ERINDALE: A Polynomial Based Hashing Algorithm

V. Kumar Murty[1] and Nikolajs Volkovs[2]

[1] Department of Mathematics,
University of Toronto,
Toronto, Ontario, CANADA M5S 2E4
murty@math.toronto.edu
[2] GANITA Lab, Department of Mathematical and Computational Sciences,
University of Toronto at Mississauga
3359 Mississauga Road North
Mississauga, Ontario, CANADA L5L 1C6
n.volkovs@utoronto.ca

Abstract. The aim of this article is to describe a new hash algorithm using polynomials over finite fields. In software, it runs at speeds comparable to SHA-384. Hardware implementation of a slightly modified version of the algorithm presented here runs at significantly faster speeds, namely at 2 Gbits/sec on an FPGA Virtex V of frequency 300 MHz. Modelling suggests that this speed can be increased to 3.4 Gbits/sec. Unlike most other existing hash algorithms, our construction does not follow the Damgard-Merkle philosophy. The hash has several attractive features in terms of its flexibility. In particular, the length of the hash is a parameter that can be set at the outset. Moreover, the estimated degree of collision resistance is measured in terms of another parameter whose value can be varied.

1 Introduction

There is much discussion now about how to construct a good hash algorithm. The main difficulty stems from the fact that the design principles of a hash function are not completely understood. However, some desirable features of a future hash function have been enumerated in several NIST workshops. In particular, the algorithm should provide for a changeable length of hash, collision resistance measured in terms of a tunable parameter and perhaps different functions for online and offline usage.

There are some methods that have been studied in the literature to produce new hash functions from old functions. For example, one might consider the concatenation of two existing hash functions. If one of the hash functions is based on the Damgard-Merkle construction, it is known that (Joux [3]) collision resistance is weakened. One might increase the number of rounds in an existing function but it is not clear that this has an essential impact on collision resistance. Several attempts have been made to identify good design principles of hash functions (see Preneel [7]) but this work seems to still be evolving.

C. Xing et al. (Eds.): IWCC 2009, LNCS 5557, pp. 157–170, 2009.
© Springer-Verlag Berlin Heidelberg 2009

The aim of this article is to describe a new hash algorithm, which incorporates some of the features mentioned above. In particular, the output length of the function is a parameter that can be set at the beginning. Moreover, the degree of collision resistance is expected to depend on another parameter that can also be specified at the outset. An announcement of our work can be found in [6].

The performance of the algorithm is comparable to that of the SHA-family. This comparison was made on an AMD Sempron 2GHz processor 3400+ using 1GB of RAM. In particular, for a 384 bit hash, the speed is comparable to SHA-384. For a 512 bit hash, the speed is 5% faster than SHA-512. The hardware implementation of the algorithm is especially effective. We have shown that the speed of the function reaches 2 Gbits/sec (on an FPGA V running at 300 MHz). Analysis also suggests that with a slightly different choice of parameters and on the same FPGA, we may obtain speeds of approximately 3.4 Gbits/sec. We note that the structure of the algorithm is such that the performance improves for longer files and larger hash sizes. The algorithm has two phases, the second of which does not depend on the length of the message being hashed.

The collision resistance of the function seems to be dependent on the difficulty of solving a family of systems of iterated exponential equations in a finite field. This problem does not seem to have been extensively studied in the literature. However, it does not seem to be tractable by standard methods of analytic number theory.

Summarizing, the hash function that we construct has the following important attributes. Firstly, the length of the output can be changed simply by changing a few steps of the calculation. Secondly, the computation is a bit-stream procedure as opposed to a block procedure. Thirdly, several aspects of the construction, namely the CUrrent Register (CUR) construction (described in Section 3), the compression function and the truncation and exponentiation seem to be novel constructs. Moreover, there seems to be the possibility of some control over collision resistance by the use of "bit strings" as well as the iterated CUR construction, the latter requiring the solution of a system of non-linear iterated exponential equations to invert. As far as we are aware, such equations cannot be solved by standard methods of analytic number theory. Moreover, we are not aware of any other hash function in the literature whose collision resistance involves such iterated exponential equations.

