} bit strings e^{n_j} of length sz^{n_j}, that is, the length of the bit strings for different sequences CUR^{n_j} can be different. Denote by $e_{is}^{n_j}$, $j = 1,...,c$, $i = 1,...,r^{n_j}$, $s = 1,...,2^{n_j}$, the bit string that corresponds to the suitable element of $Tb_{is}^{n_j}$ of table $Tb_i^{n_j}$. Denote, at last, by $e_{is}^{n_j}(t), t = 0,...,sz^{n_j}$ the $t-th$ bit of the bit string $e_{is}^{n_j}$. Before the calculation all the bit strings set to 0.

We emphasize the fact that, in general, for different sequences CUR^{n_j} we generate different numbers of tables r^{n_j}.

We say that table $Tb_l^{n_j}$ is activated on the $w-th$ step of the calculation, if the table was chosen by, say, rule (7), that is, if on the $w-th$ step for the corresponding j holds

$$u = ind_1^{n_j}(w).$$

So, if the table $Tb_u^{n_j}$ is activated on the $w-th$ step, the element $Tb_{uv}^{n_j}$, where $v = CUR_w^{n_j}$, will be incremented by 1.

Denote

$$sh_l^{n_j} = \left\lceil \frac{\sum_{k=1}^{l}(\mathrm{int}(CUR_k^{n_j}) + i)}{r_{n_j}} \right\rceil.$$

Assume that on the $w-th$ step of the calculation the table $Tb_u^{n_j}$ is activated and the element $Tb_{uv}^{n_j}$ of the table, where $v = CUR_w^{n_j}$, is incremented by 1. Denote $q = sh_w^{n_j}$.

We want to describe the procedure of changing the values of the bit strings iteratively.

Let $q \le sz^{n_j}$. Recalling that all the bits of all bit strings set to 0 before the calculation, we set

$$e_{us}^{n_j}(q)=1,$$

where $s = v+1$.

Let $q > sz^{n_j}$. We describe three ways of updating the bit strings in that case.

CASE I. We calculate $q' = q \mod sz^{n_j}$ and assume that before the activation of the table $Tb_u^{n_j}$, the value of the bit $e_{us}^{n_j}(q')$ was B. So, we compute the new value of the bit as follows

$$e_{us}^{n_j}(q') = B \oplus 1.$$

CASE II. At the end of each iteration we provide intermediate calculations, save the results of the calculation, set to 0 all bit strings $e_{ik}^{n_j}$, $k = 0,...,2^{n_j}$, and start updating all the bit strings from the very beginning. We will show below how to provide the calculation in that case.

CASE III. We carry out the factorization of the elements of the tables $Tb_i^{n_j}$, $i = 1,..., r^{n_j}$, calculate the final integers, based on the intermediate values of factors and the bit strings, set all considered tables and all bit strings $e_{ik}^{n_j}$, $k = 0,...,2^{n_j}$ to 0, and start generating all the tables and the bit strings from the very beginning. We will show later how to process the computation in that case.

The bit strings play a role of the second, say, "time" dimension for the elements of the tables. Indeed, when we activate the tables, we just increment their values and we do not have any idea about the "time", when the activation took place. Using the bit strings, we fix the "time" of the activation. Saying "time" we mean the value $sh_i^{n_j}$, defined above. In other words, we are interested to know during which iteration of the distribution of elements of CUR^{n_j} between the tables, that or another table was activating.

It is also very important that if we generate all bit strings $e_{is}^{n_j}$, for all $s = 1,...,2^{n_j}$ at least for one fixed n_j, we can restore the whole sequence CUR^{n_j}, based just on the bit strings and the values of the corresponding tables. Note that such a restoration is impossible without this second "time" dimension, that is, without the bit strings. Recall that the sequence CUR^{n_j} is uniquely related to an initial file. Of course, we need in that case to use the procedure of updating the values of the bit strings (and tables), described above as CASE III.

In general, there is no need to generate all the bit strings $e_{is}^{n_j}$, for all $s = 1,...,2^{n_j}$, as it can reduce the speed of the program and will require extra memory. There are possible a few variants of collecting the bit strings.

We prepare just one bit string $e_{i0}^{n_j}$ for each table $Tb_i^{n_j}$ and when the table is activated (no matter which exactly element of the table is incremented), the bit string is updated. In that case, of

course, we are not able to restore the initial sequence CUR^{n_j}, as we will not have the possibility to detect at which "moment" different values of the table were incremented, but we save the memory and time for the calculation of the final hash value.

At the same time we can prepare bit string $e_{i0}^{n_j}$ for each table $Tb_i^{n_j}$ and a few bit strings $e_{ik}^{n_j}, k = 1,..., y_i$, and we will update the bit strings directly just for those elements, for which we prepare the corresponding bit strings, for the rest of the elements of the tables bit string $e_{i0}^{n_j}$ will be updated. Such elements can be selected, say, during the process of forming the tables. So we prepare a bit string $e_{i0}^{n_j}$, and for the first $x < 2^{n_j}$ elements of the tables we will generate the corresponding bit strings. If some of the first elements are repeated, we choose the next "available" element of the table. The number of the prepared bit strings and the elements of the tables, for which the bit strings are prepared, for different tables (even for the same sequence CUR^{n_j}) can be different and can be defined by a key in the case of MAC. We can also consider the situation when for different iterations the number of prepared bit strings will be different.

At last, all bit strings for all the elements of all the tables or of a part of the tables, excluding $e_{i0}^{n_j}$ can be used. This is the case when the features of a file are collected in the most complete way.

When we finish the processing a file, the following methods of the calculation, based on a value of the bit strings, can be considered. Let's start with the case I of updating the bit strings.

At the end of the calculation we obtain $e_{ik}^{n_j}$ bit strings $i = 1,..., r^{n_j}$, $k \in \Theta_i$, where Θ_i is the set of the values of the table $Tb_i^{n_j}$, for which the bit strings are generated, more precisely, if we form a bit string for element $Tb_{is}^{n_j}$, then Θ_i will contain element $s+1$. It is clear that Θ_i contains at least one element 0. If we use just one bit string, namely, $e_{i0}^{n_j}$ we get

$$BS^{n_j}(i) = \mathrm{int}(e_{i0}^{n_j}). \qquad (11)$$

In general, we compute

$$BS^{n_j}(i) = \sum_{x \in \Theta_i} (x+1)\,\mathrm{int}(e_x^{n_j}). \tag{12}$$

CASE II. As we set to 0 all bit strings at the end of each iteration, we need to form BS^{n_j}, which is based on the intermediate values of the bit strings. Again, we fix some j and assume that the number of iterations (the number of setting the bit strings to 0) is num_j. Denote by $IBS_k(u)$, $u = 1,\ldots,r_j$ the intermediate values (integers) of the bit strings, calculated by (11) or (12) at the end of k-th iteration, $k = 1,\ldots, num_j$. In that case we get

$$BS^{n_j}(i) = IBS_1(i) + 2 * IBS_2(i) + \ldots + num_j * IBS_{num_j}(i). \tag{13}$$

We will use the integers $BS^{n_j}(i)$, for $i = 1,\ldots, r^{n_j}$ when we calculate the knapsack.

