The DCH Hash Function

David A. Wilson
dwilson@alum.mit.edu

October 23, 2008

This document provides a definition of the DCH cryptographic hash function, a candidate for the NIST SHA-3 specification. It is organized following section 2.B of the Federal Register notice, Docket No. 070911510-7512-01.

1 The DCH Algorithm

1.1 Overview

The DCH algorithm is a byte-oriented, endian-neutral, block-cipher-based hash function. It generally follows the Merkle-Damgård structure [4] [11], and compression is performed via Miyaguchi-Preneel iteration [13] on successive message blocks.

Each message block consists of 504 bits (63 bytes); the block is then preprocessed to form a 512-bit input to the compression function. The algorithm uses standard MD-strengthening; the final block is padded to 504 bits and one additional block is added containing the length.

The compression function consists of several rounds, each of which includes a nonlinear substitution, a diffusive linear transform, and a round key addition. The message itself and the previous compression function output are then added together to generate the output.

After all message blocks have been processed, the final hash function output is simply the last compression function output, truncated if necessary to meet the desired digest length. A full definition of DCH follows.

Note. DCH frequently treats data bytes as elements of the Galois field $GF(2^8)$; thus, additions and other operations on data bytes should be considered to be over $GF(2^8)$ unless otherwise specified. This extends to text descriptions; for example, we will frequently use the term "added" to refer

to addition over $GF(2^8)$ (that is, the binary xor). We use the primitive polynomial $\alpha^8 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha^0$, or 0x11D in standard hexadecimal vector notation. This paper assumes that the reader has a basic knowledge of operations in $GF(2^8)$.

1.2 Input Processing

In order to combat certain cryptographic attacks, DCH pads each message block with a *square-free sequence*, as suggested by Rivest [15]. This consists of prepending an 8-bit data-independent value to each 504-bit message block, generating the input to the compression function. This "sequence byte" is reused in the last message block to indicate the termination, as described below.

1.2.1 Square-free Sequence

A *square-free sequence* is a sequence of characters over some alphabet in which no subsequence is repeated. Thus, given the English alphabet, "aa", "banana", and "abcdefghijklmnopqrstuvwxyzopqrstuvwxyz" are not square-free, since they contain the repeated substrings "a", "an", and "opqrstuvwxyz" respectively. "abcabdabeabf" and "nationalinstituteofstandardsandtechnology" are square-free.

We wish to ensure that the sequence of 512-bit blocks is square-free. In order to do this, it suffices to ensure that the first byte of each block is square-free; this in turn can be achieved by combining a square-free sequence of three-bit values with a five-bit counter.

We use the well-known Towers of Hanoi problem to generate our square-free sequence. We define the sequence as follows:

Given three pegs labeled 0, 1, and 2, let peg 0 contain an infinite[1] number of successively larger disks, and pegs 1 and 2 begin empty. The first move consists of moving the smallest disk from peg 0 to 1. Following that, each successive move consists of the next move in the optimal solution to the Towers of Hanoi problem, moving the disks such that a larger disk is never on top of a smaller disk. On each move, the output value of the function is $a + 3b$ where a is the number of the peg the disk moved from and b is the number of the peg the disk moved to. This sequence is square-free [8], and can be computed efficiently in constant time.

[1] In DCH implementations, of course, this number will not be infinite; the reference implementation uses 63 disks, which is sufficient for $2^{63} - 1$ message blocks or almost 2^{77} message bits. For the idealized definition of DCH, however, this number is unbounded.

Move	Value
$0 \rightarrow 1$	3 (011)
$0 \rightarrow 2$	6 (110)
$1 \rightarrow 0$	1 (001)
$1 \rightarrow 2$	7 (111)
$2 \rightarrow 0$	2 (010)
$2 \rightarrow 1$	5 (101)

Table 1: Correspondence between Towers of Hanoi moves and square-free sequence values.

The move value forms the high-order three bits of the sequence byte. The low-order five bits form a counter; the high-order square-free sequence starts with the initial move $0 \rightarrow 1$ and advances each time the counter overflows. Thus, if m_i is the ith message block, we have $m_0 = 01100000$, $m_1 = 01100001$, $...m_{31} = 01111111$, $m_{32} = 11000000$ (since the second move is to move the size-2 disk from peg 0 to peg 2), and so on.

1.2.2 Final Block Padding

DCH uses standard MD-strengthening on the input messages. A single 1 bit is appended to the end of the message. Then, enough 0 bits are appended to the message to reach 64 bits less than a multiple of 504. Finally, the length of the message in bits is appended as a 64-bit value.

The sequence byte in the last block is treated specially. The five-bit counter increments normally; however, the three high-order bits are defined to be 000. Since 000 is never used as a "move" in the square-free sequence defined above, this unambiguously demarks the final message block.

