The Skein Hash Function Family

Version 1.2 — 15 Sep 2009

Niels Ferguson Microsoft Corp., niels@microsoft.com
Stefan Lucks Bauhaus-Universität Weimar, stefan.lucks@uni-weimar.de
Bruce Schneier BT Group plc, schneier@schneier.com
Doug Whiting Hifn, Inc. dwhiting@hifn.com
Mihir Bellare University of California San Diego, mihir@cs.ucsd.edu
Tadayoshi Kohno University of Washington, yoshi@cs.washington.edu
Jon Callas PGP Corp., jon@pgp.com
Jesse Walker Intel Corp., jesse.walker@intel.com

Executive Summary

Skein is a new family of cryptographic hash functions. Its design combines speed, security, simplicity, and a great deal of flexibility in a modular package that is easy to analyze.

Skein is fast. Skein-512—our primary proposal—hashes data at 6.1 clock cycles per byte on a 64-bit CPU. This means that on a 3.1 GHz x64 Core 2 Duo CPU, Skein hashes data at 500 MBytes/second per core—almost twice as fast as SHA-512 and three times faster than SHA-256. An optional hash-tree mode speeds up parallelizable implementations even more. Skein is fast for short messages, too; Skein-512 hashes short messages in about 1000 clock cycles.

Skein is secure. Its conservative design is based on the Threefish block cipher. Our current best attack on Threefish-512 is on 25 of 72 rounds, for a safety factor of 2.9. For comparison, at a similar stage in the standardization process, the AES encryption algorithm had an attack on 6 of 10 rounds, for a safety factor of only 1.7. Additionally, Skein has a number of provably secure properties, greatly increasing confidence in the algorithm.

Skein is simple. Using only three primitive operations, the Skein compression function can be easily understood and remembered. The rest of the algorithm is a straightforward iteration of this function.

Skein is flexible. Skein is defined for three different internal state sizes—256 bits, 512 bits, and 1024 bits—and any output size. This allows Skein to be a drop-in replacement for the entire SHA family of hash functions. A completely optional and extendable argument system makes Skein an efficient tool to use for a very large number of functions: a PRNG, a stream cipher, a key derivation function, authentication without the overhead of HMAC, and a personalization capability. All these features can be implemented with very low overhead. Together with the Threefish large-block cipher at Skein's core, this design provides a full set of symmetric cryptographic primitives suitable for most modern applications.

Skein is efficient on a variety of platforms, both hardware and software. Skein-512 can be implemented in about 200 bytes of state. Small devices, such as 8-bit smart cards, can implement Skein-256 using about 100 bytes of memory. Larger devices can implement the larger versions of Skein to achieve faster speeds.

Skein was designed by a team of highly experienced cryptographic experts from academia and industry, with expertise in cryptography, security analysis, software, chip design, and implementation of real-world cryptographic systems. This breadth of knowledge allowed them to create a balanced design that works well in all environments.

Contents

1 Introduction

Cryptographic hash functions are the workhorses of cryptography, and can be found everywhere. Originally created to make digital signatures more efficient, they are now used to secure the very fundamentals of our information infrastructure: in password logins, secure web connections, encryption key management, virus- and malware-scanning, and almost every cryptographic protocol in current use. Without hash functions, the Internet would simply not work.

The most commonly used hash functions are those of the SHA family: SHA-0 [80], SHA-1 [81], SHA-256, and SHA-512 [83], all based on MD4 [93] and MD5 [94]. These SHA variants were all developed by the National Security Agency (NSA) and certified by the National Institute for Standards and Technology (NIST) [80, 81, 83], and are part of several NIST standards [84, 85, 3, 4] and many Internet standards.

Over the past few years, cryptanalysis of these functions has found serious weaknesses. Practical collisions have been demonstrated in MD4 [30, 106, 56, 107], MD5 [106, 108, 56, 57, 58, 59, 103], and SHA-0 [22, 106, 109]. Known collision attacks against SHA-1 are not yet practical, but they are still more than 10,000 times faster than what was expected [109]. To date, no flaws have been found in SHA-256 and SHA-512 [43], but the common heritage and design principles of all these functions makes them suspect. More seriously, if SHA-256 and SHA-512 were to be broken, the industry would be left without any generally accepted hash functions.

To address this undesirable situation, NIST created a design competition for the next generation of hash functions [86]. NIST has asked for proposals [87] and will likely select one as the new SHA-3 hash algorithm sometime in the year 2012. While there is no immediate need to migrate to this new standard, it is assumed that SHA-3 will see widespread use world-wide as applications and standards start using it.

This document introduces Skein[1], our submission to the SHA-3 competition.

2 Skein

2.1 Overview

Skein is a family of hash functions with three different internal state sizes: 256, 512, and 1024 bits.

- Skein-512 is our primary proposal. It can safely be used for all current hashing applications, and should remain secure for the foreseeable future.
- Skein-1024 is our ultra-conservative variant. Because it has twice the internal-state size of Skein-512, it is failure friendly; even if some future attack managed to break Skein-512, it is quite likely that Skein-1024 would remain secure. Skein-1024 can also run nearly twice as fast as Skein-512 in dedicated hardware implementations.
- Skein-256 is our low-memory variant. It can be implemented using about 100 bytes of RAM.

Each of these state sizes can support any output size. When a drop-in replacement is required for MD5 or one of the existing SHA hash functions, we recommend one of the configurations in

[1] A "skein"—pronounced \skān\ and rhymes with "rain"—is a loosely coiled length of yarn or thread wound on a reel.

		State	Output
Replace	With	Size	Size
MD5	Skein-256-128	256	128
	Skein-512-128	512	128
SHA-1	Skein-256-160	256	160
	Skein-512-160	512	160
SHA-224	Skein-256-224	256	224
	Skein-512-224	512	224
SHA-256	Skein-256-256	256	256
	Skein-512-256	512	256
SHA-384	Skein-512-384	512	384
	Skein-1024-384	1024	384
SHA-512	Skein-512-512	512	512
	Skein-1024-512	1024	512

Table 1: Drop-in replacements for MD5, SHA-1 and SHA-2.

Table 1.

Skein's novel idea is to build a hash function out of a tweakable block cipher. The use of a tweakable block cipher allows Skein to hash configuration data along with the input text in every block, and make every instance of the compression function unique. This property directly addresses many attacks on hash functions, and greatly improves Skein's flexibility.

More specifically, Skein is built from these three new components:

- **Threefish.** Threefish is the tweakable block cipher at the core of Skein, defined with a 256-, 512-, and 1024-bit block size.

- **Unique Block Iteration (UBI).** UBI is a chaining mode that uses Threefish to build a compression function that maps an arbitrary input size to a fixed output size.

- **Optional Argument System.** This allows Skein to support a variety of optional features without imposing any overhead on implementations and applications that do not use the features.

Dividing up our design in this way makes Skein easier to understand, analyze, and prove properties about. The underlying Threefish algorithm draws upon years of knowledge of block cipher design and analysis. UBI is provably secure and can be used with *any* tweakable cipher. The optional argument system allows Skein to be tailored for different purposes. These three components are independent, and are usable on their own, but it's their combination that provides real advantages. And every aspect of Skein was designed to optimize those advantages.

In the following subsections, we describe each component of Skein. While this description is comprehensive enough for a reader to understand how Skein works, many details are either hidden or glossed over. For a complete description of Skein, see the full specification in Section 3.

2.2 The Threefish Block Cipher

Threefish is a large, tweakable block cipher [66]. It is defined for three different block sizes: 256 bits, 512 bits, and 1024 bits. The key is the same size as the block, and the tweak value is 128 bits for all block sizes.

The core design principle of Threefish is that a larger number of simple rounds is more secure than fewer complex rounds. Threefish uses only three mathematical operations—exclusive-or (XOR), addition, and constant rotations—on 64-bit words—and is very fast on modern 64-bit CPUs.

Figure 1 illustrates the core of Threefish: a simple non-linear mixing function, called MIX, that operates on two 64-bit words. Each MIX function consists of a single addition, a rotation by a constant, and an XOR.

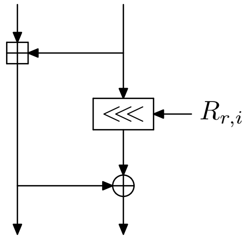

Figure 1: The MIX function.

Figure 2 shows how MIX functions are used to build Threefish-512. Each of Skein-512's 72 rounds consists of four MIX functions followed by a permutation of the eight 64-bit words. A subkey is injected every four rounds. The word permutation, ”Permute,” is the same for every round; the rotation constants are chosen to maximize diffusion and repeat every eight rounds.

The key schedule generates the subkeys from the key and the tweak. Each subkey consists of three contributions: key words, tweak words, and a counter value. To create the key schedule, the key and tweak are each extended with one extra parity word that is the XOR of all the other words. Each subkey is a combination of all but one of the extended key words, two of the three extended tweak words, and the subkey number as shown in Figure 3. Between subkeys, both the extended key and extended tweak are rotated by one word position. (For more details, see Section 3.3.2.) The entire key schedule can be computed in just a few CPU cycles, which minimizes the cost of using a new key—a critical consideration when using a block cipher in a hash function.

Figure 4 shows Threefish-256. Threefish-1024 is similar, except that it has eight MIX functions per round and 80 rounds total. The rotation constants and round permutations are different for each Threefish version, and were selected to maximize diffusion across the entire Threefish block. (See Section 8.3 for details on how the rotation constants and permutations were chosen.)

The nonlinearity in Threefish comes from the carry bits in the additions, each of which is a majority function of two input bits and another carry bit. The MIX/permute structure has been designed to provide full diffusion in 9 rounds for Threefish-256, 10 rounds for Threefish-512, and 11 rounds for Threefish-1024. At 72 and 80 rounds, Threefish has more full diffusions than most other block ciphers.

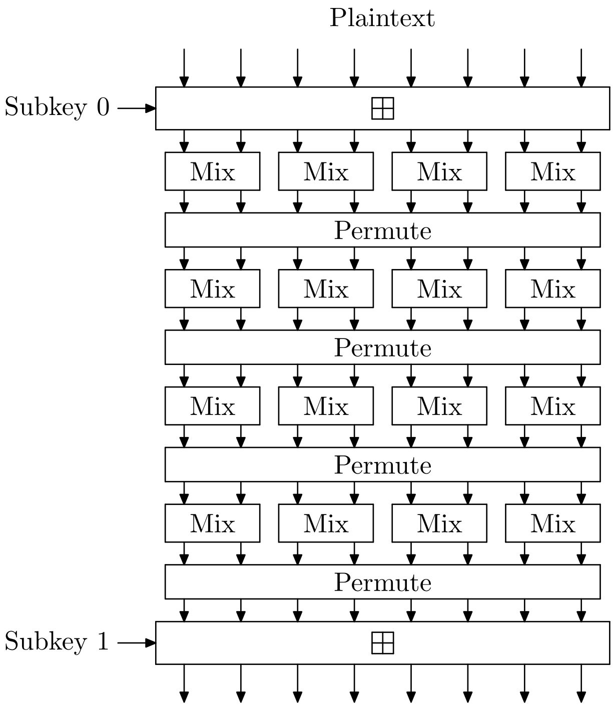

Figure 2: Four of the 72 rounds of the Threefish-512 block cipher.

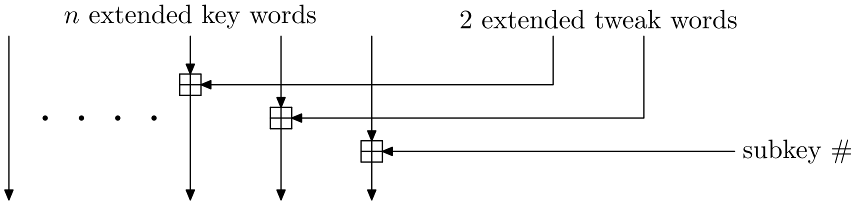

Figure 3: Constructing a Threefish subkey.

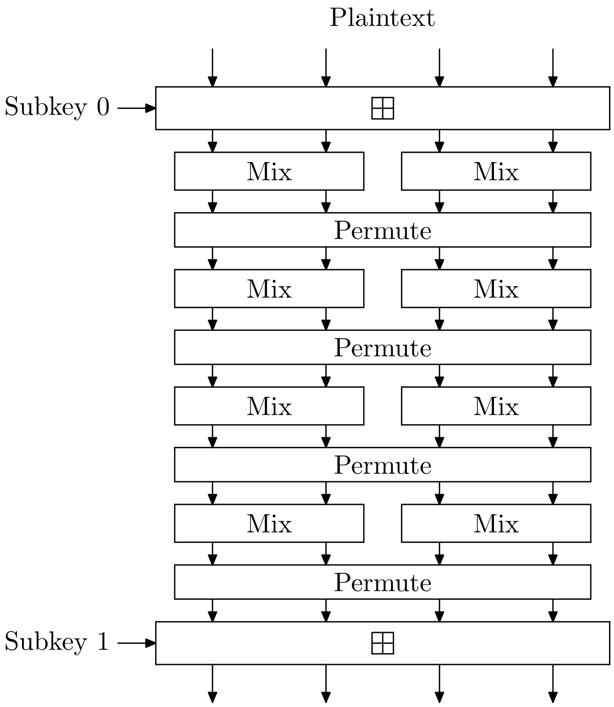

Figure 4: Four of the 72 rounds of the Threefish-256 block cipher.

2.3 The UBI Chaining Mode

The Unique Block Iteration (UBI) chaining mode combines an input chaining value with an arbitrary length input string and produces a fixed-size output. The easiest way to explain this is with an example. Figure 5 shows a UBI computation for Skein-512 on a 166-byte (three-block) input, which uses three calls to Threefish-512.

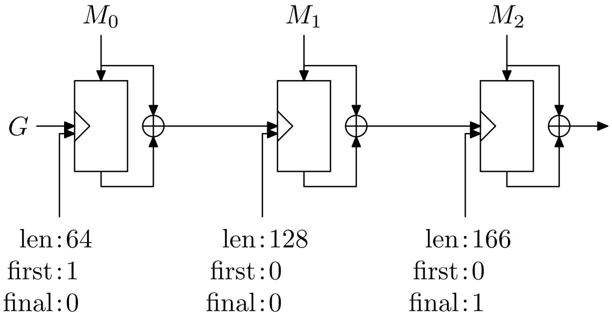

Figure 5: Hashing a three-block message using UBI mode.

Message blocks M_0 and M_1 contain 64 bytes of data each, and M_2 is the padded final block containing 38 bytes of data. The tweak value for each block encodes how many bytes have been processed so far, and whether this is the first and/or last block of the UBI computation. The tweak also encodes a "type" field—not shown in the figure—that is used to distinguish different uses of the UBI mode from each other.

The tweak is the heart of UBI. By using a tweakable cipher, UBI chaining mode ensures that every block is processed with a unique variant of the compression function. This stops a large variety of cut-and-paste attacks; a message piece that produces one result in one location will produce a different result in a different location.

UBI is a variant of the Matyas-Meyer-Oseas [70] hash mode. Unlike many other modes, the message input to the hash function is the same as the plaintext input to the block cipher. Since the attacker has the greatest control over the message input, this provides an additional level of security.

2.4 Skein Hashing

Skein is built on multiple invocations of UBI. Figure 6 shows Skein as a straightforward hash function. Starting with a chaining value of 0, there are three UBI invocations: one each for the configuration block, the message (up to $2^{96} - 1$ bytes long), and the output transform.

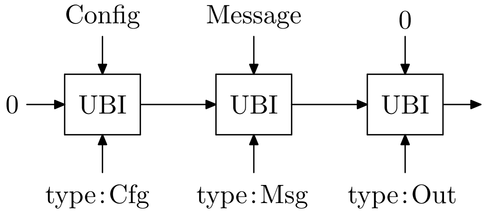

Figure 6: Skein in normal hashing mode.

The 32-byte configuration string encodes the desired output length and some parameters to support tree hashing. If Skein is used as a standard hash function—a fixed output size and no tree hashing or MAC key—the result of the configuration block UBI computation is constant for all messages and can be precomputed as an IV. A list of suitable precomputed chaining values is given in Appendix B.

The output transform is required to achieve hashing-appropriate randomness. It also allows Skein to produce any size output up to 2^{64} bits. If a single output block is not enough, run the output transform several times, as shown in Figure 7. The chaining input to all output transforms is the same, and the data field consists of an 8-byte counter. Essentially, this uses Threefish in counter mode. Producing large outputs is often convenient, but—of course—the security of Skein is limited by the internal state size.

2.5 Optional Arguments

In order to increase the flexibility of Skein, several optional inputs can be enabled as needed. These options are all driven by real-world applications we have worked on.

- **Key** (Optional) A key that turns Skein into a MAC or KDF function. The key is always processed first to support some of our security proofs.
- **Configuration** (Required) The configuration block discussed above.
- **Personalization** (Optional) A string that applications can use to create different functions for different uses.

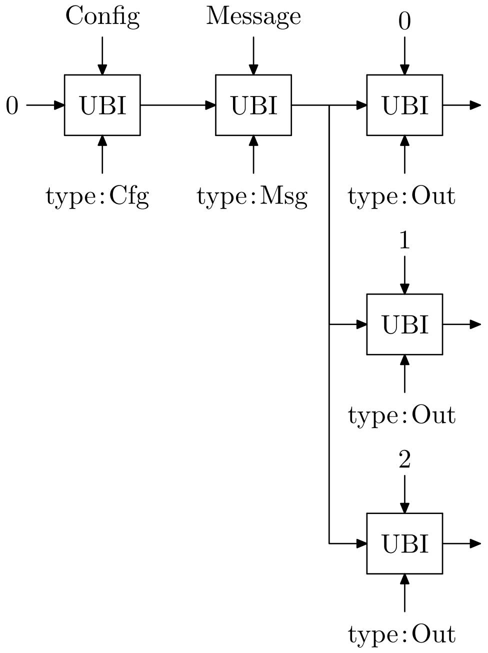

Figure 7: Skein with larger output size.

- **Public Key** (Optional) Used to hash the public key when hashing a message for signing. This ties the signature hash to the public key. Thus, this feature ensures that the same message generates different hashes for different public keys.

- **Key Derivation Identifier** (Optional) Used for key derivation. To derive a key, provide the master key as the key input, and the identifier of the requested derived key here.

- **Nonce** (Optional) Nonce value for use in stream cipher mode and randomized hashing.

- **Message** (Optional) The normal message input of the hash function.

- **Output** (Required) The output transform.

A Skein computation consists of processing these options in order, using UBI. Each input has a different "type" value for the tweak, ensuring that inputs are not interchangeable.

None of these impact the performance and complexity of the basic hash function in any way; different implementations can choose which options to implement and which to ignore.

Obviously, Skein can be extended with other optional arguments. These can be added at any time, even when the function has already been standardized, as adding new optional arguments is backwards-compatible. We welcome suggestions for other optional arguments.

2.6 Skein-MAC

The standard way to use a hash function for authentication is to use the HMAC construction [6, 85]. Skein can—of course—be used with HMAC, but this requires at least two hash computations for every authentication, which is inefficient for short messages. Skein has zero per-message overhead when used as a MAC function.

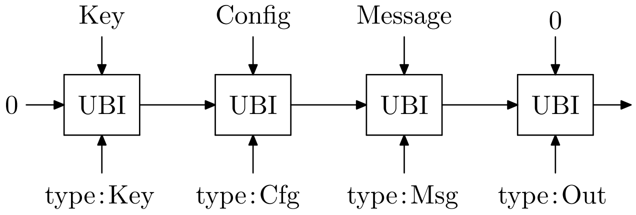

Figure 8: Skein-MAC.

Turning Skein into a MAC is simple, as illustrated in Figure 8. Instead of starting with zero and processing the configuration block, start with zero, process the key, and then the configuration block. Or, looking at it the other way, Skein hashing is simply Skein-MAC with a null key. And just as Skein's output of the configuration block is a precomputable constant for a given state and output size, Skein-MAC's output of the configuration block can be precomputed for a given key. Since the most common way to use a MAC is to authenticate multiple messages with a single key, this considerably increases performance for short messages.

2.7 Tree Hashing with Skein

When hashing very large amounts of data, the linear structure of a classical linear hash function becomes a limitation; it prevents a multi-core CPU from using multiple cores at the same time. Also, a common use of hash functions is to verify the integrity of a large amount of data. With a linear hash function, all the data has to be verified at the same time. This can be very inefficient, as it is often desirable to verify the integrity of only a small part of the data.

A hash tree [73, 74] solves both these problems. Rather than hashing the data as one large string, the data is cut into pieces. Each piece is hashed, and the resulting hashes are treated as a new message. This procedure can be applied recursively until the result is a single hash value.

Skein includes an optional hash tree mode to support these type of applications. As different applications have different requirements, there are three parameters that the application can choose among to optimize the hash tree for its particular use: the leaf node size, the tree fan-out, and the maximum tree height. This structure is explained more fully in Section 3.5.6.

3 A Full Specification of Skein

This section provides a complete specification of Skein. Readers not interested in technical details might want to skip to the "Using Skein" section on page 18.

3.1 Strings

When we talk about a "string of X's," we mean a sequence of zero or more values, each of which has type X. For example: a string of bytes is a sequence of zero or more bytes. We write strings as comma-separated lists, and typically number the items starting at zero; for example, a string t of 7 values is written:

$$t = t_0, t_1, \ldots, t_6$$

The concatenation operator $\|$ denotes concatenation of strings. We use 0^n to denote a string of n zeroes, where the type of zeroes (bits or bytes) will be clear from the context.

3.2 Bit and Byte Order

The order of bits and bytes is a common source of confusion in cryptographic algorithms. In short: Skein always uses the least-significant-byte-first convention. But to ensure there are no misunderstandings, we give formal definitions of our data type conversions.

The basic universal data type in modern CPUs is a string of bytes. Each byte has a value in the range 0..255. A byte is also often viewed as a sequence of 8 bits $b_7, b_6, \ldots, b_0$, where each b_i is either 0 or 1 and the byte value b is given by:

$$b := \sum_{i=0}^{7} b_i \cdot 2^i$$

Value b_i is often referred to as "bit i" of b.

A string of bits is stored as a string of bytes. For the hash function competition, NIST specifies a particular mapping from a string of bits to a string of bytes. Every group of 8 bits is encoded in a byte; the first bit goes into bit 7 of the byte, the next into bit 6 of the byte, etc. If the length of the bit string is not a multiple of 8, the last byte is only partially used, with the lower bit positions going unused.

To convert from a sequence of bytes to an integer, we use the least-significant-byte-first convention. Let $b_0, \ldots, b_{n-1}$ be a string of n bytes. We define:

$$\text{ToInt}(b_0, b_1, \ldots, b_{n-1}) := \sum_{i=0}^{n-1} b_i \cdot 256^i$$

The reverse mapping is provided by the ToBytes function:

$$\text{ToBytes}(v, n) := b_0, b_1, \ldots, b_{n-1} \qquad \text{where } b_i := \left\lfloor \frac{v}{256^i} \right\rfloor \bmod 256$$

This function is only applied when $0 \leq v < 256^n$ so that the bytes fully encode the value v.

We often convert between a string of $8n$ bytes and a string of n 64-bit words and back. Let $b_0, \ldots, b_{8n-1}$ be the bytes. We define:

$$\text{BytesToWords}(b_0, \ldots, b_{8n-1}) := w_0, \ldots, w_{n-1} \qquad \text{where } w_i := \text{ToInt}(b_{8i}, b_{8i+1}, \ldots, b_{8i+7})$$

The reverse mapping is given by:

$$\text{WordsToBytes}(w_0, \ldots, w_{n-1}) := \text{ToBytes}(w_0, 8) \,\|\, \text{ToBytes}(w_1, 8) \,\|\, \cdots \,\|\, \text{ToBytes}(w_{n-1}, 8)$$

3.3 A Full Specification of Threefish

Threefish is a tweakable block cipher with a block size of 256, 512, or 1024 bits. The tweak input is always 128 bits.

The encryption function $E(K, T, P)$ takes the following arguments:

K	Block cipher key; a string of 32, 64, or 128 bytes (256, 512, or 1024 bits).
T	Tweak, a string of 16 bytes (128 bits).
P	Plaintext, a string of bytes of length equal to the key.

