CERIAS Tech Report 2005-68

FASH: A FAST AND SECURE HASH

by William Speirs

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

FASH: A Fast And Secure Hash

William R. Speirs, wspeirs@cs.purdue.edu

Center for Education and Research in Information Assurance and Security (CERIAS)
Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

Abstract. FASH is a cryptographic hash function that is more then 5 times faster then SHA1 making it more suited for large amounts of data. However, this increase in speed comes at the cost of security. Although all tests performed in this paper show that FASH is as secure as SHA1, FASH has a higher rate of collision. FASH was created as a replacement for SHA1 in applications where speed is much more important than security.

1 Introduction

Representing variable length data with a fixed length hash is the foundation for almost all cryptographic integrity checks. FASH (A Fast And Secure Hash) is designed as a cryptographically secure hash function meaning that it provides three properties: pre-image resistance, strong collision resistance, and weak collision resistance. While SHA1 has displayed these properties over time, its main drawback is speed. Whereas other hashes such as MD4 and MD5 show evidence that the collision resistance property does not hold primarily because of their smaller hash size. Drawing from this FASH creates a 160 bit hash to compete with SHA1 while being more than 5 times faster. This allows FASH to be of great use in applications where simply distinguishing one set of data from another quickly (files for example) is far more important than protecting the data (passwords for example).

2 An Overview of FASH

FASH was designed to be a computationally secure cryptographic hashing algorithm that is significantly faster then SHA1. FASH sacrifices some computational collision resistance for speed; however, it is still believed to be computationally secure and exhibits results comparable to SHA1 when tested for the avalanching effect, pseudorandom number generation, and uniformity in distribution. Since FASH's compression ratio (5 to 1) is worse than SHA1's (3.2 to 1) there are obviously more opportunities for collisions. FASH's digest size was chosen to be 160 bits, the same size as SHA1, so that all tests done could be compared to SHA1. FASH's primary goal is to achieve computational collision resistance as quickly as possible. While pre-image resistance was not a major factor in designing FASH, statistical tests have given evidence that FASH has this property, making it a cryptographically secure hash function.

3 Algorithm Specification

The main design goal of FASH was to avalanche data[1] as fast as possible. It is also designed to be used on a 32-bit architecture (for example the 8086 Intel architecture) making it practical and easy to implement on most personal computers. The algorithm achieves the avalanching effect quickly by making every operation affect every bit of input. FASH uses two steps to hash a block of data: input shuffling and compression. These steps are much like what is found in SHA1 with the exception that FASH's input shuffling step does not expand the input data like SHA1's expansion step. It is believed that FASH is computationally collision resistant even without the input shuffling, although in practice input shuffling should be used. It was found that using both steps improves the uniformity of the hash distribution.

3.1 Pseudocode of Algorithm

Described below is the FASH algorithm in pseudocode (see the appendix for a C implementation of the FASH algorithm). To differentiate between 8-bit and 32-bit quantities, BYTE and DWORD will be used respectively. All addition is modulo the maximum size of the variable. (Ex: 255 + 1 = 0 when using BYTEs.)

The algorithm below describes a single call to FASH (without padding) that processes 100 BYTEs of input data, here on referred to as the message (m). All messages are padded to a multiple of 100 BYTEs by the following method. The last block is padded to 91 BYTEs. If the last block of the message is already larger then 91 BYTEs, then it is padded to 100 BYTEs and a new block is then padded to 91 BYTEs. Then an 8 bit representation of the padding size is concatenated to the end of the block. Finally, the message's size as a 64 bit number is concatenated to the end of the block making this final block a multiple of 100 BYTEs. Padding is done by concatenating a bit of value 1 followed by as many bits as required for the remaining padding with value(s) of 0. This padding is almost identical to that of SHA1 except SHA1 does not include the padding length.[5]

Input Shuffling

$m_i \leftarrow m_i + m_{26-i}$ where m_i is the i^{th} DWORD of the message and $i = 1 \ldots 25$