For its performance characteristics,novel design features and dependence on what appears to be an intractible mathematical problem, we believe this hash function is worthy of further attention.

Our construction uses polynomials over finite fields. We note that earlier works have used polynomials over finite fields in the construction of hash algorithms. However, our use of polynomials is very different.

Many well-known hash algorithms that are currently in use are based on the Damgard-Merkle [2], [5] approach. The reader can find a description of such algorithms in the book of Menezes, van Oorschot and Vanstone [4]. While this approach is very elegant, recent work has caused the Damgard-Merkle design methodology to come under close scrutiny. Indeed, the ground-breaking work

of Wang [9] has exhibited weaknesses in some of the most popular Damgard-Merkle based hash functions, including MD5 and SHA-1. Moreover, in the case of MD5, multicollisions can be found by exploiting the Damgard-Merkle structure. Moreover, as explained above, multicollisions can be found for any Damgard-Merkle hash function [3]. Our approach, however, is not based on the Damgard-Merkle methodology.

We introduce three constructions which may be of interest in their own right. The first is the CUR construction. This takes as input a sequence of k polynomials over $\mathbb{F}_2$ of degree $< n$ and produces another such sequence. The second is the compression routine. We describe a construction that takes as input a binary sequence of length k, a sequence of k polynomials over $\mathbb{F}_2$ of degree $< n$, an integer r with $2^n < r \leq k$, and a sufficiently large integer λ, and produces r matrices of 2^n rows and $1 + \lambda$ columns. This construction is invertible. In other words, given the matrices, one can reconstruct both the binary sequence as well as the sequence of polynomials. The compression function is obtained by deleting columns 2 to $1 + \lambda$ of selected rows of these matrices. The third construction is truncation followed by exponentiation in a finite group. In essence, our hash function takes a message given as a binary string of length k and performs a preliminary operation on it to transform it into a sequence of k polynomials over $\mathbb{F}_2$ of degree $< n$. It then invokes the CUR construction and the compression routine. The entries of the compressed matrices are then combined in a Cantor enumeration to produce a single integer. This integer is used in the truncation-exponentiation routine to produce a hash value. There are additional steps that provide the transition between the above constructions, but the above description gives the essence.

2 Binary String to Polynomial Sequence

We assume that the message is sufficiently long, something that can be achieved by padding. (We used 4096 bits of a fixed string as padding. A variant of our algorithm uses padding of a length depending on the size of the message.) There are, of course, many ways to apply padding. Some study of this (in terms of "stretching") can be found in the work of Aiello, Haber and Venkatesan [8] in which a randomized function is used to perform stretching. In our algorithm, the purpose of padding is to ensure that we are able to populate certain auxiliary bit strings. Let us denote by k the length in bits of the padded message M.

Choose an integer n with $3 < n < 11$. The message is then split into overlapping segments which are interpreted as polynomials over $\mathbb{F}_2$ of degree $< n$ More precisely, denote by $M(i, j)$ the substring of M beginning with the i-th bit and ending with the j-th bit. Also, denote by $M[i]$ the i-th bit of M. Let us define $S(M, n)$ to be the set $M(1, n), M(2, n + 1), \cdots, M(k - n + 1, k), M(k - n + 2)M[1], M(k - n + 3)M(1, 2), \cdots M[k]M(1, n - 1)$. Each $M(i, i + n - 1)$ may be thought of as a polynomial of degree $< n$ over $\mathbb{F}_2$. Thus, $S(M, n)$ consists of k polynomials of degree $< n$. Note that the construction of the $S(M, n)$ is a stream procedure.

Thus, from the message M which is a binary string of k bits, we produce k polynomials $M_1, \cdots, M_k$ of degree $< n$.

