The MD6 hash function
A proposal to NIST for SHA-3

Ronald L. Rivest
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139
rivest@mit.edu

Benjamin Agre Daniel V. Bailey Christopher Crutchfield
Yevgeniy Dodis Kermin Elliott Fleming Asif Khan
Jayant Krishnamurthy Yuncheng Lin Leo Reyzin
Emily Shen Jim Sukha Drew Sutherland
Eran Tromer Yiqun Lisa Yin

April 16, 2009

Abstract

This report describes and analyzes the MD6 hash function and is part of our submission package for MD6 as an entry in the NIST SHA-3 hash function competition[1].

Significant features of MD6 include:

- Accepts input messages of any length up to $2^{64} - 1$ bits, and produces message digests of any desired size from 1 to 512 bits, inclusive, including the SHA-3 required sizes of 224, 256, 384, and 512 bits.

- Security—MD6 is by design very conservative. We aim for provable security whenever possible; we provide reduction proofs for the security of the MD6 mode of operation, and prove that standard differential attacks against the compression function are less efficient than birthday attacks for finding collisions. We also show that when used as a MAC within NIST recommendedations, the keyed version of MD6 is not vulnerable to linear cryptanalysis. The compression function and the mode of operation are each shown to be indifferentiable from a random oracle under reasonable assumptions.

- MD6 has good efficiency: 22.4–44.1M bytes/second on a 2.4GHz Core 2 Duo laptop with 32-bit code compiled with Microsoft Visual Studio 2005 for digest sizes in the range 160–512 bits. When compiled for 64-bit operation, it runs at 61.8–120.8M bytes/second, compiled with MS VS, running on a 3.0GHz E6850 Core Duo processor.

- MD6 works extremely well for multicore and parallel processors; we have demonstrated hash rates of over 1GB/second on one 16-core system, and over 427MB/sec on an 8-core system, both for 256-bit digests. We have also demonstrated MD6 hashing rates of 375 MB/second on a typical desktop GPU (graphics processing unit) card. We also show that MD6 runs very well on special-purpose hardware.

- MD6 uses a single compression function, no matter what the desired digest size, to map input data blocks of 4096 bits to output blocks of 1024 bits—a fourfold reduction. (The number of rounds does, however, increase for larger digest sizes.) The compression function has auxiliary inputs: a "key" (K), a "number of rounds" (r), a "control word" (V), and a "unique ID" word (U).

- The standard mode of operation is tree-based: the data enters at the leaves of a 4-ary tree, and the hash value is computed at the root. See Figure 2.1. This standard mode of operation is highly parallelizable.

[1] `http://www.csrc.nist.gov/pki/HashWorkshop/index.html`

- Since the standard MD6 mode requires storage proportional to the height of the tree, there is an alternative low-storage variant mode obtained by adjusting the optional parameter L that decreases both the storage requirements and the parallelizability; setting $L = 0$ results in a Merkle-Damgård-like sequential mode of operation.

- All intermediate "chaining values" passed up the tree are 1024 bits in length; the final output value is obtained by truncating the final 1024-bit compression function output to the desired length. This "wide-pipe" design makes "internal collisions" extremely unlikely.

- MD6 automatically permits the computation of message authentication codes (MAC's), since the auxiliary 512-bit key input (K) to the compression function may be secret. The key may alternatively be set to a random value, for randomized hashing applications.

- MD6 is defined for 64-bit machines, but is very easy to implement on machines of other word sizes (e.g. 32-bit or 8-bit).

- The only data operations used are XOR, AND, and SHIFT (right and left shifts by fixed amounts); all operating on 64-bit words. There are no data-dependent table lookups or other similar data-dependent operations. In hardware, each round of the compression function can be executed in constant time–only a few gate delays.

- The compression function can be viewed as encryption with a fixed key (or equivalently, as applying a fixed random permutation of the message space) followed by truncation. The inner loop can be represented as an invertible non-linear feedback shift register (NLFSR). Security can be adjusted by adjusting the number of compression function rounds.

- Simplicity—the MD6 mode of operation and compression function are very simple: see Figure 2.1 for the mode of operation and Figure 2.10 for the compression operation (each Figure is one page).

- Flexibility—MD6 is easily adapted for applications or analysis needing non-default parameter values, such as reduced-round versions.

(Some of the detailed analyses are in our companion papers.)

Contents

Chapter 1

Introduction

A cryptographic hash function h maps an input M–a bit string of arbitrary length—to an output string $h(M)$ of some fixed bit-length d.

Cryptographic hash functions have many applications; for example, they are used in digital signatures, time-stamping methods, and file modification detection methods.

To be useful in such applications, the hash function h must not only provide fixed-length outputs, but also satisfy some (informally stated) cryptographic properties:

- **One-wayness, or Pre-image Resistance:** It should be infeasible for an adversary, given y, to compute any M such that $h(M) = y$.
- **Collision-Resistance:** It should be infeasible for an adversary to find distinct values M, M' such that $h(M) = h(M')$.
- **Second Pre-image Resistance:** It should be infeasible for an adversary, given M, to find a different value M' such that $h(M) = h(M')$.
- **Pseudo-randomness**: The function h must appear to be a "random" (but deterministic) function of its input. This property requires some care to define properly.

A history of cryptographic hash functions can be found in Menezes et al. [63, Ch. 9]; a more recent survey is provided by Preneel [80].

The purpose of this report, however, is to describe and analyze the MD6 hash function, not to survey the prior art (which is considerable).

Some readers may find it helpful to begin their introduction to MD6 by reviewing the powerpoint slides:

`http://group.csail.mit.edu/cis/md6/Rivest-TheMD6HashFunction.ppt`

from Rivest's CRYPTO'08 invited talk.

1.1 NIST SHA-3 competition

This document is part of our submission of MD6 to NIST for the SHA-3 competition [70].

We have attempted to respond to all of the requirements and requests for information given in the request for candidate SHA-3 algorithm nominations.

This report does not contain computer code implementing MD6 or other documents relevant to our submission. These can all be found in our submission package to NIST, and on our web site:

`http://groups.csail.mit.edu/cis/md6` .

Updated versions of this report, and other MD6-related materials, may also be available on the MD6 web site.

1.2 Overview

This report is organized as follows.

Chapter 2 gives a careful description of the MD6 hash function, including its compression function and mode of operation.

Chapter 3 describes the design rationale for MD6.

Chapter 4 describes efficient software implementations of MD6, including parallel implementations on multi-core processors and on graphics processing units.

Chapter 5 describes efficient hardware implementations of MD6 on FPGA's, special-purpose multi-core chips, and ASIC's.

Chapter 6 analyzes the security of the MD6 compression function.

Chapter 7 analyzes the security of the MD6 mode of operation.

Chapter 8 discusses issues of compatibility with existing standards and applications.

Chapter 9 discusses variations on the MD6 hash function; that is, how MD6 can be "re-parameterized" easily to give new hash functions in the "MD6 family".

Appendices A and B describe the constants Q and S used in the MD6 computation.

Appendix C gives some sample computations of MD6.

Appendix D summarizes our notations.

Appendix E describes the additional documents we are submitting with this proposal.

Appendix F gives information about each of the MD6 team members, including contact information.

Chapter 2

MD6 Specification

This chapter provides a detailed specification of the MD6 hash algorithm, sufficient to implement MD6. MD6 is remarkably simple, and these few pages give all the necessary details.

Before reading this chapter, the reader may wish to browse Chapter 3, which discusses some of the design decisions made in MD6.

Section 2.1 provides an overview of the notation we use; additional notation is listed in Appendix D.

Section 2.2 describes the inputs to MD6: the message to be hashed and the desired message digest length d, as well as the optional inputs: a "key" K, a 'mode control" L, and a number of rounds r.

Section 2.3 describes the MD6 output.

Section 2.4 describes MD6's "mode of operation"—how MD6 repeatedly uses a compression function to hash a long message in a tree-based manner.

Section 2.5 specifies the MD6 compression function f, which maps an input of 89 64-bit words (64 data words and 25 auxiliary information words) to an output of 16 64-bit words.

2.1 Notation

Let w denote the word size, in bits. MD6 is defined in terms of a default word size of $w = 64$ bits. However, its design supports efficient implementation using other word sizes, and variant flavors of MD6 can easily be defined in terms of other word sizes (see Chapter 9).

In this document a "word" always refers to a 64-bit (8-byte) word of $w = 64$ bits.

We let $\mathbf{W}$ denote the set $\{0,1\}^w$ of all w-bit words.

If A (or any other capital letter) denotes an array of information, then a (lower case) usually represents its length (the number of data items in A). (Our

use of **W** above is an exception.) We let both $A[i]$ and A_i denote the i-th element of A. We use 0-origin indexing. The notation $A[i..j]$ (or $A_{i..j}$) denotes the subarray of A from $A[i]$ to $A[j]$, inclusive.

MD6 is defined in a big-endian way: the high-order byte of a word is defined to be the "first" (leftmost) byte. This is as in the SHA hash functions, but different from in MD5. Big-endian is also frequently known as "network order," as Internet network protocols normally use big-endian byte ordering. We number the bytes of a word starting with byte 0 as the high-order byte, and similarly number the bits of a byte or word so that bit 0 is the most-significant bit.

(The underlying hardware may be little-endian or big-endian; this doesn't matter to us.)

Other more-or-less standard notation we may use includes:

$\oplus$**:** denotes the bitwise "XOR" operator on words.

$\wedge$**:** denotes the bitwise "AND" operator on words.

$\vee$**:** denotes the bitwise "OR" operator on words.

$\neg x$**:** denotes the bitwise negation of word x.

$x \ll b$**:** denotes x left-shifted by b bits (zeros shifting in).

$x \gg b$**:** denotes x right-shifted by b bits (zeros shifting in).

$x \lll b$**:** denotes x rotated left by b bits.

$x \ggg b$**:** denotes x rotated right by b bits.

$||$**:** denotes concatenation.

`0x...`**:** denotes a hexadecimal constant.

Additional notation can be found in Appendix D.

2.2 MD6 Inputs

This section describes the inputs to MD6. Two inputs are mandatory, while the other three inputs are optional.

M – the message to be hashed (mandatory).

d – message digest length desired, in bits (mandatory).

K – key value (optional).

L – mode control (optional).

r – number of rounds (optional).

The only mandatory inputs are the message M to be hashed and the desired message digest length d. Optional inputs have default values if any value is not supplied.

We let H denote the MD6 hash function; subscripts may be used to indicate MD6 parameters.

2.2.1 Message M to be hashed

The first mandatory input to MD6 is the message M to be hashed, which is a sequence of bits of some finite length m, where

$$0 \leq m < 2^{64} \ .$$

In accordance with the NIST requirements, the length m of the input message M is measured in bits, not bytes, even though in practice an input will typically consist of some integral number $(m/8)$ of bytes.

The length m does not need to be known before MD6 hashing can begin. The NIST API for SHA-3[1] provides the input message sequentially in an arbitrary number of pieces, each of arbitrary size, through an `Update` routine. A call to `Final` then signals that the input has ended and that the final hash value is desired.

MD6 is tree-based and highly parallelizable. If the entire message M is available initially, then a number of different processors may begin hashing operations at a variety of starting points within the message; their results may then be combined.

2.2.2 Message digest length d

The second input to MD6 is the desired bit-length d of the hash function output, where

$$0 < d \leq 512 \ .$$

The value d must be known at the beginning of the hash computation, as it not only determines the length of the final MD6 output, but also affects the MD6 computation at every intermediate operation.

Changing the value of d should result in an "entirely different" hash function—not only will the output now have a different length, but its value should appear to be unrelated to hash-values computed for the same message for other values of d.

MD6 naturally supports the digest lengths required for SHA-3: $d = 224, 256, 384$ and 512 bits, as they are within the allowable range for d.

2.2.3 Key K (optional)

Often it is desirable to work with a family $\{H_{d,K}\}$ of hash functions, indexed not only by the digest size d but also by a key K drawn from some finite set. These instances should appear to be unrelated—the behavior of $H_{d,K}$ should not have any discernible relation to that of $H_{d,K'}$, for $K \neq K'$.

The MD6 user may provide a K of *keylen* bytes, for any key length *keylen*, where

$$0 \leq keylen \leq 64 \ .$$

[1] `http://csrc.nist.gov/groups/ST/hash/sha-3/Submission_Reqs/crypto_API.html`

(It is convenient to use lower-case k to denote the maximum number 8 of 64-bit words in the key, so we use *keylen* to denote the actual number of key bytes supplied.)

There is one MD6 hash function $H_{d,K}$ for each combination of digest length d and key K. The default value for an unspecified key is key **nil** of length 0. $H_d = H_{d,\mathbf{nil}}$.

The key may be a "salt," as is commonly used for hashing passwords.

The key K may be secret. Thus, MD6 may be used directly to compute message authentication codes, or MAC's. MD6 tries to ensure that no useful information about the key leaks into MD6's output, so that the key is protected from disclosure.

The key could also be a randomly chosen value [23], for randomized hashing applications.

Within MD6, the key is padded with zero bytes until its length is exactly 64 bytes. The original length *keylen* of the key in bytes is preserved and is an auxiliary input to the MD6 compression function.

The maximum key length (64 bytes) is quite long, which allows for the key to be a concatenation of subfields used for different purposes (e.g. part for a secret key, part for a randomization value) if desired.

If the desired key is longer than 512 bits, it can first be hashed with MD6, using, for example, $d = 512$ and $K =$ **nil**; the result can then be supplied to MD6 as the key.

2.2.4 Mode control L (optional)

The standard mode of operation for MD6 is a tree-based and hierarchical, as illustrated in Figure 2.1.

Data from the message to be hashed is placed at the leaves of a sufficiently large 4-ary tree. Computation proceeds from the leaves towards the root. Each non-leaf node of the tree corresponds to one compression function execution, which takes $n = 64$ words of input and produces $c = 16$ words of output. The last d bits of the output produced at the root are taken as the hash function output.

It is straightforward to implement MD6 so that it uses an amount of storage no more than proportional to the height of the tree.

In some cases (such as with very simple RFID chips), the MD6 standard mode of operation may nonetheless require too much memory. In such cases, a variant of MD6 may be specified that uses less memory (but which is also less parallelizable).

This option is exercised with an optional "mode of operation parameter" L. By varying L, MD6 varies smoothly between a low-memory Merkle-Damgård-like sequential mode of operation ($L = 0$) and a highly-parallelizable tree-based mode of operation ($L = 64$).

The standard mode of operation has $L = 64$, for fully hierarchical operation. Actually, any value of $L \geq 27$ will give a hierarchical hash; $L = 64$ is chosen as

the default in order to represent a value "sufficiently large" that the sequential mode of operation is never invoked.

Section 2.4 gives more details on MD6's mode of operation.

2.2.5 Number of rounds r (optional)

The MD6 compression function f has a controllable number r of rounds. Roughly speaking, each round corresponds to one clock cycle in a typical hardware implementation, or 16 steps in a software implementation.

The default value of r is

$$r = 40 + \lfloor d/4 \rfloor \; ; \tag{2.1}$$

so $H_{d,K,L} = H_{d,K,L,40+\lfloor d/4 \rfloor}$. For $d = 160$, MD6 thus has a default of $r = 80$ rounds; for $d = 512$, MD6 has a default of $r = 168$ rounds. One may increase r for increased security, or decrease r for improved performance, trading off security for performance.

However, we also require that when MD6 is used in keyed mode, that $r \geq 80$. This provides protection for the key, even when the desired output is short (as it might be for a MAC). Thus, when MD6 has a nonempty key, the default value of r is

$$r = \max(80, 40 + \lfloor d/4 \rfloor) \; . \tag{2.2}$$

The round parameter r is exposed in the MD6 API, so it may be explicitly varied by the user. This is done since reduced-round versions of MD6 may be of interest for security analysis, or for applications with tight timing constraints and reduced security requirements. Or, one could increase r above the default to accommodate various levels of paranoia. Also, if there is a key, but it is non-secret, then fewer than 80 rounds could be specified if desired.

Arguably, the current need to consider a new hash function standard might have seemed unnecessary if the API for the prior standards had included a variable number of rounds.

2.2.6 Other MD6 parameters

There are other parameters to the MD6 hash function that could also be varied, at least in principle (e.g. w, Q, c, $t_0 \ldots t_5$, r_i, ℓ_i, S_i for $0 \leq i < rc$). For the purpose of defining what "MD6" means, these quantities should all be considered fixed with default values as described herein. But variant MD6 hash functions that use other settings for these parameters could be considered and studied. See Chapter 9 for a description of how these parameters might be varied.

2.2.7 Naming versions of MD6

We suggest the following approach for naming various versions of MD6.

In the simplest case, we only need to specify the digest size: MD6-d specifies the version of MD6 having digest size d. This version also has the zero-length

key **nil**, $L = 64$ (i.e. fully hierarchical operation), and a number r of rounds that is the default for that digest size. These are the MD6 versions most relevant for SHA-3:

```
MD6-224
MD6-256
MD6-384
MD6-512.
```

Some of our experiments also consider MD6-160, as it is comparable to SHA-1.

Software implementations of MD6 typically use the lower-case version of the name MD6, as in "`md6sum`".

Naming non-standard variants of MD6 is discussed in Section 9.1.

The CRYPTO 2008 invited talk by Rivest also called MD6 the "pumpkin hash", noting that the due date for SHA-3 submissions is Halloween 2008. One could thus also label the MD6 variants as PH-256, etc. ...

2.3 MD6 Output

The output of MD6 is a bit string D of exactly d bits in length:

$$D = H_{d,K,L,r}(M) \ ;$$

D is the hash value of input message M. It is also often called a "message digest." The "MD" in the name "MD6" reflects this terminology.

In some contexts, the MD6 output may be defined to include other parameters. For example, with digital signatures, a hash function needs to be applied once by the sender, and once again by the recipient, to the same message. These computations should yield the same result. For this to work, the recipient needs to know not only the message M and the message digest length d, but also the values of any of the parameters K, L, and r that have non-default values. In such applications, these parameters (other than K) could be considered as part of the hash function output. At least, they need to be communicated to the receiver along with the hash function value D, communicated in some other way from sender to receiver, or agreed in advance to have some particular non-default settings.

2.4 MD6 Mode of Operation

A hash function is typically constructed from a "compression function", which maps fixed-length inputs to (shorter) fixed length outputs. A "mode of operation" then specifies how the compression function can be used repeatedly to enable hashing inputs of arbitrary nonnegative length to produce a fixed-length output.

To describe a hash function, one thus needs to describe:

- its mode of operation,
- its compression function, and
- various constants used in the computation.

The MD6 compression function f takes inputs of a fixed length ($n = 89$ words), and produces outputs of a fixed but shorter length ($c = 16$ words):

$$f : \mathbf{W}^{89} \longrightarrow \mathbf{W}^{16} \ .$$

(Recall that $\mathbf{W}$ is the set of all binary words of length $w = 64$ bits.)

For convenience, we call a c-word block a "chunk". The chaining variables produced by MD6 are all "chunks".

The 89-word input to the compression function f contains a 15-word constant Q, an 8-word key K, a "unique ID" word U, a "control word V", and a 64-word data block B.

Since Q is constant, the "effective" or "reduced" MD6 compression function f_Q maps 74-word inputs to 16-word outputs:

$$f_Q : \mathbf{W}^{74} \longrightarrow \mathbf{W}^{16}.$$

via the relationship:

$$f_Q(x) = f(Q||x) \ .$$

Thus, the MD6 compression function achieves a fourfold reduction in size from data block to output—four chunks fit exactly into one data block, and one chunk is output.

The next section describes the MD6 mode of operation; Section 2.5 then describes the MD6 compression function.

2.4.1 A hierarchical mode of operation

The standard mode of operation for MD6 is tree-based. See Figure 2.1. An implementation of this hierarchical mode requires storage at least proportional to the height of the tree.

Since some very small devices may not have sufficient storage available, MD6 provides a height-limiting parameter L. When the height reaches $L + 1$, MD6 switches from the parallel compression operator PAR to the sequential compression operator SEQ.

The MD6 mode of operation is thus optionally parameterized by the integer L, $0 \leq L \leq 64$, which allows a smooth transition from the default tree-based hierarchical mode of operation (for large L) down to an iterative mode of operation (for $L = 0$). When $L = 0$, MD6 works in a manner similar to that of the well-known Merkle-Damgård method [65, 64, 34] construction or to the HAIFA method [18]. See Figure 2.2.

In our description of the MD6 mode of operation, MD6 makes up to L "parallel" passes over the data, each one reducing the size of the data by a

factor of four, and then performs (if necessary) a single sequential pass to finish up.

Since the input size must be less than 2^{64} bits and the final compression function produces an output of $2^{10} = 1024$ bits (before final truncation to d bits), there will be at most 27 such parallel passes (since $27 = \log_4(2^{64}/2^{10})$. The default value $L = 64$, since it is greater than 27, ensures that by default MD6 will be full hierarchical.

Graphically, MD6 creates a sequence of 4-ary trees of height at most L, each containing 4^L leaf chunks (of $c = 16$ words each), then combines the values produced at their roots (if there is more than one) in a sequential Merkle-Damgård-like manner.

If 4^L is larger than the number of 16-word chunks in the input message, then only one tree is created, and MD6 becomes a purely tree-based method.

On the other hand, if $L = 0$, then no trees are created, and the input is divided into 48-word (three-chunk) data-blocks to be combined in a sequential Merkle-Damgård-like manner. (There are now only three data chunks in a data block, since one chunk is the chaining variable from the previous compression operation at the node immediately to the left.)

For intermediate values of L, we trade off tree height (and thus minimum memory requirements) for opportunities for parallelism (and thus perhaps greater speed).

Figure 2.4 gives the top-level procedure for the MD6 mode of operation, which is described in a bottom-up, level by level manner.

First, all of the compression operations on level $\ell = 1$ are performed. Then all of the compression operations on level $\ell = 2$ are performed, etc.

MD6 is described in this manner for maximum clarity. A practical implementation may be organized somewhat differently (but, of course, in a way that computes the same function). For example, operations at different levels may be intermixed, with a compression operation being performed as soon as its inputs are available. See Chapter 4 for some discussion of implementation issues.

Each such compression operation is by default performed with the PAR operation, described in Figure 2.5. The PAR operation may be implemented in parallel (hence its name). Given the data on level $\ell - 1$, it produces the data on level ℓ, which will be only one-fourth as large. This is repeated until a level is reached where the remaining data is only 16 words long. This data is truncated to become the final hash output.

Figure 2.6 describes SEQ—it is very similar to Merkle-Damgård in operation. It works sequentially through the input data on the last level and produces the final hash output.

2.4.2 Compression function input

MD6's mode of operation formats the input to the compression function f in the following way. There are $n = 89$ words, formatted as follows with the default sizes. See Figure 2.7. The first four items Q, K, U, V, are "auxiliary inputs", while the last item B is the data payload.

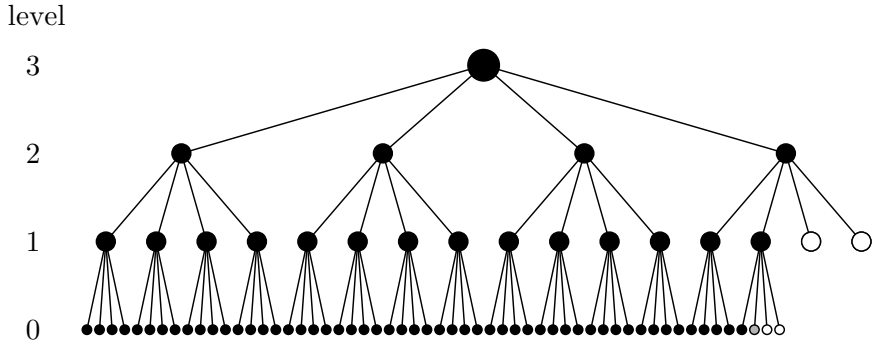

Figure 2.1: Structure of the standard MD6 mode of operation ($L = 64$). Computation proceeds from bottom to top: the input is on level 0, and the final hash value is output from the root of the tree. Each edge between two nodes represents a 16 word (128 byte or 1024-bit) chunk. Each small black dot on level 0 corresponds to a 16-word chunk of input message. The grey dot on level 0 corresponds to a last partial chunk (less than 16 words) that is padded with zeros until it is 16 words long. A white dot (on any level) corresponds to a padding chunk of all zeros. Each medium or large black dot above level zero corresponds to an application of the compression function. The large black dot represents the final compression operation; here it is at the root. The final MD6 hash value is obtained by truncating the value computed there.

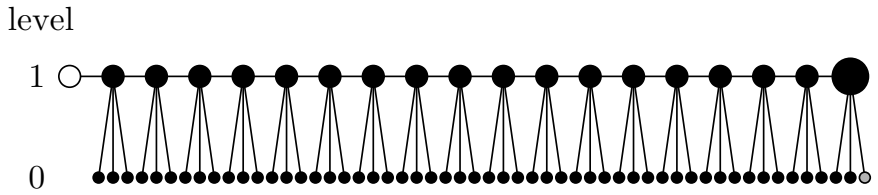

Figure 2.2: Structure of the MD6 sequential mode of operation ($L = 0$). Computation proceeds from left to right only; level 1 represents processing by SEQ. The hash function output is produced by the rightmost node on level 1. This is similar to standard Merkle-Damgård processing. The white circle at the left on level 1 is the 1024-bit all-zero initialization vector for the sequential computation at that level. Each node has four 1024-bit inputs: one from the left, and three from below; the effective "message block size" is thus 384 bytes, since 128 bytes of the 512-byte compression function input are used for the chaining variable.

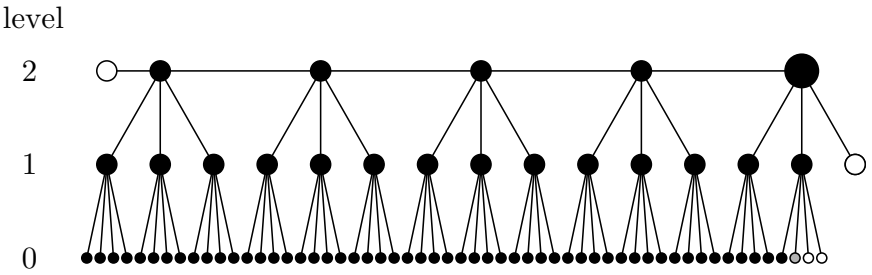

Figure 2.3: Structure of the MD6 mode of operation with an intermediate mode of operation value ($L = 1$). Computation proceeds from bottom to top and left to right; level 2 represents processing by SEQ. The hash function output is produced by the rightmost node on level 2. The white circle at the left on level 2 is the all-zero initialization vector for the sequential computation at that level.

Q – a constant vector (giving an approximation to the fractional part of the square root of 6) of length $q = 15$ words. See Appendix A.

K – a "key" (serving as salt, tag, tweak, secret key, etc.) of length $k = 8$ words containing a supplied key of *keylen* bytes.

U – a one-word "unique node ID".

V – a one-word "control word".

B – a data block of length $b = 64$ words.

2.4.2.1 Unique Node ID U

The "unique node ID" U is a one-word compression function auxiliary input (ℓ, i). It uniquely specifies the particular compression function operation being performed, by giving both the level number for this operation and its index within the level. See Figure 2.9.

- ℓ — one byte giving the level number.
- i — seven bytes giving the position within the level, with the leftmost (first) compression function operation labelled as $i = 0$.

For example, the very first compression operation performed always has $U = (\ell, i) = (1, 0)$. A node (ℓ, i) at level ℓ, $1 < \ell \leq L$ (produced by PAR) has children at $(\ell - 1, 4i)$, $(\ell - 1, 4i + 1)$,$(\ell - 1, 4i + 2)$, and $(\ell - 1, 4i + 3)$. A node (ℓ, i) at level $1 < \ell = L + 1$ (produced by SEQ) has children at $(\ell - 1, 3i)$, $(\ell - 1, 3i + 1)$, and $(\ell - 1, 3i + 2)$.

2.4.2.2 Control Word V

The "control word" V is a one-word compression function auxiliary input that gives parameters relevant to the computation. See Figure 2.8.

The MD6 Mode of Operation

Input:

M: A message M of some non-negative length m in bits.

d: The length d (in bits) of the desired hash output, $1 \leq d \leq 512$.

K: An arbitrary $k = 8$ word "key" value, containing a supplied key of *keylen* bytes padded on the right with $(64 - keylen)$ zero bytes.

L : A non-negative mode parameter (maximum level number, or number of parallel passes).

r : A non-negative number of rounds.

Output:

D: A d-bit hash value $D = H_{d,K,L,r}(M)$.

Procedure:

Initialize:

- Let $\ell = 0$, $M_0 = M$, and $m_0 = m$.

Main level-by-level loop:

- Let $\ell = \ell + 1$.
- If $\ell = L+1$, return $\mathrm{SEQ}(M_{\ell-1}, d, K, L, r)$ as the hash function output.
- Let $M_\ell = \mathrm{PAR}(M_{\ell-1}, d, K, L, r, \ell)$. Let m_ℓ denote the length of M_ℓ in bits.
- If $m_\ell = cw$ (i.e. if M_ℓ is c words long), return the last d bits of M_ℓ as the hash function output. Otherwise, return to the top of the main level-by-level loop.

Figure 2.4: The MD6 Mode of Operation. With the default setting of $L = 64$, the SEQ operation is never used; the PAR operation is repeatedly called to reduce the input size by a factor of $b/c = 64/16 = 4$ until a single 16-word chunk remains. On the other hand, setting L equal to 0 yields an entirely sequential mode of MD6 operation.

The MD6 PAR Operation

Input:

$M_{\ell-1}$**:** A message of some non-negative length $m_{\ell-1}$ in bits.

d**:** The length d (in bits) of the desired hash output, $1 \leq d \leq 512$.

K**:** An arbitrary $k = 8$ word "key" value, containing a supplied key of *keylen* bytes.

L : A non-negative mode parameter (maximum level number, or number of parallel passes).

r : A non-negative number of rounds.

ℓ : A non-negative integer level number, $1 \leq \ell \leq L$.

Output:

M_ℓ**:** A message of length m_ℓ bits, where $m_\ell = 1024 \cdot \max(1, \lceil m_{\ell-1}/4096 \rceil)$

Procedure:

Initialize:

- Let Q denote the array of length $q = 15$ words giving the fractional part of $\sqrt{6}$. (See Appendix A.)
- Let f denote the MD6 compression function mapping 89 words of input (including a 64-word data block B) to a 16-word output chunk C using r rounds of computation.

Shrink:

- Extend input $M_{\ell-1}$ if necessary (and only if necessary) by appending zero bits until its length becomes a positive integral multiple of $b = 64$ words. Then $M_{\ell-1}$ can be viewed as a sequence B_0, B_1, ..., B_{j-1} of b-word blocks, where $j = \max(1, \lceil m_{\ell-1}/bw \rceil)$.
- For each b-word block B_i, $i = 0, 1, \ldots, j-1$, compute C_i in parallel as follows:
  - Let p denote the number of padding bits in B_i; $0 \leq p \leq 4096$. (p can only be nonzero for $i = j - 1$.)
  - Let $z = 1$ if $j = 1$, otherwise let $z = 0$. ($z = 1$ only for the last block to be compressed in the complete MD6 computation.)
  - Let V be the one-word value $r||L||z||p||keylen||d$ (see Figure 2.8).
  - Let $U = \ell * 2^{56} + i$ be a "unique node ID"—a one-word value unique to this compression function operation.
  - Let $C_i = f(Q||K||U||V||B_i)$. (C_i has length $c = 16$ words).
- Return $M_\ell = C_0||C_1||\ldots||C_{j-1}$.

Figure 2.5: The MD6 PAR operator is a parallel compression operation producing level ℓ of the tree from level $\ell - 1$. With the default setting $L = 64$, this routine is used repeatedly at each layer of the tree to generate the next higher layer, until the value at the root is produced.

The (Optional) MD6 SEQ Operation

Input:

M_L**:** A message of some non-negative length m_L in bits.

d**:** The length d (in bits) of the desired hash output, $1 \leq d \leq 512$.

K**:** An arbitrary $k = 8$ word key value, containing a supplied key of *keylen* bytes.

L : A non-negative mode parameter (maximum tree height).

r : A non-negative number of rounds.

Output:

D**:** A d-bit hash value.

Procedure:

Initialize:

- Let Q denote the array of length $q = 15$ words giving the fractional part of $\sqrt{6}$. (See Appendix A.)
- Let f denote the MD6 compression function mapping an 89-word input (including a 64-word data block B) to a 16-word output block C using r rounds of computation.

Main loop:

- Let C_{-1} be the zero vector of length $c = 16$ words. (This is the "IV".)
- Extend input M_L if necessary (and only if necessary) by appending zero bits until its length becomes a positive integral multiple of $(b-c) = 48$ words. Then M_L can be viewed as a sequence B_0, B_1, ..., B_{j-1} of $(b-c)$-word blocks, where $j = \max(1, \lceil m_L/(b-c)w \rceil)$.
- For each $(b-c)$-word block B_i, $i = 0, 1, \ldots, j-1$ in sequence, compute C_i as follows:
  - Let p be the number of padding bits in B_i; $0 \leq p \leq 3072$. (p can only be nonzero when $i = j-1$.)
  - Let $z = 1$ if $i = j-1$, otherwise let $z = 0$. ($z = 1$ only for the last block to be compressed in the complete MD6 computation.)
  - Let V be the one-word value $r||L||z||p||keylen||d$ (see Figure 2.8).
  - Let $U = (L+1) \cdot 2^{56} + i$ be a "unique node ID"—a one-word value unique to this compression function operation.
  - Let $C_i = f(Q||K||U||V||C_{i-1}||B_i)$. ($C_i$ has length $c = 16$ words).
- Return the last d bits of C_{j-1} as the hash function output.

Figure 2.6: The MD6 SEQ Operator is a sequential Merkle-Damgård-like hash operation producing a final hash output value. With the default setting of $L = 64$, SEQ is never used.

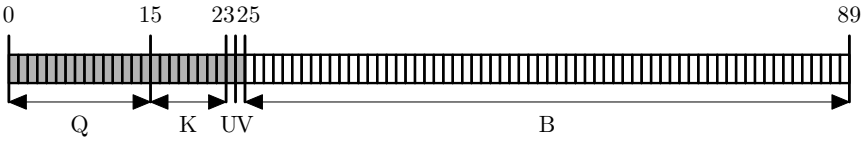

Figure 2.7: The compression function input contains 89 64-bit words: a 15-word constant vector Q, an 8-word key K, a one-word unique node ID U, a one-word control variable V, and a 64-word data block B. The first four items form the auxiliary information, shown in grey.

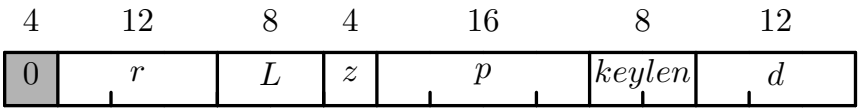

Figure 2.8: Layout of the control word V. The high-order 4 bits are zero (reserved for future use). The size (in bits) of each field is given above the field.

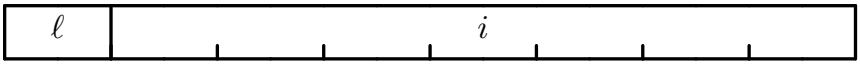

Figure 2.9: Layout of the unique node ID word U. The high-order byte is ℓ, the level number. The seven-byte field i gives the index of the node within the level ($i = 0, 1, \ldots$).

- r — number of rounds in the compression function (12 bits).
- L — mode parameter (maximum level) (8 bits).
- z — set to 1 if this is the final compression operation, otherwise 0 (4 bits).

  In Figures 2.1, 2.3, 2.2, these final operations are indicated by the very large black circles.
- p — the number of padding data bits (appended zero bits) in the current input block B (16 bits).
- *keylen* — the original length (in bytes) of the supplied key K (8 bits).
- d — the desired length in bits of the digest output (12 bits).

2.5 MD6 Compression Function

This section describes the operation of the MD6 compression function.

The compression function f takes as input an array N of length $n = 89$ words. It outputs an array C of length $c = 16$ words.

Here f is described as having a single 89-word input N, although it may also be viewed as having a 25-word "auxiliary" input $(Q||K||U||V)$ followed by a 64-word "data" input block B.

The compression function f is computed as shown in Figure 2.10.

The compression function may be viewed as consisting of an encryption of N with a fixed arbitrary key S (which is not secret), followed by a truncation operation that returns only the last 16 words of $E_S(N)$. See Figure 2.11. Here S determines the "round constants" of the encryption algorithm.

Internally, the compression function has a main loop of r rounds (each consisting of $c = 16$ steps), followed by the truncation operation that truncates the final result to $c = 16$ words.

The main loop thus performs a total of $t = rc$ steps, where each step computes a one-word value. This loop can be implemented by loading the input into the first n words of an array A of length $n + t$, then computing each of the remaining t words in turn. See Figure 2.12. Equivalently, this loop can be implemented as a nonlinear feedback shift register with 89 words of state. See Figure 2.14.

The truncation operation merely returns the last 16 words of A as the compression function output (which we also call the "chaining variable"). The compression function always outputs $c = 16$ words (1024 bits) for this chaining variable. This is at least twice as large as any possible MD6 hash function output, in line with the "wide-pipe" strategy suggested by Lucks [55].

The compression function takes the "feedback tap positions" t_0, t_1, t_2, t_3, t_4, each in the range 1 to $n-1 = 88$, as parameters. (Note the slight overloading for the symbol t: when subscripted, it refers to a tap position; when unsubscripted, it refers to the number of computation steps $t = rc$.)

The MD6 compression function f

Input:

N: An input array $N[0..n-1]$ of $n = 89$ words. (This typically consists of a 25-word "auxiliary information" block, followed by a a $b = 64$-word data block B, but all inputs are treated uniformly here.)

r : A non-negative number of rounds.

Output:

C: An output array $C[0..c-1]$ of length $c = 16$ words.

Parameters: c,r,t_0,t_1,t_2,t_3,t_4,r_i,ℓ_i,S_i, where $1 \leq t_i \leq n$ for all $0 \leq i \leq 4$, $0 < r_i, \ell_i \leq w/2$ for all i, and S_i is a w-bit word for each i, $0 \leq i < t$.

Procedure:

Let $t = rc$. (Each round has $c = 16$ steps.)

Let $A[0..t+n-1]$ be an array of $t+n$ words.

Initialization:

$A[0..n-1]$ is initialized with the input $N[0..n-1]$.

Main computation loop:

for $i = n$ **to** $t+n-1$: /* t steps */

$$
\begin{array}{lcll}
x & = & S_{i-n} \oplus A_{i-n} \oplus A_{i-t_0} & \text{[line 1]} \\
x & = & x \oplus (A_{i-t_1} \wedge A_{i-t_2}) \oplus (A_{i-t_3} \wedge A_{i-t_4}) & \text{[line 2]} \\
x & = & x \oplus (x \gg r_{i-n}) & \text{[line 3]} \\
A_i & = & x \oplus (x \ll \ell_{i-n}) & \text{[line 4]}
\end{array}
$$

Truncation and Output:

Output $A[t+n-c..t+n-1]$ as the output array $C[0..c-1]$ of length $c = 16$.

Figure 2.10: The MD6 Compression Function f.

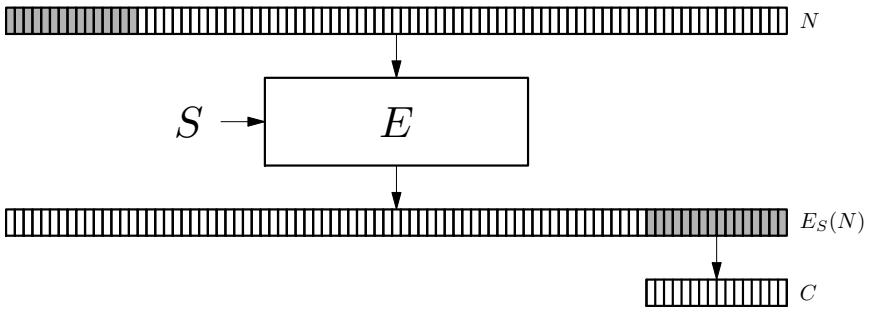

Figure 2.11: The compression function f viewed as an encryption operation followed by a truncation operation. The 89-word input N (with the first 15 words shaded to indicate that they are constant) is encrypted under control of key S to yield $E_S(N)$. The key S is arbitrary but fixed and public. The last 16 words of $E_S(N)$ form the desired compression function output C.

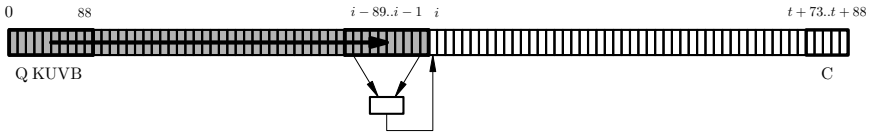

Figure 2.12: The compression function main loop. The array portion $A[0..88]$ is initialized with the compression function input (Q, K, U, V, B) (See Figure 2.7). Then, for $i = 89, 90, \ldots, t + 88$, $A[i]$ is computed as a feedback function of $A[i-89..i-1]$. The portion $A[i-89..i-1]$ can be viewed as a "sliding window" moving left to right as the computed portion (shown in gray) grows. The last 16 words of final window $A[t..t+88]$, that is $A[t+73..t+88]$, forms the output C.

MD6 constants

Tap positions:

t_0	t_1	t_2	t_3	t_4
17	18	21	31	67

Shift amounts:
The values r_{i-n}, ℓ_{i-n} are the initial right-shift amount and the subsequent left-shift amount within the step with index i; the index $i-n$ is taken modulo $c = 16$.

$(i-n) \bmod 16$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
r_{i-n}	10	5	13	10	11	12	2	7	14	15	7	13	11	7	6	12
ℓ_{i-n}	11	24	9	16	15	9	27	15	6	2	29	8	15	5	31	9

Round constants:
The values S_{i-n} for $i \geq n$ are determined as follows. Because S_{i-n} is constant within a round, but changes from round to round, S_{i-n} is defined in terms of an auxiliary sequence S'_j. The constants S'_0 and S^* determine the entire sequence.

$$S_{i-n} = S'_{\lfloor (i-n)/16 \rfloor}$$

$$S'_0 = \texttt{0x0123456789abcdef}$$

$$S^* = \texttt{0x7311c2812425cfa0}$$

$$S'_{j+1} = (S'_j \lll 1) \ \oplus \ (S'_j \wedge S^*) \ .$$

Figure 2.13: MD6 Constants

In addition, for each i, $0 \leq i < t$, there are parameters r_i, ℓ_i denoting fixed "right-shift" and "left-shift" amounts: these are different nonzero values $0 < r_i, \ell_i \leq w/2 = 32$. Finally, there is a one-word "feedback constant" value S_i for each i, $0 \leq i < t$.

Lines 1–2 of the loop in Figure 2.10 are entirely bitwise parallel, with no mixing between different bit positions. Line 1 contains the constant term S_i and the linear terms A_{i-n} (the "end-around" term) and A_{i-t_0}. Line 2 contains the nonlinear "quadratic terms" $(A_{i-t_1} \wedge A_{i-t_2})$ and $(A_{i-t_3} \wedge A_{i-t_4})$.

Note that no information is lost during the compression function computation until the final truncation to an output of size c words. Indeed, the basic compression step is invertible, because the recurrence exclusive-ors $A[i-n]$ with values depending only on the other input elements $A[i-1..i-n+1]$, and because

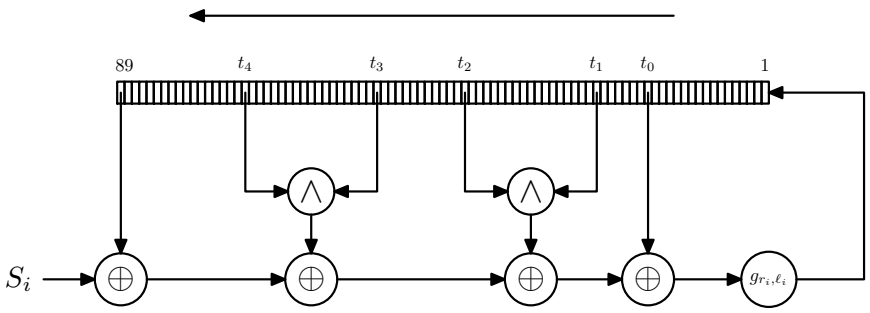

Figure 2.14: The main computation loop of the compression function viewed as a nonlinear feedback shift register. Each shift register cell, each wire, and each S_i represents a 64-bit word. Because the shift register shifts from right to left, it is convenient to number the cells from 1 on the right to 89 on the left; the cell labelled 1 in the figure corresponds to location $A[i-1]$ in Figure 2.10, and the cell labelled 89 here corresponds to location $A[i-89]$. The tap positions shown are correct for $t_0 = 17$ and for position $n = 89$, but for $t_1 \ldots t_4$ are only illustrative (for ease of drawing). The function g_{r_i,ℓ_i} is the function described in equation 2.3. Each 64-bit feedback word depends on one S_i word and six words of the register.

xoring a value with a shifted version of itself is invertible. (Xoring a value with a shift of itself is known as the "xorshift" operator; it is well-known and studied in the random-number literature [56, 75].) Thus, the mapping inside f of the first n-words of A to the last n words of A is invertible; this corresponds to the mapping π given in Figure 2.15.

In hardware, the basic iteration step can be implemented with constant circuit depth—there are no multiplications, additions, or other operators that require carry propagation or other potentially non-constant delays. This means a hardware implementation can be extremely fast. Note that since $c \leq \min(t_0, t_1, t_2, t_3, t_4)$, there is no dependence of one step on the output of another for at least c steps. Since the steps within a round are independent, all steps within a round can be executed simultaneously. Overall, each round requires only a single clock tick, and a complete compression function can be computed in r clock cycles.

There are no data-dependent table lookups or other operations whose timing might be data-dependent. This reduces the vulnerability to side-channel attacks.

2.5.1 Steps, rounds and rotations

The $t = rc$ steps can be thought of as a sequence of rotations, each of which consists of $n = 89$ steps and thereby fully cycles/refills the n-word feedback shift register by computing the next n words of state. We note that one rotation consists of $n/c = 89/16 = 5\ 9/16$ rounds, and two rotations correspond to nearly 11 rounds.

Each rotation produces a new n-word vector; you can view the computation as rc/n rotations, each of which produces a n-word state vector invertibly from the previous n-word state vector. The input is the first n-word state vector; the output is the last c words of the last n-word state vector. (Note that rc/n isn't necessarily an integer. That doesn't matter here; the description just given is still accurate.)

The compression function can also be viewed as r 16-step rounds each of which computes the next 16 words of state.

We prefer the latter view.

The MD6 feedback constants are organized in a manner that reflects the round-oriented viewpoint. The round constant S_i stays the same for all 16 steps of a round and changes for the next round. The shift amounts $r(\cdot)$ and $\ell(\cdot)$ vary within a round, but then have the same pattern of variation within successive rounds.

For a software implementation of MD6, it is convenient to perform 16-fold loop-unrolling, so that each round is implemented as a branch-free basic block of code.

2.5.2 Intra-word Diffusion via xorshifts

Some method is required to effect diffusion between the various bit positions within a word. This intra-word diffusion is provided by the g_{r_i,ℓ_i} operator implicit in lines 3–4 of the loop in Figure 2.10:

$$\begin{array}{rcl} g_{r_i,\ell_i}(x) & = & \{\ y = x \ \oplus \ (x \gg r_i); \\ & & \quad \textbf{return}\ y \ \oplus \ (y \ll \ell_i) \\ & & \} \end{array} \tag{2.3}$$

(Here "ℓ" and r are overloaded; ℓ_i and r_i refer to a shift amounts, while r refers to the number of rounds, and ℓ is used in the mode of operation to refer to the level number; these distinctions should be clear from context.)

The function g is linear and invertible (lossless).

The operators $x = x \ \oplus \ (x \gg r_i)$ and $x = x \ \oplus \ (x \ll \ell_i)$ are known as "xorshift" operators; their properties have been studied in [56, 75].

2.5.3 Shift amounts

The shift values are indexed mod $c = 16$ (i.e., they repeat every round).

The values given here for MD6 were the result of extensive experiments using one million randomly generated tables of such shift values. See Section 3.9.3

2.5.4 Round Constants

The round constants S'_j provide some variability between round. Each round j has its own constant S'_j; the steps within a round all use the same round constant.

The following recurrence generates these constants:

$$\begin{aligned} S'_0 &= \texttt{0x0123456789abcdef} \\ S^* &= \texttt{0x7311c2812425cfa0} \\ S'_{j+1} &= (S'_j \lll 1) \oplus \\ &\quad (S'_j \wedge S^*) \ . \end{aligned} \tag{2.4}$$

This recurrence relation is one-to-one—from S'_j one can determine S'_{j+1}, and vice versa. (See Schnorr et al. [88, Lemma 5])

2.5.5 Alternative representations of the compression function

There is another useful way of representing the compression function. Since the 15-word portion Q of the input is a fixed constant, we can define the function f_Q as

$$f_Q(x) = f(Q||x) \ ,$$

where f_Q maps inputs of length $n - q = 74$ words to outputs of length $c = 16$ words. This representation of f is useful in our security proofs of the compression function in Section 6.1.2. We may refer to either f or f_Q as "the MD6 compression function"; they are intimately related. It is easier to talk about f when discussing the specification of MD6, but better to talk about f_Q when discussing its security. We may refer to f as the "full" compression function, and f_Q as the "reduced" compresssion function.

Another relevant representation is defined as follows. If we return to the representation of Figure 2.11, wherein f is defined in terms of an encryption operation, we note that S (and thus $E_S(\cdot)$) is public and fixed. It may thus be preferable to view the compression function as a fixed but randomly chosen public permutation π of the input space $\mathbf{W}^n$, followed by truncation to c words. See Figure 2.15. We make use of this representation in our discussion of compression function security.

2.6 Summary

This chapter has presented the full specification of the MD6 hash function. Reference code provided with our submission to NIST for the SHA-3 competition provides a different, but equivalent, definition.

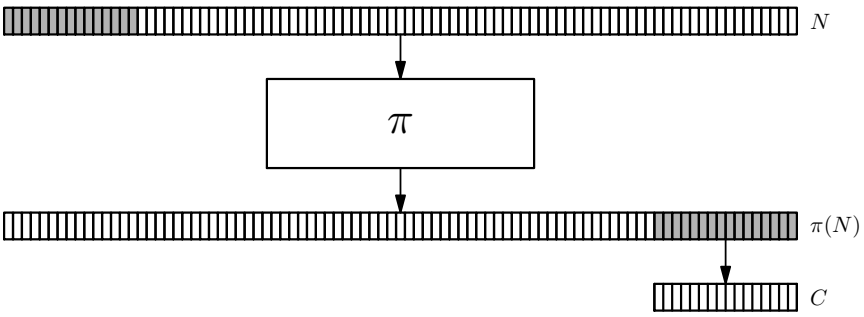

Figure 2.15: An alternative view: the compression function f viewed as a fixed public random permutation π of the input space $\mathbf{W}^n$ followed by a truncation operation. The 89-word input N (with the first 15 words shaded to indicate that they are constant) is mapped by π to a random value $\pi(N)$. The last 16 words of $\pi(N)$ form the desired compression function output C.

Chapter 3

Design Rationale

This chapter describes the considerations and reasoning behind many of the design decisions in MD6.

The landscape for designing hash functions has certainly changed in the last two decades (since MD5 was designed)! Advances in technology have provided fewer constraints, and more opportunities. Theoretical advances provide newer tools for the both the attacker and the designer.

The sections in this chapter summarize some of the considerations that went in to the design choices for MD6. A related presentation of these issues is available in Rivest's CRYPTO'08 slides (included with this submission).

Of course, the primary objective of a cryptographic hash function is security—meeting the stated cryptographic objectives. And, to the extent possible, doing so in a provable way. Chapters 6 and 7 provide detailed proofs regarding the security properties of the MD6 compression function and mode of operation.

Some of the other significant considerations include:

- Increased availability of memory, allowing larger block sizes that include new auxiliary inputs to each compression function call.
- The forthcoming flood of multicore CPU's, some of which may contain hundreds of cores. This consideration demands an approach that can exploit available parallelism.
- Side-channel attacks that limit the instructions that one deems "safe" in an application potentially utilizing secret keys.

The following sections document these and related considerations.

3.1 Compression function inputs

It is worthwhile to begin by revisiting the description of the inputs to a compression function and associated terminology.

3.1.1 Main inputs: message and chaining variable

Traditionally, compression functions have two main input ports: one for a chaining variable and one for a message block.

Such a traditional compression function can be characterized as a mapping from a c-word input chaining variable and a b-word input message block to a c-word output chaining variable:

$$f : \mathbf{W}^c \times \mathbf{W}^b \rightarrow \mathbf{W}^c \tag{3.1}$$

Here $\mathbf{W}$ denotes the set of w-bit words. The compression function produces a single chaining variable of length c words as the output.

Such traditional compression functions are "hard-wired" to fit within a certain mode of operation. More precisely, these input ports are dedicated to their particular use within the Merkle-Damgård mode of operation.

The compression function signature (3.1) may also relate to the manner in which the compression function is constructed. For example, the chaining variable may be used as input to an encryption algorithm, as in the Davies-Meyer construction [63, Ch. 9].

To move beyond the Merkle-Damgård paradigm, we need to take a somewhat more general and flexible view of compression, along the lines of a general pseudo-random function with fixed input and output sizes.

Note, for example, in a tree-based hash algorithm like MD6, compression functions working on the leaves of the tree have only message data as input, while compression functions higher in the tree contain no message data, but instead consists of several chaining variables passed up from their children nodes. It thus doesn't make sense to have a compression function with dedicated "message" and "chaining variable" inputs.

We thus consider for the moment revising our compression function signature by dropping the specific c-word "chaining variable input", and considering the compression function to be a mapping from b words of main input to c words of output:

$$f_B : \mathbf{W}^b \rightarrow \mathbf{W}^c \ . \tag{3.2}$$

(The subscript B here indicates that we are only considering the inputs corresponding to the B portion–the data payload.) We'll see in a moment that we'll revise this signature again, slightly.

Whether the bw input bits come from the message, or are chaining variables from previous compression function operations, will depend on the details of the mode of operation. We still call the output of a compression function computation a "chaining variable," even though the computations may be linked in a tree-like or other manner, rather than as a chain as in Merkle-Damgård.

The compression ratio of a compression function is the ratio b/c of the number of main input bits to the number of output bits. Ratios in the range 3–5 are typical; MD6 has a compression ratio of 4. For MD6 the $b = 4c$ main input words may consist of either message data (as for a leaf node) or four chaining variables (from child nodes) of size c words each.

3.1.2 Auxiliary inputs: key, unique nodeID, control word

A compression function may have "auxiliary" inputs in addition to the "main" inputs described above.

A traditional compression function doesn't have any such auxiliary inputs, but recently there have been a number of proposals (such as [18, 84]) to include such inputs. Such auxiliary inputs can have great value in defeating or reducing vulnerabilities associated with the standard Merkle-Damgård mode of operation. They also facilitate proofs of security, as one can often treat restrictions of the compression function to different auxiliary inputs as, essentially, independent-drawn random functions.

MD6 makes liberal use of such auxiliary inputs, which for MD6 are:

- a k-word "key" K,
- a u-word "unique node ID" U, and
- a v-word "control word" V.

If the total length of such auxiliary inputs is $\alpha = k + u + v$ words, then the compression function has the signature:

$$f_Q : \mathbf{W}^\alpha \times \mathbf{W}^b \to \mathbf{W}^c \ . \tag{3.3}$$

(The subscript Q here indicates that we are considering all inputs except the constant value Q; the actual implementation of the MD6 compression function also incorporates a fixed constant "input" value Q, but we do not need to discuss Q further here, as it is constant.)

3.2 Provable security

To the extent possible while maintaining good efficiency, MD6 is based on provable security.

We provide numerous reductions demonstrating that the MD6 mode of operation achieves various security goals, assuming that the MD6 compression function satisfies certain properties.

We also provide strong evidence that the MD6 compression function has the desired properties. This includes not only statistical analysis and SAT-solver attacks, but also provable lower bounds on the workload required by certain differential attacks.

The tree-based MD6 mode of operation is also interesting in that certain of its proofs admit a tighter security reduction than the corresponding known results for the Merkle-Damgård mode of operation.

Our results on the mode of operation security relate to pre-image resistance, collision-resistance, pseudorandomness, and indifferentiability from a random oracle.

3.3 Memory usage is less of a constraint

Continuing improvements in integrated circuit and memory technology have made memory usage much less of a concern in hash function design. While still a concern for low-end devices, the situation is overall much improved, and one can reasonably propose hash functions with significantly larger memory usage than for designs proposed in the 1990's. Doing so can provide significant benefits in flexibility and security.

Indeed, MD5 was designed in 1991; 17 years have since passed. Since then, the memory capacity of chips has increased at about 60% per year, so such chips now have over 1000 times as much capacity. Similarly, the cost per bit of memory has decreased at rate of about 30% per year.

Even "embedded" processors may have substantial amounts of memory. An embedded processor today is typically an ARM processor: a 32-bit processor with many kilobytes of cache memory and access to potentially many megabytes of off-chip RAM. Even simple "smart card" chips have substantial memory. For example, a typical SIM card in a cell phone is a Java Card with 64KB of code, 16KB of EEPROM and 1KB RAM [26].

RFID chips are the most resource-constrained environment to consider. At the moment, they are so severely constrained that doing serious cryptography on an RFID chip is nearly impossible. But even this is evolving quickly; RFID chips with 8KB memory (ROM, not RAM) are now available. Programming a respectable hash function onto an RFID chip may soon be a realistic proposition.

Thus, it is reasonable to consider hash function designs that use memory more freely than the designs of the early 1990's.

We will thus take as a design goal that MD6 should be implementable using at most 1KB RAM.

An MD6 compression function calculation can easily be performed within 1KB of RAM, and for $L = 0$ the entire MD6 hash function fits well within the 1KB RAM limit. Larger systems can use more memory for increased parallelism and greater efficiency by using the default value $L = 64$.

3.3.1 Larger block size

A major benefit of the increased availability of memory is the ability to use compression functions with larger inputs.

MD4 and MD5 have message+chaining sizes of $512 + 128 = 640$ bits. SHA-1 has message+chaining sizes of $512 + 160 = 672$ bits. The first members of the SHA-2 family (SHA-224, SHA-256) have message+chaining sizes of $512 + 256 = 768$ bits, while SHA-384 and SHA-512 have message+chaining sizes of $1024 + 512 = 1536$ bits.

Such input block sizes are arguably too small for what is needed for SHA-3, where we want a hash function producing 512-bit outputs. The compression function function must produce at least 512 bits of output, and preferably more. With 1536-bit inputs, we can barely get a compression factor of three. There is little room left over for auxiliary inputs, and no possibility of following the

"double-width pipe" strategy suggested by Lucks [55], wherein chaining variables are twice as large as the final hash function output.

Thus, MD6 chooses an input data block size of 512 bytes (i.e. 64 words, or 4096 bits), and a compression factor of four for its compression function. The compression function output size is 1024 bits—twice as large as any message digest required for SHA-3, so MD6 can follow the double-width pipe strategy.

Many benefits follow from having relatively large compression function inputs.

First, having large inputs allows us to easily incorporate nontrivial auxiliary inputs into the compression function computation, since these auxiliary inputs then take up a smaller fraction of the compression function inputs.

Second, large inputs allows for more potential parallelism within the compression function computation. In MD6, all 16 steps within a single round can be computed at once in parallel.

Third, a large message input block can accommodate a small message within a single block. For example, 512 bytes is a common size for disk blocks on a hard drive; such a block can be hashed with a single compression function call. Similarly, a network packet of up to 512 bytes can be hashed with a single compression function call.

Fourth, having a larger compression function input block should make cryptanalysis (even automated cryptanalysis) more difficult, since the number of compression function computation steps also increases. For example, with a differential attack, the probability of success typically goes down exponentially with the number of computation steps. Existing hash functions have a number of computation steps that may be fewer than the number of hash function output bits desired, whereas MD6 has many more than 512 steps. If each such step has a probability of at most 1/2 of following a given differential path, then the large number of computation steps under MD6 makes such attacks very unlikely to succeed.

3.3.2 Enabling parallelism

A second major benefit of the increased availability of memory is the ability to enable parallel computation.

> SHA-3 should be able to exploit the parallelism provided by the forthcoming flood of multicore processors.

Yet, a parallel approach requires more memory than a sequential approach. In the best case, a tree-based approach requires memory at least proportional to the height of the tree–that is, is, it requires memory at least proportional to the logarithm of the message size. With a branching factor of 4, a leaf size of 4096 bits, and a maximum message size of 2^{64} bits, this logarithmic factor is 26.

On a typical desktop computer, storage of $26 \cdot 512$ bytes (less than 14KB) is a trivial consideration. The easy availability of memory enables a multicore desktop to hash large files and streams very quickly in parallel.

3.4 Parallelism

We are at a transition point in the design of processors.

No longer can we expect clock rates to continue to increase dramatically with each processor generation. Indeed, clock rates may have plateaued, after having increased by a factor of 4000 in the last ten years. The reason is that processor power usage increases linearly with clock frequency, with all other things being equal. (In practice, all other things aren't equal, and the power usage increases nonlinearly, perhaps even quadratically or cubically, with clock frequency.) Further increases in clock rate would require exotic cooling technologies to handle the extremely high power densities that would result from the high clock rates.

Instead, performance gains will now be obtained mostly through the use of parallelism—specifically, through the use of multicore processors. Dual and quad-core processors are now easily available, and processors with more cores are becoming so. We can reasonably expect the number of cores per processor to double with each successive processor generation, while the clock rates may change little, or even decline slightly.

Anwar Ghoulum, at Intel's Microprocessor Technology Lab, says "developers should start thinking about tens, hundreds, and thousands of cores now."

A related expected trend is that, an increase in the parallel computational power of chips (measured as transistors $\times$ clock rate) compared to their I/O bandwidth, as transistor density improves faster than that of I/O pins and pads. Over the next 15 years, this ratio is expected to roughly quadruple, while serial computational power (measured in clock rate) compared to chip I/O bandwidth will decrease by a factor of roughly 10.[1] Hence, in the long term parallelizable functions can afford more computation (and, hopefully, higher security) for a given level of I/O utilization.

A tree-based design is the most natural approach for exploiting parallelism in a hash function. Ralph Merkle's PhD thesis [65] described the first such approach, now known as "Merkle trees." The computation proceeds from the leaves towards the root, with each tree node corresponding to one compression function computation. If the compression function has a compression factor of 4, then each tree node will have four children. MD6 follows this approach.

Damgård [34] also described parallel approaches to hash functions. These methods can be viewed as constructing a tree by a level-by-level bottom-up computation, wherein the nodes within each level of the tree are computed in parallel. Damgård also suggested stopping the parallel level-by-level computation after some fixed number of levels, and finishing up what is left with a sequential computation. MD6 also follows this approach when L is small; here

[1] These figures apply to high-performance chips and are derived from the International Technology Roadmap for Semiconductors 2007 [40], specifically Tables 1i/j, 3a/b and AP2a/b.

L (which he calls j) describes the number of levels to compute in parallel before switching to a sequential mode of computation.

Other authors have also explored parallel or tree-based hash function design. Perhaps most relevant to MD6 are various interesting and excellent proposals by P. Sarkar, such as [87]. His constructions are somewhat different, however, in that message input values are consumed at every tree node, not just at the leaves.

MD6 thus adopts a tree-based approach for maximum parallelizability.

However, there is definitely a trade-off between parallelizability and memory usage; a fully parallelizable approach may have a memory footprint that is too large for some small embedded devices.