Compression

1. $w_1 = h_1$, $w_2 = h_2$, $w_3 = h_3$, $w_4 = h_4$, $w_5 = h_5$; where $H = h_1 \parallel h_2 \parallel h_3 \parallel h_4 \parallel h_5$
   Initially h_1 = 0x67452301, h_2 = 0xefcdab89, h_3 = 0x98badcfe, h_4 = 0x10325476, h_5 =0xc3d2e1f0
2. $s_1 = 5$ $s_2 = 7$ $s_3 = 13$ $s_4 = 9$ $s_5 = 20$
3. **for** i = 1, 6, 11, ... , 26
4. $w_1 \leftarrow w_1 + (w_2 \oplus m_{i+3}) + (h_4 \oplus m_{i+1})$
5. $w_2 \leftarrow w_2 + (w_3 + m_{i+4}) \oplus (h_5 + m_{i+2})$

[1] "Whenever one input bit is changed, every output bit must change with probability 1/2"[4], pg 277

6. $w_3 \leftarrow w_3 + (w_4 \oplus m_i) + (h_1 \oplus m_{i+3})$
7. $w_4 \leftarrow w_4 + (w_5 + m_{i+1}) \oplus (h_2 + m_{i+4})$
8. $w_5 \leftarrow w_5 + (w_1 \oplus m_{i+2}) + (h_3 \oplus m_i)$
   Where m_s, h_s, and w_s are of size DWORD.
9. $w_1 \leftarrow w_1 <<< s_1$
10. $w_2 \leftarrow w_2 <<< s_2$
11. $w_3 \leftarrow w_3 <<< s_3$
12. $w_4 \leftarrow w_4 <<< s_4$
13. $w_5 \leftarrow w_5 <<< s_5$
14. $s_1 \leftarrow w_1 + w_2 + w_3$;
15. $s_2 \leftarrow w_2 + w_3 + w_4$;
16. $s_3 \leftarrow w_3 + w_4 + w_5$;
17. $s_4 \leftarrow w_4 + w_5 + w_1$;
18. $s_5 \leftarrow w_5 + w_1 + w_2$;
    Where s_s are of size DWORD.
19. **end for**
20. $H \leftarrow w_1 \parallel w_2 \parallel w_3 \parallel w_4 \parallel w_5$

3.2 Sample Hashes

Below are a few sample hash values computed with padding to check implementations for correctness.

“”	c863f36ed09d712f0473ee382138c8bc631db5ae
“a”	6afb67568a4277aa936e89509b8fcdf66a3ac643
“ab”	f4887b9edfe7567f1a4e824b56d7cdd0596e9d3e
“abcdefghijklmnopqrstuvwxyz”	095ea06492d36188bb7ea83747d462ce6f017bba

4 Design Reasoning

FASH was designed with inspiration from SHA1, MD5, MD4 and from a recently published encryption and authentication algorithm Helix[1]. SHA1 was created from MD5 and MD4. The National Security Agency has given SHA1 its blessing, National Institute of Standards and Technology has adopted it as a standard[5], and more importantly it has stood the test of time. Since the NSA worked on SHA1 the reasons behind the design are not known to this day. This makes it impossible to find answers to questions like, "Why were those particular constant values used?" Whenever possible, ideas and values were borrowed from SHA1; the initial values for w_1, w_2, w_3, w_4, w_5, for example.[2]

The inspiration that Helix provided was that it used only addition, exclusive ORing, and rotations to achieve its security. While Helix, created by a group of experienced cryptographers, has not yet stood the test of time it has passed the initial rounds of testing[1]. The other piece of inspiration taken from Helix was that it seemed to "inject" a piece of data, or a key, at different stages in the encryption. Taking these ideas, along with what was learned from studying SHA1, MD5, and MD4, FASH was designed.