3 Masking: The *CUR* Construction

Many Damgard-Merkle based hash functions have their compression functions based on block cipher structures. The analogue of that here is the CUR construction which we shall now describe. Given n and a sequence of k polynomials $M_1, \cdots, M_k$ over $\mathbb{F}_2$ of degree $< n$, this construction produces a new sequence of k polynomials, $CUR_1, \cdots, CUR_k$, also of degree $< n$. The CUR construction is one-to-one and length preserving. The construction involves finite field arithmetic. At any given time, we need to store 2^n of these polynomials.

We observe some important aspects of the CUR construction. Firstly, to calculate CUR_i we need to have calculated all CUR_j for $j < i$. Secondly, it is easy to recover $M_1, M_2, ..., M_i$ given the *ordered* sequence $CUR_1, CUR_2, .., CUR_i$, $i < k$. However, during the course of the algorithm and computation of the final hash value, the CUR_i are used to form a weighted sum of integers. From such a sum, it does not seem to be easy to recover the message (see Appendix).

Let $f(x) \in \mathbb{F}_2[x]$ be irreducible of degree n. Thus, there is an isomorphism of fields

$$\mathbb{F}_2[x]/(f(x)) \simeq \mathbb{F}_{2^n}.$$

Denote by ϕ_f the isomorphism of $\mathbb{F}_2$-vector spaces

$$\mathbb{F}_2[x]/(f(x)) \longrightarrow \mathbb{F}_2^n.$$

Let δ (resp. β) be a generator of $(\mathbb{F}_2[x]/(f(x)))^\times$ (resp. $(\mathbb{F}_2[x]/(g(x)))^\times$). We set

$$CUR_1 = M_1 \oplus \phi_f(\delta) \oplus \phi_g(\beta), \tag{1}$$

$$CUR_2 = M_2 \oplus \phi_f(\delta^{int(M_1)}) \oplus \phi_g(\beta^{int(CUR_1)}). \tag{2}$$

For $2 < i \leq 2^n + 1$, we set

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

For $i \geq 2^n + 2$, define two functions d_1 and d_2 as follows. For any bit string B, we define $int(B)$ to be the integer whose base 2 expansion is B. Set

$$d_1 = d_1(i) = i - 2 - int(M_{i-1}), \tag{3}$$

$$d_2 = d_2(i) = i - 2 - int(CUR_{i-1}).$$

Now set

$$CUR_i = M_i \oplus \tag{4}$$

$$\phi_f(\delta^{int(M_{i-1})+int(CUR_{d_1})mod2^n})\oplus$$
$$\oplus\phi_g(\beta^{(int(CUR_{i-1})+int(CUR_{d_2})mod2^n)})$$

for $i = 2^n + 2, ..., k$ with d_1 and d_2 defined by (3).

Once again, we stress that the procedure just described for calculating the values CUR_i is a stream procedure. Moreover, as the result below indicates, the values of the CUR_i uniquely determine the original message M.

Table 1. Sample calculations of CUR

	1	2	3	4	5	6	7	8
$int(M)$	0	0	0	0	0	0	0	0
$int(CUR)$	4	7	6	15	1	9	9	0
$int(CUR^{(2)})$	0	4	2	12	12	15	11	14

	1	2	3	4	5	6	7	8
$int(M)$	0	0	0	0	0	0	0	1
$int(CUR)$	5	8	4	1	9	0	0	4
$int(CUR^{(2)})$	1	9	0	4	12	3	7	1

Proposition 1. *Let M and M' be messages of length k with $CUR_i(M) = CUR_i(M')$ for $i = 1, \cdots, k$. Then $M = M'$.*

The CUR construction can be iterated. We ran implementations in which we performed seven iterations. We remark that the iterated CUR contruction seems to have good diffusion properties. Below we give a "baby" example with a message of length 8 bits and two iterations without padding.