Therefore, MD6 follows Damgård's lead by parameterizing the amount of parallelization. MD6 allows the user to set the parameter L to be the number of parallel passes before switching to a sequential mode. By setting L to 0, MD6 acts in a purely sequential mode and uses minimal memory (under 1KB). By setting L large, parallelism is maximized and memory usage is proportional to the logarithm of the input message size. (The value of L should of course be communicated along with the hash function output in cases where MD6 is used with a non-standard value.)

3.4.1 Hierarchical mode of operation

The initial design for MD6 did not have a parameter L controlling the mode of operation; the hash was always fully hierarchical.

However, it was felt that (a) there might be substantial need for an MD6 version that met tighter storage limits, and (b) it was easy to add an optional control parameter L that limits the height of trees used (and thus the storage used).

For small L, MD6 makes a collection trees of height at most L (which can be done in parallel), and then, if there is enough data to make more than one such tree, combines their root values with a sequential Merkle-Damgård pass.

When $L > 0$, an implementation with infinite parallelism could compute all of the trees in time L, leaving a sequential computation requiring time $O(m/4^L)$, where m is the size of the original message.

In practice, infinite parallelism is unavailable, but even so, using MD6 to hash a long message on a multicore processor with P processors can result in a speedup by a factor of P, for any P reasonable to imagine over the next several decades.

Envisioning that multi-core processors will be very common, we set the default value for an unspecified L to be $L = 64$ (giving tree-based hashing): $H_{d,K} = H_{d,K,64}$.

However, applications on very restricted processors may wish to choose $L = 0$ for a purely sequential hash function with minimal memory requirements.

3.4.2 Branching factor of four

A very early version of the MD6 design had chaining variables of size 512 bits and a branching factor of eight (instead of the current branching factor of four). However, it was felt to be more important to ensure that the wide-pipe principle was maintained for all values of d (particularly for $d = 512$, where security requirements are the toughest), so the chaining variable size was increased to 1024 bits.

It would be possible, of course, to maintain the wide-pipe principle, while improving the branching factor when $d < 512$. For example, one could have $c = 1024$ for $d = 512$ or $d = 384$ (a branching factor of four), and $c = 512$ for $d = 256$ or $d = 224$ (a branching factor of eight). But this adds complexity and doesn't actually do much in terms of improved efficiency (it makes a difference of about 16%), since almost all of the actual work is in the leaves.

3.5 A keyed hash function

Many existing hash functions, like MD5 and the SHA hash functions, are "keyless"—there is only one version of a hash function (per digest size). There is no standardized way to supply a key, salt, or index value to get different versions of the hash function.

However, it is often proposed to convert a hash function into a MAC (message authentication code) by augmenting the hash function message input with a secret key input that is shared between sender and receiver.

The natural ways of doing this, however, turn out to be fraught with unexpected risks [49, 82, 83]; the usual Merkle-Damgård iterative structure of a hash function interacts badly with the idea of just appending and/or prepending the secret key to the message. With some care, however, it can be done; HMAC is one such approach [6] that is popular and well-studied.

There are other approaches, such as the "XOR-MAC" of Bellare et al. [7]. For our purposes, XOR-MAC is interesting because (a) the key is involved in every block of the computation and (b) the position i of the i-th block is involved in the computation on the i-th block. (MD6 shares these properties.) However, the XOR-MAC construction doesn't work at all for unkeyed hash functions or for applications where the key is public, since the outer XOR's can be defeated by suitable use of linear algebra to produce inversions or collisions.

There are also other applications of keyed hash functions. For example, it is common to "salt" the computation of a hash function on a password before storing it. Here is salt is effectively a non-secret key (or index). The use of the salt provides variability so that an adversary can't easily pre-compute a dictionary of the hashes of commonly used passwords; he has to redo the computation for each such salt.

See Bellare and Ristenpart [10] for further excellent discussion of keyed hash functions and their advantages.

In some situations, it is useful to choose keys randomly; this yield randomized

hashing techniques [23], with many interesting applications and security benefits.

MD6 has a 64-byte (that is, 512-bit) "key" that may be used as a secret key for a MAC, a public value for a salt, or as a randomized index for the hash function.

The reasons for making the key input relatively large are: (a) having a large input block size makes this relatively cheap, (b) it allows for easy use of text-based keys, and (c) it enables the use of MD6 outputs (even 512-bit outputs) as keys. The last advantage makes it possible to hash a very long value for use as a key in some other hash computation.

The MD6 key is actually of variable length (between 0 and 512 bits, inclusive). By preserving the original length of the supplied key in the control word, we ensure that the MD6 key doesn't suffer from certain "extension" or "related-key" attacks—a key K and a key K' will cause MD6 to behave differently even if K is a prefix of K'.

3.6 Pervasive auxiliary inputs

As noted above, in MD6, auxiliary inputs are supplied to each compression function computation. This point is worth emphasizing and elaborating.

3.6.1 Pervasive key

For example, the MD6 key is an auxiliary input to every compression function computation. The key is not just, say, appended or prepended to the message. The pervasive presence of the key prevents length-extension attacks and other mischief.

3.6.2 Pervasive location information: "position-awareness"

Also, each compression function operation "knows where it is"—the unique node ID U specifies exactly where the present compression function invocation lies in the dataflow graph. We call this "position-awareness".

Haifa [18] proposed such position-awareness for sequential Merkle-Damgård-like computations. MD6 generalizes this to tree-based hash functions—the compression function at a node has as input its level number and its position within the level. Moreover, it knows whether or not it is the "final" (or "root") node.

Such position-awareness provides major benefits in addressing number of vulnerabilities for non-position-aware modes of operation. The reason is that any hash function subcomputation (i.e. a sub-chain or subtree) can only be used in one place in the overall computation.

In other words, position-awareness precludes "cut-and-paste" attacks—moving a hash-function computation from one place to another. They also preclude attacks (such as Kelsey's) that perform hash function subcomputations, hoping that they will fit in "somewhere" to produce a collision.

3.6.3 Pervasive control word information

MD6 also has "control words" supplied pervasively to each compression function computation.

The control words also specify the structure of the tree, by indicating how many message bits are in each leaf node and how many children are present for each non-leaf node. The MD6 control words specifies amount of "padding" at each node and the node's level number. This representation makes unnecessary the use of traditional padding techniques (such as appending a one, then as many zeros as necessary) while still uniquely specifying the tree structure.

Also, the root of the tree is specially marked via a bit in the control word. That is, the compression function computing the final output value is distinguished; it has an input $z = 1$ that doesn't occur for any other node. This has benefits for preventing various forms of "extension attacks" and other mischief.

Such pervasive structural information ensures that the length of the underlying message (at the leaves the tree) is uniquely represented. (MD6 does not explicitly represent the overall length of the message as an explicit input to any one compression function computation; rather it is uniquely determined by the tree structure information represented as inputs to each node.)

The control words also specify other information about the desired computation: the desired hash function output size d, the number r of rounds, the optional "mode parameter" L, and the length of the key K.

The incorporation of the control word V helps to ensure that the hash function indeed acts like a "new, unrelated" function whenever any of the components of the hash function change. Otherwise, for example, changing the output hash size or the number of rounds might yield outputs that are clearly related to the original outputs.

In general, MD6 follows a policy of "pervasive explicitness." As we have learned through many painful lessons in cryptographic protocol design, leaving parameters implicit is a good recipe for introducing vulnerabilities. The same is true in hash function design (e.g. the lack of position-awareness). It is better to make sure that each potentially relevant or descriptive parameters is explicitly an input to each compression function.

3.7 Restricted instruction set

MD6 is defined in terms of a very restricted set of logical operations, giving advantages in terms of security, simplicity, and efficiency.

3.7.1 No operations with input-dependent timing

Timing attacks have turned out to be surprisingly effective against cryptosystems that utilize primitive operations whose running time depends upon the input data being manipulated. We have thus learned to avoid whenever possible:

- Complex instructions such as multiplications and divisions whose running time may be faster when their inputs are simpler (as exploited, e.g., by Brumley et al. [22]).
- Shift or rotate instructions with input-dependent shift or rotate amounts.
- Input-dependent conditional branches (exploited, e.g, by Aciiçmez et al. [1][2]).
- Table-lookup instructions where the table index is input-dependent. Due to cache effects, such instructions can cause input-dependent running time for both the cryptographic code itself (e.g., by Tsunoo et al., [92], Bernstein [12] and Osvik et al. [74]) and independent adversarial processes on the same processor (e.g., Osvik et al. [74][91] and Percival [76]).

The last limitation (no table-lookups) precludes the use of one of cryptographer's favorite components, the "S-box", unless the the S-box is sufficiently small to be implemented purely in registers or via bitslicing.

3.7.2 Few operations

The MD6 compression function is implementable using only a very small set of logical operations (on 64-bit words):

- **XOR**: the bit-wise exclusive-or of two words
- **AND**: the bit-wise and of two words
- **FIXED SHIFT**: the left-shift or right-shift of an input word by a fixed amount (with zeros shifted in).

Other operations can be built from these in the usual way:

- **NOT**: By XORing with `0xFF...FF`, the all-ones word:
- **OR**: Using DeMorgan's Law: $(x \vee y) = \overline{x \wedge y}$.
- **FIXED ROTATE:** Using two fixed shift operations, and **OR**.

3.7.3 Efficiency

All of the above instructions can be implemented in constant time regardless of the word size; there are no carry-propagations to worry about, for example, as there would be for an adder.

(A 64-bit adder is rather useful for computing the "unique node ID" U in the MD6 mode of operation. But here the nodeID is just a counter; there are no data-dependent inputs. The addition counter increment can be performed using the above data-independent instructions (somewhat awkwardly), or by using a regular addition instruction or circuit. Since it is computed outside the inner loop, it was felt that the use of standard binary notation for the nodeID counter was more helpful in terms of clarity of design than the "purity" of trying to purge all possible uses of addition from the design. Conversely, the recurrence for S inside the inner loop was chosen to avoid the use of additions.)

3.8 Wide-pipe strategy

The size of the intermediate chaining variables (linking one portion of the hash computation to another) may have a very significant effect on the security of the overall scheme.

Stefan Lucks was the first to articulate [54, 55] the important of the size of these internal variables, stating,

> "The size of the internal hash values is a security parameter in its own right."

Lucks proposed the "wide-pipe" strategy, wherein the internal chaining variables have a size that is independent of (and larger than) the final hash function output. In the double-pipe strategy, the internal variables are twice as long as the final hash function output.

A double-pipe strategy ensures that the chance that two chaining variables collide (have the same value) is essentially negligible. The double- (or wide-) pipe strategy is an effective anti-dote that an adversary might otherwise obtain from the birthday paradox, as in recent "multicollision attacks" studied by Joux [46], Nandi et al. [68, 69] Hoch et al. [44], and Yu et al. [97].

A wide-pipe strategy has been invoked in other recent designs, such as [13].

MD6 adopts a wide-pipe strategy: all internal chaining variables are $cw = 1024$ bits in length, which is at least twice the length of the final hash function output.

3.9 Nonlinear feedback shift register

Many hash functions can be easily viewed as an NLFSR, with the understanding that each step also makes use of additional inputs. For example, MD4 looks like a NLFSR with four 32-bit words of state.

MD6 places all inputs in the shift register before the computation begins, and has no non-constant inputs entering "from the side."

Because there are no inputs "from the side", it may be more difficult for a cryptanalyst to find differential attacks on the collision-resistance of MD6.

The regular structure of the NLFSR also facilitates many aspects of our security analysis.

The following subsections discuss some of the details of the design choices made in implementing the MD6 compression function as a NLFSR.

3.9.1 Tap positions

The tap positions t_0, t_1, and t_2 are intentionally chosen to be somewhat small, so that differences will propagate more rapidly. On the other hand, $\min_i(t_i) = 17$ is not too small, in order to support parallelism within the compression function computation.

Note that $\gcd(t_0, n) = \gcd(t_1, n) = \gcd(t_2, n) = \gcd(t_2 - t_1, n) = \gcd(t_4 - t_3, n) = 1$.

The tap positions $t_0,\ldots,t_4$ were chosen as follows:

- The tap position $t_0 = 17$ was chosen to be relatively small (for faster avalanche effect), yet not too small (so that MD6 can exploit parallelism by computing a full round in parallel), and relatively prime to the length of a round (so that different shift amounts come in to play during the feedback propagation). This tap is a linear feedback position for fastest avalanche.

- The tap position $n = 89$ was necessarily chosen as a tap position for linear feedback since we wish the feedback operation to be invertible (that is, we can compute A_{i-n} from $A_{i-n+1..i}$; that is, we can run the shift register "backwards").

- All tap positions were chosen to be nonzero modulo $c = 16$ and to have different residues modulo $c = 16$. This minimizes possible negative interactions from basing shift amounts on the step size modulo 16. The tap positions must also satisfy $t_4 - t_3 \neq t_2 - t_1$, so that the "and" gates will pair inputs at different distances from each other.

- Tap positions $t_1,\ldots,t_4$ were chosen using a computer search. It was determined by a simple brute force search that the chosen values ($t_1 = 18$, $t_2 = 21$, $t_3 = 31$, and $t_4 = 67$) minimize "dep" (here equal to 102), where "dep" has the property that if at least "dep" computation steps are performed, then each of the c most recent values computed is guaranteed to depend (in a formal sense) on all 89 input words to the compression function. The program (`tapopt.c`) is available. See Table 9.1.

3.9.2 Round constants

As noted earlier, the recurrence relation 2.4 for the round constants is one-to-one (see Schnorr et al. [88, Lemma 5]).

Andrew Sutherland, who developed some great algorithms and tools for determining the orders of elements in groups for his Ph.D. thesis [90], determined with only a few seconds of computer time that the sequence S'_j has period

$$\begin{aligned} &6052837899185946603 \\ &= 3 * 3 * 7 * 59 * 233 * 1103 * 2089 * 3033169 \\ &\approx 2^{62.4} \ . \end{aligned}$$

Of course, for MD6, we don't need more than the first two hundred of these elements, so the actual period length doesn't matter as long as it isn't very short.

3.9.3 Intra-word diffusion operation g

We originally considered obtaining intra-word diffusion by using rotations of words, but a rotation is typically implemented using two shifts, and it seems likely that two shifts can yield more benefit than simply using them to implement a rotation.

Note that whatever shifts are chosen (as long as $r_i \neq \ell_i$), a one-bit change to the input of g will cause from two to four bits of the output of g to change.

Note also that g is one-to-one (i.e., invertible). An input change of any sort always causes an output change.

To find shift values r_i, ℓ_i, $0 \leq i < c$ that provide a good diffusion operator g, we ran a computer search. The shifts in g were determined by a computer search. The program (`shiftopt2.c`) is available. It attempts to maximize the rate of diffusion, given the tap positions and subject to some constraints on the choice of r and ℓ values.

The following constraints were observed (they are heuristic in nature, and were felt to improve the quality of the design, although a straightforward randomly generated table might do nearly as well). Each shift value must be nonzero and at most $w/2$, and r_i and ℓ_i must not be multiples of each other. The function g_{r_i,ℓ_i} must be such that an output of hamming weight one cannot be generated by an input word of weight less than five. If $w \geq 32$, then r_i and ℓ_j must also not be multiples of each other, for any j such that $(i-j) \in \{t_0, t_5, t_5 - t_0\}$ (all subscripts taken modulo c). These latter conditions (which are hard to satisfy for $w < 32$) help ensure that a left shift in one round is not followed by a right shift of the same amount (or a multiple thereof) in a latter round.

A table of shift values was chosen to provide the fastest avalanche effect among those examined. The program for computing these shift tables is available.

The operations $x = x \oplus (x << a)$ and $x = x \oplus (x >> b)$ are known as xorshift operations in the random-number literature. They were recently introduced by Marsaglia [56], and studied by Panneton and L'Ecuyer [75]. Our usage is vaguely similar to Marsaglia's proposal, although MD6 is considerably more complex than Marsaglia's simple generator. (As it should be, as MD6 must meet cryptographic requirements that are not relevant for a simple random number generator.)

3.9.4 Constant Vector Q

The presence of the constant vector Q helps to defeat inversion attacks, since someone working the compression function backwards from a desired output would need to match all the bits in Q.

For the purposes of security analysis, Q should be considered as hard-wired; it is truly a constant. Although we talk about it as an "input" to the compression function, it is a constant input that may not vary. Thus, for example, it doesn't count as "inverting" the compression function to come up with an input $N =$

(Q, K, U, V, B) where Q has a value other than the standard fixed constant value. Our calling Q an "input" is merely for for expository convenience; the true inputs to the compression function are just B, V, U, and K. Producing "inputs" to the compression function with a different value of Q is no more legitimate than "solving" an equation $7x^5 - y^2 = 1$ by giving a solution for "a different value of 7". In our security analysis, therefore, we focus on the compression function f_Q mapping $\mathbf{W}^{n-q}$ to $\mathbf{W}^c$ induced by fixing the Q input to f.

3.10 Input symmetry

The MD6 compression function doesn't treat any of its inputs (B,V,U,K, or Q) in any special manner; they are treated essentially "uniformly". This can be formalized in the following design principle:

> **Principle of Input Symmetry:** For any fixed permutation π of the nw input bit positions to the MD6 compression function, the security properties of MD6 should be effectively unchanged if the input bits were re-ordered according to π before the compression function were applied.

3.11 Output symmetry

Similarly, output bits of the MD6 compression function can similarly be treated uniformly. This can be formalized in the following design principle:

> **Principle of Output Symmetry:** For any fixed permutation π of the cw output bit positions of the MD6 compression function, the security properties of MD6 should be unchanged if the output bits of the compression function were re-ordered according to π.

In particular, it shouldn't matter that the final output is the first d bits of the compression function output; it could have been any set of d bits, in any order.

3.12 Relation to encryption

The compression function mapping may be viewed as a possible encryption algorithm. The plaintext size and ciphertext size is 89 words. We may consider the sequence S'_j ($j \geq 0$) as a sequence of "round keys"; each S'_j is used in exactly one round. These keys could be generated by another process from a supplied key; in MD6, the values S'_j are fixed. The simplicity of the operations used by MD6 (in particular, no table lookups) makes this encryption algorithm an interesting alternative to AES, which has suffered heavily due to "cache attacks" against its table lookup operations (see Section 3.7.1).

Many previous hash function designs can be viewed as having an encryption algorithm buried in their compression functions. Such hash functions typically use the Davies-Meyer construction: given a chaining variable C_{i-1} and a message block M_i, the next chaining variable C_i is computed as $C_{i-1} \oplus E(M_i, C_{i-1})$. That is, it is the xor of the previous chaining variable and the encryption of that variable under the key M_i.

MD6 does not use the Davies-Meyer construction, since we have the luxury of designing a hash function that includes the message, key, control, and other variables as uniformly treated inputs. This may make security analysis simpler.

3.13 Truncation

The c-bit output of the compression function at the root is truncated to produce the final d-bit hash function output.

Although other operations could have been used here (e.g., some form of "extractor"), truncation should suffice if MD6 with output length $d = c$ is a good pseudo-random function.

3.14 Summary

This chapter has provided an overview of the reasons for the major design decisions in MD6. Some additional relevant discussion can be found in Chapters 7 and 6 on security.

Chapter 4

Software Implementations

This chapter describes various software implementations of MD6, and reports on the efficiencies of these implementations. MD6 adapts easily to a wide variety of platforms with good efficiency, and works extremely well on platforms providing support for parallel computation. For example, throughput rates in excess of 1GB/second are obtained on a 16-core CPU.

We note that our reference implementation, 32-bit optimized implementation, and 64-bit optimized implementation are the same code.

All of our software is provided under the open source "MIT License" [1], which has very liberal terms for copying, redistribution, modification, etc.

4.1 Software implementation strategies

This section discusses various approaches to implementing MD6 in software.

4.1.1 Mode of operation

We first review two approaches to implementing the MD6 mode of operation: a "layer-by-layer" approach, and then a "data-driven tree-building" approach.

4.1.1.1 Layer-by-layer

Chapter 2 describes the MD6 hash algorithm in a "layer-by-layer" manner: there are up to L parallel passes that reduce each layer in turn in size by a factor of four to produce the next layer up, and then if necessary a sequential pass to produce the final output.

A layer-by-layer approach was chosen for the exposition of MD6 in Chapter 2 since this approach is the simplest to describe.

However, a layer-by-layer approach is not suitable for the software implementations required by NIST's SHA-3 competition since it presupposes that

[1] `http://www.opensource.org/licenses/mit-license.php`

the entire message is available at once, and since it uses storage proportional to the length of the message being hashed.

We have explored layer-by-layer implementations during our work on MD6, but our submitted reference and optimized implementations are based on a different approach, which we call the "data-driven tree-building" approach.

4.1.1.2 Data-driven tree-building

The API required for SHA-3 allows message data to be provided in pieces of essentially arbitrary size. After the initial call to `Init`, a number of calls to `Update` may be made, each supplying a piece of the message. (Only the last call to `Update` may supply a message piece containing a non-integral number of bytes, according to NIST.) Then, a call to `Final` finishes the computation and produces the desired hash value. The length of the message is not known to the hash function implementation until the call to `Final` is made.

Our implementations of MD6 via this API are data-driven. This means that each compression function operation will be evaluated "greedily", as soon as all of its inputs are available.

At any point in time during the computation, there may be one compression function operation at each level of the tree that is still waiting to have its inputs determined. These pending operations form a path in the MD6 computation tree from the current root of the tree down to a leaf node.

Thus, the storage required by an MD6 implementation for a given message will be proportional to the actual height of the MD6 tree during the hash function computation for that message.

In the case of MD6, there can be at most 27 nodes awaiting data, since a message has length at most 2^{64} bits, the final chaining variable output has length $cw = 2^{10}$ bits, the tree has branching factor four, $2^{64}/2^{10} = 2^{54}$, and $\log_4(2^{54}) = 27$. If the message has length less than 2^m bits, then at most $(m - 10)/2$ nodes will be awaiting data at any time. If $L = 0$ for a sequential implementation, at most 1 node will be awaiting input at any time. In general, the number of nodes waiting for input won't be larger than $L + 1$.

As more data is received, the tree being constructed may grow in height. The node X that currently represents the root may have its subtree reach capacity, and a new tentative root node Y may need to be added at the top, as the parent of X. Note that this growth will only happen when it is truly necessary; Y shouldn't be added if no more input is coming.

The tree grows in a "bottom-up, left-to-right" manner.

The final height of the tree, and the final number of nodes at each level, won't be known until all input has been received (when `Final` is called).

When `Final` is called, the pending computation nodes are completed in a bottom-up manner, with results passed up to the next level, and with padding supplied as necessary to provide the missing input at each level. Finally, the compression function for the root node is performed, yielding the final hash output.

(Note that the root node compression can't be done before `Final` is called, since the current root may or may not be the final root node, and isn't clear whether the root node should be compressed with input $z = 0$ or input $z = 1$ until we know whether more input is coming or not.)

A slight modification is needed in the case that the height of the current root is $L+1$, since nodes at level $L+1$ are processed in a sequential rather than a hierarchical manner. Such nodes contain an IV (for the first such node) or a chaining variable "from the left" (for the other such nodes); when such a node is compressed its output is passed "to the right" (to a new node at level $L+1$) rather than "up".

We note that our numbering scheme for unique node ID's works well within such a data driven approach, since the level number and index within level of a node are known without knowing if more data will be coming. This would not have been true had we numbered nodes in a top-down manner starting at the root.

This data-driven approach nicely accommodates the requirements of the NIST SHA-3 API, while retaining formal equivalence to the layer-by-layer approach.

4.1.2 Compression function

We now review strategies for implementing the MD6 compression function in software. Fortunately, the MD6 compression function is simple enough that clever optimization techniques are not required in order to obtain excellent performance.

The only optimization applied to our program was loop-unrolling: the inner loop of the compression function is unrolled 16 times. That is, the 16 steps of the inner loop, comprising one round, were unrolled. This optimization provided substantial speedup, yet could be done without hurting the clarity of the code.

Other aspects of code generation, such as managing 64-bit values efficiently, were left entirely to the compiler.

The code spends about 97.5% of its time in the compression function inner loop, so optimization outside of the inner loop is nearly pointless.

As far as we can tell, it is possible to get essentialy optimal code for the MD6 inner loop from standard C compilers. We don't know of any optimizations for the inner loop, even using assembler, that would produce code better than the best compiler output. This is a nice feature of MD6: the inner loop is so simple (even when unrolled 16 times) that a good compiler can produce code that is effectively optimal.

One potential "optimization" is to reduce the memory footprint of the main compression loop to improve utilization of the cache and/or reduce the overall memory usage. Instead of allocating $rc+n$ words for the array A, allocate some smaller number (perhaps only $n = 89$). This strategy makes sense under cache memory pressure, e.g., due to very small caches or simultaenous multithreading. It did not result in any speed improvement in our single-process test platforms.

4.2 "Standard" MD6 implementation(s)

We have a produced a single implementation of MD6 as our Reference, Optimized 32-bit, and Optimized 64-bit submissions.

This code uses the data-driven tree-building approach and one-round loop unrolling, as described above.

There are three files defining this implementation:

- `md6.h` – defining the constants and data structures used by MD6.
- `md6_compress.c` – defining the MD6 compression function.
- `md6_mode.c` – defining the MD6 mode of operation.

For compiling on Microsoft Visual Studio, compatible versions of the C99 standard header files `inttypes.h` and `stdint.h` need to be supplied (primarily for the definition of `uint64_t`). Our submission includes suitable versions.[2]

In addition, two files are needed to provide compatibility with the NIST SHA-3 API:

- `md6_nist.h` – defines the NIST SHA-3 API interface.
- `md6_nist.c` – implements the NIST SHA-3 API interface.

Thus, for NIST to compile and test the MD6 code supplied, NIST needs the two header files (`md6.h` and `md6_nist.h`, and the three programs (`md6_compress.c`, `md6_mode.c`, and `md6_nist.c`).

We have also supplied a standard driver program, `md6sum`, which is similar to the well-known program `md5sum`. This program can be used to hash files, run timing tests, print intermediate results, and experiment with different parameter settings. Its use is described further in Section 8.4.

4.2.1 Reference Implementation

As noted above, our one submitted implementation is the Reference Implementation.

4.2.2 Optimized Implementations

4.2.2.1 Optimized 32-bit version

As noted above, our one submitted implementation is also the Optimized 32-bit implementation.

4.2.2.2 Optimized 64-bit version

As noted above, our one submitted implementation is also the Optimized 64-bit implementation.

[2] Source: `http://code.google.com/p/msinttypes`

4.2.3 Clean-room implementation

As part of our development process, a new team member (Jayant Krishnamurthy) was given only the MD6 documentation, and asked to develop an MD6 implementation in a "clean-room" manner (i.e., without access to our pre-existing MD6 code base). This implementation was done in a layer-by-layer manner, to minimize the chance of incompatibilities with the documentation. Since the clean-room implementation is only for testing, and not for submission, the fact that the clean-room implementation had overall length limitations was not too much of a concern.

The two implementations were then tested extensively for equivalence on random inputs.

A few corner-case bugs in the documentation and in the programs were uncovered and fixed. At this point the programs agree on all inputs we have tested, and we have high confidence that they correctly implement MD6 as documented in this report.

4.3 MD6 Software Efficiency Measurement Approach

This section describes the approach used to measure the running time of our code, and the results we obtained. We report results separately for the code as compiled by GCC and the code as compiled by Microsoft Visual Studio 2005, since the results are somewhat different (especially for 32-bit code).

We report elapsed time in seconds and in "ticks" (clock cycles, as measured by the `RDTSC` instruction).

4.3.1 Platforms

We describe here the platforms (processors) used for testing the efficiency of our code.

4.3.1.1 32-bit

Our 32-bit test platform is a Lenovo ThinkPad T61 laptop with an Intel Core 2 Duo T7700 processor running at 2.39GHz and 2GB of RAM, running Windows XP Professional (Service Pack 2). This CPU has L1/L2 caches of sizes (128KB,4MB). (The L1 cache has 32KB instruction cache and 32KB data cache in each of its two cores.)

This is essentially equivalent to the 32-bit NIST Reference Platform described in Section 6.B (page 62219) of the NIST Federal Register Notice [70] describing the SHA-3 submission requirements (modulo running Windows XP rather than Windows Vista).

The compilers we used are Visual Studio 2005, and GCC 3.4.4 under Cygwin.

4.3.1.2 64-bit

We used three 64-bit platforms: one running Windows and two running Linux.

The Windows platform (aka "lumen") runs Windows 64-bit XP Professional. This platform has a 3.0GHz clock rate and an Intel E6850 Core 2 Duo processor with L2 cache of size 4MB and and L1 data cache of 32KB per core (64MB total).

The first Linux platform (aka "scooby-doo") is a 64-bit Core 2 Quad Xeon running at 2.6GHz. It has L1/L2 data caches of size (32KB,2MB).

The second Linux platform (aka "T61-Ubuntu") is a Lenovo ThinkPad T61 laptop with an Intel Core 2 Duo T7700 processor running at 2.39GHz and 2GB of RAM, runing Ubuntu. This is the same laptop as our 32-bit T61 laptop, but booted into Ubuntu rather than into Windows XP. Since the Core 2 Duo supports 64-bit instructions, we can use this platform for 64-bit testing as well. The GCC compiler version is 4.2.3.

4.3.1.3 8-bit

Our 8-bit platform is the Atmel AVR [5, 4]. We chose this platform because of its popularity, robust development tool support, and clean architecture. The AVR is a RISC CPU with a load/store architecture and thirty-two 8-bit registers. Logical and arithmetic operations are only performed on data in registers. Among its architectural advantages are single-cycle operations on data contained in registers, offering up to 1 MIPS/MHz. Loading (or storing) a byte from SRAM takes two cycles. Given the simplified instruction set, producing code from a high-level language like C is comparatively easy. Among the tools available for the AVR is avr-gcc, a cross-compiler based on the same toolchain as GCC. Support for this compiler meant minimal code changes to produce a working implementation on the AVR.

The AVR family consists of embedded processors with a wide variety of features, from 64 bytes of SRAM up to 32 KB and a maximum clock frequency of 1 MHz up to 20 MHz. For testing and debugging purposes, we chose the Atmel ATmega1284p, a part available with up to 16 KB of SRAM, and up to 20 MHz clock. In addition, it is supported by WinAVR, allowing for full software simulation and cycle counts.

4.4 MD6 Setup and Initialization Efficiency

The setup time for MD6 is independent of the desired length d of the output hash function digest.

Figure 4.1 gives the number of clock cycles required for MD6 initialization on various platform/compiler options. These were obtained using our `md6sum` application, e.g.

```
md6sum -s1e6
```

to obtain the time and cycle counts needed for one million initializations.

Platform+compiler	ticks/initialization
(32-bit) Cygwin + GCC (T61)	1156
(32-bit) XP + MSVS (T61)	1450
(64-bit) Cygwin + GCC (scooby-doo)	2111
(64-bit) Ubuntu + GCC (T61-Ubuntu)	2062
(64-bit) XP + MSVS (lumen)	2070

Figure 4.1: MD6 initialization times. (Here a "tick" is one clock cycle.)

Setup is very efficient: less than 7% of the cost of a typical compression function call.

4.5 MD6 speed in software

This section provides the results of timing tests on our software MD6 implementation on various platforms.

The timing results are made using our `md6sum` application, e.g.

```
md6sum -t -d160 -B1e9
```

to measure the time and cycles taken to produce a 160-bit hash value for a one-gigabtye dummy file.

4.5.1 32-bit processors

Compiling MD6 on a 32-bit processor relies on the ability of the compiler to produce good code for handling 64-bit quantities.

Figure 4.2 gives the speed of MD6 on our 32-bit Cygwin Linux platform using GCC 3.4.4. Here the compiler optimization switch is `-O3`.

32-bit Core 2 Duo 2.4GHz (T61)			
	Cygwin+GCC		
	ticks/byte	ticks/comp. fn	speed (MB/sec)
MD6-160	76	29545	31.2
MD6-224	91	35043	26.3
MD6-256	98	37638	24.4
MD6-384	126	48639	18.9
MD6-512	155	59707	15.4

Figure 4.2: 32-bit MD6 speed using Cygwin + GCC 3.4.4

MS Visual Studio 2005 produced code that was significantly better (about 35% better), as shown in Figure 4.3. Here the compiler optimization switch is `/O2`.

32-bit Core 2 Duo 2.4GHz (T61)			
	XP + MSVS 2005		
	ticks/byte	ticks/comp. fn	speed (MB/sec)
MD6-160	54	20855	44.1
MD6-224	63	24363	37.7
MD6-256	68	26464	34.7
MD6-384	87	33607	27.4
MD6-512	106	40975	22.4

Figure 4.3: 32-bit MD6 speed using MSVS

hash algorithm	speed (MB/sec)
MD4	509
MD5	370
SHA-1	209
SHA-224	108
SHA-256	106
SHA-384	37
SHA-512	38
RIPEMD-160	138
Tiger	108
HAVAL (5 passes)	171
Whirlpool	30

Figure 4.4: Speed of some other hash functions, as measured by `sphlib`, on our 32-bit T61 laptop, with compiler `gcc`.

For comparison, we can measure other hash functions using sphlib, as shown in Figure 4.4.

4.5.2 64-bit processors

Because the MD6 is 64-bit oriented, and because the inner loop of the compression function is so simple, a good compiler can produce excellent or optimal code for the MD6 inner loop on a 64-bit processor.

For example, GCC compiles the inner loop of MD6 so that each step takes only 16 instructions on a modern 64-bit processor (3 loads, 2 ands, 6 xors, 2 register-to-register transfers, 2 shifts, and 1 store). We don't know how to improve on this, even coding by hand.

Figure 4.5 gives the running time for MD6 on our 64-bit Linux platform, using GCC. (Note: the clock rate here, 2.6GHz, is slightly higher than for the standard 64-bit Windows platform.)

Ubuntu + GCC 4.2.3 produced code that was close, but slightly better (note

64-bit Xeon 2.6GHz (scooby-doo)			
	Cygwin+GCC		
	ticks/byte	ticks/comp. fn	speed (MB/sec)
MD6-160	25	9625	106.3
MD6-224	29	11292	90.5
MD6-256	31	12059	84.7
MD6-384	39	15351	66.6
MD6-512	48	18680	54.8

Figure 4.5: 64-bit MD6 speed using GCC (2.66 GHz processor)

the different clock rates); see Figure 4.6.

32-bit Core 2 Duo 2.4GHz (T61)			
	Ubuntu + GCC 4.2.3		
	ticks/byte	ticks/comp. fn	speed (MB/sec)
MD6-160	22	8717	105.0
MD6-224	26	10257	89.7
MD6-256	28	11057	83.2
MD6-384	36	14165	64.9
MD6-512	44	17278	53.2

Figure 4.6: 64-bit MD6 speed using Ubuntu + GCC 4.2.3

Windows Visual Studio gave the following results. (The MD6 software was compiled by MS Visual Studio on our T61 laptop on the Visual Studio 2005 x64 Cross Tools Command Prompt, then the executable was ported to lumen for execution.)

Here Microsoft Visual Studio produces code that takes very nearly the same number of clock cycles as the code produced by our two GCC platforms. (When comparing charts 4.7 and 4.5 it is best to focus on the clock ticks per byte or per compression function call, and not the megabytes/second, as the clock rates for the machines used in these two charts differ.)

For comparison, we can measure other hash functions using sphlib:

Compared to SHA-512, this single-threaded implementation of MD6 is slower by a factor of 1.7 (for 32-bit code) to 4 (for 64-bit code).

4.5.3 8-bit

The NIST Federal Register notice [70, 2.B.2] requires that submitters supply efficiency estimates for "8-bit processors."

NIST does not specify any particular 8-bit processor to consider, so we focus on the Atmel AVR and the ATmega1284p in particular. See [5, 4].

64-bit Windows XP (lumen)				
	XP + MSVS 2005			
	ticks/byte	ticks/comp. fn	speed (MB/sec)	speed(MB/sec)*2.4/3.0
MD6-160	24	9505	120.8	96.6
MD6-224	29	11251	102.2	81.6
MD6-256	31	11984	96.0	76.8
MD6-384	40	15483	74.3	59.4
MD6-512	48	18617	61.8	49.4

Figure 4.7: 64-bit MD6 speed using MSVS (3.0 GHz processor); last column scales speed down to 2.4 GHZ clock.

hash algorithm	speed (MB/sec)
MD4	769
MD5	477
SHA-1	238
SHA-224	158
SHA-256	158
SHA-384	202
SHA-512	202
RIPEMD-160	197
Tiger	396
HAVAL (5 passes)	241
Whirlpool	55

Figure 4.8: Speed of various hash algorithms on our 64-bit cpu "scooby-doo" (a Linux machine; speed measured with `sphlib` as compiled with GCC).

MD6 implementations are possible on more-constrained members of the AVR family, but our motivation here was to measure results using as much of the existing reference implementation as possible, leading us to favor parts with relatively larger SRAM sizes. In addition, this experience allows us to examine GCC's disassembled output in order to make accurate estimates of what could be achieved using custom assembly code.

These two approaches allow us to report results for code size optimization (C code compiled with the `-Os` flag) and speed optimization (assembly estimates). Practical embedded implementations are often optimized for code size, rather than pure computation speed, as limited flash storage for program code discourages typical speed vs. code size tradeoffs like loop unrolling and function inlining.

The reference implementation uses a working array of $rc+n$ 64-bit words. Together with overhead such as the call stack, the code fits MD6-160 and MD6-224

into the 16 KB memory of the ATmega1284p and actual timing measurements were taken, reflected in the table. With the memory strategy of the reference implementation (where A occupies $rc+n$ words), even using the sequential ($L = 0$) MD6 option, the working array for the larger digest sizes exceeds the 16 KB available on the AVR, so by measuring that a step of MD6 takes 1,443 cycles, we calculate the number of cycles required overall for the larger digest sizes. These measurements and calculations for the inner loop of MD6 are reflected in the table.

d	r	rc	cycles	time (20 MHz)
160	80	1280	1,848,013	92 ms
224	96	1536	2,217,549	110 ms
256	104	1664	2,401,152 (est.)	120 ms
384	136	2176	3,139,968 (est.)	157 ms
512	168	2688	3,878,784 (est.)	194 ms

Figure 4.9: Efficiency of MD6 on an Atmel 8-bit processor (one compression function call).

As noted earlier, an implementation of MD6 with smaller memory footprint is possible. Such an implementation uses $89 \times 8 = 712$ bytes of RAM to store the current state of the NLFSR. Longer messages would use the sequential MD6 option, so that working memory usage would stay under 1KB. Our standard x86 platforms have sufficiently large L1 cache that such an implementation strategy appears to have little or no benefit.

By contrast, available SRAM is at a premium on 8-bit devices. The NLFSR realization would allow us to implement $d = 256, 384$ and 512 on the AVR, or the smaller d values with much less memory. The memory savings come at the cost of more overhead for pointer arithmetic, amounting to about 5% of the total. With this approach, we can implement the NLFSR by using pointers to represent the head, tail, and tap positions instead of explicitly shifting the locations of values in SRAM.

When using the -Os flag to optimize for code size, the compiler takes the obvious approach to the word-wise AND and XOR operations. The two 64-bit operands are loaded from SRAM, consuming 32 cycles. Eight byte-wise logical operations consume 8 cycles, followed by 16 cycles to store the eight-byte result, taking a combined total of 56 cycles. Apart from the possibility of careful register allocation to save some load/store operations, it is unclear how to improve on this approach given the simplicity of the processor.

By contrast, in the SHIFT operation, the compiler takes a somewhat conservative approach to save code size. The AVR is an 8-bit processor that can shift one byte by one position in one cycle. Rather than being lost, the former bit of the least or greatest significance is stored in the carry flag. This feature simplifies shifting across multiple bytes, as the shift instruction automatically propagates the carry flag's value into the newly-vacated bit position. Naturally

at the beginning of a multi-word shift operation, one must be sure to clear the carry flag.

Greater shift distances are handled by repeating the shift instruction. To handle multi-word shifting, the compiler emits a subroutine: the shift distance is loaded and used as a loop index. The overhead for subroutine setup, call, and return, along with the housekeeping required for looping is substantial.

In MD6, the number of places l_i and r_i by which our operands need be shifted is not data-dependent. Because the shift distances are known in advance, at the cost of increased code size, one can eliminate loops and subroutine calls. Moreover, we can take advantage of an optimization due to the byte-wise nature of the AVR: we can shift any multiple of eight positions merely by register renaming. In effect, the register file becomes a sort of sliding window: if a word is stored in registers R0-R7, we can shift it right by eight positions by zeroizing R8 and considering the register R1 to represent the least-significant eight bits. Handling the remaining positions *mod* 8 can be achieved in an unrolled loop.

Recall the operation we are computing is

```
y = x XOR (x >> r_i);
return y XOR (y << l_i);
```

To handle the first line, we spend 32 cycles loading x from SRAM, storing a copy in registers before the SHIFT, and finally storing a copy in registers afterward. For the the SHIFT itself, we need $r_i/8$ cycles to zeroize the "free" bytes as part of the shift-by-8 optimization described above. Then $(8 - r_i/8 + 1) \times [r_i \bmod 8]$ cycles to handle the number of bytes to be right-shifted times the number of bits to shift each one. Finally, we need to account for 8 additional cycles for the XOR operation itself.

The second line proceeds similarly with the exception that instead of spending 32 cycles loading and copying data, we need only 16 cycles to store the final result. Plugging the actual values for l_i and r_i into these equations yields a combined total of 2,113 cycles to handle all the shifting in one iteration of the main compression loop. The table below gives the overall timing estimates for a speed-optimized assembly implementation. We estimate that the NLFSR approach consumes 5% more cycles overall than the figures below.

d	r	rc	cycles	time (20MHz)
160	80	1280	742,480	37 ms
224	96	1536	890,976	44 ms
256	104	1664	965,224	48 ms
384	136	2176	1,262,216	63 ms
512	168	2688	1,559,208	78 ms

Figure 4.10: Speed estimates for assembly implementation of MD6 on an Atmel 8-bit processor (one compression function call).

4.5.3.0.1 Whither small processors? When measured by total units sold, 8-bit processors are still the world's most popular. Their low cost continues to make them the processor of choice in embedded systems. The high sales volumes mean that fast-paced innovation continues, especially as vendors offer more features while remaining at relatively low price points. The immense cost pressures dictate that the 8-bit market will remain highly fragmented, with parts being highly optimized for a particular application.

Eight-bit processors remain popular in smart cards, for example. Given their need for a standardized I/O interface, custom hardware accelerators for symmetric and public-key algorithms, and physical tamper-resistance, these devices often have a very different architecture and feature set than the typical microcontroller deployed in a television remote control, for instance. Because of the broad diversity of processor architecture and instruction set, algorithm performance will vary from processor to processor. Simple things like the number of clock cycles needed to read (or operate on) SRAM can make a big difference, making comparisons difficult. But because much of the time in the implementation is spent ferrying data to and from memory, the time needed for a processor's load-operate-store sequence (or equivalent) will make a big difference. The AVR has a relatively fast path to memory (only two cycles), so given slower memory access, an MD6 implementation would be correspondingly slower.

Product roadmaps for 8-bit microcontrollers tend to focus on the integration into the processor of components formerly requiring a separate chip. Obvious examples are on-chip display-driver capabilities, with some processors now directly carrying the circuitry to control a small liquid-crystal display. In addition, the communication features continue to expand. A variety of low-power wired and wireless data transports are now available including CAN, Bluetooth, ZigBee, WiBree and many proprietary schemes. To better support this kind of data I/O, some processors are gaining features like Direct Memory Access (DMA) that speed the flow of data to and from off-chip resources.

Closer to our purposes, some 8-bit microcontrollers like the AVR XMEGA (even those not packaged for inclusion in a smart card) now offer hardware DES and AES accelerators. Naturally, smart card processors have had unique needs for both symmetric and public-key hardware acceleration as well as tamper-resistance for quite some time. The fact that these features are cropping up in more general-purpose processors is a testament to the increasing connectivity even among these smallest of systems.

Ultimately, in a few years' time, we can expect hardware accelerators for the new hash standard to appear in 8-bit processors just as AES accelerators are widely available.

4.6 MD6 Memory Usage

The size of the MD6 state, as implemented, is 15504 bytes. Most of this is for the stack of values: there are 29 levels to the stack, each holds 64 words of data, and each word is 8 bytes; this gives 14,848 bytes just for the stack.

The reference implementation provides for handling the full maximum message length of $2^{64} - 1$ bits. Should there be a smaller limit in practice, a correspondingly reduced stack size could be used. For example, if in an application there was a known maximum message size of 32GB (2^{40} bits), then a stack size of 17 levels would suffice, reducing the necessary size of the MD6 state to roughly 9K bytes.

When $L = 0$, so that MD6 is operating entirely sequentially, the size of MD6 state is even further reduced. In this case, MD6 should be implementable using not much more than the size of one 89-word compression input, which is 712 bytes.

4.7 Parallel Implementations

Computing platforms are quickly becoming highly parallel, as chip manufacturers realize little more performance is available by increasing clock rates. Increased performance is instead being obtained by utilizing increased parallelism, as through multi-core chip designs.

Indeed, typical chips may become very highly parallel, very soon.

Anwar Ghoulum, at Intel's Microprocessor Technology Lab, says "developers should start thinking about tens, hundreds, and thousands of cores now."

Section 4.7.1 shows how the speed of MD6 can be dramatically increased by utilizing the CILK software system for programming multicore computers.

Then Section 4.7.2 shows how the speed of MD6 can be dramatically increase by implementing MD6 on a typical graphics card (which is highly parallel internally).

4.7.1 CILK Implementation

We implemented MD6 for multicore processors using the CILK extension to the C programming language developed by Professor Charles Leiserson and colleagues. CILK began as an MIT research project, and is now a startup company based in Lexington, Mass.

The CILK technology makes multicore programming quite straightforward. The CILK programmer identifies her C procedures that are to be managed by the CILK runtime routines by the `cilk` keyword. She can identify procedure calls to be potentially handled by other processors by the `spawn` keyword. She synchronizes a number of spawned procedure calls using the `sync` statement. Details can be found at the MIT CILK web site[3] or at the company's web site[4].

Our implementation of MD6 in CILK used the layer-by-layer approach (so it assumes that the input message is available all at once). It processes each layer in turn, but uses parallelism to process a layer efficiently.

[3] `http://supertech.csail.mit.edu/cilk/`

[4] `http://www.cilk.com`

Bradley Kuszmaul, Charles Leiserson, Stephen Lewin-Berlin, and others associated with CILK have been extremely helpful in our experiments with MD6 (thanks!).

Using CILK, MD6's parallel performance is the best one can hope for: its throughput increases linearly with the number of processors (cores) available.

We have experimented with 4-core, 8-core, and 16-core machines using our CILK implementation.

The 16-core machine shows best how MD6 scales up with more cores. This machine is a 2.2 GHz AMD Barcelona (64-bit) machine. See Figure 4.11.

Number of cores	Processing speed (MB/sec)
1	40.4
2	121.6
3	202.2
4	270.2
5	338.3
6	407.7
7	474.1
8	539.8
9	607.2
10	674.2
11	740.0
12	805.8
13	875.9
14	940.0
15	1004.0
16	1069.9

Figure 4.11: Speed of MD6 using various numbers of cores on a 16-core machine using CILK.

4.7.2 GPU Implementation

A typical desktop or laptop computer contains not only the main CPU (which may be multicore, as noted earlier), but also a potentially high-performance, highly-parallel graphics processor (GPU).

Such general purpose graphics processing units are beginning to be used as cryptographic processors (e.g. Cool et al [29] and Harrison et al. [43]), since they provide a rich set of general-purpose logical instructions and a high degree of parallelism.

Current GPGPUs trade a traditional cache hierarchy and complex pipeline control for expanded vector-parallel computational resources. For example, the 8800GT (release in 2007) has 112 vector processing elements (PEs) arranged in gangs of 8, forming a cluster of 14 SIMD thread processing units (TPUs).

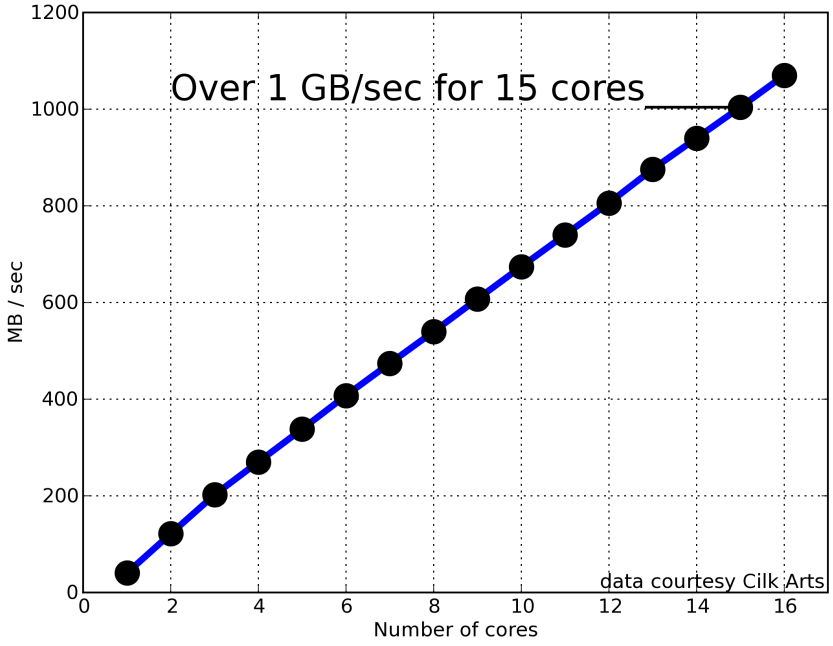

Figure 4.12: Chart of MD6 speed using various numbers of cores on a 16-core machine using CILK.

The TPU operate on blocks of threads each of which must operate on the same vector instruction. If divergent control flow exists across the threads in a TPU thread block, the PEs revert to slow serial execution. TPUs, however, may follow different control paths from one another. Unlike traditional out-of-order processors, which hide high-latency operations with instructions from the same computation thread, the TPUs hide latency by running multiple blocks of threads in parallel. Indeed, the TPU cannot even execute back to back instructions from the same thread block in back to back cycles – the TPU will idle if there are no other thread blocks to execute. Thus, to exploit the resources of the GPU fully, a program must have suffcient vector parallelism to saturate a TPU, limited control flow within the TPU, and enough data-parallel tasks to keep all of the TPUs in the system busy.

MD6 is ideally suited to exploit the parallelism presented by such GPGPU architectures. The 16 steps in a compression round are vector parallel, and the compression function itself has statically determined control flow. Since individual compression functions are data parallel, multiple compression functions maybe run in parallel on the processing units of the GPGPU, thereby achieving high GPU utilization. MD6 can achieve throughputs as high as 610 MB/s on a single GPU,

We tested several GPU implementations by hashing a 512MB block of memory. Performance was determined by measuring the wall clock time between the completion of data initialization until the completion of hashing. In particular, heap memory allocation is not accounted for in the reported performance. Although memory allocation incurs substantial performance overhead, MD6 is likely to be used as part of a library in which the cost of memory allocation is amortized across many hash operations.

The GPU implementation of MD6 operates on the MD6 compression tree in the same manner as the multicore implementation, that is layer by layer (see Section 4.1.1.1). The routine uses the GPU to compress blocks of data, computing a minimum of 64 compression functions in parallel (32 kilobytes) per GPU invocation. Smaller data sizes, including the tip of the compression tree are calculated on the main system processor, which is faster than the GPU if insuffcient parallelism is available. In contrast to the traditional software implementation detailed in Section 4.1.2 which required little manual modification to achieve good performance, a number of transformations must be made to the MD6 reference code in order to achieve any reasonable performance on the GPU. Unfortunately, many of these transformations depend on detailed knowledge of the GPU processor and memory architecture. The following paragraphs will present some details about the transformations required to achieve high GPU performance, but may be bypassed without affecting the remaining discussion.

Running the reference implementation of MD6 on the GPU results in the abysmal compression performance of 3MB/s. The naive parallel decomposition of the code into multiple data-independent compression functions, similar to the algorithm used in CILK, gives a low throughput of 14MB/s. The original compression function is not expressed in a vector parallel fashion; as such, the GPU has many idle processing elements and as a result gives low performance.

Unfortunately, vectorizing the code provides a throughput of only 23 MB/s.

A major performance issue with the original code is that it uses a large array in which each element is written exacly once. While this approach is fine for a general purpose processor with a managed cache as discussed in Section 4.1.2, the GPU incurs a substantial performance penalty each time the on-board DRAM, the only place where such a large array can be stored, is accessed. In MD6 compression, the ratio of computation per load is low and so the GPU is unable to hide the latency of main memory accesses with computation, resulting in severly degraded performance. To avoid this memory access penalty, the GPU implementation uses a smaller wrap-around shift register mapped in the TPU scratch pad memory. Since vector parallel instructions may write beyond the end of the data array, the wrap-around arrays require a halo of memory around them to avoid data corruption. Thus, the shift register is sized at 121 words, which includes a 32 word halo. The wrap-around implementation solves the memory access latency problem, but introduces costly modulo arithmetic to compute array indices. However, the benefit of high memory bandwidth outweighs the cost of modular indexing, and this implementation achieves a throughput of 140MB/s. As an aside, it might seem that increasing the shift register to size 128 would reduce the modulo operator to a simple right shift. However, this optimization results in an array of shift registers that does not fit in the 16KB TPU scratchpad memory.

Although modular arithmetic is needed to compute indices into the reduced shift register, the index progression is fixed and index values can be precomputed and stored in a lookup table in scratchpad memory. This optimization raises GPGPU thoughput to 224 MB/s. However, the table lookups still require modulo arithmetic operator. By removing the modulus operators and statically unrolling the entire compression loop, a performance of 360 MB/s was achieved.

To this point, the GPU kernel has operated on large blocks consisting of a constant number of MD6 compression functions, but by operating the GPU at a finer granularity some speedup can be obtained. However, large scale parallelism is required to obtain good GPU performance. MD6 is no exception - the larger the block processed, the greater the processing effciency. It was empirically determined that 64 parallel compressions was the point at which GPU and CPU performance were roughly equivalent. Support for smaller block sizes increased MD6 throughput to 400 MB/s.

Modern machines have the capability to copy large pieces of physical memory to bus devices, such as the GPU. These direct memory access or DMA transfers are high-bandwidth and may be performed in parallel with computation - either at the GPU or CPU. By enabling GPU DMA, a further speedup to 475 MB/s was obtained for MD6-512. Unfortunately, configuring memory for use with DMA has non-neglidgable cost. However, if MD6 were used in the context of library, this cost would be amortized over many calls to the compression function.

Utilizing these methods on the newer, faster 9800GTX card we can achieve MD6 throughput of 610 MB/s for MD6-512. We also obtain high throughput for shorter hash lengths, as shown in Figure 4.13. In particular for extremely

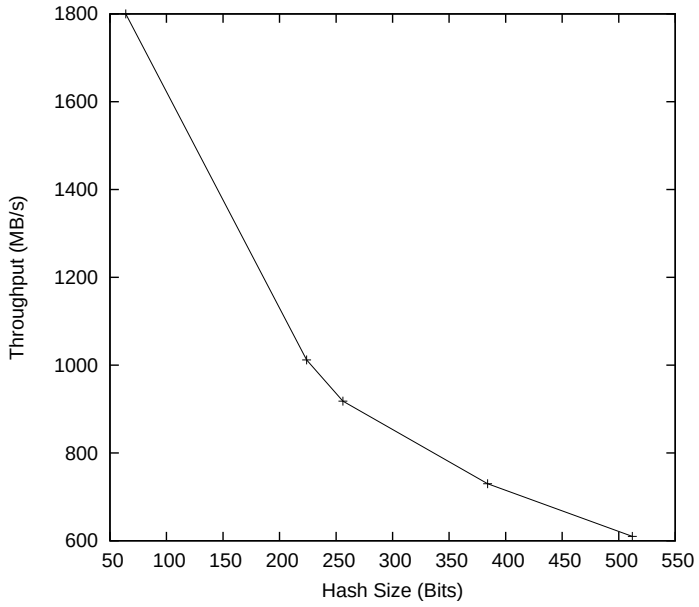

Figure 4.13: GPGPU Throughput Versus Hash Length, 9800 GTX

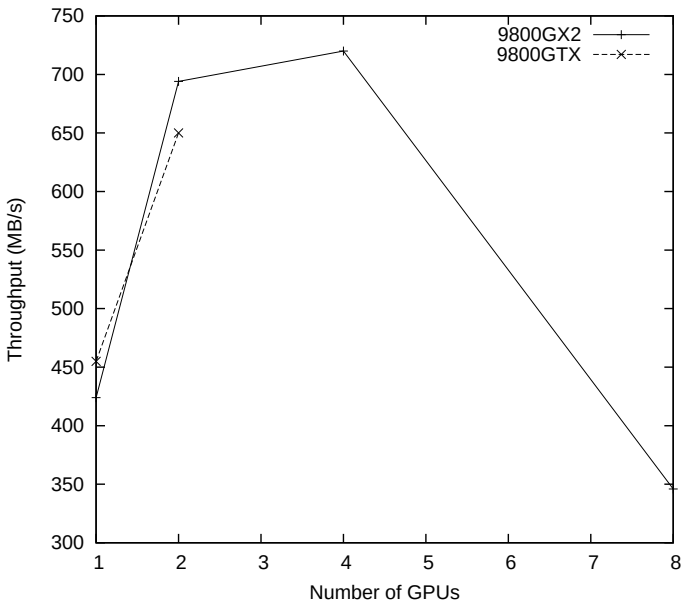

Figure 4.14: GPGPU Throughput Versus Number GPU, 9800GX2

short hash lengths, we obtain throughputs in excess of 1600 MB/s. Although short hash lengths are not crytographically interesting, their high throughput gives some notion of the maximum throughput that can be achieved with future, higher-performance GPUs.

One can install multiple graphics cards in a single desktop to obtain a higher MD6 throughput. The MD6 processing tree can be trivially partitioned and subtrees allocated to various graphics cards within the system, theoretically obtaining a linear speedup in operation. The multi-GPU version of MD6 naively partitions the the MD6 tree into equally sized subtrees and assigns the subtrees to the available GPUs in the system. Once the subtree computation is complete, the host CPU gathers the subtree results and finishes the hash computation.

Figure 4.14 shows the MD6 throughputs achieved for MD6 on two multi-GPU platforms. The first used platform has two 9800GTX+ cards. The second of the platforms has four 9800GX2 cards, each of which has two 9800 series GPUs. The second platform has more aggregate compute, although the individual 9800GTX+ GPUs are superior to the GPUs used in the 9800GX2. For small increases in the number of GPUs, some performance increase is obtained.

The rapidly diminising return for using multiple graphics cards can be attributed to two main causes. The first is the decreased GPU efficiency due to smaller problem size. To achieve good throughput GPUs require hundreds or thousands of threads. If the hash tree is partitioned at too fine a grain, GPUs suffer idle cycles during computation. The second is the competition for memory bandwidth among the cards. Current motherboards multiplex the PCI-E bus when multiple graphics cards are in use, decreasing the effective memory bandwidth to the all cards in the system.

4.8 Summary

MD6 has efficient software implementations on 8-bit, 32-bit, and (especially) 64-bit processors, without complex optimization techniques.

Because of its tree-based mode of operation, MD6 is particularly well-suited for parallel implementions on multicore processors and GPU's. Speeds of many hundreds of megabytes per second are easily obtained, and speeds of 1-2 gigabytes/second are very achievable.

Processor architecture is currently trending to larger numbers homogenous cores—both CPUs and GPUs are following this trend. Becuase the performance individual cores is not improving, the throughput of traditional sequential algorithms, which used to have exponential performance growth, has stagnated. On the other hand, highly parallel algorithms, like MD6, are likely to continue to see improved throughput well into the forseeable future as coarse-grained machine parallelism increases. Approximately nine months passed between the release of the 8800GT GPU (October 2007) and the 9800GTX GPU (July 3008), which runs MD6 nearly 30% faster than the 8800GT; it is unlikely that any existing sequential algorithm would have demonstrated such a marked performance gain over the same time period.

Chapter 5

Hardware Implementations

Cryptographic operations are often computationally intensive – MD6 is no exception. In our increasingly interconnected world, embedded platforms require cryptographic authentication to establish trusted connections with users. However, the limited general-purpose compute that is typically present in such systems may be incapable of satisfying the power-performance requirements imposed on such systems. To alieviate these issues, dedicated hardware implementations are deployed in these devices to meet performance requirements while using a fraction of the power required by general-purpose compute. Recent general-purpose processors [52] have included cryptographic accelerators, precisely because these common operations are compute-intensive. Therefore, any standard cryptographic operation must be efficiently implementable in hardware.

MD6 is highly parallelizable and exhibits strong data locality, enabling the development of efficient, extremely low power hardware implementations.

Section 5.1 first discusses our general hardware implementation strategy, paying particular attention to important hardware design tradeoffs.

We then present implementation results for a number of hardware designs for FPGA in Section 5.2. For example, throughputs as high as 233 MB/s are obtained on a common FPGA platform while consuming only 5 Watts of power.

Section 5.3 provides some discussion of ASIC implementations; this is followed in Section 5.4 with a discussion of MD6 implementations on a custom multi-core embedded system.

5.1 Hardware Implementation

Our hardware implementation matches the version of MD6 submitted for the NIST contest. Therefore, some options and operational modes included in the definition of the algorithm, but not included in the proposed standard, were omitted to reduce the complexity of the hardware. We have obtained a functional FPGA implementation of MD6 that achieves throughput on par with a

modern quad-core general-purpose processor. We also give some intial metrics for an ASIC implementation of MD6 in a 90nm technology. We use the same RTL source to generate both FPGA and ASIC implementations.

All hardware source code is provided under the open source "MIT License" and can be obtained from OpenCores[1].

5.1.1 Compression Function

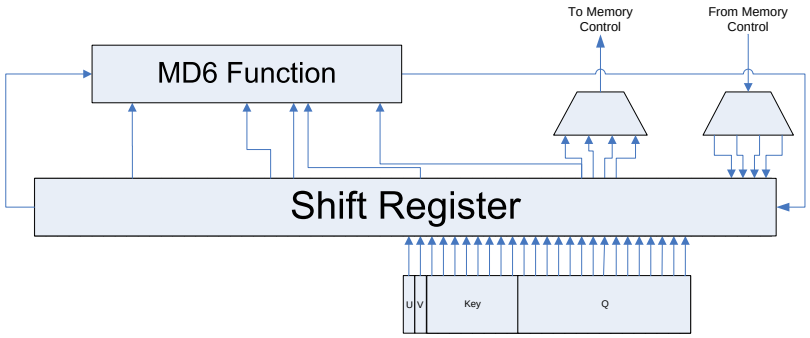

Figure 5.1: Compression Function Hardware

The MD6 compression function is essentially a linear feedback shift register, as depicted in Figure 5.1. To reduce the hardware overhead of the shift register, we constrain shifts to have a constant length, the number of compression steps performed per cycle. Since the shift length is constant, low-cost direct-wire connections between logically adjacent registers can be made. The logic used to compute the MD6 feedback function is similarly wired directly to the correct points in the shift register. Input to and output from the shift register are achieved by tying multiplexors to certain word registers. Some additional logic is used for bookkeeping during operation, but this state logic has limited impact on the operation of the compression function.

The compression occurs in four stages: initialization, data input, compression, and data output. In the initialization stage, the 25-word auxilliary input block is loaded into the shift register. During data input, the data input values are shifted into the shift register. During compression, the step function is applied to the shift register, until the compression operation is completed. During data output, the hash result is streamed out.