4.1 Shuffling Algorithm

The compression function in FASH is proceeded by a shuffling function. This shuffling function is simple to keep the overall algorithm fast and yet it has a statistical effect on the hash. The shuffling function works by taking the first block of the message, adding it to the the last block of the message, and letting this be the new first block of the message. This process continues through all of the blocks of the message. All of the additions are automatically modulo the maximum value of the block by the hardware. Since the shuffling works on the message, the size of a block is a DWORD. That way the length of the loop used is kept small as to not dramatically impact the speed of the overall algorithm. This method is shown algebraically below with the block size abstracted away:

$$\begin{array}{rcccc} \text{INPUT:} & A & B & C & D \\ \text{OUTPUT:} & A+D & B+C & C+B+C & D+A+D \end{array}$$

Since all of the addition is modulo the maximum value of the block the "double" additions of blocks in the second half of the output, two Cs for example, is not a weakness of this shuffling function. For example you can never have $D + D$ result in D when modulo the maximum size of the block unless $D = 0$. In that case this "double" addition will prevent an attack that simply switches the first and last blocks:

$$\begin{array}{rcccc} \text{INPUT:} & A & B & C & 0 \\ \text{OUTPUT:} & A & B+C & C+B+C & A \end{array}$$

$$\begin{array}{rcccc} \text{INPUT:} & 0 & B & C & A \\ \text{OUTPUT:} & A & B+C & C+B+C & A+A \end{array}$$

Unfortunately this shuffling function does not help to make the algorithm pre-image resistant. It is just simple addition, and a system of equations can be constructed to solve for each of the original variables.

[2]Some testing was done with other numbers and these seemed to work well.

4.2 Compression Algorithm

The compression algorithm has the property of being able to have any bit of input affect, both directly and indirectly, any other bit of input causing a fast avalanche effect. Instead of compressing a single DWORD of the message like SHA1, FASH compresses 5 DWORDs of data per compression iteration. FASH is also more dynamic then SHA1 which, it should be noted, can be extremely dangerous. Making any cryptographic algorithm dynamic, in the sense that the input controls what happens in the algorithm, takes the control out of the designer's hands and puts it into the hands of an attacker. In almost all hashing or encryption algorithms, the constants are defined and are not changed. For example, SHA1 does not dynamically change the amount of rotation based on input. DES and AES do not change their respective S-boxes based on input. FASH does this to make the data avalanche faster. The idea being that starting with "strong" constants and then letting these values change as the algorithm moves forward will cause more complexity for someone trying to create a collision than keeping them constant. However, it should be noted that this dynamic property might be FASH's fatal flaw.

Like SHA1, FASH takes advantage of both linear and non-linear functions. The main difference, with respect to the compression function, is that SHA1 has four (three distinct) non-linear functions that it uses in its compression algorithm. FASH does not create specific non-linear functions but rather imposes non-linearity on the data by using regular addition in conjunction with XORing values. This was done so that FASH could evenly distribute non-linearity among all blocks in a round; whereas SHA1 only applies its non-linear functions to one out of five blocks per round. This is what contributes to FASH's ability to avalanche data very quickly, everything is some what symmetric.

5 FASH Test Results

Statistical tests were run on FASH to help prove its computational collision resistance. If a hash function has a truly unbiased or uniform output then hashing biased input should result in what appears to be an unbiased pseudorandom string of bits. There are a number of statistical tests that can be performed to check how much like a random string of bits something produces. Three sets of tests were run against FASH and SHA1, using SHA1 as a benchmark. It has also been shown that if a function can create a pseudorandom string of bits then that function is a one-way function[2] or pre-image resistance, one of the three criteria for a cryptographic hash function.

5.1 Collision & Avalanche Tests

To test for collisions and the avalanching effect of the algorithms five different sets of data were run through both FASH and SHA1. The five sets of data represent what is believed to be some of the most difficult sets of data for a hashing algorithm to pass. Each set of data only has a single bit of change from input to input[3], with the exception of "all_same" and "alt_bytes". The five sets are: "all_same", where all of the bytes in the input are the

[3]Input sizes were adjusted so that FASH's input was 800 bits and SHA1's input was 512 bits.

same and from input to input the byte used is changed; "alt_bytes", where bytes alternate through all of the byte values (ex. 0x0001...to 0xFFEE...); "55_base", where all bytes are 0x55 with a single bit flipped moving through the bits from least significant to most significant; "AA_base", same as "55_base" except using 0xAA as a base instead of 0x55; "all_bit_change", where the base byte goes from 0x00 to 0xFF and from input to input a single bit is flipped moving through the bits from least to most significant.