Threefish operates entirely on unsigned 64-bit words (i.e., values in the range $0..2^{64}-1$). All inputs are converted to strings of 64-bit words. Let N_w be the number of words in the key (and thus also in the plaintext). The key K is interpreted as key words $(k_0, k_1, \ldots, k_{N_w-1})$, the tweak T is interpreted as words (t_0, t_1), and the plaintext P as $(p_0, p_1, \ldots, p_{N_w-1})$.

$$\begin{aligned} k_0, \ldots, k_{N_w-1} &:= \text{BytesToWords}(K) \\ t_0, t_1 &:= \text{BytesToWords}(T) \\ p_0, \ldots, p_{N_w-1} &:= \text{BytesToWords}(P) \end{aligned}$$

The number of rounds, N_r, is a function of the block size as shown in Table 2.

Block/Key	# Words	# Rounds
Size	N_w	N_r
256	4	72
512	8	72
1024	16	80

Table 2: Number of rounds for different block sizes.

The key schedule (documented below) turns the key and tweak into a sequence of $N_r/4+1$ subkeys, each of which consists of N_w words. We denote the words of subkey s by $(k_{s,0}, \ldots, k_{s,N_w-1})$.

Let $v_{d,i}$ be the value of the ith word of the encryption state after d rounds. We start out with:

$$v_{0,i} := p_i \quad \text{for } i = 0, \ldots, N_w - 1$$

and then apply N_r rounds numbered $d = 0, \ldots, N_r - 1$.

For each round, we add a subkey if $d \bmod 4 = 0$. For $i = 0, \ldots, N_w - 1$ we have:

$$e_{d,i} := \begin{cases} (v_{d,i} + k_{d/4,i}) \bmod 2^{64} & \text{if } d \bmod 4 = 0 \\ v_{d,i} & \text{otherwise} \end{cases}$$

The mixing and word permutations are defined by:

$$\begin{aligned} (f_{d,2j}, f_{d,2j+1}) &:= \text{MIX}_{d,j}(e_{d,2j}, e_{d,2j+1}) && \text{for } j = 0, \ldots, N_w/2 - 1 \\ v_{d+1,i} &:= f_{d,\pi(i)} && \text{for } i = 0, \ldots, N_w - 1 \end{aligned}$$

	$i =$															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	0	3	2	1												
$N_w =$ 8	2	1	4	7	6	5	0	3								
16	0	9	2	13	6	11	4	15	10	7	12	3	14	5	8	1

Table 3: Values for the word permutation $\pi(i)$.

The $f_{d,i}$ values are the results of the MIX functions (defined below); and the output of the word permutation is the output of the round. The permutation $\pi()$ is given in Table 3.

The ciphertext C is given by:

$$c_i := (v_{N_r,i} + k_{N_r/4,i}) \bmod 2^{64} \qquad \text{for } i = 0, \ldots, N_w - 1$$

$$C := \text{WordsToBytes}(c_0, \ldots, c_{N_w-1})$$

3.3.1 MIX Functions

Function $\text{MIX}_{d,j}$ has two input words (x_0, x_1) and produces two output words (y_0, y_1) using the following relations:

$$y_0 := (x_0 + x_1) \bmod 2^{64}$$

$$y_1 := (x_1 \lll R_{(d \bmod 8),j}) \oplus y_0$$

where $\lll$ is the rotate-left operator. The constants $R_{d,j}$ are shown in Table 4. (These constants were changed in version 1.2 of the paper. See Appendix D for details.)

N_w	4		8				16							
j	0	1	0	1	2	3	0	1	2	3	4	5	6	7
0	14	16	46	36	19	37	24	13	8	47	8	17	22	37
1	52	57	33	27	14	42	38	19	10	55	49	18	23	52
2	23	40	17	49	36	39	33	4	51	13	34	41	59	17
$d =$ 3	5	37	44	9	54	56	5	20	48	41	47	28	16	25
4	25	33	39	30	34	24	41	9	37	31	12	47	44	30
5	46	12	13	50	10	17	16	34	56	51	4	53	42	41
6	58	22	25	29	39	43	31	44	47	46	19	42	44	25
7	32	32	8	35	56	22	9	48	35	52	23	31	37	20

Table 4: Rotation constants $R_{d,j}$ for each N_w.

3.3.2 The Key Schedule

The key schedule starts by defining two additional words k_{N_w} and t_2 by:

$$k_{N_w} := \lfloor 2^{64}/3 \rfloor \oplus \bigoplus_{i=0}^{N_w-1} k_i \qquad \text{and} \qquad t_2 := t_0 \oplus t_1$$

The constant $\lfloor 2^{64}/3 \rfloor$ ensures that the extended key cannot be all zeroes. The key schedule is now defined by:

$$\begin{aligned}
k_{s,i} &:= k_{(s+i) \bmod (N_w+1)} && \text{for } i = 0, \ldots, N_w - 4 \\
k_{s,i} &:= k_{(s+i) \bmod (N_w+1)} + t_{s \bmod 3} && \text{for } i = N_w - 3 \\
k_{s,i} &:= k_{(s+i) \bmod (N_w+1)} + t_{(s+1) \bmod 3} && \text{for } i = N_w - 2 \\
k_{s,i} &:= k_{(s+i) \bmod (N_w+1)} + s && \text{for } i = N_w - 1
\end{aligned}$$

where the additions are all modulo 2^{64}.

3.3.3 Decryption

The Threefish decryption operation is the obvious inverse of the encryption operation. Subkeys are used in reverse order and each round consists of applying the inverse word permutation followed by the inverse MIX functions.

3.4 A Full Specification of UBI

The UBI chaining mode is built on a tweakable block cipher with a block size and key size of N_b bytes, and a tweak size of 16 bytes. The function UBI(G, M, T_s) has inputs:

G	a starting value of N_b bytes.
M	a message string of arbitrary bit length up to $2^{99} - 8$ bits, encoded in a string of bytes.
T_s	a 128-bit integer that is the starting value for the tweak. (See below for some restrictions on the value of T_s.)

UBI processes the message in blocks using a unique tweak value for each block. The fields in the tweak are shown in Figure 9 and Table 5. To avoid having many different parameters, we treat the tweak as a single 128-bit value. This simplifies our notation but it imposes some restrictions on the value T_s can have. The BitPad, First, and Final field must be zero; the Position field must have a value such that the sum of the Position field plus the length of M in bytes does not exceed $2^{96} - 1$.

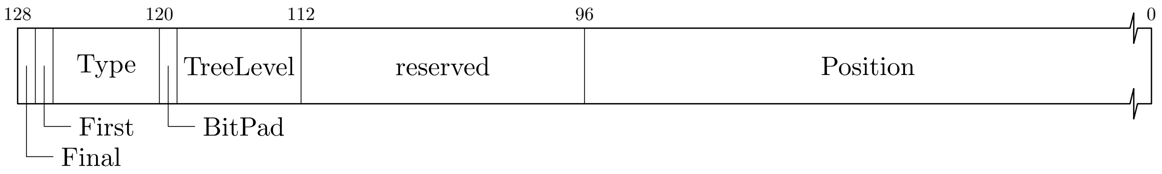

Figure 9: The fields in the tweak value.

If the number of bits in the data M is a multiple of 8, we define $B := 0$ and $M' := M$. If the number of bits in M is not a multiple of 8, the last byte is only partially used. The most significant bit positions of the last byte contain data. We pad the last byte by setting the most significant unused bit to 1 and the remaining unused bits (if any) to zero. We define $B := 1$ and let M' be M with the bit-padding applied.

Name	Bits	Description
Position	0– 95	The number of bytes in the string processed so far (including this block)
reserved	96–111	Reserved for future use, must be zero
TreeLevel	112–118	Level in the hash tree, zero for non-tree computations.
BitPad	119	Set if this block contains the last byte of an input whose length was not an integral number of bytes. 0 otherwise.
Type	120–125	Type of the field (config, message, output, etc.)
First	126	Set for the first block of a UBI compression.
Final	127	Set for the last block of a UBI compression.

Table 5: The fields in the tweak value.

Let N_M be the number of bytes in M'. The input is restricted to $N_M < 2^{96}$.

We pad M' with p zero bytes until the length is a multiple of the block size, ensuring that we get at least one whole block.

$$\begin{aligned} p &:= \begin{cases} N_b & \text{if } N_M = 0 \\ (-N_M) \bmod N_b & \text{otherwise} \end{cases} \\ M'' &:= M' \parallel 0^p \end{aligned}$$

We split M'' into k message blocks $M_0, \ldots, M_{k-1}$, each of N_b bytes. The UBI result is computed as

$$\begin{aligned} H_0 &:= G \\ H_{i+1} &:= E(H_i, \mathrm{ToBytes}(T_s + \min(N_M, (i+1)N_b) + a_i 2^{126} + b_i(B2^{119} + 2^{127}), 16), M_i) \oplus M_i \end{aligned}$$

where $a_0 = b_{k-1} = 1$, all other a_i and b_i values are 0, $E()$ is the tweakable block cipher encryption function, and H_k is the result of the UBI chaining mode.

The tweak value for each block is constructed by the addition

$$T_s + \min(N_M, (i+1)N_b) + a_i 2^{126} + b_i(B2^{119} + 2^{127})$$

The first term is T_s, which specifies the TreeLevel and Type fields, and optionally provides an offset for the Position field. The $\min(N_M, (i+1)N_b)$ term modifies only the Position field. For each block, the Position field is the number of bytes processed so far, including all the bytes in the current block, plus the offset from T_s. The T_s restrictions above ensure there is never a carry out of the Position field from this addition that could modify another field. The $a_i 2^{126}$ term sets the First flag, but only in the first block of a UBI computation. The $b_i(B2^{119} + 2^{127})$ term does two things. For any block except the last one, $b_i = 0$ so this term does nothing. In the last block, the Final flag is set (bit position 127) and if any bit padding was applied, then the BitPad flag is set (bit position 119).

3.5 A Full Specification of Skein

3.5.1 Type Values

Skein has many possible parameters. Each parameter, whether optional or mandatory, has its own unique type identifier and value. Type values are in the range 0..63. Skein processes the parameters in numerically increasing order of type value, as listed in Table 6.

Symbol	Value	Description
T_{key}	0	Key (for MAC and KDF)
T_{cfg}	4	Configuration block
T_{prs}	8	Personalization string
T_{PK}	12	Public key (for digital signature hashing)
T_{kdf}	16	Key identifier (for KDF)
T_{non}	20	Nonce (for stream cipher or randomized hashing)
T_{msg}	48	Message
T_{out}	63	Output

Table 6: Values for the type field.

3.5.2 The Configuration String

The configuration string contains the following data:

- A schema identifier. This is a literal constant. If some other standardization body wants to define an entirely different function based on UBI and Threefish, it can chose a different schema identifier and ensure that its function is different from Skein.
- A version number, to support future extensions.
- N_o: the output length of the computation, in bits. This ensures that two Skein computations that differ only in the number of output bits give unrelated results.
- Y_l: Tree leaf size encoding. Set to 0 if tree hashing is not used.
- Y_f: Tree fan-out encoding. Set to 0 if tree hashing is not used.
- Y_m: Max tree height. Set to 0 if tree hashing is not used.

The values for the tree parameters are detailed in Section 3.5.6. The layout of the 32-byte configuration string C is given in Table 7.

The reserved fields are present to support future extensions in a backward-compatible way.

3.5.3 The Output Function

The function Output(G, N_o) takes the following parameters:

G the chaining value.

Offset	Size in Bytes	Name	Description
0	4	Schema identifier	The ASCII string “SHA3” = (0x53, 0x48, 0x41, 0x33), or ToBytes(0x33414853,4)
4	2	Version number	Currently set to 1: ToBytes$(1, 2)$
6	2		Reserved, set to 0
8	8	Output length	ToBytes$(N_o, 8)$
16	1	Tree leaf size enc.	Y_l
17	1	Tree fan-out enc.	Y_f
18	1	Max. tree height	Y_m
19	13		Reserved, set to 0

Table 7: The Fields in the configuration value.

N_o the number of output bits required.

and produces N_o bits of output.

The result consists of the leading $\lceil N_o/8 \rceil$ bytes of:

$$\begin{aligned} O := &\mathrm{UBI}(G, \mathrm{ToBytes}(0, 8), T_{\mathrm{out}}2^{120})\| \\ &\mathrm{UBI}(G, \mathrm{ToBytes}(1, 8), T_{\mathrm{out}}2^{120})\| \\ &\mathrm{UBI}(G, \mathrm{ToBytes}(2, 8), T_{\mathrm{out}}2^{120})\| \\ &\cdots \end{aligned}$$

If $N_o \bmod 8 = 0$ the output is an integral number of bytes. If $N_o \bmod 8 \neq 0$ the last byte is only partially used.

3.5.4 Simple Hashing

A simple Skein hash computation has the following inputs:

N_b The internal state size, in bytes. Must be 32, 64, or 128.

N_o The output size, in bits.

M The message to be hashed, a string of up to $2^{99} - 8$ bits ($2^{96} - 1$ bytes).

Let C be the configuration string defined in Section 3.5.2 with $Y_l = Y_f = Y_m = 0$

We define:

$$\begin{aligned} K' &:= 0^{N_b} && \text{a string of } N_b \text{ zero bytes} \\ G_0 &:= \mathrm{UBI}(K', C, T_{\mathrm{cfg}}2^{120}) \\ G_1 &:= \mathrm{UBI}(G_0, M, T_{\mathrm{msg}}2^{120}) \\ H &:= \mathrm{Output}(G_1, N_o) \end{aligned}$$

where H is the result of the hash.

3.5.5 Full Skein

In its full general form, a Skein computation has the following inputs:

N_b	The internal state size, in bytes. Must be 32, 64, or 128.
N_o	The output size, in bits.
K	A key of N_k bytes. Set to the empty string ($N_k = 0$) if no key is desired.
Y_l	Tree hash leaf size encoding.
Y_f	Tree hash fan-out encoding.
Y_m	Maximum tree height.
L	List of t tuples (T_i, M_i) where T_i is a type value and M_i is a string of bits encoded in a string of bytes.

We have:

$$L := (T_0, M_0), \ldots, (T_{t-1}, M_{t-1})$$

We require that $T_{\text{cfg}} < T_0$, $T_i < T_{i+1}$ for all i, and $T_{t-1} < T_{\text{out}}$. An empty list L is allowed. Each M_i can be at most $2^{99} - 8$ bits ($= 2^{96} - 1$ bytes) long.

The first step is to process the key. If $N_k = 0$, the starting value consists of all zeroes.

$$K' := 0^{N_b}$$

If $N_k \neq 0$ we compress the key using UBI to get our starting value:

$$K' := \text{UBI}(0^{N_b}, K, T_{\text{key}} 2^{120})$$

Let C be the configuration string defined in Section 3.5.2. We compute:

$$G_0 := \text{UBI}(K', C, T_{\text{cfg}} 2^{120})$$

The parameters are then processed in order:

$$G_{i+1} := \text{UBI}(G_i, M_i, T_i 2^{120}) \qquad \text{for } i = 0, \ldots, t-1$$

with one exception: if the tree parameters Y_l, Y_f, and Y_m are not all zero, then an input tuple with $T_i = T_{\text{msg}}$ is processed as defined in Section 3.5.6, rather than with straight UBI.

And the final Skein result is given by:

$$H := \text{Output}(G_t, N_o)$$

3.5.6 Tree Processing

The message input (type T_{msg}) is special and can be processed as a tree. Figure 10 gives an example of how tree hashing works. Tree processing is controlled by the three tree parameters Y_l, Y_f, and Y_m in the config block. Normally (for non-tree hashing), these are all zero. If they are not all zero, the normal UBI function that processes the T_{msg} field is replaced by a tree hashing construction; this is a drop-in replacement of that one UBI function; all other parts of Skein are unchanged.

The tree hashing uses the following input parameters:

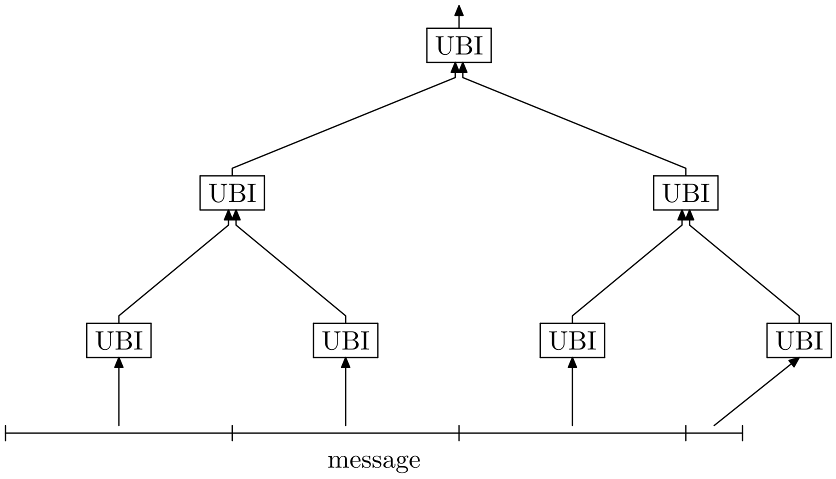

Figure 10: An overview of tree hashing.

Y_l The leaf size encoding. The size of each leaf of the tree is $N_b 2^{Y_l}$ bytes with $Y_l \geq 1$.

Y_f The fan-out encoding. The fan-out of a tree node is 2^{Y_f} with $Y_f \geq 1$.

Y_m The maximum tree height; $Y_m \geq 2$. (If the height of the tree is not limited, this parameter is set to 255.)

G The input chaining value. This is the G input of the UBI call that the tree hashing replaces, and the output of the previous UBI function in the Skein computation.

M The message data.

We define the leaf size $N_l := N_b 2^{Y_l}$ and the node size $N_n := N_b 2^{Y_f}$.

The message data M is a string of bits encoded in a string of bytes. We first split M into one or more message blocks $M_{0,0}, M_{0,1}, M_{0,2}, ..., M_{0,k-1}$. If M is the empty string, the split results in a single message block $M_{0,0}$ that is itself the empty bit string. If M is not the empty string, then blocks $M_{0,0}, \ldots, M_{0,k-2}$ all contain $8N_l$ bits and block $M_{0,k-1}$ contains between 1 and $8N_l$ bits.

We now define the first level of tree hashing:

$$M_1 := \mathop{\Big\|}_{i=0}^{k-1} \mathrm{UBI}(G, M_{0,i}, iN_l + 1 \cdot 2^{112} + T_{\mathrm{msg}} 2^{120})$$

Note that in the tweak, the tree level field is set to one and the Position field is given an offset equal to the starting offset (in bytes) of the message block.

The rest of the tree is defined iteratively. For any level $l = 1, 2, \ldots$ we use the following rules.

If M_l has length N_b then the result G_o is defined by $G_o := M_l$.

If M_l is longer than N_b bytes and $l = Y_m - 1$ then we have almost reached the maximum tree height. The result is defined by:

$$G_o := \mathrm{UBI}(G, M_l, Y_m \cdot 2^{112} + T_{\mathrm{msg}} 2^{120})$$

If neither of these conditions holds, we create the next tree level. We split M_l into blocks $M_{l,0}$, $M_{l,1}, \ldots, M_{l,k-1}$ where all blocks but the last one are N_n bytes long and the last block is between

N_b and N_n bytes long. We then define:

$$M_{l+1} := \bigg\|_{i=0}^{k-1} \mathrm{UBI}(G, M_{l,i}, iN_n + (l+1)2^{112} + T_{\mathrm{msg}}2^{120})$$

and apply the above rules to M_{l+1} again.

The result G_o is the output of the tree hashing. It becomes the chaining input to the next UBI function in Skein. (Currently there are no types defined between T_{msg} and T_{out}, so G_o becomes the chaining input to the output transformation.)

As $Y_f \geq 1$ each node of the tree has a fan-out of at least 2, so the height of the tree grows logarithmically in the size of the message input.

4 Using Skein

In this section we describe some of the many ways in which Skein can be used, and which arguments are used for what data. All Skein computations contain a configuration block and end with an output transform—so we will not mention them for every use—but there are also a wealth of different options.

4.1 Skein as a Hash Function

When used as a hash function, the message type is the only optional input type used. The output of configuration UBI becomes a precomputed initial chaining value. This is the simplest use of Skein. With the variable output size it becomes a drop-in replacement for almost any existing hash function.

4.2 Tree Hashing with Skein

Implementers of tree hashing have a number of decisions to make. There are three parameters to choose: the leaf node size, the fan-out, and the maximum tree height. For efficiency, a larger leaf node size and fan-out is better; it reduces the number of nodes and thus the overhead. But large leaf nodes and high fan-out make some uses less efficient.

An implementer that needs the hash function to process data at a very high data rate can use a leaf node size of a few kilobytes and a maximum tree height of 2. This allows multiple processors to each work on its own leaf node, with one processor doing the second level of the tree. Increasing the leaf node size makes this more efficient, but it increases the amount of memory needed for buffering, and will tend to increase latency.

Limiting the tree height is useful when memory-limited devices are involved. When computing a tree hash incrementally, the implementation must store data for each level of the tree. Limiting the tree height allows a fixed allocation of memory for small devices.

Tree hashes can also be used to create a locally verifiable and/or updatable hash. In this type of application, the message data is typically stored, as well as all the nodes of the tree hash. To verify a part of the message, only that part of the message and the tree nodes that cover it have to be verified. To modify a part of the message, the tree nodes that cover the modified data have to be recomputed. This is most efficient if the leaf node size is relatively small, and the tree fan-out is low.

4.3 Skein as a MAC

To compute a MAC, the key is used as the key input, and the message as the message input.

One useful property of Skein-MAC is that a 32-bit MAC on a particular key/message pair is completely unrelated to the 64-bit MAC on the same key/message pair. This address a class of attacks where the attacker interferes with the algorithm negotiation between two parties, and convinces one to use a 32-bit MAC and the other to use a 64-bit MAC. If the shorter MAC were merely the truncation of the longer MAC, the attacker might be able to divide the keyspace in half and break the 64-bit MAC. Of course, a good algorithm negotiation protocol does not allow this attack, but we've seen this type of attack work against a number of proprietary protocols that we have analyzed in the past.

4.4 HMAC

HMAC [5, 6] represents one of the most common usages of hash functions. Skein can easily be used in HMAC mode, which will use it directly as a hash function as specified by [85].

4.5 Randomized Hashing

To use randomized hashing [42, 28], use the Nonce input to specify a differentiator for every hash computation.

4.6 Skein as a Hash Function for Digital Signatures

For digital signatures, Skein allows the option of hashing the public key as well. The message is processed into the message input and the public key into the public key input. This forces message hashes to depend on the public key, and proves that someone with access to the actual document intended to have it signed by that key. This can be relevant in systems that process signatures on documents separately from the documents. An attacker that only sees a signature cannot extract the hash and sign the document himself. Depending on the application and situation such phantom signatures might be a problem; for example, they might allow an attacker to convince an arbitrator that he was involved in developing a document because his signatures are in the audit trail. When the public key is included in the hash, the attacker needs access to the original document to sign it, or convince someone who has access to the document to hash it for his public key.

The presence of the public key in the input to the hash also serves to slow down the rate of digital signature compromise in the case of the discovery of a collision finding attack on the hash function. The attacker has to reinvest effort for every public key that it wants to attack. In contrast, when the public key is not an input to the hash, discovery of a single collision for the hash function can be used to quickly compromise a large number of signing keys.

4.7 Skein as Key Derivation Function (KDF)

Skein can be used as a KDF [45, 31, 3, 23]. To perform a key derivation, the master key is provided as the key input, and the identifier for the derived key is provided as the KDF input. The desired key size is the output size, N_o, which is part of the configuration block.

4.8 Skein as a Password-Based Key Derivation Function (PBKDF)

A Password-Based Key Derivation Function is used to derive cryptographic keys from relatively low-entropy passwords. The application stores a random seed S, asks the user for a password P, and then performs a long computation to combine S and P. This computation is deliberately inefficient, often taking something like 100 ms of CPU time. This is acceptable if a user is logging into a computer system, but an attacker that tries to guess the password has to perform 100 ms worth of computations for every password he tries. The seed S ensures that the attacker cannot precompute a table of common passwords and their results; the table would have to be recomputed for every S value.