Factoring

For every sequence of polynomials $CU\,R^{n_j}$ (that have already been distributed between partitions and the corresponding tables), prepare new registers $R_{ik}^{n_j}$ $i = 1, \ldots, r$, $k = 1, \ldots, irr(n_j)$, $j = 1, \ldots, c$, where, again, $irr(n_j)$ is the number of irreducible polynomials of degree less than n_j. So for the sequence of polynomials $CU\,R^{n_j}$ we define r^{n_j} partitions and any partition is associated with $irr(n_j)$ registers. Set the initial value of all the registers equal to zero. The values of the registers will be determined by means of factoring the polynomials of sequences (6) in the following way.

On the i-th step, having calculated the value $CU\,R^{n_j}$, (which has been assigned to, say, partition x), we increase the values of the registers $R_{ik}^{n_j}$ by

$$FC_{n_j}(CU\,R_i^{n_j}, k)$$

for $k = 1, \ldots, irr(n_j)$. The values of the registers $R_{ik}^{n_j}$ of all the sequences (6) are the desired features of initial sequence of bits (file) M. We will call values $R_{ik}^{n_j}$ the spectrums.

We describe one more algorithm of the factoring. We will call it the conditional factoring.

We prepare r_j partitions and assume that each of the partitions "is responsible" for certain section of a file. We can do it by means of certain formula, say, elements M_1, M_{200},... and so on form "the area of responsibility" of the first partition (assume that considered r_j is equal to 200), then M_2, M_{201},... belong to the "area of responsibility" of the second partition and so forth. Such a formula can be not that straightforward, that is, any easily calculated rule of the conditional distribution can be considered. When we calculate $CU R_i^n$ we also calculate the index of partition to which it has to be assigned. Assume that partition q is chosen. In this case, if M_l from the considered $CU R^n$ belongs to the "area of responsibility " of the current partition, that is, partition q, we multiply the corresponding factors, say, by 2 (in general, by any fixed number) before adding it to the corresponding spectrum, otherwise we do not change it.

Enumerating the Spectrums

I. Cantor Enumeration

Cantor enumeration can be presented as a recursive function [4]

$$c_d : N^d \to N,$$

which enumerates any finite ordered sequences of integers. If $d = 2$ then for any pair of integers $< x,y >$ we have

$$c_2(x,y) = \frac{(x+y)^2 + 3x + y}{2}.$$

Let $n = c(x,y)$ for an arbitrary pair $< x,y >$. Then there exist two functions $l(n)$ and $r(n)$ such that $x = l(n)$ and $y = r(n)$, where

$$l(n) = n - 0.5\left\lceil \frac{\left[\sqrt{8n+1}\right]+1}{2} \right\rceil \left\lceil \frac{\left[\sqrt{8n+1}\right]-1}{2} \right\rceil$$

and

$$r(n)=\left\lceil \frac{\left[\sqrt{8n+1}\right]+1}{2} \right\rceil - l(n) - 1.$$

The functions $c(x,y)$, $l(n)$ and $r(n)$ are related by the following correspondences

$$c(l(n), r(n)) = n,\ l(c(x,y)) = x,\ r(c(x,y)) = y.$$

One can easily enumerate triples, quadruples of numbers and so on. Indeed,

$$c_3(x_1,x_2,x_3) = c_2(c_2(x_1,x_2), x_3),$$

More generally, define

$$c_{n+1}(x_1,x_2, \ldots, x_{n+1}) = c_n(c_2(x_1,x_2), x_3, \ldots, x_{n+1}).$$

Number

$$c_n(x_1,x_2, \ldots, x_n)$$

is called the Cantor number of the n-tuple

$$< x_1,x_2, \ldots, x_n >.$$

Note that if

$$c_n(x_1,x_2, \ldots, x_n) = x,$$

then

$$x_n = r(x),\ x_{n-1} = rl(x), \ldots, x_2 = rl\ldots l(x),\ x_1 = ll\ldots l(x).$$

II. Calculations of Cantor numbers of the Spectrums

Consider any sequence $CU\ R^n$, for which we have already calculated spectrums $R_{ij}\ i = 1, \ldots, r_l$,

$j = 1, \ldots, irr(n_l)$, $l = 1,\ldots,$ c (we omitted upper indices of the registers R_{ij} here). Define now the enumerating procedure of spectrums (registers) R_{ij}, $1 \le i \le r$, $1 \le j \le irr(n_l)$. If we compute numbers E_i for $1 \le i \le r$ by means of direct enumerating, that is, as follows

$$E_i = c_{irr(n_l)}(R_{i1}, R_{i2}, \ldots, R_{irr(n_l)}),$$

where $c_{irr(n_l)}$ is the above-described Cantor enumeration function, we can obtain really huge numbers. For instance, if we enumerate the following spectrum for $n = 6$ directly, that is, by

$$E_1 = c_{16}(5, 3, 161, 139, 37, 21, 13, 9, 6, 3, 2, 2, 3, 1, 4, 2)$$

we obtain the number in 32892 digits, which is unacceptable. Therefore we describe the following enumerating procedure.

Consider any spectrum $Spec_j = \{s_1, s_2, \ldots, s_j\}$, where $j = 7, 10, 16, 25, 43, 71, 129,$ which, in turn, corresponds to $n_j = 4,5,\ldots,10$.