1.3 Compression Function

As a block-cipher-based hash function, the compression in DCH comes from iterative application of a cipher transform to successive blocks of input. The output of the block cipher is then added to the previous cipher state and the message block itself to generate the new cipher state, a design of Miyaguchi and Preneel[13].

The transform itself consists of four rounds. Each round consists of a nonlinear substitution (S-box), a linear transform (similar to the Fourier transform), and the addition of a round key.

In the following sections, m_i refers to the 512-bit input message block

after padding. $m_{i,0}, m_{i,1}, \ldots m_{i,63}$ refer to the individual bytes of m in order; $m_{i,0}$ is the sequence byte. H_i refers to the 512-bit state of the hash function after processing block m_i. Thus, the compression function maps (H_{i-1}, m_i) to H_i. The initial cipher state H_{-1} is defined to be the zero vector.

1.3.1 Nonlinear Substitution

Each byte of the input block is first processed through a highly non-linear substitution function (S-box), $S : GF(2^8) \rightarrow GF(2^8)$. The S-box is defined as follows:

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	03	02	8D	F7	44	A4	79	B9	AE	9E	DE	9B	3E	A9	5E	95
10	DB	71	C3	5B	E3	3D	4F	65	93	DD	56	83	A3	80	48	29
20	6F	EE	3A	52	63	55	2F	89	73	D3	1C	49	25	88	30	6D
30	4B	8A	6C	2D	A7	C0	43	5D	53	21	CC	AA	A8	0F	16	E2
40	35	5C	FB	D6	91	4D	A5	07	33	8B	28	1D	15	64	46	90
50	3B	20	6B	8F	82	19	26	62	10	C2	C8	60	94	0D	34	42
60	27	54	C9	58	BA	C7	14	4E	51	8E	EC	B0	23	EF	2C	31
70	2B	D2	12	DA	EA	F8	D9	7A	D8	74	05	B8	87	CE	FD	FF
80	18	57	A2	1E	7F	CF	E7	B3	4A	32	24	2E	50	6A	01	F6
90	1B	DC	47	4C	98	BF	0C	5F	08	DF	BE	97	AF	0A	C4	A1
A0	1F	81	9C	C5	37	C1	45	06	CD	38	0E	3F	9F	0B	BD	B4
B0	84	E6	ED	68	E8	F1	BC	AC	C6	67	04	78	96	99	AD	B5
C0	11	5A	A6	36	66	BB	A0	9D	D1	F4	61	59	86	7E	AB	39
D0	2A	72	CB	F5	FA	40	D4	D5	13	70	75	7B	9A	09	1A	92
E0	17	3C	E5	F3	85	B2	E1	F2	F9	77	F0	B7	6E	22	B1	69
F0	E0	E4	B6	E9	00	8C	D0	CA	41	D7	EB	76	7C	FC	7D	FE

Table 2: Nonlinear substitution (S-box) values for DCH.

This S-box comes from the function $S[x] = x^{-1} + \texttt{0x03}$, where the inversion and addition occur over $GF(2^8)$. (The zero element is defined to be its own inverse for the purposes of this S-box.)

This formulation was specifically chosen due to the high nonlinear degree of the inverse transform; the addition of the constant `0x03` eliminates fixed points. Unlike the substitution boxes used by other algorithms (such as AES [6] and Whirlpool [14]), we do not require a large amount of diffusion from the substitution box itself; thus, the S-box definition was kept as simple as possible.

We define $m'_{i,j} = S[m_{i,j}]$.

1.3.2 Linear Transform

To perform diffusion, a linear transform (effectively, a partial Fourier transform) is calculated of the outputs of the S-box as follows:

$$m''_{i,j} = \sum_{k=0}^{63} m'_{i,k}(\beta_j)^k$$

Here α^j are elements of $GF(2^8)$ and the $m'_{i,k}$ values are treates as coefficients of the transform. The β_j values for $j = 0...63$ are defined as follows (reading across):

0x01	0x02	0x04	0x08	0x99	0x2F	0x5E	0xBC
0x4F	0x9E	0x21	0x42	0xDD	0xA7	0x53	0xA6
0x4C	0x98	0x2D	0x5A	0xB4	0xD6	0xB1	0x7F
0xFE	0x92	0x39	0x72	0xE4	0x45	0x8A	0x09
0x12	0x22	0x4E	0x9C	0x25	0x4A	0x44	0x88
0x0D	0x1A	0xD7	0xB3	0x7B	0xF6	0x0B	0x16
0x2C	0x58	0x6E	0x0A	0x14	0x28	0x50	0x93
0x3B	0x76	0xEC	0xDC	0xA5	0x57	0xAE	0xA1

Table 3: Bases β_j for the DCH linear transform. Within one block, the jth output byte is equal to $\sum_i m_i(\beta_j)^i$.