The most commonly used PBKDFs [45, 3] use repeated hash function computations. Of course, Skein can be used in any of these constructions.

Skein also provides an alternative method for PBKDFs. The password P is provided as the key input. The seed S is repeated a very large number of times and becomes the message input. The PBKDF result is then computed using Skein with tree parameters $Y_l = 1$, $Y_f = 1$, $Y_m = 255$. The total size of the message input determines the speed of the PBKDF and can be chosen appropriately. (Existing PBKDFs typically have an iteration count of some sort that has the same function.)

This approach is not ideal with a linear hash function; the long computation on the repeated S can lose entropy with regard to the original password. The tree hashing keeps the individual UBI chains short and avoids this problem.

An even simpler PBKDF is to simply create a very long repetition of S and P; e.g., $S\|P\|S\|P\|S\cdots$, and hash that using Skein. (Any other optional data can also be included in the repetition.) This approach is not ideal with a normal hash function, as the computation could fall into a loop. But in Skein, every block has a different tweak and is thus processed differently.

4.9 Skein as a PRNG

Skein can be used as a PRNG with the same security properties as the SP 800-90 PRNGs [4] (as well as Yarrow [50] and Fortuna [35]): After generating data from the PRNG, the state no longer contains the necessary information to recover that data.

The Skein-PRNG state S consists of N_b bytes. If an application requests N random bytes, the PRNG computes the Skein output function using the state S as the chaining input and produces $N + N_b$ bytes of output. The first N_b bytes of output become the next state for the next request; the rest of the output bytes are given to the application. Once this function completes and the old S state is overwritten, the PRNG can no longer recover the random bytes produced for the application.

To reseed the PRNG with seed data D, set the state to the Skein hash of $S \| D$ (using the natural output size). The initial seeding of the PRNG is done by setting the state to all zeroes and performing a reseed with the provided seed data.

Skein-PRNG is fast; it can produce random data at the same speed that it hashes data. For small requests, Skein-PRNG has to process a minimum of two Threefish encryptions; it is more efficient to get larger blocks of random bytes in one request and buffer the result.

4.10 Skein as a Stream Cipher

To use Skein as a stream cipher, supply the key to the key input and the nonce (that selects the key stream to generate) to the nonce input. By convention, since the length of the desired key stream is not known in advance, set the output size in the configuration value (see Table 7) to $2^{64} - 1$. Implementations can then compute any part of the key stream as desired. For encryption and decryption, the key stream is XORed with the plaintext or ciphertext.

There is a fundamental difference between Skein-PRNG and using Skein as a stream cipher to generate random bits. The outputs of a PRNG are typically not reproducible. Skein-PRNG actually does work to ensure that once an output has been produced, the PRNG state no longer contains the necessary information to reconstruct the output. A stream cipher creates reproducible random data. Depending on the application, one or the other might be desirable.

An application that needs random access to a large random string can use the Skein stream cipher mode in two ways. It can use a single nonce and selectively generate output blocks, or it can include a counter in the nonce and generate a fixed size block for each nonce value. In general, we recommend the second approach as it does not require a new API for selectively generating parts of the output string, and thus is easier to implement using an existing Skein implementation.

4.11 Personalization

All Skein applications (except the PRNG output production) can be personalized with the personalization input. We recommend that all application designers seriously consider doing this; we have seen many protocols where a hash that is computed in one part of the protocol can be used in an entirely different part because two hash computations were done on similar or related data, and the attacker can force the application to make the hash inputs the same [53, 34]. Personalizing each hash function used in the protocol summarily stops this type of attack.

When using the personalization input, we recommend that applications use a unique string that starts with a date followed by an email address. The date consists of 8 digits in YYYYMMDD format (Gregorian calendar); this is immediately followed by a space, an email address owned by the creator of the application on the date specified, and a space. After the space, the creator of the application can use any data to distinguish different applications and uses.

For example, the personalization string for the application FOO might be the UTF8 Unicode string:

```
20081031 somebody@example.com FOO/bar
```

where “bar” is the personalization within the application.

This convention allows anybody to generate unique personalization strings that are distinct from all other personalization strings. To support all languages, the string is a UTF8-encoded Unicode string[2].

[2] For readers unfamiliar with UTF8 and Unicode: an ASCII string with all characters < 128 is a valid UTF8-encoded Unicode string.

Alternatively, implementors can generate a 16-byte random value using a high-quality random number generator, and start all their personalization strings with that fixed random value.

4.12 Choosing the Output Size

For any of these Skein applications, there can be situations in which the desired output size is not known in advance. This can be resolved in two ways. The simplest way is to compute the result using the natural output size and use this as key to the stream cipher mode to produce the desired output size. Alternatively, applications can set $N_o = 2^{64} - 1$ and use only as many of the output bytes as they need. In general, we recommend against this second approach, as the leading bytes of different output sizes are the same. Furthermore, it requires a non-standard implementation that can produce only part of the specified output.

4.13 Threefish as a Block Cipher

Threefish can be used as a normal block cipher in any of the well-known block cipher modes [35].

Threefish decryption is generally slower than encryption due to the MIX function having less parallelism in the decryption direction. Some block ciphers modes use both encryption and decryption (e.g., CBC) but others use only encryption (e.g., CFB and OFB). Since in most applications decryption happens more often than encryption, when using Threefish as a standalone cipher in a mode that requires decryption, it might be useful to switch the encrypt/decrypt direction. But this is a minor point, given the raw speed of Threefish.

Several recent block cipher modes, such as Offset Codebook (OCB) [97], turn a plain block cipher into something similar to a tweakable block cipher using a value added to both the plaintext and the ciphertext. We believe that a native tweakable block cipher, like Threefish, will lead to newer, more efficient modes, where the tweak value is used directly.

Other modes of operation will likely benefit from the extended input space of tweakable block ciphers (plaintext or ciphertext *plus* tweak), compared to conventional block ciphers (plaintext or ciphertext *only*). For example, the Counter–Cipher Feedback (CCFB) mode [68] uses a conventional block cipher for authenticated encryption. We are working on a variant of that mode, providing more efficient authenticated encryption at the same level of security, when employing a tweakable block cipher instead of a conventional one.

Most block ciphers modes are only secure up to the birthday bound; thus, we can expect most uses of AES to start failing after processing 2^{64} blocks. In general, to achieve a security level of n bits it would be nice to have a block cipher with a block size of $2n$ bits. Threefish has a large enough block size to eliminate all collision-style attacks and provide high security even when processing large amounts of data.

5 Skein Performance

5.1 Software Performance

Skein is designed to be fast on 64-bit CPUs. Table 8 give a summary of the speed measurements we have made for large messages, in 64- and 32-bit mode on an Intel Core 2 Duo CPU, in assembly language and C.

	Skein-		
	256	512	1024
64-bit ASM	7.6	6.1	6.5
64-bit C	9.2	6.5	12.3
32-bit ASM	32.8	32.5	37.5
32-bit C	35.8	40.1	49.0

Table 8: Summary of skein speeds (clocks/byte).

The following series of tables gives performance figures for Skein-256, Skein-512, and Skein-1024 with a variety of message sizes. All measurements were taken on the NIST reference platform: an Intel Core 2 Duo CPU running Windows Vista, using the Microsoft Visual C Studio 2008 compiler. There are several different levels of loop unrolling for each version of Skein, and each table lists the result from the fastest version of the code, which is not always the fully unrolled version.

The times to hash 1 and 10 bytes are the same: each is less than one block for all block sizes, and Skein requires two Threefish calls to hash a one-block message. A 100 byte message requires five Threefish calls for Skein-256 (four for the block and one for the output transform), three Threefish calls for Skein-512, and only two for Skein-1024.

For longer message lengths—1000, 10,000, and 100,000 bytes—Skein is making many Threefish calls and the true performance of the algorithm can be measured. It should be noted that these powers of ten are not multiples of the native block size, so the “rounding” error there affects the results somewhat.

64-bit Implementations. Table 9 gives performance figures for Skein, hand-coded in assembly language. Table 10 gives preliminary performance figures for Skein, coded in C.

	Message Length (bytes)					
	1	10	100	1000	10,000	100,000
Skein-256	666	65	14.3	8.2	7.6	7.6
Skein-512	1068	107	15.0	7.0	6.2	6.1
Skein-1024	1902	191	19.3	7.8	6.7	6.5

Table 9: Skein speeds (clocks/byte) in ASM on a 64-bit CPU.

	Message Length (bytes)					
	1	10	100	1000	10,000	100,000
Skein-256	774	77	16.6	9.8	9.2	9.2
Skein-512	1086	110	15.6	7.3	6.6	6.5
Skein-1024	3295	330	33.2	14.2	12.3	12.3

Table 10: Skein speeds (clocks/byte) in C on a 64-bit CPU.

For comparison, Table 11 lists the performance of the SHA family in C on an Intel Core 2 Duo CPU [38, 39]. At 6.5 clocks/byte, Skein-512 is more than twice as fast as SHA-512’s 13.3 clocks/byte on the NIST reference platform CPU.

	Message Length (bytes)					
	1	10	100	1000	10,000	100,000
SHA-1	677	74.2	14.0	10.4	10.0	10.0
SHA-224	1379	143.1	27.4	20.7	20.1	20.0
SHA-256	1405	145.7	27.6	20.7	20.1	20.0
SHA-384	1821	187.3	19.6	13.7	13.4	13.3
SHA-512	1899	192.5	20.6	13.8	13.4	13.3

Table 11: SHA speeds (clocks/byte) in C on a 64-bit CPU.

All of these Skein numbers are based on an implementation of Skein optimized for speed. It is possible to trade speed for code size, allowing Skein to run on platforms with limited memory, as shown in Table 12 for assembly code. Similar trade-offs exist for the C code. The code size shown is in bytes, for the Skein block processing function. The speed is given in CPU clocks per byte, and the final column indicates how many rounds of the block cipher are unrolled in the Skein block processing function. In general, the looping versions of the code are all fairly close to the speed of the fully unrolled version, which is always the fastest. Among the looping versions, the speed difference between different amounts of unrolling is very minimal—typically not even visible when rounded to the nearest tenth of clocks/byte—so unrolling 8 rounds seems to be the best option when code size is critical. The Skein block function could also be coded with even les s memory by not unrolling the Threefish algorithm at all, and looping it 72 or 80 times. We have not implemented that variant.

	Code Size	Speed	Unrolled Rounds
Skein-256	2323	7.6	72
Skein-256	1288	7.8	24
Skein-256	664	7.8	8
Skein-512	4733	6.1	72
Skein-512	2182	6.4	24
Skein-512	1074	6.4	8
Skein-1024	11817	6.5	80
Skein-1024	7133	6.9	40
Skein-1024	3449	7.1	16
Skein-1024	2221	7.0	8

Table 12: Code size/speed trade-offs on 64-bit CPUs in ASM.

The sizes of the API functions—Init, Update, and Final—are not included in Table 12, since they are all in C and do not have any significant speed/size trade-offs. The combined code size of these API functions is roughly 1000 bytes for each Skein block size, varying slightly depending on how much function inlining the compiler chooses to do.

32-bit Implementations. On a 32-bit CPU, performance is slower; see Tables 13 and Table 14. It should be noted that in some cases, other compilers (e.g., GCC) give slightly faster results for 32-bit code.

	Message Length (bytes)					
	1	10	100	1000	10,000	100,000
Skein-256	2310	230	54.1	34.1	32.9	32.8
Skein-512	4460	484	65.3	35.5	32.5	32.5
Skein-1024	9730	974	97.8	42.7	37.5	37.5

Table 13: Skein speeds (clocks/byte) in ASM in 32-bit mode.

	Message Length (bytes)					
	1	10	100	1000	10,000	100,000
Skein-256	2544	257	60.0	38.1	35.8	35.8
Skein-512	5508	549	81.2	44.3	40.1	40.1
Skein-1024	12624	1262	126	55.4	49.0	49.0

Table 14: Skein speeds (clocks/byte) in C in 32-bit mode.

For comparison, Table 15 lists the performance of the SHA family in C in 32-bit mode [38, 39]. SHA-1, SHA-224, and SHA-256 are optimized for 32-bit words, and are faster on this platform. SHA-384, SHA-512, and Skein are optimized for 64-bit words, and are slower on a 32-bit CPU. But Skein-512 is still faster than SHA-512.

	Message Length (bytes)					
	1	10	100	1000	10,000	100,000
SHA-1	716	71.6	15.1	10.4	10.0	9.9
SHA-224	1522	152.2	29.1	21.6	20.1	20.9
SHA-256	1522	153.5	29.5	21.6	20.9	20.9
SHA-384	5747	574.7	58.8	42.9	41.9	41.4
SHA-512	5851	586.4	60.2	43.0	41.9	41.4

Table 15: SHA speeds (clocks/byte) in C in 32-bit mode.

8-bit Implementations. Table 16 gives Skein's speed, using compiled C code, on an Atmel AVR® 8-Bit RISC processor. The implementation unrolls the code to 8 rounds. These speed numbers are for long messages.

	code size (bytes)	clocks/ block	block time @ 16 MHz	large-message throughput
Skein-256	22,500	208k	13 ms	2.5 kB/s
Skein-512	46,300	341k	27 ms	2.4 kB/s
Skein-1024	91,500	940k	59 ms	2.2 kB/s

Table 16: Skein speed in C on an 8-bit CPU.

Table 17 contains our ASM speed estimates on the same 8-bit CPU. The corresponding results are slightly more than ten times faster than the C versions, probably due to an inefficient implementation of the 64-bit rotation in the compiler's C library. These assembly estimates are optimized for speed, not for code size. It would also be possible to cut the code size in half (or better) by sacrificing some performance. The last row is an implementation that exploits the fact that the 256-bit state fits entirely in the 32 registers of the AVR CPU.

	code size (bytes)	clocks/ block	block time @ 16 MHz	large-message throughput
Skein-256	4,800	19k	1.2 ms	26 kB/s
Skein-512	8,300	37k	2.3 ms	28 kB/s
Skein-1024	13,200	80k	5.0 ms	26 kB/s
Skein-256		9.5k	0.6 ms	54 kB/s

Table 17: Skein speed estimates in ASM on an 8-bit CPU.

5.2 Hardware Performance

ASIC Implementation. The Skein compression function consists of five steps:

1. Loading the key and plaintext,
2. Building the Threefish key schedule,
3. Executing 72 or 80 rounds for Skein, with key injections every 4 rounds,
4. Doing the feed-forward step,
5. Saving the result.

This description allows us to estimate the gate cost and performance of the Skein compression function implemented by an ASIC. We provide this estimate for Skein-512 only. Estimates for Skein-256 and Skein-1024 are analogous.

The gate count for any implementation is primarily determined by step 3, so we will estimate this first: A Threefish-512 round consists of four parallel MIX operations and a permutation. A MIX operation consists of a 64-bit XOR, a 64-bit rotate, and a 64-bit add. A 64-bit XOR can be implemented in 192 gates. A 64-bit add can be implemented in about 800 gates. This means a MIX costs about 1000 gates. The delay through this circuit is conservatively about 1 nanosecond, using a 65 nm CMOS process.

Threefish defines distinct rotation constants for eight rounds, with distinct rotation constants for each MIX. Hence, it is necessary to implement 32 different MIX circuits for Threefish and Skein. Since the permutations can be implemented by simply routing the internal state appropriately, this means that the Threefish round functions collectively require about 32K gates.

Threefish-512 requires storage for its internal state and the feed-forward value. Each of these can be implemented with 512 bit flip-flops at about 5K gates each. The Threefish-512 key schedule requires 768 bits of storage, including the key (chaining variable), tweak, and overall parity words. This can be implemented using 768 bits of flip-flop, costing approximately 8K gates. The multiplexers for loading and shifting all these flip-flop bits values add about another 8K gates.

The Threefish-512 subkey injection can be implemented using eleven 64-bit adders (eight adders for the key words, two for the tweak words, and one for the injection count), which we estimate at approximately 9K gates. Computing the parity over the key and tweak words requires 512 two-input XOR gates (2K gates), and we assume that the key schedule values are rotated using shift registers after each key injection.

This gives an estimated gate count of roughly $32 + 5 + 5 + 8 + 8 + 9 + 2 = 69$K gates. The actual gate count will probably be somewhat higher due to additional routing area required by the fixed rotations, so the overall equivalent chip area might be closer to about 80K gates. The delay through the circuit would be 8 nanoseconds, which we round up to 10 nanoseconds (100 MHz) to be conservative.

Skein-512 simply iterates its compression function to hash a string longer than one block, and would require 10 clocks per block (9 clocks for 72 rounds, plus one for setup), or 10M blocks/second. This gives a total throughput of roughly 5 Gb/s. It should be noted that a custom layout, particularly of the adders, could probably increase this performance by more than factor of two.

At the time of writing, the fastest Intel Core 2 CPU can be clocked at 3.4 GHz. At 6.1 cycles/byte, each core can hash data at around 500 MB/s or 4 Gb/s. Thus, ASIC hardware is not much faster than a fast CPU core, although it might be far cheaper and use far less power. At first glance, it is surprising that a software implementation would be that fast, but modern CPUs use highly specialized layouts and cutting-edge chip technologies, whereas ASICs are often made with standard cell libraries and older (cheaper) chip technologies.

Obviously, a company like Intel could use the same chip technology found in CPUs to make faster Skein hardware, but we doubt that will ever happen.

FPGA Implementation. We are building a reference FPGA Skein implementation; our technical report will be available before the NIST Hash Workshop in February 2009.

5.3 Threefish Software Performance

Table 18 gives preliminary relative performance figures for Threefish—both encryption and decryption—in C, on the NIST Reference platform CPU in 64-bit mode. These numbers are for Skein using Threefish encryption versus Skein using Threefish decryption. That is, both operations include the plaintext feed-forward and the key schedule, so the encryption number here is identical to the Skein performance. The point is to show the relative slowdown of using decryption. Encryption-only and decryption-only versions would each be slightly faster.

	Speed	
	Encrypt	Decrypt
Threefish-256	9.2	13.5
Threefish-512	6.5	7.7
Threefish-1024	12.3	not implemented

Table 18: Threefish speeds (clocks/byte) in C on an Intel Core 2 Duo CPU.

Of course, Threefish would be faster in ASM.

5.4 The Word Size As a Tunable Parameter

All versions of Skein are specified with 64-bit words. The word size can be seen as a tunable parameter; we can define a Skein variant with 32-bit words. This variant would run much faster on 32-bit CPUs, but significantly slower on 64-bit CPUs.

At this point, we have not searched for rotation or permutation constants for a 32-bit variant, nor have we analyzed it to determine how many rounds would be required for security. However, given the knowledge obtained from the 64-bit variants, this would not be complicated.

6 Skein Security Claims

6.1 Basic Security Claims for Skein

Skein has been developed to be secure for a wide range of applications, including but not limited to digital signatures, key derivation, pseudorandom number generation, and stream cipher usage. Skein supports personalized and randomized hashing. Under a secret key, Skein can be used for message authentication and as a pseudorandom function.

Below, we write n for the state size, and m for the minimum of state and output size. We claim the following levels of security against standard attacks[3]:

- First preimage resistance up to 2^m.
- Second preimage resistance up to 2^m.
- Collision resistance up to $2^{m/2}$.
- Resistance against r-collisions up to roughly $\min\{2^{n/2}, 2^{(r-1)m/r}\}$. (An r collision consists of r different messages $M_1, \ldots, M_r$ with $H(M_1) = \cdots = H(M_r)$.)

Furthermore, we make the following security claims for Skein:

- When used as a message authentication code (MAC) or as a pseudorandom function, either via the HMAC construction or by using Skein's native MAC/PRF support under a secret key, we claim resistance to key recovery, forgery, or distinguishing attacks up to $\min(2^{n/2}, 2^m)$.
- For randomized hashing, we claim security up to 2^m against the following eTCR attack scenario of [42]: The attacker chooses a message M_1 and receives r_1 and $H_{r_1}(M_1)$, the randomized hash of M_1. Here r_1 is an n-bit random value not chosen by the adversary. Now the attacker has to find an r_2 and a message M_2 with $H_{r_2}(M_2) = H_{r_1}(M_1)$.
- Old Merkle-Damgård hash functions suffer from a length extension property: Given $H(M)$, without knowing anything about M except for its length, it is feasible to compute an extension E and the hash $H(M||E)$. This kind of attack succeeds with probability 1 for SHA-256 and SHA-512, for example.

[3] Our claims regarding collision resistance, pseudo-collision resistance and corresponding near misses follow Rogaway's formalism [96].

> Skeins UBI mode defends against length extension. If the entropy of M is sufficiently large, such that the adversary cannot guess M, the probability of success for a length extension attack is roughly 2^{-m}.

In addition to exact collisions, preimages and second preimages for the hash function, near misses are also relevant. For example, a near-collision with Hamming-weight $h \geq 1$ consists of two messages $M \neq M'$ with $H(M) \neq H(M')$, where $n - h$ of the bits in $H(M)$ and $H(M')$ are the same, and h bits differ.

Computing a near miss may be simpler than computing an exact hit, but if it is too simple, this indicates a weakness. For Skein, we claim that finding a near miss (i.e., a near-collision, a near-preimage or a near-second-preimage) is no more than

$$\binom{n}{h} = \frac{n!}{h! \cdot (n-h)!}$$

times faster than the corresponding exact hit.

6.2 The Security of Skein's Compression Function and the Threefish Block Cipher

We make the following claims about the block compression function inside Skein, as used by the UBI mode. Following an old tradition from cryptography, attacks which deal with the compression function rather than the hash function are marked by the prefix "pseudo."

- Pseudo first-preimage resistance of 2^n, where n is the size of the chaining value.
- Pseudo second-preimage resistance of 2^n, where n is the size of the chaining value.
- Pseudo-collision resistance of $2^{n/2}$, where n is the size of the chaining value.
- Resistance against r-pseudo-collisions up to roughly $2^{(r-1)n/r}$.

For the collision resistance of UBI, we restrict ourselves to collisions in which the starting positions in the starting tweaks are identical, where n is the size of the chaining value. This provides an additional line of defense: We claim security against pseudo-collisions in general, but even the ability to find pseudo-collisions would not allow an adversary to break Skein, if these colliding inputs for the Skein compression function have different tweaks.

Security against near misses for the compression function may degenerate by the same factor $\binom{n}{h}$ we claimed for near misses against the Skein hash function.

Furthermore, we claim Threefish to be secure against all standard attacks against a tweakable block cipher: chosen-plaintext attacks, related-key attacks, chosen-tweak attacks, and so on.

6.3 Security Proofs

The claims made about Skein's security are backed by proofs [9]. Here we briefly explain what these proofs mean and provide.

The base (also called atomic) primitives underlying Skein are the tweakable block cipher Threefish and its derived compression function. Skein is built on top of these. A proof that Skein possesses some security property S is a proof of a statement of the form: "If the atomic primitive has security property A, then Skein is guaranteed to have security property S." The proof takes the form of a reduction that, given an attacker violating property S of Skein, constructs an attacker violating property A of the atomic primitive. We will be providing such proofs for various choices of S.

It should be understood that a proof of security does not say that it would be impossible to find attacks violating security property S for Skein. What it says is that it would be impossible to find such attacks without uncovering attacks violating security property A of the atomic primitive. The proof transfers confidence from the atomic primitive to Skein. It validates the mode of operation, meaning the higher-level design. It says there are no flaws in this design. The practical consequence is that cryptanalysis can be confined to the atomic primitives. There is no need to attempt to attack Skein itself. One might as well invest effort in attacking Threefish and the compression function.

The first and most basic property about which we have proofs is collision resistance. However, this isn't the only security property we support via proofs. A look at the contemporary usage of hash functions makes it clear that they are used in ways that call for security properties well beyond, and different from, collision resistance. In particular, hash functions are used for message authentication (e.g. HMAC [6, 5]) and as pseudorandom functions (PRFs) in key derivation. (These usages refer to keyed versions of the hash function.) They are also used to instantiate random oracles in public-key cryptography schemes. We believe this type of usage will continue, and modern hash functions should support it. This is the design philosophy that underlies Skein.