CASE 1. $j = 7$. We set

$$E_7 = c(c(c(c(s_1, s_2), s_3), c(s_4, s_5)), c(s_6, s_7)).$$

CASE 2. $j = 10$. We set

$$E_{10} = c(c(c(c(s_1, s_2), s_3), c(s_4, s_5)), c(c(s_6, s_7), c(c(s_8, s_9), s_{10}))).$$

CASE 3. $j = 16$. We set

$$E'_{16} = c(c(c(s_1, s_2), c(s_3, s_4)), c(c(s_5, s_6), c(s_7, s_8))),$$

$$E''_{16} = c(c(c(s_9, s_{10}), c(s_{11}, s_{12})), c(c(s_{13}, s_{14}), c(s_{15}, s_{16}))),$$

and finally

$$E_{16} = c(E'_{16}, E''_{16}).$$

CASE 4. $j = 25, 43$. We divide 25 and 43 elements of the spectrums into 7 and 10 groups, correspondingly, in the following way. We take the first nine elements of $Spec_{25}$ and compute integers

$$c(s_1, c(s_2, s_3)), \quad c(s_4, c(s_5, s_6)), \quad c(s_7, c(s_8, s_9)).$$

Then we divide 16 remaining elements of $Spec_{25}$ into four groups by four elements in each group and calculate four integers by

$$c(c(s_{10}, s_{11}), c(s_{12}, s_{13})), \quad c(c(s_{14}, s_{15}), c(s_{16}, s_{17})),$$

$$c(c(s_{18}, s_{19}), c(s_{20}, s_{21})), \quad c(c(s_{22}, s_{23}), c(s_{24}, s_{25}).$$

We eventually get seven integers, which will be enumerated as it was described in CASE 1. Considering $Spec_{43}$ we act in the same manner, just divide the elements of the spectrum into 7 groups by 4 elements each and 15 last elements by 5 elements each. Below we show how to enumerate the group of five elements a, b, c, d, e. We set

$$c(c(a, c(b, c)), c(d, e)).$$

We obtain ten integers and apply the procedure, described in CASE 2 above.

CASE 5. $j = 71, 129$. Consider $Spec_{71}$. We divide the elements of the spectrum into 16 groups. First nine groups contain 4 elements each and 7 remaining groups contain 5 elements each. We have already shown how to enumerate a group of four or five elements. We get 16 integers and apply the method of CASE 3.

All the elements of $Spec_{129}$ are divided into 25 groups. The first 25 groups contain 5 elements each and the last group contains 9 elements, which we enumerate in accordance with

$$c(c(c(a, c(b, d)), \quad c(e, c(f, h))), \quad c(g, c(i, j))).$$

We get 24 integers, which we enumerate in accordance with the method of CASE 4.

As we calculate the enumeration procedure for every partition $i = 1, \ldots, r_l$ and for every sequence $CU\ R^n$, $l = 1, \ldots, c$ we obtain $r_l\ c$ integers E_{ij}.

Organizing a Knapsack

As a result of enumeration we obtain integers E_{ij}, $i = 1, \ldots, r, j = 1, \ldots, c$. These integers are associated with the corresponding tables that we obtain after the factorization. The bit strings also are calculated for the tables and therefore we want to show how E_{ij} and the corresponding bit strings should be combined. As the partitioning procedure and the procedure of collecting factors can be realized in different ways, we will consider various possibilities of forming the knapsack. In fact, we have to consider three cases that were described in the subsection "Forming the bit strings".

CASE I, II. As the bit strings already processed, that is, as integers $BS^{n_j}(i), i = 1,\ldots, r_j$ are calculated for $j = 1, \ldots, c$, we compute c knapsacks

$$V_j = E_{1j} + BS^{n_j}(1) + 2(E_{2j} + BS^{n_j}(2)) + \ldots + r_j(E_{r_j} + BS^{n_j}(r_j)).$$

CASE III. We set to 0 all the bit strings and all the tables after updating the bit strings. Again, we fix some j and assume that the number of iterations (the number of setting the bit strings to 0) is num_j. Denote by $IBS_k(u)$, the intermediate values (integers) of the bit strings, calculated by (12) or (13) at the end of k-th iteration, $k = 1,\ldots, num_j$, and by $IE_k(u)$, where $u = 1,\ldots,r_j$, $k = 1,\ldots, num_j$ the enumerations of the corresponding intermediate values of register R. Then we calculate (remind that j is fixed)

$$\begin{aligned} V_j = {} & IE_1(1) + IBS_1(1) + 2(IE_1(2) + IBS_1(2)) + \ldots + r_j(IE_1(r_j) + IBS_1(r_j)) + \\ & 2(IE_2(1) + IBS_2(1)) + 3(IE_2(2) + IBS_2(2)) + \ldots + (r_j + 1)(IE_2(r_j) + IBS_2(r_j)) + \\ & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ & + num_j(IE_{num_j}(1) + IBS_{num_j}(1)) + (1 + num_j)(IE_{num_j}(2) + IBS_{num_j}(2)) + \ldots + \\ & + (r_j + num_j - 1)(IE_{num_j}(r_j) + IBS_{num_j}(r_j)) \end{aligned}$$

Keeping in mind the construction of all standard hash functions we can conclude that the proposed hash function possesses the following specific feature.

In the Damgard-Merkle construction, each block of fixed size is hashed and the calculated intermediate hash value is combined with the next block. In the construction described above, we have a different situation. Each block and associated bit string yields a factor table. Moreover, the bit string is *xor*-ed to the bit string of the next block (see CASE I in "Forming the bit strings"), and the factor table construction is iterated. Finally, we obtain a bit string and a cumulative factor table.

Exponentiation

Let, further, G be a cyclic group for which the discrete logarithm problem is hard, and assume that g is a generator of G. Let τ be the size (in bits) of presentation of any element from G. We define a general scheme of computing points H_i of group G, based on V_i, $i = 1,...,c$. We set

$$H_i = g^{V_i}$$

We show two examples of the calculation of H_i, based on the above-described general scheme.

Example 1. Let τ be a length of a hash-value. Choose a primitive element

$$\gamma \in GF(2^\tau)$$

Then values H_i of G are defined by

(14) $$H_i = \gamma^{V_i}$$

Example 2. Choose a $GF(2^\tau)$ and let E be an elliptic curve over $GF(2^\tau)$ with good cryptographic properties [3]. Let γ be a generator of $E(GF(2^\tau))$. Then the points H_i of $E(GF(2^\tau))$ are calculated by

(15) $$H_i = V_i \gamma$$

The Final Hash Value Calculation

Thus, we have c elements $H_i = g^{V_i}$ of the corresponding group G. We generate the final hash value H of string of bits M in accordance with

$$H = H_1 \times H_2 \times \ldots \times H_c,$$

where $\times$ is the group operation. For instance, in the case of group $E(GF(2^r))$ for some elliptic curve E over $GF(2^r)$ we get

$$H = H_1 + H_2 + \ldots + H_c.$$

In that case, the x-coordinate of H is the final hash value of initial message (sequence of bits) M.

Message Authentication Coding

A secure hash function, or MAC is a map

$$H : \{0,1\}^* \times \mathbf{K} \rightarrow \{0,1\}^r,$$

where $\mathbf{K}$ is a key space. A key can be involved in the process of constructing a fingerprint in some of the considered stages of generation of a hash value. The analysis of all the cases will be done below.

We want to present some of the possibilities of usage of a key. So, let K be a key. Using K as input for an appropriate pseudo-random generator let us generate a sequence S of size k, where k is the length (in bits) of initial sequence M. Denote by S_i the i-th bit of sequence S.