Example 1. We consider the case $n = 4$ and choose polynomials $f(x) = x^4 + x+1$ and $g(x) = x^4+x^3+1$. We choose generators $\delta = x+1$ and $\beta = x^2+x+1$. For illustrative purposes, we work with a message M of length 11 bits.

4 The Compression Function

In the Damgard-Merkle methodology, a hash function consists of a compression function and a domain extender. The compression function takes as input a string of fixed length and produces a shorter string of fixed length. The domain extender provides the means of dividing a string of arbitrary length into substrings that can be fed into the compression function. Our method is not based on this approach. However, at the heart of our construction is a compression function, which we describe in this section.

4.1 The Compression Function

We construct a compression function by extracting "features" of the input message. The final hash value will be computed by "packing" this collection of features.

Let k, n be positive integers, r an integer with $2^n < r \leq k$, and let λ be a sufficiently large integer. Suppose we are given

- a binary sequence (a message) M of length k, the i-th element of which will be denoted $M[i]$
- a sequence C of length k consisting of integers in the range $\{0, 1, \cdots, 2^n - 1\}$
- a one-to-one "random walk" function (depending on M)

$$h \;=\; h_M : \{0, 1 \cdots, k\} \;\longrightarrow\; \{1, 2, \cdots, r\lambda\} \;\subset\; \mathbb{N}$$

with the property that the composite map

$$\{0, 1 \cdots, k\} \;\longrightarrow\; \mathbb{N} \;\hookrightarrow\; \mathbb{Z} \;\longrightarrow\; \mathbb{Z}/r\mathbb{Z}$$

is surjective.

Then, we define $\mathcal{T}_h(M, C)$. It consists of r matrices $\mathcal{M}_1, \cdots, \mathcal{M}_r$, each having 2^n rows and $1 + \lambda$ columns. Except for the first column, the entries of these matrices are 0 or 1. The first column consists of integers in the range $\{0, 1, \cdots, k\}$. We initialize all of the matrices so that every entry is zero. Set $f(i) = \lceil h(i)/r \rceil$. Note that $0 < f(i) \leq \lambda$. For each value of a and b, set

$$\mathcal{M}_a(b, 0) \;=\; \sum_{\substack{h(i) \pmod r \;=\; a \\ b \;=\; C_i}} M[i].$$

Also, set

$$\mathcal{M}_{h(i) \pmod r}(C_i, f(i)) \;= \mathcal{M}[i]. \tag{5}$$

Then we have the following result.

Proposition 2. *Given the entries of $\mathcal{M}_1, \cdots, \mathcal{M}_r$, we can reconstruct M and C.*

Proof. For each triple (a, b, j) with $1 \leq a \leq r$, $0 \leq b \leq 2^n - 1$ and $j > 1$, consider the entry $\mathcal{M}_a(b, j)$. If this is non-zero, it must be equal to 1. In this case, for some i, we must have $h(i) = a + rj$. Such an i is unique as h is one-to-one. For this value of i, we must have $C_i = b$. As we range over a and j, we will get all values of i. Thus, this determines h. Finally, M is determined by (5).

Now we consider a variant of this construction in which we reduce the size of the matrices. This variant may be seen as a compression function. For each $1 \leq a \leq r$, let Θ_a be a subset of $\{0, 1, \cdots, 2^n - 1\}$. Then, the first variant is to replace (5) with the following rule.

$$\mathcal{M}_{h(i) \pmod r}(C_i, f(i)) \;=\; \begin{cases} M[i] & \text{if } C_i \in \Theta_{h(i) \mod r} \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