The fundamental operation of compression is the application of the step function , 16 of which comprise a round; in hardware, to achieve high throughput, multiple steps and rounds may be composed within a single cycle. Since the first feedback tap of the shift register is located at index 17, up to 16 steps

[1] `http://www.opencores.org`

may be carried out in parallel, without extending the circuit critical path. Of course, multiple 16-step rounds may be further composed to obtain as much parallelism as necessary; this lengthens the critical path, introducing a design tradeoff between throughput and clock frequency.

Since the number of hash rounds is a dynamic parameter, some hash lengths are not computable by some multiple-round per cycle implementations. For example, a 3-round-in-parallel implementation cannot compute a 256 bit hash (104, the number of rounds for this hash size, is not divisible by 3). Although this implementation will compute the correct hash result for all round lengths, the result will have a variable location in the shift register. To generalize such an implementation, additional control and multiplexing logic would be required to collect to the hash result.

5.1.2 Memory Control Logic

The hardware implementation can be viewed as a gang of compression function units orchestrated by a memory controller. The function of the memory controller is simple: it maintains top-level status information and issues memory requests to and from a memory store, or, in the case of a streaming implementation, I/O buffers. As in the case of software, there are two possible implementations of the control logic, as discussed in Section 4.1.

The layer-by-layer approach uses the least amount of logic: fewer memory pointers must be mantained, resulting in slightly less complicated control logic; this implementation is also amenable to exteremly long burst transfers, which may have a power-performance advantage in some systems. However, the layer-by-layer approach requires the entire message to be present in the system at the beginning of the run, requires large amounts of memory, and exhibits poor temporal locality.

As was the case in software, the tree-building approach solves the problems of the layer-by-layer approach at the cost of slightly increased logic area. Morever, the tree building approach introduces the possiblity of a layer cache. The layer cache temporarily stores recently computed portions of the compression tree, that will be used in the near future and some coherence status bits. As compression nodes are computed, their results are stored in the layer cache. Values in the layer cache are loaded as needed for computation. The highest levels of the compression tree need not be stored in the cache – the performance and power impact of caching these values is negligible.

Irrespective of architecture, the memory requests that the memory controller issues take the form of long DMA bursts, which maximize memory bandwidth. The MD6 compression function, by design, expects input data beginning with the least significant word and ending with the most significant word. This word ordering conforms to the burst word transfer order for all bus specifications of which the authors are aware, and greatly simplifies the memory controller implementation, since a re-order buffer is not required to handle burst transfers.

Two versions of the controller were implemented. One implementation issues memory requests in-order. A second version of the controller adds a module

that does coherence bookkeeping between loads and stores, allowing loads and stores to proceed in dataflow rather than in the temporal ordering in which they were issued. This ordering allows some memory requests to be overlapped with computation, since future loads may be serviced before present computation has completed. Overlapped memory accesses provide a great opportunity for system-level parallelism, but typically require extra hadware. In general, load-store bypassing also require a large amount of logic to ensure correctness, however, in the constrained case of MD6 it is fairly straightforward to track inter-level dependencies with a handful of status bits. Our relaxed memory implementation requires some extra buffering to achieve full memory utilization, since requested transactions may not be immediately serviced.

5.2 FPGA

FPGAs are a reprogrammable, highly parallel substrate commonly used for prototyping hardware. Structurally, the FPGA is comprised of a grid of homogenous logic slices surrounded by a configurable interconnect network. Slices are comprised of a lookup table used to implement logic functionality and a handful of registers used to store state. To first order, FPGA implementation area can be estimated by the number of slices required to implement the design. FPGA synthesis tools seamless map Register-Transfer-Level, i.e. Verilog, logic onto the FPGA substrate to obtain a cycle-accurate emulator of the RTL design. FPGAs may additionally contain small SRAM and DSP blocks which are used to reduce the overhead of implementing common but logically complex structures like multipliers. FPGAs are fundamentally more constrained than application specific integrated circuits (ASIC); as a result the silicon area required for an FPGA implementation is larger and the maximum operational frequency is lower than an implementation in a comparable ASIC process. Nonetheless FPGAs serve as an useful platform for evaluating microarchitecture, and most production RTL designs are at least partially verified on an FPGA platform.

Several implementations of MD6 were benchmarked using the Xilinx XUP development platform, which features a Xilinx Virtex-II Pro V30 FPGA. The test system consisted of the MD6 hardware, a PowerPC core (embedded in th FPGA), and a DDR DRAM. The PowerPC served to orchestrate the other hardware and to provide some debugging capacity. All hardware components in the system were run at 100 MHz. All synthesis results were obtained using Synopsis Synplify Pro for synthesis and the Xilinx PAR tool for backend place and route.

The hardware was benchmarked by writing a memory array to the DRAM memory, and then invoking a driver to begin the hardware hash. The result of the hash was then verified against the reference software implementation. All reported benchmark statistics are derived from hardware performance registers embedded in the MD6 FPGA design. Although the harware was tested with many bit lengths to ensure the correctness of the implementation, only whole block results are reported here, since partial blocks effectively take the same

Submodule	Slice Usage	Percent
Step Function	3249	43.1
Shift Register	3286	43.6
S Generation	206	2.7
Control Logic	788	10.5
Total	7529	100

Figure 5.2: Compression Function Area Breakdown, 32-Parallel

Parallel Steps	Cycles per Compression	Slice Usage	$freq_{MAX}$ MHz
1		3778	151.9
2		4070	146.1
4		4233	144.8
8	556.1	4449	150.6
16	249.1	5313	150.3
32	165.7	7529	141.7

Figure 5.3: Compression Function Parallelism

amount of time to process as a full block.

Currently, we have only benchmarked the level-by-level implementation of the MD6 hardware. In general, the performance on the FPGA was quite good, with throughputs as high as 233 MB/s obtained for MD6-512, on par with a quad-core CPU implementation. As shown in Figure 5.4, some inefficiencies exist in processing small messages. Most of these inefficiencies can be attributed to the overhead of filling and draining the processing pipeline. For large messages and low degrees of parallelism, the cycles per compression reaches its asymptote at the point suggested by the level of parallelism in the compression function, indicating that the compression function bottlenecks the system.

Figure 5.2 and Figure 5.3 relate some synthesis results. The shift register

Message Blocks	Cycles per Compression
1	540.0
4	263.6
16	189.3
64	173.6
256	167.9
1024	166.3
2048	165.7

Figure 5.4: Cycles per Compression, MD6-512, 32-Parallel

Hash Size (Bits)	PPC 405 32-bit Risc	32 Parallel IP Core (100Mhz)	
		In-order	Out-of-Order
64	93000	163.2	163.1
224	155000	178.1	163.1
256	167000	181.8	163.2
384	217000	196.7	165.2
512	256000	213.5	165.7

Figure 5.5: Cycles per Compression, various implementations

hardware takes a large portion of all the designs – its size is rougly 3200 slices in all cases. The dominance of the shift register is not a surprise; MD6 requires a long memory which helps provide increased security. As the step parallelism increases, the size of the step function increases linearly with the number of steps performed. The control logic and S generation logic make up a small portion of the total area. In general, the size of the control logic scales linearly with the number of parallel steps, since most of the control logic area is comprised of the I/O muxes. These muxes get larger as more steps are performed in parallel.

Usually, increasing the size of a circuit reduces the maxmimum attainable clock frequency. However, the maximum clock frequency increases with the number of parallelel steps between 4 and 16 steps. This is likely due to the simplification of the logic generating the S and constant propagation for the left and right shift amounts, which repeat every 16 cycles. At 32 parallel steps, a longer critical path is introduced by the second round circuitry, some of which depends upon the result of the first round.

The maximum memory bandwidth of the system is approximately 427 MB/s, simplex. Our non-overlapped controller architecture is unable to fully utilize the available memory bandwidth, since it must sometimes stall waiting for computation to complete. Conversely, our out-of-order memory controller, coupled with highly parallel (32 steps or more) compression functions fully utilizes all available memory bandwidth. This can be seen in Figure 5.5, in which the cycles per compression are nearly the same for MD6-512, MD6-256, and the shorter bit lengths, even though the shorter bit lengths require far less computation than MD6-512.

In our FPGA designs, increasing the number of parallel steps per cycle in an individual compression unit was preferable to introducing a new compression function unit, mostly due to the high overhead of the shift register. Indeed, two compression functions could be implemented in the Virtex-II Pro 30 only if the number of parallel steps in each compression function was constrained to be less than 4. In the FPGA implementation a single compression function with 32-step parallelism was able to fully saturate the memory bandwidth for MD6-512, roughly 427 MB/s.

Our benchmark implementation of MD6 uses a single 100 MHz clock domain, but higher MD6 performance could be obtained by using multiple clock domains.

Algorithm	Slice Usage	$freq_{MAX}$ MHz	Throughput(Mbps)
Whirlpool [78]	1456	131	382
Whirlpool [62]	4956	94.6	4790
Whirlpool [51]	3751	93	2380
SHA-1 [66]	2526	98	2526
SHA-2,256 [89]	2384	74	291
SHA-2,384 [89]	2384	74	350
SHA-2,512 [89]	2384	74	467
SHA-2,256 [28]	1373	133	1009
SHA-2,512 [28]	4107	46	1466
MD5 [45]	5732	84.1	652
MD6-512, 16-Parallel	5313	150.3	1232
MD6-512, 32-Parallel	7529	141.6	1894

Figure 5.6: Various Cryptographic Hash FPGA Implementations

The critical paths in the FPGA implementation run through the system bus and DDR control, rather than through the MD6 hardware. By running the MD6 hardware in a faster clock domain, it is possible to saturate the DDR bandwidth using less hardware.

The PowerPC processor on the Xilinx FPGA is a 32-bit, in-order, scalar RISC pipeline with 8-KB of code and data cache, and can be clocked at 300 Mhz. For the sake of comparing the performance of embedded processors and the MD6 hardware in a similar environment, we measure the performance of the reference MD6 code on the PowerPC embedded in the FPGA. The software implementation requires 256,000 cycles per MD6-512 compression, three orders of magnitude more time than the hardware requires. We estimate that the system implementation without hardware acceleration draws 4.6 Watts, while the system implementation with hardware acceleration draws 5.2 Watts. From a power perspective, this implies that the energy consumption of an embedded processor running MD6 is much greater than that of the hardware accelerated MD6, since the software implementation takes orders of magnitude longer to complete. Power consumption was determined by tying an ammeter to the system power supply and then testing various system configurations.

It is useful to compare MD6 to existing FPGA implementations of other cryptographic hash functions. A number of implementations of Whirlpool, MD5, SHA-1, and SHA-2 are presented in Figure 5.6, although this list is by no means complete. In general, MD6 compares quite favorably with these implementations, both in terms of throughput and area usage, particularly since MD6 requires a longer memory and more computation rounds than any of these hash algorithms.

Parallel Steps	Gate Count	Synthesis Area (μm^2)
1	65595	148946
2	69119	156948
4	74571	169329
8	77691	176414
16	87627	198975
32	114862	260819
48	144717	328610

Figure 5.7: Compression Function PAR Results

Compression Cores	Parallel Steps	Gate Count	Synthesis Area (μm^2)
1	16	105102	238655
2	16	194379	441376

Figure 5.8: Full Implementation PAR results

5.3 ASIC/Custom

Although we have not implemented MD6 in silicon, we have run a common backend toolchain on our design to estimate the silicon implementation area of MD6. Cadence RTL compiler was used for synthesis and Cadence Encounter was used for place and route. We target a 200 MHz operating frequency using the GPDSK 90nm library. Synthesis results are provided in terms of gate count and implementation area.

Figure 5.3 contains the synthesis results for the compression function only. As was the case with FPGA synthesis, the ASIC synthesis area is dominated by the MD6 shift register. The marginal cost per parallel step initially decreases, e.g. for 8 parallel steps each step costs 3924 μm^2, but for 16 parallel steps the cost is 3335 μm^2 per step. As the number of parallel steps gets large (above 16 steps in parallel), the marginal cost per parallel step increases due to the increased size of the gates needed to drive the longer critical path that arises due to computing multiple rounds in parallel.

Figure 5.3 shows synthesis results for the compression function and memory control logic. The system-specific glue logic, i.e. a bus adapter, necessary to attach the ASIC core to the rest of the system is not inicluded; this glue logic is minimal. Although the memory control logic is not seperable from the compression function logic, the area required for the implementation of the memory control logic is 39680 μm^2. Most of the area in the control logic is due to the small I/O buffers required to support overlapped burst transfers.

5.4 Multi-core

To explore the parallelism of MD6 on multi-core chips, we implemented it on the Tile64 Processor chip using the iLib API.

The Tile64 Processor began as an MIT project and is now a startup company, Tilera Corporation[2].

The Tile64 Processor chip has 64 32-bit, 3-way VLIW, general-purpose processor cores with 5 MB of on-chip distributed cache. It boasts 32 Tbps of on-chip interconnect bandwidth, 40 Gbps of I/O bandwidth and 200 Gbps of main memory bandwidth.

The iLib API from Tilera is a library/run-time system that allows programmers to effectively utilize the resources of the Tile64 Processor architecture from C programs. iLib's interprocessor communication features include channels, message passing and shared memory.

We implemented MD6-512, keeping all variables at their default values except for r which was set to 178 and L which was set to 31. The parallelism of MD6 was explored at two levels: the compression function and the tree structure.

There was little parallelsim to be exploited in the compression function. The steps within a round can be computed on different cores, but because of the dependencies between two consecutive rounds a lot of data needs to be communicated among cores. The computation involved in each step is not intensive enough to mitigate the high communication cost. Hence, it was decided to divide the steps within a round among two or four cores. The results are presented in Figure 5.9.

	Cycles	**Speedup**
1 core sliding window	754,290	1.0
1 core circular buffer	2,551,111	0.3
2 cores with channels	464,161	1.6
4 cores with channels	358,302	2.1

Figure 5.9: Results for different implementations of the compression function. Cycles represent the number of cycles utilized to compute one compression function. 1 core sliding window implementation is treated as the base case and speedups are calculated relative to its cycle count.

The base case has the entire compression function implemented on one core as a sliding window. We also provide the results for the circular buffer implementation of the compression function to compare against hardware implementations. As expected, there was a considerable slow down because of the pointer manipulation involved. Both the 2-core and 4-core parallel implementations utilize channels to communicate among processors because they provide the fastest

[2] `http://www.tilera.com`

means of communication for our purposes. Both these implementations show modest speedups.

The tree structure of MD6 presents many opportunites for parallelism. We took two different approaches to exploit this parallelism.

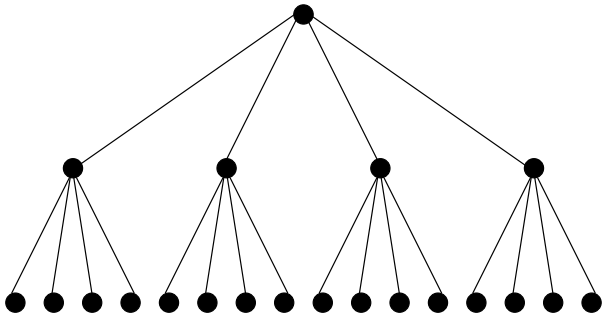

Figure 5.10: 16-4-1 Sub-tree Structure.

In the first approach, we connected 21 cores in a 16-4-1 sub-tree structure as shown in Figure 5.10. The cores were connected using channels. This structure computed three levels of the MD6 tree structure in one iteration. The progression of the computation is shown in Figure 5.11.

In the second approach, we used 17 cores to form a work queue. One core was connected to the remaining 16 cores using bidirectional channels and served as the scheduler. The other 16 cores carried out all the computation. The structure of the connections of cores is shown in Figure 5.12 and the progression of the computation is shown in Figure 5.13.

The results are presented in Figure 5.14. We used two different input sizes: 512 KB and 32 MB. The speedups obtained from both the approaches clearly demonstrate the potential that MD6 has for exploting parallelism, while the subtree structure shows promise for larger input sizes.

5.5 Summary

It is clear from most of the previous software implementation discussion the MD6 has abundant parallelism – extremely high throughputs can be obtained using either multi-core servers or speciality vector processors. Of course, most digital systems do not have the monetary or power budget that these general- purpose systems are permitted to have. In this section, we demonstrate that MD6 may also be efficiently implemented in hardware, and that high throughput may still be obtained even if the system is constrained by a tight area or power budget. Indeed many features of MD6 – its simple, highly-parallel compression function, its amenability to low-power, high-speed burst transfers, its easy computation

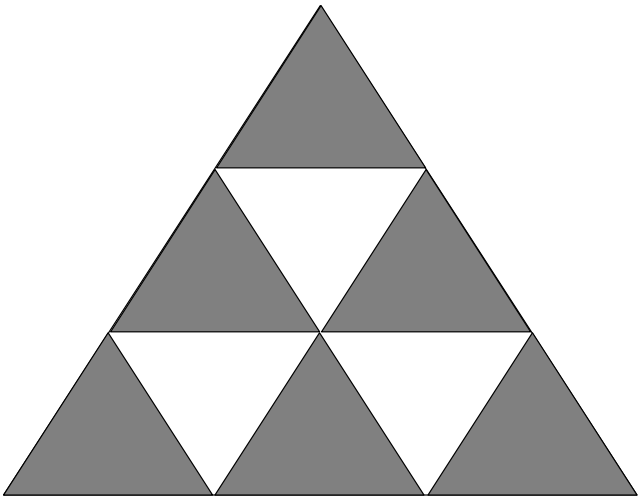

Figure 5.11: Progression of computation for the sub-tree structure. Each gray triangle represents a 16-4-1 sub-tree of the MD6 computation tree structure, which is represented by the large triangle. The gray triangle at the bottom left is computed first, followed by the gray triangle to its right, until the gray triangle at end of the level is reached. The computation then proceeds on to the leftmost gray triangle of the next higher level. This progression of the computation ends at the gray triangle at the root of the MD6 tree.

partitioning – make it an excellent candidate for high-quality cryptographic hashing in situations where power-performance is at a premium.

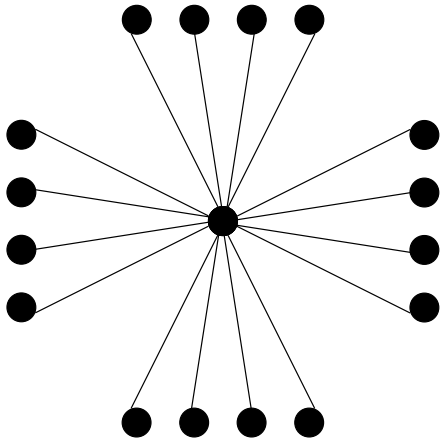

Figure 5.12: Work Queue Model. The middle node represents the scheduler core which assigns work to and collects results from the 16 peripheral cores, represented by the 16 outer nodes. Each peripheral core is connected to the schedular core through a bidirectional channel.

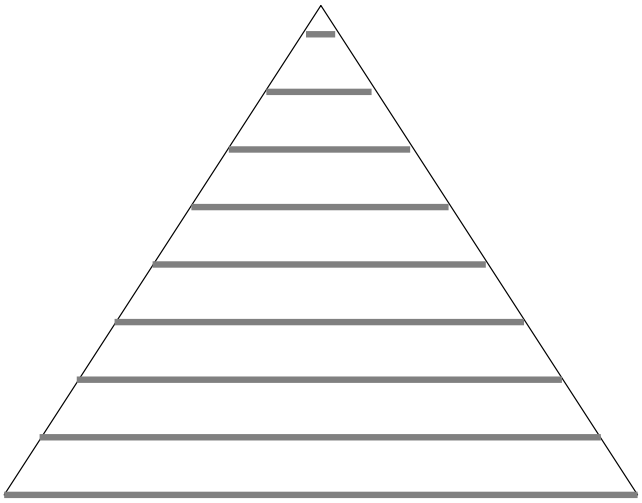

Figure 5.13: Progression of computation for the work queue model depicted by the gray lines. The large triangle depicts the MD6 computation tree structure. Computation starts at the left corner of the bottom gray line and proceeds along the line to the right. When it reaches the end of the gray line, it moves to the left corner of the next higher gray line. This progression of the computation ends at the gray dot at the root of the MD6 tree.

	512KB		**32MB**	
	Processing Rate	**Speedup**	**Processing Rate**	**Speedup**
Sequential	0.34 MB/sec	1.0	0.34 MB/sec	1.0
Sub-Tree	4.54 MB/sec	13.4	5.36 MB/sec	15.8
Work Queue	3.90 MB/sec	11.5	4.30 MB/sec	12.6

Figure 5.14: Results for different implementation of the tree structure. Two different input sizes were used: 512KB and 32MB, and two differnt parallel implementaions were explored: sub-tree and work queue. Sequential implementation is treated as the base case and speedups are calculated relative to its processing rate.

Chapter 6

Compression Function Security

This chapter analyzes the security of the MD6 compression function. Since the compression function is the "heart" of a hash function, it is important to investigate its security from all possible angles. This chapter does so for the MD6 compression function, looking at theoretical foundations, statistical attacks, algebraic attacks, SAT-solver attacks, reductions, etc. MD6 stands up very well; we were not able to discern any significant weakness in the design of the MD6 compression function.

Section 6.1 provides theoretical support for the architecture of the MD6 compression function, which can be viewed both as a blockcipher-based hash function, or a permutation-based hash function. This section proves that the MD6 compression function is indifferentiable from a random oracle—a major confirmation of the soundness of the MD6 compression function design.

Section 6.2 discusses the choice of the constants in MD6, and their security implications. The use of the programs `tapopt.c` and `shiftopt.c` to choose the MD6 tap positions and shift amounts is explained. This section also reviews the "avalanche properties" of MD6, and discusses how MD6 is designed in a way that makes "trapdoors" extremely implausible.

Section 6.3 reviews evidence presented for the collision-resistance of the MD6 compression function.

Section 6.4 similarly reviews evidence for the preimage-resistance of the MD6 compression function, while Section 6.5 presents evidence for the second preimage-resistance of the MD6 compression function.

Section 6.6 discusses the pseudorandomness of the MD6 compression function. This includes results from a standard statistical test suite (`testu01`) as well as results from another heuristic statistical test we designed for testing the MD6 hash function.

Section 6.7 discusses evidence for the unpredictability of the MD6 compression function (thus making it useful for message authentication codes).

Section 6.8 studies a compression function property called "key blinding" that we use in our study of the MD6 mode of operation; this supports the use of the key-blinding property in our analysis of the MD6 mode of operation.

Section 6.9 gives a major result, that the MD6 is not vulnerable to "standard" differential attacks. More precisely, it presents a lower bound on the work needed by an adversary to find a collision in MD6 using a standard differential attack, and show that it is greater than the work required by a "birthday attack", for all digest sizes required by NIST for SHA-3.

Section 6.10 similarly presents a major result on the ability of MD6 to resist linear cryptanalysis. Specifically, it shows that a cryptanalyst following NIST key management guidelines (changing secret keys every two years) would not be able to successfully mount a linear cryptanalytic attack against MD6 (used in MAC mode), because the number of required input/output pairs (2^{210}) is just too large.

Section 6.11 gives some preliminary experimental and heuristic evidence that MD6 is resistant to "algebraic" attacks. These include estimates of the degrees of algebraic normal form polynomials representing the MD6 compression function output bits, as well as some experimental results kindly provided to us by others on algebraic attacks, such as the Dinur/Shamir "cube" attack and somewhat related attacks developed by Aumasson.

Section 6.12 discusses the applicability of SAT solver techniques, and shows experimental evidence that generic SAT solver (minisat) attacks are ineffective on MD6 beyond 11 rounds for finding collisions in MD6 or for inverting MD6.

Finally, Section 6.13 revisits and discusses the issue of the default number of rounds in MD6. In brief, we note that we have seen no evidence of weakness in MD6 for greater than 18 rounds, while MD6 as defined has 96–168 rounds for the required NIST SHA-3 output sizes; MD6 has a very substantial "margin of safety" built-in.

6.1 Theoretical foundations

This section studies the theoretical foundations of the MD6 compression function. It establishes that the overall structure of the MD6 compression function is theoretically sound. That is, if the components of the MD6 compression function are secure, then so is the compression function itself.

We first consider in Section 6.1.1 the perspective that the MD6 compression function is created from an embedded blockcipher. However this approach is unrealistic, as the "encryption key" is public.

Section 6.1.2 thus considers the perspective that the MD6 compression function is based upon a single fixed public random permutation π of $\mathbf{W}^n$.

We show that fixing some portion of π's input to a fixed value, and then truncating the output, yields a compression function that is indifferentiable from a random oracle. This implies that such a compression function will exhibit the properties expected of a random oracle, such as collision-resistance, first-preimage resistance, second-preimage-resistance, etc.

6.1.1 Blockcipher-based Hash Functions

The blockcipher embedded within the MD6 compression function, which we denote here as $E_S(N)$, is implemented by the inner-loop of the compression function. Here N is an 89-word input, and S is the "encryption key" for the encryption operation (it generates the round keys S_0, S_1, ...); see Figure 2.11.

For notational convenience, we define the "chop" function $\chi_\beta : \{0,1\}^* \to \{0,1\}^\beta$, which returns the last β bits of its input, for any β. Thus, the MD6 compression function $f : \mathbf{W}^n \to \mathbf{W}^c$ can then be defined by the equation

$$f(N) = \chi_{cw}(E_S(N))$$

for any $N \in \mathbf{W}^n$.

Blockcipher-based hash functions have been studied extensively recently [20, 19, 79]. In particular, Black et al. [20] prove a negative result: namely, that a highly efficient blockcipher-based hash function cannot be provably collision-resistant, in that a computationally unbounded adversary can find a collision with a small number of queries to the underlying blockcipher (which is available as an oracle to the adversary). Here "highly efficient" means that the blockcipher is called at most once for each compression function call.

Even though MD6 can be viewed as using one block-cipher call per compression function call, the Black et al. negative result does not apply to MD6, because MD6's input block size for the encryption algorithm is large enough to include both the chaining variable and the message input block. This violates the assumptions of their framework and renders their proof inapplicable.

On the positive side, we can show the following simple lemma about this view of the MD6 compression function.

Lemma 1 *If E is indistinguishable from an ideal block cipher, then*

$$f(N) = \chi_{cw}(E_S(N)) \tag{6.1}$$

is indistinguishable from a pseudorandom function.

Here the "encryption key" S is unknown to the adversary, so the encryption key S becomes in effect the PRF key. The security parameter here is c, the length of the final result, not n, the length of the input N. (Here the assumption that E is indistinguishable from an ideal block cipher implies that the length of the key S is sufficiently large.) The proof of the lemma is straightforward, since viewing S as a key makes E_S a pseudorandom permutation, which is indistinguishable from a pseudorandom function (modulo the usual birthday bound), and truncating a pseudorandom function leaves it pseudorandom. However, this simple observation is only marginally relevant because in MD6 the "encryption key" S is public.

6.1.2 Permutation-based hash functions and indifferentiability from random oracles

Instead of viewing the MD6 compression function as blockcipher-based, it is more realistic to view it as permutation-based. Since the "encryption key" S is fixed to a constant, the fixed encryption operation $E_S(\cdot)$ can be modeled as a single fixed public random permutation $\pi(\cdot)$. (See Figure 2.15.)

Since S is public, an adversary can compute both π and π^{-1} easily. So, we need to consider an adversarial model where the adversary has these powers.

Note that in this model the adversary can both invert f and find collisions for f easily, if we don't do something additional. (This is because the adversary can take the c-word output C, prepend $n-c$ words of random junk, then apply π^{-1} to get a valid pre-image for C. He can do this twice, with different junk values, to get a collision.) However, MD6 does have an important additional feature: a valid compression function input must begin with a fixed constant Q. We now proceed to show that this yields a compression function that behaves like a random oracle when π is a random permutation.

Recall that $\mathbf{W} = \{0,1\}^w$ denotes the set of all w-bit words, that f takes n-word inputs, and that $f_Q(x) = f(Q||x)$ denotes the reduced compression function that takes $(n-q)$-word inputs, prepends the fixed prefix Q, and runs f. To make it explicit that in this section we are modeling f and f_Q in terms of a random permutation π on $\mathbf{W}^n$, we will write

$$f_Q^\pi(x) = \chi_{cw}(\pi(Q||x))\,, \tag{6.2}$$

where $\chi_{cw}(y)$ returns the last cw bits of y, and where x is in $\mathbf{W}^{n-q}$.

Let the ideal functionality be represented by $F : \mathbf{W}^{n-q} \to \mathbf{W}^c$, a random oracle with same signature as f_Q^π. We want to show that f_Q^π is indistinguishable from F even when the distinguisher has oracle access to π. This notion, known as indifferentiability, is formalized as follows [59].

Definition 1 *A Turing machine C^G with oracle access to a function G is indifferentiable from a function F if there exists a simulator S such that no distinguisher D can distinguish (except with negligible probability) between the following two scenarios:*

(A) D has oracle access to C and G.

(B) D has oracle access to F and S.

In (B), the simulator program S may invoke F, but it does not know what calls D has made to F. In terms of concrete security, we will say that C^G is (t, q_F, q_S, ϵ)-indifferentiable from F if any distinguisher D making at most q_F queries to C or F and q_S queries to G or S has advantage at most ϵ in distinguishing the above experiments. Further, the simulator S should run in time at most t.

Indifferentiability is a powerful notion: Coron et al. [31] show that if f_Q^π is indifferentiable from F, then F may be replaced by f_Q^π in any cryptographic

applications, with only negligible loss of security (even against adversaries that can invoke π or π^{-1}).

Theorem 1 *If π is a random permutation and Q is arbitrary, the reduced MD6 compression function f_Q^π defined by equation (6.2) is (t, q_F, q_S, ϵ)-indifferentiable from a random oracle F, for any number of queries q_F and q_S, for distinguishing advantage*

$$\epsilon = \frac{(q_S + q_F)^2}{2^{nw}} + \frac{q_S}{2^{qw}} + \frac{q_S q_F}{2^{(n-c)w}} \tag{6.3}$$

and for the running time of the simulator $t = O(q_S nw)$.

Proof: We use the approach of Coron et al. [31], who showed that the indifferentiability framework can be successfully applied to the analysis of hash functions built from simpler primitives (such as block ciphers or compression functions). We note that related results have been obtained by Bertoni et al. [14] in their proof of indifferentiability of "sponge functions."

In our case, because π is a permutation, the oracle G contains both π and π^{-1}, and we need to simulate them both. Slightly abusing notation, we will write S for the simulator of π and S^{-1} for the simulator of π^{-1}. Thus, we need to construct simulator programs S and S^{-1} for π and π^{-1} such that no distinguisher D can distinguish (except with negligible probability) between the following two scenarios:

(A) The distinguisher has oracle access to f_Q^π, to π, and to π^{-1}.

(B) The distinguisher has oracle access to F, S, and S^{-1}.

We define the simulators S, S^{-1} for π, π^{-1} as follows:

1. S and S^{-1} always act consistently with each other and with previous calls, if possible. If not possible (i.e., there are multiple answers for a given question), they abort.

2. To evaluate $S(X)$ where $X = Q||x$: compute $y = F(x)$, then return $R||y$ where R is chosen randomly from in $\mathbf{W}^{n-c}$.

3. To evaluate $S(X)$ where X does not start with Q: return a value R chosen randomly from $\mathbf{W}^n$.

4. To evaluate $S^{-1}(Y)$: return a random N in $\mathbf{W}^n$ which does not start with Q (i.e., from $\mathbf{W}^n \backslash (Q||\mathbf{W}^{n-q})$).

The running time of the simulators is at most $t = O(q_S nw)$. Next, we argue the indifferentiability of our construction. To this end, consider any distinguisher D making at most q_S to queries to S/π and S^{-1}/π^{-1} and at most q_F queries to F/f_Q^π. To analyze the advantage of this distinguisher, we consider several games $G_0, G_1, \ldots, G_7$. For each game G_i below, let $p_i = \Pr(D \text{ outputs } 1 \text{ in } G_i)$. Intuitively, G_0 will be the "real" game, G_7 will be the

"ideal" game, and the intermediate game will slowly transform these games into each other.

Game G_0. This is the interaction of D with f_Q^π, π, π^{-1}.

Game G_1. The game is identical to G_0 except the permutation π is chosen in a "lazy" manner. Namely, one can imagine a controller C_π which keeps the current table T_π consisting of all currently defined values (X, Y) such that $\pi(X) = Y$. Initially, this table is empty. Then, whenever a value $\pi(X)$ or $\pi^{-1}(Y)$ is needed, C_π first checks in T_π whether the corresponding value is already defined. If yes, it supplies it consistently. Else, it chooses the corresponding value at random subject to respecting the "permutation constraint". Namely, if $T_\pi = \{(X_i, Y_i)\}$, then $\pi(X)$ is drawn uniformly from $\mathbf{W}^n \backslash \{Y_i\}$ and $\pi^{-1}(Y)$ is drawn uniformly from $\mathbf{W}^n \backslash \{X_i\}$. It is clear that G_1 is simply a syntactic rewriting of G_0. Thus, $p_1 = p_0$.

Game G_2. This game is identical to G_1 except the controller C_π does not make an effort to respect the permutation constraint above. Instead, it simply chooses undefined values $\pi(X)$ and $\pi^{-1}(Y)$ completely at random from $\mathbf{W}^n$, but explicitly aborts the game in case the permutation constraint is not satisfied. It is clear that $|p_2 - p_1|$ is at most the probability of such an abort, which, in turn, is at most $(q_S + q_F)^2/2^{nw}$.

Game G_3. This game is identical to G_2 except the controller C_π does not choose the values starting with Q when answering the new inverse queries $\pi^{-1}(Y)$. Namely, instead of choosing such queries at random from $\mathbf{W}^n$, it chooses them at random from $\mathbf{W}^n \backslash (Q||\mathbf{W}^{n-q})$. It is easy to see that $|p_3 - p_2|$ is at most the probability that C_π would choose an inverse starting with Q in the game G_2, which is at most $q_S/2^{qw}$.

Game G_4. This game is identical to G_3 except we modify the controller C_π as follows. Recall, there are three possible ways in which C_π would add an extra entry to the table T_π:

1. D makes a query $\pi(X)$ to π, in which case a new value (X, Y) might be added (for random Y). We call such additions forward.
2. D makes a query $\pi^{-1}(Y)$ to π^{-1}, in which case a new value (X, Y) is added (for random X not starting with Q). We call such additions backward.
3. D makes a query $f_Q^\pi(x) = \chi_{cw}(\pi(Q||x))$, in which case C_π needs to evaluate $\pi(Q||x)$ and add a value $(Q||x, Y)$ (for random Y). We call such additions forced.

We start by making a syntactic change. When a forced addition $(Q||x, Y)$ to T_π is made, C_π will mark it by a special symbol, and will call this entry marked. C_π will keep it marked until D asks the usual forward query to $\pi(Q||x)$, in which case the entry will become unmarked, just like all the regular forward and backward additions to T_π. With this syntactic addition, we can now make a key semantic change in the behavior of the controller C_π.

- In the game G_3, when a backward query $\pi^{-1}(Y)$ is made, C_π used to scan the entire table T_π to see is an entry of the form (X, Y) is present. In the new game G_4, C_π will only scan the unmarked entries in T_π, completely ignoring the (currently) marked entries.

We can see that the only way the distinguisher D will notice the difference between G_3 and G_4 is if D can produce a backward query $\pi^{-1}(Y)$ such that the current table T_π contains a marked entry of the form $(Q||x, Y)$. Let us call this event E, and let us upper-bound the probability of E. For each forced addition $(Q||x, Y)$, the value Y is chosen at random from $\mathbf{W}^n$, and the distinguisher D only learns the "chopped" value $y = \chi_{cw}(Y)$. In other words, D does not see $(n-c)w$ completely random bits of Y. Thus, for any particular forced addition, the probability that D ever "guesses" these missing bits is $2^{-(n-c)w}$. Since D gets at most q_S attempts, and there are at most q_F forced values to guess, we get that $\Pr(E) \leq q_S q_F / 2^{(n-c)w}$. Thus, $|p_4 - p_3| \leq \frac{q_S q_F}{2^{(n-c)w}}$.

Game G_5. We introduce a new controller C_F, which is simply imitating a random function $F : W^{n-q} \to W^c$. Namely, C_F keeps a table T_F, initially empty. When a query x is made, C_F checks if there is an entry (x, y) in T_F. If so, it outputs y. Else, it picks y at random from $\mathbf{W}^c$, adds (x, y) to T_F and outputs y. Now, we modify the behaviors of the controller C_π for π/π^{-1} from the game G_4 as follows. In the Game G_4, when a new forward query $(Q||x)$ was made to π, or a new query x was made to f_Q^π, C_π chose a random Y from $\mathbf{W}^n$ and set $\pi(Q||x) = Y$. In the new game, in either one of these cases, C_π will send a query x to the controller C_F, gets the answer y, and then set $Y = R||y$, where R is chosen at random from $\mathbf{W}^{n-c}$.

We notice that the game G_5 is simply a syntactic rewriting of the game G_4, since choosing a random value in $\mathbf{W}^n$ is equivalent to concatenating two random values in $\mathbf{W}^{n-c}$ and $\mathbf{W}^c$. Thus, $p_5 = p_4$.

Game G_6. Before describing this game, we make the following observations about the game G_5. First, we claim that all the entries of the form $(Q||x, Y)$ in T_π, whether marked or unmarked, have come from the explicit interaction with the controller C_F. Indeed, because in game G_3 we restricted C_π to never answer a backward query so that the answer starts with Q, all such entries in T have come either from a forward query $\pi(Q||x)$, or the f_Q^π-query $f_Q^\pi(x)$. In either case, in game G_5 the controller C_π "consulted" C_F before making the answer. In fact, we can say more about the f_Q^π-query $f_Q^\pi(x)$. The answer to this query was simply the value y which C_F returned to C_π on input x. Moreover, because of the rules introduced in game G_4, C_π immediately marked the entry $(Q||x, R||y)$ which it added to T_π, and completely ignored this entry when answering the future backward queries to π^{-1} (until a query $\pi(Q||x)$ to π was made).

Thus, we will make the following change in the new game G_6. When D asks a new query $f_Q^\pi(x)$, the value x no longer goes to C_π (who would then attempt to define $\pi(Q||x)$ by talking to C_F). Instead, this query goes directly to C_F, and D is given the answer y. In particular, C_π will no longer need to mark any of the entries in T_π, since all the f_Q^π queries are now handled directly by C_F.

More precisely, C_π will only "directly" define the forward queries $\pi(X)$ and the backward queries $\pi^{-1}(Y)$ (in the same way it did in Game G_5), but no longer need to worry about defining $\pi(Q||x)$ due to D's call to $f_Q^\pi(x)$.

We claim that the new Game G_6 is, once again, only a syntactic rewriting of the Game G_5. Indeed, the only change between the two Games is that, in Game G_5, T_π will contain some marked entries $(Q||x, R||y)$, which will be anyway ignored in answering all the inverse queries, while in Game G_6 such entries will be simply absent. There is only one very minor subtlety. In Game G_5, if D first asks $f_Q^\pi(x)$, and later asks $\pi(Q||x)$, the latter answer $R||y$ will already be stored in T_π at the time of the first question $f_Q^\pi(x)$. However, it will be marked and ignored until the second question $\pi(Q||x)$ is made. In contrast, in Game G_6 this answer will only be stored in T_π after the second question. However, since in both cases C_π would answer by choosing a random R and concatenating it with C_F's answer y to x, this minor difference results in the same view for D. To sum up, $p_6 = p_5$.

Game G_7. This is our "target" game where D interacts with S/S^{-1} and a true random oracle F. We claim this interaction is identical to the one in Game G_6. Indeed, C_F is simply a "lazy" evaluation of the random oracle F. Also, after all our changes, the controller C_π in Game G_6 is precisely equivalent to our simulators S and S^{-1}. Thus, $p_7 = p_6$.

Collecting all the pieces together, we get that the advantage of D in distinguishing Game G_0 and Game G_7 is at most the claimed value

$$\epsilon \leq \frac{(q_S + q_F)^2}{2^{nw}} + \frac{q_S}{2^{qw}} + \frac{q_S q_F}{2^{(n-c)w}}$$

∎

(In practice, there are other inputs to consider, such as the key input K, the unique ID U, and the control word V. The above proof applies as given, assuming that these inputs are available for the distinguisher to control. This is the correct assumption to make from the viewpoint of the MD6 mode of operation or the viewpoint of other applications using the MD6 compression function.)

Corollary 1 *(Preimage Resistance)*
It is infeasible for an adversary, given access to a random permutation π, its inverse π^{-1} and a random c-word value C, to find any n-word input $N = Q||x$ on such that $f(N) = C$, where $f(N) = \chi_{cw}(\pi(N))$.

Corollary 2 *(Collision Resistance) It is infeasible for an adversary, given access to a random permutation π and its inverse π^{-1}, to find two distinct n-word inputs N and N' such that $f(N) = f(N')$, where $f(N) = \chi_{cw}(\pi(N))$.*

The term "infeasible" in Corollary 1 means complexity 2^{-cw}, and in Corollary 2 means complexity $2^{-cw/2}$.

6.1.3 Can MD6 generate any even permutation?

We conjecture that the structure of the MD6 compression function is sufficiently "expressive" and "robust" that it can express any even permutation of $\mathbf{W}^n$.

Coppersmith and Grossman [30] give related results about block ciphers similar to the Data Encryption Standard. It follows from their results that MD6 can only generate even permutations, since its basic operations can all be formulated as a combination of simpler operations that update a single word of the shift-register state by xoring it with a function of (less than all of) the other words.

Conjecture 1 *Let π be an arbitrary even permutation of $\mathbf{W}^n$. Then there exists a number r of rounds, a sequence $S_0, S_1, S_2, \ldots, S_{rc-1}$ of words, and two sequences $r_o, r_1, \ldots, r_{rc-1}$ and $\ell_o, \ell_1, \ldots, \ell_{rc-1}$ of shift amounts such that the MD6 compression function—not including the final truncation— implements π.*

Note that in the above conjecture the constants S_i and the shift amounts r_i and ℓ_i may change arbitrarily every step.

Because all of the MD6 operations are even permutations of $\mathbf{W}^n$, the group they generate is at most the alternating group rather than the symmetric group. (The alternating group is half the size of the symmetric group, and contains all the even permutations.)

If this conjecture can be proved, it provides evidence that the MD6 compression function is not handicapped by being only able to generate a proper subset of the set of all even permutations of $\mathbf{W}^n$. (If the conjecture is false, however, there may well be no negative implications for the security of MD6, due to the presence of the truncation operation.)

We note (without proof) that the indifferentiability proof of the previous section can be easily modified to handle the requirement that π is restricted to be an arbitrary even permutation, rather than an arbitrary permutation.

We have performed some experiments on reduced-round and reduced-wordsize variants of MD6 to verify this conjecture for those cases. In particular, we have verified that the basic MD6 operations generate the entire alternating group on $\mathbf{W}^n$ when $n = 6$ and $w = 2$. These (rather nontrivial) experiments used the Bratus-Pak method [21]. They provide, we believe, strong support for the above conjecture.

6.1.4 Keyed permutation-based hash functions

The results of the preceeding section are for the"unkeyed" case, where π and π^{-1} are public.

MD6 does have an (optional) key, but it is provided through the input. That is, there is a key input K (of length k words) that is also part of the n-word input. Rather than viewing S as the "key" as in Lemma 1—which wasn't very satisfactory because S is public—we now turn to the way MD6 really keys its compression function: by including K as part of the input.

If f_Q is indifferentiable from a random oracle, as proven in Theorem 1, then using a portion of its bits for a secret key clearly gives something like a "keyed random oracle"—a family of functions, each indifferentiable from a random oracle, with one such function for each key. In other words, the MD6 compression function is then a pseudorandom function with seed K.

6.1.5 m-bit to n-bit truncation

The NIST requirements state (where n denotes the size of the message digest output) in Section 4.A.iii:

> Any m-bit hash function specified by taking a fixed subset of the candidate function's output bits is expected to meet the above requirements with m replacing n. (Note that an attacker can choose the m-bit subset specifically to allow a limited number of precomputed message digests to collide, but once the subset has been chosen, finding additional violations of the above properties is expected to be as hard as described above.)

To respond to this requirement, we note that:

- MD6 was designed to follow the principle of "output symmetry" (see Section 3.11): each output bit is intended to be equivalent to any other output bit in terms of its effect on security.
- The indifferentiability proof given above suggests that MD6 should not suffer in any surprising ways if its output is truncated, since the security of a random oracle does not suffer in any surprising way if its output is truncated.
- MD6 follows the "wide-pipe" strategy suggested by Lucks [54, 55], wherein the internal chaining variables have at least twice as many bits as the final output hash value. The 1024-bit output at the root of the MD6 computation tree is truncated to d bits to obtain the final MD6 output. MD6 was designed to be "truncation friendly."

However, we note that the truncation approach suggested in the NIST proposal is at variance with the MD6 design philosophy, since to change the output digest size for MD6, one also changes the input to every compression function call, in order to avoid the situation where a d-bit hash output and a d'-bit hash output are obviously related to each other in a trivial way.

6.2 Choice of compression function constants

6.2.1 Constant Q

The constant Q is chosen rather arbitrarily as the binary representation of the fractional part of $\sqrt{6}$. (Why 6? Because it is MD6!)

Other values for Q could have been chosen, even 0. Our proofs do not depend on Q having any particular value; merely that it is fixed.

However, it was felt that a "complex" value such as $\sqrt{6}-2$ might have some advantages over an all-zero value; having many 1-bits in Q would force some AND gates to have non-zero inputs early on. Some of the algebraic attacks do seem to suggest that fewer rounds are needed for security if Q is chosen to be non-zero in a "random-looking" way as MD6 does.

6.2.2 Tap positions

The tap positions were computed by a program, `tapopt.c`, which is included with our MD6 submission to NIST.

The program takes as input the desired values for n and c, and produces as output "optimal" values for the tap positions t_0, t_1, t_2, t_3, t_4, and t_5, subject to the constraints that

- $$c < t_0 < t_1 < t_2 < t_3 < t_4 < t_5 = n$$
- The tap positions must all be nonzero modulo c.
- The tap positions must not be equal to each other modulo c.
- The tap positions, other than t_5 must be relatively prime to n. (This is a trivial condition if n is prime.)
- The difference $t_4 - t_3$ must not be equal to the difference $t_2 - t_1$.

The search for an optimal set of tap positions is brute-force; every seqeuence of possible tap positions satisfying the above constraints is considered.

For a given set of tap positions, the computation of the compression function is simulated, and for each value $A[i]$ computed, it is recorded which of the input words $A[0..n-1]$ it "formally depends upon." A word $A[i]$ formally depends upon input $A[j]$ if $i = j$ or if $i > n$ and at least one of the words $A[i-t_0], A[i-t_1], A[i-t_2], A[t_3], A[t_4]$, or $A[i-t_5]$ formally depends upon $A[j]$.

We can say a word $A[i]$ is "complete" if it formally depends upon all input words $A[0] \ldots A[n-1]$.

The "measure" of a set of taps is $i-(n-1)+c$, where i is the largest i such that $A[i]$ is not complete. This value is the least number of steps needed to ensure that all c outputs are formally dependent on all n inputs (and that running more steps won't change this fact).

The optimization for $n = 89$, $c = 16$ takes just a few minutes on a laptop.

As discussed and illustrated in Chapter 9, the program `tapopt.c` can be used to find optimal tap positions for other choices of n and c.

6.2.3 Shift amounts and properties of g

Once the tap positions were determined, the shift amounts were determined by a second program, `shiftopt.c`.

Because none of the shift amounts are zero, the function g is one-to-one (i.e., invertible). Thus, an input change always causes an output change.

Also, as noted in Section 3.9.3, a one-bit change to the input of g will cause from two to four bits of the output of g to change.

Furthermore, the shift amounts were chosen so that in order to get a one-bit output change, at least five bits of input must be changed.

These properties were used in our proof in Section 6.9 of the resistance of MD6 to differential attacks.

6.2.4 Avalanche properties

The tap positions and the shift amounts were both selected by programs "`tapopt`" and "`shiftopt`" in a heuristic but deterministic way that attempts to optimize the influence of each input bit on each output bit in a minimum number of steps.

The tap positions were chosen as described in Section 3.9.1. This "optimization" only paid attention to word-level dependencies, and ignored bit-position (intraword) effects. It was found that after 102 steps (i.e. just over six rounds) each word computed depends, in a formal sense, on each input word.

After the tap positions were chosen, the shift amounts were chosen deterministically from a large pseudorandom sample of shift tables to heuristically optimize the rate of diffusion at the bit-level. The program "`shiftopt`" explicitly considers the constant and linear terms of the algebraic normal form representing each output bit as a function of the input bits, and measures how close each such polynomial is to having approximately half of the maximum number of possible linear terms present. After a total 11 rounds, the selected shift amounts heuristically optimized this measure; the output bits appeared "random" based on this metric.

These optimization methods support the hypothesis that the MD6 compression function begins to look "fairly random" after only 11 rounds.

Once the tap positions and shift amounts were selected, other tests, described in the following sections, help to assess the cryptographic strength of the resulting compression function.

6.2.5 Absence of trapdoors

MD6 was designed to be demonstrably free of trapdoors.

The architecture of MD6 is remarkably simple; such simplicity makes the insertion of trapdoors infeasible.

Moreover, the various constants of MD6, such as $Q = \sqrt{6} - 2$, the tap positions, and the shift amounts, are all computed deterministically by programs

that are available for inspection. The computations and optimizations they implement are directed towards maximizing the cryptographic strength of MD6.

Thus, while there is of course no guarantee that MD6 is free from cryptographic weakness, it is arguably free of maliciously inserted trapdoors.

6.3 Collision resistance

This section studies the security of the MD6 compression function against collision-finding attacks. Since a "birthday attack" requires effort only $O(2^{d/2})$ to find a collision for a hash function with d-bit outputs, the goal here is to find evidence supporting the hypothesis that no attack on MD6 is more efficient, i.e., that MD6 is collision-resistant.

The theoretical foundations for the MD6 compression function presented in Section 6.1 support this hypothesis that MD6 is collision-resistant, since a birthday attack is the most efficient collision-finding attack on a random oracle, and MD6 is indifferentiable from a random oracle (under the assumptions of Section 6.1.2).

There could nonetheless be efficient collision-finding attacks against MD6 — perhaps the underlying assumptions are flawed somehow? It is thus worthwhile to consider known specific collision-finding attacks, to see how well they might work. Studies like this also shed light on the question of the number of rounds needed in MD6.

The two most powerful attacks we know of for finding collisions in a compression function are SAT-solver attacks, and differential attacks. Section 6.12 presents our results on SAT solver attacks; Section 6.9 presents a major result: that MD6 is resistant to collision-finding based on standard differential attacks.

As we shall see, neither SAT solver attacks nor standard differential attacks are able to find collisions better than the traditional "birthday attacks," once the number of rounds is sufficiently large.

6.4 Preimage Resistance

A good hash function will be preimage-resistant: an adversary will need effort roughly 2^d to find any preimage of a d-bit hash.

Again, the theoretical foundations for the MD6 compression function presented in Section 6.1 support the hypothesis that MD6 is preimage-resistant, since a brute-force attack is the most efficient inversion attack on a random oracle, and MD6 is indifferentiable from a random oracle (under the assumptions of Section 6.1.2).

Nonetheless, it is worthwhile studying specific attacks that might defeat the preimage-resistance of MD6. Section 6.12 presents our results on SAT solver attacks. Again, such studies can also help guide the choice of the appropriate number of rounds in MD6.

6.5 Second Preimage Resistance

A good hash function will also be second preimage-resistant: an adversary will need effort $O(2^d)$ to find any preimage of a d-bit hash that is different than an initially given first preimage.

Again, the theoretical foundations for the MD6 compression function presented in Section 6.1 support the hypothesis that MD6 is second preimage-resistant, since a brute-force attack is the most efficient pre-image attack on a random oracle, and MD6 is indifferentiable from a random oracle (under the assumptions of Section 6.1.2).

Of course, a hash function that is not preimage-resistant is also not second preimage resistant. But a hash function may be preimage-resistant but not second preimage resistant.

We do not know of any attacks on the second preimage resistance of MD6 that are not just attacks on its preimage resistance.

In particular, it is not clear how one might adapt a SAT-solver attack on the preimage resistance of MD6 to become an attack on the second preimage-resistance of MD6. Our efforts to devise such an adaptation were unsuccessful.

6.6 Pseudorandomness (PRF)

A good (keyed) hash function will appear to be a pseudorandom function (family). For each key, the hash function will appear to be an independent random function.

The theoretical foundations for the MD6 compression function presented in Section 6.1 support the hypothesis that MD6 is a pseudorandom function family, since that section argues for the hypothesis that the MD6 compression function is indistinguishable from a random oracle, (under the assumptions given in Section 6.1.2), and if you declare part of the input of a random oracle to be "key" and the remainder to be "input" (as MD6 does), you end up with a pseudorandom function family.

Conversely, any adversary that could distinguish MD6 (as keyed) from a pseudorandom function family could be correctly interpreted as an adversary for distinguishing the MD6 compression function f_Q from a random oracle.

We know of no effective attacks for distinguishing (keyed) MD6 compression function from a pseudorandom function family with the same key, input, and output sizes.

The following subsections discuss our attempts to distinguish MD6 from a pseudorandom function family by various statistical tests.

6.6.1 Standard statistical tests

A good hash function will pass any standard statistical tests for randomness, such as the NIST Statistical Test Suite. This section reports our efforts to distinguish the MD6 compression function from a random oracle using two statistical

test suites: the NIST Statistical Test Suite, and TestU01.

Both of these statistical test suites are designed to test pseudorandom number generators. We created a pseudorandom number generator from MD6 by running MD6 in counter mode. Let $h_{r,d}$ denote the MD6 hash function parameterized with with digest size d and r rounds. $h_{r,d}(m)$ is the d-bit output of the parameterized MD6 function on input m. Given a 64-bit seed s, our PRNG generates the sequence $h_{r,d}(s)||h_{r,d}(s+1)||....$ We report the results of the chosen statistical tests for $d = 512$, $s = 0$ and various choices of r.

These statistical tests fail to detect any nonrandomness in MD6 when MD6 has 11 or more rounds.

6.6.1.1 NIST Statistical Test Suite

The NIST Statistical Test Suite is available from `http://csrc.nist.gov/groups/ST/toolkit/rng/index.html`. The test suite contains implementations of 15 different randomness tests.

We ran the NIST statistical tests on MD6 for all $r \in [0, 35]$. For each r, we generated 1000 sequences of 1 million bits. Every test from the NIST Test Suite was run against every bit sequence, generating a list of p-values for every test. To determine if a test was successful at distinguishing MD6 from a random number generator, we tested the output p-values for uniformity using a Kolmogorov-Smirnov test. We also compared the actual number of statistically-significant p-values to the expected number. Running the NIST suite for all $r \in [0, 35]$ took approximately 10 days.

To our surprise, a few of the NIST tests found MD6 non-random beyond 25 rounds. We believe, however, that these tests are incorrectly implemented. We tested this hypothesis by running the NIST tests on SHA-1 (with the full number of rounds), and found that several of the NIST tests also deem SHA-1 non-random.

The following table lists the maximum number of rounds of MD6 that each NIST test is able to distinguish from a random oracle. We have included the results of the tests we believe to be correctly implemented, i.e. the tests passed by SHA-1.

SHA-1 failed the FFT, OverlappingTemplate, RandomExcursionsVariant and Serial tests, and we have omitted the results for MD6 on these tests. We replicated these four tests with the TestU01 suite, however, and found MD6 passes these tests for $r \geq 9$. The following section describes our experiments with TestU01 in more detail.

6.6.1.2 TestU01

We also ran MD6 through the TestU01 suite of tests for random number generators. TestU01 was developed by Pierre L'Ecuyer and Richard Simard and is available from `http://www.iro.umontreal.ca/~simardr/testu01/tu01.html`. Further details on TestU01 are available in [53].

Test Name	r_0
Runs	8
Frequency	8
Maurer's Universal Statistic	7
Rank	7
Longest Run	7
Block Frequency	8
Approximate Entropy	8
Non-overlapping Template	8
Linear Complexity	4
Cumulative Sums	9
Random Excursions	6

Table 6.1: Table showing the maximum number r_0 of rounds of MD6 each of the NIST tests can distinguish from random.

TestU01 contains a wide variety of statistical tests organized into several test *batteries*. We selected three of the provided batteries, SmallCrush, Crush and BigCrush, to test MD6. We also created a test battery to mimic the failing NIST tests by following recommendations in the TestU01 documentation. We ran these tests on MD6 for $0 < r \leq 20$, to determine how many rounds of MD6 these tests can distinguish from random.

The SmallCrush test battery runs in a few minutes, and all of the tests in the SmallCrush test battery pass for $r \geq 9$. The Crush battery requires approximately 8 hours per round tested. All of the tests in the Crush test battery pass for $r \geq 11$. The only test that fails for $r = 10$ is the LongestHeadRun test. The BigCrush test takes approximately 4 days to run, so we only ran it for $r = 10$ and $r = 11$. MD6 passes the BigCrush battery for both $r = 10$ and $r = 11$.

Our NIST test battery contains the following three tests:

- sspectral_Fourier1,
- smultin_MultinomialBitsOver, and
- swalk_RandomWalk1 .

These tests are similar to the four NIST tests SHA-1 failed. We ran the test smultin_MultinomialBitsOver and the test swalk_RandomWalk1 on 1000 1000000-bit strings, that is, under the same conditions as the NIST tests. MD6 passes these tests for $r \geq 9$. We ran the TestU01 sspectral_Fourier1 test on 46 1000000-bit strings, as the test is only valid for a small number of input bit strings. Under these conditions, MD6 passes the sspectral_Fourier1 test for $r \geq 9$. All three of these tests run in a few hours.

Our experiments with both the NIST test suite and TestU01 show that the

standard statistical tests for pseudorandom number generators cannot distinguish MD6 from a random oracle when MD6 has 11 or more rounds.

6.6.2 Other statistical tests

This section reports our efforts to distinguish the MD6 compression function from a random oracle using other statistical tests we have devised or adapted from the literature.

We created several *influence* tests to measure correlations between input and output bit positions. Given an input bit position b and an output bit position c, our tests measure p_{bc}, the probability that flipping bit b causes bit c to flip. For a random oracle, we expect $p_{bc} = \frac{1}{2}$ for any choice of b and c.

Our tests measure p_{bc} by using the following procedure:

1. Choose an input bit position b, an output bit position c, a number of rounds r, and a number of trials n.
2. Use RC4 to generate n random 89-word inputs $x_1, x_2, ..., x_n$ to the MD6 compression function f.
3. For each input x_i, generate a corresponding input x'_i by flipping the bth bit of x_i.
4. Compare the outputs of the r-round compression function $f_r(x'_i)$ and $f_r(x_i)$. Compute the output difference $o = f_r(x'_i) \oplus f_r(x_i)$.
5. Count c_{bc}, the number of times o has a 1 in output position c. Compute $p_{bc} = \frac{c_{bc}}{n}$.

For a given input bit b, our influence test simultaneously computes p_{bc} for all output bits c. To test if MD6 behaves significantly differently from a random oracle, we compared the distribution of the measured p_{bc} values with the expected distribution from a random oracle. In a random oracle, the p_{bc} values are drawn from a binomial distribution with $p = \frac{1}{2}$ and n trials. For large n, this distribution is essentially normal with mean $\frac{1}{2}$ and standard deviation $\frac{1}{2\sqrt{n}}$. We used an Anderson-Darling test and a chi-square test to measure the probability of our p_{bc} values being drawn from this theoretical distribution.

Table 6.2 reports the results of running the above test on input bit 3648 with 1000000 trials. We chose this bit because it is one of the worst-case inputs; it is the first bit of the 57th input word, which is the last word to be incorporated into the hash function computation. Therefore, we expect the influence of bits in the 57th word to be more detectable than the influence of bits in any other word.

An Anderson-Darling A^{*2} score above 1.035 is significant at the $P = 0.01$ level. Our χ^2 test results in a Z-score, for which a score above 2.57 is significant at the $P = 0.01$ level. These results show that the basic influence test cannot differentiate between MD6 and a random oracle beyond 10 rounds. Running this test required only a few minutes.

r	Mean	Standard Deviation	Anderson-Darling A^{*2}	χ^2 Z-Score
1	0.00000	0.000000	DNE	22627394.37055
2	0.00098	0.022112	DNE	22583200.31969
3	0.00880	0.070245	DNE	22284642.00478
4	0.04094	0.120308	269.07229	20383323.51708
5	0.10784	0.165396	124.15361	16395698.58622
6	0.19551	0.187628	42.07427	11577960.41165
7	0.32541	0.179635	29.50064	5679459.06660
8	0.43923	0.110198	123.03832	1433321.38957
9	0.49252	0.028513	262.46513	78630.41306
10	0.49984	0.001440	138.43952	167.39026
11	0.50001	0.000515	0.29285	1.43492
12	0.50002	0.000495	0.84664	-0.40540
13	0.50001	0.000506	0.47437	0.59054
14	0.50000	0.000509	0.30570	0.79043
15	0.50001	0.000494	0.25221	-0.54828
16	0.49998	0.000488	0.31075	-1.01786
17	0.49998	0.000503	0.63886	0.33177
18	0.49998	0.000507	0.51248	0.64595
19	0.50001	0.000497	0.22655	-0.29475

Table 6.2: Table of test statistics for the influence test on reduced-round MD6 compression functions. $A^{*2} > 1.035$ or $\chi^2 > 2.57$ are significant at the $P = 0.01$ level.

We ran another influence test called the dibit influence test. This test is similar to the previous influence test, except it operates on pairs of adjacent bits, called *dibits*. A dibit can take on 4 values: $00, 01, 10, 11$. The intuition behind this test is that adjacent bits tend to stay together through the MD6 compression function, so adjacent input bits may have undue influence on adjacent output bits.

For this test, we treated the input to the compression function as an array of 89 words of 32 dibits. We followed the same procedure as the previous influence test, except we created 3 inputs from each random input x_i (since there are 3 possible ways an input can differ in one dibit position). In the final step, we counted each of the 4 possible output difference patterns independently, resulting in 4 output counts, c_{00}, c_{01}, c_{10} and c_{11}. For large n, the c values of a random oracle are normally distributed with mean $\frac{n}{4}$ and standard deviation $\sqrt{\frac{3}{16n}}$. We again used an Anderson-Darling test and a χ^2 test to compare the measured c values to the expected distribution.

Table 6.3 reports the results of the first dibit of the 57th word, for the same reason for using the first bit of the 57th word in the standard influence tests. The following table shows the most and least significant dibit pairs for each of the included rounds.

An Anderson-Darling A^{*2} score above 1.035 is significant at the $P = 0.01$ level. Our χ^2 test results in a Z-score, for which a score above 2.57 is significant at the $P = 0.01$ level. Our results show that the dibit influence test cannot differentiate between MD6 and a random oracle beyond approximately 10 rounds. Although there are some statistically significant results for beyond 10 rounds, around 2.4 statistically significant scores are expected since there are a total of 240 measurements for each statistic. Running this test took only a few minutes.

Our work with the influence test and its variants shows that these tests cannot distinguish MD6 from a random oracle beyond 10 rounds.

6.7 Unpredictability (MAC)

A good (keyed) hash function will also be unpredictable, and thus be usable as a MAC. Unpredictability is implied by pseudorandomness, but is potentially a weaker condition.

We have argued above for the pseudorandomness of the MD6 compression function. We do not have further evidence specific to unpredictability.

6.8 Key-Blinding

In Section 7.6, we set out to prove that the overall MD6 hash function has the property of unpredictability, assuming that the compression function used also has this property. Unfortunately, we were not able to provide such a proof with only that base assumption; however, if we make an additional non-trivial assumption about the compression function we are able to prove the unpredictability of MD6.