We approach providing provable support for these additional properties by showing that the mode of operation underlying Skein is MPP (Multi-Property Preserving) [10]. This means that a number of different security attributes, if possessed by the atomic primitive, are guaranteed to be possessed by Skein. The first such property is collision resistance. The second is pseudo-randomness, as a consequence of which we obtaine provable support for the use of keyed Skein as a KDF and MAC. The third is indifferentiability from a random oracle.

One of the most widespread current usages of hash functions is for HMAC [6, 85]. This use is supported by proofs of security for the current generation of hash functions that use Merkle-Damgård mode [6, 5]. We expect that any future hash function will continue to be utilized in HMAC mode and that such use should continue to be supported by proofs of security. We supply these proofs.

We also provide provable support for the use of Skein as a PRNG and as a stream cipher.

Although the outcomes of proofs in this document are discussed in a qualitative sense, the theorems and proofs in [9] provide concrete reductions; that is, a concrete quantitative analysis of the relations between the resources of an adversary, and the adversarial advantage.

Figure 19 summarizes the provable security results regarding Skein; For each property, we indicate the assumption on the atomic primitive under which it is established. We now discuss these items in more detail. The formal definitions, result statements, and proofs that back up the claims made below will be provided in a supporting document that will be available before the NIST Hash Workshop in February 2009 [9].

Collision resistance. We prove that if the compression function is collision resistant, then so is Skein. (Referring to the above discussion, here S is the collision resistance of Skein and A is the collision resistance of the compression function.) The implication is that it is no easier to find collisions for Skein than for its compression function. Given that (strengthened) Merkle-

Skein Property / Mode	Assumption on Atomic Primitive
Hash (collision resistance)	The compression function, C, is collision resistant
PRF	Threefish is a (tweakable) PRP
KDF	Threefish is a (tweakable) PRP
MAC	Threefish is a (tweakable) PRP
Indifferentiability from random oracle	Threefish is an ideal (tweakable) cipher
HMAC	Threefish is a (tweakable) PRP
PRNG	Threefish is a (tweakable) PRP
Stream cipher	Threefish is a (tweakable) PRP

Table 19: Summary of provable security attributes of Skein.

Damgård [27, 75], used in the SHA family, is backed by a similar security guarantee, such a guarantee would seem to be a necessary requirement for a new hash function. We are asserting that we can provide this.

PRF, MAC, and KDF. We prove that if Threefish is a tweakable PRP (pseudorandom permutation), then Skein is a PRF. It is important to understand that we are referring, in this context, to the keyed version of Skein. The PRF property is that the input-output behavior of keyed Skein should look like that of a random function to an attacker *who is not given the key.* This proof supports the usage of keyed Skein for key derivation (KDF). It also supports the use of keyed Skein as a MAC. This is true because any PRF is a secure MAC [8].

The PRF property reflects the increased versatility of Skein compared to the SHA family. The functions in the latter family are not PRFs when keyed in the natural way; namely, via the initialization vector. This is because of the extension attack.

We highlight an attractive feature of the proof of PRF security. Namely, the assumption made pertains to the (tweakable) block cipher rather than to the compression function. Additionally, this is the standard assumption on a tweakable block cipher: that it is a PRP. Indeed, in the case of other modes such as EMD [10] that are PRF preserving, the assumption is that the compression function is a PRF, which relies on the underlying block cipher being a PRF when keyed through the message rather than the key port. The difference in Skein arises because the compression function runs the block cipher in Matyas-Meyer-Oseas mode.

We emphasize that we provide provable support for the use of keyed Skein as a MAC. This is by dint of the fact that we show keyed Skein is a secure MAC, under the assumption that Threefish is a PRP. (This in turn is because, as indicated above, under this assumption, keyed Skein is a PRF, and any PRF is a secure MAC.)

A novel feature of Skein in these modes is the variable output length. The desired output length is one of the inputs to the hash function. Skein has been designed so that its output values are independent for different values of this output length parameter, even if other inputs (such as the message) are the same. This attribute of Skein is also supported by the security proofs. We define the (new) concept of a VOL (Variable Output Length) PRF. This is what the proofs show Skein to achieve, under the assumption that Threefish is a PRP.

Keyed Skein is a fast alternative to HMAC-Skein with regard to providing a PRF and secure MAC. To support legacy applications, however, we will also support HMAC-Skein via proofs.

Indifferentiability from a random oracle. We prove that the Skein mode of operation preserves indifferentiability from a random oracle. This has, since [24, 10], become an important requirement for hash functions, due to their use for instantiating random oracles.

What the results say is that if we replace Threefish with an ideal block cipher, the resulting hash function produced by the Skein mode of operation behaves like a random oracle. Technically, it is indifferentiable from a random oracle. Indifferentiability [72, 24] is a technical term underlain by a formal definition. If a function is indifferentiable from a random oracle, it means we can securely replace a random oracle with this function in most (not all) usages of the random oracle.

This can be viewed as saying the Skein mode of operation has no structural weaknesses. It is evidence that attacks that differentiate it from a random oracle, such as the extension attack, won't work.

We should, however, add a word of warning and explanation. The result pertains to the mode of operation, not to the block cipher. In the proof, the latter has been replaced by an ideal block cipher. The subtle point here is that there is no formal notion or assumption that we can state to capture "Threefish is, or approximates, an ideal block cipher." This result is different from the other results discussed above. It is, for example, perfectly meaningful to say that Threefish is a PRP. We emphasize that the subtleties associated with indifferentiability are not peculiar to our results, but instead are endemic to the notion as a whole. They are, and will be, present for any hash function for which a proof of indifferentiability from a random oracle is supplied.

All this notwithstanding, the general consensus in the community is that indifferentiability buys you something. It is just difficult to *formally* say exactly what.

Support for HMAC mode. Current hash functions are used in HMAC mode to obtain a MAC or a PRF. The widespread standardization and use of HMAC means this represents a large and important domain of hash function usage. (HMAC is standardized via an IETF RFC [63], a NIST FIPS [85], and ANSI X9.71 [1]. It is in IEEE 802.11. It is implemented in SSL, SSH, IPsec, and TLS, among other places.) It is thus important that any new hash function continue to support usage in HMAC mode.

The issue this raises with regard to proofs is as follows. For hash functions that use Merkle-Damgård [27, 75] mode (in particular the MD and SHA families), HMAC mode is supported by proofs [6, 5] that arguably played an important role in the widespread and continuing adoption of HMAC. Current support for HMAC in this domain is represented by [5], which showed that HMAC with a Merkle-Damgård hash function is a secure PRF (and hence MAC), assuming that the compression function is itself a secure PRF. If Skein is to become a replacement for current hash functions, it is important that we provide a similar provable guarantee for its usage in HMAC mode. But since our underlying iteration method is not Merkle-Damgård, the previous proofs do not apply.

Our contribution in this regard is to supply new proofs. These show the analog of the above-mentioned result. Namely, if the compression function is a PRF, then so is HMAC-Skein. This means that Skein has the same provable guarantees in HMAC mode as existing hash functions.

As a result, there are two different modes of operation in which Skein can provide a PRF or MAC: HMAC mode and Skein's native keyed mode as discussed above. The latter is faster. However, the former needs to be supported for legacy reasons.

PRNG and stream cipher. The target security property for a stream cipher is that of [19, 111]: given a random seed, the output should be computationally indistinguishable from random. The goal for the PRNG is that it should be forward-secure, as defined by [11]. We prove both these properties under the assumption that Threefish is a PRP.

6.4 Security Above the Birthday Bound

There has recently been significant attention drawn to new security models for hash functions, whereby additional properties are required to defend against attacks with greater complexity than $2^{(n/2)}$. For example, Joux found that if an attacker can expend sufficient work to find a collision in the internal state of an MD hash function, the attacker could amplify that attack to find a large number of additional collisions. Joux called this a "multi-collision" attack [44].

Similarly, we found it is possible to exploit collisions on the internal state of a hash function to find second preimages faster than one might naively otherwise expect [49], and we show how to exploit collisions on the internal state of a hash function to mount what we call "herding" attacks [48].

These "attacks above the birthday bound" are unique for several reasons. First, they target traditionally non-standard properties of the hash function. For example, whereas previous research focused on measuring how hard it would be for an attacker to find a *single* collision, these new works *begin* with the assumption that an attacker can find one collision, and then ask what else an attacker might be able to do with it. Second, given the nature of these attacks, we are currently forced to argue a hash function's resistance against them using *ad hoc* means, rather than proofs of security.

These attacks above the birthday bound are theoretically interesting, but unimportant in practice. Designers who desire n bits of security should use a hash function with at least $2n$ bits of state. This is already common practice, and it pushes these type of attacks beyond the capabilities of any attacker. The Skein state sizes are large enough to achieve this for all commonly used security levels.

6.5 Tunable Security Parameter

Although the number of Threefish rounds is specified for all Skein variants, this represents a tunable security parameter. It would be straightforward to increase or decrease the number of rounds by multiples of four. To increase or decrease the number of rounds by a number that is not a multiple of four, we would want to investigate changing the rotation constants and the word permutation as well.

7 Implementing Skein

7.1 Software Implementations

7.1.1 Threefish

In software, most of the work of Threefish is in the MIX function. For that reason, we designed it to be relatively easy to implement. MIX is optimized for 64-bit CPUs, and implementing the MIX function on those platforms is trivial.

On a 32-bit CPU, the MIX function requires a 64-bit rotation and addition. The 64-bit rotations are typically built out of four 32-bit shifts and some mask/combine operations. On the x86 architecture, the SHLD instruction implements half of a 64-bit rotation. The 64-bit additions are typically built from two additions, the second one using the carry bit from the first one.

On an 8-bit CPU, the 64-bit addition must be built from eight 8-bit additions. The rotation is harder; most 8-bit CPUs do not have a barrel shifter and are limited to 1-bit rotations. A 64-bit rotation is typically implemented as a byte re-order and between zero and four 1-bit left or right rotations. Each 1-bit rotation can be implemented as eight or nine 8-bit rotate-through-carry instructions.

The Threefish round functions can be rolled into a nested loop or a single loop, or they can be fully unrolled. The smallest and slowest option is a double loop: the outer loop for the rounds and the inner loop for the MIX functions in a round. For fast implementations, Threefish is typically unrolled to 8 rounds or fully unrolled. Once 8 rounds are unrolled, the rotation constants become fixed—they repeat every 8 rounds—and can be embedded in the code itself.

The key schedule can be implemented in several ways. The simplest way is to store the expanded key and tweak, and compute each subkey when needed. When used in Skein, Threefish only uses a key to encrypt one block, so this is also efficient. If the same key is used many times—if Threefish is encrypting a large block of text—the subkeys can be fully precomputed. Note that different subkeys can use the same sum of a tweak word and a key word. Implementations can precompute those values, or store the results the first time they are computed.

Small memory implementations might not want to store the entire tweak. When Threefish is used in Skein for small messages (and without tree hashing or bit padding), most of the tweak is zero. The first few bytes contain the message length so far, and the last byte is one of two or three values. Storing just a few bytes is enough to reconstruct the tweak value, and the necessary tweak words can be computed on the fly when they are needed.

When Threefish is used for data encryption rather than hashing, decryption is slower than encryption. As data is typically decrypted more often than it is encrypted, implementations might want to swap the two directions: using what we describe as encryption for decryption and vice versa. There are no security implications in making this change.

7.1.2 UBI

Unless specifically necessary, we recommend that implementations support only inputs that are an integral number of bytes. In most circumstances, odd-bit-length inputs are not used, and including the option merely complicates the coding and testing. It is easy to not support odd bit lengths; just ignore the issue. There is no bit padding to apply, and the BitPad bit in the tweak is left at zero. We stress that this is not a security issue; an implementation for arbitrary bit lengths is as secure as implementation supporting only integral numbers of bytes.

Implementations that allow messages to be processed incrementally need to buffer one block's worth of data. This is because a block cannot be processed until it is known whether it is the last block of the message. High-speed implementations might want to create a single loop that processes multiple blocks of data. This avoids the overhead of a function call for every block.

To process a block, the implementation needs to store the following information:

- The chaining value/Threefish key

- The current state of the Threefish encryption
- The message block to be XORed at the end
- The tweak, or information to allow it to be constructed on the fly

Thus, UBI requires slightly more than $3N_b$ bytes of memory. Low-memory implementations should consider using Skein-256, as it can be implemented in approximately 100 bytes of RAM (assuming the messages are not too long).

On modern operating systems, memory areas are frequently mapped in such a way that they are accessible from multiple contexts. For example, a kernel mode function might read data from memory in a user mode process; another thread in that process could be modifying the memory at the same time that the kernel mode thread was reading the data.

This opens up a possible line of attack. An implementer might be tempted not to buffer the message block but read it twice from memory: once to start the encryption and once for the feed-forward XOR. If another thread modifies the message block between these two operations, it can inject a chosen difference in the chaining state—something that is normally not possible. We do not know whether this leads to an attack—it seems difficult to exploit in Skein—but it violates the properties that our security proofs depend upon. As a rule of thumb, a cryptographic algorithm should only read its inputs once, which is how the Skein code provided to NIST operates.

7.1.3 Skein

Any implementer of Skein has to choose which options to enable. The simplest implementations only implement straight hashing with a fixed output size. After that, the most useful options to support are probably:

- Variable output sizes (in byte increments) up to one block
- Longer outputs
- Key input for a MAC
- PRNG
- Personalization

We expect that the public-key field, key derivation, and tree hashing will be used less frequently.

Skein defines output sizes of arbitrary bit length, but we recommend that implementations restrict themselves to whole bytes. There are specific uses for odd bit lengths (e.g., elliptic curves) and the odd bit length provides a symmetry with the arbitrary bit length of the inputs, but in practice, we rarely see arbitrary bit length values being used.

7.2 Hardware Implementations

7.2.1 Threefish

The core of Threefish is the MIX function. In hardware, this is straightforward to implement. To achieve high performance it is important to use a fast-carry adder and not a ripple-carry adder.

Ripple-carry adders are very slow in the worst case; the carry ripples from the least significant bit to the most significant bit, which limits the maximum clock frequency. There are well-known techniques for fast carry propagation in adders, and these should be used for speed-sensitive implementations.

The rotations and word permutations do not require any gates, but they do take up routing space.

The most natural way to implement Threefish is to either implement 8 rounds, or the full 72 or 80 rounds. An implementation that tries to implement only 1 or 4 rounds needs to accommodate different rotation constants in each MIX, leading to a number of multiplexers.

The key schedule can be implemented in several ways. The simplest one is to store the extended key and extended tweak in two shift registers and clock the shift registers once for each subkey. Note that the final state of the shift registers can be directly computed, so implementations that want to perform decryption can efficiently generate the subkeys in reverse order.

7.2.2 UBI

In hardware, UBI is implemented like any other block chaining mode. There are no special considerations, other than the need to buffer the last input block until it is known whether this is the last block of the message or not.

7.2.3 Skein

For high-speed implementations, the output transform is a problem. If the core Threefish implementation can barely keep up with incoming data, there is no time to compute the output transform between two messages. Implementations have to ensure that the core is twice as fast as the maximum data rate, have two Threefish implementations (one for the data and one for the output transform), or reduce throughput when short messages are processed.

8 Skein Design

8.1 Design Philosophy

There were several principles that we kept in mind throughout the design process.

Simplicity. Simplicity is important in any cryptographic primitive: the easier an algorithm is to understand, the easier it is to analyze. And the easier it is to analyze, the more confidence the cryptographic community has in its analysis. Because of this, simplicity was one of our core design goals. We wanted a design that could be easily explained and remembered.

Security per clock cycle. In all our design trade-offs, security per clock cycle on a 64-bit CPU was the primary measure. This is a method for evaluating algorithms that we developed previously [101], and have used in the design of Twofish [100], Helix [36], and Phelix [105].

Implementability on a wide range of platforms. Any standardized hash function ought to run on as many different platforms as possible. Most critical here are low-end platforms: smartcards, embedded systems, sensor network motes, RFID-tags, and so on. To ensure implementability on these low-end systems, we avoided hardware-expensive operations—such as multiplications—and large constant tables. We also ensured that Skein and Threefish could be implemented in very small code size and with very limited RAM.

Of course, we did not just focus on low-end platforms. We wanted Skein to perform well on modern 64-bit CPUs. Skein employs simple 64-bit operations, which allow these modern CPUs to perform several operations in parallel. (Skein-512 and Skein-1024 are better at this than Skein-256.) To support multicore architectures and grid computing, Skein provides an optional mode for tree hashing. The memory requirements for tree hashing grow linearly with the tree height. To avoid excluding low-end systems, the user can define a maximum tree height Y_m. For the same reason, we made sequential hashing the default and tree hashing optional.

Many simple rounds. We considered many complications to Threefish—additional MIX operations, a more complex key schedule, and so on—but in each case our analysis showed that additional simple rounds was the better alternative. For example, consider a more complicated MIX function. Going from three to five operations per MIX makes the algorithm more secure, but there's an additional 66% cost in clock cycles. We compared this change with increasing the number of three-operation MIX rounds by 66%, and our analysis showed that adding additional smaller rounds provided more security than making the MIX operations more complicated.

There are advantages to using many simple rounds. The resultant algorithm is easier to understand and analyze. Implementations can be chosen to be small and slow by iterating every round, large and fast by unrolling all rounds, or somewhere in between. Cryptographically, specific design complications may protect against a particular type of attack—differential [16], related-key [14, 51, 52], etc.—but adding more rounds has the advantage that it protects against almost all attacks and thus almost always adds security. (Slide attacks [17, 18] are the exception.) This general principle can be found again and again in block-cipher cryptanalysis: more rounds defeat attacks.

Maximum diffusion. Looking back on the general trend in cryptanalytic attacks over the past couple of decades, one aspect jumps out: they take advantage of insufficient diffusion. Differential attacks [16], linear attacks [71], and correlation attacks [25] are all based on the fact that the diffusion across the algorithm is uneven and incomplete. Similarly, the recent attacks against the MD and SHA family of hash functions have at their core methods of exploiting insufficient diffusion [15, 106, 56, 107, 108, 109, 56, 57, 58, 59, 103].

We designed Skein to maximize diffusion at every level, and have defined the number of rounds to be high enough to allow for many full diffusions. Each input bit position affects every output bit position in 10 rounds for Skein-512 (9 rounds for Skein-256 and 11 rounds for Skein-1024), so the algorithm is specified with 7–8 full diffusions. By comparison, AES-128 and Twofish have only 5 full diffusions.

Simple CPU operations. Modern CPUs are super-scalar and can execute multiple instructions in one clock cycle. To maximize this capability, an algorithm should only use simple operations such as addition, XOR, rotation by a constant, and so on. As an added benefit, these operations are also efficient on smaller CPUs.

Skein does not use complex CPU operations such as multiplication, rotation by a variable number of bits, or any of the multimedia extension instructions in various CPUs. These operations are often expensive to implement in hardware and on smaller CPUs that do not provide direct support for these operations. For example, the AES submissions Mars [21] and RC6 [95] used 32-bit multiplication, which is efficient on large CPUs but quite expensive in hardware and on small CPUs. We chose not to use the AES round function, which will be available as a hardware instruction on many high-end CPUs starting in 2009 [41], for the same reason (and because older CPUs would have to rely on table lookups—see below).

No table lookups. Modern CPUs have multi-level memory cache systems that help the processor run faster. Unfortunately, the current designs have a side-effect in that the memory access time that one processor thread experiences is dependent on the memory locations accessed by other threads, even if those other threads are in different processes. This provides a side channel [54]: one thread receives information about what another thread is doing. There are practical attacks where one thread can determine the cipher key used by another thread [91]. This is a potential problem for an encryption algorithm running in software on a modern operating system. For example, AES has been successfully cryptanalyzed using a side channel associated with its table lookups [12, 20].

Skein solves the problem by not using any table lookups at all.[4] Or more precisely, Skein has no table lookups whose address is not predictable in advance. A thread that uses a table of rotation constants does not leak anything other than the fact that Skein is running. And that fact is already known from the memory access pattern of the code itself.

Minimal loads and stores on reference platform. If an algorithm's internal state fits entirely within the CPU's registers, the CPU can run at full speed. If, on the other hand, the internal state exceeds the registers, any implementation has to perform loads and stores to move information between the registers and memory. Memory accesses are relatively slow, and don't add any cryptographic strength. Furthermore, in severe cases, they can provide a side channel to the attacker [61, 62, 54].

An x64 CPU has 15 available 64-bit registers. Threefish-256 and Threefish-512 fit comfortably within these registers. Threefish-1024 requires 16 registers, so its performance suffers slightly because it needs a few loads and stores every round.

Variable internal state. To be able to replace SHA-512, we needed a state size of at least 512 bits. On the other hand, some people hold that a hash function requires n-bit security against collision attacks, which requires an internal state size of $2n$.

There is a class of attacks that relies on internal collisions of the hash function (see Section 6.4). For an n-bit state, these start to be relevant when the attacker can perform $2^{n/2}$ operations. At worst, they limit the security level of the hash function to $n/2$ bits. For $n = 512$, a generic collision attack requires $2^{n/2} = 2^{256}$ time, which is safe enough for any foreseeable application.

Note that if the internal state size is n bits and the output size is n bits, we have the following undesirable property: A collision $H(X) = H(Y)$ between two messages $X \neq Y$ of the same length can be extended to a collision between longer messages $(X||Z) \neq (Y||Z)$ by appending the same string Z to both messages. This has been used in the past to turn random MD5 collisions into meaningful ones [46, 29, 76, 64, 37, 104]. In a more general context, Joux [44] used the same property to create huge multi-collisions very cheaply: a 2^k-collision just needs time $k \cdot 2^{n/2}$.

The main defense against that kind of attack is collision resistance—the adversary should be unable to find any collision at all. An output size of $n \geq 512$ ought to be beyond hope for the adversary. But it still would be desirable to provide a second line of defense. Even if one day finding collisions turns out to be somehow feasible—as for MD5—exploiting that weakness for creating either multi-collisions or meaningful collisions should remain infeasible. This requires us to increase the internal state size, which is the core idea for the failure friendly "wide-pipe" design [69]. Thus, if we want a 256-bit hash function to be failure friendly, we need 512 bits of internal state, and if we want a failure-friendly 512-bit hash function, we need 1024 bits of internal state.

[4] There are software techniques for doing table lookups with fixed memory access patterns, but these are so inefficient that they are very rarely used.

In general, we regard the internal state size as the main security parameter for a hash function. All versions of Skein support variable-sized outputs. We provide three different versions of Skein, supporting three different internal state sizes:

- Skein-256, the low-end version, which we consider more than adequately secure for typical applications, as one would expect from a well-designed plain 256-bit hash function.
- Skein-512, which we feel is sufficiently secure for essentially all applications. One can view Skein-512 as a wide-pipe 256-bit hash function, or as a plain 512-bit hash function.
- Skein-1024, for users who specifically require an exceptionally high level of security assurance.

We considered having a parameterized state size, but that creates considerable extra complication for very little gain. For the same reason, we dismissed designing a variant with more than 1024 bits of internal state.

Flexibility. Hash functions are used in a dizzying variety of applications: digital signatures, message authentication codes, key derivation, pseudo-random number generators, nonce generators, integrity checkers, cookie generation, and so on. We wanted our hash function to have the flexibility to be securely used in these widely diverse ways.

8.2 General Design Decisions

These are the basic decisions we made in the design of Skein.

Stream design vs. block design. Roughly speaking, a stream design has a continuous churning of the internal state and mixes in the message a little bit at a time, while a block design divides the message into larger blocks and thoroughly mixes each block into the internal state in turn.

The commonly used hash functions, like the MD [93, 94] and SHA [80, 81, 83] families, are all block designs. They have a block cipher at the core, and a mode of operation that turns the block cipher into a hash function. Some of the newer hash function designs, such as RadioGatún [13], are stream designs.

Block designs have the advantage of being easier to analyze than stream designs. Cryptanalysts can leverage the knowledge, tools, and techniques they have developed over the years for analyzing block ciphers. Analyzing stream constructions is harder. In the last decade, there have only been a few serious proposals for stream hash functions, and relatively little work has been done in analyzing them. Several of the basic tools of block cipher analysis do not apply to streaming modes. For example, block ciphers are almost always analyzed in a reduced-round versions, and it is far harder to design cryptanalytically useful reduced-strength versions of stream designs.