0x00	0x01	0x02	0x03	0x44	0x45	0x46	0x47
0x88	0x89	0x8A	0x8B	0xCC	0xCD	0xCE	0xCF
0x10	0x11	0x12	0x13	0x14	0x55	0x56	0x57
0x58	0x99	0x9A	0x9B	0x9C	0xDD	0xDE	0xDF
0xE0	0x65	0x22	0x23	0x24	0x25	0x66	0x67
0x68	0x69	0xAA	0xAB	0xAC	0xAD	0xEE	0xEF
0xF0	0xF1	0xBA	0x33	0x34	0x35	0x36	0x77
0x78	0x79	0x7A	0xBB	0xBC	0xBD	0xBE	0x3F

Table 4: β_j expressed as powers of α in $GF(2^8)$.

This structure organizes the values by differences of 0x11 (17), the largest prime factor of 255 (the number of unique values of α^i in $GF(2^8)$),

in order to compute the transform efficiently. Sixty of the values are computed in this fashion; three (β_{16}, β_{33}, and β_{51} differ by `0x55` (85) for similar reasons. The last value (β_{64}) is arbitrarily set.

1.3.3 Key Addition

In order to introduce byte-level variations, as well as to create differentiation between rounds of the compression function, a round key is added to the message block on each round.

$$m_i''' = K_r + m_i''$$

Aside from pathological cases (such as the zero vector), it is not expected that certain keys are provably stronger than others. Thus, to save on memory footprint, the round keys are taken directly from the definition of the S-box. If $K_r(i)$ denotes the ith byte of the key for the rth (zero-indexed) round, we define $K_r(i) = S[64r + i]$ for $0 \leq r < 4$. Since the original invocation of the S-box is keyed on byte value and the round key addition depends on byte position, it is not expected that the reuse of the S-box in this manner will lead to any vulnerabilities.[2]

1.3.4 Round and Block Chaining

The output of the key addition is used as the input to the next round of the compression function. The compression function operates for four rounds. At that point, if m_i^* is the output of the final key addition, then we set

$$H_i = H_{i-1} + m_i^* + m_i$$

and move on to process block $i + 1$.

1.4 Output

After processing the final message block m_i as described above, the output of the DCH function is simply the final compression output H_i, truncated to the desired digest length. DCH thus supports any digest length up to 512 bits.

[2]In the version of DCH presented in this paper, the four 64-byte round keys can be exactly provided by the 256-byte S-box. If, however, the block size and/or number of rounds are changed (see "Tunable Parameters", Sec. 1.6), additional round keys may be either taken from alternate sections of the S-box (e.g. $K_4(i) = S[32 + i]$) or the round keys may be replaced with arbitrary constants.

1.5 Security Argument

Here we provide security rationales for the design choices detailed above. Further analysis can be found in section 5.

1.5.1 Message Padding

Overall, the message padding works to combat various multiple-block attacks. The well-known MD-strengthening unambiguously pads the message to a multiple of the block size and, by incorporating the length, prevents prefixing attacks in which any collision in the intermediate state of a hash function yields a collision in the function itself–the length itself must be matched as well.

The message padding incorporates a *square-free sequence*[3] in order to foil additional message-extension attacks, such as those proposed by Dean [5] and Kelsey and Schneier [9]. In particular, the sequence prevents creation of a *fixed point* of the hash function, in which the state of the hash computation is not affected by the insertion of one or several message blocks, allowing an attacker to defeat MD-strengthening by arbitrarily adding message blocks.

The addition of a sequence byte also allows unambiguous demarcation of the final message block. This prevents extension attacks where the attacker uses a hash value as a chaining input to additional iterations of the compression function, without requiring additional postprocessing (frequently involving inefficient dropping of output bits or additional computation, such as the methods proposed by Coron et al. [2]).

1.5.2 Compression Function

The first section of the compression function is the nonlinear substitution. The S-box has been designed to provide a very simple, highly nonlinear [12] function with no fixed points. The simplicity of the S-box means that in itself it does not provide optimal diffusion of a single bit difference into its byte; however, it is immediately followed by a linear transform which is designed with optimal diffusion in mind.

The Fourier-based linear transform ensures that every input byte affects

[3]The use of a square-free sequence in this manner was proposed by Rivest [15], who suggests using an *abelian* square-free sequence rather than a simple square-free sequence. We opt for a simpler sequence since it is not at all clear that the security of the algorithm would be enhanced by using an abelian square-free sequence instead; for known attacks it appears that any square-free sequence is sufficient.

every output byte in every round of the compression function.[4] In addition, as a mathematical transform rather than a logical combination of bits, a one-bit input difference can yield a variable amount of output differences within *each* byte of output. Given the large number of bit variations caused by a single input bit change, this is expected to make attempts differential cryptanalysis very computationally intensive.