The idea for the following is that the flag bit z which indicates that the compression function application is the final one in the computation of MD6 should "blind" the key values. That is, if we are given oracle access to two functions defined as follows

$$f_Q^0(\cdot) = f_Q(z = 0, K = \kappa_0, \cdot)$$
$$f_Q^1(\cdot) = f_Q(z = 1, K = \kappa_1, \cdot)$$

then we should not be able to guess whether κ_0 and κ_1 are equivalent or not with any significant advantage. The intuition behind this assumption is that we need f_Q to behave in a "pseudorandom" fashion, but only on the z bit of the input.

Definition 2 (Key-Blinding Assumption) *Let D be a distinguisher given oracle access to two functions f_Q^0 and f_Q^1 that map $\mathbf{W}^k \times \mathbf{W}^{n-q-k-1} \rightarrow \mathbf{W}^c$. We define its advantage as follows.*

$$\boldsymbol{Adv}_{D,f_Q}^{blind}(t,q) = \left| \Pr\left[\begin{array}{c} \kappa_0 \xleftarrow{\$} \mathbf{W}^k; \kappa_1 \xleftarrow{\$} \mathbf{W}^k; b \xleftarrow{\$} \{0,1\}; \\ a \leftarrow D^{f_Q(z=0,\kappa_0,\cdot), f_Q(z=1,\kappa_b,\cdot)} \end{array} : \quad a = b \right] - \frac{1}{2} \right|$$

r	in	out	mean	std dev	Anderson-Darling A^{*2}	χ^2 Z-Score
1	1	0	1.00000	0.000000	DNE	47999984.0000
1	1	1	0.00000	0.000000	DNE	5333317.33333
2	2	0	0.99804	0.031241	197.39955	47833178.56519
2	3	1	0.00097	0.015569	DNE	5312484.05495
3	2	0	0.98387	0.096617	181.11400	46754145.47520
3	3	2	0.01050	0.064672	DNE	5251459.45638
4	1	0	0.92969	0.161478	113.54763	41647769.59913
4	3	2	0.04078	0.095791	112.42278	4518137.30509
5	2	0	0.82394	0.233163	48.31323	32748717.20545
5	3	2	0.09269	0.111891	46.85176	3179889.04078
6	2	0	0.69010	0.263053	13.72142	22432777.23596
6	3	2	0.14815	0.104830	26.92808	1822935.90528
7	2	0	0.49774	0.251650	17.06451	10641313.29884
7	3	2	0.20608	0.078529	46.12267	690808.84310
8	2	0	0.33130	0.149265	69.57594	2465222.03179
8	3	2	0.24315	0.030481	109.51572	83269.25936
9	2	0	0.25928	0.036741	DNE	122520.00014
9	3	2	0.24997	0.003609	DNE	1095.64469
10	2	0	0.25020	0.001607	DNE	207.90828
10	3	1	0.24998	0.000435	0.65544	0.21589
11	1	2	0.25001	0.000431	0.33894	-0.13039
11	3	1	0.25001	0.000450	0.20539	1.30168
12	1	3	0.25003	0.000457	0.29739	1.91450
12	2	1	0.24998	0.000433	1.01729	0.01331
12	2	2	0.25002	0.000428	1.06135	-0.33905
13	2	0	0.24998	0.000455	0.76901	1.66687
13	2	1	0.25003	0.000434	0.42587	0.18333
14	2	0	0.25000	0.000470	0.58531	2.88009
14	3	1	0.25000	0.000433	0.22131	0.03616
15	1	3	0.25002	0.000401	0.17983	-2.26640
15	2	0	0.24999	0.000434	0.19632	0.07774
16	3	1	0.24999	0.000435	0.18534	0.19229
16	3	3	0.25002	0.000408	0.39370	-1.73557
17	1	2	0.24997	0.000431	0.25114	-0.09631
17	2	2	0.24997	0.000424	1.03184	-0.63054
17	3	0	0.25001	0.000462	0.13665	2.20993
18	2	3	0.24999	0.000434	0.29173	0.08666
18	3	3	0.25004	0.000460	0.26491	2.21393
19	2	1	0.25001	0.000396	0.65036	-2.60010
19	2	3	0.25001	0.000433	0.39709	0.00462

Table 6.3: Table summarizing test results of the dibit influence test on reduced-round MD6 compression functions. For each number of rounds, the table shows the most and least significant input and output dibit values. $A^{*2} > 1.035$ or $\chi^2 > 2.57$ are significant at the $P = 0.01$ level.

The goal of D is to try to determine the value of b, i.e. whether f_Q^0 and f_Q^1 use the same key or not. We can define the overall insecurity of the key-blinding property of f_Q to be

$$\boldsymbol{InSec}_{f_Q}^{blind}(t,q) = \max_D \left\{ \boldsymbol{Adv}_{D,f_Q}^{blind}(t,q) \right\},$$

where D is given resource constraints of (t,q).

Given our work above supporting the proposition that MD6 is a pseudorandom function, the restriction of this proposition to the weaker proposition that MD6 satisfies the key-blinding assumption appears quite reasonable.

6.9 Differential cryptanalysis

Differential cryptanalysis was pioneered by Biham and Shamir [15] in the early 90s, and it was the first type of sophisicated analytical attacks on block ciphers including the Data Encryption Standard. Subsequently, Preneel et al. [81] proposed the differential cryptanalysis of hash functions based on block ciphers.

Differential attacks have in fact been surprisingly successful and effective against a number of hash functions. MD4, RIPEMD, and 3-pass HAVAL [93, 97], MD5 [95], SHA-0 [27, 16, 17, 96] and SHA-1 [17, 94] have all fallen. Professor Xiaoyun Wang has led these attacks.

In this section, we analyze the security of MD6 with respect to differential cryptanalysis. As in most security analysis, our goal is two-fold—we hope to provide both "attacks (upper bounds)" and "lower bounds". Our major result is a lower bound showing that there is no standard differential attack against MD6 that is more effective than a naive "birthday" attack.

The rest of the section is organized as follows. In Section 6.9.1, we describe some useful definitions in differential cryptanalysis. In Section 6.9.2 we analyze differential properties of the MD6 step function by analyzing individual operations. Throughout our discussions, we will pay special attention to definitions and properties that are important in the later lower bound analysis. In Section 6.9.3, we prove a lower bound on the workload of any differential-style collision attacks. Finally, in Section 6.9.4, we present some preliminary results related to upper bound analysis.

6.9.1 Basic definitions and assumptions

Differential cryptanalysis of block ciphers falls into the category of chosen plaintext attacks. The basic idea is that two plaintexts with a carefully chosen difference can encipher to two ciphertexts such that their difference has a specific value with non-negligible probability, and by taking sufficiently many plaintext/ciphertext pairs and analyzing their differences, certain bits of the secret key may be revealed.

In the context of a (keyless) hash function, the goal of differential cryptanalysis is to find message pairs with an appropriately chosen difference such that

the pair of hash function outputs would have no difference—meaning a collision. If the probabilty for such collisions is non-negligible, then by hashing enough input messages, a collision of the hash function can be found.

The first step of differential cryptanalysis is to define a proper measure of differences relating the computations, and the choice of differences may vary, depending on the mathematical operations involved in the hash function or block cipher. The most commonly used measure of difference is exclusive-or. Let A and A' be a pair of values. Their difference is defined to be $\Delta A = A \oplus A'$. So an exclusive difference identifies individual bit positions for which the pair of values differ.

A differential path for a given hash function is the set of differences between the pair of inputs, all corresponding intermediate states, and the final hash outputs. Since MD6's state variables form a simple sequence, the differential path for the MD6 compression function can be expressed concisely as

$$\{\Delta A_i\} \text{ for } i = 0, ...t + n - 1.$$

A differential path with the property that $\Delta A_i = 0$ for $i = t + n - c, ..., t + n - 1$ is called a collision differential path for the compression function.

The most important property of a differential path is its associated probability. The probability of the differential path in step i, denoted by p_i, is defined to be the probability that the pair of outputs (from step i) follows the differential path given that the pair of inputs (to step i) satisfy the difference specified by the differential path. So we can express p_i as

$$p_i = prob[\Delta A_i | \Delta A_{i-t_j}, j = 0, 1, ..., 5].$$

The input difference $\{\Delta A_{i-t_j}, j = 0, 1, ..., 5\}$ and the output difference ΔA_i, together with its associated probability p_i, are often referred to as a differential characteristic of step i. The total probability p of the entire differential path is the product of probability p_is from individual steps, assuming that computations in individual steps are independent.

While the definition seems fairly straightfoward, we need to pay special attention to a few issues and implicit assumptions in the analysis of differential paths and their probabilities.

First, in most existing work on differential cryptanalysis of hash functions and block ciphers, it is commonly assumed that, after a certain number of rounds, the output from each step appears random and the computations in different steps are independent of each other. Such randomness and independence assumptions are made in terms of statistical properties of the underlying function, and they are generally necessary to carry out the probability analysis. (The same assumptions are needed for linear cryptanalysis as well.)

More specifically, in our differential cryptanalysis of MD6, the probability p_i is computed by assuming that the pair of inputs to step i are chosen at random with the only constraint that they satisfy the given difference. In addition, it is assumed that computations in different steps are independent, and so the probability p_is can be multiplied when computing the total probability p.

We remark that, for MD6, the above assumptions are well supported by our statistical analysis results. Recall that in Section 6.6, we show that MD6 passes a variety of statistical tests after 10 rounds.

Next, the workload of a differential collision attack (using a given differential path) is computed as the inverse of some probability p', which can generally be much larger than p for two reasons. First, by choosing appropriate inputs to the hash function, we can force $p_i = 1$ for a number of steps at the beginning of the computation (message modification). Second, for given input difference and output difference, there can potentially be multiple differential paths (with different intermediate differences ΔA_i) that satisfy the input and output constraints. The overall probability would then be the sum of the probabilities for these individual paths.

Lastly, we introduce the notion of differential path weight pattern. Let $\{\Delta A_i\}$ be a differential path of MD6. The corresponding differential path weight pattern is the sequence $\{D_i\}$, where

$$D_i = |\Delta A_i|$$

is the Hamming weight of ΔA_i. That is, D_i is the number of bits that differ between A_i and A'_i.

Differential path weight pattern turns out to be a useful notion in the security analysis of MD6. As we will see, it allows us to separate the effect of tap positions and shift amounts with respect to differential cryptanalysis. Roughly speaking, the tap positions help to propagate the differences forward from one step to the next, while the shift amounts help to spread the differences within a word. Since the sequence $\{D_i\}$ mainly reflects how the differences propagate forward, it facilitates us to study the effect of the tap positions without paying too much attention to how the bit differences line up within a word. This also greatly simplifies our analysis.

6.9.2 Analyzing the step function

The MD6 step function consists of three operations: XOR, AND, and the g operator. We will analyze differential properties of each operation in terms of how a difference and its Hamming weight propagates from the input to the ouput. Then, we will study differential properties of the step function and derive some inequalities that will be useful in the later lower bound proof.

For ease of discussion, we first introduce some notations. For each operation to be studied, we use uppercase letters X, Y, Z to denote the w-bit inputs and output. We use $\Delta X, \Delta Y, \Delta Z$ to denote the differences and D_X, D_Y, D_Z to denote the Hamming weight of these differences. We use lowercase letters x, y, z to denote a single bit in the w-bit words.

6.9.2.1 XOR gate

Differential properties of XOR is straightforward. In particular, the equation $\Delta Z = \Delta X \oplus \Delta Y$ holds with probability one. In terms of the Hamming weight

of the difference, we have

$$\max(D_X, D_Y) - \min(D_X, D_Y) \leq D_Z \leq D_X + D_Y.$$

6.9.2.2 AND gate

We can view the AND operation between two w-bit words as a layer of w independent AND gates, with each AND gate taking two input bits x and y and producing one output bit z. The differential behavior of the AND gate depends on its input differences Δx and Δy. We consider the following two cases.

- If $\Delta x = \Delta y = 0$, then $\Pr[\Delta z = 0] = 1$. We will call this AND gate "inactive".
- If $\Delta x = 1$ or $\Delta y = 1$, then $\Pr[\Delta z = 0] = \Pr[\Delta z = 1] = 1/2$. We will call this AND gate "active". [1]

In terms of the Hamming weight of the difference, we have

$$0 \leq D_Z \leq D_X + D_Y.$$

The notion of active AND gates plays an important role in the later lower bound proof. In particular, an active AND gate always contributes a probability of 1/2 to the overall probability of the differential path, no matter what the output difference of the AND gate is. Since the AND operation is the only non-trivial operation in terms of differential probabilities[2], the total number of active AND gates in the differential path is closely related to the total probability of the path.

In a way, the role of AND gates in MD6 resembles that of S-boxes in AES. The main distinction is that an AND gate operates at the bit level in MD6 while an S-box operates at the byte level in AES.

6.9.2.3 g operator

The $g_{r,\ell}$ operator provides intra-word diffusion by mixing up data within a word. Let $Z = g_{r,\ell}(X)$. We know that $\Delta Z = g_{r,\ell}(\Delta X)$ holds with probability one.

It is easy to derive an upper bound on D_Z. Since the combination of one shift and XOR can at most double the number of differences, we have

$$D_Z \leq 4D_X. \tag{6.4}$$

Lower bound analysis on D_Z is more interesting, and it relates to the design choices for the shift amounts. More specifically, each pair of shift amounts (r, ℓ) in MD6 were chosen in such a way that

$$D_X \leq 4 \text{ would imply } D_Z \geq 2. \tag{6.5}$$

[1] We assume that the inputs (x, y) and (x', y') are chosen at random with the constraint that they follow the required difference.

[2] That is, probabilities associated with the input and output differences are not always zero or one.

In other words, in order for the output to have only a single bit difference, the minimum Hamming weight of the input difference is at least 5. The main purpose of such a design choice is to prevent an adversary from constructing differential paths of MD6 with very low Hamming weight in every step. For example, given Inequality 6.5, it would be impossible to construct a sparse differential path in which $D_i = |\Delta A_i|$ is at most one for all i.

For $D_X > 4$, we do not have any non-trivial lower bounds on D_Z other than it has to be positive, since non-zero input difference would imply non-zero output difference. So $D_X > 4$ would imply that $D_Z \geq 1$.

In a way, the role of the g operator in MD6 is similar to that of the MDS matrix in AES. It effectively imposes a non-trivial lower bounds on the Hamming weight sum of the input and output difference. The main distinction is g operates at the bit level while the MDS operates at the byte level.

6.9.2.4 Combining individual operations

We have studied differential properties of the three operations in the MD6 step function, and the results are summarized in Figure (6.1) and Figure (6.2).

operation	output difference ΔZ	probability
$Z = X \oplus Y$	$\Delta Z = \Delta X \oplus \Delta Y$	1
$z = x \wedge y$	if $\Delta x = \Delta y = 0$, then $\Delta z = 0$	1
	if $\Delta x = 1$ or $\Delta y = 1$, then $\Delta z = 0$ or 1	1/2
$Z = g(X)$	$\Delta Z = g(\Delta X)$	1

Figure 6.1: Differential characteristics for $\oplus$, $\wedge$, and g. Note that for $\wedge$, the result is for an AND gate at the bit level.

operation	upper bound	lower bound
$Z = X \oplus Y$	$D_Z \leq D_X + D_Y$	$D_Z \geq \max(D_X, D_Y) - \min(D_X, D_Y)$
$Z = X \wedge Y$	$D_Z \leq D_X + D_Y$	$D_Z \geq 0$
$Z = g(X)$	$D_Z \leq 4D_X$	$D_Z \geq 2$, if $0 < D_X \leq 4$ $D_Z \geq 1$, if $D_X > 4$

Figure 6.2: Hamming weight of differential characteristics for $\oplus$, $\wedge$, and g.

In what follows, we analyze how the input/output differences for individual operations within a step can be joined together to form a differential characteristics for the step with non-zero probability. We pay special attention to how the Hamming weight of the differences propagate from input to output.

First, we decompose the step function into two sub-steps:

$$\begin{aligned} X &= A_{i-t_0} \oplus A_{i-t_5} \oplus (A_{i-t_1} \wedge A_{i-t_2}) \oplus (A_{i-t_3} \wedge A_{i-t_4}), \\ A_i &= g(X). \end{aligned}$$

Using the inequalities in Figure (6.2), we can derive upper and lower bounds on $D_X = |\Delta X|$, in terms of the Hamming weight of the six input differences in step i. We obtain the following two inequalities.

$$D_X \leq UB_X = \sum_{k=0}^{5} D_{i-t_k}, \tag{6.6}$$

$$D_X \geq LB_X = \max(D_{i-t_0}, D_{i-t_5}) - \min(D_{i-t_0}, D_{i-t_5}) - \sum_{k=1}^{4} D_{i-t_k}. \tag{6.7}$$

The above two inequalities together define a range for D_X. Given D_X, the Hamming weight of the output differences $D_i = |\Delta A_i|$ follows the constraints given in Figure (6.2).

We remark that by focusing on the Hamming weight rather the actual value of the differences, we avoid the potential complication of analyzing how individual bit differences can line up properly from one operation to another. We lose some accuracy in the analysis since we can only obtain a range of possible values for the Hamming weight. Nevertheless, we will see that the approach not only simplies the analysis but also greatly reduces the complexity of searching for valid differential path weight patterns, thereby making a computer-aided search possible.

6.9.3 Lower bound proof

6.9.3.1 Goal and approach

The goal of our analysis is to prove a lower bound on the workload of any collision search attack that utilizes standard differential techniques. More concretely, we aim to prove that the probability associated with any valid collision differential path of the MD6 compression function is at most $2^{-d/2}$, where d is the length of the hash output in bits. This would imply that the workload of a differential attack is at least $2^{d/2}$, which is the theoretical bound from the birthday paradox.

Before we present all the technical details of our lower bound proof, we provide an outline on how we tackle the above problem by reducing it to relatively easier problems that can be solved by a combination of theoretical and computer-aided analysis.

First, based on the discussion in Section 6.9.2.2, we know that each active AND gate in a differential path contributes a probability of 1/2. Hence, if we could show that the total number of active AND gates in any valid differential path of MD6 is at least $d/2$, then it would imply that the probability associated with the path is at most $2^{-d/2}$.

Second, we analyze the relationship between the number of active AND gates and differential path weight patterns. This reduces the problem of counting active AND gates to the problem of finding valid differential path weight patterns.

Lastly, we use computer-aided method to search for valid differential path weight patterns up to s rounds (for small s) and then extend the results to the full r rounds of MD6. Putting in concrete terms, if we use AAG_s to denote the minimum number of active AND gates in s consecutive rounds in any valid differential path weight pattern of MD6, then we have

$$AAG_r \geq AAG_s \times \lfloor r/s \rfloor. \tag{6.8}$$

For example, for output size $d = 512$, the total number of rounds $r = 168$. If we could show that $AAG_{10} \geq 16$, then we would be able to prove that $AAG_{168} \geq 16 \times \lfloor 168/10 \rfloor = 256$.

The rest of the section is devoted to the lower bound proof that we just outlined. We will also discuss some potential routes to improve the analysis and compare the analysis with that of the AES.

6.9.3.2 Counting the number of active AND gates

Suppose we are given a differential path weight pattern for up to s rounds. In this section we show how to count the number of active AND gates in the pattern. We say that a bit difference $\Delta A_{u,j}$ (bit j of ΔA_u) activates an AND gate in step i if A_u is one of the four inputs of the two AND operations in step i. It is not difficult to see that $i - u$ must be equal to t_1, t_2, t_3 or t_4. Hence, a bit difference can activate up to four AND gates in four distinct steps.

In most cases, a bit difference can indeed activate four AND gates. However, we need to be careful with the following special cases when counting the total number of active AND gates within an s-round segment.

Case 1: Two bit differences activate the same AND gate.

If two bit differences $\Delta A_{u,j}$ and $\Delta A_{v,j}$ are exactly 3 steps apart ($u - v = 3$), then the AND gate $A[u] \wedge A[v]$ in step $u + t_1$ can only be activate once, not twice. Similar situation also happens if the two differences are 36 steps apart.

Case 2: Two AND gates get activated in the same step.

Let α, β be the jth output bit of the first and second AND operation, respectively. Once the input and output differences of step i are fixed, there is only one possible value for

$$T = \Delta\alpha \oplus \Delta\beta,$$

since both the XOR operation and the g operator are linear. However, if both AND gates are active at bit j, there will be two possible values for $(\Delta\alpha, \Delta\beta)$, namely $(0, T)$ and $(1, T \oplus 1)$, and each holds with probability $1/4$. Hence, the two active AND gates together contribute a probability of $1/2$, as opposed to $1/4$. Effectively, they should be counted as a single active AND gate in our differential analysis.

Case 3: an AND gate goes across the round boundary.

To avoid potential over counting, we only count active AND gates which have both of its inputs in the boundary of the s-round segment in which the search is carried out.

6.9.3.3 Searching for differential path weight patterns

In this section, we describe how to search for valid differential path weight patterns of MD6 up to s rounds. Since each D_i can potentially take any value from 0 to w, exhaustive search is infeasible even for very small s. Instead, we proceed with a more intelligent search by eliminating as many invalid patterns as possible.

Let $D_0, D_1, ...D_{i-1}$ be a differential path weight pattern up to $i-1$ steps. Let us focus on step i and consider the last operation $Z = g(X)$. We know that D_X lies in the range $[LB_X, UB_X]$. For each possible $D_i = D_Z \in [0, w]$, we can eliminate many impossible values by checking against several conditions on D_i, UB_X, and LB_X. In particular, the following values of D_i are invalid:

1. $D_i = 0$ and $LB_X > 0$.
   (Output difference can't be zero if input difference to g is non-zero.)
2. $D_i > 0$ and $D_i > 4UB_X$.
   (See Inequality (6.4).)
3. $D_i = 1$ and $UB_X < 5$.
   (See Inequality (6.5).)

We next outline an algorithmic procedure to search for valid differential path weight patterns and count the number of active AND gates for a segment of s rounds of the MD6 computation. The procedure, called `SearchDiff()`, is described in Figure (6.3).

At a high level, the search is carried out in a depth-first manner with nodes at level i of the search tree representing possible values of D_i. For each node, the total number of AND gates up to this point is computed. If the number is larger than the preset threshold, all search branches below the node are terminated.

We remark that it is possible that a small number of invalid D_i may pass through during the search, but it will not affect our lower bound analysis in the sense that we might prove a lower bound that is smaller than the actual value for AAG_s.

6.9.3.4 Deriving lower bounds through computer-aided search

We implemented the algorithmic procedure given in Figure (6.3). The search program runs for increasing number of rounds s. For each s, the threshold on the maximum number of active AND gates in the segment, $maxAAG$, is incremented until a valid differential weight pattern is found, and the stopping value for $maxAAG$ then yields on lower bound on AAG_s. The complete experimental results are given in Figure (6.4).

`SearchDiff`$(i, s, maxAAG)$

1. Check whether the search reaches the end of the s-round segment. If not, proceed to the next step. Otherwise, output the differential path weight pattern and the number of active AND gates. Stop.
2. Compute upper bound UB_X and lower bound LB_X for D_X in step i.
3. For $D_i = 0, 1, ...w - 1$
   (a) Check whether D_i is an invalid value given (LB_X, UB_X). If so, proceed to the next D_i.
   (b) Compute the number of new active AND gates given D_i.
   (c) Compute $sumAAG$, the total number of active AND gates up to step i. If $sumAAG > maxAAG$, proceed to the next D_i.
   (d) `SearchDiff`$(i + 1, s, maxAAG)$.

Figure 6.3: Algorithmic procedure to search for valid linear paths.

s	≤ 5	6	7	8	9	10	11	12	13	14	15
LB on AAG_s	0	3	4	4	4	4	7	13	19	20	26

Figure 6.4: Lower bounds on the number of active AND gates in a differential path up to s rounds. The results were obtained through computer search.

Here we elaborate a little on the output result from our search program. Let `i.d` denote that $D_i = d$. For $s = 15$, our program produces the following differential path weight pattern:

`54.1 71.2 143.2 232.2 .`

The total Hamming weight of the pattern is $1 + 2 + 2 + 2 = 7$, and the total number of active AND gates is $7 \times 4 - 2 = 26$. Note that the "-2" in the calculation is due to the fact that $i = 232$ is less than $t_4 - t_3 = 36$ steps from the right boundary 240, which is one of the special cases (Case 3) for counting the number of active AND gates.

Finally, we are ready to derive a lower bound on the workload of any differential-style collision search attack on MD6. As we discussed early, the workload is at least 2^{AAG_r}, where AAG_r is the minimum number of active AND gates in any r-round differential path of MD6.

It is tempting to immediately combine Inequality (6.8) with our experimental results in Figure (6.4). However, we still need to consider the important issue of security margins[3]. It is possible for an attacker to penetrate a few rounds at the

[3] Roughly, a security margin is the number of rounds that can be removed from the

beginning of the hash computation by manipulating the inputs and influencing the behavior of the differential path. To be conservative, we eliminate 15 rounds from the specified total number of rounds r when calculating the lower bounds on the number of AND gates.

Using Inequality (6.8), we have

$$AAG_{r-15} \geq AAG_{15} \times \lfloor \frac{r-15}{15} \rfloor. \tag{6.9}$$

Figure (6.5) gives the lower bounds ($LB = 2^{AAG_{r-15}}$) on the workload of any differential style collision attack for various output sizes d. The comparison with the birthday bound ($BB = 2^{d/2}$) is also given.

d	r	$r-15$	$\lfloor \frac{r-15}{15} \rfloor$	$AAG_{r-15} \geq$	$LB \geq$	BB
40	50	35	2	52	2^{52}	2^{20}
80	60	45	3	78	2^{78}	2^{40}
128	72	57	3	78	2^{78}	2^{64}
160	80	65	4	104	2^{104}	2^{80}
224	96	81	5	130	2^{130}	2^{112}
256	104	89	5	150	2^{150}	2^{128}
384	136	121	8	208	2^{208}	2^{192}
512	168	153	10	260	2^{260}	2^{256}

Figure 6.5: Lower bounds on the workload of differential collision attacks. The result for $d = 256$ was computed as $AAG_{89} \geq AAG_{15} \times 5 + AAG_{14} = 26 \times 15 + 20 = 150$. All other results were obtained using Inequality (6.9). We see that differential attacks are less efficient than a simple birthday attack.

We have thus obtained the desired result:

The workload for a standard differential attack against MD6 is provably larger than the workload for a simple "birthday attack," for all NIST specified output sizes.

In fact, we have a stronger result—the workload for a standard differential attack against MD6 is provably larger than the workload for a simple "birthday attack," for all output sizes $1 \leq d \leq 512$.

A standard differential attack will not be effective against MD6—there are no differential paths with sufficiently high probability to make such an attack more efficient than a simple birthday bound attack looking for collisions.

We consider this a very significant result, since in large part it has been the success of differential attacks against hash functions that motivated NIST to organize the SHA-3 hash function competition. The fact that MD6 is not vulnerable to standard differential attacks is very appealing.

cipher or hash function without compromising the desired security level. Security margins are commonly studied in block cipher design and analysis. For example, all AES candidates have clearly stated security margins.

Of course, there may be nonstandard differential attacks (e.g. that used various forms of generalized differentials, as in [25]) that fall outside the scope of our proof. Further research is needed to explore and exclude such possibilities; our proof is merely a first step in such analysis.

On the other hand, there is probably a significant amount of "slack" in our result, and thus the bound could be made tighter and/or extended to more general attacks. It is quite possible that the actual lower bound on the number of active AND gates can be much larger. Here we list some of the reasons:

- Some of the differential path weight patterns may not correspond to any valid differential path of MD6.
- The s-round differential path weight pattern that our program found is not iterative, and so it cannot be concatenated to yield a pattern for r rounds.
- We did not count the AND gates across the boundary of two consecutive rounds.

If we can address some of the above problems, then we can further improve our lower bound analysis. It would also be interesting to explore how these bounds might be affected by other choices for the tap positions.

6.9.3.5 Related work

The use of computer-aided search in our lower bound proof was motivated by the work of Julta and Patthak. In [48], they showed how to derive lower bounds on collision probabilities of SHA-1 using computer-aided proof.

Here we briefly describe the key idea in their proof. In SHA-1, the input message, represented by a 16-by-32 0-1 matrix, is expanded to an 80-by-32 matrix W. Based on earlier differential cryptanalysis of SHA (e.g. [27]), each set bit in W contributes a probability q (q averaging $2^{-2.5}$) to the overall probability of the differential collision path. Hence, a lower bound on the number of 1's in W will yield an upper bound on the collision probability. The proof in [48] proceeds by first identifying possible configurations of W and then using computer-aided search to find lower bounds on the number of 1's for some of the configurations.

The high-level reasoning in our lower bound proof is in a way similar to that of the AES [33]. Roughly, the proof arguments for AES being secure against standard differential attacks are as follows:

- Each active S-box in AES has a differential probability of at most 2^{-5}.
- In four consecutive rounds of AES, there are at least 25 active S-boxes if the plaintext difference is non-zero[4].
- So the probability of any four-round differential path is at most 2^{-150}.

[4]This is due to the following differential property of the MDS matrix: the sum of the number of non-zero input bytes and non-zero output bytes is at least 5.

As we noted earlier, active AND gates in MD6 correspond to active S-boxes in AES. In addition, the effect of the g operator in MD6 is similar to that of the MDS matrix in AES. Therefore it is not a coincidence that the proofs of security bear some similarities.

6.9.4 Preliminary results related to upper bounds

Our research on finding a lower bound on the complexity of standard differential attacks worked by exploring the various possibilities an adversary might try to actually find a standard differential attack. However, it organized this exploration by looking at differential path weight patterns rather than differential paths, and derived a lower bound by looking at 15-round segments rather than a complete MD6 computation.

An interesting question is whether we can derive any differential attack with a complexity that is anywhere near our lower bound. So far we have been unsuccessful.

To begin with, one would need to find a differential weight pattern that spanned an entire MD6 computation. Following common approaches in differential cryptanalysis of block ciphers, we try to find iterative differential path weight patterns for MD6, since an iterative pattern allows one to construct a differential path weight pattern for as many rounds as possible.

By modifying our search program for the lower bound proof, we are able to find some interesting iterative differential path weight patterns for MD6. One of the iterative differential path weight pattern for 5 rotation is described below.

0.2	89.2	106.2	123.2	140.2
156.2	157.2	178.2	212.2	229.2
267.2	284.2	301.2	356.2	373.2
445.2	534.2	551.2	...	

In the above iterative differential weight pattern, the pattern starts to repeat at step 445, since $534 - 445 = 89 - 2 = 89$. The total number of differences is 30 in the segment of 445 steps, and the number of active AND gates is $30 \times 4 - 2 = 118$.

It is worth comparing this upper bound result with our lower bound result in terms of $avgAAG$, the average number of active AND gates per round. First, since $118/(445/16) \approx 4.24$, the iterative pattern for 445 steps implies that $avgAAG \leq 4.24$. Next, our earlier lower bound on AAG_{15} (see Figure (6.4)) implies that $avgAAG \geq 26/15 \approx 1.73$. So there is a gap between the upper and lower bound. The gap, although small per round, is actually quite significantly when it is amplified to the full rounds of MD6.

Can we build a differential path from the differential path weight pattern? To do so, we would need to analyze how to set the bit positions of a difference (given the Hamming weight of the difference) so that they can line up correctly within a given step and from one step to another. For now, actually constructing an effective differential path for more than a few rounds is more than we know how to do.

6.10 Linear cryptanalysis

Linear cryptanalysis, pioneered by Matsui [58], is a type of attack on block ciphers that in many ways resembles differential cryptanalysis. Indeed, the duality between the two has facilitated simultaneous security analysis of a block cipher against differential and linear cryptanalysis. One excellent example is the analysis of Rijndael by its designers [33, Section 6.3].

In this section, we analyze the strength of keyed MD6 against linear cryptanalysis. Although the structure of MD6 does not allows straightforward parallel analysis with respect to the two types of attacks, the general approach and methodology is very similar. Our main result is a lower bound proof showing that any standard linear cryptanalytical attacks on keyed MD6 requires at least 2^{210} hash input/output pairs associated with the unknown key. Hence, a key-usage policy that requires the key change after every 2^{210} messages would defeat such attacks.

The rest of the section is organized as follows. In Section 6.10.1, we consider some significant differences between keyed and keyless hash functions in terms of security requirements for their applications. In Section 6.10.2, we review basic definitions and analysis tools in linear cryptanalysis. In Section 6.10.3 we study linear properties of the MD6 step function by analyzing individual operations. In Section 6.10.4, we prove a lower bound on the data requirement of any standard linear attacks on the keyed MD6 compression function.

6.10.1 Keyed vs. keyless hash functions

Before presenting the technical details of our work on linear cryptanalysis of MD6, we feel that it is necessary to elaborate on the differences between a keyed and a keyless hash function in terms of security parameters and requirements for their applications in practice.

In ordinary usage, a hash function is keyless, and the relevant parameters controlling its security are its output digest length and perhaps also its internal number of rounds. It is significant to observe that in the application of a unkeyed hash function, an adversary's attack can be "off-line", requiring no cooperation of the hash function user. For example, a differential attack can be mounted against an unkeyed hash function without any such cooperation of the hash function user; the adversary can simply generate any desired number of input/output pairs by himself. Indeed, this was the case for all collision attacks on existing hash functions in the literature.

The situation is significantly different for a keyed cryptographic hash function, since the adversary can no longer obtain input/output pairs for the keyed function without the cooperation (willing, or perhaps unwilling somehow) of the hash function user who possesses the key. The adversary does not have the key, and so must rely on input/output pairs generated by the user. This introduces an additional security parameter, or control, over the ability of the adversary to mount his attack: this is the number λ of input/output pairs that the user is willing to let the adversary obtain for a particular key.

In most practical applications of keyed hash functions, the goal of the adversary is typically to either recover the key or forge valid input/output pairs without knowing the key given the available number of input/output pairs. This is because once the user has hashed λ messages using a particular key K, that key may be retired, and replaced in usage by a new key K'.

A key-usage policy controlled by such a parameter λ represents a meaningful, and significant security control available for keyed hash functions. There is an implicit key-usage control of 2^{128} for AES, since the number of distinct inputs is at most 2^{128}, even though the number of keys can be significantly larger (up to 2^{256}). More explicitly, NIST gives recommendations [71, Section 5.3.6.3(b), see also Table 1] for cryptoperiods for a MAC key or other symmetric keys (in terms of creating MAC'ed input/output pairs) of two years.

Presumably it is administratively easier to administer a key usage policy in terms of time periods rather than in terms of number of messages MAC'ed or encrypted. However, one trillion computers each producing a MAC'ed input/output pair every femtosecond for two years would produce "only" $2^{115.6}$ input/output pairs. We see that MD6 is well within the security guidelines recommended by NIST.

In sum, while we are interested in any form of attacks from a theoretical viewpoint, it is reasonable to judge attacks on keyed hash functions based on their ability to utilize (or not) a given number λ of hash function input/output pairs, for a given digest size d and number of rounds r. We will elaborate on how this security parameter is treated in our analysis of the keyed MD6 compression function in the rest of the section.

6.10.2 Basic definitions and analysis tools

In linear cryptanalysis of a block cipher, we aim to find effective linear expressions connecting some bits of the plaintext, the ciphertext, and the key. By taking sufficiently many plaintext/ciphertext pairs, the correct value of certain key bits can be recovered based on the derived linear expressions. In the context of a keyed hash function, we would be interested in linear expressions involving certain bits of the message input, the key input, and the hash output.

The linear expression is often called a linear approximation, which can be conveniently represented by the dot product of two vectors. Let X, K, Z denote the message input, key input, and hash output, and let $\Gamma_X, \Gamma_K, \Gamma_Z$ denote binary vectors of the same length as X, K, Z, respectively. Then

$$(\Gamma_X \cdot X) \oplus (\Gamma_Z \cdot Z) = \Gamma_K \oplus K \tag{6.10}$$

represents a linear approximation connecting the inputs and the putput of the hash function. The vector Γ is referred to as a selection vector. For example, if all three section vectors are 1, then the linear approximation becomes $X[0] \oplus Z[0] = K[0]$. Let p be the probability that a linear approximation holds, where the probability is taken over all possible inputs. The bias[5] ϵ of a linear approximation is defined to be $\epsilon = p - 1/2$ or $\epsilon = |p - 1/2|$.

[5]We will adopt the definition with absolute value in this report, since the sign of bias is

Typically, linear approximations of the full hash function as given by Equation (6.10) is constructed by joining linear approximations of individual steps. For MD6 compression, a linear approximation of the step function is of the form

$$\mathcal{LA}_i : \ \Gamma_i \cdot A_i = \oplus_{k=0}^{5} (\Gamma_{i-t_k} \cdot A_{i-t_k}). \tag{6.11}$$

For ease of reference, we will refer to Equation (6.11) as a local linear approximation and Equation (6.10) as a global linear approximation. A linear path[6] of MD6 is defined to be a sequence of local linear approximations

$$\{\mathcal{LA}_i, i = i_1, i_2, ...\}$$

that can be joined together to yield a global linear approximation.

To compute the bias of a joined approximations $LA = LA_1 \oplus LA_2$, we can use Matsui's piling-up lemma. which states that the effective bias of approximation LA is $\epsilon_1 \times \epsilon_2 \times 2$. Now suppose that the bias ϵ for the linear approximation of the full hash function has been computed. Then the number of input/output pairs required to exploit this bias is proportional to ϵ^{-2}. Therefore, the smaller the bias ϵ is, the larger the data requirement is.

We remark that the randomness and independence assumption about the MD6 step function (see Section 6.9.1) is also needed in our linear cryptanalysis. In particular, when computing the bias of a local linear approximation for step i, we assume that the input to step i appears random. In addition, to apply Matsui's piling-up lemma, we assume that computations for different steps are independent of each other.

6.10.3 Analyzing the step function

In this section, we analyze linear properties of individual operations in the MD6 step function and show how to construct linear approximations for each step. We also introduce the notion of active AND gates and threads, which will be used in the lower bound proof.

For each operation to be studied, we use uppercase letters X, Y, Z to denote the w-bit inputs and output. We use $\Gamma_X, \Gamma_Y, \Gamma_Z$ to denote the selection vectors of a linear approximation, and L_X, L_Y, L_Z to denote the Hamming weight of these selection vectors. We use lowercase letters x, y, z to denote a single bit in the w-bit words.

6.10.3.1 XOR gate

The XOR operation $Z = X \oplus Y$ is a linear operation. Any linear approximation of the form $\Gamma_Z = \Gamma_X \oplus \Gamma_Y$ holds with probability either one or 1/2. In order for the approximation to hold with probabily one (bias 1/2), the selection vectors must satisfy the following equality:

not crucial for our analysis.

[6] Although the term "differential path" is commonly used in the literature, the term "linear path" is not. We use it here for ease of discussion and also to draw analogy with our work on differential cryptanalysis.

$$\Gamma_Z = \Gamma_X = \Gamma_Y.$$

So the Hamming weights of the selection vectors are the same. That is, $L_Z = L_X = L_Y$.

6.10.3.2 AND gate

As in our differential cryptanalysis, we decompose the AND operation $Z = X \wedge Y$ as an array of w independent AND gate: $z = x \wedge y$. There are four possible linear approximations for the AND gate[7]:

$$z = x \oplus y, z = x, z = y, \text{ and } z = 0.$$

Assuming that the two input bits x and y are chosen at random, we can show (with straightforward probability calculation) that all four linear approximations hold with exactly the same bias, which is $1/4$.

We next introduce the notion of active AND gate with respect to a given linear path of MD6. If we were to follow the same approach as in our differential cryptanalysis, then we would define an active AND gate as an AND gate for which $\Gamma_x = 1$ or $\Gamma_y = 1$. That is, at least one of the two input bits is selected in the linear approximation of the AND gate. However, the situation is more complex due to the way how the AND operation is combined with the XOR operation in MD6. Let

$$u = v \oplus (x \wedge y). \tag{6.12}$$

Consider the linear approximation $\{LA : u = v\}$ for Equation (6.12). That is, the AND gate $z = (x \wedge y)$ is approximated by constant $z = 1$, and so neither input bit of the AND gate is selected. We observe that the approximation LA holds with bias $1/4$, since the AND gate still contributes a bias of $1/4$ even though neither input bit is involved in the linear approximation LA.

So this suggests that a different definition for active AND gates is needed to better capture the above situation in linear cryptanalysis. Now suppose a linear path of MD6 is given. Let LA be the linear approximation for Equation (6.12) in step i of the linear path. We say that $z = (x \wedge y)$ is an active AND gate in step i of the linear path if $\Gamma_v = 1$ (the selection vector for v, not x, y) in the linear approximation LA. The intuition behind the definition is that an AND gate is considered "active" if it effectively contributes a "noise" (bias $1/4$) for a bit that we are trying to approximate with some local linear approximation.

Just like differential cryptanalysis of MD6, the notion of active AND gate plays an important role in the our lower bound proof for linear cryptanalysis. In particular, each active AND gate always contributes $1/4$ to the overall bias of the global linear approximation, no matter which of the four linear approximation is selected for the AND gate. Since the AND operation is the only non-linear operation, the total number of active AND gates in the linear path is closely related to the total bias of the linear path.

[7] We ignore the constant term here. For example, $z = x$ and $z = x \oplus 1$ are considered as the same linear approximation.

6.10.3.3 g operator

The g operator $Z = g(X)$ is a linear operation, since its three components $(\oplus, <<, \gg)$ are all linear. Any linear approximation for g of the form $\Gamma_Z \cdot Z = \Gamma_X \cdot X$ holds with probability either one or $1/2$.

In order for the approximation to hold with probabily one (bias $1/2$), the two selection vectors must satisfy a certain relation. The specific form of the relation[8] is not directly relevant to our analysis, except that Γ_Z is uniquely determined by Γ_X. Instead, we are more interested in the lower bound on the Hamming weight of L_Z in terms of L_X.

Recall that the shift amounts in g was carefully chosen in the design stage with differential cryptanalysis in mind. In particular, each pair of shift amounts possess the property that "if the Hamming weight of the input difference $D_X \leq 4$, then the Hamming weight of the output difference $D_Z \geq 2$" (See Inequality (6.5)). This property helps to prevent differential paths in which all differences ΔA_i have Hamming weight at most one.

A similar property also holds for the shift amounts in g with respect to linear cryptanalysis, although not as strong as the one stated above. In terms of how Hamming weights of selection vectors propagate through the g function, we can show that for the 16 pairs of shift amounts defined in MD6,

$$L_X = 1 \text{ would imply } L_Z \geq 2. \tag{6.13}$$

For $L_X \geq 2$, we do not have any non-trivial lower bounds on L_Z other than it has to be positive (since non-zero Γ_X would imply non-zero Γ_Z). So we have

$$L_X \geq 2 \text{ would imply } L_Z \geq 1. \tag{6.14}$$

6.10.3.4 Combining individual operations

We have analyzed linear properties of the three operations in the MD6 step function and the results are summarized in Figure (6.6) and Figure (6.7).

operation	linear approximation	bias
$Z = X \oplus Y$	$\Gamma_Z = \Gamma_X = \Gamma_Y$	$1/2$
$z = x \wedge y$	four possible approximations	$1/4$
$Z = g(X)$	$\Gamma_X = g^{rev}_{r,\ell}(\Gamma_Z)$	$1/2$

Figure 6.6: Linear approximations for $\oplus$, $\wedge$, and g. Note that for $\wedge$, the result is for an AND gate at the bit level.

[8] For $Z = g_{r,\ell}(X)$, we can define its "reverse operation" (not inverse) $X = g^{rev}_{r,\ell}(Z)$ as follows: $temp = Z \oplus (Z \gg \ell), X = temp \oplus (temp \ll r)$. We can show that if the linear approximation $\Gamma_Z \cdot Z = \Gamma_X \cdot X$ for $g_{r,\ell}$ holds with probability one, then the two selection vectors must satisfy the relation $\Gamma_X = g^{rev}_{r,\ell}(\Gamma_Z)$.

operation	upper bound	lower bound
$Z = X \oplus Y$	$L_Z = L_X = L_Y$	same as upper bound
$Z = X \wedge Y$	$L_Z \leq L_X + L_Y$	$L_Z \geq 0$
$Z = g(X)$	$L_Z \leq 4L_X$	If $L_X = 1$ then $L_Z \geq 2$ If $L_X \geq 2$ then $L_Z \geq 1$

Figure 6.7: Hamming weight of linear approximation for $\oplus$, $\wedge$, and g.

We are now ready to construct linear approximations for the step function by combining linear approximations of individaul operations. First, we decompose the step function into two sub-steps:

$$\begin{aligned} X &= A_{i-t_0} \oplus A_{i-t_5} \oplus (A_{i-t_1} \wedge A_{i-t_2}) \oplus (A_{i-t_3} \wedge A_{i-t_4}), \\ A_i &= g(X). \end{aligned}$$

The linear approximation $\mathcal{LA}_i$ for the step (given by Equation (6.11)) can be naturally decomposed into the following two approximations:

$$\Gamma_X \cdot X = \oplus_{k=0}^{5} (\Gamma_{i-t_k} \cdot A_{i-t_k}) \quad (6.15)$$

$$\Gamma_{A_i} \cdot A_i = \Gamma_X \cdot X \quad (6.16)$$

In order for $\mathcal{LA}_i$ to hold with a non-zero bias, the approximation defined by Equation (6.15) must hold with a non-zero bias. Given the linear property of XOR, the following conditions on the selection vectors must hold:

- $\Gamma_X = \Gamma_{i-t_0} = \Gamma_{i-t_5} \stackrel{\text{def}}{=} \gamma$.
  (Selection vectors for output and the two XOR inputs must equal.)
- For $t = t_1, t_2, t_3, t_4$, the set of non-zero bits in selection vector Γ_{i-t} must be a subset of the non-zero bits in γ.

Based on our earlier discussion on AND gates (in Section 6.10.3.2), each non-zero bit in γ activates exactly two AND gates in the linear approximation. Here, we introduce a new notion called thread—we say that each non-zero bit of γ defines a thread in step i. That is, a thread corresponds to the computation at a particular bit position before applying the g function. The number of threads in step i is equal to the Hamming weight of γ, which fully determines the bias of linear approximation $\mathcal{LA}_i$.

We remark that the number of active AND gates is always equal to twice the number of threads in the linear path. As we will see, the notion of thread and its relation to the number of active AND gates will be crucial in our lower bound proof.

6.10.4 Lower bound proof

6.10.4.1 Goal and approach

The goal of our analysis is to prove a lower bound on the data requirement of any standard linear attack on the keyed MD6 compression function. In light of earlier discussion on keyed hash functions Section 6.10.1, we will take into account the effect of the security parameter λ, the number of input/output pairs that are available to the attacker.

Similar to our approach in the differential case, the lower bound proof in the linear case is achieved through several steps of problem reductions and facilitated by computer-aided search.

First, deriving lower bounds on data requirement is reduced to deriving upper bounds on the total bias of a valid linear path. Then, deriving upper bounds on the total bias is reduced to deriving lower bounds on the total number of active AND gates in the path, which is in turn reduced to deriving lower bounds on the total number of threads (defined in Section 6.10.3.4) in the path.

In order to count the number of threads in a valid linear path, we analyze how local linear approximations—linear approximations for a single step—can be joined to form a valid linear path. This yields a set of necessary conditions on the local approximations. Based on these conditions, we carry out a computer-aided search for small number of rounds. The lower bounds for small number of rounds are then extended to produce expected lower bounds for the full r rounds of MD6 for various output sizes d.

6.10.4.2 Searching for linear path and counting threads

Recall that a linear path of MD6 is represented by a sequence of local linear approximations for individual steps

$$\{\mathcal{LA}_i, i = i_1, i_2, ...\},$$

where each $\mathcal{LA}_i$ is of the general form

$$\mathcal{LA}_i : \ \Gamma_i \cdot A_i = \oplus_{k=0}^{5} \ (\Gamma_{i-t_k} \cdot A_{i-t_k}).$$

At a first glance, there seem to be many possible choices for each local approximation, and so enumerating all possible linear paths is certainly infeasible even for very small s. Therefore, we need to eliminate as many invalid paths as possible in the search.

A key observation is that for a linear path to be useful in launching a linear attack, the sequence of local linear approximations must "collectively" be equivalent to the global linear approximation of MD6 given by Equation (6.10). That is, all non-zero selection vectors for intermediate variables must be canceled out with each other.

Here we elaborate on the reasoning behind this important observation for our linear cryptanalysis. Suppose that after the cancelation of intermediate

selection vectors, one of the internal bits, $A_j[t]$, remains uncanceled. Then the resulting global linear aproximation would be of the form

$$(\Gamma_X \cdot X) \oplus (\Gamma_Z \cdot Z) \oplus A_j[t] = \Gamma_K \oplus K,$$

where X is the input message, K is the input secret key, and Z is the hash ouput of MD6. Since the bit $A_j[t]$ is hidden from the attacker, it would appear to be just a random bit from the attacker's viewpoint. (Here we again use the randomness assumption about the MD6 step function.) Hence, the above global approximation would hold with probability 1/2 (bias zero), which would not be useful for launching a linear attack.

The above requirement regarding cancelation helps to quickly filter out a significant portion of the search branches in the early stage before they end up producing an invalid linear path.

To capture the above requirement in a more analytical way and facilitate discussion, below we introduce a few terms that distinguish different types of selection vectors in a local linear approximation.

Consider an intermediate variable A_j. Since the step function consists of six inputs and one output, A_j appears in the following seven steps:

$$\text{step } i = j, j + t_0, j + t_1, j + t_2, j + t_3, j + t_4, j + t_5.$$

If the selection vector corresponding to A_j is non-zero in the local linear approximation $\mathcal{LA}_i$, we say that A_j gets hit in step i. Depending on the role of A_j in the step, we further categorize the hits on A_j into three different types:

- If A_j an XOR input, we say that A_j gets a hard hit.
  (In this case, $j = i - t_0$ or $i - t_5$.)
- If A_j is an AND input, we say that A_j gets a soft hit.
  (In this case, $j = i - t_1, i - t_2, i - t_3$ or $i - t_4$.)
- If A_j is the output of step i, we say that A_j gets a direct hit.
  (In this case, $j = i$.)

The main reason for distinguishing different types of hits is that these hits follow different rules of cancelation when two or more local approximations involving A_j are combined. In particular, a direct hit or a hard hit must be canceled by another hit, while a soft hit can either be canceled by another hit or be "set to zero". That is, when A_j gets a soft hit in step i, we may choose to set the corresponding selection vector $\Gamma_j = 0$ if the soft hit is not needed to cancel other direct or hard hits. This would not affect the bias of the local approximation, since all four possible approximations of AND hold with the same bias (see discussions in Section 6.10.3.2).

Returning to our first goal of searching for valid linear paths, with the newly introduced terms, we can now restate the earlier requirement that "all selection vectors for intermediate variables must be canceled out with each other" as "all three types of hits on an intermediate variable A_j must be canceled out with each other within the seven steps in which A_j is involved."

Returning to our second goal of counting the number of threads during the search process, we analyze the relationship between the number of threads and the number of hits within each step. Let dn, hn, sn denote[9] the number of direct hits, hard hits, and soft hits on A_j in step i, respectively, and let T_i denote the number of threads in step i. For each type of hit, the relation between the number of hits and number of threads T_j is given in Figure (6.8).

A_j	$j =$	hit type	relation to T_i
XOR input	$i - t_k, k = 0, 5$	hard hit	$hn = T_i$
AND input	$i - t_k, k = 1, 2, 3, 4$	soft hit	$sn = T_i$
output	i	direct hit	(1) if $T_i = 1$ then $dn \geq 2$ (2) if $T_i \geq 2$ then $dn \geq 1$

Figure 6.8: Relations between number of threads T_i and number of hard, soft, and direct hits in step i. Note that dn and T_i follow the constraints that we have derived for the input and output of the g function.

We are now ready to outline an algorithmic procedure to search for valid linear paths and count the number of threads for a segment of s rounds of the MD6 computation. The procedure, called `SearchLinear()`, is described in Figure (6.9). Roughly speaking, the search is conducted in a depth-first manner with nodes at level i of the search tree representing all possible values of T_i. The number of hits is then set given their relations with T_i. As soon as failure of cancelation for hits on A_j is detected, all search branches below node $j + n$ are terminated[10].

The most complex step of the search is Step 3 in which the set of hits on A_j is checked to see whether they can get canceled with each other. To accomplish the check, we exam each possible combination of direct, hard, and soft hits and associate each case with a "yes" or "no" label. In the search program, we simplify the analysis by assigning a "yes" label to all cases for which the sum of the number of hits is larger than three and unifying some of the remaining cases. Further details of our search algorithm are given in the C code.

6.10.4.3 Deriving lower bounds through computer-aided search

In this section, we first present the experimental results from our computer-aided search of linear paths for small number of rounds. We then extend the experimental results to derive lower bounds on the data requirement of any standard linear attack against the full r-round MD6 for different output sizes d.

Our method for extending the experimental results depends on the relation between the data requirement of a linear attack and the number of active AND

[9]To simply notation, we do not include subscripts here, although the numbers are specific to intermediate variable A_j in step i.

[10]This is because any hits beyond step $j + 89$ is more than 89 steps away from j and can no longer be used to cancel the hits on A_j.

`SearchLinear`$(i, s, maxAAG)$

1. Check whether the search reaches the end of the s-round segment. If not, proceed to the next step. Otherwise, output the linear path found and the number of active AND gates. Stop.
2. Check whether certain boundary conditions are met. If so, proceed to the next step. Otherwise, stop the current branch of search.
3. Let $j = i - n - 1$. Check whether all hits on A_j can be canceled. If so, proceed to the next step. Otherwise, stop the current branch of search.
4. For $T_i = 0, 1, 2, 3$
   (Here "3" represents all possible values of $T_i \geq 3$.)
   (a) Set dn as the minimum number of direct hits consistent with T_i. (So possible values for dn are $0, 1, 2$.)
   (b) Compute the sum of T_i up to step i.
   (c) Compute $sumAAG$,the sum of active AND gates to up step i. (This is always twice the number of threads.) If $sumAAG > maxAAG$, proceed to the next T_i.
   (d) `SearchLinear`$(i + 1, s, maxAAG)$.

Figure 6.9: Algorithmic procedure to search for valid linear paths.

gates in the underlying linear path (of the attack), as already derived in earlier analysis. To recap, the data requirement is computed as ϵ^{-2}, where ϵ is the total bias of the linear path. The total bias ϵ is computed as

$$\epsilon = (1/4)^{AAG} \times 2^{AAG-1} = 2^{-AAG-1},$$

where AAG is the number of active AND gates in the path.

Let AAG_s^L be the minimum number of active AND gates in s consecutive rounds of a valid linear path of MD6. Then we can lower bound AAG_r^L using AAG_s^L as follows.

$$AAG_r^L \geq AAG_s^L \times \lfloor r/s \rfloor + AAG_{s'}^L, \tag{6.17}$$

where s' is r modulo s.

Our computer-aided search program implements the algorithmic procedure given in Figure (6.9). The program runs for increasing number of rounds s, starting at $s = 6$ (so the search segment is at least one-rotation long). Besides checking the condition on "hit cancelation" in Step 3, the program also checks the following boundary condition in Step 2:

- The linear path must involve some bits in the first rotation and some bits in the last rotation of the s-round segment.

Note that this boundary condition is necessary in order for two s-round linear paths to join together to form a valid $2s$-round linear path.

At the time of writing, we have completed the search for up to $s = 34$ rounds. Experimental results for AAG_s^L are given in Figure (6.10). Plugging these results into Inequality (6.17), we can easily derive lower bounds on the number of active AND gates for larger number of rounds. Adopting the same security margin as that in differential cryptanalysis, we eliminate 15 rounds from the total number of rounds r when computing the lower bounds. For example, since $168 - 15 = 153 = 30 \times 5 + 3$, we have

$$AAG_{168}^L \geq AAG_{153}^L \geq AAG_{30}^L \times 5 + AAG_3^L \geq 40 \times 5 + 0 = 200.$$

Lower bounds on AAG_r^L corresponding to different output sizes d are summarized in Figure (6.11).

s	11	12	13	14	15	16	17	18	19	20	21	22
$AAG_s^L \geq$	8	10	10	10	14	14	18	18	18	22	24	24
s	23	24	25	26	27	28	29	30	31	32	33	34
$AAG_s^L \geq$	26	30	32	32	34	34	38	40	40	40	44	44

Figure 6.10: Lower bounds on the number of active AND gates in a linear path up to s rounds. These results were obtained by computer search.

d	r	$r - 15$	$AAG_{r-15}^L \geq$	$LB \geq$
160	80	$65 = 30 \times 2 + 5$	$40 \times 2 + 0 = 80$	2^{162}
224	96	$81 = 30 \times 2 + 21$	$40 \times 2 + 24 = 104$	2^{210}
256	104	$65 = 30 \times 2 + 29$	$40 \times 2 + 38 = 118$	2^{238}
384	136	$65 = 30 \times 4 + 1$	$40 \times 4 + 0 = 160$	2^{322}
512	168	$65 = 30 \times 5 + 3$	$40 \times 5 + 0 = 200$	2^{402}

Figure 6.11: Lower bounds on the data requirement of standard linear attacks on r-round MD6. Lower bounds on AAG_{r-15}^L are obtained using Inequality (6.17). Lower bounds on LB are computed as $2^{2AAG_{r-15}^L+2}$.

Figure (6.11) shows that the data requirement and workload of a standard linear attack against MD6 is at least 2^{210} for output size $d \geq 224$. We can now conclude with a strong result with respect to linear cryptanalysis:

A key-usage policy that requires the key change after $\lambda = 2^{210}$ messages would defeat standard linear cryptanalytical attacks on keyed MD6 for all NIST specified hash sizes.

Our work to improve these linear cryptanalytic results is ongoing. While we believe that our lower bounds on the data requirement can in fact be significantly

improved, we note that it does not actually need to be so improved in order to provide security against massive linear cryptanalysis attacks, as long as a key-usage policy such as the above is enforced.

(We note that if someone wishes to use keyed MD6 for d-bit MAC computations where $d \leq 160$, we recommend that they keep $r = 80$, rather than allowing r to decrease to 40 as d approaches 1 as is done for unkeyed usage of MD6. That is, in such cases the output length may not be a good determinant of the desired security level, since we are now worried about key protection rather than collision-resistance. Requiring $r \geq 80$ appears to provide adequate protection against linear cryptanalysis attacks, for example, even for very short output lengths. Further analysis might show that fewer rounds are also safe, but for now a recommended minimum of 80 rounds seems a conservative approach. The default number of rounds for MD6 follows this approach.)

In sum, decoupling the key-usage parameter λ from the other security parameters d and r is, we feel, a significant and useful way to proceed. Hence, even with the preliminary results given in this report, we have demonstrated that in this expanded analysis framework, MD6 offers strong security against linear cryptanalysis attacks.

6.10.4.4 Related work

Our lower bound proof with respective to linear cryptanalysis is also similar to that of the AES [33]. The proof that AES is secure against standard linear attacks goes as follows:

- Each active S-box in AES has a maximum input/output correlation of at most 2^{-3}.
- In four consecutive rounds of AES, there are at least 25 active S-boxes.
- So the input/output correlation of any four-round linear path is at most 2^{-75}, implying a data requirement of at least 2^{150} to launch a standard linear attack.

Finally, we remark that active AND gates in MD6 play the same role to active S-boxes in AES with respect to linear cryptanalysis. The g operator in MD6 also has some limited effect in terms of enforcing a small number of active AND gates across two consecutive steps, although it does not directly compare to the strong diffusion property of the MDS matrix.

6.11 Algebraic attacks

There are a number of algebraic attacks one might consider trying against the MD6 compression function.

6.11.1 Degree Estimates

Many algebraic attacks, such as Shamir's "cube" attack, require that the algebraic normal form describing the output bits have relatively small degree in terms of the input bits being considered. There are $89 \cdot 64 = 5696$ input variables, one for each of the 64 bits in each of the 89 input words to the compression function f. A particular algebraic attack may consider only a subset of these, and leave the others fixed.

Each bit of each word $A[i]$ is a function of the $nw = 5696$ input variables, that can be described in a unique way by a polynomial in algebraic normal form (ANF). A particular MD6 output bit can be represented in a unique way as an ANF polynomial with input variables $x_1, x_2, \ldots, x_{nw}$ and coefficients and outputs in $GF(2)$, e.g.

$$a_0 \oplus a_1 x_1 \oplus \ldots \oplus a_{nw} x_{nw} \oplus a_{nw+1} x_1 x_2 \oplus \ldots \oplus a_{2^{nw}-1} x_1 x_2 \cdots x_{nw} \ . \quad (6.18)$$

There are 2^{nw} terms in this sum, one for each subset of the variables. The coefficient for a term determines whether that term is "present" (coefficient = 1) or "absent" (coefficient = 0).

6.11.1.1 Maximum degree estimates

We can estimate the degree of the polynomial for such bit in $A[i]$ as follows. We estimate a common degree of the polynomials for each of the 64 bits occuring word $A[i]$. Let δ_i to denote the estimated common degree for bits in $A[i]$:

$$\delta_i = 1 \text{ for } i = 0, 1, \ldots, n-1 \quad (6.19)$$

and

$$\delta_i = \min(\delta_{i-t_5}, \delta_{i-t_0}, \chi(\delta_{i-t_1}, \delta_{i-t_2}), \chi(\delta_{i-t_3}, \delta_{i-t_4})) \text{ for } i > 88 \ ,$$

where

$$\chi(d_1, d_2) = d_1 + d_2 - d_1 d_2 / nw \ .$$

(The term $d_1 d_2 / nw$ accounts for the fact that random terms of degree d_1 and d_2 should have about $d_1 d_2 / nw$ variables in common, so the degree of their product will be smaller by this amount than the sum $d_1 + d_2$ of their degrees.) We obtain the following estimate for the minimum degree of any bit computed in each round.

According to these estimates (see Table 6.4), after 12 rounds the minimum degree should easily exceed 512, and after 20 rounds the minimum degree should almost certainly equal the maximum possible degree $nw = 5696$.

Similar computations can be performed when one cares about only some of the input variables (i.e., when some of the variables are fixed, and we only care about the degree in terms of the remaining variables).

rounds	minimum degree
1	2
2	2
3	4
4	8
5	16
6	24
7	42
8	66
9	128
10	252
11	443
12	665
13	1164
14	1819
15	2910
16	4268
17	5156
18	5565
19	5692
20	5696
21	5696
22	5696
23	5696
24	5696

Table 6.4: Table of estimated degrees of polynomials after a given number of rounds. MD6 is estimated to have polynomials of maximum possible degree after 20 rounds.

6.11.1.2 Density estimates

An algebraic attack may depend not only on the degree of the polynomials, but also on how dense they are. For example, if a polynomial is not dense, then fixing some of the variables may dramatically reduce the degree in the remaining variables. This section provides some crude estimates of the density of the MD6 polynomials.

We give an informal definition. We say that a polynomial in nw variables has "dense degree d_1" if the number of terms of any degree $d_2 \leq d_1$ is near its expected value $(1/2)\binom{nw}{d_2}$. This definition is informal because we don't carefully define what we mean by "near".

We base our estimates on the following informal proposition.

Proposition 1 *The product of two ANF polynomials P_1 and P_2 of dense degrees d_1 and d_2 respectively is a ANF polynomial P_3 of dense degree d_3, where*

$$d_3 = min(nw, d_1 + d_2) \ .$$

(Here P_1 and P_2 are defined in terms of the same set of nw variables.)