Key scheme on the stage of splitting

Assume that we have a key K of a size l (in bits) and denote by S_i , $i = 1,\ldots,l$ the i-th bit of the key. Below we describe various possibilities of implementing a key into the proposed construction of a hash function.

A. At the stage of splitting for some fixed j we form the sequence of polynomials $S(M, n_j)$ in the following way. For any i we consider the value of S_i. If S_i is equal to 1, we include element M_i into $S(M, n_j)$, otherwise, element M_i is not included in $S(M, n_j)$.

B. We apply the same procedure of including elements M_i into $S(M, n_j)$, depending on the value of S_i for all $j = 1, \ldots, c$.

C. For some fixed j for any i if S_i is equal to 1, we include element M_i into $S(M, n_j)$, otherwise, instead of M_i, we include its negation, that is, element $\overline{M_i}$.

D. The procedure described in C is applied to all $j = 1, \ldots, c$.

E. The j, for which we apply schemes A and D, is defined by a key K, that is, it is a part of the key K.

F. In all considered schemes A – E, we use bits of a key K as elements of S_i. In that case, we just need to use $k/l_K + 1$ times key K, where l_K is the length of K in bits.

We do not change any of the remaining stages of the generation of a fingerprint of M.

Key scheme on the stage of masking

A. For some fixed j and for any $i = 1, \ldots, k$ if S_i, for instance, is 1, we calculate

$$CU\,R_i^{n_j} = M_i \oplus \delta^{(\mathrm{int}(M_{i-1})+\mathrm{ind}(M_{d_1}))\bmod 2^{n_j}} \oplus \beta^{(\mathrm{int}(CUR_{i-1}^{n_j})+\mathrm{int}(CU\,R_{d_2}))\bmod 2^{n_j}}.$$

If S_i is 0, then

$$CU\,R_i^{n_j} = \overline{M_i \oplus \delta^{(\mathrm{int}(M_{i-1})+\mathrm{ind}(M_{d_1}))\bmod 2^{n_j}} \oplus \beta^{(\mathrm{int}(CUR_{i-1}^{n_j})+\mathrm{int}(CU\,R_{d_2}))\bmod 2^{n_j}}}.$$

B. We apply the procedure, described in A to all $j = 1, \ldots, c$.

C. The j, for which we apply scheme A, is defined by a key K, that is, it is a part of a key K.

D. In the scheme A – E, we can use bits of a key K as elements of S_i.

We do not change any of the remaining stages of the generation of a fingerprint of M.

Key scheme on the stage of partitioning

We describe the scheme of involving a key on the stage of partitioning elements $CU\,R_i^{n_j}$.

A. Based on a key K, generate randomly k integers x_i and form the following sequence of integers y_i

$$y_i = \sum_{t=1}^{k} x_t \bmod r .$$

For some fixed j, $CU\ R_i^{nj}$ is assigned to partition y_i+1.

B. Fix some j and consider S_1. If S_1 is 1, we assign $CU\ R_1^{nj}$ to partition 1, otherwise, to partition 2. If then, S_2 is 1, we assign $CU\ R_2^{nj}$ to the current partition, that is, to partition 2, otherwise, polynomial $CU\ R_2^{nj}$ will be sent to partition 3 and so on. So, on the i-th steps of the consideration, let the current partition be partition ω. If S_{i+1} is 1, we assign polynomial $CU\ R_{i+1}^{nj}$ to partition ω, otherwise, we send it to partition $\omega+1$. If $\omega+1$ is greater than r_j, where r_j is the number of partitions, we start from the very beginning, that is, partition 1 will be chosen as partition number $\omega+1$.

C. The procedures described in A and B are applied to all $j = 1, \ldots, c$.

D. In schemes A and B, the fixed j is defined by a part of a key K.

We do not change any of the remaining stages of the generation of a fingerprint of M.

Key scheme on the stage of forming the bit strings

A. A key K defines the sequence of the elements for the corresponding tables for which we will prepare the bit strings for a certain j.

B. The same as A but for all j.

C. A key K defines the tables, for which we will generate some fixed number of the bit strings for the first coming elements for a certain j.

D. The same as C but for all j.

We do not change any of the remaining stages of the generation of a fingerprint of M.

Analysis of Hash Function

Running Time

First of all, let us define the running time of the algorithm. The main problem in realizing the algorithm is the factorization of $CU\ R_i^{n_j}$, $i = 1, \ldots, k, j = 1, \ldots, c$. But for small n, say $n \leq 10$, we do not have many irreducible polynomials, for instance, for $n = 10$, the number of the polynomials is 127. Besides, first, we can just count all the elements of any partition. It means that it is necessary to use more registers, that is, exactly 2^{n_j} registers, and then we can use the table of factors for all 2^{n_j} elements. Since n_j is not large, the table is not large either.

In that case, to form the tables we need to calculate $2*(k-1)*n_j$ *xor* operations, $k*n_j$ bit sums and $(k-1)*n_j$ sums by modulo 2^{n_j}, where $j = 1, \ldots, c$. To form the bit strings we need to perform additionally (depending on the scheme of updating) at least k *xor* bit operations. We ignore the time needed for the calculation of V_i and the final hash value H.

It can immediately be seen that the greater c the more time we need to generate a hash value.

Analysing the bit strings, we want to emphasize the fact that having the possibility to vary different possibilities of the usage of the bit strings, we, in fact, obtain very fascinating construction of the hash function with varying level of security.

To understand it, assume that we do not take into account the values of the bit strings and let's try to analyze the possibility to generate two collided files, that is, two files with the same partition tables, as the final hash value will directly depend on the values of the partition tables.

Consider a finite state machine (FSM) with the file to be hashed as input and a list of indices into the partition tables as output. These indices indicate which partitions should be incremented as a result of processing each bit. Note that in that case varying the order of these indices does not change the resulting hash.

We only consider the possible states the machine may be in at the end of processing each bit. Only CUR^{n_j} and the current column in the partition table for every n_j form the state space, while,

for instance, the partition tables are not a part of the state of this machine. The machine may need additional states to handle the intermediate stages of processing a single bit, but we ignore these. Consider, for example, the case $c = 2$, say, $n_1 = 4, n_2 = 6$. Assume also that the number of the tables for each of the cases n_1 and n_2 $r \le 256$. We need to evaluate the number of states of the FSM. So, we have that to represent CUR^4 and the table, to which it will be associated, we need 8 bits. Accordingly, to represent CUR^6 and the number of the corresponding table we need 6 and 8 bits, correspondingly.

It means that this FSM has at most 2^{26} distinct states, which is not enough, as two collided files can be generated in feasible time. We present here very rough calculation of the states of FSM. (In fact, FSM has (much) more states, as any CUR_i depends on the corresponding chain of CUR_j as well as any M_i depends on a collection of M_j.) However, again, we simplify the situation in order to understand the influence of the bit strings.