To emphasize the strength of the diffusion properties, a reduced-round version of DCH was subjected to Strict Avalanche Criterion [19] testing. After only two[5] rounds of the compression function, the results of the SAC testing were virtually indistinguishable from random. No systematic bias was detected in any particular output bit with the change of any particular input bit. The full results, being a 504x512 table, are too large to reproduce in this paper; however, the results were consistent with a standard binomial distribution for each output bit with each input bit flip. The overall calculated variance of the distribution over a large sample of inputs was within 0.07% of the mathematically predicted value.

The addition of a round key introduces byte-level variability into the function, and prevents successful cryptanalysis of a single round of the compression function (e.g. finding a fixed point of the (S-box + linear transform) operation) from directly breaking the security of the entire function.

The actual compression occurs by adding the processed message block, the message block itself, and the previous hash state together. This is a common design developed by Miyaguchi and Preneel [13]; the combination of the previous hash value and the processed value provides compression, while the addition of the original message block prevents an attacker from performing a simple inversion (since the round key addition, the linear transform, and the nonlinear substitution are all easily invertible).

1.6 Tunable Parameters

The most obvious tunable parameter of DCH is the number of rounds in the compression function. As with most round-based hash functions, the

[4] By comparison, in the Whirlpool hash function [14], a single input byte only affects eight output bytes in a single round; multiple rounds are required in order for a small change to diffuse through to the entire output. It is therefore expected that DCH will require fewer rounds than Whirlpool to achieve the same level of diffusion.

[5] Since the nonlinear substitution occurs before the diffusion step, after a single round a one-bit difference in messages will correspond to a somewhat predictable output variation. Though the differences will propagate to every output byte, the differences between the output bytes will be related. Thus, two rounds were used for the statistical tests despite the diffusive power of a single round.

number of rounds can be increased or decreased with a corresponding linear difference in performance. It is expected that an increase in the number of rounds will also have a direct correlation with the security of the hash function; the addition of unique round constants ensures that cryptanalytic attention must be paid to each individual round, and the Miyaguchi-Preneel iteration ensures that a collision in the compression function cannot result from a collision in any of the individual rounds.

Additionally, the block size of DCH can be tuned. Although in this document (and the reference implementation) the compression function is defined to handle blocks of 64 bytes (yielding a message block size of 504 bits), and the round keys and other constants are defined with such in mind, nothing about the essential design of DCH necessitates any particular block size. The advantage of a larger block size can be seen in the diffusion step; the transform by its nature diffuses each byte over an entire block, so a larger block size will generate a higher amount of diffusion within each block processing. The disadvantage is, again, performance–even in the ideal case the Fast Fourier transform algorithm operates in $O(nlgn)$ time, and since we operate in the field $GF(2^8)$ the linear transform is less efficient. Thus, doubling the size of the input and output will more than double the time taken, yielding a net efficiency loss in time per byte processed. Since we perform the transform over bytes in $GF(2^8)$, the maximum block size for the compression function is $2^8 - 1$ bytes or 2040 bits, yielding a 2032-bit message block size.

It should also be noted that for digest lengths less than 512 bits, it is possible to reduce the block length, resulting in performance gains. The obvious choice would be to operate on 256-bit blocks for digest lengths less than or equal to 256 bits (somewhat analogously to SHA-256 operating separately from SHA-512); however, any block length can be used if the linear transform for that block length is properly defined. For simplicity, however, in this initial submission only the 512-bit block size is used so that a single algorithm can satisfy all required output lengths.

2 Computational Efficiency

Since DCH as defined in this paper performs the same operations for all digest lengths (only truncating the final result if shorter outputs are necessary), all digest lengths have the same performance.

For 32- and 64-bit performance testing, the following PC was used:

- Processor: Intel Core 2 Duo E7200 (2.53 GHz)

- Memory: 2GB PC5300 DDR2
- Operating systems: Ubuntu 8.04 (32-bit), Ubuntu 8.04 (x86-64)
- Compiler: gcc 4.2.3
- Compiler flags: `-O4 -funroll-loops`

To estimate performance on the NIST SHA-3 reference platform, the performance results were linearly scaled by a factor of 18/19 to account for the difference in processor speed. Note that all of the below numbers give performance running on a single core.

2.1 32-bit Performance

The estimated performance on the 32-bit reference platform is 10.6 MB/s on a single core. This corresponds to approximately 14200 clock cycles per message block, or 230 cycles per input byte.

The first time DCH is run, it sets up a multiplication table over $GF(2^8)$, taking approximately 530000 processor cycles; once this table is set up, subsequent calls to Init() each take approximately 1150 cycles.

2.2 64-bit Performance

The estimated performance on the 64-bit reference platform is 14.0 MB/s on a single core. This corresponds to approximately 10900 clock cycles per message block, or 170 cycles per input byte.

Setting up the initial table takes approximately 530000 processor cycles; subsequent calls to Init() each take approximately 1100 cycles.

2.3 8-bit Performance

To estimate the performance on 8-bit processors, the μcSim (s51) microcontroller simulator was used to simulate an MCS51-compatible processor. The code was compiled using sdcc 2.7.0.