The "proof" of this proposition notes that any given possible term t of degree d_3 can be formed in many possible ways as the product of a term t_1 of degree d_1 from P_1 and a term t_2 of degree t_2 from P_2. If we view P_1 and P_2 as randomly chosen polynomials of dense degress d_1 and d_2 respectively, then the various such products $t_1 t_2$ will be present or absent independently, so that t will be present with probability $1/2$. Thus, P_3 is dense with degree d_3.

Of course, in an actual computation the relevant polynomials P_1 and P_2 may not be "chosen randomly", and may not be independent of each other, so this proposition is not rigorous. Nonetheless, it forms the basis for an interesting heuristic estimation of the density of the MD6 polynomials.

In this estimation, we let δ_i be our degree estimate such that we estimate that for each bit of $A[i]$, the corresponding ANF polynomial has dense degree δ_i.

We let $\delta_i = 0$ for all $i < n + 11c$, and set $\delta_i = 1$ for all i such that $n + 11c \leq i < n + 12c$. That is, we assume that the polynomials are not of dense degree 1 until the 12-th round. This corresponds to the output seen in our program `shiftopt.c`, which optimized the shift amounts for MD6 by considering the density of the linear portions of the polynomials.

Table 6.5 then lists lower bounds on estimated dense degrees for words in each round 12, ..., 31, based on the above proposition and the structure of MD6. We see that after 28 rounds we estimate that the MD6 ANF are of full dense degree (i.e., of dense degree 5696).

6.11.2 Monomial Tests

Jean-Philippe Aumasson has kindly allowed us to report on some of his initial experiments using a test related to those in [36, 39, 73].

The test attemps to determine whether reduced-round versions of the MD6 compression function have a low degree over GF(2), with respect to some small

rounds	dense degree
1..11	0
12	1
13	1
14	2
15	4
16	8
17	12
18	21
19	33
20	64
21	127
22	227
23	349
24	642
25	1079
26	2004
27	3877
28	5696
29	5696
30	5696
31	5696

Table 6.5: Table of estimated dense degrees of polynomials after a given number of rounds. MD6 is estimated to have polynomials of dense degree 5696 after 28 rounds.

subset of input variables. The experiments run consider families of functions parametrized by bits of $A[54]$, and considering the other bits of $A[54]$ as input variables.

The test was able to detect nonrandomness in the MD6 compression function after for 18 rounds in about 2^{17} computations of the function.

Extensions of these studies are still underway.

6.11.3 The Dinur/Shamir "Cube" Attack

At his CRYPTO 2008 keynote talk, Adi Shamir presented an interesting algebraic attack on keyed cryptosystems that is capable of key recovery, if the cryptosystem has a sufficiently simple algebraic representation. (This attack is joint work with Itai Dinur.) The cube attack is related to, but more sophisticated than, the Englund et al. maximum degree monomial test. It searches for sums over subcubes of the full Boolean input that give values for linear equations over the unknown key bits. With enough such values, the equations can be solved to yield the key.

We have begun some initial collaborations with Itai Dinur (and Adi Shamir) to evaluate the effectiveness of the "cube attack" on MD6.

Our initial results are very preliminary and tentative. It appears that the cube attack can distinguish key MD6 from a random function up to 15 rounds, and extensions of the cube attack will probably be able to extract the MD6 key in the same number of rounds. It is plausible that these results can be improved by a few rounds; experiments are ongoing.

6.12 SAT solver attacks

This section discusses the applicability of SAT solver attacks to MD6, and gives experimental evidence that they are ineffective beyond about a dozen rounds of the MD6 compression function.

SAT solver attacks are a fairly new technique, in which a cryptanalytic problem is represented as a SAT instance: a Conjunctive Normal Form (CNF) formula is written down, such that any assignment satisfying this formula corresponds to a a solution to the cryptanalytic problem. The CNF formula is then fed into a generic SAT solver program, which will hopefully find an assignment — from which the solution can be decoded. Unlike most cryptanalytic techniques, this technique exploits the circuit representation of the cryptographic primitive in a non-blackbox manner, rendering it very effective for some problems. SAT solver have been applied to block ciphers (e.g., [57]), digital signatures (e.g., [38]) and hash functions (e.g., [47][67]). However, since the general case of solving SAT instances is NP-complete, the use of this technique for typical cryptanalytic problems is heuristic and its complexity is poorly understood.

The natural way to use SAT solvers to find (partial) preimages of the MD6 compression function f is to convert its circuit representation into a CNF formula $\phi_f(N, C)$ relating the bits of the input N, the bits of the output C, and

some ancillary bits for the intermediate state of the compression function. One then extend $\phi_f(N, C)$ with further clauses restricting C to a desired value and expressing any desired constraints on N, and solves this instance to obtain the full input N. Similarly, to find an arbitrary pair of colliding inputs N, N', one can use the formula $\phi_f(N, C) \wedge \phi_f(N', C) \wedge (N_i \vee N'_i) \wedge (\overline{N_i} \vee \overline{N'_i})$, where i is the index of some bit forced to differ between N and N'.

Following this approach, we used a Python script to create a variety of CNF instances representing generic attacks on the MD6 compression function. In particular, we explored the following variants (the notation is used in the subsequent figures):

1. Task: finding a preimage of the all-zero output (`invert`) or finding an arbitrary collision of two inputs differing in a specific bit (`collide`).
2. Digest length: `d=128`, `d=160`, `d=224`, `d=256`, `d=384` or `d=512`.
3. Constraints on the sought compression function input: only Q fixed to its prescribed $\sqrt{6}$ value (`fix=Q`), only Q fixed to 0 (`fix=Q0`), or all of Q, K, U and V fixed to values corresponding to the first block in a long message (`fix=QKUV`).

To solve these SAT instances, we used Minisat2 beta 070721 (including its simplification preprocessing), which is one of the best general-purpose SAT solvers available and is popular in cryptanalytic context. We have also evaluated HaifaSat, march, picosat-632, rsat and SATzilla; these consistently performed worse than Minisat2 on our problem instances (for non-trivial problem sizes). In both figures, the lower cluster corresponds to the `invert` instances, whereas the upper cluster corresponds to the `collide` instances.[11]

Figures 6.12 and 6.13 summarize the running time and memory consumption of Minisat on all of the above instances. For ease of visualization, failed executions are always listed as running for 1000 seconds and consuming 1GB (no trial succeeded after using more than either of these).

These experiments were run on a Dual-Core AMD Opteron 221 (2.4GHz) with 2GB of DRAM running Linux under Xen. Available memory was initially constrainted to 1GB. When Minisat2 failed to solve a given instance within the allotted 1GB of memory, we increased the memory limit to about 1.5GB and also patched Minisat2 to reduce its memory use in exchange for a longer running time (by setting the internal variables `learntsize_factor=0.2` and `learntsize_inc=1.02`). In all of the above cases, the modified Minisat was still unable to solve these instances within the allotted memory.

As can be seen, our method of using SAT solvers to attack Minisat succeeded for no more than $r = 10$ rounds at the standard SHA-3 digest sizes ($d \geq 224$), and at most $r = 11$ rounds even for reduced digest sizes ($d = 128$ and $d = 160$). Moreover, after 6-7 rounds, both running time and memory usage appear to grow superexponentially in the number of rounds. While varying the CNF

[11] One would expect finding collisions to be easier than inversion, but apparently the SAT solver is bogged down by the larger number of variables.

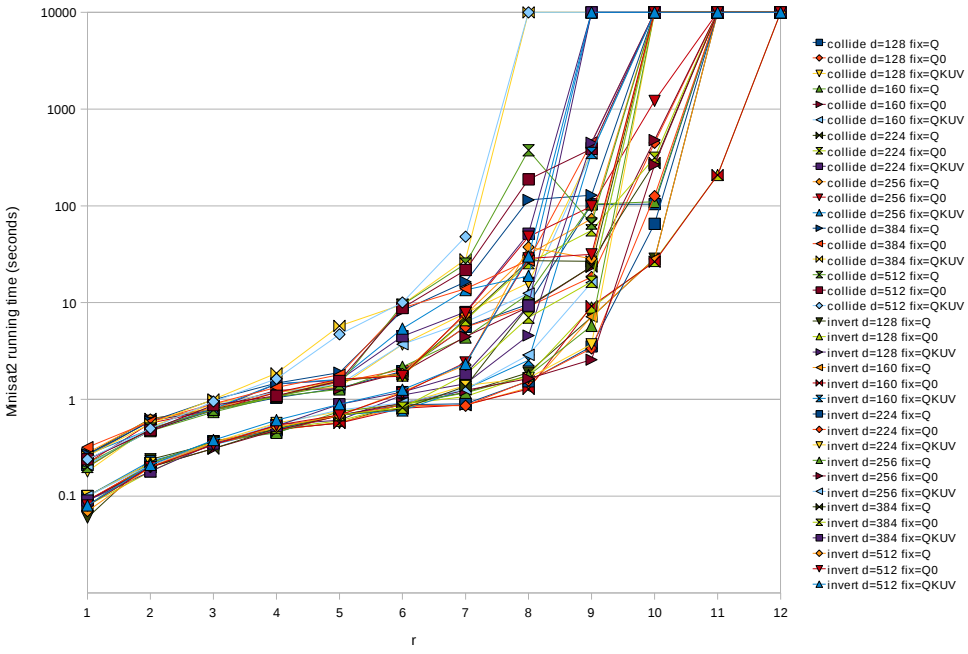

Figure 6.12: Minisat running time. A running time of 10000 seconds designates a trial that ran out of memory (no trial succeeded after a longer time).

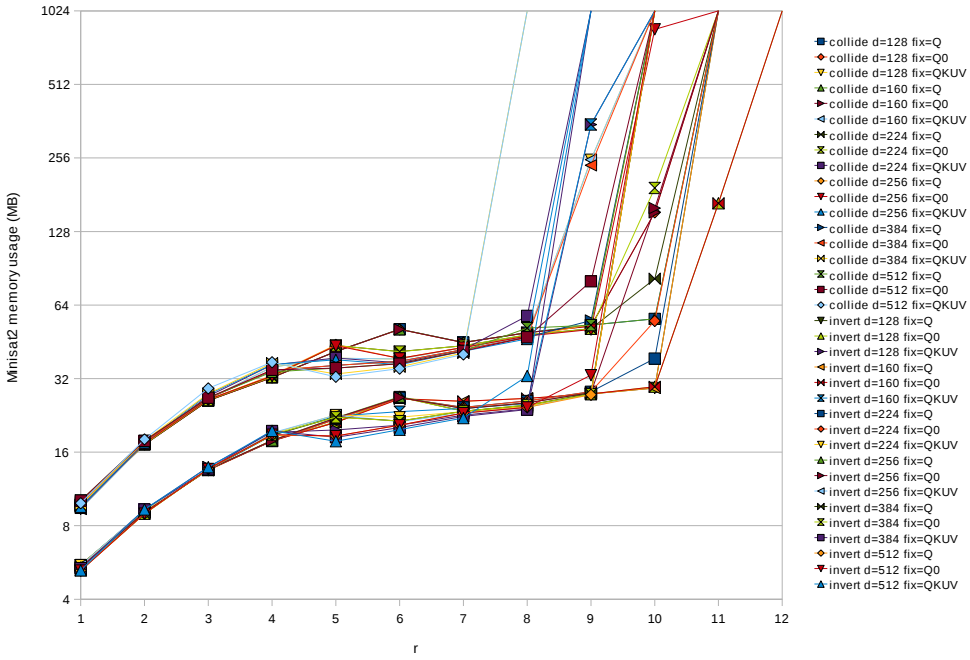

Figure 6.13: Minisat memory consumption. A memory consumption of 1024MB designates a trial that ran out of memory (no trial succeeded after using more than 1024MB).

circuit representation and the choice of SAT solver program may somewhat alter these numbers, it typically does not have a dramatic effect. We thus conclude that at the current state of the art in SAT solvers, this form of SAT solver attack is ineffective against the full MD6 compression function.

6.13 Number of rounds

Given the studies and tests of the preceding sections, it is appropriate to revisit the question of the number of rounds that is appropriate for MD6.

None of our tests are able to distinguish MD6 with 11 or more rounds from a random function. Jean-Philippe Aumasson's test of Section 6.11.2 can distinguish MD6 from a random function up to 18 rounds, but not more (at least at present).

Our choice of $r = 40 + (d/4)$ rounds as the default thus seems quite conservative.

Perhaps after further study, the default number of rounds for MD6 could be justifiably and significantly reduced.

6.14 Summary

The compression function of MD6 appears to have all of the desired properties: preimage-resistance, second-preimage resistance, collision-resistance, pseudorandomness, resistance to linear and differential attacks, resistance to SAT-solver attacks, and indifferentiability from a random oracle. It provides an excellent foundation on which to build the complete MD6 hash function using the MD6 mode of operation.

Chapter 7

Mode of Operation Security

A **mode of operation** $\mathcal{M}$ is an algorithm that, given a fixed-length compression function or block cipher f_Q, describes how to apply f_Q repeatedly on fixed-length chunks of the arbitrarily-sized input in order to produce a fixed-length output for the whole. In this way, one can construct Variable Input Length (VIL) cryptographic primitives from Fixed Input Length (FIL) cryptographic primitives, which is a functionality commonly referred to as domain extension [35].

In this chapter we address the cryptographic properties of collision resistance, first-preimage resistance, second-preimage resistance, psuedorandomness, unpredictability, and indifferentiability from random oracles, as pertains particularly to the MD6 mode of operation itself[1]. That is, in this chapter we take an agnostic approach to the compression function used and consider it as only a black-box function f_Q with some desirable property P (e.g. collision resistance, pseudorandomness, etc.). Our goal is then to show that the MD6 mode of operation acts as a domain extender for f_Q that preserves the property P. The question we attempt to answer is, does the MD6 mode of operation dilute the security of the black-box function with respect to P, and if so, by how much? For collision resistance, first-preimage resistance, pseudorandomness, and indifferentiability we give concrete security bounds for the property preservation. For the MAC functionality (unpredictability), we do the same, although we must introduce an additional nontrivial assumption on the compression function. For the property of second-preimage resistance, we are unable to demonstrate provably that it preserves this property (although we reduce to a weaker property instead).

The organization of this chapter is as follows. After some preliminary remarks in Section 7.1, Sections 7.2, 7.3, 7.4 respectively show how the MD6 mode of operation is a domain extender for collision-resistance, preimage-resistance (inversion resistance), and second-preimage resistance. Then Section 7.5 uses

[1] Many of the results presented here were derived by Christopher Crutchfield in his MIT EECS Master's thesis [32]; we summarize those results here but refer the reader to the thesis for the full exposition and derivation.

Mauer's Random Systems Framework to show that the MD6 mode of operation is a domain extender for pseudo-random functions. The more delicate issue of showing that the MD6 mode of operation is a domain extender for unpredictability is handled in Section 7.6. Section 7.7 shows that the MD6 mode of operation provides indifferentiability of MD6 from a random oracle assuming that the compression function is a random oracle (and that the adversary has access to the compression function). Sections 7.8 and 7.9 discuss, in response to NIST requirements, the issues of multi-collision attacks and length-extension attacks. Section 7.10 then summarizes the results of this chapter.

7.1 Preliminaries

In this section we prove certain results about the collision and preimage resistance of the MD6 hash function mode of operation. In particular, we show that the MD6 mode of operation acts as a **domain extender** for various properties of the fixed input length compression function used. Our goal is to show that if we assume that the compression function is collision resistant (respectively, preimage resistant), then the entire hash function will be collision resistant (respectively, preimage resistant) as well.

One important caveat is that the notions of collision resistance and preimage resistance are only defined for keyed hash functions. As observed by Rogaway [85], an unkeyed hash function $H : \{0,1\}^* \to \{0,1\}^d$ is trivially non-collision resistant: the existence of an efficient algorithm that can find distinct strings M and M' such that $H(M) = H(M')$ is guaranteed, simply by virtue of the existence of such collisions (as Rogaway says, the algorithm has the collision "hardwired in"). Therefore for an unkeyed hash function H, what is meant by saying that H is collision resistant is not that an efficient collision-finding algorithm does not exist, but rather that no efficient collision-finding algorithm exists that is known to man (the emphasis is Rogaway's). Similar concerns can also be expressed for the preimage resistance of unkeyed hash functions. Although MD6 can behave as a keyed hash function, certain applications call for the use of the unkeyed variant (where the key field is simply set to λ, the empty string), such as any application that uses a public hash function. Thus we would like to argue, with some amount of rigor, that its unkeyed variant is collision or preimage resistant as well.

Fortunately, we can perform reductions for finding collisions and preimages. Specifically, we will show that if one has a collision or preimage for the entire hash, then one can construct a collision or preimage for the underlying compression function. Therefore if one assumes that there is no known algorithm for finding collisions or preimages in the compression function, then there should be no known algorithm for finding collisions or preimages in the overall hash function. While this is not a completely rigorous notion of security (as it relies on the extent of human knowledge for finding collisions, which is impossible to formalize), it is the best we can do in these circumstances.

After proving reductions for these properties, we show that they apply to

the keyed hash function variant of MD6 as well (for certain — now rigorous — definitions of collision resistance and preimage resistance). That is, we will demonstrate that if an algorithm exists that can break the property of collision resistance or preimage resistance for keyed MD6, then we can use this algorithm as a black box for breaking the collision resistance or preimage resistance of the underlying compression function. We may then conclude that breaking either property is at least as difficult as breaking the respective property for the compression function.

Some of the proofs in this section are similar to those of Sarkar and Schellenberg [87], owing to the fact that their mode of operation is also based on a tree-like construction. However, the differences between the MD6 mode of operation and that of Sarkar and Schellenberg are significant enough to warrant entirely new proofs of security.

7.1.1 Definitions

We will often use the following notational conventions when precisely defining the advantage of an adversary for attacking some cryptographic property of the compression function or hash function.

$$\mathbf{Adv}^{P}_{\mathsf{A}} = \Pr\left[\begin{array}{c} \text{Random values are chosen;} \\ \mathsf{A} \text{ is given these values as input;} \\ \mathsf{A} \text{ produces some output} \end{array} \;:\; \mathsf{A}\text{'s output violates } P \right]$$

For example, one definition of the advantage of A for breaking the collision resistance of f_Q is due to Rogaway and Shrimpton [86].

$$\mathbf{Adv}^{\text{fil-cr}}_{\mathsf{A}} = \Pr\left[\begin{array}{c} K \xleftarrow{\$} \{0,1\}^k; \\ (m, m') \leftarrow \mathsf{A}(K) \end{array} \;:\; \begin{array}{c} m \neq m', \\ f_Q(K, m) = f_Q(K, m') \end{array} \right]$$

Here the probability is taken over the uniform random choice of key value K from the keyspace $\{0,1\}^k$ (where this random choice is denoted by \$). To be more precise, we should also take into account the internal randomness of the algorithm A, which in general is not a deterministic algorithm. We could denote this similarly as $(m, m') \xleftarrow{\$} \mathsf{A}(K)$, but throughout this paper we will often take this as a given and omit the \$.

Definition 3 (Fixed Input Length) *A function f_Q mapping domain $\mathcal{D}$ to $\mathcal{R}$ is a fixed input length (FIL) function if $\mathcal{D} = \{0,1\}^i$ for some positive integer i. That is, its inputs consist only of bit strings of some fixed length.*

Definition 4 (Variable Input Length) *A function H mapping domain $\mathcal{D}$ to $\mathcal{R}$ is a variable input length (VIL) function if $\mathcal{D} = \{0,1\}^*$. That is, its inputs are bit strings of variable length.*

Definition 5 (Domain Extender) *We say some algorithm* A *is a domain extender for property* P *if, given a fixed input length function* f_Q *that is FIL-*P *(that is, has the property of* P *for fixed input length), then* A^f *(*A *when given oracle access to* f_Q*) is a variable input length function that has the property of VIL-*P *(that is, it has the property of* P *for variable input length).*

Note that traditionally a domain extender only extends the domain of the function and provides no guarantees that the new function satisfies any properties. However, in our context we will mean that A *is a domain extender and a property* P *preserver.*

Definition 6 (Running Time) *Oftentimes we will say that an algorithm* A *has "running time"* t*. By this we mean that* t *includes both the amount of time taken by an invocation of* A *as well as the size of the code implementing* A*, measured in some fixed RAM model of computation (as in [3]).*

Definition 7 (Mode of Operation) *Throughout this chapter we will use* $\mathcal{M}$ *to denote MD6's mode of operation. This is defined irrespective of the compression function used, although we will often use* $\mathcal{M}^{f_Q} : \mathbf{W}^k \times \{0,1\}^* \to \{0,1\}^d$ *to denote the MD6 mode of operation applied to the compression function* f_Q *(i.e.* $\mathcal{M}^{f_Q}(M)$ *is the MD6 hash of a message* M*), the superscript denoting that the mode of operation only makes black-box use of* f_Q*. When needed, we will parameterize* $\mathcal{M}^{f_Q}$ *by* L*, denoting this as* $\mathcal{M}_L^{f_Q}$*.*

On occasion we will consider the output of $\mathcal{M}^{f_Q}$ *without the final compression function application. Recall that the final compression function has the* z *bit set to 1, and then its output is chopped from* c *words down to* d *bits. Therefore we write the mode of operation without the final compression function application as* $\mathcal{H}^{f_Q} : \mathbf{W}^k \times \{0,1\}^* \to \mathbf{W}^{n-q-k}$*, where*

$$\mathcal{M}^{f_Q} = \chi_d \circ f \circ \mathcal{H}^{f_Q}.$$

Recall that the chop function $\chi_d : \{0,1\}^* \to \{0,1\}^d$ *operates simply by returning the last* d*-bits of its input. We can abbreviate the action of the final compression function and the chop by defining* $g = (\chi_d \circ f_Q) : \mathbf{W}^k \times \mathbf{W}^{n-q-k} \to \{0,1\}^d$*.*

Definition 8 (Height) *For a given maximum level parameter* L*, let the height of the MD6 hash tree parameterized by* L *on a message* M *of length* $\mu = |M|$ *be given as* $height_L(\mu)$*. Recall that the height of a tree is given by the distance of the root node from the farthest leaf node. For example, the tree in Figure 2.1 has height 3, the tree in Figure 2.3 has height 6, and the tree in Figure 2.2 has height 18.*

Oftentimes we will have some adversary A that we grant q queries $M_1, \ldots, M_q$ to the MD6 hash algorithm $\mathcal{M}^{f_Q}$, where we bound the total length of the queries, $\sum_i |M_i| \leq \mu$. One question we might ask is, in the course of execution of $\mathcal{M}^{f_Q}$, how many queries to f_Q will need to be made, given the above resource constraints?

Lemma 2 *Assume we are given oracle access to $f_Q(K,\cdot)$ for some K. Then if we have q messages $M_1, M_2, \ldots, M_q$ such that*

$$\sum_{i=1}^{q} |M_i| \leq \mu,$$

then in order to compute $D_i = \mathcal{M}_L^{f_Q(K,\cdot)}(M_i)$ for all i we require at most $\delta(q,\mu)$ queries to the oracle $f_Q(K,\cdot)$, where

$$\delta(q,\mu) = \frac{1}{3} \cdot \left\lceil \frac{\mu}{c} \right\rceil + q \log_4 \left(\frac{1}{q} \cdot \left\lceil \frac{\mu}{c} \right\rceil \right) + \frac{q}{3}.$$

Proof: The proof of this lemma is fairly straightforward, so we refer the interested reader to Lemma 2.1 of [32]

7.2 Collision-Resistance

To begin, we first define what it means for a keyed fixed input length (FIL) compression function f_Q to be collision resistant, and what it means for a keyed variable input length (VIL) hash function H to be collision resistant.

Definition 9 (FIL-Collision Resistance) *For a keyed FIL function f_Q mapping $\mathbf{W}^k \times \mathbf{W}^{n-q-k} \rightarrow \mathbf{W}^c$, define the advantage of an adversary A for finding a collision as*

$$\boldsymbol{Adv}_{\mathsf{A}}^{\textit{fil-cr}} = \Pr\left[\begin{array}{c|c} K \xleftarrow{\$} \mathbf{W}^k; & m \neq m', \\ (m,m') \leftarrow \mathsf{A}(K) & f_Q(K,m) = f_Q(K,m') \end{array} \right]$$

We define the insecurity of f_Q with respect to FIL-collision resistance (FIL-CR) as

$$\boldsymbol{InSec}_{f_Q}^{\textit{fil-cr}}(t) = \max_{\mathsf{A}} \left\{ \boldsymbol{Adv}_{\mathsf{A}}^{\textit{fil-cr}} \right\},$$

where the maximum is taken over all adversaries A with total running time t.

Definition 10 (VIL-Collision Resistance) *For a keyed VIL function H mapping $\mathbf{W}^k \times \{0,1\}^* \rightarrow \{0,1\}^d$, define the advantage of an adversary A for finding a collision as*

$$\boldsymbol{Adv}_{\mathsf{A}}^{\textit{vil-cr}} = \Pr\left[\begin{array}{c|c} K \xleftarrow{\$} \mathbf{W}^k; & M \neq M', \\ (M,M') \leftarrow \mathsf{A}(K) & H(K,M) = H(K,M') \end{array} \right]$$

We define the insecurity of f_Q with respect to VIL-collision resistance (VIL-CR) as

$$\boldsymbol{InSec}_{H}^{\textit{vil-cr}}(t) = \max_{\mathsf{A}} \left\{ \boldsymbol{Adv}_{\mathsf{A}}^{\textit{vil-cr}} \right\},$$

where the maximum is taken over all adversaries A with total running time t.

Here we add the additional assumption that A must have computed the hashes of these messages $H(K, M)$ and $H(K, M')$ at some point (for the message pair (M, M') that it returned), and so the total time allotment t includes the cost of computing these hashes. This is a reasonable assumption — not without precedent [3, Section 4.1] — as A should at least verify that $H(K, M) = H(K, M')$.

We now give a series of lemmas that culminate in demonstrating how one can use a collision in MD6 to construct a collision in the underlying compression function. These lemmas rely heavily on the detailed description of the MD6 mode of operation given in Section 2.4. We give a sketch of our approach as follows. The MD6 mode of operation makes use of two operations, PAR and SEQ. The global parameter L determines the maximum number of times that the operation PAR is invoked (recursively) on the input message. If the length of the input is long enough that after (up to) L iterations of PAR the resulting output is still larger than d bits, the SEQ operation is performed to bring the final digest to d bits. Therefore our lemmas will be of the following form: if we have two messages that form a collision in PAR or SEQ, we can examine the intermediary values produced in the computation of PAR or SEQ to find a collision either in the compression function f_Q or in the final compression function $g = \chi_d \circ f_Q$. We will conclude by noticing that if there is a collision in the overall MD6 hash, then there must be a collision in one of the PAR or SEQ operations.

Lemma 3 *Suppose $M, M' \in \{0,1\}^*$ are distinct and further,*

$$\mathrm{SEQ}(M, d, K, L, r) = \mathrm{SEQ}(M', d, K, L, r).$$

Then we can construct $m, m' \in \mathbf{W}^{n-q-k}$ with $m \neq m'$ such that $f_Q(K, m) = f_Q(K, m')$ or $g(K, m) = g(K, m')$.

Proof: Recall from Section 2.4 that the MD6 SEQ operation performs analogously to the Merkle-Damgård construction, and therefore this proof will proceed similarly to the well-established collision resistance proofs of security [34]. As shown in Figure 2.6, we begin by padding out M and M' with zeroes to be a positive integral multiple of $(b - c) = 48$ words, and subdivide each message into sequences $M = B_0 \| B_1 \| \cdots \| B_{j-1}$ and $M' = B'_0 \| B'_1 \| \cdots \| B'_{j'-1}$ of j and j' (respectively) $(b - c)$-word blocks. Before we can proceed, we must first define some intermediary values that we use throughout our proof. Let

$$m_i = U_i \| V_i \| C_{i-1} \| B_i,$$

where we define the auxiliary fields,

$$
\begin{aligned}
U_i &= L \cdot 2^{56} + i \\
V_i &= r \| L \| z_i \| p_i \| keylen \| d \\
z_i &= \begin{cases} 1, & \text{if } i = j-1 \\ 0, & \text{otherwise} \end{cases} \\
p_i &= \begin{cases} \#\text{ of padding bits in } B_{j-1}, & \text{if } i = j-1 \\ 0, & \text{if } i < j-1 \end{cases}
\end{aligned}
$$

and the chaining variables,

$$
\begin{aligned}
C_{-1} &= 0^{64w} \\
C_i &= f_Q(K, m_i), \text{ for } 0 \le i < j-1 \\
C_{j-1} &= g(K, m_{j-1}) = \chi_d(f_Q(K, m_{j-1})).
\end{aligned}
$$

We also define m'_i similarly for the B'_i values.

Now first suppose that $j \neq j'$; that is, the number of blocks in the input messages differ. Then we are easily able to construct a collision between the inputs to the final compression function in SEQ. Since $j \neq j'$ we have $U_{j-1} \neq U'_{j'-1}$, as the binary representation of $j-1$ and $j'-1$ are encoded into these values. This clearly implies that $m_{j-1} \neq m'_{j'-1}$. Moreover, we have our collision, since

$$
\begin{aligned}
g(K, m_{j-1}) = C_{j-1} &= \text{SEQ}(M, d, K, L, r) \\
&= \text{SEQ}(M', d, K, L, r) = C'_{j'-1} = g(K, m'_{j'-1}).
\end{aligned}
$$

On the other hand, suppose that $j = j'$ and the number of blocks in the input messages are the same. We show that at some point along the chain of computations, a collision occurs. That is, the event $C_i = C'_i$ implies that either there is a collision at block i, or the previous chaining variables are equal, $C_{i-1} = C'_{i-1}$. Thus starting from the assumption that $C_{j-1} = C'_{j-1}$ (i.e. that the outputs of SEQ are equal), we can work our way backwards through the chain to find a collision.

First, suppose that $m_{j-1} \neq m'_{j-1}$. Since $g(K, m_{j-1}) = C_{j-1} = C'_{j-1} = g(K, m'_{j-1})$, we have a collision. Now, on the contrary, suppose that $m_{j-1} = m'_{j-1}$. Then this implies that $C_{j-2} = C'_{j-2}$.

Now, fix some i such that $0 \le i < j-1$ and suppose that $C_i = C'_i$. Then either $m_i \neq m'_i$ and we have a collision $f_Q(K, m_i) = C_i = C'_i = f_Q(K, m'_i)$, or $m_i = m'_i$ and therefore $C_{i-1} = C'_{i-1}$. Thus by starting from $C_{j-1} = C'_{j-1}$ and walking backwards along the chain of computations, we either find a collision or, for each i, $m_i = m'_i$. In particular, if $m_i = m'_i$ for all i, then this implies that $B_i = B'_i$ for all i as well. Furthermore, this implies that the number of padding

bits are equal, $p_{j-1} = p'_{j-1}$. However, if this is the case then it must be that $M = M'$, which is a contradiction. Therefore a collision in f_Q or g must occur at some point along the chain.

Lemma 4 *Suppose $M, M' \in \{0,1\}^*$ are distinct and further,*

$$\mathrm{PAR}(M, d, K, L, r, \ell) = \mathrm{PAR}(M', d, K, L, r, \ell).$$

Then we can construct $m, m' \in \mathbf{W}^{n-q-k}$ with $m \neq m'$ such that $f_Q(K, m) = f_Q(K, m')$.

Proof: We proceed much in the same fashion as Lemma 3, but this proof is simpler because of the parallel nature of PAR. Following the definition of PAR shown in Figure 2.5, we pad out M and M' with zeroes until their lengths are multiples of $b = 64$ words, and subdivide each message into sequences $M = B_0 \| B_1 \| \cdots \| B_{j-1}$ and $M' = B'_0 \| B'_1 \| \cdots \| B'_{j'-1}$ of j and j' (respectively) b-word blocks. As before, we define the intermediary variables we will use in this proof. Let

$$m_i = U_i \| V_i \| B_i,$$

where we define the auxiliary fields,

$$\begin{aligned}
U_i &= \ell \cdot 2^{56} + i \\
V_i &= r \| L \| z \| p_i \| keylen \| d \\
z &= \begin{cases} 1, & \text{if } j = 1 \\ 0, & \text{otherwise} \end{cases} \\
p_i &= \begin{cases} \#\text{ of padding bits in } B_{j-1}, & \text{if } i = j-1 \\ 0, & \text{if } i < j-1 \end{cases}
\end{aligned}$$

and the output variables,

$$C_i = f_Q(K, m_i)$$

We also define m'_i similarly for the B'_i values.

From our definitions,

$$\begin{aligned}
C_0 \| C_1 \| \cdots \| C_{j-1} &= \mathrm{PAR}(M, d, K, L, r, \ell) \\
&= \mathrm{PAR}(M', d, K, L, r, \ell) = C'_0 \| C'_1 \| \cdots \| C'_{j'-1},
\end{aligned}$$

and therefore $j = j'$.

Moreover, if there exists an i such that $m_i \neq m'_i$, then we have a collision, as $f_Q(K, m_i) = C_i = C'_i = f_Q(K, m'_i)$. Now suppose that for all i, $m_i = m'_i$. This implies that for all i, the input messages $B_i = B'_i$ and the number of padding bits $p_i = p'_i$. Therefore it must be the case that $M = M'$, which is a contradiction.

Theorem 2 *Suppose $M, M' \in \{0,1\}^*$ and L, L', K, K' provide a collision in the hash function $\mathcal{M}^{f_Q}$; that is, $\mathcal{M}_L^{f_Q}(K, M) = \mathcal{M}_{L'}^{f_Q}(K', M')$. Then we can construct $m, m' \in \mathbf{W}^{n-q-k}$ with $m \neq m'$ such that $f_Q(K, m) = f_Q(K', m')$ or $g(K, m) = g(K', m')$.*

Proof: Let ℓ and ℓ' be the "layered height" of each hash tree. That is, in the computation of MD6, ℓ and ℓ' are the number of total applications of PAR and SEQ on m and m', respectively. Note this is not the height of the hash tree, since the root node of the graph in Figure 2.3 has height 6. Rather, it is the "level" of the root node in the computation — plus one, if SEQ has been applied. Thus the layered height of the hash tree in Figure 2.3 is 3, as is the layered height of the hash tree in Figure 2.1. From this point, we assume that $L = L'$, $K = K'$, and $\ell = \ell'$, since these parameters (L, K, ℓ) are included as part of the input to the final compression function g. If this is not the case, then there is a collision in g and we are done. Thus we drop the prime $'$ in the variable names and consider only L, K, and ℓ.

The rest of the proof will follow substantially from Lemmas 3 and 4. As in the definition of MD6 (see Figure 2.4), we use the following intermediary variables,

$$
\begin{aligned}
&M_0 = M \\
&M_i = \mathrm{PAR}(M_{i-1}, d, K, L, r, i), \text{ for } 1 \leq i < \ell \\
&D = M_\ell = \begin{cases} \mathrm{SEQ}(M_{\ell-1}, d, K, L, r), & \text{if } \ell = L + 1 \\ \chi_d(\mathrm{PAR}(M_{\ell-1}, d, K, L, r, i)), & \text{otherwise} \end{cases}
\end{aligned}
$$

with D' and M_i' defined similarly for M'.

By Lemma 4, $M_i \neq M_i'$ implies that either $M_{i+1} \neq M_{i+1}'$ or we can find a collision in one of the compression functions f_Q used at level i. Therefore, moving from the bottom of the hash tree up (starting from the condition $M_0 \neq M_0'$) we either find a collision in f_Q or reach level $\ell - 1$ with $M_{\ell-1} \neq M_{\ell-1}'$. If PAR is the last function executed (i.e. $\ell < L + 1$), then by Lemma 4 we have found a collision in g, since $D = D'$. If SEQ is the last function executed (i.e. $\ell = L + 1$), then by Lemma 3 we have also found a collision in either f_Q or g, again because $D = D'$.

Theorem 3 *Let $f_Q : \mathbf{W}^k \times \mathbf{W}^{n-q-k} \to \mathbf{W}^c$ be a FIL-CR function, and suppose that $g = (\chi_d \circ f_Q) : \mathbf{W}^k \times \mathbf{W}^{n-q-k} \to \{0,1\}^d$ is also FIL-CR. Then $\mathcal{M}^{f_Q} : \mathbf{W}^k \times \{0,1\}^* \to \{0,1\}^d$ is a VIL-CR function with*

$$\boldsymbol{InSec}_{\mathcal{M}^{f_Q}}^{\textit{vil-cr}}(t) \leq \boldsymbol{InSec}_{f_Q}^{\textit{fil-cr}}(2t) + \boldsymbol{InSec}_{g}^{\textit{fil-cr}}(2t).$$

Proof: Suppose A is an algorithm with the best possible chance of success for breaking the VIL-collision resistance of $\mathcal{M}^{f_Q}$ among all algorithms running in time t, so that the probability of success of A is $\mathbf{InSec}^{\text{vil-cr}}_{\mathcal{M}^{f_Q}}(t)$. We construct an algorithm C that uses A as a subroutine to attack the FIL-collision resistance of either f_Q or g.

To begin, C receives as input the hash function key K; in order to succeed, it must produce messages $m \neq m'$ such that $f_Q(K, m) = f_Q(K, m')$ or $g(K, m) = g(K, m')$, and run in time no greater than $2t$. C then invokes A on the key K, which ultimately returns messages $M \neq M'$ such that $\mathcal{M}^{f_Q}(K, M) = \mathcal{M}^{f_Q}(K, M')$.

Algorithm C
Input: K, the compression function key; A, the collision-finding algorithm
Output: $m \neq m'$, such that $f_Q(K, m) = f_Q(K, m')$ or $g(K, m) = g(K, m')$
1. $(M, M') \leftarrow \mathsf{A}(K)$
2. $D \leftarrow \mathcal{M}^{f_Q}(K, M)$ and store each input to $f_Q(K, \cdot)$
3. $D' \leftarrow \mathcal{M}^{f_Q}(K, M')$ and store each input to $f_Q(K, \cdot)$
4. **if** $M \neq M'$ and $D = D'$
5. **then** Use Theorem 2 to find $m \neq m'$ that collide in $f_Q(K, \cdot)$ or $g(K, \cdot)$
6. **return** (m, m')
7. **else** **return** $(0^n, 0^n)$

By Theorem 2, from a collision in $\mathcal{M}^{f_Q}(K, \cdot)$ we can recover a collision in $f_Q(K, \cdot)$ or $g(K, \cdot)$. This recovery process takes time at most t, since it only requires computing the hashes $\mathcal{M}^{f_Q}(K, M)$ and $\mathcal{M}^{f_Q}(K, M')$ and considering the intermediate values queried to $f_Q(K, \cdot)$ in each computation. In addition, the running time of the algorithm A is at most t. Thus with probability of success at least $\mathbf{InSec}^{\text{vil-cr}}_{\mathcal{M}^{f_Q}}(t)$, we are able to find a collision in either f_Q or g in time $2t$.

7.3 Preimage (Inversion)

Recall that the property of first-preimage resistance is an important one in many cryptographic applications. For example, most computer systems store the hashes of user passwords; the inability of any adversary to invert these hashes is imperative to preserve the security of the system[2].

Since we aim to show that the MD6 mode of operation extends the property of first-preimage resistance from the compression function to the overall hash function, we precisely define what it means for both the FIL compression function and the VIL hash function to be first-preimage resistant.

Definition 11 (FIL-Preimage Resistance) *For a keyed FIL function* $f_Q : \mathbf{W}^k \times \mathbf{W}^{n-q-k} \rightarrow \mathbf{W}^c$*, define the advantage of an adversary* A *for finding a*

[2]Although in practice, many systems use a salted hash for their password file to make dictionary-based inversion attacks much more difficult.

preimage as

$$\boldsymbol{Adv}^{\textit{fil-pr}}_{\mathsf{A}}(D) = \Pr\left[K \xleftarrow{\$} \mathbf{W}^k; m \leftarrow \mathsf{A}(K, D) : \quad f_Q(K, m) = D\right],$$
$$\boldsymbol{Adv}^{\textit{fil-pr}}_{\mathsf{A}} = \max_{D \in \{0,1\}^d} \left\{\boldsymbol{Adv}^{\textit{fil-pr}}_{\mathsf{A}}(D)\right\}.$$

We define the insecurity of f_Q with respect to FIL-preimage resistance (FIL-PR) as

$$\boldsymbol{InSec}^{\textit{fil-pr}}_{f}(t) = \max_{\mathsf{A}} \left\{\boldsymbol{Adv}^{\textit{fil-pr}}_{\mathsf{A}}\right\},$$

where the maximum is taken over all adversaries A with total running time t.

Definition 12 (VIL-Preimage Resistance) *For a keyed VIL function $H : \mathbf{W}^k \times \{0,1\}^* \to \{0,1\}^d$, define the advantage of an adversary A for finding a preimage as*

$$\boldsymbol{Adv}^{\textit{vil-pr}}_{\mathsf{A}}(D) = \Pr\left[K \xleftarrow{\$} \mathbf{W}^k; M \leftarrow \mathsf{A}(K, D) : \quad H(K, M) = D\right],$$
$$\boldsymbol{Adv}^{\textit{vil-pr}}_{\mathsf{A}} = \max_{D \in \{0,1\}^d} \left\{\boldsymbol{Adv}^{\textit{vil-pr}}_{\mathsf{A}}(D)\right\}.$$

We define the insecurity of H with respect to VIL-preimage resistance (VIL-PR) as

$$\boldsymbol{InSec}^{\textit{vil-pr}}_{H}(t) = \max_{\mathsf{A}} \left\{\boldsymbol{Adv}^{\textit{vil-pr}}_{\mathsf{A}}\right\},$$

where the maximum is taken over all adversaries A with total running time t. As before, we make the reasonable assumption that computing $H(K, M)$ counts towards the total time allotment of t, since we assume any preimage-finding algorithm must at least verify that M is a valid preimage of D.

Note that these definitions differ from the commonly regarded notion of preimage resistance (Pre) as defined by Rogaway and Shrimpton [86]:

$$\mathbf{Adv}^{\mathrm{Pre}}_{\mathsf{A}} = \Pr\left[\begin{array}{c} K \xleftarrow{\$} \mathbf{W}^k; M \xleftarrow{\$} \mathbf{W}^{n-q-k}; \\ D \leftarrow f_Q(K, M); M' \leftarrow \mathsf{A}(K, D) \end{array} : \quad f_Q(K, M') = D\right]$$

In particular, our definitions given above are a stricter notion of preimage resistance that Rogaway and Shrimpton term "everywhere preimage-resistance" (ePre); this definition attempts to capture the infeasibility of finding a preimage for a given D, over all choices of D. We adopt this definition because it simplifies our analysis. Since everywhere preimage-resistance implies preimage resistance [86], we do not lose any security by doing so.

We begin by demonstrating, via reduction, that the MD6 hash function is VIL-preimage resistant so long as its underlying compression function is FIL-preimage resistant.

Theorem 4 *Let $f_Q : \mathbf{W}^k \times \mathbf{W}^{n-q-k} \to \mathbf{W}^c$ be a FIL-PR function, and suppose that $g = (\chi_d \circ f) : \mathbf{W}^k \times \mathbf{W}^{n-q-k} \to \{0,1\}^d$ is also FIL-PR. Then $\mathcal{M}^{f_Q} : \mathbf{W}^k \times \{0,1\}^* \to \{0,1\}^d$ is a VIL-PR function with*

$$\boldsymbol{InSec}^{\textit{vil-pr}}_{\mathcal{M}^{f_Q}}(t) \leq \boldsymbol{InSec}^{\textit{fil-pr}}_{g}(2t).$$

Proof: Suppose that A is an algorithm with the best possible chance of success for breaking the VIL-preimage resistance of $\mathcal{M}^{f_Q}$ among all algorithms running in time t, so that the probability of success of A is $\mathbf{InSec}^{\text{vil-cr}}_{\mathcal{M}^{f_Q}}(t)$. We construct a new algorithm P, running in time at most $2t$, that uses A as a subroutine to attack the FIL-preimage resistance of g.

The behavior of P is straightforward: if A manages to find a valid preimage, then P can simply compute the hash and return the input to the final compression function g. However, some care must be taken in the analysis because, as mentioned in the definition of preimage resistance, the choice of target digest D that maximizes the advantage can depend on the algorithm P used.

Algorithm P
Input: K, the compression function key; A, the preimage-finding algorithm; D, the target digest
Output: m, such that $g(K, m) = D$
1. $M \leftarrow \mathsf{A}(K, D)$
2. $D' \leftarrow \mathcal{M}^{f_Q}(K, M)$
3. $m \leftarrow$ the input to the final compression $g(K, \cdot)$ in the computation of D'
4. **if** $D = D'$
5. **then return** m
6. **else return** 0^n

To begin with, P receives as input a digest $D \in \{0,1\}^d$, the key K, and the $\mathcal{M}^{f_Q}$ preimage-finding algorithm A, and its goal is to produce a preimage $m \in \mathbf{W}^{n-q-k}$ such that $g(K, m) = D$. Next, P invokes A on the target digest D and key K to receive a message $M \in \{0,1\}^*$ such that, with some probability, $\mathcal{M}^{f_Q}(K, M) = D$. If M is indeed a preimage of D, then letting $m \in \mathbf{W}^{n-q-k}$ be the input given to the final compression function $g(K, \cdot)$ in the computation of $\mathcal{M}^{f_Q}(K, M)$ will indeed give a preimage of D.

Now we wish to show that the advantage of P is at least the advantage of A. Let $\widehat{D}$ be the value of the target digest D that maximizes the advantage of A:

$$\widehat{D} = \underset{D \in \{0,1\}^d}{\arg\max} \left\{ \Pr\left[K \xleftarrow{\$} \mathbf{W}^k; M \leftarrow \mathsf{A}(K, D) : \quad \mathcal{M}^{f_Q}(K, M) = D \right] \right\}.$$

Then the advantage of P when given target digest $\widehat{D}$ is at most its advantage over the best possible D, so that

$$\begin{aligned} \mathbf{InSec}^{\text{vil-pr}}_{\mathcal{M}^{f_Q}}(t) \leq \mathbf{Adv}^{\text{vil-pr}}_{\mathsf{A}} &= \mathbf{Adv}^{\text{vil-pr}}_{\mathsf{A}}(\widehat{D}) \\ &\leq \mathbf{Adv}^{\text{fil-pr}}_{\mathsf{P}}(\widehat{D}) \leq \mathbf{Adv}^{\text{fil-pr}}_{\mathsf{P}} \leq \mathbf{InSec}^{\text{fil-pr}}_{g}(2t). \end{aligned}$$

In order to prove the bound for running time, notice that it takes time at most t to run $\mathsf{A}(K, D)$, and by our earlier assumption it takes time at most t to compute $\mathcal{M}^{f_Q}(K, M)$. Therefore the total time is at most $2t$.

7.4 Second Pre-image

Second-preimage resistance is defined as the computational infeasibility of any adversary, given a target message m, to produce a different message m' such that these two messages hash to the same value. Clearly, second-preimage resistance is a potentially stronger assumption than collision resistance, since producing second preimages also yields hash function collisions. Therefore, as in other treatments of this problem [87], it suffices in general to prove collision resistance, which we demonstrated earlier in Section 7.2.

Unfortunately, trying to prove a reduction of the FIL-second-preimage resistance of the compression function to the VIL-second-preimage resistance of the overall hash function fails to work naturally. The problem with the reduction is that we have some algorithm A that can break the second-preimage resistance of $\mathcal{M}^{f_Q}$ and we want to construct an algorithm S that uses A to break the second-preimage resistance of f_Q. So S receives a message $m \in \mathbf{W}^{n-q-k}$ and a key $K \in \mathbf{W}^k$ and must find $m' \in \mathbf{W}^{n-q-k}$ such that $m \neq m'$ and $f_Q(K, m) = f_Q(K, m')$. However, attempting to invoke A on m directly will not succeed. In particular, A is only guaranteed to succeed with some probability over the choice of K and $M \in \{0, 1\}^*$. In particular, A could be excellent at finding second-preimages when given a target M such that $|M| > n - q - k$, but absolutely miserable when $|M| = n - q - k$. Therefore we are unable to translate the success of A into the success of S and the reduction fails.

Although it seems like it should be possible to perform such a reduction, we know of no approach for reducing the property successfully. In addition, we do not know of any similar attempts in the literature to prove domain extension for second-preimage resistance. Therefore we will simply say that the collision resistance of MD6 is secure with $d/2$ bits of security, therefore it follows that the second-preimage resistance of MD6 is secure with at least $d/2$ bits of security and hope that suffices.

However, we would like to specifically address the security requirements in the NIST SHA-3 hash function specifications [70], with respect to second-preimage resistance. According to those specifications, for a hash function with a d bit message digest and a target preimage of 2^k bits, the hash function should have second-preimage security of approximately $d - k$ bits.

The rationale for this condition is the following. In a hash function with an iterative mode of operation (such as the plain Merkle-Damgård construction), a target preimage m consisting of 2^k message blocks forms a chain of 2^k invocations of the compression function. Therefore an adversary wishing to perform a second preimage attack on m can simply pick a random r and compute $f_Q(IV, r) = y$. It can then check whether y matches any of the 2^k compression function outputs y_i. If so, it outputs the message $m' = r \| m_{i+1} \| \cdots \| m_{2^k}$, which

is a valid collision with m.

This attack works only because the length of the message m is not encoded in the hash, so an attacker is able to substitute the prefix r anywhere into the message chain. One simple method to foil this attack is to append the length of the message to the end of the message, which prevents an attacker from being able to substitute a truncated message that collides with m. However, even this approach succumbs to similar cryptographic attacks, as demonstrated by Kelsey and Schneier [50].

MD6 behaves differently from these approaches. Each compression function used is given control words U and V in the input that label each compression function with its position in the hash tree. Therefore the above attacks against Merkle-Damgård and strengthened Merkle-Damgård are foiled since the adversary is no longer given the 2^k-for-1 advantage that it enjoyed for the Merkle-Damgård mode of operation (effectively, it is not able to query a substructure of the hash function).

7.5 Pseudo-Random Function

Pseudorandomness is a useful property for a hash function to have. For one, pseudorandomness implies unpredictability, meaning a pseudorandom hash function can perform as a message authentication code (MAC) [42, 41]. In addition, many cryptographic protocols are proved to be secure in the so-called "random oracle model" [37, 11], which assumes the existence of an oracle that maps $\{0,1\}^*$ into some fixed output domain $\mathcal{D}$. The oracle is a black box that responds to queries in $\{0,1\}^*$ with a uniformly random response chosen from $\mathcal{D}$ (except for inputs that have been queried previously, whereupon it is consistent). In practice, protocols that assume the existence of a random oracle use a cryptographic hash function instead, in which case we desire that the hash function family be pseudorandom or in some sense indistinguishable from a random oracle. Unfortunately, random oracles do not actually exist in real life, and therefore proofs in the random oracle model only provide a heuristic for security [24]. Nevertheless, it is still desirable to be able to show that a cryptographic hash function family is pseudorandom, under certain assumptions on the compression function used.

Previous works have shown that the Cipher Block Chaining mode of operation is a domain extender for the property of pseudorandomness [8, 9, 60]. In this chapter we will demonstrate that the MD6 mode of operation also acts as a domain extender for fixed-length pseudorandom functions.

7.5.1 Maurer's Random System Framework

Throughout much of this section we use key concepts from the Random Systems framework developed by Ueli Maurer [60]. Many of these definitions and lemmas and much of the terminology are composites of several related papers [60, 77, 61]

7.5.1.1 Notation

We generally adhere to the notation used in previous work. Characters in a calligraphic font (such as $\mathcal{X}$ or $\mathcal{Y}$) denote sets, and their corresponding italicized roman characters X and Y denote random variables that take values in $\mathcal{X}$ and $\mathcal{Y}$ (with some distribution). Superscripts for sets and random variables generally denote a tuple, so $\mathcal{X}^i = \underbrace{\mathcal{X} \times \cdots \times \mathcal{X}}_{i}$ and $X^i = (X_1, \ldots, X_i)$ is a random variable over $\mathcal{X}^i$. We reserve bold-face characters for random systems, which are defined below.

7.5.1.2 Definitions

In order to reason about the complicated interactions of certain cryptographic systems, it is helpful to use the Random Systems framework of Maurer [60]. In particular, for some cryptographic system $\mathbf{S}$ and for each i, $\mathbf{S}$ takes in an input X_i and produces (probabilistically) a corresponding output Y_i (in sequence, so next it takes input X_{i+1} and produces Y_{i+1}). If $\mathbf{S}$ is stateless, then Y_i depends only on X_i; however, we can also consider $\mathbf{S}$ as possibly stateful, and so Y_i depends on the totality of the previous values $X_1, X_2, \ldots, X_i$ and $Y_1, Y_2, \ldots, Y_{i-1}$ (which are referred to as X^i and Y^{i-1} for convenience).

Thus the behavior of the random system $\mathbf{S}$ can be defined as a sequence of conditional probability distributions, as follows.

Definition 13 (Random System) *A $(\mathcal{X}, \mathcal{Y})$-random system $\mathbf{F}$ is an infinite sequence of conditional probability distributions*

$$\mathbf{F} = \left\{ \mathsf{Pr}_{Y_i | X^i Y^{i-1}} \right\}_{i=1}^{\infty}.$$

Collectively, we denote this sequence as $\mathsf{Pr}^{\mathbf{F}}_{Y_i | X^i Y^{i-1}}$, the superscript denoting which random system this distribution corresponds to.

Definition 14 (Equivalence of Random Systems) *Two random systems $\mathbf{F}$ and $\mathbf{G}$ are said to be equivalent, written $\mathbf{F} \equiv \mathbf{G}$, if*

$$\text{for all } i \geq 1,\ \mathsf{Pr}^{\mathbf{F}}_{Y_i | X^i Y^{i-1}} \equiv \mathsf{Pr}^{\mathbf{G}}_{Y_i | X^i Y^{i-1}},$$

or equivalently, for all $i \geq 1, (y_1, \ldots, y_i) \in \mathcal{Y}^i,\ (x_1, \ldots, x_i) \in \mathcal{X}^i$,

$$\mathsf{Pr}^{\mathbf{F}}_{Y_i | X^i Y^{i-1}}(x_1, \ldots, x_i, y_1, \ldots, y_i) = \mathsf{Pr}^{\mathbf{G}}_{Y_i | X^i Y^{i-1}}(x_1, \ldots, x_i, y_1, \ldots, y_i).$$

Definition 15 (Random Function) *A random function $\mathbf{F} : \mathcal{X} \to \mathcal{Y}$ is a random variable that takes as values functions on $\mathcal{X} \to \mathcal{Y}$ (with some given distribution). Therefore $\mathbf{F}$ is also a (stateless) random system, where*

$$\mathsf{Pr}^{\mathbf{F}}_{Y_i | X^i Y^{i-1}} = \mathsf{Pr}^{\mathbf{F}}_{Y_i | X_i}$$

and this distribution is determined by the distribution of $\mathbf{F}$ on $\mathcal{X} \to \mathcal{Y}$.

Example 1 *Consider the following random functions* $\mathbf{R}$ *and* $\mathbf{P}$*.*

Uniform Random Function *Let* $\mathbf{R} : \mathcal{X} \to \mathcal{Y}$ *denote the random function with a uniform distribution over the space of all functions mapping* $\mathcal{X} \to \mathcal{Y}$*.*

Uniform Random Permutation *Let* $\mathbf{P} : \mathcal{X} \to \mathcal{X}$ *denote the random function with a uniform distribution over the space of all permutations mapping* $\mathcal{X} \to \mathcal{X}$*.*

It is a well-known fact that if we are only given $o\left(\sqrt{|\mathcal{X}|}\right)$ *queries, it is difficult to distinguish between a uniform random function* $\mathbf{R} : \mathcal{X} \to \mathcal{X}$ *and a uniform random permutation* $\mathbf{P} : \mathcal{X} \to \mathcal{X}$*. For illustrative purposes, we will prove this fact via the random system framework in Example 2.*

We now attempt to develop the formal notion of monotone conditions for random systems. Intuitively, we are trying to capture some series of events that occur on the choices of inputs and outputs of the random system. For example, suppose we are trying to distinguish between $\mathbf{R}$ and $\mathbf{P}$ given q queries, as above. If we condition on the event that we have not observed any collisions in the output of $\mathbf{R}$, then the distribution on the outputs of each random function are identical (and therefore we have no hope of being able to distinguish them). Thus in this example we might say that the monotone condition is "the event that we have not observed a collision in the output of $\mathbf{R}$ up to query i". We can formalize this intuitive notion as follows.

Definition 16 (Monotone Conditions) *A monotone condition* $\mathcal{A}$ *for a random system* $\mathbf{F}$ *is an infinite sequence* $(A_1, A_2, \dots)$ *of events with an additional monotonicity condition. We define* A_i *to be the event that the specified condition is satisfied after query* i*, and* $\overline{A_i}$ *is the negation of this event (the specified condition is not satisfied after query* i*). The monotonicity of the condition* $\mathcal{A}$ *means that once the event is not satisfied for a given query* i*, it will not be satisfied after further queries (so,* $\overline{A_i} \implies \overline{A_{i+1}}$*).*

We can additionally define the random system $\mathbf{F}$ *conditioned on* $\mathcal{A}$*,* $\mathbf{F} \,|\, \mathcal{A}$*, to be the sequence of conditional probability distributions*

$$\mathsf{Pr}^{\mathbf{F}}_{Y_i | X^i Y^{i-1} A_i}, \text{ for all } i \geq 1$$

which are simply the distribution on the output Y_i *conditioned on the previous state* $X^{i-1}Y^{i-1}$*, the current query* X_i*, and the monotone condition* A_i*. Note that we do not need to condition on the event* $A_1 \wedge \cdots \wedge A_i$*, since* $A_i \implies A_{i-1}$*. For a more formal definition, see [60, 61].*

To go back to our earlier example, it is clear that the no-collisions condition is monotone, because observing a collision after query i implies that a collision has been observed after any further queries. As we will show in Example 2, when we condition $\mathbf{R}$ on this no-collision monotone condition $\mathcal{A}$, the resulting distribution $\mathbf{R} \,|\, \mathcal{A}$ is equivalent to the uniform random permutation $\mathbf{P}$.

Definition 17 (Distinguisher) *An adaptive distinguisher for $(\mathcal{X}, \mathcal{Y})$-random systems is defined as a $(\mathcal{Y}, \mathcal{X})$-random system $\mathbf{D}$ that interactively and adaptively queries $(\mathcal{X}, \mathcal{Y})$-random systems and ultimately outputs a bit D_q after some number of queries q. If $\mathbf{D}$ is a non-adaptive distinguisher, it must first fix its queries $X_1, \ldots, X_q$ in advance before receiving the outputs $Y_1, \ldots, Y_q$ and outputting its decision bit D_q.*

The random experiment when we pair the distinguisher $\mathbf{D}$ with an $(\mathcal{X}, \mathcal{Y})$-random system $\mathbf{F}$ (where $\mathbf{D}$ submits a query to $\mathbf{F}$, $\mathbf{F}$ responds to $\mathbf{D}$, $\mathbf{D}$ submits another query, and so forth) is denoted by $\mathbf{D} \diamond \mathbf{F}$.

Definition 18 (Advantage) *We denote the advantage of a distinguisher $\mathbf{D}$ given q queries for distinguishing two $(\mathcal{X}, \mathcal{Y})$-random systems $\mathbf{F}$ and $\mathbf{G}$ as $\Delta_q^{\mathbf{D}}(\mathbf{F}, \mathbf{G})$. There are several equivalent formal definitions. We can say that*

$$\Delta_q^{\mathbf{D}}(\mathbf{F}, \mathbf{G}) = \left| \mathsf{Pr}^{\mathbf{D} \diamond \mathbf{F}}[D_q = 1] - \mathsf{Pr}^{\mathbf{D} \diamond \mathbf{G}}[D_q = 1] \right|,$$

which requires that the decision bit D_q was computed optimally for this definition to be precise, or we can also say that the advantage is the statistical difference between the distributions $\mathbf{D} \diamond \mathbf{F}$ and $\mathbf{D} \diamond \mathbf{G}$ (which are distributions over the space $\mathcal{X}^q \times \mathcal{Y}^q$)

$$\Delta_q^{\mathbf{D}}(\mathbf{F}, \mathbf{G}) = \frac{1}{2} \sum_{\mathcal{X}^q \times \mathcal{Y}^q} \left| \mathsf{Pr}^{\mathbf{D} \diamond \mathbf{F}}_{X^q Y^q} - \mathsf{Pr}^{\mathbf{D} \diamond \mathbf{G}}_{X^q Y^q} \right|.$$

The advantage of the best distinguisher on random systems $\mathbf{F}$ and $\mathbf{G}$ can be defined as

$$\Delta_q(\mathbf{F}, \mathbf{G}) = \max_{\mathbf{D}} \ \Delta_q^{\mathbf{D}}(\mathbf{F}, \mathbf{G}).$$

On occasion we may want to restrict ourselves to only distinguishers $\mathbf{D}$ given certain resource constraints. Thus by $\Delta_{q,\mu}(\mathbf{F}, \mathbf{G})$ we mean the maximum advantage over all adversaries given q queries of total bit-length μ. If we wish to furthermore constrain their running time by t, we specify this as $\Delta_{t,q,\mu}(\mathbf{F}, \mathbf{G})$.

Going back to our earlier no-collision example, we have a random function $\mathbf{R}$ and the no-collision monotone condition $\mathcal{A}$ on $\mathbf{R}$. Recall that if $\mathbf{D}$ succeeds in causing A_q to be false, then it will successfully distinguish $\mathbf{R}$ from $\mathbf{P}$ (because $\mathbf{P}$ does not have collisions). If this occurs, we say that $\mathbf{D}$ has provoked $\overline{A_q}$.

Definition 19 (Provoking Failure in a Monotone Condition) *Let $\mathbf{F}$ be a random system and let $\mathcal{A}$ be some monotone condition on $\mathbf{F}$. Denote the probability that $\mathbf{D}$ provokes the condition $\mathcal{A}$ to fail after q queries (that is, provokes $\overline{A_q}$) as*

$$\nu_q^{\mathbf{D}}\left(\mathbf{D}, \overline{A_k}\right) = \mathsf{Pr}^{\mathbf{D} \diamond \mathbf{F}}_{\overline{A_q}} = 1 - \mathsf{Pr}^{\mathbf{D} \diamond \mathbf{F}}_{A_q}.$$

Denote the probability for the best such $\mathbf{D}$ to be

$$\nu_q\left(\mathbf{F}, \overline{A_k}\right) = \max_{\mathbf{D}} \ \nu_q^{\mathbf{D}}\left(\mathbf{D}, \overline{A_k}\right).$$

When we restrict ourselves to considering only non-adaptive distinguishers $\mathbf{D}$*, we use the notation*

$$\mu_q\big(\mathbf{F}, \overline{A_k}\big) = \max_{\textit{non-adaptive } \mathbf{D}} \nu_q^{\mathbf{D}}\big(\mathbf{D}, \overline{A_k}\big)\,.$$

In the prior no-collision example, if $\mathbf{D}$ is some algorithm for finding a collision in the random function $\mathbf{R}$, then clearly its success probability is bounded from above by $\nu_q\big(\mathbf{F}, \overline{A_k}\big)$.

7.5.1.3 Bounding Distinguishability

Throughout this chapter we will make use of some important lemmas due to Maurer [60], the proofs of which we will omit for the sake of brevity, as we focus on our original results. However, we will give proof sketches of Maurer's lemmas when it is helpful to do so. For a much more rigorous treatment of the theory of random systems, we refer the reader to Maurer and Pietrzak [60, 77].