The situation is absolutely different if we take into account the bit strings associated to each table. Even considering just one bit string per table and, keeping in mind that the length of a bit string is 200, we can conclude that the number of states of FSM is increased by 2^{200} states. Moreover, increasing the number of the bit strings, associated with the tables, we increase the number of states of FSM. In other word, we can easily change the very important parameter - the number of states of FSM, associated with our hash function, which, in turn, means that the level of security of the hash function can be easily controlled and changed in accordance with necessity.

Let's now estimate the number of bits of a hashed file that will be processed in a framework of one bit string of the length sz^{n_j} before its updating. If r^{n_j} is a number of tables for considering sequence CUR^{n_j}, then, in average, using the distribution of the elements by means of indices (7) – (10), we will process

$$\left\lceil \frac{r^{n_j}}{2^{n_j}/2} \right\rceil$$

"portions" of a file, each of which is n_j bits of length, that is, we get in average

$$\left\lceil \frac{r^{n_j}}{2^{n_j}/2} \right\rceil sz^{n_j} n_j$$

bits. For example, if for $n_j = 4$, the number of tables is 200 and the length of the bit string is 200, we get

$$\left\lceil \frac{r^{n_j}}{2^{n_j}/2} \right\rceil sz^{n_j} n_j = (200/8) * 200 * 4 = 20000 .$$

That is, we are able to process an initial file, block by block having the block size about 20000 bits. Recall that the block size of all the existing hash algorithms, including all algorithms of SHA family, is 512 bits. Moreover, every time (every updating) the length of the block size is different, and, the most important, it is unique, and regulated by specifics of a file itself.

Simple calculations show that using the conditional distribution of the polynomials, say, for the case $q_j = g_j = 1$, we can process the blocks of the size up to 120 000 – 140 000 bits.

Note also that we process all the blocks bit by bit, which is also different from all the existing hash function, where the whole block has to be processed during an iteration. Finally, based on expression 20, we can see that the length of the block depends on the number of tables and the length of the bit strings. Varying these parameters allows us changing the length of the block size, that is, changing the parameters of security of the hash function, as the bigger block the more exactly, or carefully we can collect the specific features of an initial file.

Possible Attacks

To create a collision one has to know all the spectrums for all the partitions and for all n_j, $j = 1, \ldots, c$. Besides, an attacker has to form the corresponding bit strings.

Analyse, first, the possibility to create a file with such a spectrum that coincidences with the spectrum of the hashed file. Repeating all the stages of all the calculations can do it. Thus, one can obtain the spectrums

$$R^{n_1}_{irr(n_1)}, R^{n_2}_{irr(n_2)}, \ldots, R^{n_c}_{irr(n_c)} .$$

Taking into consideration expression for calculating the values of register CUR, that is, the expressions

$$CUR_i^{n_j} = M(i, i+n-1) \oplus \delta_{n_j}^{(\mathrm{int}(M_{i-1})+\mathrm{int}(M_{d_1}))\bmod 2^{n_j}} \oplus \beta_{n_j}^{(\mathrm{int}(CUR_{i-1})+\mathrm{int}(CUR_{d_2}))\bmod 2^{n_j}},$$

where δ_{n_j} and β_{n_j} are the corresponding generators of finite field $GF(2^{n_j})$, $j = 1, \ldots, c$, we can write

$$CUR_i^{n_j} = f_1^{Y_{i1}^{n_j}} f_2^{Y_{i2}^{n_j}} \ldots f_{irr(n_j)}^{Y_{irr(n_j)}^{n_j}}.$$

Here $f_1, f_2, \ldots, f_{irr(n_j)}$, $j = 1, \ldots, c$ are the ordered irreducible polynomials of degree less than n_j, with the multiplicities $Y_{i1}^{n_j}$, $Y_{i2}^{n_j}$, ..., $Y_{irr(n_j)}^{n_j}$, that we obtain during the factorization of polynomial $CUR_i^{n_j}$. We get, further

(16) $$f_1^{Y_{i1}^{n_j}} f_2^{Y_{i2}^{n_j}} \ldots f_{irr(n_j)}^{Y_{irr(n_j)}^{n_j}} = M(i, i+n-1) \oplus \delta_{n_j}^{(\mathrm{int}(M_{i-1})+\mathrm{int}(M_{d_1}))\bmod 2^{n_j}} \oplus$$

$$\oplus \beta_{n_j}^{(\mathrm{int}(CUR_{i-1})+\mathrm{int}(CUR_{d_2}))\bmod 2^{n_j}},$$

where for $i = 1, \ldots, k$ and $j = 1, \ldots, c$. Besides, we have

(17) $$\sum_{i=1}^{k_1} Y_{i1}^{n_j} = R_{11}^{n_j}, \sum_{i=k_1+1}^{k_2} Y_{i1}^{n_j} = R_{21}^{n_j}, \ldots, \sum_{i=k_{c-1}+1}^{k_c} Y_{i1}^{n_j} = R_{r1}^{n_j},$$

$$\sum_{i=1}^{k_1} Y_{i2}^{n_j} = R_{12}^{n_j}, \sum_{i=k_1+1}^{k_2} Y_{i2}^{n_j} = R_{22}^{n_j}, \ldots, \sum_{i=k_{c-1}+1}^{k_c} Y_{i2}^{n_j} = R_{r2}^{n_j},$$

.....,

$$\sum_{i=1}^{k_1} Y_{irr(n_j)}^{n_j} = R_{1irr(n_j)}^{n_j}, \sum_{i=k_1+1}^{k_2} Y_{irr(n_j)}^{n_j} = R_{2irr(n_j)}^{n_j}, \ldots, \sum_{i=k_{c-1}+1}^{k_c} Y_{irr(n_j)}^{n_j} = R_{rirr(n_j)}^{n_j}.$$

Note that, k_1, $k_2 - k_1, \ldots, k_r - k_{r-1}$ are the numbers of polynomials in the corresponding partitions.

So, to create a collision, one has to solve c systems of non-linear iterated equations (16) with additional constraints (17) with unknowns M_i. Besides, all unknowns M_i have to be related in accordance with (3).

But the most important circumstance of the construction is that even if one manages to solve the system of equations (16) for a particular fixed j, that is, even if one manages to generate the sequence M_i' that eventually gives the same point H_j of group G, it is impossible to control the rest of the c-1 systems, as all of the elements from different $S(M, n_j)$, $j = 1, \ldots, c$ are related by (4). Besides, the number of partitions for different $j = 1, \ldots, c$ is different.