The argument given in the proof of Proposition 2 shows that given $\mathcal{T}_h(M, C)$, we can determine $M[i]$, C_i and $h(i)$ for those i for which $h(i) = a + rj$ and $C_i \in \Theta_a$. Note that in order to make this determination, we need to invert h. Assuming

that the C_i are equidistributed in $\{0, 1, \cdots, 2^n - 1\}$, this determines $(|\Theta|/2^n r)k$ bits of M. For $b \notin \Theta_a$, we see that the quantity

$$\sum_{\substack{h(i) \pmod r = a \\ b = C_i}} M[i]$$

is known (as it is the $(b, 0)$-th entry in $\mathcal{M}_a$). If we assume that the $h(i) \pmod r$ and the C_i are independently equidistributed, then the number of values of i in the sum above is $\mathbf{O}(k/2^n r)$. Taking into account the constraint imposed by the above equation, we have $\mathbf{O}(2^{\frac{k}{2^n r} - 1})$ possible values for the relevant $M[i]'s$. Now multiplying this over the possible values of a and $b \notin \Theta_a$ gives

$$\mathbf{O}(2^{\left(\frac{k}{2^n r} - 1\right) 2^n r \left(1 - \frac{|\Theta|}{2^n r}\right)}$$

where we have set

$$\Theta = \cup_a \Theta_a.$$

Thus, the number of possible strings M satisfying these constraints is at most $2^{\theta(k - 2^n r)}$ where $\theta = \left(1 - \frac{|\Theta|}{2^n r}\right)$. By our choice of θ, we can thus have some control over collision resistance.

In practice, we will use bit strings that are significantly shorter. This is the second variant of the general construction. In our sample calulations, we chose the length λ to be between 100 and 300 bits. In general, we may choose the length to be a function of the length of the desired output hash. It is therefore possible that $f(i)$ may be larger than λ. If $f(i) \leq \lambda$, we proceed as described above. If $f(i) > \lambda$, then we use the same formulae as above with $f(i)$ replaced by $(f(i) \pmod \lambda) + 1$.

In addition to the above table entries, we construct r additional row vectors e_{i0} (for $1 \leq i \leq r$).

Definition 1. *The first column of matrix $\mathcal{M}_i$ will be denoted T_i and called the i-th* table. *Moreover, for each i, and for each $a \in \Theta_i$, the row vectors $(\mathcal{M}_i(a, 2), \cdots, \mathcal{M}_i(a, \lambda + 1))$ will be denoted e_{ia} and called bit strings.*

We conclude this subsection with some general remarks. The length λ and the size of Θ_a are parameters that can be chosen. In our implementation, we used up to $\Theta_a = 5$ for each table, each of length $\lambda = 200$ bits. Changing the number of bit strings sufficiently increases the number of states of the finite state machine that is related to the considered hash function. In this sense, the number of bit strings controls the collision resistance of the function, even when the length of the final hash value is fixed.

At the end of this operation, we produce tables the average entry of which is $k/(2^n * 200)$. In particular, for a message of 1MB and using $n = 4$, the average table entry will be an integer of size 2500. By construction, the sum of the entries in the tables is the same as the message length, namely k.

4.2 The Function h

We used the following function for h. Let q, g be two positive integers chosen so that $q + g < n$. (In our sample calculations, we used $q = g = 1$). Set

$$h_M(i) \ = \ iq \ + \ \alpha_M(i)g$$

where $\alpha_M(i)$ is the number of $j \leq i$ for which $M[j] \ = \ 0$. Since α_M is an increasing function, h_M is clearly one-to-one.

5 Further Compression

5.1 Factorization

For every sequence of polynomials CUR_i we define vectors R_i, $i = 1, ..., r$ of length $irr(n)$ where $irr(n)$ is the number of irreducible polynomials with coefficients in $\mathbb{F}_2$ of degree less than n. We will call the vectors R_i *spectrums*. We initialize all of the R_i to be zero. The values of the registers will be determined by means of factoring the polynomials $CUR_1, \cdots, CUR_k$.