Lemma 5 (Lemma 5(i) [60]) *For random systems,* $\mathbf{F}$*,* $\mathbf{G}$*,* $\mathbf{H}$*,*

$$\Delta_q(\mathbf{F}, \mathbf{H}) \leq \Delta_q(\mathbf{F}, \mathbf{G}) + \Delta_q(\mathbf{G}, \mathbf{H})\,.$$

Proof: This follows directly by the triangle inequality.

In the following lemma we show an upper bound on the advantage of a distinguisher $\mathbf{D}$ for random systems $\mathbf{F}$ and $\mathbf{G}$ based on the ability of $\mathbf{D}$ to provoke the failure of a monotone condition on $\mathbf{F}$.

Lemma 6 (Theorem 1(i) [60]) *For random systems* $\mathbf{F}$ *and* $\mathbf{G}$*, if* $\mathcal{A}$ *is some monotone condition such that* $\mathbf{F}\,|\,\mathcal{A} \equiv \mathbf{G}$*, then the advantage of the best distinguisher for* $\mathbf{F}$ *and* $\mathbf{G}$ *is bounded by the probability of provoking* $\overline{\mathcal{A}}$ *given the best adaptive strategy.*

$$\Delta_q(\mathbf{F}, \mathbf{G}) \leq \nu_q\big(\mathbf{F}, \overline{A_k}\big)$$

Proof Sketch 1

$$\begin{aligned}\Delta_q(\mathbf{F}, \mathbf{G}) &\leq \nu_q\big(\mathbf{F}, \overline{A_k}\big) \cdot \Delta_q\big(\mathbf{F}\,|\,\overline{\mathcal{A}}, \mathbf{G}\big) + (1 - \nu_q\big(\mathbf{F}, \overline{A_k}\big)) \cdot \Delta_q(\mathbf{F}\,|\,\mathcal{A}, \mathbf{G}) \\ &\leq \nu_q\big(\mathbf{F}, \overline{A_k}\big) \cdot 1 + (1 - \nu_q\big(\mathbf{F}, \overline{A_k}\big)) \cdot 0 \\ &= \nu_q\big(\mathbf{F}, \overline{A_k}\big)\end{aligned}$$

The first inequality holds by the law of total probability. The second is due to the fact that $\mathbf{F}\,|\,\mathcal{A} \equiv \mathbf{G}$*, hence* $\Delta_q(\mathbf{F}\,|\,\mathcal{A}, \mathbf{G}) = 0$*.*

In certain situations, adaptivity does not increase a distinguisher's advantage with respect to provoking the failure of some monotone condition. The following lemma provides a sufficient condition for this to be true.

Lemma 7 (Theorem 2 [60]) *For a random system* $\mathbf{F}$ *with a monotone condition* $\mathcal{A}$*, if there exists a random system* $\mathbf{G}$ *such that* $\mathbf{F} \mid \mathcal{A} \equiv \mathbf{G}$*, i.e.*

$$\text{for all } i \geq 1, \mathsf{Pr}^{\mathbf{F}}_{Y^i|X^iA_i} \equiv \mathsf{Pr}^{\mathbf{G}}_{Y^i|X^i}$$

then adaptivity does not help in provoking $\overline{A_q}$*:*

$$\nu_q\big(\mathbf{F}, \overline{A_k}\big) = \mu_q\big(\mathbf{F}, \overline{A_k}\big)\,.$$

Proof: See [77, Lemma 6].

To illustrate the usefulness of this framework, we return to our demonstration of the indistinguishability of a random function $\mathbf{R} : \{0,1\}^n \to \{0,1\}^n$ from a random permutation $\mathbf{P} : \{0,1\}^n \to \{0,1\}^n$ by any distinguisher $\mathbf{D}$ asking $o\big(2^{\frac{n}{2}}\big)$ queries (adapted from [77, Examples 1–3]).

Example 2 *We first begin by defining the monotone condition* $\mathcal{A} = \{A_i\}$*, where* A_i *is the event that after the* i^{th} *query all distinct inputs have produced distinct outputs. It is fairly straightforward to demonstrate that* $\mathbf{R} \mid \mathcal{A} \equiv \mathbf{P}$*: unless one has observed a collision, a random function has output distribution identical to a random permutation.*

Therefore by Lemma 6, $\Delta_q(\mathbf{R}, \mathbf{P}) \leq \nu_q\big(\mathbf{R}, \overline{A_k}\big)$*. By definition,* $\nu_q\big(\mathbf{R}, \overline{A_k}\big)$ *is the probability of success of the best distinguisher to provoke a collision (using distinct inputs) on a uniformly random function* $\mathbf{R}$*, which is clearly bounded by the Birthday Paradox (see [32, Appendix A]).*

$$\Delta_q(\mathbf{R}, \mathbf{P}) \leq \nu_q\big(\mathbf{R}, \overline{A_k}\big) \leq \frac{q(q-1)}{2^{n+1}}$$

For $q = o\big(2^{\frac{n}{2}}\big)$*, this is a negligible probability of success.*

7.5.2 MD6 as a Domain Extender for FIL-PRFs

With the random system framework and the above lemmas, we can now prove that MD6 behaves as a domain extender on FIL-PRFs that preserves pseudorandomness.

7.5.2.1 Preliminaries

Here we define some random systems used throughout this section.

- Let $\mathbf{R}$ denote the random function with uniform distribution over functions mapping $\mathbf{W}^{n-q-k} \to \mathbf{W}^c$
- Let $f_Q : \mathbf{W}^k \times \mathbf{W}^{n-q-k} \to \mathbf{W}^c$ be the compression function used. Then let $\mathbf{F} : \mathbf{W}^{n-q-k} \to \mathbf{W}^c$ denote the random function with a uniform distribution over the set

$$\big\{ f_Q(K, \cdot) \mid K \in \mathbf{W}^k \big\}\,.$$

- Let $\mathbf{O}$ denote a random function with output space $\mathcal{Y} = \{0,1\}^d$ and input space $\mathcal{X} = \{0,1\}^*$ such that for all $i \geq 1, x \in \{0,1\}^*, y \in \{0,1\}^d$,

$$\mathsf{Pr}^{\mathbf{O}}_{Y_i|X_i}(y,x) = \frac{1}{2^d}.$$

  $\mathbf{O}$ is usually referred to as a random oracle.

- For a random function $\mathbf{G} : \mathbf{W}^{n-q-k} \to \mathbf{W}^c$, let $\mathcal{H}^{\mathbf{G}}$ denote the random function mapping $\{0,1\}^* \to \{0,1\}^n$ by applying the MD6 mode of operation (without the final compression function and chop) with $\mathbf{G}$ as the compression function.

- For a random function $\mathbf{G} : \mathbf{W}^{n-q-k} \to \mathbf{W}^c$, let $\mathcal{M}^{\mathbf{G}}$ denote the random function mapping $\{0,1\}^* \to \{0,1\}^d$ by applying the MD6 mode of operation with $\mathbf{G}$ as the compression function. Thus

$$\mathcal{M}^{\mathbf{G}} = \chi_d \circ \mathbf{G} \circ \mathcal{H}^{\mathbf{G}}.$$

Our goal in this section will be to demonstrate that if $\mathbf{F}$ is a FIL-PRF (indistinguishable from $\mathbf{R}$), then $\mathcal{M}^{\mathbf{F}}$ will be a VIL-PRF (indistinguishable from $\mathbf{O}$).

Definition 20 (FIL-Pseudorandom Function) *A random function $\mathbf{G}$ mapping $\mathbf{W}^{n-q-k} \to \mathbf{W}^c$ is a (t,q,ε)-secure FIL-pseudorandom function (FIL-PRF) if it is (t,q,ε)-indistinguishable from the uniform random function on the same domain and range,*

$$\Delta_{t,q}(\mathbf{G},\mathbf{R}) \leq \varepsilon.$$

We say that $\mathbf{G}$ is a (q,ε)-secure FIL-quasirandom function (FIL-QRF) if it is a (∞,q,ε)-secure FIL-PRF. That is, we do not restrict the computational abilities of the distinguisher $\mathbf{D}$ but we restrict the number of queries that it can make. As we are considering distinguishers unconstrained by time, we omit this variable from the subscript, and write

$$\Delta_q(\mathbf{G},\mathbf{R}) \leq \varepsilon.$$

Therefore if $\mathbf{G}$ is a (q,ε)-secure FIL-QRF, then it is a (t,q,ε)-secure FIL-PRF for any choice of t.

Definition 21 (VIL-Pseudorandom Function) *A random function $\mathbf{G}$ mapping $\{0,1\}^* \to \{0,1\}^d$ is a (t,q,μ,ε)-secure VIL-pseudorandom function (VIL-PRF) if it is (t,q,μ,ε)-indistinguishable from a random oracle $\mathbf{O}$,*

$$\Delta_{t,q,\mu}(\mathbf{G},\mathbf{O}) \leq \varepsilon.$$

As above, we say that $\mathbf{G}$ is a (q,μ,ε)-secure VIL-quasirandom function (VIL-QRF) if

$$\Delta_{q,\mu}(\mathbf{G},\mathbf{O}) \leq \varepsilon.$$

7.5.2.2 Indistinguishability

Our proof that $\mathcal{M}^{\mathbf{F}}$ is indistinguishable from $\mathbf{O}$ proceeds as follows. First we notice by the triangle inequality (Lemma 5) that

$$\Delta_{q,\mu}\big(\mathcal{M}^{\mathbf{F}}, \mathbf{O}\big) \leq \Delta_{q,\mu}\big(\mathcal{M}^{\mathbf{F}}, \mathcal{M}^{\mathbf{R}}\big) + \Delta_{q,\mu}\big(\mathcal{M}^{\mathbf{R}}, \mathbf{O}\big)\,. \tag{7.1}$$

Then it only remains to bound the quantities on the right-hand side. In Lemma 8, we show how to bound the first term by the advantage for distinguishing between $\mathbf{F}$ and $\mathbf{R}$ (which by assumption is small).

Lemma 8 (Adapted from Lemma 5(ii) [60]) *For random systems* $\mathbf{G}$ *and* $\mathbf{H}$ *that map* $\mathbf{W}^{n-q-k} \to \mathbf{W}^{c}$,

$$\Delta_{t',q,\mu}\big(\mathcal{M}^{\mathbf{G}}, \mathcal{M}^{\mathbf{H}}\big) \leq \Delta_{t,\delta(q,\mu)}(\mathbf{G}, \mathbf{H})\,,$$

where $t' = t - O(\delta(q,\mu))$.

Proof: Given a distinguisher $\mathbf{D}$ for $\mathcal{M}^{\mathbf{G}}$ and $\mathcal{M}^{\mathbf{H}}$ with resource constraints t', q, and μ, we can construct an algorithm for distinguishing $\mathbf{G}$ and $\mathbf{H}$ that uses $\mathbf{D}$ as a subroutine. This algorithm responds to the queries of $\mathbf{D}$ by simulating the MD6 mode of operation $\mathcal{M}$ on $\mathbf{G}$ or $\mathbf{H}$. By Lemma 2, this requires at most $\delta(q,\mu)$ queries. Since the only work to be done is in simulating $\mathcal{M}$, this takes total time $t' + O(\delta(q,\mu))$.

For the second term of Equation (7.1), we proceed in a manner that is similar to the proof that a random function is indistinguishable from a random permutation (Example 2). That is, we will construct a monotone condition $\mathcal{A}$ such that $\mathcal{M}^{\mathbf{R}} \,\big|\, \mathcal{A} \equiv \mathbf{O}$, and we then bound the probability of provoking $\overline{\mathcal{A}}$. However, unlike a random permutation, a random oracle $\mathbf{O}$ naturally has collisions, so our previous no-collision monotone condition is not applicable. Thus we design a new monotone condition, one that involves so-called bad collisions. A bad collision in $\mathcal{M}^{\mathbf{R}}$ is one that occurs prior to the final compression function in the MD6 mode of operation. As we will show in Lemma 9, if we condition on the absence of bad collisions in $\mathcal{M}^{\mathbf{R}}$, then the distribution of its outputs are identical to that of $\mathbf{O}$.

Definition 22 (Bad Collision) *For a random function* $\mathbf{G} : \mathbf{W}^{n-q-k} \to \mathbf{W}^{c}$ *and messages* $M, M' \in \{0,1\}^*$, *let* $BC_{\mathcal{M}_L^{\mathbf{G}}}(M, M')$ *denote the event that*

$$M \neq M', \quad \textit{and} \quad \mathcal{H}_L^{\mathbf{G}}(M) = \mathcal{H}_L^{\mathbf{G}}(M').$$

Note that a bad collision necessarily implies a collision in $\mathcal{M}^{\mathbf{F}}$.

A bad collision on M and M' means that not only does $\mathcal{H}_L^{\mathbf{G}}(M) = \mathcal{H}_L^{\mathbf{G}}(M')$, but $height_L(|M|) = height_L(|M'|)$ as well. This is because one word of the input to each compression function is devoted to U, the representation of the level ℓ and index i of the compression function in the hash tree. In particular, for the final compression function, the height of the hash tree is encoded into U, since $\ell+i-2$ is the height of the tree. Therefore if $height_L(|M|) \neq height_L(|M'|)$ then it is impossible for $\mathcal{H}_L^{\mathbf{G}}(M) = \mathcal{H}_L^{\mathbf{G}}(M')$, because this height information is encoded into the output of $\mathcal{H}_L^{\mathbf{G}}$.

Lemma 9 *Let $\mathcal{A} = \{A_i\}$ be the monotone condition on $\mathcal{M}_L^{\mathbf{R}}$ such that A_i is the event that there are no bad collisions in the first i queries $(M_1, M_2, \ldots, M_i)$:*

$$A_i = \bigwedge_{1 \le j \le j' \le i} \overline{BC_{\mathcal{M}_L^{\mathbf{R}}}(M_j, M_{j'})}.$$

Then $\mathcal{M}_L^{\mathbf{R}} \,\big|\, \mathcal{A} \equiv \mathbf{O}$.

Proof: Recall that $\mathcal{M}_L^{\mathbf{R}} = \chi_d \circ \mathbf{R} \circ \mathcal{H}_L^{\mathbf{R}}$, and further, the event $\overline{\mathrm{BC}_{\mathcal{M}_L^{\mathbf{R}}}(M, M')}$ implies that $\mathcal{H}_L^{\mathbf{R}}(M) \neq \mathcal{H}_L^{\mathbf{R}}(M')$. So, effectively, we are asking about the output distribution of $(\chi_d \circ \mathbf{R}) : \mathbf{W}^{n-q-k} \to \{0,1\}^d$, conditioned on having distinct inputs, as compared to the output distribution of $\mathbf{O} : \{0,1\}^* \to \{0,1\}^d$, also conditioned on having distinct inputs. Since $\chi_d \circ \mathbf{R}$ is just the random function with uniform distribution over all functions mapping $\mathbf{W}^{n-q-k} \to \{0,1\}^d$, these distributions are identical (namely, the uniform distribution over $\{0,1\}^d$).

We can now apply the above lemma to bound the probability of distinguishing between $\mathcal{M}_L^{\mathbf{R}}$ and $\mathbf{O}$.

Lemma 10 *If $\mathcal{M}_L^{\mathbf{R}} \,\big|\, \mathcal{A} \equiv \mathbf{O}$, then*

$$\Delta_q(\mathcal{M}_L^{\mathbf{R}}, \mathbf{O}) \le \frac{q(q-1)}{2} \cdot \frac{height_L\left(\frac{\mu}{q}\right)}{2^{cw}}.$$

Proof: By Lemma 6, since $\mathcal{M}^{\mathbf{R}} \,\big|\, \mathcal{A} \equiv \mathbf{O}$, any adversary's advantage for distinguishing between $\mathcal{M}^{\mathbf{R}}$ and $\mathbf{O}$ with q queries is bounded by the probability of success by the best adaptive algorithm at provoking $\overline{A_q}$ (that is, finding a bad collision).

$$\Delta_q(\mathcal{M}^{\mathbf{R}}, \mathbf{O}) \le \nu_q(\mathcal{M}^{\mathbf{R}}, \overline{A_k}).$$

In addition, by Lemma 7, the probability of success for the best adaptive strategy is no better than that for the best non-adaptive strategy (and in fact, they are equal). Therefore,

$$\nu_q(\mathcal{M}^{\mathbf{R}}, \overline{A_k}) = \mu_q(\mathcal{M}^{\mathbf{R}}, \overline{A_k}).$$

Thus, we may simply consider the probability of provoking a bad collision non-adaptively, which is the great benefit of using Maurer's framework. To do this, we will first compute the probability that two arbitrary distinct messages M and M' will have a bad collision, and then apply the birthday bound.

Recall that a bad collision occurs when we have two messages $M \neq M'$ where $height_L(M) = height_L(M')$ and $\mathcal{H}^{\mathbf{R}}(M) = \mathcal{H}^{\mathbf{R}}(M')$. Therefore we can first restrict ourselves to considering only messages M, M' which have equal height h.

Now, with this assumption, we wish to upper bound $\Pr[\mathrm{BC}_{\mathcal{M}^{\mathbf{R}}}(M, M')]$. Our goal will be to prove by induction on the height h that

$$\Pr[\mathrm{BC}_{\mathcal{M}^{\mathbf{R}}}(M, M')] \le \frac{h}{2^{cw}}.$$

The base case here is straightforward. It is impossible for two trees of height 1 to have a bad collision. This is because height 1 trees have only one compression function, namely the final one. By definition, a bad collision is a collision that occurs before the final compression function, which simultaneously requires that $M \neq M'$ and $M = M'$. Thus we have a contradiction.

For the inductive step, note that the output of $\mathcal{H}^{\mathbf{R}}(M)$ can be described as follows. First, there exists a partition of M into $M_1 \| M_2 \| M_3 \| M_4$ of four (some possibly empty) bit strings such that

$$\mathcal{H}^{\mathbf{R}}(M) = U \| V \| H_1^{\mathbf{R}}(M_1) \| H_2^{\mathbf{R}}(M_2) \| H_3^{\mathbf{R}}(M_3) \| H_4^{\mathbf{R}}(M_4).$$

Here the $H_i^{\mathbf{R}}$ are random functions mapping $\{0,1\}^* \to \mathbf{W}^c$ and U and V are the auxiliary control information inserted by the MD6 mode of operation. In particular, when viewed as a hash tree, the height of each $H_i^{\mathbf{R}}(M_i)$ is at most $h-1$. In addition, if we also partition M' in this fashion, we notice that since $M \neq M'$ there exists a j such that $M_j \neq M'_j$. Therefore

$$\begin{aligned}
\Pr[\mathrm{BC}_{\mathcal{M}^{\mathbf{R}}}(M, M')] &= \Pr\left[\mathcal{H}^{\mathbf{R}}(M) = \mathcal{H}^{\mathbf{R}}(M')\right] \\
&\leq \Pr\left[\bigwedge_{i=1}^{4} \left(H_i^{\mathbf{R}}(M_i) = H_i^{\mathbf{R}}(M'_i)\right)\right] \\
&= \prod_{i=1}^{4} \Pr\left[H_i^{\mathbf{R}}(M_i) = H_i^{\mathbf{R}}(M'_i)\right] \\
&\leq \Pr\left[H_j^{\mathbf{R}}(M_j) = H_j^{\mathbf{R}}(M'_j)\right].
\end{aligned}$$

Thus, we've upper bounded the probability of a bad collision occurring by the probability of there being a collision for one of the $H_i^{\mathbf{R}}$. Let E denote the event that this collision, $H_j^{\mathbf{R}}(M_j) = H_j^{\mathbf{R}}(M'_j)$, occurs. Then

$$\begin{aligned}
\Pr[E] &= \Pr\left[E \;\middle|\; \mathrm{BC}_{H_j^{\mathbf{R}}}(M_j, M'_j)\right] \cdot \Pr\left[\mathrm{BC}_{H_j^{\mathbf{R}}}(M_j, M'_j)\right] \\
&\quad + \Pr\left[E \;\middle|\; \overline{\mathrm{BC}_{H_j^{\mathbf{R}}}(M_j, M'_j)}\right] \cdot \Pr\left[\overline{\mathrm{BC}_{H_j^{\mathbf{R}}}(M_j, M'_j)}\right] && (7.2) \\
&\leq 1 \cdot \Pr\left[\mathrm{BC}_{H_j^{\mathbf{R}}}(M_j, M'_j)\right] + \frac{1}{2^{c \cdot w}} \cdot \Pr\left[\overline{\mathrm{BC}_{H_j^{\mathbf{R}}}(M_j, M'_j)}\right] && (7.3) \\
&\leq \Pr\left[\mathrm{BC}_{H_j^{\mathbf{R}}}(M_j, M'_j)\right] + \frac{1}{2^{cw}} \\
&\leq \frac{h-1}{2^{cw}} + \frac{1}{2^{cw}} && (7.4) \\
&= \frac{h}{2^{cw}} && (7.5)
\end{aligned}$$

Equation (7.2) follows by the law of total probability. In the derivation of Inequality (7.3), we observe that the conditional probability $\Pr\left[E \;\middle|\; \overline{\mathrm{BC}_{H_j^{\mathbf{R}}}(M_j, M'_j)}\right]$

is just the probability of collision for the random function $\mathbf{R}$ on distinct inputs, which is equivalent to $\frac{1}{2^{cw}}$. In addition, $\Pr\left[E \;\middle|\; \mathrm{BC}_{H_j^{\mathbf{R}}}(M_j, M_j')\right]$ is the probability of collision for the random function $\mathbf{R}$ on identical inputs, which is 1. For Inequality (7.4), we can use our inductive hypothesis, since as we noted earlier $H_j^{\mathbf{R}}(M_j)$ has height at most $h-1$, thereby arriving at Inequality (7.5).

Since we've shown that for any two messages M and M' the probability of a bad collision is bounded by $\frac{h}{2^{cw}}$, it remains to bound the total probability of causing a bad collision given q non-adaptive queries of total length at most μ. Since an adversary cannot cause a bad collision by querying two messages with different heights, the best strategy for the adversary is to produce q queries of all equal height $height_L\left(\frac{\mu}{q}\right)$, and by the birthday bound

$$\Delta_q\big(\mathcal{M}^{\mathbf{R}}, \mathbf{O}\big) \leq \frac{q(q-1)}{2} \cdot \frac{height_L\left(\frac{\mu}{q}\right)}{2^{cw}}.$$

Now that we have bounded the probability of success for any distinguisher given resources q and μ, we show that MD6 acts as a domain extender for FIL-QRFs, and later we will show that it does the same for FIL-PRFs.

Theorem 5 *Fix the resource constraints q, and μ. If $\mathbf{F}$ is a $(\delta(q,\mu), \varepsilon)$-secure FIL-QRF, then $\mathcal{M}_L^{\mathbf{F}}$ is a $(q, \mu, \varepsilon + \beta(q,\mu))$-secure VIL-QRF, where*

$$\beta(q,\mu) = \frac{q(q-1)}{2} \cdot \frac{height_L\left(\frac{\mu}{q}\right)}{2^{cw}}.$$

Proof: By Lemma 5 (the triangle inequality), the advantage of any adversary given q queries and total message bit-length μ in distinguishing $\mathcal{M}^{\mathbf{F}}$ from $\mathbf{O}$ is

$$\Delta_{q,\mu}\big(\mathcal{M}_L^{\mathbf{F}}, \mathbf{O}\big) \leq \Delta_{q,\mu}\big(\mathcal{M}_L^{\mathbf{F}}, \mathcal{M}_L^{\mathbf{R}}\big) + \Delta_{q,\mu}\big(\mathcal{M}_L^{\mathbf{R}}, \mathbf{O}\big).$$

By Lemma 8, the advantage of any adversary given q queries and μ bits in distinguishing $\mathcal{M}_L^{\mathbf{F}}$ and $\mathcal{M}_L^{\mathbf{R}}$ is

$$\Delta_{q,\mu}\big(\mathcal{M}_L^{\mathbf{F}}, \mathcal{M}_L^{\mathbf{R}}\big) \leq \Delta_{\delta(q,\mu)}(\mathbf{F}, \mathbf{R}) \leq \varepsilon.$$

Therefore it remains only to bound $\Delta_q\big(\mathcal{M}^{\mathbf{R}}, \mathbf{O}\big)$. By applying Lemmas 9 and 10, we can bound this by

$$\Delta_{q,\mu}\big(\mathcal{M}_L^{\mathbf{R}}, \mathbf{O}\big) \leq \frac{q(q-1)}{2} \cdot \frac{height_L\left(\frac{\mu}{q}\right)}{2^{cw}}.$$

Finally, combining this with our previous results yields

$$\Delta_{q,\mu}\big(\mathcal{M}_L^{\mathbf{F}}, \mathbf{O}\big) \leq \varepsilon + \frac{q(q-1)}{2} \cdot \frac{height_L\left(\frac{\mu}{q}\right)}{2^{cw}}.$$

Corollary 3 *If* $\mathbf{F}$ *is a* $(t, \delta(q,\mu), \varepsilon)$*-secure FIL-PRF, then* $\mathcal{M}_L^{\mathbf{F}}$ *is a* $(t', q, \mu, \varepsilon + \beta(q,\mu))$*-secure VIL-PRF, where* $t' = t - O(\delta(q,\mu))$.

Proof: The proof proceeds almost identically to Theorem 5.

$$\begin{aligned}
\Delta_{t',q,\mu}\big(\mathcal{M}_L^{\mathbf{F}}, \mathbf{O}\big) &\leq \Delta_{t',q,\mu}\big(\mathcal{M}_L^{\mathbf{F}}, \mathcal{M}_L^{\mathbf{R}}\big) + \Delta_{t',q,\mu}\big(\mathcal{M}_L^{\mathbf{R}}, \mathbf{O}\big) \\
&\leq \Delta_{t',q,\mu}\big(\mathcal{M}_L^{\mathbf{F}}, \mathcal{M}_L^{\mathbf{R}}\big) + \beta(q,\mu) \\
&\leq \Delta_{t,\delta(q,\mu)}(\mathbf{F}, \mathbf{R}) + \beta(q,\mu) \qquad (7.6) \\
&\leq \varepsilon + \beta(q,\mu)
\end{aligned}$$

Inequality (7.6) follows as a corollary to Lemma 8, since simulating the query responses to the adversary takes time $O(\delta(q,\mu))$.

One important note is that for large values of L, $height_L(x)$ grows logarithmically in x. However, for the iterative mode of operation with $L = 0$, $height_L(x)$ grows linearly in x. Holding q fixed, this is asymptotically the same as the bound shown by Bellare et al. for the Cipher Block Chaining mode of operation [9], which was approximately $O\left(\frac{\ell q^2}{2^{cw}}\right)$, where ℓ is the block length of the longest query. Therefore the MD6 mode of operation for large L represents an asymptotic improvement (logarithmic in $height_L(\mu/q)$ as opposed to linear) over the Cipher Block Chaining mode of operation.

7.6 Unpredictability

That a family of functions has the property of pseudorandomness is a very strong assumption to make. A weaker assumption is that the family of functions is merely unpredictable, which allows it to function as a MAC (recall that pseudorandomness implies unpredictability, and thus PRFs are also MACs [42, 41].) However, if one simply wants to prove that some hash function H^{f_Q} is a secure MAC, it seems unnecessary to derive this property by assuming that the underlying compression f_Q is pseudorandom. One might wonder whether it is instead possible to prove that H^{f_Q} is a MAC if f_Q is a MAC, analogously to the proof in Section 7.5.2. If so, it might alleviate some concerns about the use of a hash function H^{f_Q} as a MAC, since a successful pseudorandomness distinguisher for f_Q would not necessarily invalidate the MAC capabilities of H^{f_Q} (unless f_Q is also shown to be predictable as well).

The above relation does not necesserily hold when H is the Cipher Block Chaining (CBC) mode of operation, H_{CBC}. Whereas one can prove concretely that CBC-MAC is pseudorandom if its underlying block cipher is pseudorandom [8, 9], An and Bellare constructed a simple FIL-MAC f such that H^f_{CBC} does not share the property of unpredictability [3]. However, they also demonstrated that a two-keyed variant of the Merkle-Damgård construction is a VIL-MAC if its compression function is a FIL-MAC. Thus it is interesting for us to consider if MD6 also is a domain extender that preserves the property of unpredictability.

7.6.1 Preliminaries

We begin with some definitions that will be essential to our later proofs.

Definition 23 (FIL-MAC) *For a keyed FIL function $f_Q : \mathbf{W}^k \times \mathbf{W}^{n-q-k} \to \mathbf{W}^c$, define the advantage of an adversary A for forging a MAC as*

$$\boldsymbol{Adv}^{\textit{fil-mac}}_{\mathsf{A}} = \Pr\left[\begin{array}{c|c} K \xleftarrow{\$} \mathbf{W}^k; & f_Q(K,m) = D, \\ (m,D) \leftarrow \mathsf{A}^{f_Q(K,\cdot)} & m \textit{ was not a query to } f_Q(K,\cdot) \end{array}\right]$$

We define the insecurity of the FIL-MAC f_Q to be

$$\boldsymbol{InSec}^{\textit{fil-mac}}_{f}(t,q) = \max_{\mathsf{A}}\left\{\boldsymbol{Adv}^{\textit{fil-mac}}_{\mathsf{A}}\right\},$$

where the maximum is taken over all adversaries A with running time t and number of $f_Q(K,\cdot)$ oracle queries q.

Definition 24 (VIL-MAC) *For a keyed VIL function $H : \mathbf{W}^k \times \{0,1\}^* \to \{0,1\}^d$, define the advantage of an adversary A for forging a MAC as*

$$\boldsymbol{Adv}^{\textit{vil-mac}}_{\mathsf{A}} = \Pr\left[\begin{array}{c|c} K \xleftarrow{\$} \mathbf{W}^k; & H(K,M) = D, \\ (M,D) \leftarrow \mathsf{A}^{H(K,\cdot)} & M \textit{ was not a query to } H(K,\cdot) \end{array}\right]$$

We define the insecurity of the VIL-MAC H to be

$$\boldsymbol{InSec}^{\textit{vil-mac}}_{H}(t,q,\mu) = \max_{\mathsf{A}}\left\{\boldsymbol{Adv}^{\textit{vil-mac}}_{\mathsf{A}}\right\},$$

where the maximum is taken over all adversaries A with running time t and number of $H(K,\cdot)$ oracle queries q and total query bit-length μ.

To proceed, we must define the notion of weak collision resistance. As the name implies, this is a weaker notion of the collision resistance defined in Section 7.2; rather than giving the collision-finding algorithm the key K, we instead give it only oracle access to the keyed function.

Definition 25 (FIL-WCR) *For a keyed FIL function $f_Q : \mathbf{W}^k \times \mathbf{W}^{n-q-k} \to \mathbf{W}^c$, define the advantage of an adversary A for FIL-weak collision resistance as*

$$\boldsymbol{Adv}^{\textit{fil-wcr}}_{\mathsf{A}} = \Pr\left[\begin{array}{c|c} K \xleftarrow{\$} \mathbf{W}^k; & m \neq m' \\ (m,m') \leftarrow \mathsf{A}^{f_Q(K,\cdot)} & f_Q(K,m) = f_Q(K,m'), \end{array}\right]$$

We define the insecurity of the FIL-WCR f_Q to be

$$\boldsymbol{InSec}^{\textit{fil-wcr}}_{f}(t,q) = \max_{\mathsf{A}}\left\{\boldsymbol{Adv}^{\textit{fil-wcr}}_{\mathsf{A}}\right\},$$

where the maximum is taken over all adversaries A with running time t and number of $f_Q(K,\cdot)$ oracle queries q.

Definition 26 (VIL-WCR) *For a keyed VIL function $H : \mathbf{W}^k \times \{0,1\}^* \to \mathbf{W}^c$, define the advantage of an adversary A for VIL-weak collision resistance as*

$$\boldsymbol{Adv}^{\textit{vil-wcr}}_{\mathsf{A}} = \Pr\left[\begin{array}{c|c} K \xleftarrow{\$} \mathbf{W}^k; & M \neq M', \\ (M, M') \leftarrow \mathsf{A}^{H(K,\cdot)} & H(K,M) = H(K,M') \end{array}\right]$$

We define the insecurity of the VIL-WCR H to be

$$\boldsymbol{InSec}^{\textit{vil-wcr}}_{H}(t,q,\mu) = \max_{\mathsf{A}}\left\{\boldsymbol{Adv}^{\textit{vil-wcr}}_{\mathsf{A}}\right\},$$

where the maximum is taken over all adversaries A with running time t and number of $H(K,\cdot)$ oracle queries q and total query bit-length μ.

We also make use of some lemmas from An and Bellare. We restate them here, without rigorous proofs, which can be found in [3].

7.6.1.1 Important Lemmas

The two-keyed variant of Merkle-Damgård shown in An and Bellare is defined as follows. Given a keyed compression function $g : \{0,1\}^k \times \{0,1\}^{\ell+b} \to \{0,1\}^\ell$ and two keys K_1 and K_2, use the strengthened Merkle-Damgård construction where all compression functions except for the final one use key K_1. The final compression function then uses K_2. In their proof that this is a domain extender for FIL-MACs, An and Bellare used several steps.

1. Show that a FIL-MAC g is also FIL-weak collision resistant.
2. Prove that the Merkle-Damgård construction (without the last compression function) is a domain extender for FIL-weak collision resistance. That is, if the compression function g used is FIL-WCR, then the overall hash function h (without the last compression function) is VIL-WCR.
3. Demonstrate that composing a FIL-MAC g with a VIL-WCR function h (with independent keys) is a VIL-MAC. Therefore by points 1 and 2, we have a VIL-MAC from a FIL-MAC.

We begin with a formal statement of the last point.

Lemma 11 (Lemma 4.2 [3]) *Let $g : \mathbf{W}^k \times \mathbf{W}^{n-q-k} \to \{0,1\}^d$ be a FIL-MAC and let $h : \mathbf{W}^k \times \{0,1\}^* \to \mathbf{W}^{n-q-k}$ be a VIL-WCR function. Define $H : \mathbf{W}^{2k} \times \{0,1\}^* \to \{0,1\}^d$ as*

$$H(K_1, K_2, M) = f_Q(K_2, h(K_1, M))$$

for keys $K_1, K_2 \in \mathbf{W}^k$ and $M \in \{0,1\}^*$. Then H is a VIL-MAC with

$$\boldsymbol{InSec}^{\textit{vil-mac}}_{H}(t,q,\mu) \leq \boldsymbol{InSec}^{\textit{fil-mac}}_{g}(t,q) + \boldsymbol{InSec}^{\textit{vil-wcr}}_{h}(t,q,\mu).$$

Proof: See [3, Appendix A.1].

In addition, we will also make use of the following lemma, which states formally point 1 from above.

Lemma 12 (Lemma 4.4 [3]) *Let $f_Q : \mathbf{W}^k \times \mathbf{W}^{n-q-k} \rightarrow \mathbf{W}^c$ be a FIL-MAC family of functions. Then it is also a FIL-WCR family with*

$$\boldsymbol{InSec}_f^{\textit{fil-wcr}}(t,q) \leq \frac{q(q-1)}{2} \cdot \boldsymbol{InSec}_f^{\textit{fil-mac}}(t + O(q), q).$$

Proof: See [3, Appendix A.3].

Since the MD6 mode of operation is different from the two-keyed Merkle-Damgård construction of An and Bellare, we omit the formal statement for point 2. Instead, we will prove our own version in Lemma 13 for the MD6 mode of operation.

7.6.1.2 A Two-Keyed Variant of MD6

Note that in Lemma 11 the hash function H has a keyspace with twice as many bits as the underlying functions g and h. Therefore, a straight adaptation of the techniques of An and Bellare [3] for MD6 does not immediately follow, as MD6 has only a single key. Thus, we demonstrate here that a two-keyed variant of MD6, which we will refer to as $\mathcal{M}\langle 2\rangle$,is a domain extender for FIL-MACs, in much the same fashion as An and Bellare's approach. That is, we show that the function

$$\mathcal{M}\langle 2\rangle^{f_Q}(K_1, K_2, M) = \chi_d(f_Q(K_2, \mathcal{H}^{f_Q}(K_1, M)))$$

where $\mathcal{M}\langle 2\rangle^{f_Q} : \mathbf{W}^{2k} \times \{0,1\}^* \rightarrow \{0,1\}^d$ is a VIL-MAC if f_Q and $\chi_d \circ f_Q$ are FIL-MACs.

We begin by proving that if the compression function f_Q is a FIL-WCR, then $\mathcal{H}^{f_Q}$ is a VIL-WCR function. This is analogous to [3, Lemma 4.3], where An and Bellare prove that the Merkle-Damgård construction also acts as a domain extender for FIL-WCRs. The proof of this lemma is very similar to the proof of Theorem 3, as weak collision resistance is a weaker notion of the standard collision resistance. However, due to the additional query resource constraints in the weak collision resistance definition, we must be precise in our adaptation.

Lemma 13 *Let $f_Q : \mathbf{W}^k \times \mathbf{W}^{n-q-k} \rightarrow \mathbf{W}^c$ be a FIL-WCR family of functions. Then $\mathcal{H}^{f_Q} : \mathbf{W}^k \times \{0,1\}^* \rightarrow \mathbf{W}^{n-q-k}$ is a VIL-WCR family of functions with*

$$\boldsymbol{InSec}_{\mathcal{H}^{f_Q}}^{\textit{vil-wcr}}(t,q,\mu) \leq \boldsymbol{InSec}_f^{\textit{fil-wcr}}(2t, \delta(q,\mu)).$$

Proof: We proceed in a manner that is very similar to the proof for Theorem 3. Let A be an algorithm with the best possible success for breaking the VIL-weak collision resistance of $\mathcal{H}^{f_Q}$, given resources t, q and μ. As before, we construct an algorithm C that uses A as a subroutine to attack the FIL-weak collision resistance of f_Q.

Recall that in the weak collision resistance setting, instead of being given the key K as input, we are instead only given oracle access to the function keyed by K. Therefore $\mathsf{C}^{f_Q(K,\cdot)}$ is given time $2t$, q queries, and μ total bits queried, and must produce messages $m \neq m'$ such that $f_Q(K,m) = f_Q(K,m')$.

Algorithm $\mathsf{C}^{f_Q(K,\cdot)}$
Input: A, the $\mathcal{H}^{f_Q}$ collision-finding algorithm
Output: $m \neq m'$, such that $f_Q(K,m) = f_Q(K,m')$
1. **for** $i \leq 1$ **to** q
2. **do** $\mathsf{A} \rightarrow M_i$,
3. $\mathsf{A} \leftarrow \mathcal{H}^{f_Q(K,\cdot)}(M_i)$
4. $\mathsf{A} \rightarrow (M, M')$
5. $P \leftarrow \mathcal{H}^{f_Q(K,\cdot)}(M)$ and store each input to $f_Q(K,\cdot)$
6. $P' \leftarrow \mathcal{H}^{f_Q(K,\cdot)}(M')$ and store each input to $f_Q(K,\cdot)$
7. **if** $M \neq M'$ and $P = P'$
8. **then** Use Theorem 2 to find two messages $m \neq m'$ s.t. $f_Q(K,m) = f_Q(K,m')$
9. **return** (m, m')
10. **else** **return** $(0^n, 0^n)$

Note that although we are using $\mathcal{H}^{f_Q}$ instead of $\mathcal{M}^{f_Q}$, Theorem 2 still applies. This is because $M \neq M'$ and $\mathcal{M}^{f_Q}(K,M) = \mathcal{M}^{f_Q}(K,M')$, and thus a collision in f_Q exists. However, it does not occur during the last compression function f_Q, since $P = P'$. Therefore it must be "contained in" $\mathcal{H}^{f_Q}$.

The advantage for C for breaking the weak collision resistance of f_Q is at least the advantage of A for breaking the weak collision resistance of $\mathcal{H}^{f_Q}$, because C succeeds precisely when A succeeds. As in Theorem 3, the amount of required is at most $2t$ (the time it takes to run A plus the time it takes to perform the hashes). In addition, by Lemma 2, the total number of oracle queries to $f_Q(K,\cdot)$ that we need to make is bounded by $\delta(q,\mu)$.

We can now prove that the two-keyed variant of MD6 $\mathcal{M}\langle 2\rangle^{f_Q}$ is a VIL-MAC, assuming that f_Q is a FIL-MAC and also that $g = \chi_d \circ f$ is a FIL-MAC. Note that the fact that g is a FIL-MAC does not follow automatically from the fact that f_Q is a FIL-MAC, so we will treat them as functions with separate levels of security.

Theorem 6 *Let $f_Q : \mathbf{W}^k \times \mathbf{W}^{n-q-k} \rightarrow \mathbf{W}^c$ be a FIL-MAC. Define the final compression function $g = (\chi_d \circ f) : \mathbf{W}^k \times \mathbf{W}^{n-q-k} \rightarrow \{0,1\}^d$ and suppose that it is also a FIL-MAC. Then the two-keyed variant of MD6 $\mathcal{M}\langle 2\rangle^{f_Q} : \mathbf{W}^{2k} \times \{0,1\}^* \rightarrow \{0,1\}^d$ is a VIL-MAC with*

$$\boldsymbol{InSec}^{vil\text{-}mac}_{\mathcal{M}\langle 2\rangle^{f_Q}}(t,q,\mu) \leq \boldsymbol{InSec}^{fil\text{-}mac}_{g}(t,q) + \frac{\delta(q,\mu)^2}{2} \cdot \boldsymbol{InSec}^{fil\text{-}mac}_{f}(2t + O(\delta(q,\mu)), \delta(q,\mu)) .$$

Proof: As in [3, Theorem 4.1], the proof of this theorem follows from Lemmas 11, 12 and 13.

$$\begin{aligned}\mathbf{InSec}^{\text{vil-mac}}_{\mathcal{M}\langle 2\rangle^{f_Q}}(t,q,\mu) &\leq \mathbf{InSec}^{\text{fil-mac}}_{g}(t,q) + \mathbf{InSec}^{\text{vil-wcr}}_{\mathcal{H}^{f_Q}}(t,q,\mu) \\ &\leq \mathbf{InSec}^{\text{fil-mac}}_{g}(t,q) + \mathbf{InSec}^{\text{fil-wcr}}_{f}(2t,\delta(q,\mu)) \\ &\leq \mathbf{InSec}^{\text{fil-mac}}_{g}(t,q) + \frac{\delta(q,\mu)^2}{2}\cdot\mathbf{InSec}^{\text{fil-mac}}_{f}(2t+O(\delta(q,\mu)),\delta(q,\mu))\end{aligned}$$

7.6.2 MD6 as a Domain Extender for FIL-MACs

It seems somewhat artificial to double the number of bits in the keyspace of MD6 simply to prove that it acts as a domain extender for FIL-MACs. Ideally, we would like to prove that it has this property as is, while still only making the assumption that the underlying compression functions f_Q and g are FIL-MACs. Unfortunately, at the moment we do not know how to prove this directly (although we have no counterexample for this property). Thus, in order to prove this statement for the standard version of MD6, we will make use of they key-blinding assumption defined in Section 6.8.

Recall that the key-blinding assumption states that the flag bit z effectively "blinds" the key values. That is, given oracle access to two functions:

$$\begin{aligned}f^0_Q(\cdot) &= f_Q(z=0,K=\kappa_0,\cdot)\\ f^1_Q(\cdot) &= f_Q(z=1,K=\kappa_1,\cdot)\end{aligned}$$

then we should not be able to guess whether $\kappa_0=\kappa_1$ or not with any significant advantage. Recalling Definition 2:

Definition 27 (Key-Blinding Assumption) *Let D be a distinguisher given oracle access to two functions f^0_Q and f^1_Q that map $\mathbf{W}^k\times\mathbf{W}^{n-q-k-1}\rightarrow\mathbf{W}^c$. We define its advantage as follows.*

$$\boldsymbol{Adv}^{blind}_{D,f_Q}(t,q) = \left|\Pr\left[\begin{array}{c}\kappa_0\xleftarrow{\$}\mathbf{W}^k;\kappa_1\xleftarrow{\$}\mathbf{W}^k;b\xleftarrow{\$}\{0,1\};\\ a\leftarrow D^{f_Q(z=0,\kappa_0,\cdot),f_Q(z=1,\kappa_b,\cdot)}\end{array} : \quad a=b\right]-\frac{1}{2}\right|$$

The goal of D is to try to determine the value of b, i.e. whether f^0_Q and f^1_Q use the same key or not. We can define the overall insecurity of the key-blinding property of f_Q to be

$$\boldsymbol{InSec}^{blind}_{f_Q}(t,q)=\max_D\left\{\boldsymbol{Adv}^{blind}_{D,f_Q}(t,q)\right\},$$

where D is given resource constraints of (t,q).

With this assumption, we now prove that the single-keyed version of MD6 acts as a domain extender for FIL-MACs (with the key-blinding assumption). The essential concept here is that by the key-blinding property of f_Q, $\mathcal{M}^{f_Q}$ behaves almost exactly like $\mathcal{M}\langle 2\rangle^{f_Q}$ to any algorithm given only oracle access to $\mathcal{M}^{f_Q}$.

Lemma 14 *Let* $f_Q : \mathbf{W}^k \times \mathbf{W}^{n-q-k} \rightarrow \mathbf{W}^c$. *Then*

$$\boldsymbol{InSec}^{vil\text{-}mac}_{\mathcal{M}^{f_Q}}(t,q,\mu) \leq \boldsymbol{InSec}^{vil\text{-}mac}_{\mathcal{M}\langle 2\rangle^{f_Q}}(t,q,\mu) + 2 \cdot \boldsymbol{InSec}^{blind}_{f}(t,\delta(q,\mu))\,.$$

Proof: Let A be a forger with the best possible success for attacking the single-keyed version of MD6, $\mathcal{M}^{f_Q}$. Therefore, given resources t, q, and μ, the probability of A succeeding is $\mathbf{InSec}^{\text{vil-mac}}_{\mathcal{M}^{f_Q}}(t,q,\mu)$. We use A to construct a distinguisher D that attacks the key-blinding property of f_Q.

Recall that the distinguisher D is given oracle access to two functions $f^0_Q(\cdot) = f_Q(z = 0, K = \kappa_0, \cdot)$ and $f^1_Q(\cdot) = f_Q(z = 1, K = \kappa_1, \cdot)$, which map $\mathbf{W}^k \times \mathbf{W}^{n-q-k-1} \rightarrow \mathbf{W}^c$, and must determine whether $\kappa_0 = \kappa_1$.

Algorithm $\mathsf{D}^{f^0_Q, f^1_Q}$
Output: 1 if $\kappa_0 \neq \kappa_1$, 0 otherwise
1. **for** $i = 1$ **to** q
2. **do** $\mathsf{A} \rightarrow M_i$
3. $\mathsf{A} \leftarrow \chi_d(f^1_Q(\mathcal{H}^{f^0_Q}(M_i)))$
4. $\mathsf{A} \rightarrow (M, D)$
5. **if** $\chi_d(f^1_Q(\mathcal{H}^{f^0_Q}(M))) = D$ and for all i, $M \neq M_i$
6. **then return** 0
7. **else return** 1

The distinguisher uses A in the following manner: it constructs the function $\chi_d \circ f^1_Q \circ \mathcal{H}^{f^0_Q}$ using its two oracles and responds to hash queries by A with this function. There are two cases:

1. If $b = 1$, then $\kappa_0 \neq \kappa_1$. Therefore the hash function that A is querying is
$$\chi_d \circ f^1_Q \circ \mathcal{H}^{f^0_Q} = \mathcal{M}\langle 2\rangle^{f_Q}.$$
2. If $b = 0$, then $\kappa_0 = \kappa_1$. Therefore the hash function that A is querying is
$$\chi_d \circ f^1_Q \circ \mathcal{H}^{f^0_Q} = \mathcal{M}^{f_Q}.$$

Recall that A is an optimal forger for $\mathcal{M}^{f_Q}$ (although it may also be a forger for $\mathcal{M}\langle 2\rangle^{f_Q}$ as well), and outputs a forgery (M, D). D outputs 0 if this is a valid forgery (i.e. if A succeeds, then we suspect that $\kappa_0 = \kappa_1$). If the pair is not a valid forgery, we output 1, as we believe that the reason for this failure is that $\kappa_0 \neq \kappa_1$. We can analyze the success probability of D as follows.

$$
\begin{aligned}
\Pr[\mathsf{D}\text{ succeeds}] &= \Pr[b=1]\cdot\Pr\Big[\mathsf{D}^{f_Q(z=0,K=\kappa_0,\cdot),\,f_Q(z=1,K=\kappa_b,\cdot)}\to 1 \mid b=1\Big] \\
&\quad + \Pr[b=0]\cdot\Pr\Big[\mathsf{D}^{f_Q(z=0,K=\kappa_0,\cdot),\,f_Q(z=1,K=\kappa_b,\cdot)}\to 0 \mid b=0\Big] \\
&= \frac{1}{2}\cdot\Pr\big[\mathsf{A}\text{ fails to forge }\mathcal{M}\langle 2\rangle^{f_Q}\big] \\
&\quad + \frac{1}{2}\cdot\Pr\big[\mathsf{A}\text{ successfully forges }\mathcal{M}^{f_Q}\big] \\
&\geq \frac{1}{2}\cdot\Big(1-\mathbf{InSec}^{\text{vil-mac}}_{\mathcal{M}\langle 2\rangle^{f_Q}}(t,q,\mu)\Big)+\frac{1}{2}\cdot\mathbf{InSec}^{\text{vil-mac}}_{\mathcal{M}^{f_Q}}(t,q,\mu) \qquad (7.7)\\
&= \frac{1}{2}+\frac{\mathbf{InSec}^{\text{vil-mac}}_{\mathcal{M}^{f_Q}}(t,q,\mu)-\mathbf{InSec}^{\text{vil-mac}}_{\mathcal{M}\langle 2\rangle^{f_Q}}(t,q,\mu)}{2}
\end{aligned}
$$

For Inequality (7.7), note that A could potentially succeed as a forger for $\mathcal{M}\langle 2\rangle^{f_Q}$. However, its advantage is bounded, as $\mathbf{Adv}^{\text{vil-mac}}_{\mathsf{A},\mathcal{M}\langle 2\rangle^{f_Q}}(t,q,\mu)\leq\mathbf{InSec}^{\text{vil-mac}}_{\mathcal{M}\langle 2\rangle^{f_Q}}(t,q,\mu)$, whereby we derive the inequality. Therefore,

$$
\begin{aligned}
\mathbf{Adv}^{\text{blind}}_{D,f_Q}(t,\delta(q,\mu)) &= \left|\Pr[\mathsf{D}\text{ succeeds}]-\frac{1}{2}\right| \\
&\geq \frac{\left|\mathbf{InSec}^{\text{vil-mac}}_{\mathcal{M}^{f_Q}}(t,q,\mu)-\mathbf{InSec}^{\text{vil-mac}}_{\mathcal{M}\langle 2\rangle^{f_Q}}(t,q,\mu)\right|}{2} \\
\mathbf{InSec}^{\text{blind}}_{f_Q}(t,\delta(q,\mu)) &\geq \frac{\left|\mathbf{InSec}^{\text{vil-mac}}_{\mathcal{M}^{f_Q}}(t,q,\mu)-\mathbf{InSec}^{\text{vil-mac}}_{\mathcal{M}\langle 2\rangle^{f_Q}}(t,q,\mu)\right|}{2}
\end{aligned}
$$

In particular, this implies

$$
\mathbf{InSec}^{\text{vil-mac}}_{\mathcal{M}^{f_Q}}(t,q,\mu)\leq\mathbf{InSec}^{\text{vil-mac}}_{\mathcal{M}\langle 2\rangle^{f_Q}}(t,q,\mu)+2\cdot\mathbf{InSec}^{\text{blind}}_{f_Q}(t,\delta(q,\mu))
$$

We now conclude by applying the above lemma to derive a bound on the insecurity of $\mathcal{M}^{f_Q}$.

Theorem 7 *Let $f_Q:\mathbf{W}^k\times\mathbf{W}^{n-q-k}\to\mathbf{W}^c$ be a FIL-MAC and suppose that it has the key-blinding property. Define $g=(\chi_d\circ f):\mathbf{W}^k\times\mathbf{W}^{n-q-k}\to\{0,1\}^d$ and suppose that it is also a FIL-MAC. Then $\mathcal{M}^{f_Q}:\mathbf{W}^k\times\{0,1\}^*\to\{0,1\}^d$ is a VIL-MAC with*

$$
\begin{aligned}
\boldsymbol{InSec}^{vil\text{-}mac}_{\mathcal{M}^{f_Q}}(t,q,\mu)\leq\frac{\delta(q,\mu)^2}{2}\cdot\boldsymbol{InSec}^{fil\text{-}mac}_{f_Q}(2t+O(\delta(q,\mu)),\delta(q,\mu))+ \\
\boldsymbol{InSec}^{fil\text{-}mac}_{g}(t,q)+2\cdot\boldsymbol{InSec}^{blind}_{f_Q}(t,\delta(q,\mu)).
\end{aligned}
$$

Proof: This follows directly by the application of Theorem 6 and Lemma 14.

Therefore we have shown that the MD6 mode of operation $\mathcal{M}^{f_Q}$ acts as a domain extender for FIL-MACs, assuming that both f_Q and g are FIL-MACs

and that f_Q has the key-blinding property. Unfortunately we needed to make an additional assumption about the compression function f_Q, and as such this is not a true "domain extension" result. However, at the moment we know of no other way to prove that MD6 has this property, without making additional assumptions.

7.7 Indifferentiability from Random Oracle

In this section, we present a proof that the MD6 mode of operation is indifferentiable from a random oracle assuming that the MD6 compression function is a random oracle. Let $f_Q : \mathbf{W}^{n-q} \rightarrow \mathbf{W}^c$ be the reduced compression function defined in Section 2.4, and let $\mathcal{M}^{f_Q}$ be the MD6 mode of operation applied to f_Q. Section 6.1.2 presents a proof that the reduced MD6 compression function f_Q is indifferentiable from a random oracle under the assumption that f_Q is based on a random permutation. Thus, in this section, we will assume that f_Q is a random oracle.

We use Definition 1 from Section 6.1.2 for the notion of indifferentiability, again relying on the framework of Maurer et al. [59] and the approach of Coron et al. [31], but this time to analyze the MD6 mode of operation rather than the MD6 compression function.

Theorem 8 *If $f_Q : \mathbf{W}^{n-q} \rightarrow \mathbf{W}^c$ is a random oracle, then $\mathcal{M}^{f_Q}$ is (t, q_F, q_S, ϵ)-indifferentiable from a random oracle $F : \{0,1\}^* \rightarrow \{0,1\}^d$, with $\epsilon = 2q_t^2/2^{cw}$ and $t = O(q_S^2)$, for any q_F and q_S such that the total number of compression function calls from the mode of operation queries and the compression function queries of the distinguisher is at most q_t.*

Proof: Fix a distinguisher D that makes exactly q_S compression function queries and generates exactly q_t compression function calls from its mode of operation queries and its compression function queries. We will assume that all of D's compression function queries are of length $(n-q)$ words.

Let $\gamma_a(X)$ denote a function that truncates its input X by dropping its last a bits.

We will define a simulator S with access to a random oracle $F : \{0,1\}^* \rightarrow \{0,1\}^d$ as follows.

Upon receiving a compression function query x^*, S will use the algorithm described below to decide to either answer with its own fresh random string or consult F and return an answer based on the answer of F. (However, no matter what, S will repeat the same answer if the query x^* has been seen before.) This decision is made as follows: S consults F on those queries x^* that correspond to the final compression function call in an MD6 computation for which all intermediate compression function calls have already been queried to S. We will call such compression function queries "final". For all other queries x^*, S answers with its own fresh random string.

More formally, S maintains a table T, initially empty, containing the pairs $(x, C) \in \mathbf{W}^{n-q} \times \mathbf{W}^c$ such that S has answered C in response to a query x. For

each pair (x, C), S also maintains the most recent time τ that it answered C in response to the query x. For simplicity, in the description of S we will omit the recording of these times.

Upon receiving a compression function query x^*, S executes the following algorithm.

1. Search T for a pair (x^*, C^*). If such a pair is found, return C^* as the answer to the query.
2. Parse x^* into K^*, ℓ^*, i^*, r^*, L^*, z^*, p^*, $keylen^*$, d^*, B^*.
3. If any of the following conditions is true, choose a fresh random string $C^* \in \mathbf{W}^c$ and return C^* as the answer to the query:
   (a) $z^* \neq 1$
   (b) $\ell^* > L^* + 1$
   (c) The last p^* bits of B^* are not all 0.
   (d) $0 \leq \ell^* \leq L$ and $i^* \neq 0$
   (e) $d^* \neq d$
4. The simulator now decides whether x^* is a final query by searching for a legitimate way to build a complete MD6 computation from previous compression function queries to S, with x^* as the input to the final compression function call.

   If $0 \leq \ell^* \leq L$ (only the PAR operation is used), let $B_{\ell^*,0} = B^*$, $j_{\ell^*} = 0$, and $p_{\ell^*,0} = p^*$. Search T for a sequence of pairs $(x_{\ell,i}, C_{\ell,i})$ satisfying the following condition:

   (a) For $\ell = 1, 2, \ldots, \ell^* - 1$,
   $B_{\ell+1,0}||B_{\ell+1,1}||\ldots||B_{\ell+1,j_{\ell+1}} = C_{\ell,0}||C_{\ell,1}||\ldots||C_{\ell,j_\ell}||0^{p_{\ell+1,j_{\ell+1}}}$ and
      - For $i = 0, 1, \ldots, j_\ell$,
      $x_{\ell,i} = K^*||U_{\ell,i}||r^*||L^*||0||p_{\ell,i}||keylen^*||d^*||B_{\ell,i}$, where $U_{\ell,i} = \ell \cdot 2^{56} + i$, $p_{\ell,i} = 0$ for $i = 0, 1, \ldots, j_\ell - 1$, and the last p_{ℓ,j_ℓ} bits of $B_{\ell,i}$ are 0.

   If no satisfying sequence is found, choose a fresh random string $C^* \in \mathbf{W}^c$. Insert (x^*, C^*) into T and return C^* as the answer to the query.

   If a satisfying sequence is found, let $M^* = B_{1,0}||B_{1,1}||\ldots||\gamma_{p_1,j_1}(B_{1,j_1})$. (Recall that $\gamma_a(X)$ denotes a function that truncates its input X by dropping its last a bits.)

   If multiple satisfying sequences are found, yielding different values of M^*, choose the one which comes last chronologically, in the following sense. For each satisfying sequence, sort the compression function queries by node identifier U. Choose the sequence with the most recent time τ recorded for the first compression function query ($\ell = 1, i = 0$). If there are multiple

such sequences, choose the sequence with the most recent time τ for the next compression function query ($\ell = 1, i = 1$), and so on, until a unique M^* is obtained.

Query M^* to F, and let $F(M^*) \in \{0,1\}^d$ denote F's answer. Pad $F(M^*)$ to the full length of the compression function output by choosing a random string $C' \in \{0,1\}^{cw-d}$ and letting $C^* = C'||F(M^*)$. Insert (x^*, C^*) into T and return C^* as the answer to the query.

5. If $\ell^* = L^* + 1$ (the SEQ operation is used), write B^* as $C_{\ell^*,i^*-1}||B_{\ell^*,i^*}$ where $C_{\ell^*,i^*-1} \in \mathbf{W}^c$ and $B_{\ell^*,i^*} \in \mathbf{W}^{b-c}$. Search T for a sequence of pairs $(x_{\ell^*,i}, C_{\ell^*,i})$ such that:

   (a) For $i = 0, 1, \ldots, i^*$,
   $x_{\ell^*,i} = K^*||U_{\ell^*,i}||r^*||L^*||0||0||keylen^*||d^*||C_{\ell^*,i-1}||B_{\ell^*,i}$, where $U_{\ell^*,i} = \ell^* \cdot 2^{56} + i$ and $C_{\ell^*,-1}$ is the c-word zero vector.

   If no satisfying sequence is found, choose a fresh random string $C^* \in \mathbf{W}^c$. Insert (x^*, C^*) into T and return C^* as the answer to the query.

   Otherwise, for each satisfying sequence, let $j_{\ell^*} = i^*$ and $p_{\ell^*,i^*} = p^*$, and search T for a sequence of pairs $(x_{\ell,i}, C_{\ell,i})$ satisfying the condition given above, for the PAR operation, in item 4a.

   If no satisfying sequence is found, choose a fresh random string $C^* \in \mathbf{W}^c$. Insert (x^*, C^*) into T and return C^* as the answer to the query.

   If a satisfying sequence is found, let $M^* = B_{1,0}||B_{1,1}||\ldots||\gamma_{p_1,j_1}(B_{1,j_1})$. If multiple satisfying sequences are found, yielding different values of M^*, choose the one which comes last chronologically (as described previously). Query M^* to F, and let $F(M^*) \in \{0,1\}^d$ denote F's answer. Pad $F(M^*)$ to the full length of the compression function output by choosing a random string $C' \in \{0,1\}^{cw-d}$ and letting $C^* = C'||F(M^*)$. Insert (x^*, C^*) into T and return C^* as the answer to the query.

This completes the description of the simulator S. Notice that if the simulator is queried on the final compression function call in the computation of the MD6 hash of a message after all intermediate compression function calls have already been queried to it, the simulator will find an M^* and determine the query to be a final query.

Using a data structure such as a linked list to maintain T in sorted order, S takes time at most $O(q_S)$ to respond to each of the q_S compression function queries. Therefore, the running time of the simulator is $O(q_S^2)$.

To prove the indifferentiability of the MD6 mode of operation, we consider any distinguisher D making exactly q_S compression function queries and generating exactly q_t compression function calls from its MD6 queries and its compression function queries. We consider a sequence of games, G_0 through G_4. For each game G_i, let p_i denote the probability that D outputs 1 in G_i. We start with the "ideal" game and end with the "real" game. We show that between

consecutive games, the view of the distinguisher cannot differ with more than negligible probability.

Game G_0. This is the interaction of D with the random oracle F and the polynomial time simulator S defined above.

Game G_1. In game G_1, D interacts with F and a modified simulator S'. At the start of the game, S' uses its random coins to specify a random oracle $\mathcal{O}_S : \mathbf{W}^{n-q} \rightarrow \mathbf{W}^c$. The behavior of the new simulator S' is identical to that of S, except that wherever S uses a fresh random string $C^* \in \mathbf{W}^c$ to answer a query x^*, S' uses $\mathcal{O}_S(x^*)$. Wherever S uses a fresh random string $C' \in \{0,1\}^{cw-d}$ in constructing the answer to a query x^*, S' uses the first $cw - d$ bits of $\mathcal{O}_S(x^*)$. In other words, in G_1, S generates its random bits in a lazy manner, whereas in G_2, S' generates its random bits by calling a random oracle $\mathcal{O}_S$ (specified by the random coins of S'). In both games, the simulator produces a fresh random string in exactly the same cases. Clearly, the view of the distinguisher is the same in G_1 and in G_0, so $p_1 = p_0$.

Game G_2. In G_2, D interacts with a dummy relay algorithm R_0, instead of with F. The relay algorithm R_0 has oracle access to F, and upon receiving a hash function query, it forwards the query to F and forwards F's response to D. Clearly, the view of D is unchanged from G_0, so $p_2 = p_1$.

Game G_3. In G_3, we modify the relay algorithm. The new relay algorithm R_1 does not query F. Instead, R_1 computes the MD6 mode of operation on its input, querying the simulator S' for each call to the compression function f_Q.

In order to argue that the view of D differs between G_2 and G_3 with only negligible probability, we first define some "bad" events. We then argue that, as long as no bad events occur, the view of D is identical in games G_2 and G_3.

For a compression function query x_j, suppose x_j parses into $K_j, \ell_j, i_j, r_j, L_j, z_j, p_j, keylen_j, d_j, B_j$. We define three types of bad events that can occur in either game.