It means that even if one can find such a sequence of M_i' (which is different from the M_i, $i = 1, \ldots, k$) for a particular j, this will only result in the possibility of obtaining just one point H_j, while the rest of the points – elements of group G – will be calculated automatically and one does not have any possibility to change them, because of the established relations (4) between the elements from different $S(M, n_j)$, $j = 1, \ldots, c$. It means that one does not have any systematic, algorithmic possibility to control all the group elements H_j, which eventually means that one does not have any possibility to control changing the final hash value H.

We stress that the number of iterations of the proposed hash function is at least, two. So the considered above system of non-linear equations, (which was written for the first iteration) must be solved again, but just one needs to consider CUR_2 instead of M_1.

However, even overcoming all the problems, related to finding such a sequence of bits M' that gives the same spectrum, does not allow creating a collision. The key point (on a top of solving the collection of systems (16)) is the difficulty of the creation of such a sequence of bits M' that at the same time (on a top of satisfying the collection of systems (16)) gives the same collection of the bit strings.

Analysis of MAC

In the case of MAC, one cannot restore all the spectrums as they are formed with the usage of a key K. Therefore, there is only one way to obtain all the spectrums in order to consider all the

systems of non-linear equations 16. Firstly, the discrete logarithm problem has to be solved in order to find H. The, for instance, for the case of elliptic curve, taking into consideration that

$$H = H_1 + H_2 + \ldots + H_c,$$

additionally an attacker needs to "guess" $H_1, H_2, \ldots, H_c$. If one even manages to find all H_j, $j = 1, \ldots, c$, he needs to open c knapsack. Keeping in mind that since we use not less than 200 partitions, the difficulty of the opening one knapsack is

$O(2^r)$.

The Key Attack

The key attack is also quite difficult to apply to the considered MAC function. Indeed, an attacker has to perform around $2^{lK/2}$ attempts in order to carry the attack out, where lK is the size of a key K. If, for instance, $lK = 160$, it means that an attacker needs to provide around 2^{80} exponentiations.

In the case of usage, for example, NIST curves, times for the calculation of kP for the corresponding curves are presented below. The calculations were made on a Pentium II, 400 MHz.

The data on the calculation of kP can be found in [9]

Nn.	NIST curve	Time (μs)
1	K-163	1176
2	K-233	2243
3	K-283	3330
3	K-409	7611
3	K-571	18118

From the table, it becomes clear that the attack can be hardly realized.

Enhancing the Security

The hash function consists of at least two iterations. The security of the hash (and MAC) function can be sharply increased by means of increasing the bit strings, as such an increasing implies the increasing the number of states of (FSM).

Iterated masking; see the flow charts "Hashing with iterations" and "Message authentication coding with iterations" also increases the resistance to creating a collision, as in that case, the collection of the systems system of the corresponding non-linear iterated equations will be much more difficult.

Implementation

As one example, the method of the present invention can be readily implemented in a Dynamically Linked Library or DLL which is linked to a computer program that utilizes an algorithm that embodies the hash function or MAC function described above, for example, an encryption, decryption or authentication utility that is operable to apply said algorithm.

The computer program of the present invention is therefore best understood as a computer program that includes computer instructions operable to implement an operation consisting of the calculation of the hash value or MAC value as described above.

Another aspect of the present invention is a computer system that is linked to a computer program that is operable to implement on the computer system the transformation of a MAC-value, in accordance with the present invention.

This invention will be of use in any environment where hash functions and MAC functions are used for data integrity or authentication (digital signatures being an example).

An example is secure email. Several widely used systems for secure email (such as PGP and S/MIME) use SHA-1 as the hash algorithm.

Another application is to secure Virtual Private Networks (VPNs) by operation of the present invention. Such networks allow clients to use standard internet connections to access closed private networks. The authentication of such networks is based on a protocol such as IPSec or

SSL. In both cases, the authentication uses a MAC algorithm such as SHA-1. Thus, vulnerability in SHA-1 will result in vulnerability of the VPN. By introducing a MAC algorithm based on the above, these vulnerabilities are mitigated.

One more application is to secure Digital Signatures by operation of the present invention. One of the algorithmic stages of any Digital Signatures is hashing. In fact, a Digital Signatures algorithm is usually applied to a hash (or MAC) value of a file. And almost all Digital Signatures algorithms now using SHA-1 as such a hash (or MAC) algorithm. And, again, vulnerability in SHA-1 will result in vulnerability of all the Digital Signatures algorithms.

As another example, the method of the present invention can be readily implemented in a specially constructed hardware device. As discussed above, the bit stream specific is one of the most important features of the present invention. It means that the hardware implementation of the present invention is very convenient and easily realisable. Moreover, such a hardware implementation of the present invention enables a dramatic increase in the speed of hashing, as all the hardware implementations of stream algorithms are usually much faster than the corresponding software implementations. We stress here that all the algorithms of the SHA family (as well as the MD and RIPEMD families) are block-based algorithms and are not convenient for hardware implementations.

An integrated circuit can be created to perform the calculations necessary to create a hash value or MAC value. Other computer hardware can perform the same function. Alternatively, computer software can be created to program existing computer hardware to create hash or MAC values.

REFERENCES

[1] D. Coppersmith, A. Odlyzko and R. Schroeppel, Discrete logarithms in GF(p), Algorithmica, 1(1986), 1-15.

[2] A. Nijenhius and H. Wilf, Combinatorial Algorithms for Computers and Calculators, 2nd ed. New York: Academic Press, 1978.

[3] N. Koblitz, Elliptic Curve cryptosystems, Math. Comp., 48(1987), 203-209.

[4] A.LMal'cev, Algorithms and recursive functions, Wolters-Noordhoff Pub.Co. 1970.

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

[6] V. Kumar Murty and N. Volkovs, Hash and Mac functions with transformation, preprint, 2005.

[7] T. Krovetz and P. Rogaway. Fast universal hashing with small keys and no preprocessing: the PolyR construction, in: Information Security and Cryptology ICICS 2000, pp. 73-89, ed. D. H. Won, Lecture Notes in Computer Science, 2015, Springer-Verlag, 2000.

[8] W. Stallings, Cryptography and Network Security: Principles and Practice, 2nd edition, Prentice Hall, 1999.

[9] M. Brown, D. Henkerson, J. Lopez, and A. Menezes, Software Implementation of the NIST Elliptic Curves Over Prime Fields,www.eng.auburn.edu/users/hamilton/security/pubs/Software Implementation of the NIST Elliptic.pdf.

[10] R.C. Merkle, Secrecy, Authentication, and Public Key Systems, UMI Research Press, Ann Arbor, Michigan, 1979.

[11] L. Carter and M. Wegman Universal classes of hash functions. J. of Computer and System Sciences 18, 1979, 143-154.