5.2 The Spectrum

Recall that T_i is a column vector having 2^n entries. Each integer $0 \leq j < 2^n$ may be interpreted as a polynomial P_j over $\mathbb{F}_2$ of degree $< n$. Let us denote by $irr(n)$ the number of irreducible polynomials over $\mathbb{F}_2$ of degree $< n$ and let us list them in some order $Q_1, \cdots, Q_{irr(n)}$ (for example, lexicographically). Factoring the P_j gives us equations

$$P_j \ = \ \prod_k Q_k^{m_{k,j}}.$$

The $\{m_{k,j}\}$ define a $irr(n) \times 2^n$ matrix $\mathcal{R}$ (say). Now we define

$$R_i \ = \ \mathcal{R}T_i.$$

As an example, suppose that $n = 4$ and T_i is the transpose of

$$(13, 10, 11, 11, 10, 9, 12, 4, 5, 15, 12, 8, 4, 13, 14, 13).$$

There are 7 irreducible polynomials which we list in the order

$$Q_1 \ = \ 0, \ Q_2 \ = \ 1, \ Q_3 \ = \ x, \ Q_4 \ = \ x + 1,$$

$$Q_5 \ = \ x^2 + x + 1, \ Q_6 \ = \ x^3 + x + 1, \ Q_7 \ = \ x^3 + x^2 + 1.$$

Applying the above construction, we deduce that R_i is the transpose of

$$(13, 10, 92, 123, 33, 8, 13).$$

The maximum value of a spectrum entry is approximately $k/10$ and the average value is $k/(2^{n+1} * 10)$. In particular, for a message of length 1MB, the spectrum from each table contains about 25 bits. It is not necessary to store more than one spectrum at a time.

5.3 Enumeration

Finally, we produce a single integer from the vectors R_i using Cantor enumeration. For each $d \geq 1$, there is a bijective map

$$c_d : \mathbb{N}^d \longrightarrow \mathbb{N}.$$

For $d = 1$, we can take the identity and for $d = 2$, we can take

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

Given c_n, the map

$$(x_1, \cdots, x_{n+1}) \mapsto c_n(c_2(x_1, x_2), x_3, \cdots, x_{n+1})$$

is a candidate for c_{n+1}. However, there are more optimal choices.

We will be applying the enumeration function to the vectors R_i and as such, we need the functions c_m where $m = irr(n)$ for $4 \leq n \leq 10$. We first record the values of the pairs (n, m) in Table 2. Explicit candidates that produce numbers

Table 2. Number of irreducible polynomials

n	4	5	6	7	8	9	10
m	7	10	16	25	43	71	129

of manageable size for $n = 4, 5, 6$ are given in the appendix. For example, for $n = 4$, we used

$$c_7(s_1, s_2, s_3, s_4, s_5, s_6, s_7) = c(c(c(c(s_1, s_2), s_3), c(s_4, s_5)), c(s_6, s_7)).$$

This number is of the order $(k/10)^{16}$. In particular, for a 1MB file, this is about 40 bytes.

5.4 Knapsack

Based on the bit strings, we compute for each table t, an integer as follows.

$$BS(t) = \sum_{x \in \Theta_t} (x + 1) int(e_{tx}). \tag{7}$$

We will use these integers when we calculate the knapsack.

Thus, to each table, we have integers $c(R_t)$ and $BS(t)$. We set $I_t = c(R_t) + BS(t)$. Now we form the sum

$$I = \sum tI_t$$

which for a 1MB file is an integer of length 320 bits.

6 Truncation and Exponentiation

The third of the "primitives" (the first two being the CUR construction and the compression function) is called truncation followed by exponentiation in a group, and is described as follows. Let

$$H : \{0,1\}^* \longrightarrow \{0,1\}^\kappa$$

be a hash function with output length κ. Let G be a finite abelian group and select an element $g \in G$. Let

$$F : G \longrightarrow \{0,1\}^\tau$$

be a function with $\tau < \kappa$. For a string $M \in \{0,1\}^*$, consider the new function

$$\mathcal{H} : M \mapsto F(g^{int(H(M))}) \in \{0,1\}^\tau.$$

Here, *int* is the function that associates to a bit string the integer that it represents in base 2.