Below are the results of comparing the hashed values of the five sets of data without padding. The hashes were first checked for collisions, none were found for either algorithm. Next the bits of the hash were compared from hash to hash, in position, checking to see how many bits were the same. The total number of bits that were the same in all of the hashes, and the max, min, and average bits from hash to hash that were the same are reported below.

FILE: all_same	**FASH**	**SHA1**	**Diff**
Total Bits Same	20370	20417	N/A
Max Hash Bits Same	99	94	-5
Average Hash Bits Same	79.570	79.754	-0.184
Min Hash Bits Same	66	61	+5

FILE: alt_bytes	**FASH**	**SHA1**	**Diff**
Total Bits Same	5240757	5243478	N/A
Max Hash Bits Same	106	105	-1
Average Hash Bits Same	79.968	80.009	-0.023
Min Hash Bits Same	53	47	+6

FILE: 55_base	**FASH**	**SHA1**	**Diff**
Total Bits Same	64458	40710	N/A
Max Hash Bits Same	100	100	0
Average Hash Bits Same	80.573	79.512	-0.085
Min Hash Bits Same	61	60	+1

FILE: AA_base	**FASH**	**SHA1**	**Diff**
Total Bits Same	63400	41128	N/A
Max Hash Bits Same	96	98	+2
Average Hash Bits Same	79.250	80.328	-0.422
Min Hash Bits Same	60	62	-2

FILE: all_bit_change	**FASH**	**SHA1**	**Diff**
Total Bits Same	16386170	10487977	N/A
Max Hash Bits Same	107	112	+5
Average Hash Bits Same	80.011	80.017	+0.006
Min Hash Bits Same	52	53	-1

Looking at the results of the above tests one can conclude that FASH avalanches data almost as completely as SHA1 in these five sets of data. In a perfect avalanche we would see

80 or half of the bits change from hash to hash. The numbers in the **Diff** column represent how much better or worse FASH did against SHA1. A positive number means that FASH was closer to the optimum of 80 then SHA1, and a negative number means that SHA1 was closer to the optimum. In most of the sets of data FASH did worse then SHA1, but not worse then -5 for maximum and minimum bits the same. The average hash bits that were the same, or what is believed to be the most important data point, shows that FASH did only slightly worse then SHA1 with the exception of "all_bit_change". In all cases it was only worse by 0.5 bits. Considering that mathematically FASH has more collisions, it appears that FASH does a good job of avalanching the input.

5.2 Pseudorandom Number Generator Tests

The next set of tests were designed to determine if FASH show properties of a pseudorandom number generator. To test for this property the five basic random number generator tests in the *Handbook of Applied Cryptography* on page 181-182 were used [4]. These five tests; frequency test (monobit test), serial test (two-bit test), poker test, runs test, and autocorrelation test, all give a resulting χ^2 value. Below are the results of these tests, in order, for FASH and SHA1 when the same five data sets as above were used as input.

FILE: all_same	**FASH**	**SHA1**	**Diff**
TEST 1			
Min Zeros	66	61	+5
Max Zeros	99	94	-5
Min Ones	61	66	-5
Max Ones	94	99	+5
Average χ^2	0.957	1.112	+0.155
TEST 2			
Average χ^2	1.948	2.115	+0.167
TEST 3	**m=2 & v=3**		
Average χ^2	2.926	3.271	+0.345
	m=3 & v=7		
Average χ^2	7.008	7.120	+0.112
TEST 4	**v=4**		
Average χ^2	4.923	5.189	+0.266
TEST 5	**d = 1:80**		
Average χ^2	1958.023	1946.349	-11.674

FILE: alt_bytes	FASH	SHA1	Diff
TEST 1			
Min Zeros	53	47	+6
Max Zeros	106	105	-1
Min Ones	54	55	-1
Max Ones	107	113	+6
Average χ^2	0.994	1.003	+0.009
TEST 2			
Average χ^2	1.991	2.007	+0.016
TEST 3	**m=2 & v=3**		
Average χ^2	2.977	3.007	+0.030
	m=3 & v=7		
Average χ^2	7.017	7.003	-0.014
TEST 4	**v=4**		
Average χ^2	4.957	4.946	-0.011
TEST 5	**d = 1:80**		
Average χ^2	1948.504	1947.093	-1.411

