Message Fingerprint Algorithm (MFA-256)

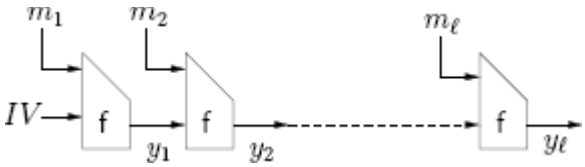
Micah Fullerton
April, 2006

Abstract: The Message Fingerprint Algorithm is designed to be a candidate cryptographically secure hash algorithm. MFA was created by utilizing several design concepts from other popular hash algorithms. One goal of this algorithm is to introduce some possibly new ideas and concepts about iterated hashing. This algorithm is released free to everyone and is to be kept in the public domain.

Introduction

After the break of MD5 and the SHA-1 a new hash algorithm is needed to take computer security into the future. The design of new hash algorithms must take into the account the increasing power of computer systems as well as the advances in complexity theory and cryptanalysis that give birth to new breaks and attacks. These new algorithms must not only have larger output hash values, but their inner workings must be more complex as to produce more chaotic results. The basic structures and primitives of hash algorithms must themselves also be augmented in order to bring hashing to a new tier of security that will stand up against attacks until vibrant advances in cryptanalysis arise.

The formal name for the Message Fingerprint Algorithm is MFA-256, but it may be hereby referred to simply as MFA.



The classic Merkle-Damgard construction of iterated hash algorithms

The basic design of MFA is based upon the well-known Merkle-Damgard (MD) construction of iterated hashing. Using this as a start, MFA implements a few new concepts that further increase the strength of this design. MFA takes a message of arbitrary length and produces a 256 bit hash value. The purpose of the rest of this paper will be to describe the Message Fingerprint Algorithm as well as the ideas and concepts behind its design.

Description of the MFA-256 hash algorithm

MFA was designed with 32-bit machine architecture in mind, therefore a *word* will be defined as an unsigned 32 bit integer. Like usual, addition with 32 bit words is under modulo 2^{32} which contains the value of any word from 0 to 2^{32} - 1 inclusive.

A Twist of the Classic Design

As stated earlier, MFA uses the MD construction which iteratively implements a compression function. Unlike MD5 and SHA-1, as well as several other algorithms, which only employ the single compression function design which the MD construction defines, MFA introduces the concept of using multiple compression functions cooperatively. These are named F, G and H. To further make this design effective, the functions are applied in a square free, non repeating, sequence (See Appendix A). Using them in this fashion further increases the security of the multi-function design. If each function were used one after the other in a purely cyclic manner, then a possible attack might be constructed by attacking the functions together as one larger compression function instead of individually.

Instead of only giving the compression function a block of the message as input, which is what the MD construction defines, an additional 64-bit input is given. This input, denoted C, is made of two 32-bit halves denoted C^{low} and C^{high}. C is linked between each iteration of the compression functions in the same fashion as the chaining variables are. Like in SHA-1, a message expansion scheme is used in MFA on the input to each compression function. In SHA-1, 512 bit blocks of the message are given to each iteration of its compression function. It takes these 16 words, and expands them into an 80 word vector. The last 64 words are the result of a simple recurrence relation. MFA builds from this design to offer a more secure message expansion scheme. Each compression function now takes in a 20 word block of the message denoted M and the 64-bit input C. This makes the block length 640 bits, 20 words, but the total input to each function as 22 words. These 22 words are then expanded into a 70 word vector denoted W. This gives MFA 70 rounds, unlike the 80 rounds used in SHA-1. The 48 words that result from the expansion are from two

recurrence relations, the first and second 24 words being the result of a different equation. The purpose of adding C into the message expansion scheme is to take full control of the message expansion away from an attacker. Each function also injects C into a different position inside W to make each expansion behave differently. After W has been calculated, W_{68} and W_{69} are used as the C^{high} and C^{low} for the next iterated function.

Here are the message expansion schemes of each compression function.

Message Expansion for function F

$$W_i = \begin{cases} M_i \ (for\ i = 0 \rightarrow 19) \\ C_{high} \ (for\ i = 20) \\ C_{low} \ (for\ i = 21) \\ (W_{i-1} <<< 7) + (W_{i-5} \oplus W_{i-11} \oplus W_{i-16}) + ((W_{i-22} + W_{i-3}) <<< 27) \ (for\ i = 22 \rightarrow 45) \\ (W_{i-1} <<< 9) + (W_{i-6} \oplus W_{i-16} \oplus W_{i-31}) + ((W_{i-24} + W_{i-2}) <<< 19) \ (for\ i = 46 \rightarrow 69) \end{cases}$$

Message Expansion for function G

$$W_i = \begin{cases} C_{high} \ (for\ i = 0) \\ C_{low} \ (for\ i = 1) \\ M_i \ (for\ i = 2 \rightarrow 21) \\ (W_{i-1} <<< 23) + (W_{i-3} \oplus W_{i-7} \oplus W_{i-18}) + ((W_{i-22} + W_{i-2}) <<< 7) (for\ i = 22 \rightarrow 45) \\ (W_{i-1} <<< 7) + (W_{i-6} \oplus W_{i-12} \oplus W_{i-21}) + ((W_{i-24} + W_{i-2}) <<< 27) (for\ i = 46 \rightarrow 69) \end{cases}$$