In our implementation, we used $G = \mathbb{F}_{2^\tau}^\times$ or $G = E(\mathbb{F}_{2^\tau})$ for a cryptographically suitable elliptic curve E over $\mathbb{F}_{2^\tau}$. In the first case F is an identification of $\mathbb{F}_{2^\tau}$ with $\mathbb{F}_2^\tau$. In the second case, F is the function that takes the x-coordinate of a point. Note that F is an isomorphism in the first case, and has a kernel of order 2 (counting multiplicites) in the second case.

Note that in both of these cases, the value of $\mathcal{H}(M)$ only depends on the last τ bits. Thus, the calculation is truncation followed by exponentiation in a group. The idea is that the group operation may help to increase the entropy following truncation. This view can be made somewhat plausible by the following result.

The measure of equidistribution of a sequence of elements of a group is governed by the size of certain exponential sums. If we consider the set of residue classes $\{1 \le s \le t\}$ for some parameter $p/2 < t < p$, then the sum

$$\sum_{1 \le s \le t} \exp\{2\pi i a s/p\}$$

is

$$e^{2\pi i a/p}(1 - e^{2\pi a t/p})/(1 - e^{2\pi i a/p})$$

and the denominator shows that for a bounded as a function of p, and t sufficiently large (for example, $t \gg p$), this is not $\ll p^{1-\epsilon}$ for any $\epsilon > 0$. On the other hand, if we let g be a primitive root modulo p, then by a result of Bourgain [1], we have

$$\sum_{1 \le s \le t} \exp\{2\pi i a g^s/p\} \ll p^{1-\epsilon}$$

for some $\epsilon > 0$. In fact, this is true under even the weaker hypothess $t > p^\delta$ for some $\delta > 0$. This result suggests that exponentiation may increase the degree of equidistribution.

7 The Final Steps

7.1 Multiple Splitting

We can perform all of the above calculations for several values of n instead of just a single one. Let $c \geq 1$ and let $n_1, \cdots, n_c$ be integers in the interval $[4, 10]$.

7.2 Outline of the Algorithm

Our algorithm, then, can be described in brief as follows.

PARAMETERS: c, $n_1, \cdots, n_c$, $\{r_j, s_j, g_j, q_j\}$, τ
INPUT: Message M of length k
OUTPUT: Hash value H of M of τ bits
1. Compute the stretching and splitting $S(M, n_j)$ $(1 \leq j \leq c)$
2. Compute the masking $CUR_i^{(n_j)}$ for $1 \leq j \leq c$ and $1 \leq i \leq k$.
3. Compute the tables $T_i^{(n_j)}$ for $1 \leq j \leq c$ and $1 \leq i \leq r_j$. Each table has 2^{n_j} entries and each entry has s_j bits.
4. Compute bit strings and their associated integers $BS_i^{(n_j)}$.
5. From the tables, compute the spectra $R_i^{(n_j)}$ and their associated integers $c(R_i^{(n_j)})$.
6. Use both sets of integers to compute an integer I_j (for $1 \leq j \leq c$).
7. Compute H_j in the group.
8. The final hash value H is the sum of the H_j in the group G.

8 Statistical Tests

8.1 Collision Resistance

The collision resistance of the algorithm is difficult to analyze, but seems to depend on the difficulty of solving iterated exponential equations over a finite field. Indeed, if one were to try to begin with the spectra and reconstruct the values of the CUR, one is immediately led to such equations when one goes from $CUR^{(i)}$ to $CUR^{(i-1)}$. There is also the additional complication of reconstructing the bit strings.

While it is difficult to quantify the collision resistance of the algorithm, it is possible to analyze the output statistically.

8.2 Statistical Tests

We tested the algorithm on all of the statistical tests provided by NIST at http://csrc/nist.gov/groups/ST/toolkit/rng/stats tests.html. We ran the tests for output lengths of 160, 224, 256, 384 and 512 bits. In all cases, the algorithm passed the tests with reserve. Further testing (such as avalanche, etc.) is in progress.