FILE: 55_base	FASH	SHA1	Diff
TEST 1			
Min Zeros	61	60	+1
Max Zeros	100	100	0
Min Ones	60	66	-6
Max Ones	99	100	+1
Average χ^2	0.997	1.137	+0.140
TEST 2			
Average χ^2	1.883	2.088	+0.205
TEST 3	**m=2 & v=3**		
Average χ^2	2.916	3.180	+0.264
	m=3 & v=7		
Average χ^2	7.011	7.251	+0.240
TEST 4	**v=4**		
Average χ^2	4.917	5.016	+0.099
TEST 5	**d = 1:80**		
Average χ^2	1933.160	1953.430	+20.270

FILE: AA_base	FASH	SHA1	Diff
TEST 1			
Min Zeros	60	62	-2
Max Zeros	96	98	+2
Min Ones	64	62	+2
Max Ones	100	98	-2
Average χ^2	0.913	0.979	+0.066
TEST 2			
Average χ^2	1.892	1.903	+0.011
TEST 3	**m=2 & v=3**		
Average χ^2	2.946	2.840	-0.106
	m=3 & v=7		
Average χ^2	6.954	7.173	+0.219
TEST 4	**v=4**		
Average χ^2	4.887	4.796	-0.091
TEST 5	**d = 1:80**		
Average χ^2	1936.344	1943.670	+7.326

FILE: all_bit_change	FASH	SHA1	Diff
TEST 1			
Min Zeros	52	53	-1
Max Zeros	107	112	+5
Min Ones	53	48	+5
Max Ones	108	107	-1
Average χ^2	1.000	1.000	0
TEST 2			
Average χ^2	2.008	1.997	-0.011
TEST 3	**m=2 & v=3**		
Average χ^2	2.998	3.005	+0.007
	m=3 & v=7		
Average χ^2	7.004	7.012	+0.008
TEST 4	**v=4**		
Average χ^2	4.956	4.954	-0.002
TEST 5	**d = 1:80**		
Average χ^2	1948.215	1947.910	-0.305

It is clear from the tables above that FASH performed about the same as SHA1 on almost all of the data sets, and tests. From these results one can conclude that FASH shows pseudorandom-number-generator properties and is therefore a one-way or pre-image resistant function. Obviously more testing is needed on more data sets. It should be noted that most of the statistical tests used normally require a minimum number of bits far greater then 160 so the accuracy of the results might be skewed. This is why FASH's results were compared to SHA1's results instead of simply comparing against known χ^2 values for each

test.

5.3 Uniform Distribution Tests

This test was used to test the distribution of hashes into the possible hash values from biased message sets. Three sets of messages were used for these tests: the first, middle, and last 2^{20} possible input values for each algorithm. These ranges are: 0x0000...0000 to 0x0000...000FFFFF, 0x7FFF...FFF80000 to 0x8000...0007FFFF, and 0xFFFF...FFF00000 to 0xFFFF...FFFFFFFF[4]. If the algorithms have perfect uniform distribution into the hash space then the average value of the the hashes will be the average value of the overall hash space. Below are the results for the above data sets.

FILE: first	**FASH**	**SHA1**	**Diff**
Max Hash Value	0x649E**e**36	0xE504**e**36	-0x8066**e**36
Min Hash Value	0x25E8**e**27	0xE504**e**31	+0xE503**e**31
Difference	0x649E**e**36	0xE504**e**36	-0x8066**e**36
Average	0x7FFC**e**36	0xE504**e**36	N/A
Diff From Optimal	-0x3726**e**33	0x6504**e**36	+0x6500**e**36
FILE: middle	**FASH**	**SHA1**	**Diff**
Max Hash Value	0xEADB**e**36	0xC8C2**e**36	+0x2219**e**36
Min Hash Value	0xCFA3**e**26	0xC8C2**e**31	+0xC8C1**e**31
Difference	0xEADB**e**36	0xC8C2**e**36	+0x2219**e**36
Average	0x7FEC**e**36	0xC8C2**e**36	N/A
Diff From Optimal	-0x1315**e**34	0x48C2**e**36	+0x48AE**e**36
FILE: last	**FASH**	**SHA1**	**Diff**
Max Hash Value	0xBF7A**e**36	0xD739**e**36	-0x17BF**e**36
Min Hash Value	0x4070**e**26	0x1EF8**e**31	+0x1EF7**e**31
Difference	0xBF7A**e**36	0xD739**e**36	-0x17BF**e**36
Average	0x7FE9**e**36	0x7B19**e**36	N/A
Diff From Optimal	-0x163C**e**34	-0x4E6E**e**35	+0x4D0A**e**35