CLAIMS:

1. Computer software for performing a secure hashing method comprising the steps of:

(1) representing an initial sequence of bits as a specially constructed set of polynomials;

(2) transformation of this set by masking;

(3) partitioning the transformed set of polynomials into a plurality of classes;

(4) forming the bit string during the partitioning;

(5) for each of the plurality of classes, factoring each of the polynomials and so as to define a set of irreducible polynomials and collecting these factors in registers defined for each of the plurality of classes;

(6) wrapping the values of the registers from the plurality of classes by means of an enumeration;

(7) organizing the enumerations and the bit strings into a knapsack; and

(8) performing an exponentiation in a group to obtain the hash value or the MAC value.

2. An integrated circuit adapted to create a hash value by performing the steps of:

(1) representing an initial sequence of bits as a specially constructed set of polynomials;

(2) transformation of this set by masking;

(3) partitioning the transformed set of polynomials into a plurality of classes;

(4) forming the bit string during the partitioning;

(5) for each of the plurality of classes, factoring each of the polynomials so as to define a set of irreducible polynomials and collecting the factors in registers defined for each of the plurality of classes;

(6) wrapping the values of the registers from the plurality of classes by means of an enumeration;

(7) organizing the enumerations and the bit strings into a knapsack;

(8) performing an exponentiation in a group to obtain the hash value or the MAC value.

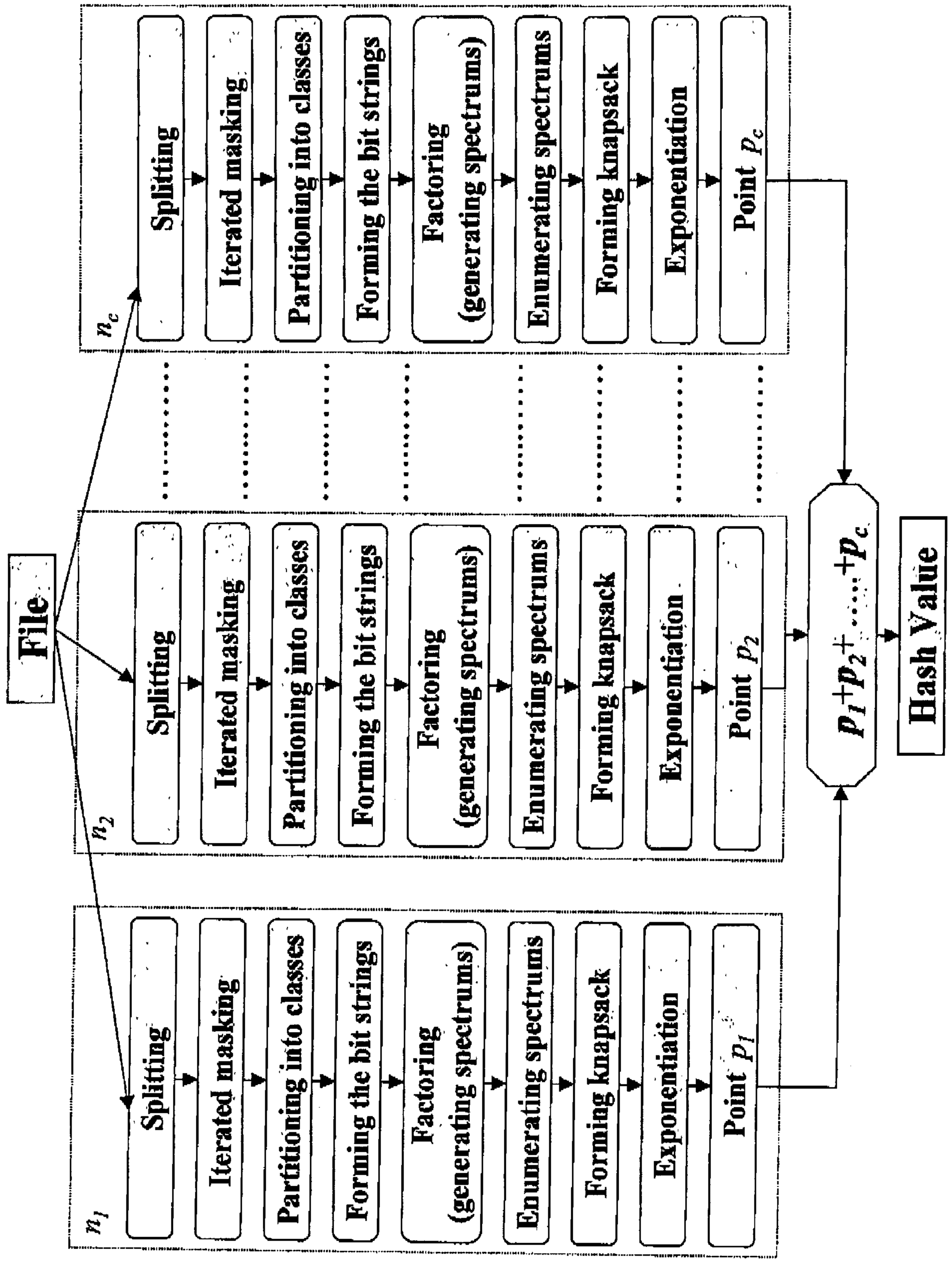

Scheme of Hashing with Separate Generation of Spectrums
File
n_1
n_2
n_c
Splitting
Iterated masking
Partitioning into classes
Forming the bit strings
Factoring (generating spectrums)
Enumerating spectrums
Forming knapsack
Exponentiation
Point p_1
Point p_2
Point p_c
$p_1+p_2+\ldots+p_c$
Hash Value