The DCH implementation used takes approximately 10kB of ROM and 500 bytes of temporary storage, in addition to the memory required for the message itself. The algorithm takes approximately 110000 clock cycles per message block (1800 cycles per byte) for long messages. In addition, each Init() takes approximately 10000 cycles. It is likely that these figure can be reduced significantly through dedicated 8-bit assembly programming.

2.4 Speed/Memory Tradeoffs

The most relevant speed/memory tradeoff is the manner in which multiplication is performed in $GF(2^8)$. First of all, it is convenient to construct lookup tables to hold both the $i \rightarrow \alpha^i$ mapping and its inverse. Multiplication can then be performed with a few operations (through logarithmic addition). Such tables require only 512 bytes each (256 bytes for each table). The optimized implementations, however, precompute a full multiplication table over $GF(2^8)$. This table greatly speeds up multiplication (since it results in one table lookup rather than several); however, it requires a 256x256 table, using 64kB of extra memory (as well as an up-front performance cost to compute the table).

3 Test Outputs

A few selected outputs are given below. For the full range of test outputs, please refer to the files in the KAT_MCT directory accompanying this paper.

Note: The following values are for 512-bit DCH. For shorter digest lengths, truncate the output.

Message:	"" *(zero-length message)*
Digest:	`49432EF52B4B024E 44317DA3D021E9A6`
	`AF1096B83D5C3019 289FD037A00C1C21`
	`E119FF032FF9E017 E20C268FC272CD96`
	`9F5F72C3927EA35D 94E3C1FC97E8E4D7`
Message:	`0xCC` *(one byte)*
Digest:	`9D4C87F1B9A6DCB4 1E60AE3526DAF54A`
	`E0CE624C102C7425 13A23CB6CFA227A2`
	`A20F88403EFE844D 00E44FA1B7510789`
	`50E3C480E4B03329 E3CA9C022081531D`
Message:	16777216 repetitions of the ASCII text
	`abcdefghbcdefghicdefghijdefghijk`
	`efghijklfghijklmghijklmnhijklmno`
	(1GiB)
Digest:	`4032F50185CD9B28 C787540CB41CC179`
	`9807A603BA54E21A C8A85C3F2FF8BEB6`
	`E9A280BE7CE1CB40 74BB3B2C0144F108`
	`3B95CCF5894836AF 482551D73FF1043B`

4 Expected Strength

Given the high diffusion, nonlinearity, and relatively simple structure of DCH, we expect it to provide a level of security consistent with an ideal cryptographic hash function. Specifically,

- We expect that given an n-bit output of DCH, when used with HMAC to construct a PRF, that PRF resists any distinguishing attack requiring much less than $2^{n/2}$ queries.
- We expect DCH to be *collision-resistant*: It should take at least $2^{n/2}$ work to find two messages that have the same n-bit DCH hash value.
- We expect DCH to be *preimage-resistant*: Given an n-bit output of DCH, finding an input message that hashes to the given output should require at least 2^n work.
- We expect DCH to be *second-preimage-resistant*: Given an n-bit output of DCH and a message that hashes to that output, generating a second message that hashes to the same output should require at least 2^n work.
- Given the MD-strengthening and sequence byte, DCH is fully resistant to *length-extension attacks* (that is, such attacks do no better than the work factors above).
- Given any fixed m-bit subset of the output of DCH, we expect the above conditions to hold with m replacing n (apart from trivial attacks in which an attacker computes hash values before fixing the m-bit subset).

5 Analysis

5.1 Known Attacks

5.1.1 Prefixing, Fixed-Point Insertion, Appending, and Other Block-Level Attacks

Due to the combination of MD-strengthening and the sequence byte, DCH is impervious to a large number of block-level attacks.

- The addition of the message block length prevents arbitrary prefix collisions from resulting in a hash-function collision; the length of the input messages must match as well.

- Should an attacker attempt to find a fixed point of the hash function in order to obtain a set of varying-length messages with similar internal state until the message length is hashed (in an attempt to evade the previous point), the square-free sequence will not match, causing the hash values to be entirely different.

- The sequence byte is used to specially flag the last block of a hash computation; thus, the final output of the hash function does not match what the internal state of the hash function would be when processing a message consisting of the original message plus the MD padding. Thus, an attacker cannot perform an extension attack in which he uses one hash value to compute another hash of a related message by simply adding extra message blocks to the end.

While an attacker can certainly precompute a "rainbow table" of many possible message blocks, the above factors prevent the table values from being used except by messages with the exact same length and block positions. This results in such attacks being no better than brute force.

5.1.2 Differential Cryptanalysis and Message Modification

Recent attacks on MD5 [18] and SHA-1 [17] have relied heavily on differential cryptanalysis, using message modification to derive constraints on the internal state of the hash function and, thus, on the original message. DCH has an entirely different structure from the MD4 family of hash functions, so it is unlikely that the techniques used in the cryptanalysis of SHA-1 will be effective against DCH.