From the results above FASH does better then SHA1 for all three sets of data. The most important data point being the difference between the optimal and the average hash value. FASH's difference is two hex digits smaller then SHA1 in two of the three data sets. It is interesting that SHA1 has the same four most significant hex digits for the first and middle sets of data.

5.4 Test Conclusions

Although the tests performed against FASH were limited both in the number of different tests and the data sets run for each test, it appears that FASH's avalanching effect, pseudo-random properties, and uniformity in distribution are as strong if not stronger then SHA1. These tests are a very good indication that FASH is computationally collision resistant and

[4]The number of bits for each range was adjusted for the proper input size of each algorithm.

cryptographically secure. Further cryptanalysis is obviously needed on FASH. All comments on the merit of the tests performed, and on the general security of FASH are welcomed. The discovery of collisions using FASH are especially welcome.

6 Algorithm Time Complexity Analysis

Since the main goal of FASH is to be faster then SHA1 a complexity analysis was done on the operations used in FASH verses those used in SHA1. To aid in removing hardware dependent details certain assumptions were made. First, it was assumed that memory access for both BYTEs and DWORDs are instantaneous. Second, it was assumed that assignments, additions, XORing, ANDing, ORing, NOT, etc., all take a single clock cycle. The time required for loop index calculations was also not considered because those calculations are not the dominating factor of either algorithm.

SHA1			
Event	Cycles	Number	Total
Expansion Part 1	1	16	16
Expansion Part 2	7	64	448
Round 1	20	20	400
Round 2	18	20	360
Round 3	21	20	420
Round 4	18	20	360
Additions	2	5	10
		Total	2,014

FASH			
Event	Cycles	Number	Total
Input Shuffling	4	25	100
Constant Assignment	1	5	5
Round 1	35	5	175
Round 2	30	5	150
Shift Calculation	15	5	75
Assignment	1	5	5
		Total	510

Comparing the number of pseudoclock cycles FASH requires 3.95 times less pseudoclock cycles then SHA1. It is important to remember that a single call of FASH processes 100 BYTEs of message whereas SHA1 only processes 64 BYTEs. Taking this into consideration SHA1 requires 3.93 pseudoclock cycles to hash a single bit of message whereas FASH can hash that same bit in just 0.74 pseudoclock cycles making FASH actually 5.31 times faster then SHA1. Some of this performance might be degraded because of assumptions mentioned above like longer memory accesses on BYTEs then DWORDs, etc. To check the speed of FASH against SHA1 in the real world 6.4 MB, 64 MB, and 640 MB of data was hashed without padding on a 1.33GHz machine.

Data Size	FASH	SHA1	x Faster
6.4 MB	real 0.03	real 0.36	12.00
	user 0.03	user 0.34	11.33
64 MB	real 0.32	real 3.71	11.59
	user 0.31	user 3.50	11.29
640 MB	real 3.23	real 36.67	11.35
	user 3.02	user 34.96	11.58

The results from the real world comparison show just how much faster FASH is than SHA1. The large difference in times between the real world comparison and the time complexity analysis are due to having to make more calls to the SHA1 function then the FASH function. However, these real world numbers should not be discredited because in any real application one would need this increased number of calls as well.