Message Expansion for function H

$$W_i = \begin{cases} C_{high} \ (for\ i = 0) \\ M_i \ (for\ i = 1 \rightarrow 20) \\ C_{low} \ (for\ i = 21) \\ (W_{i-2} <<< 29) + (W_{i-4} \oplus W_{i-9} \oplus W_{i-15}) + ((W_{i-22} + W_{i-1}) <<< 5) (for\ i = 22 \rightarrow 45) \\ (W_{i-24} <<< 13) + (W_{i-5} \oplus W_{i-10} \oplus W_{i-19}) + ((W_{i-2} + W_{i-1}) <<< 19) (for\ i = 46 \rightarrow 69) \end{cases}$$

Each recurrence relation in the expansion scheme takes the basic form.

$$W_i = (W_{i-1} <<< s1) + (W_{i-x1} \oplus W_{i-x2} \oplus W_{i-x3}) + ((W_{i-2} + W_{i-22}) <<< s2) \ (for\ i = 22 \rightarrow 45)$$
$$W_i = (W_{i-1} <<< s3) + (W_{i-x4} \oplus W_{i-x5} \oplus W_{i-x6}) + ((W_{i-2} + W_{i-24}) <<< s4) \ (for\ i = 46 \rightarrow 69)$$

- Xn was each chosen at random but kept largely different in the recurrence relations of each function.
- Sn was also randomly chosen but was kept relatively prime to 32, the word size.
- i-1 and i-2 were chosen to increase the diffusion effect. This is because each word in the expansion is result of the previous two words.

- If i-22 weren't in the relation, then the first message word would have no effect on the expanded output.
- i-24 was used in the second equation so that the first expanded word in the first half of the expansion has an effect in the first word of the second half of the expansion.

By chaining C between each function in this fashion, it is difficult for an attacker to control the state of this variable while also generating a useful W to use for the state update operations. The outcome of the message expansion has a direct effect on C, which then has a direct effect on the expansion of the next message block, or the special block that is appended to the message if no more message blocks are left to calculate. The inputed message block now not only has a direct effect on the chaining variables, but also C itself. Therefore, a small change in the message block would not only effect W, but the next iteration of C in the message expansion scheme which has a large effect on the expanded output of the next iterated compression function. This design should inevitably increase the work needed to successfully generate a multi-block collision.

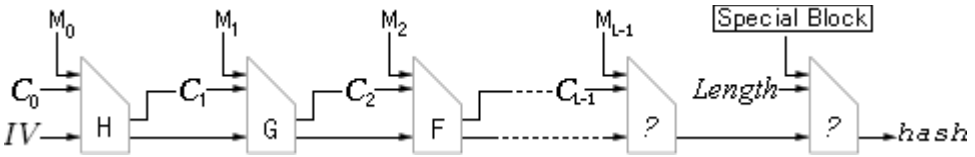
Padding the Message

MFA is designed to compute the hash value for a message of any length. Unlike SHA-1, there is no maximum size of an inputted message. As usual, the message that is given to compute is to be viewed purely as a bit string. The message padding scheme is quite simple. Zero's are added onto the end of the given message to create an even block length. No one bit is appended first. Then, a special block is created and appended to the end of the message. This special block is calculated with the following recurrence relation.

$$M_n = (C^{high} <<< n) + C^{low} \quad (for\ n = 0 \rightarrow 19)$$

Because the special block is calculated by using C, then instead of giving the last iterated compression function it's normal C input, it is replaced with the 64-bit number representing the bit length of the message. Specifically, C^{low} is replaced with the low 32 bits of the length, and C^{high} is replaced with the high 32 bits.

The purpose of the special block is firstly to be hard to control as input to the last iteration of compression function. An attacker has to keep in mind the state of C even after all normal message blocks have been processed. It isn't just thrown away. Even the nature of the recurrence relation used makes it hard generate a specific and useful M to be used in the message expansion. Using the length is a form of MD strengthening, which makes it harder for two messages of different length to hash to the same output.