9 Implementation

9.1 Software Implementation

We implemented ERINDALE in C on an AMD Sempron 3400+ processor running at 2GHz and using 1 GB of RAM. The most time consuming step of the algorithm is masking which, on average, takes up to 40 percent of the total time of the computation. Another highly time consuming step is the distribution of the elements between tables and updating the bit strings, taking as much as 50 to 55 percent of the total time. The speed of the algorithm is close to one shown by SHA-384.

9.2 Examples and Test Vectors

In all of the examples below, we computed the exponentiation in $GF(2^{160})^*$ using the generator

$$b957ac049c0efdf70b169150cf39a5763f19134f,$$

(presented in hex form) and all the operations in the field were made modulo

$$d10000000000000000000000000000000000000008.$$

Let M be 50000 repetitions of the ten ASCII characters 0123456789. This is a message of length 0.5MB. We computed a 160 bit hash using 10 tables with 5 bit strings per table of length 256 bits each. The result is:

$$\text{ERINDALE}(M) \;=\; 67dfeb5a9ac7c7bd03776dd3abc8963eb3cbf7db$$

Let M be as above except that we changed the last byte (in hexadecimal notation, from 39 to 38). We computed a 160 bit hash with the same parameters as above to get

$$\text{ERINDALE}(M) \;= dc48d7cda46cc9dda71424fef93c4201513187ac$$

Let M be $500,000$ repetitions of 0 and the same parameters as above. We get

$$\text{ERINDALE}(M) \;=\; fb879ebd36a72ca27e141129bef05cdab24afed5$$

9.3 Hardware Implementation

We used a FPGA Virtex V (at a frequency 300 MHz). Depending on the choice of parameters, the speed of the function can be parameterized and can vary from approximately 300 Mbit/sec up to 2 Gbits/sec.

10 Summary

ERINDALE offers several attractive features such as tunable output length and a tunable measure of collision resistance. Moreover, in hardware it is running at

2 Gbits/sec on an FPGA (Virtex V) and it is expected to run even faster on an ASIC. The design methodology is not based on the Damgard-Merkle approach and is a bit-stream procedure.

References

1. Bourgain, J.: New bounds on exponential sums related to the Diffie-Hellman distributions. C. R. Acad. Sci. Paris Ser. I 338, 825–830 (2004)
2. Damgard, I.: A design principle for hash functions. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 416–427. Springer, Heidelberg (1990)
3. Joux, A.: Multicollisions in iterated hash functions. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 306–316. Springer, Heidelberg (2004)
4. Menezes, A., van Oorschot, P., Vanstone, S.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1997)
5. Merkle, R.: One way hash functions and DES. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 428–446. Springer, Heidelberg (1990)
6. Kumar Murty, V., Volkovs, N.: A polynomial based hashing algorithm. In: SECRYPT 2008, pp. 103–106 (2008)
7. Preneel, B.: Analysis and Design of Cryptographic Hash Functions, Ph. D. Thesis (2003)
8. Aiello, W., Haber, S., Venkatesan, R.: New constructions for secure hash functions (extended abstract). In: Vaudenay, S. (ed.) FSE 1998. LNCS, vol. 1372, pp. 150–167. Springer, Heidelberg (1998)
9. Wang, X.: How to break MD5 and other hash function. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 19–35. Springer, Heidelberg (2005)

A Formulas for the Enumeration Function

Consider the values $n = 4, \cdots, 10$ with corresponding $m = irr(n)$ given by $m = 7, 10, 16, 25, 43, 71, 129$. We give explicit enumeration functions for the first three values. Variations on these can be used to produce formulas for the remaining values.

CASE 1. $m = 7$. We set

$$c_7 = c(c(c(c(s_1, s_2), s_3), c(s_4, s_5)), c(s_6, s_7)).$$

CASE 2. $m = 10$. We set

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

CASE 3. $m = 16$. 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

$$c_{16} = c(E_{16}^{'}, E_{16}^{''}).$$