A stream-oriented hash function—such as one in the spirit of Helix [36] and Phelix [105]—could perhaps be faster than a conventional hash function based on an internal block cipher. But the additional speed—if any—might well be due to optimistic design decisions, lacking cryptanalytic experience for stream designs. Perhaps new attack techniques are just waiting for their discovery? For example, slide attacks are a well-understood tool for the cryptanalysis of block ciphers. But until very recently, slide attacks had not been considered for the analysis of hash functions. The authors of Grindahl [60], another recent stream-oriented hash function, were not aware of potential slide attacks. It turned out that Grindahl can be attacked that way [40].

Given the current state of cryptanalysis, we feel that a block-oriented design is more conservative and better suited for a new standard.

Tweakable block cipher. Although block design is better understood, a number of attacks against block-cipher-based hash functions directly attack the way that the hash functions process message blocks. While we shied away from a streaming design, we understand that “streamingness” is a desirable property. This led us towards using a tweakable block cipher. By directly constructing our underlying cipher so that each output block is different—that a message block yields a different result no matter where it is fed into the hash function—we produce “streamingness” while still using a block cipher. Our proofs of security are extensions of existing proofs about block design. Someone familiar with existing block ciphers can easily understand Skein as well as the security claims.

Padding vs. counter. Hash functions need to ensure that he message length is somehow encoded into the hash. Typically, this is done by appending the message length to the message [27]. Our design uses a block counter rather than padding. The counter provides the same security as the message length, but ensures that each message block is hashed in a unique way.

8.3 Threefish Design Decisions

This is the rationale behind the decisions we made in the design of Threefish.

SP network. Threefish uses an SP network [32] like AES [26, 82], rather than a Feistel network [32] like DES [79] or Twofish [100]. An SP network has the advantage that of providing more inherent parallelism, which modern CPUs can exploit with their superscalar architecture.

MIX function. Threefish’s MIX function is derived from Helix [36] and Phelix [105]. Initially, we had a more complex MIX function, with 2 adds, 2 XORs, and 4 rotations. The advantage of a more complex mixing function is that x86 CPUs, which have only 7 usable 32-bit registers, can load all of the function’s inputs into registers and execute the entire MIX function without loads or stores. However, our cryptographic analysis showed that more rounds of a simpler mixing function are more secure, for a given number of CPU clock cycles.

Another candidate design included a MIX function with 3 add/XOR operations and 2 rotations, but our performance measurements also showed that—contrary to what the chip’s documentation suggests—the current generation of Intel CPUs can only perform one rotate operation per clock cycle. This leads to a significant speed penalty on x64 CPUs, so we abandoned it, in keeping with the principle that more rounds more than made up for the simpler MIX function.

The current MIX function has 1 rotate and 2 add/XOR operations, which can be done in 1 clock cycle (amortized) on the current generation of Intel CPUs.

The basic non-linearity comes from the mixing of addition modulo 2^{64} and XOR. Add and XOR are very similar at low Hamming weights (or low Hamming weight differentials), but at average Hamming weights, they are very different. The good diffusion of our design ensures that low Hamming weight values or differentials quickly diffuse to average Hamming weights. With enough rounds, our MIX function provides excellent nonlinearity and diffusion.

Rotation constants. Our goal was to choose rotation constants that maximized diffusion across the entire cipher. A population of 1024 candidate sets of rotation constants evolved through multiple “generations,” as described below. In all cases, we rejected rotation constants with value 0, $+1$, and -1, since the add and XOR operations in the MIX function already provided diffusion to adjacent bits.

To fill the population initially, we selected candidate sets of rotation constants that maximized the

Hamming weight of a simplified version of Threefish. In this modified version, we replaced the addition and XOR operations in the Threefish MIX function with the logical OR operation. We then generated a random set of rotation constants and, using an all-zero plaintext, injected a single input bit difference at each input bit location. After R rounds, we measured the minimum Hamming weight of each of the N output words across all input difference locations. If the Hamming weight value was less than a threshold W, we rejected the rotation set and randomly chose another. If it was greater than or equal to W, we saved it for the next phase.

We selected values of R and W empirically based on the block size. The general idea was to choose values that were at the knee of the diffusion curve. In other words, if we chose R to be too small, all rotation sets looked alike. If we chose R to be too large, the minimum Hamming weight quickly reached 64 bits. Similarly, if we chose W to be too small, all rotation sets passed; and if we chose W to be too large, none passed. After some experimentation, we settled on the (R, W) sets of $(8, 50)$, $(8, 36)$, and $(9, 40)$ for Threefish-256, -512, and -1024, respectively.

This initialization mechanism is important because it is much faster than running the actual Threefish rounds, primarily because this metric is rotationally invariant. That is, we actually ran the diffusion test using only a single bit difference position per word, which sped up this phase by a factor of 64. We could also have used XOR instead of logical OR here, but the former would have included cancellations and hidden the true diffusion rate of a candidate set of rotation constants, so we felt that using OR was a better choice.

With the initial population of 1024 valid candidates, we employed an evolutionary algorithm, suggested by Guillaume Sevestre [102], to generate the final set of rotation constants. The evolution from generation to generation proceeded as follows. For each set in the population, we selected K random plaintexts and injected a one-bit difference in each possible input bit location, using the actual Threefish round function. We chose K to be 1024: small enough to run fairly quickly, but large enough to grade the rotation sets with reasonable probability. We generated a histogram for each output bit based on whether that bit changed after R rounds for each input bit difference, ignoring the key injection. For example, in Threefish-512, this meant the histogram had an array of 512x512 (256K) entries.

For a truly random function, the expected value for each histogram entry would be $K/2$ with a binomial distribution, with $p = 0.5$. Of course, with these small values of R, the function is not truly random, but the goal was to choose a reasonable metric with which to grade the sets of rotation constants. When averaged over the entire histogram, we expect the deviation to be very small, so we defined a metric that concentrates on the maximum deviation within a single bin, to smooth out the bit-to-bit variations as much as possible. In particular, for each set of rotation constants, we computed the maximum deviation (or "bias"), called B_{max}, from $K/2$ across all histogram entries, as the search metric. In addition, we computed the maximum bias using not only rounds $0..R-1$, but also rounds $4..R+3$, to maximize diffusion from any key injection point.

The 1024 candidates in the population were then sorted based on their maximum bias. The best 64 ones (i.e. with the lowest bias) formed the "keep set." The 640 worst ones were discarded and replaced by 10 copies of each of the candidates in the keep set. Then each candidate *not* in the keep set (i.e., 960 candidates, including the 640 replicated ones) were slightly modified, which involved changing a small number (1–8) of rotation values in each candidate set at random. The number of values to change, as well as what value to change them to, were all selected at random.

The algorithm was run for approximately two days for each block size on a 3.6 GHz CPU (5500 generations for 256 bits, 1600 generations for 512 bits, and 400 generations for 1024 bits), with the keep set slowly evolving toward smaller values of the bias. We limited the search time to get some actual rotation constants in order to proceed with analysis, since in each case the best bias values in the keep set converged to their final values (a local minimum) well before the run was complete. As a sanity check, we later ran a six-day search, and the improvements in the metric over the two-day results were inconsequential, as we expected.

At this point, the candidates in the final keep set were re-graded using larger values of K—4096, 8192, and 16,384—to minimize the expected statistical sampling error. For each bit position during the re-grading process, we also used an input difference pattern of up to three bits, with a nonzero difference in the first bit; i.e., the bit patterns 001, 011, 101, and 111. As we expected, the difference pattern 001 usually produced the worst case bias, but we nonetheless felt it was important to measure the maximum bias across multi-bit input differences.

Based on the relative rankings of the rotation constant sets in this re-grade step, we chose the winner for each block size. A copy of our search program, along with the resulting search log files, has been submitted to NIST and is also available on the Skein website, so anyone can duplicate and verify our work.

Word Permutation. Threefish's word permutations—one for each block size—were chosen to have the following properties:

- Each input word difference can affect all output words after only $\lceil \log_2(N) \rceil$ rounds, where $N = 4/8/16$.
- The period of the permutation must be a divisor of 8, so that the round function can be nicely looped after two key injections.

  In fact, all three word permutations have a period of 2 or 4. This means that after four iterations of $\pi()$, all the words are back where they started. Thus, software implementations that implement $\pi()$ by merely using different registers in each round can loop after four rounds without having to add the overhead of a word shuffle to the end of the loop.
- Even words are permuted with even words, and odd words with odd words. Due to the asymmetry between even and odd words after only one mixing step, this property was found to maximize diffusion. This means that there are $((N/2)!)^2$ possible permutations.

We believe that any permutation that satisfies these properties is suitable for Threefish. We performed an exhaustive search—up to renaming the words—and found two permutations each for $N = 4, 8, 16$. Table 3 list the ones we chose for Threefish, and Table 20 lists the other—not chosen—permutations.

		$i =$															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	4	2	1	0	3												
$N_w =$	8	0	5	2	7	6	3	4	1								
	16	2	1	4	9	6	15	0	3	10	13	12	11	14	7	8	5

Table 20: Alternate values for $\pi(i)$.

Rounds and cycles. Threefish has an unusual design face that, like Mars [21], does not inject the key every round. Key injections are on a separate schedule of a "cycle" of four rounds.

Like other features of Threefish, this comes from our core principle that adding rounds is usually the best way to strengthen a cipher. Hence, for Threefish-256 and -512, a variant with 60 rounds and a 2-round cycle would run approximately as fast as the 72 rounds and 4-round cycle we finally chose. Similarly, the 80 rounds and 4-round cycle for Threefish-1024 are approximately as fast as a variant with 66 rounds and a 2-round cycle would be. This is a trade-off that we needed to make: number of full diffusions versus number of key injections.

We examined cycle sizes of 4, 6, and 8 rounds (there were no suitable word permutations with period 5 or 10) and with a number of rounds from 64 to 96 total, always in some integral number of cycles. Our two best options were 72 rounds and 4-round cycles, and 80 rounds and 8-round cycles.

Various related-key attack methods typically get one or two cycles for free (with a zero differential) and attacks on 3–4 cycles are relatively easy to construct. Our own preliminary cryptanalysis in Section 9 will illustrate this. The 8-round variant has only 10 cycles; this leads to attacks on a significant fraction of the cipher. The advantage of 4-round cycles is that related-key attacks get fewer free rounds and are not nearly as successful.

We kept the rotation constants on their own cycle of 8 rounds because it comes at no performance cost, and iterative characteristics are harder to construct.

Number of rounds. The number of rounds represents a balance among several different considerations: the number of key injections, the number of full diffusions, and the ratio of input bits to output bits. That last consideration may need some explanation. Looking at Skein generally, the hash function uses Threefish as a compression function: plaintext bits plus key bits plus tweak bits compress into output bits. The number of input bits determines the attacker's degrees of freedom, and the attacker also gets to control the output bits (at least for pseudo-collision, pseudo-preimage, and pseudo-second-preimage attacks). A large ratio of input to output bits helps the attacker. Threefish-256 has a 2.5-to-1 ratio, Threefish-512 has a 2.25-to-1 ratio, and Threefish-1024 has a 2.125-to-1 ratio. This is why the number of rounds of Skein-256 isn't less than the number of rounds of Skein-512. Skein-1024 has more rounds because full diffusion is one round slower.

The current number of rounds is intentionally conservative. We will continue to evaluate Threefish and Skein, and may revise the number of rounds either upwards or downwards, depending on the results of our analysis.

Key schedule. Most key schedules are complicated, and require many clock cycles to set up. This doesn't matter when encrypting large blocks of text, but hurts performance considerably when encrypting small messages, or when changing key for every block, as UBI does. And, as always, a more complex key schedule means fewer rounds, from the security-per-clock-cycle principle.

The Threefish key schedule was inspired by Skipjack [88]. The Skipjack key schedule uses the bytes of a 10-byte key in order, cyclically. We found the simplicity very attractive. The Threefish key schedule is slightly more complicated than that, but it is still very simple compared to other block ciphers.

Our key schedule has the following properties:

- Given any subkey, it is possible to extract the full key for a known tweak and subkey number.
- Given any subkey, it is possible to extract the full tweak for a known key and subkey number.
- Given any two consecutive subkeys, it is possible to extract the full key, tweak, and subkey number.

- In a differential related-key attack, the distance between zero subkey differences is at least seven subkeys.
- The subkey values do not repeat with low period.
- The minimal repeat period for subkey differences is three.
- The key schedule can be implemented in a loop efficiently, without special branches or case statements based on the subkey number.

Recovering the key, tweak, and subkey number from two consecutive subkeys is somewhat complicated. Given the redundancy in the extended key and tweak, it is possible to recover the least significant bit of each key word, tweak word, and the subkey number. This knowledge provides all the carries going into the next bit position, which allows the recovery of the next bit of each value.

Subkey counter. The subkey counter prevents slide attacks [17, 18] and any other attacks based on identical subkeys. It also destroys any word-rotational symmetry in the cipher.

Back doors. Threefish and Skein have no back doors. We understand that the super-paranoid might wonder if the rotation constants were selected so as to create a cipher with a back door in it. To assuage those fears, we have made public the program that generated the rotation constants for anyone to run and verify.

Ultimately, although there is no way to dispel this paranoia, we can offer the following comment. One of the things we know mathematically is that a block cipher with an invisible back door is equivalent to a public-key algorithm. If we had created a public-key encryption algorithm that had 512 bits of security and ran twice as fast as AES, we wouldn't be secretly using it as a block cipher. Instead, we'd be revolutionizing public-key cryptography.

8.4 UBI Design

UBI is a variation of the cascade construction [7] built upon a compression function constructed out of a tweakable block cipher.

Matyas-Meyer-Oseas. We chose Matyas-Meyer-Oseas [70] over Davies-Meyer [77, 92] to simplify the mathematical security arguments. We can prove that the compression function is a PRF, assuming that the block cipher is a PRP—a standard assumption on a block cipher.

Less formally, Matyas-Meyer-Oseas is desirable because the primary attack model for a hash function allows the attacker to choose the data input. In Davies-Meyer mode, this corresponds to a chosen-key attack on the block cipher. In UBI, this corresponds to a chosen-plaintext attack. The cryptographic community has a great deal of experience protecting block ciphers against chosen-plaintext attacks, but less experience in the area of chosen-key attacks. For a new standard, it is always preferable to stay with what you know.

This is even clearer when we look at attacks on the underlying block cipher. Differential attacks are probably the most important class of attacks to consider. In UBI, a difference in a data block leads to a difference in each round. With Davies-Meyer, a difference can be canceled out at one subkey and reintroduced at a subsequent one; it could even happen repeatedly in one block. This gives the differential a free pass through some of the rounds, which is highly undesirable. It also makes it much harder to provide a useful estimate for the upper bound of a differential characteristic.

Tweak. The purpose of the tweak is to make each block operation in Skein unique. Different Skein input fields use different field types in the modifier, and different blocks within one field use a different position value.

First and final flags. These flags exist primarily to support our proofs of security and to simplify the security properties of UBI. As defined, Skein would be secure without these flags.

It is possible, however, to create a collision in UBI without the First flag: the hash of a two-block message, M_1, M_2, collides with a hash of M_2 and an appropriate tweak value. This collision could not occur in Skein, as the tweak value is defined in such a way as to not permit it. But UBI has potential applications outside of Skein, and we consider it safer to define it for more general security.

Maximum message length. Skein is defined for messages up to $2^{96} - 1$ bytes, or 64 kilo-tera-terabytes, long. We consider this length to be long enough for the foreseeable future, and have reserved 16 bits of the tweak for future use, instead of increasing the maximum message length to $2^{112} - 1$ bytes.

8.5 Optional Argument System

Configuration block. The best way to think of the configuration block is as a method of computing the starting value for the chaining state. Other hash function families do the same thing; for example, SHA-384 is identical to SHA-512, except that the starting value is different and the output is truncated. Rather than define a large number of random-looking starting values, we compute them using the configuration block.

Output transformation. Originally we applied the output transformation only if the output size was larger than the state size. Unfortunately, without the output transform, you can construct two messages M and M' such that $H(M) \oplus H(M')$ is the same as the XOR of the last blocks of M and M'. (A similar property has recently been described for SHA-1 [98].) This violates the requirement that the hash function behave like a random mapping.

We chose the simplest solution to this problem: always apply the output transformation. This both increases robustness and makes our security proofs easier, but it halves Skein's speed for hashing small messages. We looked at many other solutions, such as applying a half-block fixed padding to the message. This solution made the obvious construction for the XOR-property not work, but it felt like a hack and we were not convinced that it addressed any still-undiscovered variations of that attack. We decided to accept the performance penalty and chose a solution that addressed all our concerns.

In most real-world applications, the application's own per-message overhead is already significant, and often larger than the cost of hashing a short message. Thus, the overhead of the output transformation does not decrease the practical throughput as much as one would think. The exceptions are applications like IPsec hardware, where short-message performance is very important.

The output transformation is a one-way function, which isolates the output from the last point a user-chosen value affects the computation: the feed-forward of the last message block.

Multiple optional arguments. Cryptosystems use hash functions for a plethora of purposes. This agility requirement creates an added challenge for hash function design. Developers will use the same hash function for radically different purposes, and—as time goes on—they will invent new ways to use that same hash function. As cryptographers, we can caution developers to only use a hash function in certain specific ways, or not to use it for multiple purposes, but our experience shows that it doesn't work in practice. A better alternative is to design a hash function assuming that it will be used and abused.

Skein's system of optional arguments addresses this by letting the user specify the purpose of the hash function, and encoding that specification into the hash function itself, to make it unique for that purpose. Thus, Skein-for-signatures is a slightly different hash function than Skein-for-key-derivation or Skein-for-MACs. The nonce argument also allows for building randomized hashing into the core of the hash function, which will be a boon for anyone using Skein for Tripwire-like data integrity systems [55]. A given host that computes file hashes can make those hashes unique for that host, something that makes the attacker's job that much harder. (Of course, the application could also use the MAC mode and use MACs rather than hashes to check the integrity of the data.) We also allow for these optional arguments to be combined. A cryptosystem can directly use the nonce along with public-key specialization.

We believe that this is an important innovation in Skein's design. We turn a source of unease about the way cryptographic engineers use hash functions into a strength. Every purpose served by the hash function creates a unique hash function. Additionally, engineers can trivially create their own personalized hash functions, and be assured of its cryptographic integrity.

Skein can be generalized to allow the arguments in any order, or allow the same argument type to be used multiple times. Although interesting from a theoretical point of view, such flexibility is likely to lead to confusion and lack of interoperability between different implementations and applications of Skein. Furthermore, such generalizations would affect the security proofs, and require careful analysis.

Key input. The most logical place for processing the key input would be somewhere after the configuration block. However, we chose to always process the key first to make our security proofs simpler.

The security analysis is in two parts. The first UBI call maps the key into a chaining state. Assuming that UBI behaves like a random mapping (which we already require), this maps the key into a secret chaining state. From that point on, the chaining state is a key, and always goes into the key input of the Threefish block cipher. This uses the block cipher exactly as a normal block cipher is used: with a secret key and public plaintext. This simplifies the security proofs and allows them to use standard block cipher security assumptions.

9 Preliminary Cryptanalysis of Threefish and Skein

Our Skein analysis concentrates on the security of the compression function—primarily, security against pseudo-collisions and pseudo-second-preimages—and on the security of the Threefish block cipher. If it isn't possible to find a pseudo-collision for the compression function, it's likewise not possible to find a collision for the hash function. Similarly, it's not possible to find preimages, second preimages, and near misses.

Furthermore, our security analysis focuses on XOR-differential characteristics. Other algorithms that make use of Threefish's basic operations—for example, Helix [36] and Phelix [105]—have proved vulnerable to differential cryptanalysis based on XOR differences [78, 89, 90, 110].

We stress that the designers of a cryptosystem are not the best ones qualified to analyze their own cryptosystem for potential weaknesses. Furthermore, our own analysis has been guided by the need to decide on possible modifications of Threefish and Skein, including the number of rounds. As long as we where confident that our attacks would not extend to anything near the specified number of rounds, we did not try to push our attacks through another two or three rounds – we rather leave this to third-party cryptanalysis. In fact, by documenting our effort in analyzing Skein and Threefish[5], we hope to inspire more third-party cryptanalysis.

9.1 Pseudo-Near-Collisions for the Skein-256 Compression Function Reduced to Eight Rounds

Consider eight rounds (two cycles) of the Threefish-256 block cipher. Before the first round, after round 4, and after round 8, a subkey is added. Table 21 gives an overview of these three subkeys. The values K_i are the key words, and T_i the tweak words. $K_\oplus$ is the XOR of all the key words and similarly, $T_\oplus$ is the XOR of both tweak words. C_5 is a fixed constant, and $\langle i \rangle$ is the current round constant.

subkey	injected	word 0	word 1	word 2	word 3
first	before round 1	K_0	$K_1 + T_0$	$K_2 + T_1$	$K_3 + \langle 0 \rangle$
second	after round 4	K_1	$K_2 + T_1$	$K_3 + T_\oplus$	$(K_\oplus \oplus C_5) + \langle 1 \rangle$
third	after round 8	K_2	$K_3 + T_\oplus$	$(K_\oplus \oplus C_5) + T_0$	$K_0 + \langle 2 \rangle$

Table 21: The first three subkeys of the Threefish-256 key schedule.

Assume we chose two key/tweak pairs:

$$((K_0, K_1, K_2, K_3), (T_0, T_1)) \neq ((K'_0, K'_1, K'_2, K'_3), (T'_0, T'_1))$$

such that there is no difference in the second subkey—the one added after round 4. This implies

$$K_1 = K'_1, \quad K_2 + T_1 = K'_2 + T'_1, \quad K_3 + (T_0 \oplus T_1) = K'_3 + (T'_0 \oplus T'_1),$$

and

$$(K_0 \oplus K_1 \oplus K_2 \oplus K_3 \oplus C_5) + 0\ldots0001 = (K'_0 \oplus K'_1 \oplus K'_2 \oplus K'_3 \oplus C_5) + 0\ldots0001.$$

Now define $\delta = 1000\ldots0$, i.e., the difference is isolated in the most significant bit. In this case, differences propagate under addition exactly as under XOR, i.e., in the context of a differential attack, the distinction between "+" and "$\oplus$" disappears. Set

$$K_0 \oplus K'_0 = \delta, \quad K_2 \oplus K'_2 = \delta, \quad T_1 \oplus T'_1 = \delta, \quad T_0 \oplus T'_0 = \delta,$$

and

$$K_1 = K'_1, \quad K_3 = K'_3.$$

In this case, the difference in the first subkey is $(\delta, \delta, 0, 0)$, and the difference in the third subkey is $(\delta, 0, \delta, \delta)$.

Choose a pair of messages with the same difference as in the first subkey; i.e., $(\delta, \delta, 0, 0)$. All the differences in message and subkey cancel out, so we have some kind of a *local collision*, which

[5] Of course, there was much more internal cryptanalysis on preliminary and alternate versions of Threefish, UBI, and Skein. While it was useful to guide our design decisions, most of it is irrelevant for the current version.

propagates through rounds 1 to 4. After round 4, the second subkey is injected, with a zero difference of its own. Thus, the *local collision* propagates further to round 8. Then, finally, a subkey with a nonzero difference is injected, and the local collision breaks apart, leaving a difference $(\delta, 0, \delta, \delta)$ in the state. This is the output of our block cipher (namely, Threefish-256, reduced to eight rounds). The chaining mode of Skein requires us to XOR the message to the final block cipher output $H_i := C(H_{i-1}, T_i, M_i) := \text{block\_cipher}_{H_{i-1},T_i}(M_i) \oplus M_i$. So the output difference of the compression function (using eight rounds of Threefish-256 as the underlying block cipher) is $(0, \delta, \delta, \delta)$. As $\delta = 1000\ldots0$, all these differences appear with probability one. This gives the attacker a near-collision with Hamming difference three: all the output bits of our reduced-round compression function are the same, except for exactly three bits, which remain differently.

One can generalize this attack probabilistically, for some $\delta \neq 1000\ldots0$, as long as the Hamming weight of the 63 least significant bits of δ remains low.