In particular, a difference in a single message bit corresponds to a difference in only a few expanded message bits in SHA-1, meaning that different message bits were only introduced in a few of the 80 rounds. This enables a message modification attack in which constraints placed on the input message can increase the probability that a second preimage can be generated. By contrast, in each round of DCH, every byte of the input affects every byte of the output of that particular round. Thus, a cryptanalyst attempting to perform message modification (or, in general, any sort of differential cryptanalysis) must engineer a collision in not simply a few differing bits at a time, but in an entire message block.

5.1.3 Side-Channel Attacks

Certain recent attacks have exploited side-channels; rather than cryptanalyzing the algorithm in the abstract, they detect information during the processing of an algorithm through analysis of the process of computation itself; for example, through timing or power usage.

Data-dependent algorithm operation. Certain algorithms operate in different manners based on the input data. If the time of execution depends on the input data, for example, then timing analysis of the execution of the algorithm may lead to information leakage about the original message (or, in the case of encryption functions, the secret key). For example, the RC5 block cipher uses data-dependent rotations, making it vulnerable to this type of attack [7].

DCH is not generally vulnerable to this manner of attack; no aspect of the algorithm definition is data-dependent. In the reference implementation, the only data-dependent operation occurs during multiplication of the zero element in $GF(2^8)$; however, in the optimized implementation both the zero and nonzero multiplication is performed identically (by table lookup).

Cache timing attacks. Recently, online attacks have been mounted against the AES block cipher [1]. Briefly, they exploit not the cipher itself, but the fact that the cipher is often run on a general-purpose computer with shared memory; by strategically making memory requests while AES is running, another program on the same machine can cause parts of the AES table to be replaced in the processor's cache. By timing analysis, the program can then determine when a cache miss occurs in the execution of AES, and thus gain information about the internal processing of the algorithm (including the secret key).

Since the DCH implementation similarly uses lookup tables for the S-box as well as for multiplication in $GF(2^8)$, it is in principle vulnerable to the same timing attacks. While the "key" in DCH is fixed and public, it is possible that online cache timing analysis may yield information about the input message block.

We make no effort to directly combat these timing attacks; the problem is not algorithmic but implementational. Implementations of DCH (specifically, those running on a shared cache) may take steps to combat this attack, include delaying features in order to obfuscate the running time or OS-level controls on viewing process resource consumption. These attacks, and the defense against them, are tangential to the hash function definition itself.

5.2 Constants

5.2.1 S-box

The nonlinear substitution, as defined in section 1.3.1, was created via the transformation

$$S[x] = x^{-1} + \texttt{0x03}$$

where all operations occur in $GF(2^8)$ and the inverse of 0 is defined to be 0. The inverse transform was chosen due to its simplicity and high degree of nonlinearity [12]; the constant `0x03` was chosen as the smallest additive constant that would cause the S-box to have no fixed points.

There are a number of transforms with a similarly high nonlinearity and lack of fixed points; NIST may choose to replace the S-box with another if desired.

5.2.2 Linear Transformation

The overall structure of the linear transform was chosen to ensure that all rows were linearly independent. The exact values for the bases β_j were chosen specifically in order to compute efficiently using an analogue to the recursive algorithm for the Fast Fourier Transform.

Standard $O(nlgn)$ algorithms for the Fourier transform do not work in general over Galois fields since they rely on the fact that two distinct elements in the field have the same square. Since the nonzero elements of $GF(2^8)$ are α^i for $0 \leq i \leq 254$ and $\alpha^{i+255} = \alpha^i$, every element has a unique square root ($\alpha^{i/2}$ for i even, $\alpha^{(i+255)/2}$ for i odd) and, consequently, no two elements have the same square.

However, since $255 = 3 \cdot 5 \cdot 17$, multiple elements have the same cube ($(\alpha^i)^3 = (\alpha^{i+85})^3 = (\alpha^{i+170})^3$), and similarly for the fifth power. Thus, an analogue to the recursive calculation of the FFT can be performed two levels down, effectively calculating 15 transformed values simultaneously.

The β_j values in the linear transform therefore start with α^i for $i = 0$, 1, 2, 3 and, for each of these, also calculate $i+17$, $i+34$, $i+51$, ... $i+238$. This provides sixty of the sixty-four required β_j values; three of the others can be computed by one level of FFT-analogous recursion ($i = 16$ was arbitrarily chosen as the starting point, also yielding 101 and 186), and the final β_j value was arbitrarily chosen to be 63.

NIST may choose other constants for the β_j values if desired. So long as they are all distinct, the resulting system will be linearly independent; if values differing by 17 are chosen it is possible to greatly speed computation.

5.2.3 Round Keys

The round keys of DCH were chosen to coincide with the S-box based on ease of implementation (particularly in low-memory environments).