Structure of the Message Fingerprint Algorithm

State Update Operations

One more strengthened component of MFA is the step operation. In DHA256, a proposed cryptographic hash algorithm, a message word is applied in the step operation twice. The reason for this is to make it more difficult to create a useful local collision pattern. This concept is used in MFA. The local collision was the crack in the dike that lead to the breaking of MD5, SHA-1 as well as several other closely related functions. The step operations for each function are defined as follows.

$$\sigma_0(x) = ((x + x <<< 7) \oplus x <<< 22)$$
$$\sigma_1(x) = ((x \oplus x <<< 13) + x <<< 27)$$
$$\sigma_2(x) = (x + x <<< 16)$$
$$\sigma_3(x) = ((x \oplus x <<< 3) \oplus x <<< 17)$$

Step Operations for F

$B = B + A + W_i$
$A = A <<< 3$
$C = C + K_{B \bmod 64}$
$D = D + \sigma_0(A)$
$F = F + E + W_i$
$G = G + K_{W_i \bmod 64}$
$H = H + \sigma_1(E)$
$rotate()$

Step Operations for G

$B = B + A + W_i$
$C = C + K_{\sigma_2 \bmod 64}$
$D = D + A$
$F = F + E + W_i$
$G = G + K_{F \bmod 64}$
$H = (H <<< 7) + E$
$E = \neg E$
$rotate()$

Step Operations for H

$B = B <<< 9$
$D = D + B$
$F = F <<< 5$
$H = H + F$
$A = A + K_{i \bmod 64} + W_i$
$C = C + \sigma_3(A)$
$E = E + K_{D \bmod 64} + W_i + D$
$G = G + E$
$rotate()$

Rotate() simply rotates each chaining variable so that B = A, C = B, ... , H = A

C is initialized at the start of hashing as follows:

C^{high} = 0x6F2DEC53
C^{low} = 0xA4523B59

These numbers were generated using the following equations.

$$C^{high} = (1 + \log_{10}(e)) * 0\text{xFFFFFFFF}$$
$$C^{low} = (\log_e(\pi) + \log_{10}(\pi)) * 0\text{xFFFFFFFF}$$

K is a constant table made up of 64 words. It is defined as:

```
K = {
      0x46E97E9A, 0x1BCE81B6, 0xBB8EFFB8, 0xE4A2353E,
      0xC07E77EB, 0xB326E7D7, 0xEF16CA66, 0x493D7A1F,
      0x76A8D32C, 0x66FE81D1, 0x5429A7BA, 0x78EEF40B,
      0xC69A78B0, 0xF2C7DEF2, 0xD09F33E8, 0x8EDC5D7A,
      0x8CF9EB54, 0xEEE27FAD, 0x67606050, 0x7F2DD66F,
      0x1751201E, 0x9A48010B, 0x9A2BEE62, 0x35C7B6A6,
      0xE7AF1E8E, 0x00F25B18, 0x5E9E0445, 0x9E8666B5,
      0x8E31E502, 0x5946572C, 0x4CCA2155, 0x7BB503BC,
      0xAD83A969, 0xA6D2EB0C, 0x749F44CB, 0x60574AA3,
      0x97B88D86, 0xEE21E31F, 0x059DF378, 0xB9EA645C,
      0x56727A7B, 0x50F9EA81, 0xC8B66202, 0x54FCEF49,
      0x657EA0F3, 0xDB99936C, 0x3B2687AC, 0x3141F7CA,
      0xE0825A6E, 0xAC89F129, 0xC9CAF1C8, 0x0D507859,
      0x28E068C2, 0x078AAFC9, 0xE0EA14D0, 0xF1631487,
      0x2DFBC84D, 0x4E73FAFA, 0x2455A0CA, 0xDA4F47DB,
      0xCCF6D828, 0x2108321A, 0x8CFD512F, 0x9CA7AB20 }
```

It is calculated from:

$$K_i = (\log_e(i) + \cos(i)) * 0\text{xFFFFFFFF} \; (for\ i = 2 \rightarrow 65)$$

The chaining variables are initialized at the start of hashing as follows:

A = 0xFE826539
B = 0x93634602
C = 0x4EF3D23A
D = 0xCA7F833C
E = 0x70752CF1
F = 0xADCBE99E
G = 0x104C2B8F
H = 0x3923261C

These numbers were generated using the following equation.

$$(\log_e(n) + \log_{10}(n)) * 0\text{xFFFFFFFF} \; (for\ n = \{2, 3, 5, 7, 11, 13, 17, 19\})$$

These numbers were generated with the goal of being random and unbiased. Therefore, the exact equations used in their calculations have no special purpose other than to produce random results.

A message of zero length is not calculated at all. Instead it's hash consists of the initialization vectors. Therefore a message of zero length would return the hash:

FE826539936346024EF3D23ACA7F833C70752CF1ADCBE99E104C2B8F3923261C

Appendix A: Square-free Sequences

A sequence S over a finite alphabet is considered to be square free if it contains no adjacent non-empty subsequences X and Y such that X = Y. Another way of saying this is that S cannot be written as S = XYY, for example. An example of such a sequence is “keyboard”

because it contains no identical adjacent subsequences. The sequence of letters “banana” is not square free because it contains the repeating adjacent “an.”

MFA uses an infinite square free sequence over the alphabet {0, 1, 2}. This sequence starts: 210201210120210…

This sequence can be generated using the parity sequence, which is the parities of all the non-negative real numbers: 01101001100… Using the parity sequence, a square free sequence over a ternary alphabet can be created by calculating the distance from each zero in the parity sequence. Using this sequence, function H is mapped to 2, G is mapped to 1 and F is mapped to 0. So the order in which each function is applied starts HGFHFGHGF...