Additionally, we get another near pseudo-collision—actually an even better one with Hamming difference 2—by setting

$$K_2 \oplus K'_2 = \delta, \quad T_1 \oplus T'_1 = \delta, \quad K_3 \oplus K'_3 = \delta,$$

and

$$K_1 = K'_1, \quad T_0 = T'_0, \quad K_0 = K'_0.$$

In this case, the difference in the first subkey is $(0, 0, 0, \delta)$, and the difference in the third subkey is $(\delta, 0, 0, 0)$. This is the output difference after eight rounds of Threefish-256. Note that the Hamming weight of the difference is one, for $\delta = 1000\ldots0$. Applying the chaining mode then doubles the Hamming weight; the difference is now $(\delta, 0, 0, \delta)$.

Note that the above pseudo-near-collision attack did actually allow the adversary to arbitrarily choose two different triples (Tweak, Chaining-Value, Message) and (Tweak′, Chaining-Value′, Message′) with a certain difference. The attack even works if one triple (Tweak, Chaining-Value, Message) has been fixed in advance. So this isn't just a near pseudo-collision, it even is a near pseudo-second-preimage.

9.2 Pseudo-Near-Collisions for Eight Rounds of the Skein-512 and -1024 Compression Functions

It is straightforward to apply the same attack principles to the Skein-512 and Skein-1024 compression functions:

- Choose key and tweak differences such that there is a zero difference in the second subkey.
- Choose the difference of the first subkey as the message difference, to get a local collision for the first eight rounds, excluding the key addition. If our differences are in the most significant bit only, the local collision occurs with probability one.

After the key addition and the message addition, we get some near-collision, exactly as for Skein-256.

Set $N = 4$ for Skein-256, $N = 8$ for Skein-512, and $N = 16$ for Skein-1024. Set $\delta = 1000\ldots0$. We can choose

$$K_{N-1} + K'_{N-1} = \delta, \quad K_{N-2} + K'_{N-2} = \delta, \quad \text{and} \quad T_1 + T'_1 = \delta,$$

and

$$K_i = K'i \ \ \text{for} i \in \{0, \ldots, N-3\}, \quad \text{and} \quad T_0 = T_0'.$$

This gives the subkeys added before the first round the differences

$$(0,0,0,\delta), \quad (0,0,0,0,0,0,0,\delta), \quad (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\delta),$$

for Skein-256, -512, and -1024, respectively. Similarly, the subkey differences after round eight are

$$(\delta,0,0,0), \quad (0,0,0,0,\delta,0,0,0), \quad (0,0,0,0,0,0,0,0,0,0,0,0,\delta,0,0,0).$$

With message differences

$$(0,0,0,\delta), \quad (0,0,0,0,0,0,0,\delta), \quad (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\delta),$$

we get an 8-round local collision, with probability one. This local collision is finally destroyed by the key addition after round 8. The output-difference after eight rounds of either Threefish-256, Threefish-512, or Threefish-1024 has Hamming weight 1, and after applying the chaining mode, the corresponding near-pseudo-collision for the compression function of Skein-256, -512, or -1024 has Hamming weight 2.

In any case, this attack on the compression functions of Skein-512 and Skein-1024 is more than just a near-pseudo-collision attack: One can fix one triple (Tweak, Chaining-Value, Message) in advance, thus implying a near-pseudo-second-preimage attack.

9.3 Related-Key Attacks for the Threefish Block Cipher

Now we consider the Threefish block cipher on its own, disregarding the chaining mode. Recall that by choosing appropriate differences in tweak, cipher key, and message, we were able to get an output difference with Hamming weight 1, after eight rounds of any variant of the Threefish block cipher, including the key addition. In a related-key attack on tweakable block ciphers, the key is secret, but the adversary can choose tweak and message at will. In our case, by making just two related-key queries with the appropriate differences in tweak and message, the adversary can predict the some ciphertext difference with high probability—even with probability one.

In contrast to attacking the compression function, attacking the block cipher itself extends nicely to a couple of additional rounds. Since we are able to predict a low-Hamming-weight difference after round eight of Threefish, we can probabilistically predict the differential behavior for a few more additional rounds, even under an unknown key. We will first consider distinguishing attacks, and then deal with key recovery.

In the context of this preliminary cryptanalysis, we focus on the bias of isolated ciphertext bits after executing a reduced-round Threefish block cipher. Of course, more advanced distinguishers are possible, but based on our simple distinguishers, we are very confident that no such attack could come close to penetrating the full number of rounds of either Threefish-256, Threefish-512, or Threefish-1024. This is confirmed by the third-party cryptanalysis of Threefish/Skein we have seen so far, see Section 9.5.

As we will describe below, we determined the bias of individual bits by generating a lot of random plaintext pairs with a fixed difference, and then empirically computing the probability of this bit to be flipped. Ideally, this probability should be 0.5 (bias 0.0). By the sample sizes we can afford (20 to 50 million pairs), a bias of more than 0.001 is significant. I.e., below we claim a distinguisher if there is at least one bit with a bias exceeding 0.001. Pushing this threshold to the next order of magnitude (bias 0.0001) would approximately require to extend the sample size by two orders of magnitude.

9.3.1 Empirical Observations for Threefish-256

If we have a Hamming-weight-one difference after round eight, including the key addition, what will be the differences in the next few rounds? Consider the following experiment: Generate a pair of triples (Key, Tweak, Message), each pair consisting of

- a random key K, a random tweak T, and
- a random message M

and

- a key K' and a tweak T' such that the difference to (K, T) in the first subkey is $(0, 0, 0, \delta)$, the difference in the second subkey (added after round four) is $(0, 0, 0, 0)$, and the difference in the third subkey is $(\delta, 0, 0, 0)$, and
- a message M' with the difference $(0, 0, 0, \delta)$ to M.

This is precisely the setting for the eight-round local collision and the difference $(\delta, 0, 0, 0)$ afterwards. We assume $\delta = 1000 \dots 0$.

Write $W^r_{w,b} \in \{0, 1\}$ for b-th bit in word W^r_w, where $(W^r_0, \dots, W^r_3)$ is the output after r rounds of encrypting M under the key K and the tweak T. Similarly, for the r-round encryption of M' under K' and T', write $(W')^r_{w,b} \in \{0, 1\}$. In any case, $b \in \{0, \dots, 63\}$ and, for Threefish-256, $w \in \{0, 1, 2, 3\}$.

For Threefish-256, we repeated the experiment fifty million times ($50{,}000{,}000 \approx 2^{25.6}$), thus generating fifty million random pairs

$$\big((W^r_0, W^r_1, W^r_2, W^r_3), ((W')^r_0, (W')^r_1, (W')^r_2, (W')^r_3) \big)$$

with the specified difference for each round. We then counted how often the individual bits in $W^r_{w,b}$ and $(W')^r_{w,b}$ were the same, thus estimating the probabilities

$$p^r_{w,b} \quad = \quad \text{Prob}[W^r_{w,b} = (W')^r_{w,b}]$$

for each word $w \in \{0, 1, 2, 3\}$ and each bit $b \in \{0, \dots, 63\}$. Note that if r rounds of Threefish did behave like an ideal cipher (aka “Shannon cipher”), all these probabilities would be 0.5.

We define the “bias” by

$$|p^r_{b,w} - 0.5|.$$

If $p^r_{b,w} \in \{0, 1\}$, then the bias $|p^r_{b,w} - 0.5|$ is exactly 0.5, or “full.” Table 22 summarizes our results.

The first, leftmost column is the number of the round, after which we computed the bias (“after round 0” means before the first round). For each round r (up to a certain upper bound, when nothing “interesting” can be seen any more), the table gives the number of bits with “large” bias for each round; i.e., the number of bits with full bias, and with a bias exceeding 10%, 1%, and

round	# bits with bias				average
r	full	> 0.1	> 0.01	> 0.001	bias
0:	256	256	256	256	0.50000
1–9:	256	256	256	256	0.50000
10:	256	256	256	256	0.50000
11:	242	254	254	254	0.49225
12:	120	242	242	242	0.43372
13:	41	222	223	223	0.34853
14:	9	168	189	189	0.19401
15:	0	63	130	152	0.04981
16:	0	3	40	61	0.00349
17:	0	0	0	7	0.00010
18:	0	0	0	0	0.00006
19:	0	0	0	0	0.00006
20:	0	0	0	0	0.00006

Table 22: Empirical results for Threefish-256, sample size 50,000,000 pairs.

0.1%, respectively. The table also gives the average bias over all the 256 bits considered. By “full bias,” we actually mean $p^r_{w,b} =\in \{0, 1\}$; i.e., the number of bits which behave linearly.

At the beginning, everything is deterministic—all the bits have bias 0.5; i.e., either $p^r_{w,b} = 1.0$ or $p^r_{w,b} = 0.0$. This continues throughl the end of round 10. From round 11 on, the number of highly biased bits quickly declines. After round 18, the statistical noise dominates the bias observed.

Thus, there is a very simple distinguisher for 17 rounds of Threefish-256, in the context of a related-key chosen-tweak chosen-plaintext attack: Determine the bit $W^{17}_{w,b}$ with the largest bias. Choose a few thousand input pairs with the appropriate differences. For each such pair, count how often bit b in word w of the two outputs is the same. If the result is close to 50% of all pairs, we have a random permutation. If it is significantly divergent from 50%, we have Threefish-256.

Instead of counting $W^r_{w,b} \oplus (W')^r_{w,b}$, for $r = 17$ and some “good” w, b, we could search for correlations between $W^r_{w,b} \oplus (W')^r_{w,b}$ and $W^r_{w,b'} \oplus (W')^r_{w,b'}$ for some “good” w, b, b'. We did not study that approach in detail, but we would expect to get a distinguisher for 18 rounds of Threefish-256 that way.

9.3.2 Empirical Observations for Threefish-512 and Threefish-1024

For Threefish-512 and Threefish-1024, we can perform essentially the same experiment we did for Threefish-256. That is, we choose tweak, key, and message such that we get a local collision in the first eight rounds, *ex*cluding the key addition. The key addition injects

$$\text{difference} \quad (0, 0, 0, 0, \delta, 0, 0, 0) \quad \text{(for Threefish-512)}$$

and

$$\text{difference} \quad (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \delta, 0, 0, 0) \quad \text{(for Threefish-1024)},$$

which then becomes the difference before round nine. As above, $\delta = 1000\ldots0$, and everything in the first few rounds happens with probability one. As for Threefish-256, we repeated these experiments 50 million times each for Threefish-512 and Threefish-1024, and computed $p^r_{w,b}$ for rounds $r \in \{9, \ldots, 24\}$, $b \in \{0, \ldots, 63\}$ and $w \in \{0, \ldots N-1\}$, with $N = 8$ for Threefish-512 and $N = 16$ for Threefish-1024. Tables 23 and 24 summarize our results.

round	# bits with bias				average
r	full	> 0.1	> 0.01	> 0.001	bias
0:	512	512	512	512	0.50000
1–9:	512	512	512	512	0.50000
10:	512	512	512	512	0.50000
11:	458	510	510	510	0.49609
12:	270	494	494	494	0.45799
13:	69	463	463	463	0.39152
14:	29	385	400	403	0.24939
15:	0	190	267	277	0.07153
16:	0	14	64	99	0.00439
17:	0	0	1	4	0.00008
18:	0	0	0	0	0.00005
19:	0	0	0	0	0.00006
20:	0	0	0	0	0.00005

Table 23: Empirical results for Threefish-512, sample size 50,000,000 pairs.

round	# bits with bias				average
r	full	> 0.1	> 0.01	> 0.001	bias
0:	1024	1024	1024	1024	0.50000
1–9:	1024	1024	1024	1024	0.50000
10:	1024	1024	1024	1024	0.50000
11:	972	1022	1022	1022	0.49805
12:	743	1006	1006	1006	0.47936
13:	405	975	975	975	0.44662
14:	140	882	894	894	0.35347
15:	16	650	723	728	0.19930
16:	0	197	365	420	0.03414
17:	0	7	44	93	0.00135
18:	0	0	0	0	0.00006
19:	0	0	0	0	0.00005
20:	0	0	0	0	0.00006

Table 24: Empirical results for Threefish-1024, sample size 50,000,000 pairs.

These tables confirm that Threefish-512 diffuses slightly slower than Threefish-256, and Threefish-1024 diffuses slightly slower than Threefish-512. Nevertheless, for each of Threefish-512 and Threefish-1024, the statistical noise dominates the bias after round 18, as was the case for Threefish-256. We believe it is possible to penetrate one additional round by considering correlations between output bits instead of isolated bias of the bits. This would imply the same kind of plausible 18-round distinguishers for Threefish-512 and Threefish-1024 that we expect for Threefish-256.

One could try to counter the noise by greatly increasing the sample size, theoretically to almost 2^{X-1} pairs for Threefish-X. This could, perhaps, push the distinguisher a little further. We did not consider such huge sample sizes, however. First, this approach doesn't seem to scale well with our approach to key recovery attacks, which we will describe in Section 9.3.3. Second, once the number of bits with a large bias is small, it doesn't take many further rounds for the bias to disappear in the statistical noise. For example, if there are less than 20 bits with a bias of more than 0.1, then at most 2 rounds later there isn't a single bit left with a bias > 0.001, cf. Tables 22–24. Thus, a huge increase of the sample size to handle somewhat smaller thresholds for the bias would provide a very limited gain.

9.3.3 Key Recovery Attacks

The core idea for our key recovery attacks is as follows:

1. Assume a simple distinguisher for r rounds of the block cipher. Here, "simple" means that a certain property that allows us to distinguish r rounds of the cipher from random, only depends on one or two bits of a single word W_2^r of the output after round r. Using that property, we can make t related-key chosen-tweak chosen-plaintext queries to distinguish r rounds of our cipher from a random permutation.
2. Partial decryption: Attack $r + s$ rounds of the cipher, for s as large as possible. Assume a key addition after $r + s$ rounds. For the attack, we guess k bits of the final round key, and partially decrypt all the $2t$ ciphertexts, such that we get all those bits of word W^r, which are needed to apply the simple distinguisher.
3. Apply the simple distinguisher. Sort out most of the false key guesses.
4. Exhaustively search the remaining key space.

Note that t is the number of ciphertext pairs and k is the number of round key bits to be guessed. Thus, the number of partial decryptions is $2t * 2^k$. In our current context, k will be close to the full key size, which implies that t cannot be overly large.

We start with 20 rounds of Threefish-256, assuming the simple distinguisher for 18 rounds, which we "expected" " in Section 9.3.1. For concreteness, assume our simple distinguisher after round 18 deals with, say, word W_1^{18}.[6] See Figure 11.

The bth bit $W_{1,0}^{18}$ of W_1 only depends on the key words K_0, K_2, K_3 and on the least significant b bits of the intermediate variable $\mathbf{X}$. For $i \in \{0, \ldots, b\}$, changing X_{b-i} changes $W_{1,b}^{18}$ with probability 2^{-b}. Similarly, changing $K_{1,(b-c-i-j) \bmod 64}$ only affects the bit X_{b-i} with at most the probability 2^{-j}. Thus, given b and c, it is easy to decide which bits of K_1 are statistically relevant and must be guessed, and which bits of K_1 can safely be neglected. Hence, we can employ our simple distinguisher to sort out most of the false guesses. This provides a key recovery attack for 20 rounds of Threefish-256.

For related reasons, we also don't need to guess all the bits of K_0, K_2 or K_3. We anticipate that it suffices to guess between 50% and 75% of all the 256 round key bits.

Threefish-512 and Threefish-1024 use longer keys, thus allowing an attack to spend more time without being slower than an exhaustive key search. We can exploit that to go beyond 20 rounds.

To analyze attacking any variant of Threefish with r rounds, where $r \bmod 4 \neq 0$, we require an additional key addition after the final round. Otherwise, the final $r \bmod 4$ rounds could be

[6] Observe that the word W_1^{18} depends on the words W_w^{20} and K_w for $w \in \{0, 2, 3\}$ and on $\mathbf{X}$. Hence, when given the intermediate value $\mathbf{X}$, neither W_1^{20} nor K_1 is needed to determine W_1^{18}.

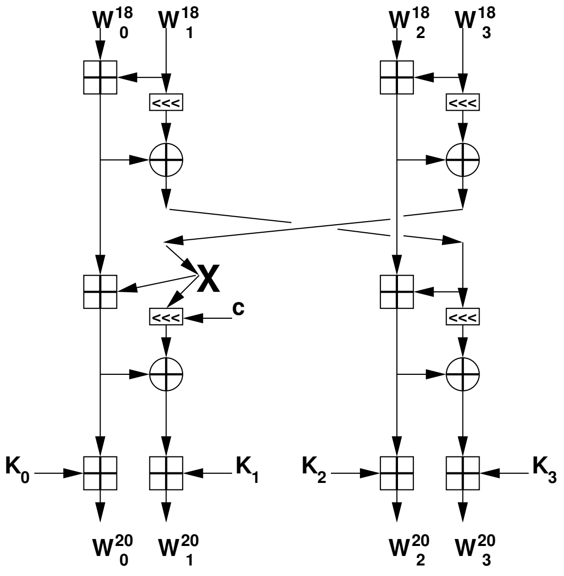

Figure 11: Simplified representation of rounds 19 and 20 of Threefish-256, including the key addition after round 20.

trivially inverted, without knowing the key, and we could effectively attack $r - (r \bmod 4)$ rounds. In the remainder of Section 9.3.3, we consider attacks on 21 rounds of Threefish-512 and 22 rounds of Threefish-1024, with a key addition after the final round. We assume that we can undo or neglect the "regular" key addition after round 20. Without this assumption, analyzing the partial decryption step becomes tricky.

- Threefish-512: Assume the same kind of distinguisher as above, for 18 rounds. Guess most of the final round key, which is added after round 21. Partially decrypt rounds 21 to 19, and apply the distinguishing property to sort out false key guesses.
- Threefish-1024: Assume a distinguisher for 18 rounds of Threefish-1024, partially decrypt rounds 22 to 20, and apply that distinguisher.

9.3.4 Pushing the Attack Further: Prepending Four Additional Rounds

To push the attack any further, we will look at the first few rounds of Threefish. In other words, we do the following:

- Apply the above attack (on 20 rounds of Threefish-256, 21 rounds of Threefish-512, and 22 rounds of Threefish-1024). But instead of starting with the first round, i.e., with round 0, we start with round 4 now.
- To bridge the first four rounds, try an appropriate message difference as the input for round 0, which will get the input difference D_5 for round 5 that we need. (In our case, for any of the three Variants, this is $D_4 = (0, \ldots, 0, \delta)$ with $\delta = 1000 \ldots 0 \in \{0,1\}^{64}$).
- We cannot expect a probability-one approach here—even the best plaintext difference D_0 would only turn into the required difference D_4 with some probability $p_{0,\ldots,3}$. Thus, the values our distinguisher sees will be much more noisy. To compensate for the additional

noise, we will have to increase the sample size by approximately a factor of $1/p^2_{0,\ldots,3}$. That is, if we needed σ samples before, we now need $\sigma/p^2_{0,\ldots,3}$.

In other words, we need a good four-round differential characteristic with the output difference $D_4 = (0, \ldots, 0, \delta)$, which is then turned into an eight-round local collision from round 4 to round 11.

Consider a single round of Threefish. If we want a specific difference D_i after round i, we can run round i backwards; in other words, in decryption direction, to compute some difference D_{i-1} before round i. To analyze this attack, we need to estimate

- the probability $p(D_{i-1} \rightarrow D_i)$ that two random inputs to round i with difference D_{i-1} produce any two outputs with difference D_i, and
- the difference D_{i-1} to maximize $p(D_{i-1} \rightarrow D_i)$.

We are only interested in a crude estimate of that probability. We will use the local Hamming weights to derive that estimate. Recall our MIX operation:

$$\text{Mix}_c(A, B) = (A + B, (B <\!\!<\!\!< c) \oplus (A + B).$$

If the Hamming weight is low, a good heuristic is to assume that the addition behaves exactly like the XOR operation. Assume $\text{Mix}_c(A, B) = (X, Y)$, and write a, b, x, and y for the Hamming weights associated to A, B, X, and Y, respectively. For our crude estimate, we will apply the following three rules:

1. $a = y + 2x$.
2. $b = x + y$.
3. The differential probability is $\approx 2^{-x-y}$.

Below, we will focus on Threefish-256, but we believe that this approach gives the adversary an additional four rounds for any of the three variants of Threefish.

Our target output difference is of the form $(0, 0, 0, \delta)$, with Hamming weight 1. The target output for the MIX operations in the final round are $(0, 1)$ and $(0, 0)$ (due to the permutation). Applying the above three rules provides an input difference with Hamming weights $(12, 7, 9, 6)$, as depicted in figure 12. Applying the third of our three rules, to estimate the probabilities of this differential behavior in every round, gives a probability of 2^{-21}.

This allows an attacker to push distinguishing attacks and key recovery attacks four rounds further, at the cost of increasing the sample size by a factor of more than 2^{40}. Our attacks apply for 24 rounds of Threefish-256, 25 rounds of Threefish-512, and 26 rounds of Threefish-1024.

Our probability estimate may be a bit too pessimistic, from the adversary's point of view. But the next logical step, namely, pushing the attack through another four rounds (one additional cycle), seems to require too large a sample size to be of any use for our key recovery attacks.

9.4 An Attack on the Threefish Block Cipher that Doesn't Quite Work

Our key schedule has been chosen with great care, such that the adversary cannot choose two different (tweak, cipher key) pairs with a zero difference in round i and round $i + 1$, or with a zero

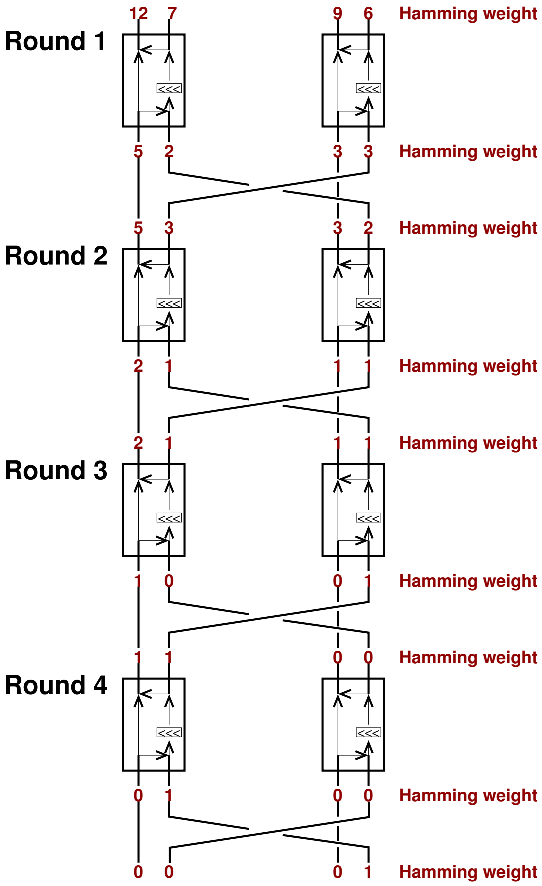

Figure 12: A differential characteristic for the first four rounds of Threefish-256.

difference in round i and $i+2$. In the first case, our local collision wouldn't break apart after eight rounds, but carry on for twelve rounds.

The case of a zero difference in round i and $i+2$, with a nonzero difference in round $i+1$, is a bit more complicated, but instructive. We can mount a boomerang attack.

Assume two key/tweak pairs $(K,T) \neq (K',T')$ with a zero difference in the second and the fourth subkeys. Choose related keys/tweaks and messages M and M', such that local collision in the first eight rounds occurs. That is, after eight rounds, before the key injection, we have the same intermediate value I, both when computing the encryption $E_{K,T}(M)$ and when computing the encryption $E_{K',T'}(M')$. After the key addition, we have a certain difference, and the next eight rounds can be expected to destroy any predictable difference. So we get two ciphertexts $C = E_{K,T}(M)$ and $C' = E_{K',T'}(M')$.

Now we choose new ciphertexts C'' and C''' with the appropriate differences, and decrypt C'' under (K',T') and C''' under (K,T). In rounds 9 to 16 (or rather, in rounds 16 to 9, when decrypting chosen ciphertext queries), we get another local collision between C'' and C, and also a local collision between C''' and C'. Decrypting further, we get two messages $M'' = D_{K',T'}(C'')$ and $M''' = D_{K,T}(C''')$ with

$$M'' \oplus M''' = M \oplus M'.$$

For an appropriately chosen difference, this holds with probability one.