Should NIST desire, other constants may be substituted for the round keys (whether related to the S-box or not).

5.2.4 Other Constants

Although the main constants used in the operation of DCH are the S-box, linear transform bases, and round keys, technically a choice has been made on certain other "incidental" constants. These choices have been summarized as follows:

1. **Block size and number of rounds**: These are explicitly listed as tunable parameters of the hash function (see section 1.6); the default values were selected to provide an acceptable level of security with a reasonable performance.

2. **Square-free sequence**: The use of the Towers of Hanoi problem was selected as an easy-to-describe method of generating a square-free sequence; moreover, it can be easily calculated in constant time. The structure of the sequence byte is a modified form of that suggested by Rivest [15]; the exact mapping between Towers of Hanoi moves and three-bit values was selected to be easy to calculate.

3. **Primitive polynomial**: The choice of 0x11D as a primitive polynomial over $GF(2^8)$ was arbitrary.

5.3 Third-Party Analysis of DCH

As of the date of this submission, there are no known published materials analyzing the security of the DCH hash function.

6 Advantages and Limitations

6.1 Diffusion Properties

DCH was created with strong diffusion properties in mind. The centerpiece of this diffusion is the Fourier-based transform in the compression function, which ensures that differences in each input byte result in differences in every output byte of the message block. Also, the mathematical nature of

the transform means that the relationships between individual bits of input and output within each round are extremely complicated (unlike e.g. SHA-1). By alternating this transform with nonlinear and key-additive steps, the compression function can achieve near-optimal diffusion in only a few rounds–the results after two rounds were essentially identical to random.

Using the Fourier transform in a cryptographic hash function is not a new idea; Schnorr [16] and more recently Lyubashevsky et al [10] have also used it in order to achieve strong diffusion (among other properties). DCH takes this idea and uses it to provide diffusion within a robust cryptographic structure, as discussed in the next section.

6.2 Cryptanalytic Basis of the DCH Structure

DCH is a block-cipher-based hash function and therefore bears some cryptanalytic similarities to ciphers such as Rijndael/AES and hash functions such as Whirlpool. In particular, unlike the descendents of MD4 (including SHA-1 and the SHA-2 algorithms), DCH is transformative–it relies not on a complicated series of bit-shifts but rather on explicit substitution, mathematical operations for diffusion, and constant addition.

DCH uses well-known and generally-accepted techniques such as the Miyaguchi-Preneel design for block iteration [13] and the inverse mapping over $GF(2^8)$ for nonlinearity [12].

Furthermore, DCH is similar in overall structure to Whirlpool [14], which has undergone extensive cryptanalysis. DCH improves upon the diffusion properties present in Whirlpool by providing a more thorough transform operation, and adds the sequence byte in order to thwart attacks upon multi-block messages.

6.3 Implementation on Different Architectures

The DCH algorithm is flexible enough to be implemented on a wide range of architectures. Two key use cases are analyzed here: 8-bit processors for embedded applications, and parallelizability for multi-core or dedicated-hardware implementations.

6.3.1 8-bit architectures

All operations in DCH are byte-oriented; DCH can thus be implemented on an 8-bit processor in a straightforward manner. Given sufficient memory, an 8-bit implementation of DCH is virtually identical to the 32-bit implementation.

DCH does require several hundred bytes of working memory in order to store the hashState object as well as temporary storage. The most significant contributor to this memory requirement is the state of the square-free sequence, which requires approximately 200 bytes in order to generate a sequence for messages up to $2^{64} - 1$ bits. If the length requirement is relaxed–that is, if the processor is only used to hash much shorter messages (a reasonable assumption, given the computationally limited nature)–then this value can be reduced significantly.[6]

6.3.2 Parallel computation

DCH is efficiently parallelizable in a number of different ways. Most obviously, the compression function–where DCH spends the vast majority of its time–operates independently for each message block. Once the sequence bytes are calculated, each message block can be processed in parallel, and the results XORed together. DCH thus scales virtually linearly with the number of independent processors for long messages.

Additionally (particularly in dedicated hardware implementations), parallelism can be used within each message block computation. The linear transform–again, the performance bottleneck–can be parallelized using known techniques for FFT parallelization, yielding a severalfold increase in throughput.

6.3.3 Dedicated hardware implementations

A large factor in the relatively slow performance of DCH is slowness of multiplication in $GF(2^8)$ on a general-purpose computer. Each multiplication takes several clock cycles even when a large (2^{16}) amount of memory is devoted to a full lookup table (such a table takes a significant amount of time to initially construct as well). The linear transform involves a large number of such multiplications and is by far the performance bottleneck.

Given dedicated hardware, multiplication in $GF(2^8)$ can be optimized and pipelined. It is thus expected that DCH will benefit greatly from hardware implementation (perhaps even moreso than other algorithms) since it has a specific bottleneck that can be addressed in hardware.