Type 1 A Type 1 event occurs when the outputs of S' on two different inputs are equal. Specifically, a Type 1 event occurs when S' inserts a pair (x_2, C) into T when there is already a pair (x_1, C) in T such that $x_1 \neq x_2$.

Type 2 A Type 2 event occurs when the output of S' on a query x_2 is such that a previous query x_1 "depends" on the answer to x_2. Specifically, a Type 2 event occurs when S' inserts a pair (x_2, C_2) into T when there is already a pair (x_1, C_1) in T such that $L_1 = L_2$ and one of the following two conditions is true:

- $0 \leq \ell_1 \leq L_1$, $\ell_1 = \ell_2 + 1$, $i_1 = \lfloor i_2 \cdot c/b \rfloor$, and C_2 is the kth c-word chunk of B_1, where $k = i_2 \bmod b/c$.
- $\ell_1 = \ell_2 = L_1 + 1$, $i_1 = i_2 + 1$, and C_2 is the first chunk of B_1.

Type 3 A Type 3 event occurs when S' receives a query x_2 from D that "depends" on an answer which was given by S' only to the relay algorithm,

never to D. Specifically, a Type 3 event occurs when S' receives a query x_2 from D when there is already a pair (x_1, C_1) in T such that $L_1 = L_2$ and one of the following two conditions is true:

- $0 \leq \ell_2 \leq L_1$, $\ell_2 = \ell_1 + 1$, $i_2 = \lfloor i_1 \cdot c/b \rfloor$, and C_1 is the kth chunk of B_2, where $k = i_1 \bmod b/c$, and C_1 has been given as the answer to a query x_1 only to R_1, never to D.
- $\ell_1 = \ell_2 = L_1 + 1$, $i_2 = i_1 + 1$, and C_1 is the first chunk of B_2, and C_1 has been given as the answer to a query x_1 only to R_1, never to D.

We argue below that if none of these bad events occur, then the view of D in G_2 and G_3 is the same.

Lemma 15 *For fixed coins of F, S', and D, if no "bad" events occur in game G_3, then the view of D is identical in games G_2 and G_3.*

Proof: Fix the random coins of F, S', and D. Assume no "bad" events occurred. We show by induction that the observable values (the responses of R_0/R_1 and S') are identical in games G_2 and G_3. (As long as the observable values are the same, D will make the same queries in both games.)

Suppose that the observable values have been the same so far. The next query by D is either to MD6 or to the compression function.

Mode of Operation Query. Consider a query M^* to $\mathcal{M}^{f_Q}$. In game G_2, R_0 always returns $F(M^*)$. In game G_3, R_1 returns $F(M^*)$ as long as on R_1's final compression function query (call this x^*) to S', S' queries F on M^*, or S' has already seen the query x^* and queried F on M^* the first time it was queried on x^*. If S' has not already seen the query x^*, S' will return $F(M^*)$, since in the case that it finds multiple values of M which could have generated x^* as its final compression function call, S' will choose the M for which the intermediate compression function calls occurred the most recently. Therefore, the only way that R_1's answer can differ from R_0's answer is if S' has already seen the query x^* and did not query F on M^* the first time it saw the query (call that time τ).

In this case, when S' was originally queried on x^* at time τ, it must have either (a) found no M or (b) found an $M \neq M^*$ which could have generated the query as its final compression function call. Assume (a), i.e., S' found no such M. Consider R_1's most recent query to S' before x^*. (Recall that R_1 computes the mode of operation by making the compression function queries in order.) That query must have also been seen by S' before time τ. Otherwise, a Type 2 event occurred in G_3 because the output of that query matched a chunk of the query at time τ. Now consider R_1's next most recent query. Again, by the same reasoning, it was also seen before time τ. By induction, all of the queries R_1 makes to S' on the message M^* must have been seen before time τ, which means that case (a) is impossible.

Now consider case (b). We have already argued that all of the queries R_1 makes to S' on the message M^* must have been seen before time τ. Therefore,

when S' was originally queried on x^* at time τ, it must have found both M^* and an $M \neq M^*$ which could have generated x^* as its final compression function call. Consider the b-word data input B^* of x^*. For each of the c-word chunks of B^*, either there was a unique pair in T at time τ with that chunk as the response, or there were multiple such pairs. If there were multiple such pairs, then a Type 1 event occurred. Otherwise, consider of all of the b-word data inputs of the compression function queries made by R_1 on the message M^*, at the next lower level. By induction, a Type 1 event must have occurred at some time; otherwise, S' would only have found one M (equal to M^*) that could have generated the final compression query. Therefore, case (b) is also impossible.

Thus, given that the values observable to D have been the same so far, if the next query is to the mode of operation, the next observable value will also be the same in games G_2 and G_3.

Compression Function Query. Now consider the case that the next query by D is a compression function query x^*. In both games, the answer will be equal to $\mathcal{O}_S(x^*)$, unless x^* is (or has previously been) determined to be the final query for some message M, in which case it will be equal to $C'||F(M)$ where $C' = \gamma_d(\mathcal{O}_S(x^*))$. Thus, there are three possible reasons for the answers in the two games to be different: (a) the answer is equal to $C'||F(M)$ in G_2 but $\mathcal{O}_S(x^*)$ in G_3; (b) the answer is equal to $C'||F(M)$ in G_3 but $\mathcal{O}_S(x^*)$ in G_2; or (c) the answer is equal to $C'||F(M)$ in G_2 but $C'||F(M')$ in G_3 for $M \neq M'$.

Consider case (a). In G_3, S' answers with $\mathcal{O}_S(x^*)$ because either (a.1) S' already answered a query on x^* before or (a.2) S' cannot find a sequence of queries that leads to M for which x^* is the final query. Case (a.1) is not possible because for S' to have the seen the query x^* before in G_3 but not in G_2 means that the query came from R_1, and if a final query comes from R_1 in G_3, then R_1 has asked all of the intermediate queries, and therefore it will be answered by $C'||F(M)$. Case (a.2) is not possible because the queries S' has received in G_2 are a subset of the queries S' has received in G_3 (in G_3, S' has received all the same queries as in G_2 from D, as well as queries from R_1). Therefore, if S' finds an M in G_2, it will also find some M in G_3.

In case (b), S' finds an M in G_3 but not in G_2. Then at least one query in the sequence used to construct M came from R_1 but not from D. But the final query (namely, x^*) came from D. By induction, a Type 3 event must have occurred, i.e., a chunk of one of D's compression function queries matched a compression function output that was answered to R_1 but never seen by D. Therefore, case (b) is impossible.

In case (c), S' finds M in G_2 and M' in G_3. But since the queries to S' in G_2 are a subset of the queries to S' in G_3, in G_3 S' must find both M and M'. Then, again, a Type 1 event must have occurred somewhere in the computation, making case (c) also impossible.

Thus, given that the values observable to D have been the same so far, if the next query is to the compression function, the next observable value will also be the same in games G_2 and G_3.

Thus, conditioned on there being no occurrences of bad events in G_3, the view of D is the same in games G_2 and G_3. This completes the proof of Lemma 15

∎

We now bound the probability of bad events in G_3. Let Pr[Type i] denote the probability that a Type i event ever occurs during the run of the simulator. Let Pr[Bad] = Pr[Type 1] + Pr[Type 2] + Pr[Type 3] denote the probability of a bad event ever occurring.

A Type 1 event corresponds to a collision between two random c-word strings among at most q_t compression function queries. Note that even though S' sometimes consults F for d bits of the random c-word output, these d random bits are fresh because on different queries to S', S' will make different queries to F (if it queries F at all). We can bound the probability of a Type 1 event using the birthday bound: $\Pr[\text{Type 1}] \leq q_t^2/2^{cw+1}$.

A Type 2 event corresponds to a collision between a random c-word string and a chunk of one of the previous queries to the compression function. For each of the q_t compression function queries x, in each of the previous compression function queries, there is at most one c-word chunk with a matching node identifier with which x could collide to cause a Type 2 event. Therefore, we can again bound the probability using the birthday bound: $\Pr[\text{Type 2}] \leq q_t^2/2^{cw+1}$.

A Type 3 event corresponds to a collision between a random c-word string that is not seen by D and a chunk of one of D's compression function queries. The probability of D "guessing" one of these hidden outputs is $1/2^{cw}$. There are at most $(q_t - q_S)$ hidden outputs to guess, and D has at most q_t guesses. Therefore, $\Pr[\text{Type 3}] \leq (q_t(q_t - q_S))/2^{cw}$.

Summing together, we get $\Pr[\text{Bad}] \leq (q_t^2 + q_t(q_t - q_S))/2^{cw} \leq 2q_t^2/2^{cw}$.

Therefore, the probability of D distinguishing between games G_2 and G_3 is at most the probability of a bad event occurring, so $|p_3 - p_2| \leq \Pr[\text{Bad}] \leq 2q_t^2/2^{cw}$.

Game G_4. This is the final game. In G_4, the relay algorithm R_1 remains unchanged but the simulator no longer consults F. Instead, the new simulator, S_1, always responds to a new query x^* with $\mathcal{O}_S(x^*)$. Thus, R_1 computes the mode of operation applied to the random oracle $\mathcal{O}_S(x^*)$.

To see that the view of the distinguisher is unchanged from G_3, consider a query x^* (from either D or R_1) to S_1. If the query has been seen before, S_1 repeats the answer it gave the first time it was asked the query. If it is a new query, then in G_4, S_1 responds with the fresh random c-word string $\mathcal{O}_S(x^*)$. In G_3, S' either responded with the fresh random c-word string $\mathcal{O}_S(x^*)$ or the fresh random c-word string obtained by concatenating the first $(cw - d)$ bits of $\mathcal{O}_S(x^*)$ with a fresh random d-bit string obtained from F. The only way that the random d-bit string from F could not be fresh is if S' already consulted F on the same message before, but this would mean that the query x^* was seen before. Thus, the view of D is unchanged from G_3 to G_4, and $p_4 = p_3$.

Summing over all the games, the total advantage of D in distinguishing between G_0 and G_4 is at most the claimed $2q_t^2/2^{cw}$. This completes the proof

of Theorem 8.

∎

7.8 Multi-Collision Attacks

In this section we again briefly discuss the class of recent "multi-collision" attacks studied by Joux [46], Nandi et al. [68, 69], Hoch et al. [44], and Yu et al. [97]. (Our previous discussion was in Section 3.8.)

These attacks leverage a compression function that is known not to be collision-resistant to obtain collisions in hash function modes of operation that are intended to compensate for or mitigate such weaknesses. Often, a surprising number of such collisions can be obtained, allowing further birthday attacks to be mounted.

MD6 adopts the "wide-pipe" strategy proposed by Lucks [54, 55]: all intermediate chaining variables in MD6 are 1024 bits in length, making collisions of the compression function extremely unlikely.

Thus, unless there is some unforeseen defect in the MD6 compression function, such "multi-collision" attacks will not be effective against MD6.

7.9 Length-Extension Attacks

In this section we review briefly the class of "length-extension" attacks.

Such attacks arise for a keyed hash function, when the (secret) key is input only at the beginning of the computation, e.g., by using it to initialize the initial state of a Merkle-Damgård sequential hash function computation.

In those functions, the problem is that an adversary who sees only X and $H_K(X)$ may be able to compute $H_K(X||Y)$ for any string Y of his choice. This happens if the output of H_K is merely the current state (e.g. of the Merkle-Damgård computation), so that continuing the computation from that point on is feasible if you know the hash function output.

MD6 defeats such attacks in many ways:

- The key K is input to every compression function operation, so that you cannot compute any key-dependent hash output without actually knowing the key.
- The final compression function is distinguished by having a z-bit input set equal to one; this explicitly prevents such length-extension attacks as well, since one hash-function computation is then never a sub-computation of another hash-function computation.
- MD6 does not output a full internal chaining variable state, but only a truncated version of it. (This is again an application of Lucks' "wide-pipe" strategy [54, 55].) Therefore, knowing the hash function output does not reveal any complete internal chaining variable state.

Therefore, MD6 is not vulnerable to such "length-extension" attacks.

7.10 Summary

We have shown that the MD6 mode of operation provides excellent protection against attack, assuming that the underlying compression function satisfies certain assumptions.

In particular, we have shown that MD6 is "property-preserving" with respect to the properties of collision-resistance and pre-image resistance, and argued that MD6 is stronger than required by NIST SHA-3 requirements with respect to second-preimage resistance.

We have shown, using Maurer's Random System framework, that MD6 is a pseudo-random function, if the MD6 compression function is pseudo-random.

We have further shown, under our "Key Blinding Assumption," that the MD6 mode of operation acts as a domain extender for fixed-input-length MAC's.

We have also shown that MD6 is indifferentiable from a random oracle; this is particularly meaningful since the framework here gives the adversary oracle access to the underlying compression function, which reflects accurately the real-world situation.

Chapter 8

Applications and Compatibility

This chapter addresses NIST requirements of section 2.B.1 [70] on compatibility with existing standards (specifically HMAC), and of section 4.A regarding certain evaluation criteria (specifically having to do with PRF's and randomized hashing).

It also describes and documents the `md6sum` utility that we provide as part of our submission package.

8.1 HMAC

The NIST requirements [70, Section 4.A.ii] state that a candidate algorithm must "have at least one construction to support HMAC as a PRF." Moreover, "when the candidate algorithm is used with HMAC as specified in the submitted package, that PRF must resist any distinguishing attack that requires much fewer than $2^{n/2}$ queries and significantly less computation than a preimage attack.

We believe that MD6 meets these requirements.

The HMAC standard is described in [72].

Although MD6 is tree-based (hierarchical) rather than iterative, and although HMAC was designed with iterative hash functions in mind, there is no difficulty in adapting MD6 to use within an HMAC framework. All that is really required is that the message block length be larger than the message digest size. Since the MD6 message block size is 512 bytes, this requirement is automatically met.

We propose using the HMAC framework with MD6 in its default fully hierarchical mode of operation, primarily for efficiency reasons.

One could also consider using MD6 in its sequential ($L = 0$) mode of operation, with a 384-byte message block length. This would be more like traditional

HMAC constructions based on iterative hash functions. But we favor the hierarchical mode of operation here.

Our security analyses (see Section 7.7 in particular) argue that MD6 should be indifferentiable from a random oracle in such applications.

8.2 PRFs

NIST invites submitters to describe any alternative (other than HMAC) methods by which their candidate algorithm may be used as a PRF.

MD6 provides the PRF functionality directly: one can set the MD6 key K to the desired value, and then apply MD6 to the input message. Our security proofs of Chapter 7 directly address the security of this mode of operation.

8.3 Randomized Hashing

The NIST evaluation requirements state [70, Section 4.A.2]:

"If a construct is specified for the use of the candidate algorithm in an n-bit randomized hashing scheme, the construct must, with overwhelming probability, provide $n - k$ bits of security against the following attack: The attacker chooses a message, $M1$ of length at most $2k$ bits. The specified construct is then used on $M1$ with a randomization value $r1$ that has been randomly chosen without the attacker's control after the attacker has supplied $M1$. Given $r1$, the attacker then attempts to find a second message $M2$ and randomization value $r2$ that yield the same randomized hash value. Note that in order to meet this specific security requirement, the specified randomized hashing construct may place restrictions on the length of the randomization value."

Since MD6 places the randomization value in the key input K, which is explicitly part of the compression function input, the specified attack is just a special case of a second-preimage attack, which we've already discussed.

8.4 `md6sum`

We provide a utility program, `md6sum`, that is similar to the traditional `md5sum` in functionality, but which is augmented for our experimental and development purposes. This section describes its usage.

The basic command-line format for `md6sum` is:

```
md6sum [OPTIONS] file1 file2 ...
```

The option letters are case-sensitive and are processed in order.

With no options, the files are each hashed. For each file, a summary line is printed giving the hash value in hex and the file name. The defaults are: digest size d is 256 bits, mode parameter L is 64, and key is **nil** (zero-length key). An initial date/time line is also printed (on a "comment line" starting with "`--`").

The "`b`" and "`B`" options provide the ability to process a standard dummy file of a specified length. The "`b`" option specifies the desired length in bits, while the "`B`" specifies the desired length in bytes. The dummy file is an initial prefix of the specified length of the infinite hexadecimal sequence:

```
11 22 33 44 55 66 77 11 22 33 44 55 66 77 11 22 ...
```

The length may be given in decimal or in scientific notation. Thus the following two commands are equivalent, and ask for the hash of a standard one-gigabyte dummy file:

```
md6sum -B1000000000
md6sum -B1e9
```

The "`b`" and "`B`" options are useful for getting MD6 timing estimates, as they do not involve disk I/O overhead.

The "`M`" option is used to specify a message directly on the command line, as in:

```
md6sum -Mabc
```

which prints the hash of the message string "`abc`". (If the message contains blanks, the message parameter should be quoted, as in `md6sum "-Ma b c"` .)

If file is '-', or if no files are given, standard input is used.

The "`d`", "`r`", "`L`", and "`K`" options are used to set MD6 parameters to non-default values. The defaults are a digest size of $d = 256$ bits, a number of rounds $r = 40 + (d/4)$, a mode parameters $L = 64$, and an empty key $K =$ **nil**. For example, the command

```
md6sum -d512 -Ksecret -r200 -L0 file1 file2
```

produces 512-bit hash values for files `file1` and `file2` computed using $r = 200$ rounds in sequential mode ($L = 0$) with a key $K =$"`secret`". The "`d`" and "`K`" options reset r to its default value for that digest size and key length, so if the "`r`" option is also given, it should follow any "`d`" and "`K`" options.

There are two options for performing timing measurements on MD6: "`t`" and "`s`".

The "`t`" option turns on the printing of timing information for each file (or dummy file) processed. The following information is printed for each file: length of the input file, the number of compression calls made, the elapsed time in seconds, the total number of clock ticks, the number of clock ticks per byte, and the number of clock ticks per compression function call. For example, the call

```
md6sum -t -B1e9
```

produces timing information for producing the MD6 hash of a standard one-gigabyte dummy file.

The "`s`" operation times the MD6 setup operation. Specifically, the option `-snnn` produces the time to perform `nnn` setup operations. For example,

```
md6sum -s1000000
```

measures the time to do one million MD6 initializations. The time is given both in clock ticks and in seconds.

Timing is measured in a straightforward way using the CPU's RDTSC instruction; no special effort is made to account for variability in the timing that might be caused by extraneous system effects. (Our experience suggests that these effects are not large when MD6 is applied to a large test input generated with the `-b` or `-B` options, on an otherwise idle machine.)

The "`i`" and "`I`" options cause intermediate values in the MD6 computation to be printed. The "`i`" option causes the input and output for each compression function call to printed in a pretty format. For example,

```
md6sum -i -Mabc
```

causes the compression function input/output to be printed for the single compression function call required to hash the message "`abc`". That is, the array values $A[0..n-1]$ (input) and $A[rc+n-c..rc+n-1]$ (output) will be printed, one word per line. The "`I`" option is similar, but also prints out the intermediate values for each MD6 compression function call. That is, the entire array $A[0..rc+n-1]$ will be printed. As an example, the call

```
md6sum -I -B20
```

will print out all intermediate values for the single compression call required to hash the standard twenty-byte dummy file.

The `md6sum` program can also be used with the "`c`" to check to see if a hash value for any of the given files has changed. If one saves the `md6sum` output, as in:

```
md6sum -d512 -L0 file1 file2 file3 >filehashes
```

(shown here with 512-bit hash values computed in sequential $L = 0$ mode) then a later call of the form:

```
md6sum -c filehashes
```

will check to see if any of the files now have hash values different than that recorded in the file `filehashes`. Any file whose hash value has changed will have its name printed out. (Comment lines in the saved file, such as the date/time line, are ignored.)

Chapter 9

Variations and Flexibility

The MD6 design exhibits great flexibility. It is easy to create "variant" designs by changing the word size, the size of a chunk, the key size, the size of Q, the number of rounds, etc.

9.1 Naming

We extend here the simple naming scheme given in Section 2.2.7.

If the other parameters *keylen*, L, or r are to receive non-default values, they can be given following the name, separated by hyphens, in the order *keylen*, L, r, with each parameter name followed immediately by its value (but using k to stand for *keylen*) as in the following examples:

```
MD6-256-L0            (sequential mode of operation)
MD6-256-r64           (64 rounds)
MD6-160-L0-r64        (sequential and 64 rounds)
MD6-384-k8-r64        (8 key bytes and 64 rounds)
MD6-512-k33-L2-r192   (33 key bytes, height bound 2, and 192 rounds)
```

In the last example, the key length is specified as 33 bytes; the key itself would be provided separately. Unspecified options receive their default values. In fully-specified notation, the standard versions of MD6 are:

```
MD6-224-k0-L64-r80       (224-bit digest, no key, tree-based, 80 rounds)
MD6-256-k0-L64-r104      (256-bit digest, no key, tree-based, 104 rounds)
MD6-384-k0-L64-r136      (384-bit digest, no key, tree-based, 136 rounds)
MD6-512-k0-L64-r168      (512-bit digest, no key, tree-based, 168 rounds)
```

But these default versions are also named MD6-224, MD6-256, MD6-384, and MD6-512.

9.2 Other values for compression input and output sizes: n and c

One can vary the values of n and c (the sizes, in words, of the compression function's input and output).

If one does so, the tap positions t_0, t_1, ..., t_5 would also need to be re-optimized.

We have written a program, `tapopt.c` to do such optimization. Table 9.1 lists some results from this program for various values of n and c. This is just a sample; it is simple to compute optimal tap positions for additional choices of n and c.

As an example, if one wished to define a variant MD6 that was practically identical to standard MD6, except that there was no key input (so $k = 0$ and $n = 81$), one could use the tap positions $17, 19, 22, 23, 52$, and 81 from the table row for $n = 81$, $c = 16$.

Or, one could imagine defining a variant of MD6 that had the same data field sizes in bits, but was defined in terms of 32-bit words. Thus one would have $w = 32$, $b = 128$, $v = u = 2$, $k = 16$, $q = 30$, $n = 178$, $c = 32$. One could use the tap positions $(33, 35, 49, 53, 111, 178)$ from the table, and the shift values given for 32-bit words in Table 9.3.

9.3 Other word sizes w

The word size w (in bits) could also be varied in the MD6 overall design. For example, the standard MD6 design could be adapted for 32-bit words, 16-bit words, or 8-bit words by setting $w = 32$, $w = 16$, or $w = 8$ rather than having $w = 64$. The information content of each field is reduced accordingly, but the designs produced in this way seem interesting and reasonable to consider.

Of course, changing the word size also requires changing the shift tables $r()$ and $\ell()$, as well as adjusting the constants used to define the recurrence for S. In addition, one has to clearly define how the constant Q is determined.

The following subsections make specific recommendations for such variations.

9.3.1 New round constants S

For different word sizes, the constants S'_0 and S^* can be obtained by truncating the values for $w = 64$ (taking just the high-order bits of each), as illustrated in Table 9.2.

These recurrences have period 217 (for $w = 8$), 63457 (for $w = 16$), 868 (for $w = 32$). (The short period for $w = 32$ shouldn't be a problem for our application, since the number of rounds is well less than 868. Or, you could change the constant S^* slightly to `0x7311c285`, and the period then jumps to 910,163,751.)

n	c	t_0	t_1	t_2	t_3	t_4	t_5	dep
17	8	9	10	12	14	15	17	26
19	8	10	11	13	14	17	19	29
32	16	17	21	25	29	31	32	67
35	8	9	12	13	19	34	35	40
35	16	17	18	22	24	31	35	70
37	8	9	11	14	26	36	37	39
37	16	17	19	22	30	31	37	68
38	16	17	23	29	31	35	38	79
39	16	17	19	22	28	37	39	73
40	16	17	21	27	31	29	40	81
41	8	9	10	12	13	27	41	39
41	16	17	19	20	26	34	41	73
42	16	17	19	23	31	41	42	83
44	16	17	19	23	37	43	44	81
45	8	11	13	14	17	31	45	54
45	16	17	19	23	29	37	45	74
46	16	17	19	25	31	45	46	84
48	16	17	19	25	29	47	48	79
64	16	17	19	23	37	57	64	90
64	32	33	35	43	55	61	64	170
70	16	17	19	27	47	69	70	98
70	32	33	39	51	53	61	70	190
72	16	17	19	25	29	71	72	99
72	32	35	41	49	61	71	72	198
74	16	17	21	27	45	73	74	102
74	32	33	35	49	55	73	74	184
76	16	17	21	23	43	73	76	102
76	32	33	35	47	53	75	76	185
78	16	17	19	23	43	73	78	103
78	32	35	37	49	59	77	78	202
81	8	10	11	14	37	55	81	63
81	16	17	19	22	23	52	81	99
81	32	34	35	43	49	74	81	173
82	16	17	23	25	47	75	82	106
82	32	33	37	51	57	73	82	190
83	16	17	18	20	29	62	83	98
83	32	33	34	37	47	75	83	169
84	16	17	23	25	53	83	84	110
84	32	37	41	53	71	79	84	218
89	8	9	10	12	35	63	89	61
89	**16**	**17**	**18**	**21**	**31**	**67**	**89**	**102**
89	32	33	35	38	44	81	89	175

n	c	t_0	t_1	t_2	t_3	t_4	t_5	dep
90	16	17	19	23	43	89	90	112
90	32	37	41	43	67	83	90	222
96	16	17	19	23	43	95	96	118
96	32	35	37	47	55	89	96	209
128	16	17	19	25	45	87	128	134
128	32	33	37	39	59	127	128	224
128	64	65	73	75	109	123	128	423
140	16	17	19	27	69	109	140	141
140	32	33	39	43	67	137	140	237
140	64	67	71	79	113	127	140	450
144	16	17	19	31	73	119	144	145
144	32	35	37	49	59	139	144	253
144	64	65	67	89	121	127	144	452
148	16	17	21	27	35	95	148	143
148	32	33	35	47	51	95	148	244
148	64	65	67	85	113	119	148	452
152	16	17	21	27	35	93	152	149
152	32	33	35	45	83	139	152	248
152	64	65	67	77	93	151	152	449
162	16	17	19	25	79	125	162	151
162	32	35	37	41	61	161	162	262
162	64	65	66	77	101	155	162	460
164	16	17	19	27	77	119	164	149
164	32	33	37	43	77	159	164	253
164	64	65	71	75	101	155	164	461
178	16	17	19	27	77	127	178	153
178	32	33	35	49	53	111	178	262
178	64	65	67	73	95	163	178	466
180	16	17	19	29	71	133	180	161
180	32	37	41	47	89	149	180	302
180	64	67	71	77	89	163	180	500
192	16	17	19	25	79	133	192	165
192	32	35	37	43	89	145	192	290
192	64	65	67	71	83	155	192	484
256	16	17	19	27	77	167	256	192
256	32	33	35	45	53	137	256	314
256	64	65	67	81	87	253	256	545

Table 9.1: Optimal feedback function tap positions $t_0 \ldots t_5$ for various values of n (the length of the feedback shift register) and c (the output size and the number of initial feedback positions excluded from being tapped). The value "dep" has the property that if at least "dep" steps are performed then each of the last c values computed depends on all n values in the initial register state. This table illustrates the potential flexibility of the MD6 design. However, standard MD6 uses the parameters from only one line of this table (shown in bold).

w	S'_0	S^*
64	0x0123456789abcdef	0x7311c2812425cfa0
32	0x01234567	0x7311c281
16	0x0123	0x7311
8	0x01	0x73

Table 9.2: Here are suggested values for S'_0 and S^* for various values of w.

9.3.2 New shift amounts r_i and ℓ_i

The shift amounts r_i and ℓ_i will need revision if another value of w is used.

Table 9.3 gives suggested shift amounts for $w = 64$ (the standard shift table), $w = 32$, $w = 16$, and $w = 8$. While they were optimized for $c = 16$ and the standard tap positions, they should work reasonably well for $c = 16$ and other tap positions.

These shift amounts were computed via the optimization program `shiftopt.c`; this program is included in our submission package. It can be examined and rerun for transparency, or to compute optimized shift tables for other parameter settings.

Roughly speaking, the optimization program attempts to maximize the rate at which input changes propagate through the computation.

To define MD6 for $w = 1$, the function g should be taken to be the identity function.

Some experiments were also done to evaluate shift amounts when the shifts were required to be the same for each step. These were found to be very noticeably worse in terms of their rate of diffusion (a crude estimate might be by as much as 50%). But it might nonetheless be of interest to see what the optimum shift values were, under this constraint. Table 9.4 gives these values.

9.3.3 Constant Q

If the word size w is changed, then the string of bits representing the fractional part of $\sqrt{6}$ just needs to be divided up into w-bit words in the natural manner. For example, $w = 8$, then $Q_0 =$ `0x73`, $Q_1 =$ `0x11`, etc.

9.4 Constant Q

One could change the constant Q to some value other than the fractional part of $\sqrt{6}$.

For example, one might consider setting Q to zero. Our proof in Section 6.1.2 of the indifferentiability of the compression function from a random oracle depends on Q being fixed, but does not require Q to have any fixed value. As far as this proof is concerned, Q could indeed be all zeros. However, some of the

$w = 64$																
i mod 16	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
r_i	10	5	13	10	11	12	2	7	14	15	7	13	11	7	6	12
ℓ_i	11	24	9	16	15	9	27	15	6	2	29	8	15	5	31	9

$w = 32$																
i mod 16	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
r_i	5	3	6	5	4	6	7	3	5	6	5	5	4	6	7	5
ℓ_i	4	7	7	9	13	8	4	14	7	4	8	11	5	8	2	11

$w = 16$																
i mod 16	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
r_i	5	4	3	5	7	5	5	2	4	3	4	3	4	7	7	2
ℓ_i	6	7	2	4	2	6	3	7	5	7	6	5	5	6	4	3

$w = 8$																
i mod 16	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
r_i	3	3	3	4	3	3	3	3	2	2	3	2	2	3	2	3
ℓ_i	2	4	2	3	2	2	2	4	3	3	2	3	3	4	3	4

Table 9.3: This table gives suggested shift amounts for various potential word sizes $w = 64, 32, 16, 8$. These were optimized for the standard taps, but should probably work well for arbitrary tap positions.

w	r	l
64	7	6
32	4	5
16	3	4
8	2	3

Table 9.4: Optimum shift amounts when every round must have the same shift amount.

algebraic attacks studied in Section 6.11.2 strongly suggest that MD6 may be stronger when Q is non-zero. Thus, setting $Q = 0$ is inadvisable.

However, setting Q to other "random-looking" constants should work as well as the current choice; but it is important that Q remain a fixed constant.

9.5 Varying the number r of rounds

It is easy to vary the number r of rounds in the MD6 compression function. Indeed, we have made r an explicit (but optional) input parameter to MD6.

One may wish to do so to study the security of reduced-round versions (as in Chapter 6), to obtain different efficiency/security tradeoff points, or to study the security of "crippled" versions of MD6.

Since MD6 is designed in a very conservative manner, it may happen that over time we will gain confidence in the security of reduced-round versions of MD6, and feel comfortable in using such reduced-round versions in practice, or even adopting them in some standards. (At the moment, for example, we don't know how to break even 24-round MD6, much less 96 or 168 round versions.) However, MD6 is still a "new design," and prudence would dictate erring on the side of conservatism, particularly given the history of hash function security to date.

9.6 Summary

The MD6 framework has tremendous flexibility. It is easy to create variant MD6 designs for study or for actual use in situations requiring non-standard parameterization. Of course, the security of such variants should be studied carefully before actual use.

Chapter 10

Acknowledgments

We'd like to thank Juniper Networks for an unrestricted gift of $15K to MIT that was used to support the development and analysis of MD6. (Just for the record, Juniper Networks has no intellectual property rights to MD6.)

Thanks to the CILK team, both at MIT and at CILK Arts, for the help with porting MD6 to multicore processors and running benchmark tests on these multicore processors. Thanks particularly to Charles Leiserson, Bradley Kuszmaul, and Steve Lewin-Berlin.

Thanks to CSAIL's TIG (The Infrastructure Group) for help with computing infrastructure.

Thanks to Sarah Cheng, who assisted in some early discussion of parallel implementations of MD6, and in some proof-reading of this report.

Thanks to Jean-Philippe Aumasson for providing his results on certain algebraic attacks, and allowing us to mention them in this report.

Thanks similarly to Itai Dinur for his collaborative efforts to evaluate the effectiveness of his "cube attack" (developed with Adi Shamir) on MD6.

Chapter 11

Conclusions

This report has presented the MD6 cryptographic hash algorithm, for consideration by NIST as a SHA-3 candidate.

We have presented a detailed description of MD6, including its design rationale, and have given in some detail our preliminary findings of its security and efficiency.

The standard MD6 design is very well-suited to take advantage of the forthcoming flood of multicore processors. The sequential variant of MD6 fits well on an 8-bit processor. MD6 works very well on special-purpose processors (e.g. GPGPU's), and is easy to implement effectively on special-purpose hardware.

From a security perspective, MD6 is very conservatively designed. While the state of the art does not enable one to rigorously prove the security of a hash function against all attacks (indeed, we don't know if $P = NP$), we believe that our security analyses of MD6, although they are only preliminary, provide strong justification for believing that MD6 easily meets the security requirements for a SHA-3 standard.

The provable resistance of MD6 to differential and linear attacks is particular noteworthy, as are the proofs that the compression function and mode of operation are indifferentiable from a random oracle.

We suggest that MD6 meets NIST's stated goals of simplicity, efficiency on a variety of platforms, security, and flexibility, and would hope that MD6 will be very seriously considered as a SHA-3 candidate hash algorithm.

Bibliography

[1] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In CT-RSA, pages 225–242, 2007.

[2] Onur Aciiçmez and Werner Schindler. A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl. In CT-RSA, pages 256–273, 2008.

[3] Jee Hea An and Mihir Bellare. Constructing VIL-MACs from FIL-MACs: Message authentication under weakened assumptions. In Michael J. Wiener, editor, CRYPTO, volume 1666 of Lecture Notes in Computer Science, pages 252–269. Springer, 1999.

[4] Atmel. Avr 8-bit RISC. `http://www.atmel.com/dyn/products/tools.asp?family_id=607`.

[5] AVR microcontroller. `http://en.wikipedia.org/wiki/Atmel_AVR`.

[6] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In N. Koblitz, editor, Proc. CRYPTO '96, number 1109 in LNCS, pages 1–15. Springer, 1996. Full version at `http://www-cse.ucsd.edu/~mihir/papers/hmac.html`.

[7] Mihir Bellare, Roch Guérin, and Phillip Rogaway. XOR MAC's: New methods for message authentication using finite pseudorandom functions. In D. Coppersmith, editor, Proc. CRYPTO '95, number 963 in LNCS, pages 15–28. Springer, 1995.

[8] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The Security of the Cipher Block Chaining Message Authentication Code. J. Comput. Syst. Sci., 61(3):362–399, 2000.

[9] Mihir Bellare, Krzysztof Pietrzak, and Phillip Rogaway. Improved Security Analyses for CBC MACs. In Victor Shoup, editor, CRYPTO, volume 3621 of Lecture Notes in Computer Science, pages 527–545. Springer, 2005.

[10] Mihir Bellare and Thomas Ristenpart. Hash functions in the dedicated-key setting: Design choices and mpp transforms. Cryptology ePrint Archive, Report 2007/271, 2007. `http://eprint.iacr.org/2007/271.pdf`.

[11] Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In ACM Conference on Computer and Communications Security, pages 62–73, 1993.

[12] Daniel J. Bernstein. Cache-timing attacks on AES, 2005. `http://cr.yp.to/papers.html#cachetiming`.

[13] Guido Bertoni, Joan Daeman, Michael Peeters, and Gilles Van Assche. Sponge functions, May 2007. `http://www.csrc.nist.gov/pki/HashWorkshop/PublicComments/2007May.html`.

[14] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In Proc. Eurocrypt 2008, volume 4965 of LNCS, pages 181–197. Springer, 2008.

[15] E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. J. Cryptology, 4(1):3–72, 1991.

[16] Eli Biham and Rafi Chen. Near-collisions of SHA-0 . In Advances in Cryptology – CRYPTO '04, number 3152 in LNCS, pages 290–305. Springer, 2004.

[17] Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, and William Jalby. Collisions of SHA-0 and reduced SHA-1. In Advances in Cryptology – EUROCRYPT '05, number 3621 in LNCS, pages 36–57, 2005.

[18] Eli Biham and Orr Dunkelman. A framework for iterative hash functions – HAIFA, August 2006. NIST 2nd Hash Function Workshop, Santa Barbara. `http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-info.cgi?2007/CS/CS-2007-15`.

[19] J. Black, P. Rogaway, and T. Shrimpton. Black-box analysis of the blockcipher-based hash-function constructions from PGV. In Proc. Crypto '02, volume 2442 of LNCS, pages 103–118. Springer, 2002. `http://www.cs.colorado.edu/~jrblack/papers/hash.pdf`.

[20] John Black, Martin Cochran, and Thomas Shrimpton. On the impossibility of highly-efficient blockcipher-based hash functions. In R. Cramer, editor, Proc. Eurocrypt 2005, volume 3494 of LNCS, pages 526–541. Springer, 2005. `http://www.cs.colorado.edu/~jrblack/papers/ohash.pdf`.

[21] Sergey Bratus and Igor Pak. Fast constructive recognition of a black box group isomorphic to s_n or a_n using goldbach's conjecture. J. Symbolic Computation, 29:33–57, 2000.

[22] David Brumley and Dan Boneh. Remote timing attacks are practical. Computer Networks, 48(5):701–716, 2005.

[23] Ran Canetti. Towards realizing random oracles: Hash functions that hide all partial information. In Lecture Notes in Computer Science, volume 1294, pages 455–469, 1997.

[24] Ran Canetti, Oded Goldreich, and Shai Halevi. The Random Oracle Methodology, Revisited. CoRR, cs.CR/0010019, 2000. informal publication.

[25] Christophe De Canniére and Christian Rechberger. Finding SHA-1 characteristics: General results and applications. In Proc. ASIACRYPT '06, volume 4284 of LNCS, pages 1–20. Springer, 2006.

[26] CardLogix. Smart card product selection guide, 2005. `http://www.cardlogix.com/pdf/ProductSelectionGuide.pdf`.

[27] Florent Chabaud and Antoine Joux. Differential collisions of SHA-0 . In Advances in Cryptology – CRYPTO '98, number 1462 in LNCS, pages 56–71. Springer, 1998.

[28] Ricardo Chaves, Georgi Kuzmanov, Leonel Sousa, and Stamatis Vassiliadis. Improving sha-2 hardware implementations. In In Workshop on Cryptographic Hardware and Embedded Systems, CHES 2006, 2006.

[29] Debra L. Cook, John Ioannidis, Angelos D. Keromytis, and Jake Luck. Cryptographics: Secret key cryptography using graphics cards. In CT-RSA, pages 334–350, 2005.

[30] D. Coppersmith and E. Grossman. Generators for certain alternating groups with applications to cryptology. SIAM J. Applied Mathematics, 29:624–627, 1975.

[31] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In Advances in Cryptology – CRYPTO '05, number 3621 in LNCS, pages 430–448. Springer, 2005.

[32] Christopher Yale Crutchfield. Security proofs for the MD6 hash function mode of operation. Master's thesis, MIT EECS Department, 2008. `http://groups.csail.mit.edu/cis/theses/crutchfield-masters-thesis.pdf`.

[33] Joan Daemen and Vincent Rijmen. The Design of Rijndael: AES–the Advanced Encryption Standard. Springer, 2002.

[34] Ivan Bjerre Damgård. A design principle for hash functions. In G. Brassard, editor, Advances in Cryptology – CRYPTO '89, number 435 in LNCS, pages 416–427. Springer, 1990.

[35] Yevgeniy Dodis, Krzysztof Pietrzak, and Prashant Puniya. A new mode of operation for block ciphers and length-preserving MACs. In Nigel P. Smart, editor, EUROCRYPT, volume 4965 of Lecture Notes in Computer Science, pages 198–219. Springer, 2008.

[36] Håkan Englund, Thomas Johansson, and Meltem Sönmez Turan. A framework for chosen iv statistical analysis of stream ciphers. In Proc. Indocrypt '07, volume 4859, pages 268–281. Springer, Dec. 2007. `http://www.impan.gov.pl/BC/Program/conferences/07Crypt-abs/Turan%20-%20A%20framework%20for%20Chosen%20IV%20Statistical%20Analysis.pdf`.

[37] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In Andrew M. Odlyzko, editor, CRYPTO, volume 263 of Lecture Notes in Computer Science, pages 186–194. Springer, 1986.

[38] Claudia Fiorini, Enrico Martinelli, and Fabio Massacci. How to fake an rsa signature by encoding modular root finding as a sat problem. Discrete Applied Mathematics, 130(2):101–127, 2003.

[39] Simon Fischer, Shahram Khazaei, and Willi Meier. Chosen IV statistical analysis for key recovery attacks on stream ciphers. In Serge Vaudenay, editor, AFRICACRYPT, volume 5023 of LNCS, pages 236–245. Springer, 2008.

[40] International Technology Roadmap for Semiconductors. International technology roadmap for semiconductors 2007, 2007. `http://www.itrs.net/Links/2007ITRS/Home2007.htm`.

[41] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. On the Cryptographic Applications of Random Functions. In CRYPTO, pages 276–288, 1984.

[42] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to Construct Random Functions. J. ACM, 33(4):792–807, 1986.

[43] Owen Harrison and John Waldron. Practical symmetric key cryptography on modern graphics hardware. In Proc. USENIX '08, pages 195–209. USENIX Assoc., 2008. `http://www.usenix.org/events/sec08/tech/harrison.html`.

[44] Jonathan J. Hoch and Adi Shamir. Breaking the ICE—finding multicollisions in iterated concatenated and expanded (ICE) hash functions. In Proc. Fast Software Encryption 2006, volume 4047 of LNCS, pages 179–194. Springer, 2006.

[45] Kimmo Jarvinen, Matti Tommiska, and Jorma Skytta. Hardware implementation analysis of the md5 hash algorithm. In HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on

System Sciences (HICSS'05) - Track 9, page 298.1, Washington, DC, USA, 2005. IEEE Computer Society.

[46] Antoine Joux. Multicollisions in iterated hash functions. In Advances in Cryptology – CRYPTO '04, number 3152 in LNCS, pages 306–316. Springer, 2004.

[47] Dejan Jovanovic and Predrag Janicic. Logical analysis of hash functions. In FroCos, pages 200–215, 2005.

[48] Charanjit S. Jutla and Anindya C. Patthak. A simple and provably good code for SHA message expansion, 2005. `http://eprint.iacr.org/2005/247`.

[49] Burt Kaliski and Matt Robshaw. Message authentication with MD5. RSA Labs Cryptobytes, 1(1):5–8, 1995. `http://citeseer.ist.psu.edu/433722.html`.

[50] John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n. In Advances in Cryptology – EUROCRYPT '05, number 3494 in LNCS, pages 474–490, 2005.

[51] P. Kitsos and O. Koufopavlou. Efficient architecture and hardware implementation of the whirlpool hash function. IEEE Trans. Consumer Electronics, 50(1):218–213, Feb 2004.

[52] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. IEEE Micro, 25(2):21–29, 2005.

[53] Pierre L'Ecuyer and Richard Simard. Testu01: A c library for empirical testing of random number generators. ACM Trans. Math. Softw., 33(4):22, 2007.

[54] Stefan Lucks. Design principles for iterated hash functions, Sep. 29 2004. `http://eprint.iacr.org/2004/253`.

[55] Stefan Lucks. A failure-friendly design principle for hash functions. In Proceedings of Asiacrypt 2005, number 3788 in Lecture Notes in Computer Science, pages 474–494. Springer-Verlag, 2005. `http://www.springerlink.com/content/f868557868208018/`.

[56] G. Marsaglia. Xorshift RNGs. J. Stat. Soft., 8(14):1–6, 2003. `http://www.jstatsoft.org/v08/i14/xorshift.pdf`.

[57] Fabio Massacci. Using walk-sat and rel-sat for cryptographic key search. In IJCAI, pages 290–295, 1999.

[58] M. Matsui. Linear cryptanalysis method for DES cipher. In Proc. EUROCRYPT'93, volume 765 of LNCS, pages 386–397. Springer, 1993.

[59] U. Maurer, R. Renner, and C. Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In Proc. TCC '04, volume 2951 of LNCS, pages 21–39. Springer, Feb. 2004.

[60] Ueli M. Maurer. Indistinguishability of Random Systems. In Lars R. Knudsen, editor, EUROCRYPT, volume 2332 of Lecture Notes in Computer Science, pages 110–132. Springer, 2002.

[61] Ueli M. Maurer and Krzysztof Pietrzak. Composition of Random Systems: When Two Weak Make One Strong. In Moni Naor, editor, TCC, volume 2951 of Lecture Notes in Computer Science, pages 410–427. Springer, 2004.

[62] Máire McLoone and Ciaran McIvor. High-speed & low area hardware architectures of the whirlpool hash function. J. VLSI Signal Process. Syst., 47(1):47–57, 2007.

[63] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. Handbook of Applied Cryptography. CRC Press, 1997. `http://www.cacr.math.uwaterloo.ca/hac/`.

[64] Ralph C. Merkle. One way hash functions and DES . In Advances in Cryptology – CRYPTO '89, number 435 in LNCS, pages 428–446. Springer, 1990.

[65] Ralphc C. Merkle. Secrecy, Authentication, and Public Key Systems. PhD thesis, Stanford University, 1982. UMI Research Press.

[66] H. E. Michail, A. P. Kakarountas, G. Theodoridis, and C. E. Goutis. A low-power and high-throughput implementation of the sha-1 hash function. In ICCOMP'05: Proceedings of the 9th WSEAS International Conference on Computers, pages 1–6, Stevens Point, Wisconsin, USA, 2005. World Scientific and Engineering Academy and Society (WSEAS).

[67] Ilya Mironov and Lintao Zhang. Applications of sat solvers to cryptanalysis of hash functions. In SAT, pages 102–115, 2006.

[68] M. Nandi and D. R. Stinson. Multicollision attacks on a class of hash functions. Cryptology ePrint Archive, Report 2004/330, 2004. `http://eprint.iacr.org/2004/330`.

[69] M. Nandi and D. R. Stinson. Multicollision attacks on some generalized sequential hash functions. IEEE Trans. Inform. Theory, 53(2):759–767, Feb. 2007.

[70] National Institute of Standards and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (sha-3) family. Federal Register Notices, 72(212):62212–62220, Nov. 2 2007. `http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf`.

[71] National Institute of Standards and Technology. Recommendation for key management — part 1: General (revised), March 2007. `http://csrc.nist.gov/groups/ST/toolkit/documents/SP800-57Part1_3-8-07.pdf`.

[72] National Institute of Standards and Technology. The keyed-hash message authentication code (HMAC). NIST Federal Information Processing Standards Publication FIPS 198-1, July 2008.

[73] Sean O'Neil. Algebraic structure defectoscopy. Cryptology ePrint Archive, Report 2007/378, 2007.

[74] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In Proc. RSA Conference Cryptographers Track (CT-RSA), number 3860 in LNCS, pages 1–20. Springer, 2006.

[75] François Panneton and Pierre L'Ecuyer. On the Xorshift random number generators. ACM Trans. Modeling and Computer Simulation, 15(4):346–361, Oct. 2005.

[76] Colin Percival. Cache missing for fun and profit, 2005. `http://www.daemonology.net/papers/htt.pdf`.

[77] Krzysztof Pietrzak. Indistinguishability and Composition of Random Systems. PhD thesis, ETH Zurich, 2006. Reprint as vol. 6 of ETH Series in Information Security and Cryptography, ISBN 3-86626-063-7, Hartung-Gorre Verlag, Konstanz, 2006.

[78] Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. A compact fpga implementation of the hash function whirlpool. In FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays, pages 159–166, New York, NY, USA, 2006. ACM.

[79] B. Preneel, R. Govaerts, and J. Vandewalle. Hash functions based on block ciphers: A synthetic approach. In Proc. Crypto '93, volume 773 of LNCS, pages 368–378. Springer, 1993.

[80] Bart Preneel. Hash functions – present state of art, 2005. ECrypt Conference on Hash Functions.

[81] Bart Preneel, Rene Govaerts, and Joos Vanderwalle. Differential cryptanalysis of hash functions based on block ciphers. In Proc. Computer and Communications Security, pages 183–188. ACM, 1993.

[82] Bart Preneel and Paul C. van Oorschot. MDx-MAC and building fast MACs from hash functions. In D. Coppersmith, editor, Proc. CRYPTO '96, number 963 in LNCS, pages 1–14. Springer, 1995.

[83] Bart Preneel and Paul C. van Oorschot. On the security of two MAC algorithms. In U. Maurer, editor, Proc. Eurocrypt '96, number 1070 in LNCS, pages 19–32. Springer, 1996.

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

[85] Phillip Rogaway. Formalizing human ignorance: Collision-resistant hashing without the keys. In Proc. Vietcrypt '06, volume 4341 of LNCS, pages 221–228. Springer, 2006.

[86] Phillip Rogaway and Thmas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision-resistance. In Proc. Fast Software Encryption 2004, number 3017 in LNCS, pages 371–388. Springer, 2004.

[87] P. Sarkar and P. J. Schellenberg. A parallelizable design principle for cryptographic hash functions, 2002. `http://eprint.iacr.org/2002/031`.

[88] Claus P. Schnorr and Serge Vaudenay. Black box cryptanalysis of hash networks based on multipermutations. In Proceedings EUROCRYPT '94, volume 950 of Lecture Notes in Computer Science, pages 47–57. Springer, 1995.

[89] N. Sklavos and O. Koufopavlou. Implementation of the sha-2 hash family standard using fpgas. J. Supercomput., 31(3):227–248, 2005.

[90] Andrew V. Sutherland. Order Computation in Generic Groups. PhD thesis, MIT Department of Mathematics, 2007. `http://groups.csail.mit.edu/cis/theses/sutherland-phd.pdf`.

[91] Eran Tromer. Hardware-Based Cryptanalysis. PhD thesis, Weizmann Institute of Science, May 2007.

[92] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In CHES, pages 62–76, 2003.

[93] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. Cryptanalysis of the hash functions MD4 and RIPEMD. In Proc. Eurocrypt 2005, number 3494 in LNCS, pages 1– 18. Springer, 2005.

[94] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In Proc. CRYPTO 2005, number 3621 in LNCS, pages 17–36. Springer, 2005.

[95] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In Proc. EUROCRYPT 2005, number 3494 in LNCS, pages 19–35. Springer, 2005.

[96] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. Efficient collision search attacks on SHA-0. In Proc. CRYPTO 2005, number 3621 in LNCS, pages 1–16. Springer, 2005.

[97] Hongbo Yu and Xiaoyun Wang. Multicollision attack on the compression functions of MD4 and 3-pass HAVAL. IACR eprint, 2007. `http://eprint.iacr.org/2007/085/`.

Appendix A

Constant Vector Q

The constant vector Q consists of 15 64-bit words that represent the fractional part of $\sqrt{6}$. That is, Q is a good approximation to the solution to the quadratic equation:

$$(2 + Q)^2 = 6 \ .$$

Here are the first 960 bits of $\sqrt{6}$ as a sequence of 15 64-bit words, in hexadecimal notation.

```
 0 0x7311c2812425cfa0
 1 0x6432286434aac8e7
 2 0xb60450e9ef68b7c1
 3 0xe8fb23908d9f06f1
 4 0xdd2e76cba691e5bf
 5 0x0cd0d63b2c30bc41
 6 0x1f8ccf6823058f8a
 7 0x54e5ed5b88e3775d
 8 0x4ad12aae0a6d6031
 9 0x3e7f16bb88222e0d
10 0x8af8671d3fb50c2c
11 0x995ad1178bd25c31
12 0xc878c1dd04c4b633
13 0x3b72066c7a1552ac
14 0x0d6f3522631effcb
```

Appendix B

Round Constants S

Here are the round constants S'_j for $0 \leq j \leq 167$. Of course, these values are easy to generate iteratively using equation (2.4); they are given here only for explicitness.

```
  0 0x0123456789abcdef
  1 0x0347cace1376567e
  2 0x058e571c26c8eadc
  3 0x0a1cec3869911f38
  4 0x16291870f3233150
  5 0x3e5330e1c66763a0
  6 0x4eb7614288eb84e0
  7 0xdf7f828511f68d60
  8 0xedee878b23c997e1
  9 0xbadd8d976792a863
 10 0x47aa9bafeb25d8e7
 11 0xcc55b5def66e796e
 12 0xd8baeb3dc8f8bbfd
 13 0xe165147a91d1fc5b
 14 0xa3cb28f523a234b7
 15 0x6497516b67646dcf
 16 0xa93fe2d7eaec961e
 17 0x736e072ef5fdaa3d
 18 0x95dc0c5dcfdede5a
 19 0x3aa818ba9bb972b5
 20 0x475031f53753a7ca
 21 0xcdb0636b4aa6c814
 22 0xda7084d795695829
 23 0xe6f1892e2ef3f873
 24 0xaff2925c79c638c7
 25 0x7cf5a6b8d388790f
 26 0x89facff1a710bb1e
 27 0x12e55d626a21fd3d
 28 0x37cbfac4f462375a
 29 0x5c963709cce469b4
```

```
30 0xe93c6c129dec9ac8
31 0xb36898253ffdbf11
32 0x55d1b04b5bdef123
33 0xfab2e097b7b92366
34 0x877501ae4b5345ed
35 0x0dfb03dc96a7ce7b
36 0x1ae70539296a52d6
37 0x27cf0a7372f4e72c
38 0x6c9f16e7c5cd0978
39 0xb92f2f4e8f9f1bd0
40 0x435f5c9d1b3b3c21
41 0xc5aff9bb36577462
42 0xca5e33f748abace5
43 0xd6ac656f9176d56b
44 0xff588ade22c96ff7
45 0x8da1973c6593904f
46 0x1a42ac78ef26a09f
47 0x2685d8f1fa69c1be
48 0x6f0a7162d4f242dc
49 0xbd14a2c5adc4c738
50 0x4b39c70a7f8d4951
51 0xd5624c14db1fdba2
52 0xfbc4d829b63a7ce5
53 0x848970524854b56b
54 0x0913a0a490adeff7
55 0x1336c1c9217e104e
56 0x357d431362d8209c
57 0x5bebc427e5b041b8
58 0xe4d6484eef40c2d0
59 0xa9bcd09dfa814721
60 0x726961bad503c963
61 0x96d383f5ae065be6
62 0x3fb6856a7808fc6d
63 0x4c7d8ad4d01134fa
64 0xd8ea9729a0236d54
65 0xe1d5ac52606797a9
66 0xa2bad8a4e0eaa8f3
67 0x676571c9e1f5d947
68 0xadcba312e3ce7b8e
69 0x7a96c425e798bc9d
70 0x873d484aeb31f5ba
71 0x0d6bd095f6422ed5
72 0x1bd661aac884532a
73 0x24bc83d5910ce574
74 0x6969852a221d0fc8
75 0xb3d28a54643f1010
76 0x54b596a8ec5b2021
77 0xf97aafd1fcb74062
78 0x83e5dd22dd4bc0e5
79 0x04ca7a45be96416b
```

```
 80 0x0994b68a5928c3f6
 81 0x1239ef94b271444c
 82 0x36621da944c3cc98
 83 0x5ec43bd38d8655b0
 84 0xef8875261f08eec0
 85 0xbc10aa4c3a111301
 86 0x4831d69854232503
 87 0xd0726fb0ac674f06
 88 0xf0f49de17cebd10d
 89 0x91f9bb43ddf6631b
 90 0x32e2f486bfc88537
 91 0x57c5298d5b918f4e
 92 0xfc8b539bb722919c
 93 0x8917e5b64a65a2b9
 94 0x133e0bec94eec7d3
 95 0x356c15592df94826
 96 0x5bd82ab37fd3d86c
 97 0xe4a057e7dba678f8
 98 0xa940ed4eb768b951
 99 0x73811a9d4af1fba3
100 0x940337bb95c23ce6
101 0x38076df62f84756d
102 0x400f9b6c7b0caffa
103 0xc01eb4d8d61dd054
104 0xc02de931a83e60a9
105 0xc05a1262705881f3
106 0xc0a426c4c0b18247
107 0xc1484f098142868f
108 0xc390dc1202858b9f
109 0xc4317824050e9cbf
110 0xc873b0480e19b5df
111 0xd0f6e0901832ee3f
112 0xf1fd01a03045125f
113 0x92eb03c0408f26bf
114 0x37d70500811b4bdf
115 0x5cbf0a010237dc3e
116 0xe96f1603044a745c
117 0xb3df2e070c94acb9
118 0x54af5e0f1d2dd5d3
119 0xf95ffe1f3e7e6e26
120 0x83ae3e3f58d8926d
121 0x045c7e7fb1b1a6fb
122 0x08a8befe4342cb56
123 0x1151ff7c86855dac
124 0x33b23cf9090ff6f8
125 0x54747973121a2b50
126 0xf8f8b2e724345da0
127 0x81e1e74f6c4cf6e1
128 0x02c20c9ffc9d2b63
129 0x078419bedd3f5de6
```

```
130 0x0c0833fdbe5bf66c
131 0x1810657a58b62af8
132 0x20308af4b1485f50
133 0x607197694290f1a0
134 0xa0f2acd3852122e0
135 0x61f5d9260e634761
136 0xa2fa724c18e7c9e2
137 0x67e4a69831ea5a65
138 0xacc9cfb043f4feea
139 0x79925de087cd3375
140 0x8234fb410b9f65ca
141 0x06793483173b8e15
142 0x0ee369872a56922a
143 0x1fc7938f74a9a674
144 0x2c8ea59fcd72cac8
145 0x791dcbbe9ec55f10
146 0x832a55fd398ff120
147 0x0554eb7b531a2361
148 0x0bb914f7a63445e2
149 0x1463296e684cce64
150 0x38c752dcf09d52e8
151 0x418fe739c13fe770
152 0xc21e0c72825a09c0
153 0xc62c18e504b41a01
154 0xce58314b0d4c3e03
155 0xdea062971e9c7207
156 0xef4087af393ca60f
157 0xbd818ddf525dca1f
158 0x4a029b3fa4be5e3f
159 0xd605b47e6d58f25e
160 0xfe0ae8fcfeb126bd
161 0x8e151179d9434bdb
162 0x1e3b22f2b287dc37
163 0x2e674765450a744e
164 0x7ecfcccb8e14ac9c
165 0x8f9e5916182dd5b8
166 0x1c2cf22c307e6ed1
167 0x2859265840d89322
```

Appendix C

Sample MD6 computations

This chapter provides three sample MD6 computations to illustrate its operation. These examples are merely small illustrative examples; larger examples responsive to the NIST requirements are provided separately.

The first example has a three-character message input "`abc`" and computes a 256-bit hash, using only a single compression function call. All intermediate variables are printed.

The second example has the 600-character message input (in hexadecimal) "`1122334455667711223 3...`" and uses three compression function calls (two leaves and the root). It has a key input "`abcde12345`", and computes a hash value of $d = 224$ bits. All intermediate variables for the three compression function calls are printed.

In order to keep these examples from being overly long, the number of rounds in each of these two examples is set to $r = 5$.

The third example illustrates the sequential ($L = 0$) mode of operation on an 800-byte input file. Three compression function function calls are required. Only the input and output for each compression function call are printed.

These examples are produced using `md6sum` (see Section 8.4).