7 Conclusion

The goal of this algorithm was to design a cryptographically secure hashing algorithm that was faster then SHA1, while showing statistical results that suggest it is as secure as SHA1. It is important to remember that mathematically FASH has a higher collision ratio then SHA1, simply because FASH maps more then 1.5 times the amount of data as SHA1 into the same space. However, from the tests conducted it seems as though FASH is as secure as SHA1, and more then 5 times faster. Like all aspects of cryptography only time will prove FASH's worth. Even if collisions can be forced using FASH, the dominance in speed will still make it useful in applications where distinguishing one very large file from another is needed and an adversary would not try to force such a collision.

FASH was not designed to completely replace SHA1 but rather to provide a faster alternative in applications where speed is much more important then security. All comments and questions about this algorithm are greatly appreciated. Please feel free to contact me via e-mail at wspeirs@cs.purdue.edu.

Appendix A: C Implementation of FASH Without Padding

```
#define rol(x,n) ( ((x) << (n)) | ((x) >> (32-(n))) )
#define mod32(n) (n & 31 )

int fash_init(DWORD *hash)
{
   hash[0] = 0x67452301;
   hash[1] = 0xefcdab89;
   hash[2] = 0x98badcfe;
   hash[3] = 0x10325476;
   hash[4] = 0xc3d2e1f0;
}


int fash(DWORD *input, DWORD *hash)
{
   BYTE   *inputBytes = (BYTE*)input;
   BYTE   *hashBytes = (BYTE*)hash;
   DWORD   shift1=5, shift2=7, shift3=13, shift4=9, shift5=20;
   DWORD   w1, w2, w3, w4, w5;
   DWORD   i;

   // input shuffling
   for(i=0; i < 25; i++) input[i] += input[25 - (i+1)];

   // set to last round's values
   w1 = hash[0]; w2 = hash[1]; w3 = hash[2]; w4 = hash[3]; w5 = hash[4];

   for(i=0; i < 25; i+=5)
   {
      // round 1
      w1 += (w2 ^ input[i+3]) + (hash[3] ^ input[i+1]);
      w2 += (w3 + input[i+4]) ^ (hash[4] + input[i+2]);
      w3 += (w4 ^ input[i])   + (hash[0] ^ input[i+3]);
      w4 += (w5 + input[i+1]) ^ (hash[1] + input[i+4]);
      w5 += (w1 ^ input[i+2]) + (hash[2] ^ input[i]);

      // round 2
      w1 = rol(w1, mod32(shift1));
      w2 = rol(w2, mod32(shift2));
      w3 = rol(w3, mod32(shift3));
      w4 = rol(w4, mod32(shift4));
      w5 = rol(w5, mod32(shift5));
```

```
      // shift calculations
      shift2 = w1 + w2 + w3;
      shift3 = w2 + w3 + w4;
      shift4 = w3 + w4 + w5;
      shift5 = w4 + w5 + w1;
      shift1 = w5 + w1 + w2;
   }

   // final assignment
   hash[0] = w1; hash[1] = w2; hash[2] = w3; hash[3] = w4; hash[4] = w5;

    return(0);
}
```

References

[1] N. Ferguson, D. Whiting, B. Schneier, J. Kelsey, S. Lucks, and T. Kohno, "Helix: Fast Encryption and Authentication in a Single Cryptographic Primitive," Available from http://www.macfergus.com/helix/, 2003.

[2] J. Håstad, R. Impagliazzo, L. Levin, and M. Luby, "A pseudorandom Generator from any One-way Function," *SIAM J. Comput. Vol 28 Num 4*, 1999, pp. 1364-1396

[3] Tony Hansen and Garrett Wollman, *RFC 3174 - US Secure Hash Algorithm 1 (SHA1)*, Available from http://www.faqs.org/rfcs/rfc3174.html, Internet RFC/STD/FYI/BCP Archives, 2001.

[4] A.J. Menezes, P.C. Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography*, Boca Raton, Florida: CRC Press LLC 1997.

[5] National Institute of Standards and Technology, NIST FIPS PUB 180-1, "Secure Hashing Standard," U.S. Department of Commerce, April 1995.

[6] Bruce Schneier, *Applied cryptography Second Edition: protocols, algorithms, and source code in C*, John Wiley & Sons, Inc., 1996.