[6] Specifically, an implementation using n "disks" will require $3(n + 1)$ bytes for the "pegs" and will support messages up to $32 \cdot (2^n - 1)$ blocks, or slightly less than 2^{n+14} bits.

6.4 Algorithmic Flexibility

DCH provides a number of general areas of flexibility beyond the SHA-3 requirements set out by NIST.

6.4.1 Tunable Parameters

As noted in Section 1.6, DCH is tunable not just in the number of rounds, but also in its block size, providing an additional level of security/performance tradeoffs. While upon official release these parameters will be fixed, they provide additional options during the analysis period, both to study weakened versions from a cryptanalytic standpoint and to give NIST additional flexibility in selecting security parameters of the algorithm.

6.4.2 Message Digest Size

Since DCH simply truncates its final output to the desired length, it can support any digest size up to its block length.

6.4.3 Long Message Support

Currently, the definition and implementation of DCH support any message up to $2^{64} - 1$ bits long. However, the only length limit in the definition is the addition of length during MD-strengthening (section 1.2.2); if desired, this value can easily be expanded to up to an entire message block. Thus, by modifying this parameter, DCH can in principle support messages up to $2^{504} - 1$ bits (not that computing the hash of such a message would be feasible). Implementation-wise, the only caveat is the square-free sequence, which requires $O(lgn)$ bits of internal storage to support messages up to length n.

6.5 Performance

The major weakness of DCH is its performance; it is significantly slower than current hash algorithms such as SHA-512. To some extent this is expected; by construction, every byte of a 512-bit block affects every other byte in every round of the compression function. In addition, the byte-oriented nature of DCH will result in a comparative performance loss against word-oriented algorithms on general-purpose processors.

However, given the attacks on MD5 and SHA-1, and the structural similarities of SHA-256 and SHA-512 to these earlier descendents of MD4, a

performance degradation for moving to a new hash function is perhaps to be expected. Through its different architecture as well as the arguments presented above, DCH provides numerous security benefits in exchange for the decrease in performance.

References

[1] Daniel J. Bernstein. Cache-timing attacks on AES. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, 2004.

[2] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle–Damgård revisited: How to construct a hash function. In Victor Shoup, editor, *Advances in cryptology: CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14–18, 2005: proceedings*, volume 3621, pages 430–??, 2005.

[3] Ronald Cramer, editor. *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*. Springer, 2005.

[4] Ivan Damgård. A design principle for hash functions. In G. Brassard, editor, *Proc. CRYPTO 89*, pages 416–427. Springer-Verlag, 1990. Lecture Notes in Computer Science No. 435.

[5] Richard Drews Dean. *Formal aspects of mobile code security*. PhD thesis, Princeton, NJ, USA, 1999.

[6] FIPS. *Advanced Encryption Standard (AES)*, November 2001.

[7] Helena Handschuh and Howard M. Heys. A timing attack on rc5. In *SAC '98: Proceedings of the Selected Areas in Cryptography*, pages 306–318, London, UK, 1999. Springer-Verlag.

[8] Jim Randall Jean-Paul Allouche, Dan Astoorian and Jeffrey Shallit. Morphisms, squarefree strings, and the tower of hanoi puzzle. *The American Mathematical Monthly*, 101(7):651–658, Aug-Sep 1994.

[9] John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In Cramer [3], pages 474–490.

[10] Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. Provably secure fft hashing. NIST 2nd Cryptographic Hash Workshop, August 2006. Available on-line at URL `http://www.csrc.nist.gov/pki/HashWorkshop/2006/program\_2006.htm`.

[11] R. C. Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Stanford University, June 1979.

[12] K. Nyberg. Differentially uniform mappings for cryptography. In T. Helleseth, editor, *Advances in Cryptology — Eurocrypt '93*, volume 765 of *Lecture Notes in Computer Science*, pages 55–64, Berlin, 1994. Springer-Verlag.

[13] B. Preneel. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, K. U. Leuven, Leuven, Belgium, January 1993.

[14] Vincent Rijmen and Paulo S. L. M. Barreto. The WHIRLPOOL hash function. `http://paginas.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html`; `http://planeta.terra.com.br/informatica/paulobarreto/whirlpool.zip`, 2001.

[15] Ronald L. Rivest. Abelian square-free dithering for iterated hash functions. `http://people.csail.mit.edu/rivest/Rivest-AbelianSquareFreeDitheringForIteratedHashFunctions.pdf`, 2005.

[16] Claus-Peter Schnorr. Fft-hash ii, efficient cryptographic hashing. In *EUROCRYPT*, pages 45–54, 1992.

[17] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Collision search attacks on SHA1. Technical report, Shandong University, Shandong, China, 2005.

[18] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In Cramer [3], pages 19–35.

[19] A. F. Webster and S. E. Tavares. On the design of s-boxes. In *Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85*, pages 523–534, New York, NY, USA, 1986. Springer-Verlag New York, Inc.