C.1 First example

The first example has a three-character message input "`abc`" and computes a 256-bit hash, using only a single compression function call with five rounds. All intermediate variables are printed.

```
> md6sum -r5 -I -Mabc
-r5
-- Mon Aug 04 18:28:03 2008
-- d =    256 (digest length in bits)
-- L =     64 (number of parallel passes)
-- r =      5 (number of rounds)
-- K = '' (key)
```

```
-- k =      0 (key length in bytes)

MD6 compression function computation (level 1, index 0):
Input (89 words):
A[   0] = 7311c2812425cfa0 Q[0]
A[   1] = 6432286434aac8e7 Q[1]
A[   2] = b60450e9ef68b7c1 Q[2]
A[   3] = e8fb23908d9f06f1 Q[3]
A[   4] = dd2e76cba691e5bf Q[4]
A[   5] = 0cd0d63b2c30bc41 Q[5]
A[   6] = 1f8ccf6823058f8a Q[6]
A[   7] = 54e5ed5b88e3775d Q[7]
A[   8] = 4ad12aae0a6d6031 Q[8]
A[   9] = 3e7f16bb88222e0d Q[9]
A[  10] = 8af8671d3fb50c2c Q[10]
A[  11] = 995ad1178bd25c31 Q[11]
A[  12] = c878c1dd04c4b633 Q[12]
A[  13] = 3b72066c7a1552ac Q[13]
A[  14] = 0d6f3522631effcb Q[14]
A[  15] = 0000000000000000 key K[0]
A[  16] = 0000000000000000 key K[1]
A[  17] = 0000000000000000 key K[2]
A[  18] = 0000000000000000 key K[3]
A[  19] = 0000000000000000 key K[4]
A[  20] = 0000000000000000 key K[5]
A[  21] = 0000000000000000 key K[6]
A[  22] = 0000000000000000 key K[7]
A[  23] = 0100000000000000 nodeID U = (ell,i) = (1,0)
A[  24] = 00054010fe800100 control word V = (r,L,z,p,keylen,d) = (5,64,1,4072,0,256)
A[  25] = 6162630000000000 data B[ 0] input message word    0
A[  26] = 0000000000000000 data B[ 1] padding
A[  27] = 0000000000000000 data B[ 2] padding
A[  28] = 0000000000000000 data B[ 3] padding
A[  29] = 0000000000000000 data B[ 4] padding
A[  30] = 0000000000000000 data B[ 5] padding
A[  31] = 0000000000000000 data B[ 6] padding
A[  32] = 0000000000000000 data B[ 7] padding
A[  33] = 0000000000000000 data B[ 8] padding
A[  34] = 0000000000000000 data B[ 9] padding
A[  35] = 0000000000000000 data B[10] padding
A[  36] = 0000000000000000 data B[11] padding
A[  37] = 0000000000000000 data B[12] padding
A[  38] = 0000000000000000 data B[13] padding
A[  39] = 0000000000000000 data B[14] padding
A[  40] = 0000000000000000 data B[15] padding
A[  41] = 0000000000000000 data B[16] padding
```

```
A[  42] = 0000000000000000 data B[17] padding
A[  43] = 0000000000000000 data B[18] padding
A[  44] = 0000000000000000 data B[19] padding
A[  45] = 0000000000000000 data B[20] padding
A[  46] = 0000000000000000 data B[21] padding
A[  47] = 0000000000000000 data B[22] padding
A[  48] = 0000000000000000 data B[23] padding
A[  49] = 0000000000000000 data B[24] padding
A[  50] = 0000000000000000 data B[25] padding
A[  51] = 0000000000000000 data B[26] padding
A[  52] = 0000000000000000 data B[27] padding
A[  53] = 0000000000000000 data B[28] padding
A[  54] = 0000000000000000 data B[29] padding
A[  55] = 0000000000000000 data B[30] padding
A[  56] = 0000000000000000 data B[31] padding
A[  57] = 0000000000000000 data B[32] padding
A[  58] = 0000000000000000 data B[33] padding
A[  59] = 0000000000000000 data B[34] padding
A[  60] = 0000000000000000 data B[35] padding
A[  61] = 0000000000000000 data B[36] padding
A[  62] = 0000000000000000 data B[37] padding
A[  63] = 0000000000000000 data B[38] padding
A[  64] = 0000000000000000 data B[39] padding
A[  65] = 0000000000000000 data B[40] padding
A[  66] = 0000000000000000 data B[41] padding
A[  67] = 0000000000000000 data B[42] padding
A[  68] = 0000000000000000 data B[43] padding
A[  69] = 0000000000000000 data B[44] padding
A[  70] = 0000000000000000 data B[45] padding
A[  71] = 0000000000000000 data B[46] padding
A[  72] = 0000000000000000 data B[47] padding
A[  73] = 0000000000000000 data B[48] padding
A[  74] = 0000000000000000 data B[49] padding
A[  75] = 0000000000000000 data B[50] padding
A[  76] = 0000000000000000 data B[51] padding
A[  77] = 0000000000000000 data B[52] padding
A[  78] = 0000000000000000 data B[53] padding
A[  79] = 0000000000000000 data B[54] padding
A[  80] = 0000000000000000 data B[55] padding
A[  81] = 0000000000000000 data B[56] padding
A[  82] = 0000000000000000 data B[57] padding
A[  83] = 0000000000000000 data B[58] padding
A[  84] = 0000000000000000 data B[59] padding
A[  85] = 0000000000000000 data B[60] padding
A[  86] = 0000000000000000 data B[61] padding
A[  87] = 0000000000000000 data B[62] padding
```

```
A[  88] = 0000000000000000 data B[63] padding
Intermediate values:
A[  89] = 027431e67f2b19cf
A[  90] = 0d990f6680e90d20
A[  91] = f27bc123aa282635
A[  92] = f90ca91b7fd9c62c
A[  93] = 85139f55bd354f15
A[  94] = eb6b874532011a19
A[  95] = 7b04461ba005d2fc
A[  96] = c7db19c96ca9abc7
A[  97] = b723400f04c813c4
A[  98] = c22c98f63ef66335
A[  99] = 42a2cbb64372fc40
A[ 100] = e52aeb1d587b9012
A[ 101] = 9ea7a2d571275633
A[ 102] = 7e99d0316f65addd
A[ 103] = 72f2b2f2fd1fe6ec
A[ 104] = 478df0ec797df153
A[ 105] = 3b9efe3b34add3eb
A[ 106] = f0155b54e33fa5cc
A[ 107] = b3b80e2309548fa4
A[ 108] = b5ef06df65e727d7
A[ 109] = ef08a1b814d205a0
A[ 110] = 367b2caf36cc81c6
A[ 111] = 343a0cf5b903d13e
A[ 112] = b4f9c1e7889e619e
A[ 113] = da463bc1b64240ad
A[ 114] = 10401204b0e3df85
A[ 115] = 4877a679f7db2705
A[ 116] = e2ff7c19283b650d
A[ 117] = 7e20b510048c8b81
A[ 118] = 2ec6248f95796fcd
A[ 119] = 0c87c7f9e1056f74
A[ 120] = 5e20250caa5b4a43
A[ 121] = 6e44865c042e3829
A[ 122] = 9529fbc6155a6a6d
A[ 123] = c44d6a63399d5e4f
A[ 124] = 04ead78d74346144
A[ 125] = 259b97c077a30362
A[ 126] = d185200a80400541
A[ 127] = b9a8bba23413f53c
A[ 128] = a439ca3d5839a512
A[ 129] = d2be51693c027782
A[ 130] = 94c0710d616da4c0
A[ 131] = 55e60934532be3b6
A[ 132] = a6e5b044f10f495d
```

```
A[ 133] = c2a4ba0dd30863e0
A[ 134] = abfa7c9a10170f52
A[ 135] = c55ba748fdfdcaaa
A[ 136] = 9e0f8e2fbf4645e7
A[ 137] = 21b0d68b36a65ab3
A[ 138] = 24e5578b36da9478
A[ 139] = 58446db406441646
A[ 140] = 1be8e6525fc16819
A[ 141] = e84464fb02c603b9
A[ 142] = a14656016a6def39
A[ 143] = 9b2b76febbe7de1f
A[ 144] = 79eda3eb98f56b99
A[ 145] = 0d4ce347389fbe8d
A[ 146] = 0e51deba9751e9ac
A[ 147] = a09984f7d2ed4785
A[ 148] = b3d375606156d954
A[ 149] = 8f7d6fb5316a6189
A[ 150] = 1b87a1d5504f7fc3
A[ 151] = e3d53e19846c0868
A[ 152] = 9dfbc0507d476a7d
Output (16 words of chaining values):
A[ 153] = 2d1abe0601b2e6b0 output chaining value C[0]
A[ 154] = 61d59fd2b7310353 output chaining value C[1]
A[ 155] = ea7da28dec708ec7 output chaining value C[2]
A[ 156] = a63a99a574e40155 output chaining value C[3]
A[ 157] = 290b4fabe80104c4 output chaining value C[4]
A[ 158] = 8c6a3503cf881a99 output chaining value C[5]
A[ 159] = e370e23d1b700cc5 output chaining value C[6]
A[ 160] = 4492e78e3fe42f13 output chaining value C[7]
A[ 161] = df6c91b7eaf3f088 output chaining value C[8]
A[ 162] = aab3e19a8f63b80a output chaining value C[9]
A[ 163] = d987bdcbda2e934f output chaining value C[10]
A[ 164] = aeae805de12b0d24 output chaining value C[11]
A[ 165] = 8854c14dc284f840 output chaining value C[12]
A[ 166] = ed71ad7ba542855c output chaining value C[13]
A[ 167] = e189633e48c797a5 output chaining value C[14]
A[ 168] = 5121a746be48cec8 output chaining value C[15]

8854c14dc284f840ed71ad7ba542855ce189633e48c797a55121a746be48cec8 -Mabc
```

The final hash value is `0x8854c14d...cec8` .

C.2 Second example

The second example has the 600-character message input (in hexadecimal) "`1122334455667711223 3...`" and uses three compression function calls (two

leaves and the root). It has a key input "abcde12345", and computes a hash value of $d = 224$ bits using five rounds. All intermediate variables for the three compression function calls are printed.

```
> md6sum -d224 -r5 -Kabcde12345 -I -B600
-d224
-r5
-Kabcde12345
-- Mon Aug 04 21:11:05 2008
-- d =    224 (digest length in bits)
-- L =     64 (number of parallel passes)
-- r =      5 (number of rounds)
-- K = 'abcde12345' (key)
-- k =     10 (key length in bytes)

MD6 compression function computation (level 1, index 0):
Input (89 words):
A[   0] = 7311c2812425cfa0 Q[0]
A[   1] = 6432286434aac8e7 Q[1]
A[   2] = b60450e9ef68b7c1 Q[2]
A[   3] = e8fb23908d9f06f1 Q[3]
A[   4] = dd2e76cba691e5bf Q[4]
A[   5] = 0cd0d63b2c30bc41 Q[5]
A[   6] = 1f8ccf6823058f8a Q[6]
A[   7] = 54e5ed5b88e3775d Q[7]
A[   8] = 4ad12aae0a6d6031 Q[8]
A[   9] = 3e7f16bb88222e0d Q[9]
A[  10] = 8af8671d3fb50c2c Q[10]
A[  11] = 995ad1178bd25c31 Q[11]
A[  12] = c878c1dd04c4b633 Q[12]
A[  13] = 3b72066c7a1552ac Q[13]
A[  14] = 0d6f3522631effcb Q[14]
A[  15] = 6162636465313233 key K[0]
A[  16] = 3435000000000000 key K[1]
A[  17] = 0000000000000000 key K[2]
A[  18] = 0000000000000000 key K[3]
A[  19] = 0000000000000000 key K[4]
A[  20] = 0000000000000000 key K[5]
A[  21] = 0000000000000000 key K[6]
A[  22] = 0000000000000000 key K[7]
A[  23] = 0100000000000000 nodeID U = (ell,i) = (1,0)
A[  24] = 000540000000a0e0 control word V = (r,L,z,p,keylen,d) = (5,64,0,0,10,224)
A[  25] = 1122334455667711 data B[ 0] input message word    0
A[  26] = 2233445566771122 data B[ 1] input message word    1
A[  27] = 3344556677112233 data B[ 2] input message word    2
A[  28] = 4455667711223344 data B[ 3] input message word    3
```

```
A[  29] = 5566771122334455 data B[ 4] input message word    4
A[  30] = 6677112233445566 data B[ 5] input message word    5
A[  31] = 7711223344556677 data B[ 6] input message word    6
A[  32] = 1122334455667711 data B[ 7] input message word    7
A[  33] = 2233445566771122 data B[ 8] input message word    8
A[  34] = 3344556677112233 data B[ 9] input message word    9
A[  35] = 4455667711223344 data B[10] input message word   10
A[  36] = 5566771122334455 data B[11] input message word   11
A[  37] = 6677112233445566 data B[12] input message word   12
A[  38] = 7711223344556677 data B[13] input message word   13
A[  39] = 1122334455667711 data B[14] input message word   14
A[  40] = 2233445566771122 data B[15] input message word   15
A[  41] = 3344556677112233 data B[16] input message word   16
A[  42] = 4455667711223344 data B[17] input message word   17
A[  43] = 5566771122334455 data B[18] input message word   18
A[  44] = 6677112233445566 data B[19] input message word   19
A[  45] = 7711223344556677 data B[20] input message word   20
A[  46] = 1122334455667711 data B[21] input message word   21
A[  47] = 2233445566771122 data B[22] input message word   22
A[  48] = 3344556677112233 data B[23] input message word   23
A[  49] = 4455667711223344 data B[24] input message word   24
A[  50] = 5566771122334455 data B[25] input message word   25
A[  51] = 6677112233445566 data B[26] input message word   26
A[  52] = 7711223344556677 data B[27] input message word   27
A[  53] = 1122334455667711 data B[28] input message word   28
A[  54] = 2233445566771122 data B[29] input message word   29
A[  55] = 3344556677112233 data B[30] input message word   30
A[  56] = 4455667711223344 data B[31] input message word   31
A[  57] = 5566771122334455 data B[32] input message word   32
A[  58] = 6677112233445566 data B[33] input message word   33
A[  59] = 7711223344556677 data B[34] input message word   34
A[  60] = 1122334455667711 data B[35] input message word   35
A[  61] = 2233445566771122 data B[36] input message word   36
A[  62] = 3344556677112233 data B[37] input message word   37
A[  63] = 4455667711223344 data B[38] input message word   38
A[  64] = 5566771122334455 data B[39] input message word   39
A[  65] = 6677112233445566 data B[40] input message word   40
A[  66] = 7711223344556677 data B[41] input message word   41
A[  67] = 1122334455667711 data B[42] input message word   42
A[  68] = 2233445566771122 data B[43] input message word   43
A[  69] = 3344556677112233 data B[44] input message word   44
A[  70] = 4455667711223344 data B[45] input message word   45
A[  71] = 5566771122334455 data B[46] input message word   46
A[  72] = 6677112233445566 data B[47] input message word   47
A[  73] = 7711223344556677 data B[48] input message word   48
A[  74] = 1122334455667711 data B[49] input message word   49
```

```
A[  75] = 2233445566771122 data B[50] input message word   50
A[  76] = 3344556677112233 data B[51] input message word   51
A[  77] = 4455667711223344 data B[52] input message word   52
A[  78] = 5566771122334455 data B[53] input message word   53
A[  79] = 6677112233445566 data B[54] input message word   54
A[  80] = 7711223344556677 data B[55] input message word   55
A[  81] = 1122334455667711 data B[56] input message word   56
A[  82] = 2233445566771122 data B[57] input message word   57
A[  83] = 3344556677112233 data B[58] input message word   58
A[  84] = 4455667711223344 data B[59] input message word   59
A[  85] = 5566771122334455 data B[60] input message word   60
A[  86] = 6677112233445566 data B[61] input message word   61
A[  87] = 7711223344556677 data B[62] input message word   62
A[  88] = 1122334455667711 data B[63] input message word   63
Intermediate values:
A[  89] = 023bc36dbadd897c
A[  90] = d2927a221b06c047
A[  91] = c43ba671cd483453
A[  92] = 71fff4ac51400ece
A[  93] = 0982eb1487ed94a0
A[  94] = b0db96aa5be6e25d
A[  95] = c79104d837fa7829
A[  96] = 74f191fac63854c7
A[  97] = 555836c2482b7073
A[  98] = 96d1cfa2960a3635
A[  99] = 91f93e1e2305defb
A[ 100] = a0d6fb3db7872b00
A[ 101] = 09b3e3efa9f46322
A[ 102] = 4f974344be9283c2
A[ 103] = af945d6202a52b8a
A[ 104] = 534ae6f0cc48c152
A[ 105] = 9a1312a7cb95e823
A[ 106] = 202d5bc69180f643
A[ 107] = 3314055f3e8bd053
A[ 108] = d5f43130bf23ceea
A[ 109] = 6626816602008668
A[ 110] = 76aeea2aca773f6e
A[ 111] = 5e85d8b579246c54
A[ 112] = 2c15d20ff9b8f8c7
A[ 113] = b63139c71d240633
A[ 114] = f34ac4fd12d5ff52
A[ 115] = e6ddcdd1c15d615e
A[ 116] = 4ff6831138609787
A[ 117] = 2d3f1ff4b46ae4d6
A[ 118] = 0db9939d0538d1e5
A[ 119] = 021d0b12738f57f8
```

```
A[ 120] = 225b9b9043fc1478
A[ 121] = 2b44b144083caec0
A[ 122] = ec7dd9746e59215a
A[ 123] = 1b16f1dbf1ee2ca7
A[ 124] = aff86e674d80dfc2
A[ 125] = 500e4a1f3414f2e9
A[ 126] = 7526f159466c0c08
A[ 127] = 21872b0c3ff8f9e5
A[ 128] = 9926c4c3dc26a41a
A[ 129] = 1605d8571a384095
A[ 130] = 36108731ac38dbbe
A[ 131] = 855e83cd250610a9
A[ 132] = e50d76ec2ffcd792
A[ 133] = 3c25e0d2a66156b0
A[ 134] = 5c0a179096bbd764
A[ 135] = 69fbb852261cf3dd
A[ 136] = 464e53ab124324fd
A[ 137] = b82c23d382fe93b0
A[ 138] = f0b2c7468873e088
A[ 139] = 1617ffed131a1888
A[ 140] = 4c96e83298992d59
A[ 141] = 4572047f487d2c7f
A[ 142] = 8dd3884af1d1fa71
A[ 143] = 4e9c5579e3a882cf
A[ 144] = e436a7a0b969237b
A[ 145] = 4c40356771ccc066
A[ 146] = c39d8cb040a1bf0f
A[ 147] = 6d10e4a2236c9cc8
A[ 148] = aee19ed455d9c494
A[ 149] = 9beb541edf2cc926
A[ 150] = 57cef68e40f3a1fe
A[ 151] = da808363958f463d
A[ 152] = dea095fbc38c581d
Output (16 words of chaining values):
A[ 153] = e86a6f805fb810ca output chaining value C[0]
A[ 154] = 991de071299831a9 output chaining value C[1]
A[ 155] = c59517fb7f5c5e74 output chaining value C[2]
A[ 156] = 0e2b5f69481c68e6 output chaining value C[3]
A[ 157] = 8ddb33a8b069b4c2 output chaining value C[4]
A[ 158] = 558b3513a0046dbd output chaining value C[5]
A[ 159] = e1dfb6726949ab7e output chaining value C[6]
A[ 160] = f48bae515e89ee94 output chaining value C[7]
A[ 161] = d31d1f87d97da302 output chaining value C[8]
A[ 162] = 5d349e9b0d69b270 output chaining value C[9]
A[ 163] = b409d2ee2c3e5577 output chaining value C[10]
A[ 164] = 997621d403cd954e output chaining value C[11]
```

```
A[ 165] = 7a353e0ef29490a3 output chaining value C[12]
A[ 166] = 716d1239dfff51dc output chaining value C[13]
A[ 167] = 59744be898cf7c0a output chaining value C[14]
A[ 168] = 07951a90e19da429 output chaining value C[15]

MD6 compression function computation (level 1, index 1):
Input (89 words):
A[   0] = 7311c2812425cfa0 Q[0]
A[   1] = 6432286434aac8e7 Q[1]
A[   2] = b60450e9ef68b7c1 Q[2]
A[   3] = e8fb23908d9f06f1 Q[3]
A[   4] = dd2e76cba691e5bf Q[4]
A[   5] = 0cd0d63b2c30bc41 Q[5]
A[   6] = 1f8ccf6823058f8a Q[6]
A[   7] = 54e5ed5b88e3775d Q[7]
A[   8] = 4ad12aae0a6d6031 Q[8]
A[   9] = 3e7f16bb88222e0d Q[9]
A[  10] = 8af8671d3fb50c2c Q[10]
A[  11] = 995ad1178bd25c31 Q[11]
A[  12] = c878c1dd04c4b633 Q[12]
A[  13] = 3b72066c7a1552ac Q[13]
A[  14] = 0d6f3522631effcb Q[14]
A[  15] = 6162636465313233 key K[0]
A[  16] = 3435000000000000 key K[1]
A[  17] = 0000000000000000 key K[2]
A[  18] = 0000000000000000 key K[3]
A[  19] = 0000000000000000 key K[4]
A[  20] = 0000000000000000 key K[5]
A[  21] = 0000000000000000 key K[6]
A[  22] = 0000000000000000 key K[7]
A[  23] = 0100000000000001 nodeID U = (ell,i) = (1,1)
A[  24] = 00054000d400a0e0 control word V = (r,L,z,p,keylen,d) = (5,64,0,3392,10,224)
A[  25] = 2233445566771122 data B[ 0] input message word   64
A[  26] = 3344556677112233 data B[ 1] input message word   65
A[  27] = 4455667711223344 data B[ 2] input message word   66
A[  28] = 5566771122334455 data B[ 3] input message word   67
A[  29] = 6677112233445566 data B[ 4] input message word   68
A[  30] = 7711223344556677 data B[ 5] input message word   69
A[  31] = 1122334455667711 data B[ 6] input message word   70
A[  32] = 2233445566771122 data B[ 7] input message word   71
A[  33] = 3344556677112233 data B[ 8] input message word   72
A[  34] = 4455667711223344 data B[ 9] input message word   73
A[  35] = 5566771122334455 data B[10] input message word   74
A[  36] = 0000000000000000 data B[11] padding
A[  37] = 0000000000000000 data B[12] padding
A[  38] = 0000000000000000 data B[13] padding
```

```
A[  39] = 0000000000000000 data B[14] padding
A[  40] = 0000000000000000 data B[15] padding
A[  41] = 0000000000000000 data B[16] padding
A[  42] = 0000000000000000 data B[17] padding
A[  43] = 0000000000000000 data B[18] padding
A[  44] = 0000000000000000 data B[19] padding
A[  45] = 0000000000000000 data B[20] padding
A[  46] = 0000000000000000 data B[21] padding
A[  47] = 0000000000000000 data B[22] padding
A[  48] = 0000000000000000 data B[23] padding
A[  49] = 0000000000000000 data B[24] padding
A[  50] = 0000000000000000 data B[25] padding
A[  51] = 0000000000000000 data B[26] padding
A[  52] = 0000000000000000 data B[27] padding
A[  53] = 0000000000000000 data B[28] padding
A[  54] = 0000000000000000 data B[29] padding
A[  55] = 0000000000000000 data B[30] padding
A[  56] = 0000000000000000 data B[31] padding
A[  57] = 0000000000000000 data B[32] padding
A[  58] = 0000000000000000 data B[33] padding
A[  59] = 0000000000000000 data B[34] padding
A[  60] = 0000000000000000 data B[35] padding
A[  61] = 0000000000000000 data B[36] padding
A[  62] = 0000000000000000 data B[37] padding
A[  63] = 0000000000000000 data B[38] padding
A[  64] = 0000000000000000 data B[39] padding
A[  65] = 0000000000000000 data B[40] padding
A[  66] = 0000000000000000 data B[41] padding
A[  67] = 0000000000000000 data B[42] padding
A[  68] = 0000000000000000 data B[43] padding
A[  69] = 0000000000000000 data B[44] padding
A[  70] = 0000000000000000 data B[45] padding
A[  71] = 0000000000000000 data B[46] padding
A[  72] = 0000000000000000 data B[47] padding
A[  73] = 0000000000000000 data B[48] padding
A[  74] = 0000000000000000 data B[49] padding
A[  75] = 0000000000000000 data B[50] padding
A[  76] = 0000000000000000 data B[51] padding
A[  77] = 0000000000000000 data B[52] padding
A[  78] = 0000000000000000 data B[53] padding
A[  79] = 0000000000000000 data B[54] padding
A[  80] = 0000000000000000 data B[55] padding
A[  81] = 0000000000000000 data B[56] padding
A[  82] = 0000000000000000 data B[57] padding
A[  83] = 0000000000000000 data B[58] padding
A[  84] = 0000000000000000 data B[59] padding
```

```
A[  85] = 0000000000000000 data B[60] padding
A[  86] = 0000000000000000 data B[61] padding
A[  87] = 0000000000000000 data B[62] padding
A[  88] = 0000000000000000 data B[63] padding
Intermediate values:
A[  89] = 027431e67f2b19cf
A[  90] = 0d990f6680e90d20
A[  91] = f27bc123aa282635
A[  92] = f90ca91b7fd9c62c
A[  93] = 85139f55bd354f15
A[  94] = eb6b874532011a19
A[  95] = 7b04461ba005d2fc
A[  96] = c7db19c96ca9abc7
A[  97] = b723400f04c813c4
A[  98] = c22c98f63ef66335
A[  99] = 42a2cbb64372fc40
A[ 100] = e52aeb1d587b9012
A[ 101] = 9ea7a2d571275633
A[ 102] = 7e99d0316f65addd
A[ 103] = 72f2b2f2fd1fe6ec
A[ 104] = ee030108c4c8d073
A[ 105] = cfccf37b34add3eb
A[ 106] = f0155b54e33fa5cc
A[ 107] = b3b80e2309548fa4
A[ 108] = b5ef06df65e727d7
A[ 109] = ef08a1b814d205a0
A[ 110] = 367b2caf36cc81c6
A[ 111] = 343a0cf5b903d13e
A[ 112] = b4f9c1e7889ee19f
A[ 113] = da463fdb2c80b3cf
A[ 114] = 5e57342f3b497579
A[ 115] = ec2a9bbbac242772
A[ 116] = f1ece4e7901b8a58
A[ 117] = 46a38ec4458509bc
A[ 118] = 9f023b8106000b81
A[ 119] = bee8b8b568415c9a
A[ 120] = 0941caa98d3a8735
A[ 121] = 1f0d393d159e88a7
A[ 122] = fd918160425fe96e
A[ 123] = 7dc1367a2e734e1a
A[ 124] = 7f151f7a5acba9c0
A[ 125] = 9c97a06aeb6620c2
A[ 126] = f74ca0e2c4400541
A[ 127] = b9a8bba23413f53c
A[ 128] = a439ca3d5839a512
A[ 129] = d2be51693c22f741
```

```
A[ 130] = 94c0657db84579bb
A[ 131] = a0261aaca9731c5f
A[ 132] = 4c9411c30ffbf9fa
A[ 133] = 4dcf01f6386f9b7e
A[ 134] = dbaaf2cbe4f486aa
A[ 135] = 7d5a36c9454ec7e4
A[ 136] = f8634c771236a7dd
A[ 137] = 78c3c019f44632b1
A[ 138] = 6cea1a934cc7ea1e
A[ 139] = fcaa77298206c0a0
A[ 140] = 465b12166eccf1ee
A[ 141] = c07d7163bcd05dfe
A[ 142] = 841dd2066df5d2a1
A[ 143] = db122831fc72f4b5
A[ 144] = 08f24baf1c9d6bd8
A[ 145] = 5eff0d4698d0fc06
A[ 146] = 2e7d9462fae63e2c
A[ 147] = bf86d1573e7fcb05
A[ 148] = 81b9602f09b1b144
A[ 149] = f7d0e892806029ac
A[ 150] = ee671dbe2f9e706c
A[ 151] = 03d9c6acabf402aa
A[ 152] = 2017d423651b402b
Output (16 words of chaining values):
A[ 153] = 34e06cf8e7e380b8 output chaining value C[0]
A[ 154] = f8736f4357f99cb8 output chaining value C[1]
A[ 155] = a3e1187da8fbd4e8 output chaining value C[2]
A[ 156] = 6c11da3b93aca37a output chaining value C[3]
A[ 157] = 5fdb88a98301b016 output chaining value C[4]
A[ 158] = 5d2a34ccc621594d output chaining value C[5]
A[ 159] = d10521d7588ce414 output chaining value C[6]
A[ 160] = 5040286fe773a8c0 output chaining value C[7]
A[ 161] = fe030f559c8a0f0b output chaining value C[8]
A[ 162] = ca289a3c963dd24b output chaining value C[9]
A[ 163] = acdccf24c7a70e53 output chaining value C[10]
A[ 164] = 1f451b9a0209f583 output chaining value C[11]
A[ 165] = da56f65e3205064d output chaining value C[12]
A[ 166] = a00e879eae6d8241 output chaining value C[13]
A[ 167] = 2a2a15bc29dc56a4 output chaining value C[14]
A[ 168] = 5d8e677905657f39 output chaining value C[15]

MD6 compression function computation (level 2, index 0):
Input (89 words):
A[   0] = 7311c2812425cfa0 Q[0]
A[   1] = 6432286434aac8e7 Q[1]
A[   2] = b60450e9ef68b7c1 Q[2]
```

```
A[   3] = e8fb23908d9f06f1 Q[3]
A[   4] = dd2e76cba691e5bf Q[4]
A[   5] = 0cd0d63b2c30bc41 Q[5]
A[   6] = 1f8ccf6823058f8a Q[6]
A[   7] = 54e5ed5b88e3775d Q[7]
A[   8] = 4ad12aae0a6d6031 Q[8]
A[   9] = 3e7f16bb88222e0d Q[9]
A[  10] = 8af8671d3fb50c2c Q[10]
A[  11] = 995ad1178bd25c31 Q[11]
A[  12] = c878c1dd04c4b633 Q[12]
A[  13] = 3b72066c7a1552ac Q[13]
A[  14] = 0d6f3522631effcb Q[14]
A[  15] = 6162636465313233 key K[0]
A[  16] = 3435000000000000 key K[1]
A[  17] = 0000000000000000 key K[2]
A[  18] = 0000000000000000 key K[3]
A[  19] = 0000000000000000 key K[4]
A[  20] = 0000000000000000 key K[5]
A[  21] = 0000000000000000 key K[6]
A[  22] = 0000000000000000 key K[7]
A[  23] = 0200000000000000 nodeID U = (ell,i) = (2,0)
A[  24] = 000540108000a0e0 control word V = (r,L,z,p,keylen,d) = (5,64,1,2048,10,224)
A[  25] = e86a6f805fb810ca data B[ 0] chaining from (1,0)
A[  26] = 991de071299831a9 data B[ 1] chaining from (1,0)
A[  27] = c59517fb7f5c5e74 data B[ 2] chaining from (1,0)
A[  28] = 0e2b5f69481c68e6 data B[ 3] chaining from (1,0)
A[  29] = 8ddb33a8b069b4c2 data B[ 4] chaining from (1,0)
A[  30] = 558b3513a0046dbd data B[ 5] chaining from (1,0)
A[  31] = e1dfb6726949ab7e data B[ 6] chaining from (1,0)
A[  32] = f48bae515e89ee94 data B[ 7] chaining from (1,0)
A[  33] = d31d1f87d97da302 data B[ 8] chaining from (1,0)
A[  34] = 5d349e9b0d69b270 data B[ 9] chaining from (1,0)
A[  35] = b409d2ee2c3e5577 data B[10] chaining from (1,0)
A[  36] = 997621d403cd954e data B[11] chaining from (1,0)
A[  37] = 7a353e0ef29490a3 data B[12] chaining from (1,0)
A[  38] = 716d1239dfff51dc data B[13] chaining from (1,0)
A[  39] = 59744be898cf7c0a data B[14] chaining from (1,0)
A[  40] = 07951a90e19da429 data B[15] chaining from (1,0)
A[  41] = 34e06cf8e7e380b8 data B[16] chaining from (1,1)
A[  42] = f8736f4357f99cb8 data B[17] chaining from (1,1)
A[  43] = a3e1187da8fbd4e8 data B[18] chaining from (1,1)
A[  44] = 6c11da3b93aca37a data B[19] chaining from (1,1)
A[  45] = 5fdb88a98301b016 data B[20] chaining from (1,1)
A[  46] = 5d2a34ccc621594d data B[21] chaining from (1,1)
A[  47] = d10521d7588ce414 data B[22] chaining from (1,1)
A[  48] = 5040286fe773a8c0 data B[23] chaining from (1,1)
```

```
A[  49] = fe030f559c8a0f0b data B[24] chaining from (1,1)
A[  50] = ca289a3c963dd24b data B[25] chaining from (1,1)
A[  51] = acdccf24c7a70e53 data B[26] chaining from (1,1)
A[  52] = 1f451b9a0209f583 data B[27] chaining from (1,1)
A[  53] = da56f65e3205064d data B[28] chaining from (1,1)
A[  54] = a00e879eae6d8241 data B[29] chaining from (1,1)
A[  55] = 2a2a15bc29dc56a4 data B[30] chaining from (1,1)
A[  56] = 5d8e677905657f39 data B[31] chaining from (1,1)
A[  57] = 0000000000000000 data B[32] padding
A[  58] = 0000000000000000 data B[33] padding
A[  59] = 0000000000000000 data B[34] padding
A[  60] = 0000000000000000 data B[35] padding
A[  61] = 0000000000000000 data B[36] padding
A[  62] = 0000000000000000 data B[37] padding
A[  63] = 0000000000000000 data B[38] padding
A[  64] = 0000000000000000 data B[39] padding
A[  65] = 0000000000000000 data B[40] padding
A[  66] = 0000000000000000 data B[41] padding
A[  67] = 0000000000000000 data B[42] padding
A[  68] = 0000000000000000 data B[43] padding
A[  69] = 0000000000000000 data B[44] padding
A[  70] = 0000000000000000 data B[45] padding
A[  71] = 0000000000000000 data B[46] padding
A[  72] = 0000000000000000 data B[47] padding
A[  73] = 0000000000000000 data B[48] padding
A[  74] = 0000000000000000 data B[49] padding
A[  75] = 0000000000000000 data B[50] padding
A[  76] = 0000000000000000 data B[51] padding
A[  77] = 0000000000000000 data B[52] padding
A[  78] = 0000000000000000 data B[53] padding
A[  79] = 0000000000000000 data B[54] padding
A[  80] = 0000000000000000 data B[55] padding
A[  81] = 0000000000000000 data B[56] padding
A[  82] = 0000000000000000 data B[57] padding
A[  83] = 0000000000000000 data B[58] padding
A[  84] = 0000000000000000 data B[59] padding
A[  85] = 0000000000000000 data B[60] padding
A[  86] = 0000000000000000 data B[61] padding
A[  87] = 0000000000000000 data B[62] padding
A[  88] = 0000000000000000 data B[63] padding
Intermediate values:
A[  89] = 027431e67f2b19cf
A[  90] = 0d990f6680e90d20
A[  91] = f27bc123aa282635
A[  92] = f90ca91b7fd9c62c
A[  93] = 85139f55bd354f15
```

```
A[  94] = eb6b874532011a19
A[  95] = 7b04461ba005d2fc
A[  96] = c7db19c96ca9abc7
A[  97] = b723400f04c813c4
A[  98] = c22c98f63ef66335
A[  99] = 42a2cbb64372fc40
A[ 100] = e52aeb1d587b9012
A[ 101] = 9ea7a2d571275633
A[ 102] = 7e99d0316f65addd
A[ 103] = 72f2b2f2fd1fe6ec
A[ 104] = ee030108c4c8d073
A[ 105] = cfccf37b34add3eb
A[ 106] = f0155b54e33fa5cc
A[ 107] = b3b80e2309548fa4
A[ 108] = b5ef06df65e727d7
A[ 109] = ef08a1b814d205a0
A[ 110] = 367b2caf36cc81c6
A[ 111] = 343a0cf5b903d13e
A[ 112] = b7ffc1e7889e619e
A[ 113] = da463bde6895e3cf
A[ 114] = bd6d76d5ecb9ced7
A[ 115] = e9211df17c1026cf
A[ 116] = b457148a6f18579b
A[ 117] = 3d0b7d88140c60ea
A[ 118] = 3a1618bc555cfd06
A[ 119] = af18e03b8f81137f
A[ 120] = b6d883f0886927fa
A[ 121] = e381b7c523751aee
A[ 122] = a67b075ce5084a43
A[ 123] = e21bdb4cbccfd550
A[ 124] = 605490a2b9633ba6
A[ 125] = 827ab3dc2455cc3e
A[ 126] = e844879b214868ab
A[ 127] = 69dee938c4137097
A[ 128] = ac40cdbd60be47e0
A[ 129] = f2ef6e974054c21d
A[ 130] = 7ca07eb5027eb3ba
A[ 131] = 44f906e854762107
A[ 132] = 1ee8fa9fe001f2fb
A[ 133] = 7e59e884e26c7334
A[ 134] = 2826f9847abcb858
A[ 135] = df781ee5037bfa6c
A[ 136] = 7ef847fbff16a0e2
A[ 137] = 7b9385aaadc629f9
A[ 138] = a31a329af6d51b66
A[ 139] = deca0d8d359124b2
```

```
A[ 140] = aa4dcd6abdd53809
A[ 141] = 54716fa013b20217
A[ 142] = 540769a3c74ee7e1
A[ 143] = 3db7182c921992a4
A[ 144] = 09f2a43ce7a2d5f9
A[ 145] = 7aae992259f2b683
A[ 146] = 9a7c3b013169a03e
A[ 147] = 5d3a735c1778b352
A[ 148] = 42d57de6de15e405
A[ 149] = 12f0c0a26450d81e
A[ 150] = aa66aa041120fc69
A[ 151] = 27c0ddcc71049201
A[ 152] = 4605822c05dc18b8
Output (16 words of chaining values):
A[ 153] = 51fe1122f4c17ec2 output chaining value C[0]
A[ 154] = a314cd812406314d output chaining value C[1]
A[ 155] = f7b08c0b30f095fe output chaining value C[2]
A[ 156] = 1ded1aee71933f09 output chaining value C[3]
A[ 157] = 2bb446cb238ed41f output chaining value C[4]
A[ 158] = 0f6460080325fe08 output chaining value C[5]
A[ 159] = 160e8b6947fcf632 output chaining value C[6]
A[ 160] = e283c3b4b88318cb output chaining value C[7]
A[ 161] = a00cab488aa9c072 output chaining value C[8]
A[ 162] = 9f2810c25189818d output chaining value C[9]
A[ 163] = 31f7f47f96cf8606 output chaining value C[10]
A[ 164] = 403f037430ec43f2 output chaining value C[11]
A[ 165] = edcbb8e5894cf059 output chaining value C[12]
A[ 166] = 8ad3288ed4bb5ac5 output chaining value C[13]
A[ 167] = df23eba0ac388a11 output chaining value C[14]
A[ 168] = b7ed2e3dd5ec5131 output chaining value C[15]

894cf0598ad3288ed4bb5ac5df23eba0ac388a11b7ed2e3dd5ec5131 -B600
```

The final hash value is `0x894cf059...5131` .

C.3 Third example

The third example illustrates the sequential ($L = 0$) mode of operation on an 800-byte input file. Three compression function function calls are required. Only the input and output for each compression function call are printed.

```
> md6sum -L0 -i -B800
-L0
-- Thu Aug 07 18:25:57 2008
-- d =    256 (digest length in bits)
```

```
-- L =      0 (number of parallel passes)
-- r =    104 (number of rounds)
-- K = '' (key)
-- k =      0 (key length in bytes)

MD6 compression function computation (level 1, index 0):
Input (89 words):
A[   0] = 7311c2812425cfa0 Q[0]
A[   1] = 6432286434aac8e7 Q[1]
A[   2] = b60450e9ef68b7c1 Q[2]
A[   3] = e8fb23908d9f06f1 Q[3]
A[   4] = dd2e76cba691e5bf Q[4]
A[   5] = 0cd0d63b2c30bc41 Q[5]
A[   6] = 1f8ccf6823058f8a Q[6]
A[   7] = 54e5ed5b88e3775d Q[7]
A[   8] = 4ad12aae0a6d6031 Q[8]
A[   9] = 3e7f16bb88222e0d Q[9]
A[  10] = 8af8671d3fb50c2c Q[10]
A[  11] = 995ad1178bd25c31 Q[11]
A[  12] = c878c1dd04c4b633 Q[12]
A[  13] = 3b72066c7a1552ac Q[13]
A[  14] = 0d6f3522631effcb Q[14]
A[  15] = 0000000000000000 key K[0]
A[  16] = 0000000000000000 key K[1]
A[  17] = 0000000000000000 key K[2]
A[  18] = 0000000000000000 key K[3]
A[  19] = 0000000000000000 key K[4]
A[  20] = 0000000000000000 key K[5]
A[  21] = 0000000000000000 key K[6]
A[  22] = 0000000000000000 key K[7]
A[  23] = 0100000000000000 nodeID U = (ell,i) = (1,0)
A[  24] = 0068000000000100 control word V = (r,L,z,p,keylen,d) = (104,0,0,0,0,256)
A[  25] = 0000000000000000 data B[ 0] IV
A[  26] = 0000000000000000 data B[ 1] IV
A[  27] = 0000000000000000 data B[ 2] IV
A[  28] = 0000000000000000 data B[ 3] IV
A[  29] = 0000000000000000 data B[ 4] IV
A[  30] = 0000000000000000 data B[ 5] IV
A[  31] = 0000000000000000 data B[ 6] IV
A[  32] = 0000000000000000 data B[ 7] IV
A[  33] = 0000000000000000 data B[ 8] IV
A[  34] = 0000000000000000 data B[ 9] IV
A[  35] = 0000000000000000 data B[10] IV
A[  36] = 0000000000000000 data B[11] IV
A[  37] = 0000000000000000 data B[12] IV
A[  38] = 0000000000000000 data B[13] IV
```

```
A[  39] = 0000000000000000 data B[14] IV
A[  40] = 0000000000000000 data B[15] IV
A[  41] = 1122334455667711 data B[16] input message word    0
A[  42] = 2233445566771122 data B[17] input message word    1
A[  43] = 3344556677112233 data B[18] input message word    2
A[  44] = 4455667711223344 data B[19] input message word    3
A[  45] = 5566771122334455 data B[20] input message word    4
A[  46] = 6677112233445566 data B[21] input message word    5
A[  47] = 7711223344556677 data B[22] input message word    6
A[  48] = 1122334455667711 data B[23] input message word    7
A[  49] = 2233445566771122 data B[24] input message word    8
A[  50] = 3344556677112233 data B[25] input message word    9
A[  51] = 4455667711223344 data B[26] input message word   10
A[  52] = 5566771122334455 data B[27] input message word   11
A[  53] = 6677112233445566 data B[28] input message word   12
A[  54] = 7711223344556677 data B[29] input message word   13
A[  55] = 1122334455667711 data B[30] input message word   14
A[  56] = 2233445566771122 data B[31] input message word   15
A[  57] = 3344556677112233 data B[32] input message word   16
A[  58] = 4455667711223344 data B[33] input message word   17
A[  59] = 5566771122334455 data B[34] input message word   18
A[  60] = 6677112233445566 data B[35] input message word   19
A[  61] = 7711223344556677 data B[36] input message word   20
A[  62] = 1122334455667711 data B[37] input message word   21
A[  63] = 2233445566771122 data B[38] input message word   22
A[  64] = 3344556677112233 data B[39] input message word   23
A[  65] = 4455667711223344 data B[40] input message word   24
A[  66] = 5566771122334455 data B[41] input message word   25
A[  67] = 6677112233445566 data B[42] input message word   26
A[  68] = 7711223344556677 data B[43] input message word   27
A[  69] = 1122334455667711 data B[44] input message word   28
A[  70] = 2233445566771122 data B[45] input message word   29
A[  71] = 3344556677112233 data B[46] input message word   30
A[  72] = 4455667711223344 data B[47] input message word   31
A[  73] = 5566771122334455 data B[48] input message word   32
A[  74] = 6677112233445566 data B[49] input message word   33
A[  75] = 7711223344556677 data B[50] input message word   34
A[  76] = 1122334455667711 data B[51] input message word   35
A[  77] = 2233445566771122 data B[52] input message word   36
A[  78] = 3344556677112233 data B[53] input message word   37
A[  79] = 4455667711223344 data B[54] input message word   38
A[  80] = 5566771122334455 data B[55] input message word   39
A[  81] = 6677112233445566 data B[56] input message word   40
A[  82] = 7711223344556677 data B[57] input message word   41
A[  83] = 1122334455667711 data B[58] input message word   42
A[  84] = 2233445566771122 data B[59] input message word   43
```

```
A[  85] = 3344556677112233 data B[60] input message word   44
A[  86] = 4455667711223344 data B[61] input message word   45
A[  87] = 5566771122334455 data B[62] input message word   46
A[  88] = 6677112233445566 data B[63] input message word   47
Intermediate values A[89..1736] omitted...
Output (16 words of chaining values):
A[1737] = d0e1686ab52f2642 output chaining value C[0]
A[1738] = 23ae78b2f1225d3d output chaining value C[1]
A[1739] = 1d3f60e3f6ebe9e6 output chaining value C[2]
A[1740] = 0e2a6dd7ca126f8f output chaining value C[3]
A[1741] = 9ced68e939d173e8 output chaining value C[4]
A[1742] = f1260e4e80af4bac output chaining value C[5]
A[1743] = dcd5ab34b0084bb0 output chaining value C[6]
A[1744] = 0d4486d2a95c0ea7 output chaining value C[7]
A[1745] = e0c1456140162bc6 output chaining value C[8]
A[1746] = 5c9b43a4afb76f91 output chaining value C[9]
A[1747] = 5e2e233293e7832b output chaining value C[10]
A[1748] = 8bb542c30087934c output chaining value C[11]
A[1749] = 801da4582d8c3b82 output chaining value C[12]
A[1750] = bdac086c24adb7e1 output chaining value C[13]
A[1751] = a4b6198b4f41a95b output chaining value C[14]
A[1752] = 5a362e4725f93b78 output chaining value C[15]

MD6 compression function computation (level 1, index 1):
Input (89 words):
A[   0] = 7311c2812425cfa0 Q[0]
A[   1] = 6432286434aac8e7 Q[1]
A[   2] = b60450e9ef68b7c1 Q[2]
A[   3] = e8fb23908d9f06f1 Q[3]
A[   4] = dd2e76cba691e5bf Q[4]
A[   5] = 0cd0d63b2c30bc41 Q[5]
A[   6] = 1f8ccf6823058f8a Q[6]
A[   7] = 54e5ed5b88e3775d Q[7]
A[   8] = 4ad12aae0a6d6031 Q[8]
A[   9] = 3e7f16bb88222e0d Q[9]
A[  10] = 8af8671d3fb50c2c Q[10]
A[  11] = 995ad1178bd25c31 Q[11]
A[  12] = c878c1dd04c4b633 Q[12]
A[  13] = 3b72066c7a1552ac Q[13]
A[  14] = 0d6f3522631effcb Q[14]
A[  15] = 0000000000000000 key K[0]
A[  16] = 0000000000000000 key K[1]
A[  17] = 0000000000000000 key K[2]
A[  18] = 0000000000000000 key K[3]
A[  19] = 0000000000000000 key K[4]
A[  20] = 0000000000000000 key K[5]
```

```
A[  21] = 0000000000000000 key K[6]
A[  22] = 0000000000000000 key K[7]
A[  23] = 0100000000000001 nodeID U = (ell,i) = (1,1)
A[  24] = 0068000000000100 control word V = (r,L,z,p,keylen,d) = (104,0,0,0,0,256)
A[  25] = d0e1686ab52f2642 data B[ 0] chaining from (1,0)
A[  26] = 23ae78b2f1225d3d data B[ 1] chaining from (1,0)
A[  27] = 1d3f60e3f6ebe9e6 data B[ 2] chaining from (1,0)
A[  28] = 0e2a6dd7ca126f8f data B[ 3] chaining from (1,0)
A[  29] = 9ced68e939d173e8 data B[ 4] chaining from (1,0)
A[  30] = f1260e4e80af4bac data B[ 5] chaining from (1,0)
A[  31] = dcd5ab34b0084bb0 data B[ 6] chaining from (1,0)
A[  32] = 0d4486d2a95c0ea7 data B[ 7] chaining from (1,0)
A[  33] = e0c1456140162bc6 data B[ 8] chaining from (1,0)
A[  34] = 5c9b43a4afb76f91 data B[ 9] chaining from (1,0)
A[  35] = 5e2e233293e7832b data B[10] chaining from (1,0)
A[  36] = 8bb542c30087934c data B[11] chaining from (1,0)
A[  37] = 801da4582d8c3b82 data B[12] chaining from (1,0)
A[  38] = bdac086c24adb7e1 data B[13] chaining from (1,0)
A[  39] = a4b6198b4f41a95b data B[14] chaining from (1,0)
A[  40] = 5a362e4725f93b78 data B[15] chaining from (1,0)
A[  41] = 7711223344556677 data B[16] input message word   48
A[  42] = 1122334455667711 data B[17] input message word   49
A[  43] = 2233445566771122 data B[18] input message word   50
A[  44] = 3344556677112233 data B[19] input message word   51
A[  45] = 4455667711223344 data B[20] input message word   52
A[  46] = 5566771122334455 data B[21] input message word   53
A[  47] = 6677112233445566 data B[22] input message word   54
A[  48] = 7711223344556677 data B[23] input message word   55
A[  49] = 1122334455667711 data B[24] input message word   56
A[  50] = 2233445566771122 data B[25] input message word   57
A[  51] = 3344556677112233 data B[26] input message word   58
A[  52] = 4455667711223344 data B[27] input message word   59
A[  53] = 5566771122334455 data B[28] input message word   60
A[  54] = 6677112233445566 data B[29] input message word   61
A[  55] = 7711223344556677 data B[30] input message word   62
A[  56] = 1122334455667711 data B[31] input message word   63
A[  57] = 2233445566771122 data B[32] input message word   64
A[  58] = 3344556677112233 data B[33] input message word   65
A[  59] = 4455667711223344 data B[34] input message word   66
A[  60] = 5566771122334455 data B[35] input message word   67
A[  61] = 6677112233445566 data B[36] input message word   68
A[  62] = 7711223344556677 data B[37] input message word   69
A[  63] = 1122334455667711 data B[38] input message word   70
A[  64] = 2233445566771122 data B[39] input message word   71
A[  65] = 3344556677112233 data B[40] input message word   72
A[  66] = 4455667711223344 data B[41] input message word   73
```

```
A[  67] = 5566771122334455 data B[42] input message word   74
A[  68] = 6677112233445566 data B[43] input message word   75
A[  69] = 7711223344556677 data B[44] input message word   76
A[  70] = 1122334455667711 data B[45] input message word   77
A[  71] = 2233445566771122 data B[46] input message word   78
A[  72] = 3344556677112233 data B[47] input message word   79
A[  73] = 4455667711223344 data B[48] input message word   80
A[  74] = 5566771122334455 data B[49] input message word   81
A[  75] = 6677112233445566 data B[50] input message word   82
A[  76] = 7711223344556677 data B[51] input message word   83
A[  77] = 1122334455667711 data B[52] input message word   84
A[  78] = 2233445566771122 data B[53] input message word   85
A[  79] = 3344556677112233 data B[54] input message word   86
A[  80] = 4455667711223344 data B[55] input message word   87
A[  81] = 5566771122334455 data B[56] input message word   88
A[  82] = 6677112233445566 data B[57] input message word   89
A[  83] = 7711223344556677 data B[58] input message word   90
A[  84] = 1122334455667711 data B[59] input message word   91
A[  85] = 2233445566771122 data B[60] input message word   92
A[  86] = 3344556677112233 data B[61] input message word   93
A[  87] = 4455667711223344 data B[62] input message word   94
A[  88] = 5566771122334455 data B[63] input message word   95
Intermediate values A[89..1736] omitted...
Output (16 words of chaining values):
A[1737] = 2fae6767b4be2806 output chaining value C[0]
A[1738] = a2c58070961ed34b output chaining value C[1]
A[1739] = e904278982816f0e output chaining value C[2]
A[1740] = 934bdf1bfaf89a48 output chaining value C[3]
A[1741] = 76b58590d48adde3 output chaining value C[4]
A[1742] = cd05fbf6a5f96726 output chaining value C[5]
A[1743] = 8ec3ff4d0c6af6c7 output chaining value C[6]
A[1744] = b87fbb6355077b92 output chaining value C[7]
A[1745] = d337638251ab837e output chaining value C[8]
A[1746] = fdafde77a9159856 output chaining value C[9]
A[1747] = ce755bb68e28b108 output chaining value C[10]
A[1748] = 3f71a3d304cc9f0b output chaining value C[11]
A[1749] = b9c42ccf1105e74a output chaining value C[12]
A[1750] = 3f0735075312a67f output chaining value C[13]
A[1751] = 1a6353195d8e1adf output chaining value C[14]
A[1752] = 6a9b5a6553635aab output chaining value C[15]

MD6 compression function computation (level 1, index 2):
Input (89 words):
A[   0] = 7311c2812425cfa0 Q[0]
A[   1] = 6432286434aac8e7 Q[1]
A[   2] = b60450e9ef68b7c1 Q[2]
```

```
A[   3] = e8fb23908d9f06f1 Q[3]
A[   4] = dd2e76cba691e5bf Q[4]
A[   5] = 0cd0d63b2c30bc41 Q[5]
A[   6] = 1f8ccf6823058f8a Q[6]
A[   7] = 54e5ed5b88e3775d Q[7]
A[   8] = 4ad12aae0a6d6031 Q[8]
A[   9] = 3e7f16bb88222e0d Q[9]
A[  10] = 8af8671d3fb50c2c Q[10]
A[  11] = 995ad1178bd25c31 Q[11]
A[  12] = c878c1dd04c4b633 Q[12]
A[  13] = 3b72066c7a1552ac Q[13]
A[  14] = 0d6f3522631effcb Q[14]
A[  15] = 0000000000000000 key K[0]
A[  16] = 0000000000000000 key K[1]
A[  17] = 0000000000000000 key K[2]
A[  18] = 0000000000000000 key K[3]
A[  19] = 0000000000000000 key K[4]
A[  20] = 0000000000000000 key K[5]
A[  21] = 0000000000000000 key K[6]
A[  22] = 0000000000000000 key K[7]
A[  23] = 0100000000000002 nodeID U = (ell,i) = (1,2)
A[  24] = 00680010b0000100 control word V = (r,L,z,p,keylen,d) = (104,0,1,2816,0,256)
A[  25] = 2fae6767b4be2806 data B[ 0] chaining from (1,1)
A[  26] = a2c58070961ed34b data B[ 1] chaining from (1,1)
A[  27] = e904278982816f0e data B[ 2] chaining from (1,1)
A[  28] = 934bdf1bfaf89a48 data B[ 3] chaining from (1,1)
A[  29] = 76b58590d48adde3 data B[ 4] chaining from (1,1)
A[  30] = cd05fbf6a5f96726 data B[ 5] chaining from (1,1)
A[  31] = 8ec3ff4d0c6af6c7 data B[ 6] chaining from (1,1)
A[  32] = b87fbb6355077b92 data B[ 7] chaining from (1,1)
A[  33] = d337638251ab837e data B[ 8] chaining from (1,1)
A[  34] = fdafde77a9159856 data B[ 9] chaining from (1,1)
A[  35] = ce755bb68e28b108 data B[10] chaining from (1,1)
A[  36] = 3f71a3d304cc9f0b data B[11] chaining from (1,1)
A[  37] = b9c42ccf1105e74a data B[12] chaining from (1,1)
A[  38] = 3f0735075312a67f data B[13] chaining from (1,1)
A[  39] = 1a6353195d8e1adf data B[14] chaining from (1,1)
A[  40] = 6a9b5a6553635aab data B[15] chaining from (1,1)
A[  41] = 6677112233445566 data B[16] input message word   96
A[  42] = 7711223344556677 data B[17] input message word   97
A[  43] = 1122334455667711 data B[18] input message word   98
A[  44] = 2233445566771122 data B[19] input message word   99
A[  45] = 0000000000000000 data B[20] padding
A[  46] = 0000000000000000 data B[21] padding
A[  47] = 0000000000000000 data B[22] padding
A[  48] = 0000000000000000 data B[23] padding
```

```
A[  49] = 0000000000000000 data B[24] padding
A[  50] = 0000000000000000 data B[25] padding
A[  51] = 0000000000000000 data B[26] padding
A[  52] = 0000000000000000 data B[27] padding
A[  53] = 0000000000000000 data B[28] padding
A[  54] = 0000000000000000 data B[29] padding
A[  55] = 0000000000000000 data B[30] padding
A[  56] = 0000000000000000 data B[31] padding
A[  57] = 0000000000000000 data B[32] padding
A[  58] = 0000000000000000 data B[33] padding
A[  59] = 0000000000000000 data B[34] padding
A[  60] = 0000000000000000 data B[35] padding
A[  61] = 0000000000000000 data B[36] padding
A[  62] = 0000000000000000 data B[37] padding
A[  63] = 0000000000000000 data B[38] padding
A[  64] = 0000000000000000 data B[39] padding
A[  65] = 0000000000000000 data B[40] padding
A[  66] = 0000000000000000 data B[41] padding
A[  67] = 0000000000000000 data B[42] padding
A[  68] = 0000000000000000 data B[43] padding
A[  69] = 0000000000000000 data B[44] padding
A[  70] = 0000000000000000 data B[45] padding
A[  71] = 0000000000000000 data B[46] padding
A[  72] = 0000000000000000 data B[47] padding
A[  73] = 0000000000000000 data B[48] padding
A[  74] = 0000000000000000 data B[49] padding
A[  75] = 0000000000000000 data B[50] padding
A[  76] = 0000000000000000 data B[51] padding
A[  77] = 0000000000000000 data B[52] padding
A[  78] = 0000000000000000 data B[53] padding
A[  79] = 0000000000000000 data B[54] padding
A[  80] = 0000000000000000 data B[55] padding
A[  81] = 0000000000000000 data B[56] padding
A[  82] = 0000000000000000 data B[57] padding
A[  83] = 0000000000000000 data B[58] padding
A[  84] = 0000000000000000 data B[59] padding
A[  85] = 0000000000000000 data B[60] padding
A[  86] = 0000000000000000 data B[61] padding
A[  87] = 0000000000000000 data B[62] padding
A[  88] = 0000000000000000 data B[63] padding
Intermediate values A[89..1736] omitted...
Output (16 words of chaining values):
A[1737] = 969f16af5a7eeda6 output chaining value C[0]
A[1738] = 75483a88c579b003 output chaining value C[1]
A[1739] = 0dc223df76efa88a output chaining value C[2]
A[1740] = 830ae40e6a3c26cb output chaining value C[3]
```

```
A[1741] = 5e6465d14c92e806 output chaining value C[4]
A[1742] = b45bfef654e87974 output chaining value C[5]
A[1743] = 0e94db3983fc413c output chaining value C[6]
A[1744] = a6fe8edf36ea4d92 output chaining value C[7]
A[1745] = 9506fe8428a971cd output chaining value C[8]
A[1746] = d4ea6a408a2ad417 output chaining value C[9]
A[1747] = c47b1bad0cd35209 output chaining value C[10]
A[1748] = 577c2a468919e473 output chaining value C[11]
A[1749] = 4e78ab5ec8926a3d output chaining value C[12]
A[1750] = b0dcfa09ed48de6c output chaining value C[13]
A[1751] = 33a7399e70f01ebf output chaining value C[14]
A[1752] = c02abb52767594e2 output chaining value C[15]

4e78ab5ec8926a3db0dcfa09ed48de6c33a7399e70f01ebfc02abb52767594e2 -B800
```

The final hash value is `0x4e78ab5e...94e2` .

Appendix D

Notation

Variable	Default	Usage
A	–	a vector of words used in the definition of f.
a	$rc + n$	length of the vector A used in the definition of f.
B	–	the data block portion of a compression function input.
b	64	the number of words in array B.
c	16	number of words in the compression function output.
d	–	number of bits in the MD6 final output ($1 \leq d \leq 512$).
f	–	the MD6 compression function mapping W^n to W^c.
f_Q	–	f without the fixed prefix, mapping W^{n-q} to W^c: $f_Q(x) = f(Q\|\|x)$.
g	–	an intra-word diffusion function.
K	0	the key variable (an input to f).
k	8	number of compression input words for the key variable K.
$keylen$	0	the length in bytes of the supplied key; $0 \leq keylen \leq kw/8$.
ℓ	–	the level number of a compression node.
ℓ_i	–	a left-shift amount for compression function step i.
L	64	mode parameter (maximum level number).
M	–	The input message to be hashed.
m	–	the length of the input message M, in bits.
N	–	the input block to the compression function.
n	89	the length of N in words.
p	–	the number of padding bits in a data block B.
Q	–	an approximation to $\sqrt{6}$.
q	15	the length of Q in words.
r	$40 + \lfloor d/4 \rfloor$	number of 16-step rounds in the compression function computation.
r_i	–	a right-shift amount for compression function step i.
S_i	–	a constant used in step i of the compression function.
S'_j	–	an auxiliary sequence of values used to define sequence S_i.
t	rc	the number of computation steps in the compression function.

Variable	Default	Usage
t_0	17	first tap position.
t_1	18	second tap position.
t_2	21	third tap position.
t_3	31	fourth tap position.
t_4	67	fifth tap position.
t_5	n	last tap position.
U	–	one-word unique node ID
u	1	length of U in words
V	–	a control word input to a compression function.
v	1	length of V in words.
W	$\{0,1\}^w$	the set of all w-bit words.
w	64	the number of bits in a word.
z	–	flag bit in V indicating this is final compression.

Appendix E

Additional documents

This appendix describes some additional documents submitted to NIST (or publicly available) that help to describe or illustrate the MD6 hash function.

E.1 Powerpoint slides from Rivest's CRYPTO talk

We have also included the Powerpoint slides from Professor Rivest's CRYPTO 2008 talk on MD6; this is file `crypto2008.ppt`.

E.2 Crutchfield thesis

The June 2008 MIT EECS Master's Thesis of Christopher Crutchfield, entitled "Security Proofs for the MD6 Mode of Operation," is included with our NIST submission.

E.3 Code

The MD6 Reference Implementation, Optimized_32Bit Implementation, and Optimized_64Bit implementation, are included with our NIST submission, as required.

Also included with our submission are the following programs:

`shiftopt.c` — A program used to optimize the shift amounts for MD6.

`tapopt.c` — A program used to optimize the tap locations for MD6.

`md6sum.c` — A utility program used to test MD6 and produce the intermediate value printouts.

All of our software is provided under the open source "MIT License" [1], which has very liberal terms for copying, redistribution, modification, etc.

E.4 KAT and MCT tests

Our submission to NIST includes the KAT (Known Answer Test) and MCT (Monte Carlo Test) results as required by NIST.

There are no NIST requirements for submitting KAT or MCT test results for keyed (or "salted") versions of a submitted hash function, but Appendix C provides some such examples (albeit with reduced number of rounds).

E.5 One-block and two-block examples

Our submission to NIST also includes one-block and two-block examples of MD6 sample computations.

These examples include the required NIST digest sizes: 224 bits, 256 bits, 384 bits, and 512 bits.

For each digest size, a one-block example and a two-block example are provided. The one-block examples have an input of length 400 bytes (recall that a single MD6 compression function call can handle inputs of length up to 512 bytes). The "two-block" examples have an input of length 800 bytes.

For each combination of digest size and input size (one-block or two-block), there are three sample outputs provided:

- Standard MD6.
- MD6 in sequential mode (i.e., with $L = 0$).
- MD6 with a nine-byte key in standard mode.

Note that the "two-block" examples show three compression function calls, since in standard mode there will be two compression function calls at level 1, and one at level 2 that produces the final output, and in sequential mode the 800-byte input doesn't quite fit into two 384-byte input blocks.

E.6 Web site

We have created a web site that makes publicly available the documents we have submitted to NIST for the SHA-3 competition, as well as copies of (or pointers to) other documents about MD6.

This web site is available at URL:

`http://groups.csail.mit.edu/cis/md6/`.

[1] `http://www.opensource.org/licenses/mit-license.php`

Appendix F

MD6 team members

Here we give a little more information about each of the MD6 team members, the aspects of the MD6 development they participated in, and contact information.

(Thanks to all of the MD6 team members for their efforts!)

- Ronald L. Rivest is the lead MD6 architect and team leader. He is a Professor in the MIT EECS Department.
  Web site: `http://people.csail.mit.edu/rivest`
  Phone: 617-253-5880
  Email: `rivest@mit.edu`

- Benjamin Agre is an MIT undergraduate. He worked on testing MD6 using various statistical packages.
  Email: `bagre@mit.edu`

- Daniel V. Bailey studied fast embedded crypto implementation with Christof Paar at WPI. He is now at RSA, and worked on the performance figures for MD6 on 8-bit processors.
  Email: `dbailey@rsa.com`

- Christopher Crutchfield finished his Master's thesis on the MD6 mode of operation at MIT in 2008, and is now working at a startup.
  Email: `cycrutchfield@gmail.com`

- Yevgeniy Dodis is a Professor in the Department of Computer Science at New York University; he contributed to the proofs of security for the MD6 compression function and for the MD6 mode of operation.
  Email: `dodis@cs.nyu.edu`

- Kermin Elliott Fleming is a graduate student in the Computation Structures Group at MIT's Computer Science and Artificial Intelligence Laboratory. He did the MD6 hardware designs reported in Chapter 5, and also the GPU implementation reported in Section 4.7.2.
  Email: `kfleming@mit.edu`

- Asif Khan is a graduate student in the Computation Structures Group at MIT's Computer Science and Artificial Intelligence Laboratory. He worked on the custom multicore designs for MD6 reported in Section 5.4.
  Email: `aik@mit.edu`

- Jayant Krishnamurthy is an MEng student in MIT's EECS Department. He has worked on a variety of software aspects of MD6, including the "clean-room" implementation, the testing framework, efficiency optimizations, and a CILK implementation.
  Email: `jayant@mit.edu`

- Yuncheng Lin graduated from MIT in 2008, and is going to graduate school at Stanford in the Mathematics Department. He has worked on evaluating the security of MD6 against differential attack, as reported in Section 6.9.
  Email: `linyc@mit.edu`

- Leo Reyzin is a Professor of Computer Science at Boston University. He has worked on the security of MD6's mode of operation.
  Email: `reyzin@cs.bu.edu`
  Web site: `http://www.cs.bu.edu/~reyzin/`

- Emily Shen is a graduate student in the Cryptography and Information Security group in MIT's Computer Science and Artificial Intelligence Laboratory. She has worked on the security of MD6's mode of operation.
  Email: `eshen@csail.mit.edu`

- Jim Sukha was a graduate student in Professor Leiserson's research group at MIT's Computer Science and Artificial Intelligence Laboratory. He developed an initial CILK implementation of MD6.
  Email: `sukhaj@mit.edu`

- Drew Sutherland received his Ph.D. from the MIT Mathematics Department in 2007, on the topic of "Order Computations in Generic Groups."
  Email:`drew@math.mit.edu`

- Eran Tromer is a post-doc in the Cryptography and Information Security group within MIT's Computer Science and Artificial Intelligence Laboratory. He has worked on a variety of aspects of MD6's design and security, including the `minisat` experiments reported in Chapter 6.
  Email: `tromer@csail.mit.edu`
  Web site: `http://people.csail.mit.edu/tromer/`

- Yiqun Lisa Yin received her PhD from MIT's Mathematics Department, and is now an independent cryptography and security consultant. She has worked on a variety of aspects of MD6's design and security, including differential cryptanalysis reported in Section 6.9 and linear cryptanalysis reported in Section 6.10.

Email: `yiqun@alum.mit.edu`
Web site: `http://people.csail.mit.edu/yiqun/`