Hence, with just two chosen-plaintext queries and another two chosen-ciphertext queries, we could easily distinguish 16 rounds of Threefish from a random permutation, using a boomerang property with probability one.

But recall that this attack requires a property that is not provided by our key schedule; namely, different key/tweak combinations with a zero difference in some subkey i and a zero difference in subkey $i+2$.

Could it help the attacker if we got some key/tweak combinations in some subkey i and $i+3$ or $i+4$? The boomerang property with probability one breaks apart, but a lower probability might still do. For example, assume a zero difference in subkeys i and $i+4$. We would get a local collision for rounds 1–8, and then try to follow the most probable differential path to round 12 (with some probability $p_1 < 1$). From round 13 to round 16, we could try to find another most probable differential path, in order to exploit another local collision from round 17 to round 24. If everything worked out that way, we would get a boomerang distinguisher with the probability $p_1^2 \cdot p_2^2$.

To defend against that kind of boomerang attack, our key schedule ensures a distance between zero-difference subkeys of at least seven subkeys.

9.5 Third-Party Cryptanalysis

In September 2009, a group of six researchers published several attacks on reduced-round versions of Threefish, focusing on the Threefish-512 [2]. They managed to turn our 8-round pseudo-near-collisions with Hamming-weight 1 into a 17-round pseudo-near-collision, albeit with a much larger Hamming weight. They also improved our related-key distinguisher for 17 rounds to 21 rounds, and presented related-key key-recovery attacks on 25 and 26 rounds of Threefish-512. The 26-round attack needs time $2^{507.8}$, exhaustive search. They also found a 21-round related-key impossible differential, a 32-round related-key boomerang key recovery attack, a 34-round related-key boomerang distinguisher, and a 35-round "known-related-key boomerang distinguisher."

Even though these results apply to Threefish with the deprecated rotation constants (see Section D.1, it is straightforward to apply most of their ideas to analyze Threefish with its new rotation constants. In fact, when outlining the attacks, [2] gives a brief description of Threefish-512 without concrete values for the rotation constants. Indeed, one can understand how the attacks work without knowing the rotation constants.

The core of the 17-round pseudo-near-collision attack is the same eight-round local collision we used for our 8-round near-collision. To push the pseudo-near-collision from 8 to 17 rounds, the authors of [2] prepend a 4-round differential trail with probability 2^{-33}, and append a 5-round differential trail with probability 2^{-24}. These trails were found by linearization; i.e., by treating the addition mod 2^{64} like an XOR and then computing the probability that it actually behaves like that. Adapting the attack to the new rotation constants just requires repeating the linearization step to find new differential trails, possibly with slightly different probabilities.

We used the eight-round local collision as a tool to build a 17-round distinguisher. Similarly, the authors of [2] used their 17-round near-collision to build a 21-round distinguisher. To find a good distinguishing property, they ran a frequency test, searching for some highly biased bit—somewhat similar to our experiments in Section 9.3.2. To deal with the new rotation constants, one would essentially have to run a new frequency test.

The paper's impossible differential employs a 13-round forward differential and a 7-round backward differential. For the forward differential, the first ten rounds do not depend on the rotation constants, but the last three rounds do. Similarly, the last four rounds of the backward differential do not depend on the rotation constants, but the first three do. It is not clear to us whether the impossible differential can be modified for the new rotation constants.

The paper's key-recovery attack from improves our own key-recovery attack through a careful analysis—we only provided a sketch—and an improved search strategy that identifies and exploits neutral key bits. We believe that the same approach would work for the new rotation constants much as it works for the deprecated ones. One would need to rerun the frequency test to identify some bits with a significant bias. Since the attack workload for the 26-round attack is already close to an exhaustive key search, a slightly lower bias could possibly imply that the adapted attack would no longer be applicable for 26 rounds of Threefish-512, but it certainly remains applicable for 25 rounds.

The paper also describes a boomerang property based on concatenating two differential trails. Each of these trails goes through some key-dependent rounds. In contrast to the trails for their local collision, the authors used a complex technique for finding good boomerang trails; namely, the Lipmaa-Moriai algorithms for finding good XOR-approximations for the addition modulo powers of two [65]. To deal with the new rotation constants, one would have to run their software again, to calculate new differential trails. The probabilities of the new trails might vary slightly from the old trails' probabilities, but otherwise the boomerang property is the same.

Exploiting a boomerang property for key-recovery, related-key, and known-related-key distinguishing is straightforward, regardless of the rotation constants.

The authors of [2] presented some excellent ideas for analyzing Threefish. Their analysis used the now-deprecated rotation constants, but most of their attacks depend on the structure of Threefish and on the key schedule—with the possible exception of the impossible differential. We are confident that one can easily adopt the attacks to Threefish-512 and its new rotation constants, mainly by finding new differential trails and performing new frequency tests.

9.6 Empirical Observations for Threefish with Random Rotation Constants

After empirically studying the diffusion properties of the standard set of rotation constants, it may be interesting to also consider alternative choices for these constants. What would happen if we used random rotation constants, instead of our optimized constants? In other words, how critical is the choice of rotation constants for the security of Threefish?

To explore this question, we performed the same empirical tests on several sets of random rotation constants that we did for our current set in Section 9.3 and for the deprecated set in Section D.1. For the sake of simplicity, we focused on Threefish-256. All experiments we describe in this section have been performed with a sample size of 25 million pairs.

Recall the following facts:

1. The rotation constants repeat every eight rounds.
2. Each round of Threefish-256 employs two mix-operations.
3. Each mix operation performs one addition, one XOR and one rotation. That is, each mix operation requires one rotation constant.

Thus, we can write the rotation constant as an eight-tuple of pairs of integers between 0 and 63. The original set of rotation constants for Threefish-256 is

$$\text{Threefish-256} := ((14,16),(52,57),(23,40),(5,37),(25,33),(46,12),(58,22),(32,32)).$$

Our experiments started with random rotation constants, derived from the decimal representation of π, e, and $\sqrt{2}$, respectively. We studied sets of rotation constants that repeat after eight rounds (as do the constants in Threefish) and four rounds.

The random rotation constants were only marginally less secure than the standard rotation constants, by providing somewhat better 17-round distinguishers. As Table 25 illustrates, none of these sets of rotation constants allowed us to distinguish more than 17 rounds (i.e., we could not identify any bit with a bias > 0.001 in round 18), even those which repeated after four rounds.

To improve the attacks, or rather, to weaken Threefish, we then considered random sets of rotation constants that repeated after two rounds. As shown in Table 26, this finally allowed us to improve the distinguisher to 18 and sometimes 19 rounds. One of our sets was ((17, 23), (32, 32), (17, 23), (32, 32),...), which we expected to perform worse than the others, as the effect of a rotation by 32 could cancel out two rounds later, when the next rotation by 32 takes place.[7] But even for this special case, we could not identify a distinguisher for more than 19 rounds.

In order to explore the limits of Threefish, we then tried to be even more malicious. Could we get improved distinguishers if we sabotaged Threefish by simply repeating the same rotation constant again and again? The surprising answer is that the random constants we tried—9, 14, 23, and 59—did not weaken Threefish nearly as much as we had expected. As shown in Table 27, we only found distinguishers for at most 26 rounds. Specific choices were worse, For example, the constant 3 allowed for a 34-round distinguisher. And the constant 32 was as bad as we had anticipated: we could distinguish 40 rounds.

[7] Studying such sets of rotation constants can be of interest, since these could possibly improve the performance of Threefish on 32-bit machines.

round	# bits with bias				average
r	full	> 0.1	> 0.01	> 0.001	bias
Threefish-256 (standard constants)					
16:	0	3	41	63	0.00351
17:	0	0	0	7	0.00012
18:	0	0	0	0	0.00008
consants derived from π (repeats every 8 rounds)					
((14,15), (35,32), (38,46), (26,43), (38,32), (50,28), (19,39), (10,58))					
16:	0	6	42	67	0.00525
17:	0	0	0	10	0.00013
18:	0	0	0	0	0.00008
constants derived from e (repeats every 8 rounds)					
((59,04), (52,35), (36,02), (47,13), (52,24), (47,09), (36,62), (24,07))					
16:	0	20	67	95	0.01180
17:	0	0	5	25	0.00039
18:	0	0	0	0	0.00007
constants derived from $\sqrt{2}$ (repeats every 8 rounds)					
((41,42), (13,56), (23,09), (50,48), (16,42), (09,56), (53,17), (24,46))					
16:	0	6	53	82	0.00579
17:	0	0	3	12	0.00027
18:	0	0	0	0	0.00008
constants derived from π (repeats every 4 rounds)					
((14,15), (35,32), (38,46), (26,43) (14,15), (35,32), (38,46), (26,43))					
16:	0	5	44	91	0.00524
17:	0	0	0	10	0.00017
18:	0	0	0	0	0.00008
constants derived from e (repeats every 4 rounds)					
((59,04), (52,35), (36,02), (47,13), (59,04), (52,35), (36,02), (47,13))					
16:	0	32	74	113	0.02048
17:	0	3	16	33	0.00210
18:	0	0	0	0	0.00008
constants derived from $\sqrt{2}$ (repeats every 4 rounds)					
((41,42), (13,56), (23,09), (50,48), (41,42), (13,56), (23,09), (50,48))					
16:	0	10	55	93	0.00792
17:	0	0	6	25	0.00052
18:	0	0	0	0	0.00007

Table 25: Empirical results for variants of Threefish with random rotation constants, sample size 25,000,000 pairs.

round	# bits with bias				average
r	full	> 0.1	> 0.01	> 0.001	bias
Threefish-256 (standard constants)					
16:	0	3	41	63	0.00351
17:	0	0	0	7	0.00012
18:	0	0	0	0	0.00008
constants derived from π (repeats every 2 rounds)					
((14,15), (35,32), (14,15), (35,32), (14,15), (35,32), (14,15), (35,32))					
16:	0	94	127	154	0.08354
17:	0	49	83	107	0.02986
18:	0	4	33	56	0.00357
19:	0	0	5	18	0.00035
20:	0	0	0	0	0.00008
constants derived from e (repeats every 2 rounds)					
((59,04), (52,35), (59,04), (52,35), (59,04), (52,35), (59,04), (52,35))					
16:	0	31	80	119	0.02318
17:	0	3	20	43	0.00288
18:	0	0	0	1	0.00008
19:	0	0	0	0	0.00008
constants derived from $\sqrt{2}$ (repeats every 2 rounds)					
((41,42), (13,56), (41,42), (13,56), (41,42), (13,56), (41,42), (13,56))					
16:	0	41	99	125	0.02551
17:	0	4	28	70	0.00366
18:	0	0	1	9	0.00016
19:	0	0	0	0	0.00008
((17,23), (32,32), (17,23), (32,32), (17,23), (32,32), (17,23), (32,32))					
16:	0	83	164	193	0.05918
17:	0	26	73	130	0.01734
18:	0	2	22	68	0.00193
19:	0	0	0	8	0.00011
20:	0	0	0	0	0.00007

Table 26: Empirical results for variants of Threefish with rotation constants repeating every two rounds, sample size 25,000,000 pairs.

round	# bits with bias				average
r	full	> 0.1	> 0.01	> 0.001	bias
constant 9					
((09,09), (09,09), (09,09), (09,09), (09,09), (09,09), (09,09), (09,09))					
20:	0	4	53	115	0.00448
21:	0	0	11	66	0.00090
22:	0	0	0	12	0.00014
23:	0	0	0	0	0.00008
constant 14					
((14,14), (14,14), (14,14), (14,14), (14,14), (14,14), (14,14), (14,14))					
20:	0	12	65	112	0.00796
21:	0	1	25	68	0.00189
22:	0	0	3	25	0.00029
23:	0	0	0	0	0.00008
constant 23					
((23,23), (23,23), (23,23), (23,23), (23,23), (23,23), (23,23), (23,23))					
22:	0	4	40	74	0.00409
23:	0	0	8	33	0.00057
24:	0	0	0	13	0.00018
25:	0	0	0	1	0.00008
26:	0	0	0	0	0.00008
constant 59					
((59,59), (59,59), (59,59), (59,59), (59,59), (59,59), (59,59), (59,59))					
23:	4	14	29	46	0.01614
24:	0	4	23	37	0.00278
25:	0	0	6	17	0.00041
26:	0	0	0	2	0.00009
27:	0	0	0	0	0.00008
constant 3					
((03,03), (03,03), (03,03), (03,03), (03,03), (03,03), (03,03), (03,03))					
31:	0	12	32	58	0.00595
32:	0	0	22	43	0.00167
33:	0	0	4	29	0.00035
34:	0	0	0	9	0.00011
35:	0	0	0	0	0.00008
constant 32					
((32,32), (32,32), (32,32), (32,32), (32,32), (32,32), (32,32), (32,32))					
33:	0	4	52	86	0.00556
⋮					⋮
38:	0	0	0	27	0.00024
39:	0	0	0	15	0.00014
40:	0	0	0	2	0.00010
41:	0	0	0	0	0.00009

Table 27: Empirical results for variants of Threefish with a single rotation constant, sample size 25,000,000 pairs.

It is possible to do even worse. Constant 0 would be trivially weak, since the least significant bits in all words are linear. Given our related-key differential, constant 63 would need at least from round 11 to round 74 to propagate the difference from the most significant bits to the least significant bits. Since the full cipher only has 72 rounds, attacking that variant of Threefish would also be easy. Constant 1 is not quite as bad as 0 or 63, but diffusion would still be very slow. We conjecture that related-key attacks against a Threefish-256 variant with one-bit rotations and 72 rounds would be practical.

In summary, we searched for sets of rotation constants that would endanger or break Threefish. The only such sets we could actually identify were *obviously* malicious. Finding innocent-looking but dangerous sets of rotation constants for Threefish is still an open problem.

9.7 Cryptanalysis Summary

As our own cryptanalysis showed, it is feasible—in fact, quite easy—to create pseudo-near-collisions and pseudo-near-second-preimages for up to eight rounds of any variant of Skein; or rather, of the Skein compression function. Here, "near" means Hamming-distance 2. Using techniques similar to those from Section 9.3.4, one can push this from eight rounds to twelve rounds, at the cost of some significant but feasible amount of work. Assuming close to 2^n units of work, it may even be possible to find pseudo-near-second-preimages for up to sixteen rounds of the Skein-n compression function, for $n = 256$, $n = 512$, or $n = 1024$.

We stress that none of these attacks are applicable to reduced-round versions of the Skein hash function itself. Our current attacks only deal with reduced-round versions of the compression function. Due to Skein's output transformation, it remains an open problem how to create collisions or second preimages for the Skein hash function, even if one can create pseudo-collisions or pseudo-second-preimages for the compression function.

We invite the reader to compare this to recent attacks on the security of the SHA-2 hash function family. The best implementable attacks on SHA-256 and SHA-512 we are aware of can generate collisions for up to 24 rounds of both SHA-256 and SHA-512 [99]. The time required for these attacks is between $2^{15.5}$ (for SHA-256, using a huge table for a speed-up trick) and $2^{32.5}$ (for SHA-512, without the huge table). As SHA-256 has 64 rounds, and SHA-512 80 rounds, these attacks are far from actually endangering any member of the SHA-2 family.

Regarding the Threefish block cipher, we have discussed attacks for Threefish reduced to 24 to 26 rounds. Namely, the attacks were for 24 rounds of Threefish-256 (full cipher: 72 rounds), for 25 rounds of Threefish-512 (full cipher: 72 rounds), and for 26 rounds of Threefish-1024 (full cipher: 80 rounds). As cryptosystem designers, we are driven by reasonable pessimism. These attacks depend on certain optimistic assumptions by the adversary.

Additionally, we studied related-key boomerang attacks against Threefish using a modified (i.e., broken) key schedule. For that broken variant, we described a probability-1 distinguisher for 16 rounds, and outlined how one might push this through a few more rounds when allowing smaller probabilities instead of probability 1. Because of the choice of our key schedule, one cannot actually apply these attacks to unmodified Threefish.

The authors of [2] presented some interesting ideas to improve the cryptanalysis of Threefish. Their best chosen-plaintext related-key key-recovery attack works for 26 rounds of Threefish-512, improving on our own conjectured attack by one round. Their best chosen plaintext/chosen ciphertext key-recovery attack is a boomerang attack on 32 rounds of Threefish-512. They extend this to a known-related-key distinguisher for 35 rounds. These attacks mostly exploit the Threefish structure and properties of the key schedule. One can adapt their observations and attacks the new rotation constants, with the possible exception of the impossible differential attack. In some cases, the adaption may slightly change the number of rounds for which the attacks can be applied.

As far as we know, the best attack on SHACAL-2, the block cipher inside SHA-256, penetrates 44 rounds [67]—more than two-thirds of the full 64 rounds. However, since it requires 2^{233} related-key chosen plaintexts and time $2^{497.2}$, the attack is entirely academic. It is based on the related-key rectangle attack scenario, using a probability 2^{-460} distinguisher for 35 rounds of SHACAL-2. Note that related-key rectangle attacks are close relatives to related-key boomerang attacks, considered by ourselves and in [2], for modified Threefish.

10 Skein Website

The Skein website is `http://www.skein-hash.info/`. In addition to the latest version of this paper, the website contains reference code, optimized code, and code to generate performance measurements, test vectors, and known answer tests. We will continue to update the page with additional security proofs, cryptanalysis results, performance measurements, implementations, and so on.

The website is always the source for the most up-to-date version of this paper, and the most up-to-date information about Skein.

11 Legal Disclaimer

To the best of our knowledge, neither the Skein hash function, the Threefish block cipher, the UBI hashing mode, nor our optional argument system, are encumbered by any patents. We have not, and will not, apply for patents on any part of our design or anything in this document, and we are unaware of any other patents or patent filings that cover this work. The example source code—and all other code on the Skein website—is in the public domain and can be freely used.

We make this submission to NIST's hash function competition solely as individuals. Our respective employers neither endorse nor condemn this submission.

12 Acknowledgements

We would like to thank NIST for overseeing this competition, and our employers for allowing us the time to work on this submission. We would also like to thank the external reviewers who analyzed our algorithm when it was in draft—Frederik Armknecht, Martin Cochran, Hal Finney, Gary Graunke, Susan Landau, Sascha Müller-Lobeck, Kerry McKay, and Ray Savarda—and the following students of Stefan Lucks: Ewan Fleischmann, Christian Forler, Michael Gorski, Dennis Hoppe, Martin Kausche, Stoyan Stoyanov, and Julian Seifert. Also Beth Friedman for editing the submission document with Sue Heim's valuable suggestions. And finally, we would like to thank our respective families for putting up with all of the time and attention this project required.

13 About the Authors

The Skein team is essentially a group of friends. We've all worked on cryptography and cryptographic engineering for many years. We've met and worked together many times; our team includes half of the Twofish team [100]. Our experiences are extensive and diverse, which was a great help in bringing all aspects of the design together. It also led to some very interesting discussions: a single e-mail thread might span mathematical proofs, PR, and political considerations, and discussions on how modern CPUs work. We had lots of fun.

We realize our affiliations read like a powerful industry consortium, but we are not. Our employers kindly agreed to let us do this work, but most of it was done on our own time. Really, they only have the vaguest idea what we're doing.

Appendix A Overview of Symbols

This appendix gives an overview and index of the symbols used in the definition of Skein.

BytesToWords A function that converts a string of bytes to a string of 64-bit words. [Page 9].

C The Threefish ciphertext [Page 11] or the configuration string [Page 15].

c_i The words of ciphertext C. [Page 11]

$e_{d,i}$ The ith word of the result of the subkey addition (if any) in round d. [Page 10]

d The round number. [Page 10]

$f_{d,i}$ The ith word of the result of the MIX functions in round d. [Page 10]

G_i Chaining values between different UBI invocations. [Page 16]

H_i Chaining values used within a UBI computation. [Page 13]

K The key, either the Threefish key [Page 10] or the Skein key. [Page 16]

K' The processed key that starts the Skein UBI chain. [Page 16]

k_i The words of the Threefish key K. [Page 10]

$k_{s,i}$ The words of subkey s. [Page 12]

M Used for various message strings.

M_i Block i of message string M.

N_b The number of bytes in the state. [Page 12]

N_o The number of output bits of Skein. [Page 14]

N_r The number of rounds in Threefish. [Page 10]

N_w The number of words in the state. [Page 10]

P The plaintext input to Threefish. [Page 10]

p_i The words of plaintext P. [Page 10]

$\pi(i)$ The permutation applied to the state words in each round. [Page 11]

$R_{d,j}$ The rotation constant for mix j in round d. [Page 11]

s The subkey number. [Page 10]

T The tweak value. [Page 13]

T_s The starting tweak value for UBI. [Page 12]

T_{xxx} Various type value constants. [Page 14]

t_i The words of tweak T. [Page 10]

ToBytes A function that converts an integer to a string of bytes, LSB first. [Page 9]

ToInt A function that converts a string of bytes to an integer, LSB first. [Page 9]

$v_{d,i}$ The value of the ith word of the Threefish encryption state after d rounds. [Page 10]

WordsToBytes A function that converts a string of 64-bit words to a string of bytes. [Page 9]

(x_0, x_1) The inputs to a MIX function. [Page 11]

Y_f Encoding of the fan-out for tree hashing. [Page 17]

Y_l Encoding of the leaf node size for tree hashing. [Page 17]

Y_m Maximum tree height for tree hashing. [Page 17]

(y_0, y_1) The outputs of a MIX function. [Page 11]

Appendix B Initial Chaining Values

These are the IV values for the configurations in Table 1. These constants are the output of the configuration UBI. If you are using Skein as a normal hash function, you can use these IV values as constants and skip the configuration step entirely. Note that these are 64-bit words, not byte strings.