FIGURE 1

Hashing with Iterations

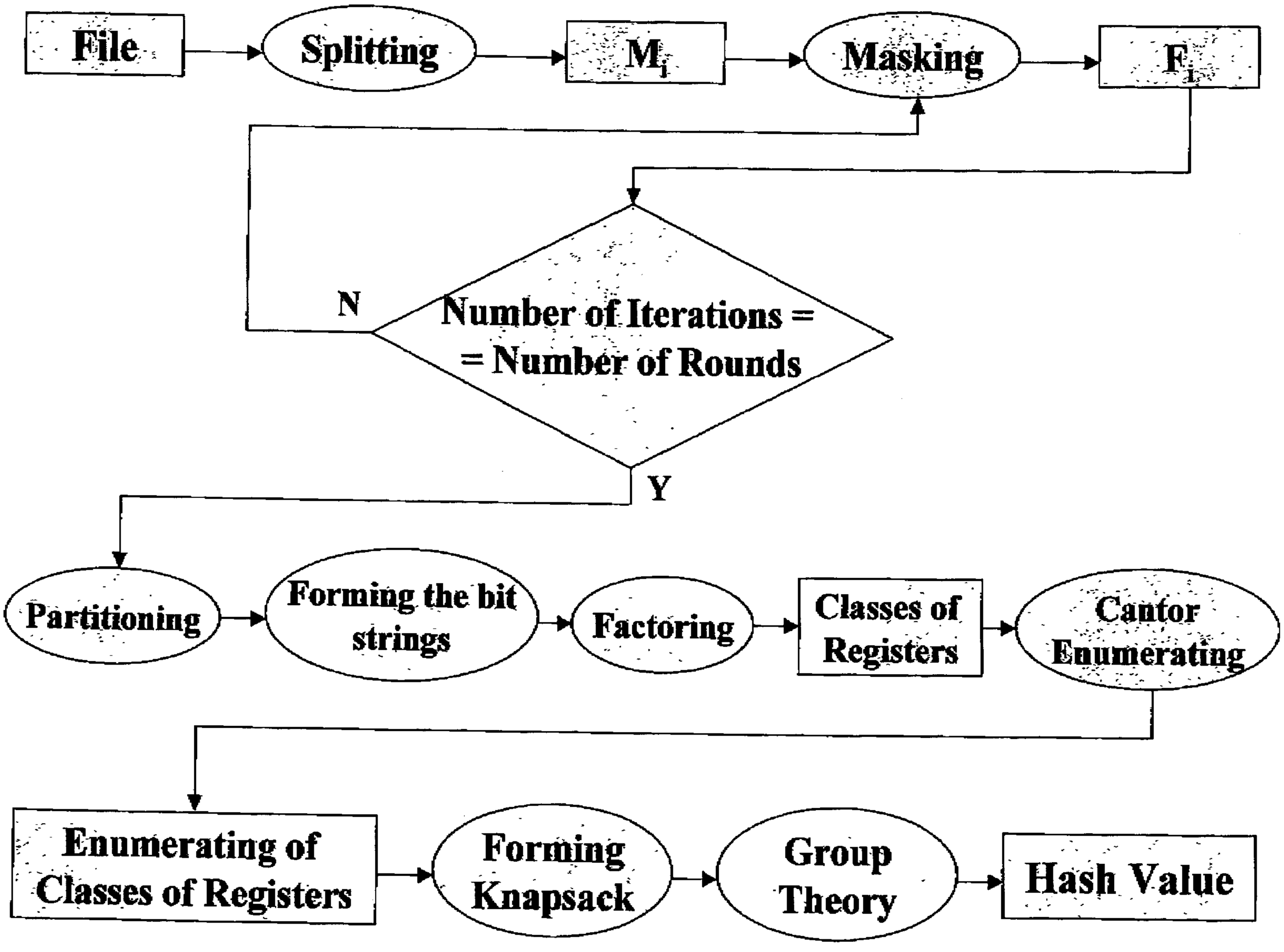


FIGURE 2

Message Authentication Coding with Iterations

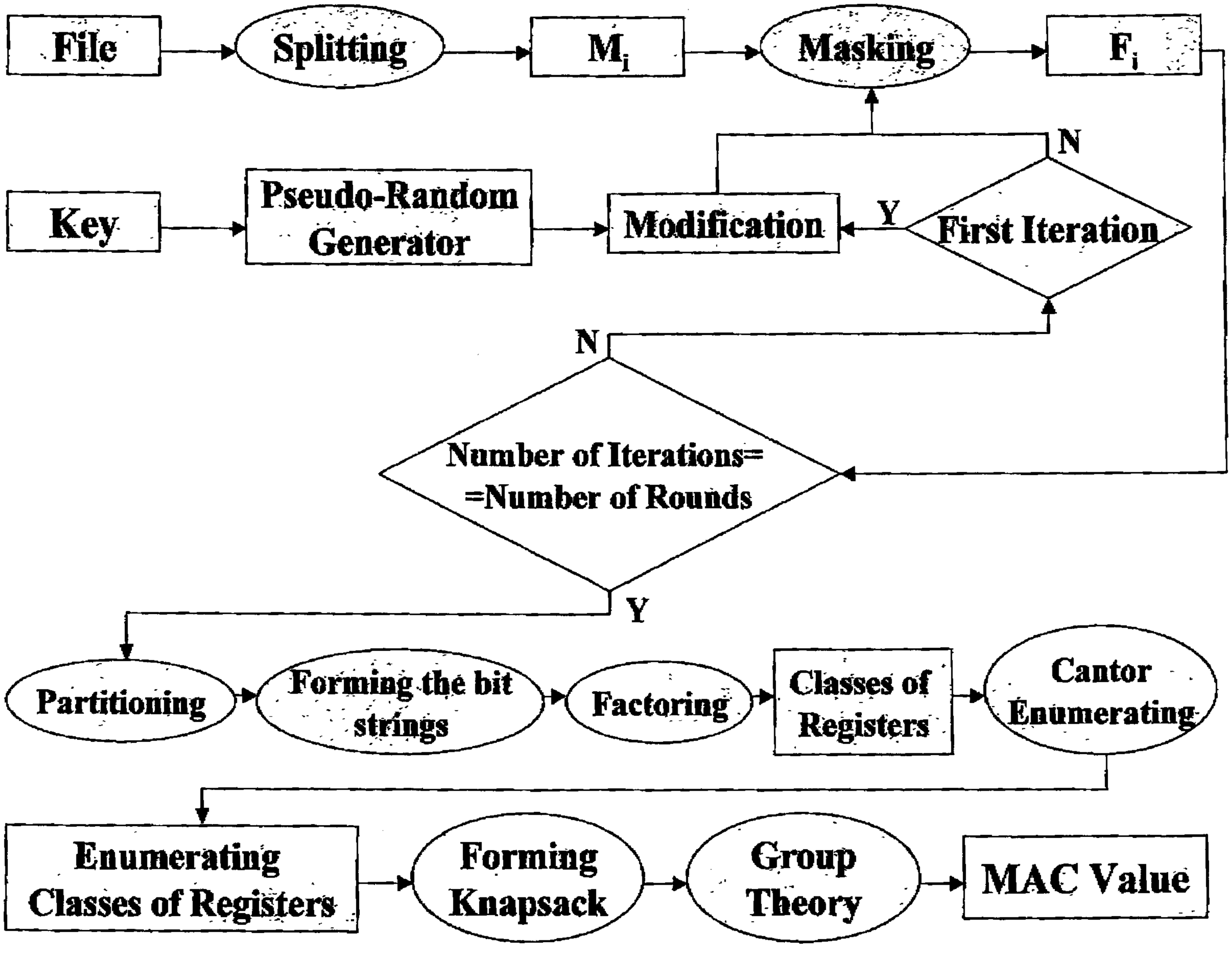


FIGURE 3

Scheme of Hashing with Separate Generation of Spectrums