B.1 Skein-256-128

```
0x46B39C3AAA418D4F, 0x681229DD06920827, 0xCBE067C978460238, 0xC388A1B74EC45EF3
```

B.2 Skein-256-160

```
0xD51846B9DAE51FBB, 0x7D47BABD6205526D, 0xA1A8703E47B89F20, 0xB97D7234C5927589
```

B.3 Skein-256-224

```
0xFE6720F45ED90A57, 0x352D51F3B01B6FBC, 0xD764B04F1785F14E, 0xE7F24611DDD59B27
```

B.4 Skein-256-256

```
0x164290A9D4EEEF1D, 0x8E7EAF44B1B0CD15, 0xA8BA0822F69D09AE, 0x0AF25C5E364A6468
```

B.5 Skein-512-128

```
0x51AF0A1B97A7DA9C, 0xEC77F8A5F4C6004C, 0x0BB7182C25CA1F6E, 0x1B22A2CB9F9339C5
0xC905E0A431216AA4, 0xAEE4D5D0BD378696, 0x92744A501953D08A, 0x2DCAD6F985777E17
```

B.6 Skein-512-160

```
0x9A73479AC7701247, 0xD657FBF8FDE0DA1A, 0xB1EE72A6B04DA375, 0xE87ED2A1C20605B8
0x220A0EFA9B925E17, 0x6D72A217EAF0B419, 0x6CD72290AA33FA72, 0x5829089E759C4256
```

B.7 Skein-512-224

```
0x10C550456BF94560, 0x59004AF1F558ACCC, 0x82BD1BF9B7461DFD, 0x46B0F3A47C2AF60E
0xECC8498CE80A8DCA, 0x50A1DA3310C836EF, 0x3538F92A39165A80, 0x896A4329CD5DCF2A
```

B.8 Skein-512-256

```
0x85A195B18B2264EC, 0x7A6DAC64C047C2B0, 0xE1A21465EE3FE124, 0x1D2117356504425A
0xC962DC0FC0046F2C, 0x8D5A3E904B1BE9C8, 0xAFB7174BBD8FEEE9, 0x7FE63D9BF94EDEB8
```

B.9 Skein-512-384

```
0x755C495716D7512B, 0xB458712714DF4CEF, 0x677D2E8C027C060A, 0x8DA4F59205232716
0xCE454B58C445AD7F, 0x23048344ACA8BC96, 0xF719BCC338768323, 0xD77E368650579DEC
```

B.10 Skein-512-512

```
0x1A9A721C8A265CA5, 0xC9ABACF5AA853978, 0x4AF6652AB80A2883, 0x66F5E8A809A773C7
0x7FA984B781BAAF5B, 0x0FE5D2D93233F397, 0x6E29F932DCB412D7, 0xD40CD9472F225C23
```

B.11 Skein-1024-384

```
0x9E887D472693F556, 0xF4553A5AB3A902D8, 0x60A1079028E4504E, 0x96FAA39D943F8ABE
0x2A769D27828A22A7, 0xB2F274F5B2C3A833, 0xC722C05247F09222, 0x377C4A92EE78B216
0x97CFE7B2039F4C9D, 0xC864ACFAC83C8364, 0x73F265791D3CF723, 0x2464DC1E5E327F97
0x135D3954F181CB1A, 0x244BBF1324C5C669, 0xE1E258BC446662E3, 0xCF1E0F47934A469C
```

B.12 Skein-1024-512

```
0x76066F1F612DD519, 0xD9B93D9575D90191, 0x582D15EA89696586, 0x4F1CA328B5F10FB3
0x686C454DEC64B419, 0x2D7BD9B4026EDABE, 0xEF3461951ACD05C4, 0x1759E8984446E275
0xACFC075AE724456D, 0x82F35D0AE7704311, 0x99D0B1039AD7E344, 0x85D6C81D29F6204B
0x0CA2A9875D57632A, 0x069A893147A448FA, 0x3C42FB5002815320, 0xF7E22C15953E3125
```

B.13 Skein-1024-1024

```
0x495E85B953876965, 0x1E3D5C1B41E754EF, 0x237254552E9C10C7, 0x0B00AAB4FA441407
0x17DDA56AA106337C, 0xF98200E9CAE13F94, 0xF2DF7F00ADFF12BF, 0xA92673D0D0CA7AD9
0xC0DD64B04B27ED98, 0x87C36A6CA0A26F90, 0x640C8526D0850A10, 0x6EBFAD0C93DA09AE
0x617E3BCDDEE4A85F, 0x05A4A1A7D82737B7, 0x002BAF2C3EB13D30, 0x28527A78C83D554C
```

Appendix C Test Vectors

C.1 Skein-256-256

```
Message data:
     FF

Result:
     42 C8 82 37 B6 3F C9 9C 55 09 88 38 A1 71 D5 0B
     FB FF CA F3 40 70 5C 92 3B BB 37 45 D1 47 15 E8

Message data:
     FF FE FD FC FB FA F9 F8 F7 F6 F5 F4 F3 F2 F1 F0
     EF EE ED EC EB EA E9 E8 E7 E6 E5 E4 E3 E2 E1 E0

Result:
     E3 3F 48 3B 13 89 BA 9F AE FF 25 25 7E 87 CF 76
     00 8C 35 28 5E 3B D5 62 BD C1 F3 EA 2A 96 06 35

Message data:
     FF FE FD FC FB FA F9 F8 F7 F6 F5 F4 F3 F2 F1 F0
     EF EE ED EC EB EA E9 E8 E7 E6 E5 E4 E3 E2 E1 E0
     DF DE DD DC DB DA D9 D8 D7 D6 D5 D4 D3 D2 D1 D0
     CF CE CD CC CB CA C9 C8 C7 C6 C5 C4 C3 C2 C1 C0

Result:
     90 E5 0C 4D CF C7 49 0A 09 F3 A1 A7 9B F3 B3 DF
     21 EA 85 44 7B 0F F0 29 C8 47 D6 59 85 6E C7 A5
```

C.2 Skein-512-512

```
Message data:
     FF

Result:
     42 AA 6B D9 CA 92 E9 0E A2 8D F6 F6 F2 D0 D9 B8
     5A 2D 19 07 EE 4D C1 B1 71 AC E7 EB 11 59 BE 3B
     D1 BC 56 58 6D 92 49 2B 6E FF 9B E0 33 06 99 4C
     65 A3 32 C4 C2 41 60 F4 66 55 04 0E 55 8E 83 29

Message data:
     FF FE FD FC FB FA F9 F8 F7 F6 F5 F4 F3 F2 F1 F0
     EF EE ED EC EB EA E9 E8 E7 E6 E5 E4 E3 E2 E1 E0
     DF DE DD DC DB DA D9 D8 D7 D6 D5 D4 D3 D2 D1 D0
     CF CE CD CC CB CA C9 C8 C7 C6 C5 C4 C3 C2 C1 C0
```

```
Result:
     04 F9 6C 6F 61 B3 E2 37 A4 FA 77 55 EE 4A CF 34
     49 42 22 96 89 54 F4 95 AD 14 7A 1A 71 5F 7A 73
     EB EC FA 1E F2 75 BE D8 7D C6 0B D1 A0 BC 60 21
     06 FA 98 F8 E7 23 7B D1 AC 09 58 E7 6D 30 66 78

Message data:
     FF FE FD FC FB FA F9 F8 F7 F6 F5 F4 F3 F2 F1 F0
     EF EE ED EC EB EA E9 E8 E7 E6 E5 E4 E3 E2 E1 E0
     DF DE DD DC DB DA D9 D8 D7 D6 D5 D4 D3 D2 D1 D0
     CF CE CD CC CB CA C9 C8 C7 C6 C5 C4 C3 C2 C1 C0
     BF BE BD BC BB BA B9 B8 B7 B6 B5 B4 B3 B2 B1 B0
     AF AE AD AC AB AA A9 A8 A7 A6 A5 A4 A3 A2 A1 A0
     9F 9E 9D 9C 9B 9A 99 98 97 96 95 94 93 92 91 90
     8F 8E 8D 8C 8B 8A 89 88 87 86 85 84 83 82 81 80

Result:
     B4 84 AE 9F B7 3E 66 20 B1 0D 52 E4 92 60 AD 26
     62 0D B2 88 3E BA FA 21 0D 70 19 22 AC A8 53 68
     08 81 44 BD F4 EF 3D 98 98 D4 7C 34 F1 30 03 1B
     0A 09 92 F0 9F 62 DD 78 B3 29 52 5A 77 7D AF 7D
```

C.3 Skein-1024-1024

```
Message data:
     FF

Result:
     A2 04 C3 94 92 34 16 B8 20 68 6E 1B 44 9C A4 27
     7A 47 7C A9 DB 08 4D D9 31 71 5B 10 2A 43 F0 45
     BC 34 83 AE E0 28 86 08 91 67 10 F8 75 DF 8D 79
     42 49 97 46 53 28 B5 17 BA C3 09 96 FB FA 64 2B
     CC 44 6E 15 7A 45 D0 2F 08 78 DA A3 55 61 39 A7
     6C 2C FE C2 83 07 21 96 B3 3A 12 83 54 CB DB 6D
     B9 DB A8 79 C7 1C B1 0C 77 E9 29 78 6D 2F BA FD
     E6 A0 55 29 E2 6B 20 FF 3B BB 7B 47 5C AA 71 15

Message data:
     FF FE FD FC FB FA F9 F8 F7 F6 F5 F4 F3 F2 F1 F0
     EF EE ED EC EB EA E9 E8 E7 E6 E5 E4 E3 E2 E1 E0
     DF DE DD DC DB DA D9 D8 D7 D6 D5 D4 D3 D2 D1 D0
     CF CE CD CC CB CA C9 C8 C7 C6 C5 C4 C3 C2 C1 C0
     BF BE BD BC BB BA B9 B8 B7 B6 B5 B4 B3 B2 B1 B0
     AF AE AD AC AB AA A9 A8 A7 A6 A5 A4 A3 A2 A1 A0
     9F 9E 9D 9C 9B 9A 99 98 97 96 95 94 93 92 91 90
```

```
    8F 8E 8D 8C 8B 8A 89 88 87 86 85 84 83 82 81 80

Result:
    C2 E6 B6 FC 04 2F 86 F2 E3 17 38 64 1D B6 02 95
    F7 42 04 AB 52 58 95 A5 DE C5 C8 06 AC 47 86 EC
    1C 98 29 20 09 5B 71 29 FE 3D 8B D4 51 F6 7E A3
    13 20 C7 8B 11 57 5E A6 DD E3 94 E7 5D C5 F5 C9
    6A 51 04 38 6D D5 50 16 D4 94 DF FA C5 AD 11 9B
    22 C9 60 DC 46 B6 58 CF 2C EB 7D 73 AF 0F D0 E1
    9C 7E 21 34 4A AD 06 AF 39 FC BE F6 C6 C5 D0 0D
    E8 96 B8 88 D9 54 56 DE DB A6 E5 37 7B 0C C5 72

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

Result:
    64 66 1F 7D C4 AB BB 50 00 3E AE A6 92 42 01 8D
    27 AE B4 15 CA 7B 89 1F BD DB C1 C6 90 40 D4 C4
    A9 82 21 33 E6 0E 22 2A F7 EE 09 34 9C 3F 1C 4C
    C8 F3 81 11 A5 05 E8 63 93 E7 28 41 80 60 2D B6
    30 AE 76 3D 9A 62 0A E4 93 2A 82 34 15 B5 95 15
    AE EC 55 64 18 ED 2E F0 82 71 89 D5 4E 65 76 D4
    1D 96 1E DD CB BD 87 1E 6F 1B 6B A6 25 DC 68 4C
    BB C2 04 88 8B 18 68 7E D6 27 50 60 70 00 6B 82
```

Appendix D NIST SHA-3 Round 2 Tweak: Rotation Constants

This specification of Skein includes a “tweak” for Round 2 of the NIST SHA-3 competition: the rotation constants shown in Table 4, which differ from those originally submitted to NIST in October 2008 [33]. All existing and future implementations of Skein *must* now use these new rotation constants to be compliant. Changing the rotation constants required updating all the Skein test vectors (Appendix C) and the precomputed initial chaining values (Appendix B).

Some comments are in order about the tweak. It is our belief is that the structure of Threefish would be secure with almost any randomly chosen set of rotation constants (see Section 9.6). Indeed, it would give us pause if a randomly generated set of rotation constants were to result in an attack.

Nonetheless, given that we have an opportunity to select the rotation constants, it makes sense to maximize diffusion as much as possible. In the original NIST submission, we had limited time to design and run a search algorithm for rotation constants. Some time after the initial submission, Guillaume Sevestre contacted us [102] with a suggestion for a new search algorithm—the evolutionary algorithm described in Section 8.3—which indeed produced significantly better results using the original diffusion metric. Many thanks are due to Guillaume for collaborating at length to educate us on the new search algorithm and how to optimize it.

In addition, the metric used in the original search (see the definition of H_{min} in Section D.1) was not the most appropriate value to optimize. This metric minimized the probability that a particular input bit flip would *not* trigger a particular output bit flip, but it did not attempt to minimize the probability such a bit flip *would* occur. A slightly different metric, minimizing the maximum bias from $K/2$ in the histogram, was more consistent with what we were trying to achieve. Not surprisingly, we found that some of the original rotation constant sets resulted in rather poor values of the maximum bias due to cases where the output bit almost always flipped for a given input bit difference. Again, we do not expect that this bias using the original rotation constants would result in an attack, but as long as we were considering a tweak, it seemed best to use the bias as the new search metric.

When we ran the new search algorithm for two days using the improved bias metric, the gains in both diffusion metrics were generally impressive. A comparison is given in Table 28, with the bias listed as the maximum deviation from 0.5 of the value $x = h_i/K$ across the entire histogram, where h_i is an entry in the histogram. Thus, a smaller value in the table indicates better diffusion, and the worst possible value (i.e., 0.50) would indicate that at least one output bit location *never* (or always) changed for a given input bit difference. Again, the original bias metric only took into account values of $x < 0.5$, while the new metric includes both positive and negative deviations, so bias values obtained using the old metric will never be larger than those using the new metric. Note that a later six-day search improved the metric by only about 0.002 for Skein-256, 0.007 for Skein-512, and 0.014 for Skein-1024 over the two-day results (i.e., those in Table 4), so we are comfortable using the latter.

	Rotation Constants		
	Old Set		New Set
	Old Metric	New Metric	New Metric
Skein-256	0.10	0.24	0.09
Skein-512	0.33	0.48	0.25
Skein-1024	0.26	0.35	0.15

Table 28: Maximum observed bias for $K = 16384$

Given these two results and the fact that NIST has allowed tweaks at this point in the competition, we feel it would be somewhat irresponsible *not* to submit the improved rotation constant set. However, if NIST decides for some reason that changing the Skein rotation constants is not allowed as a tweak, then we will happily stay with the original Skein definition.

If NIST accepts the tweak, we recommend that NIST consider as relevant any attack which uses either the old or the new rotation constants, so that all cryptanalysis efforts on the original rotation constants are still valid.

D.1 Deprecated Skein Rotation Constants

For historical reference, this section includes excerpts from originally submitted Skein specification [33], showing the (now deprecated) rotation constants in Table 29, as well as the original description of the search algorithm used to generate them.

N_w		4		8				16							
j		0	1	0	1	2	3	0	1	2	3	4	5	6	7
	0	5	56	38	30	50	53	55	43	37	40	16	22	38	12
	1	36	28	48	20	43	31	25	25	46	13	14	13	52	57
	2	13	46	34	14	15	27	33	8	18	57	21	12	32	54
$d =$	3	58	44	26	12	58	7	34	43	25	60	44	9	59	34
	4	26	20	33	49	8	42	28	7	47	48	51	9	35	41
	5	53	35	39	27	41	14	17	6	18	25	43	42	40	15
	6	11	42	29	26	11	9	58	7	32	45	19	18	2	56
	7	59	50	33	51	39	35	47	49	27	58	37	48	53	56

Table 29: Deprecated Skein rotation constants $R_{d,j}$ for each N_w.

Our goal was to choose rotation constants that maximized diffusion across the entire cipher. We used a three-phase process to select the final set.

In phase one, we selected candidate sets of rotation constants that maximized the Hamming weight of a simplified version of Threefish. In this modified version, we replaced the addition and XOR operations in the Threefish MIX function with the logical OR operation. We then generated a random set of rotation constants and, using an all-zero plaintext, injected a single input bit difference at each input bit location. After R rounds, we measured the minimum Hamming weight of each of the N output words across all input difference locations. If the Hamming weight value was less than a threshold W, we rejected the rotation set and randomly chose another. If it was greater than or equal to W, we saved it for phase two.

We selected values of R and W empirically based on the block size. The general idea was to choose values that were at the knee of the diffusion curve. In other words, if we chose R to be too small, then all rotation sets looked alike. If we chose R to be too large, then the minimum Hamming weight quickly reached 64 bits. Similarly, if we chose W to be too small, then all rotation sets passed; and if we chose W to be too large, none passed. After some experimentation, we settled on the (R, W) sets of $(8, 61)$, $(8, 47)$, and $(9, 51)$ for Threefish-256, -512, and -1024, respectively.

Our search algorithm used a hill-climbing algorithm, initially accepting rotation constant sets with Hamming weight metric $(W - 4)$ and then trying to modify pairs of rotation constants in the set to walk up to the value W, and beyond, if possible. In our random selections, we rejected any rotation constants with value 0, $+1$, and -1, since the add and XOR operations in the MIX function already provided diffusion to adjacent bits.

Phase one was very useful as an initial filter because it was much faster than running the actual Threefish rounds, primarily because this metric is rotationally invariant. That is, we actually ran the diffusion test using only a single bit difference position per word, which sped up this phase by a factor of 64. We could also have used XOR instead of logical OR here, but the former would have included cancellations and hidden the true diffusion rate of a candidate set of rotation constants, so we felt that using OR was a better choice.

In phase two, we took all the sets of rotation constants collected in the first phase. We selected K random plaintexts and injected a small difference pattern in each possible input bit location, using the actual Threefish round function. We chose K to be 1024: small enough to run fairly quickly, but large enough to grade the rotation sets with reasonable probability.

At each bit position, we used an input difference pattern of up to three bits, with a nonzero difference in the first bit; i.e., the bit patterns 001, 011, 101, and 111. We generated a histogram for each output bit as to whether that bit changed for each input difference, after R rounds, ignoring the key injection. For example, in Threefish-512 this meant that the histogram had an array of 512x512 (256K) entries. We generated separate histograms for each input difference bit pattern, for a total of four different histograms per rotation constant set.

For a truly random function, the expected value for each histogram entry would be $K/2$ with a binomial distribution. Of course, with these small values of R the function is not truly random, but the goal was simply to choose a reasonable metric to grade the sets of rotation constants. For each set of rotation constants, we computed the minimum value, called H_{min}, across all four histograms, for K plaintexts. We retained the N_f rotations sets with maximum H_{min} value as "finalists" to use in phase three, with $N_f = 16$.

For each set of rotation constants selected in the first phase, the set of rotation constants generated by scaling by any fixed odd integer (mod 64) also has the same Hamming weight properties in the simplified OR-only version of Threefish. Therefore, in the second phase, we also tested all 32 such scaled versions for the best H_{min} value.

In phase three, we re-graded the N_f finalist sets of rotation constants using larger values of K—4096, 8192, and 16,384—to minimize the expected statistical sampling error. Based on the relative rankings of the rotation constant sets in phase three, we chose the winner. In the case of Threefish-256, choosing the winner was somewhat arbitrary, as there were several leading contenders with similar H_{min} values, and the relative rankings changed slightly for different values of K.

We ran this three-phase selection process for all three Threefish block sizes. The overall run time for the search was 2–3 days on an Intel Core 2 Duo CPU running in 64-bit mode, though this was actually split up and run on separate CPUs for the separate block sizes, to minimize elapsed time.

D.2 Empirical Cryptanalysis on the Deprecated Rotation Constants

For historical reference, this section describes a cryptanalytic experiment presented in the originally submitted Skein specification [33]. This analysis is updated for the new rotation constants in Section 9.3.

In 2008, Martin Kausche [47] performed a number of experiments regarding related-key attacks on reduced-round Threefish. We cited his results in [33], in the context of our own preliminary cryptanalysis of Threefish and Skein.

For each of Threefish-256, -512, and -1024, twenty million random pairs (20,000,000 $\approx 2^{24.25}$) with the specified difference in plaintext, key, and tweak were generated. The probability that the bit is the same in both ciphertexts of a ciphertext pair is written as $p^r_{w,b}$, and the bias is $|p^r_{b,w} - 0.5|$. Table 30, 31, and 32 summarize the results. The results we described above confirm what we already

round	maximum bias at		prob.	# bits with bias				average bias
r	word w	bit b	$p^r_{w,b}$	> 0.1	> 0.05	> 0.01	> 0.001	
9	0	0	1.00000	256	256	256	256	0.50000
10	0	0	1.00000	256	256	256	256	0.50000
11	0	0	1.00000	254	254	254	254	0.49218
12	0	0	1.00000	245	245	245	245	0.45322
13	0	0	1.00000	216	223	223	223	0.34278
14	2	2	0.00418	147	175	188	188	0.17837
15	2	1	0.97631	37	60	114	132	0.04378
16	0	1	0.38875	1	1	18	55	0.00285
17	2	0	0.52350	0	0	1	3	0.00020
18	3	17	0.49969	0	0	0	0	0.00009
19	3	35	0.49971	0	0	0	0	0.00009
20	0	43	0.49961	0	0	0	0	0.00009

Table 30: Empirical results for Threefish-256 with deprecated rotation constants [47], sample size 20,000,000 pairs.

round	maximum bias at		prob.	# bits with bias				average bias
r	word w	bit b	$p^r_{w,b}$	> 0.1	> 0.05	> 0.01	> 0.001	
9	0	0	1.00000	512	512	512	512	0.50000
10	0	0	1.00000	512	512	512	512	0.50000
11	0	0	1.00000	510	510	510	510	0.49609
12	0	0	1.00000	501	501	501	501	0.47666
13	0	0	1.00000	462	466	466	466	0.39772
14	0	42	0.99999	366	389	402	403	0.25770
15	2	1	0.00963	141	197	256	278	0.07316
16	4	0	0.06533	7	21	65	100	0.00723
17	0	4	0.49655	0	0	0	4	0.00011
18	6	59	0.50036	0	0	0	0	0.00009
19	6	32	0.50031	0	0	0	0	0.00009
20	2	58	0.50036	0	0	0	0	0.00009

Table 31: Empirical results for Threefish-512 with deprecated rotation constants [47], sample size 20,000,000 pairs.

pointed out in [33]: there is a significant bias for 17 rounds of both Threefish-256 and Threefish-512 that disappears in round 18 and later, and a significant bias for 18 rounds of Threefish-1024.

For each round r, Tables 30–32 describe the coordinates w and b of a bit with maximum bias and its actual probability. (There may be other bits with the same bias.) The table also gives the number of bits with "large" bias for each round; i.e., the number of bits with a bias exceeding 10%, 5%, 1%, and 0.1%, respectively. Furthermore, the tables gives the average bias, over all the 256/512/1024 bits considered. The tables focus on the "interesting" rounds.

round	maximum bias at		prob.	# bits with bias				average bias
r	word w	bit b	$p^r_{w,b}$	> 0.1	> 0.05	> 0.01	> 0.001	
9	0	0	1.00000	1024	1024	1024	1024	0.50000
10	0	0	1.00000	1024	1024	1024	1024	0.50000
11	0	0	1.00000	1022	1022	1022	1022	0.49805
12	0	0	1.00000	1013	1013	1013	1013	0.48829
13	0	0	1.00000	981	981	981	981	0.45041
14	0	0	1.00000	869	900	907	914	0.35832
15	9	0	1.00000	598	670	743	772	0.20242
16	0	1	0.97589	148	232	381	461	0.04239
17	10	1	0.70448	5	8	31	87	0.00173
18	6	0	0.48980	0	0	1	2	0.00010
19	3	13	0.50040	0	0	0	0	0.00009
20	7	15	0.50040	0	0	0	0	0.00009

Table 32: Empirical results for Threefish-1024 with deprecated rotation constants [47], sample size 20,000,000 pairs.

D.3 New Empirical Cryptanalysis on the Deprecated Rotation Constants

To provide additional comparison between the new and old set of rotation constants, we ran the same sets of experiments for Threefish with the deprecated rotation constants as we did for Threefish in Section 9.3 in Tables 22–24.

round	# bits with bias				average
r	full	> 0.1	> 0.01	> 0.001	bias
0–10:	256	256	256	256	0.50000
11:	204	254	254	254	0.49219
12:	77	242	242	242	0.43052
13:	21	223	223	223	0.34280
14:	0	175	188	188	0.17840
15:	0	60	121	132	0.04377
16:	0	1	30	50	0.00305
17:	0	0	1	3	0.00017
18:	0	0	0	0	0.00005
19:	0	0	0	0	0.00006
20:	0	0	0	0	0.00006

Table 33: Empirical results for Threefish-256 with deprecated rotation constants, sample size 50,000,000 pairs.

As it turned out, the deprecated and new 256-bit versions are basically the same: the new Skein-256 has a bit with bias > 0.01 in round 16, but the deprecated version has more bits with a bias > 0.001. The deprecated and new 512-bit versions are similarly the same.

For Skein-1024, we are able to distinguish 18 rounds of Threefish-1024 with the deprecated rotation constants from random. There is even one bit with a bias exceeding 0.01. Since we could not find a single bit with a bias exceeding 0.001 for 18 rounds of Threefish-1024 with the new rotation constants, the new version is narrowly ahead of the deprecated version.

round	# bits with bias				average
r	full	> 0.1	> 0.01	> 0.001	bias
0–10:	512	512	512	512	0.50000
11:	470	510	510	510	0.49609
12:	266	495	495	495	0.46008
13:	64	466	466	466	0.39772
14:	0	389	403	403	0.25769
15:	0	197	269	280	0.07315
16:	0	19	74	102	0.00678
17:	0	0	0	5	0.00008
18:	0	0	0	0	0.00006
19:	0	0	0	0	0.00005
20:	0	0	0	0	0.00005

Table 34: Empirical results for Threefish-512 with deprecated rotation constants, sample size 50,000,000 pairs.

round	# bits with bias				average
r	full	> 0.1	> 0.01	> 0.001	bias
0–10:	1024	1024	1024	1024	0.50000
11:	976	1022	1022	1022	0.49805
12:	735	1008	1008	1008	0.48120
13:	403	981	981	981	0.45041
14:	99	900	912	914	0.35832
15:	8	670	753	773	0.20241
16:	0	211	389	464	0.03858
17:	0	8	47	105	0.00171
18:	0	0	1	2	0.00007
19:	0	0	0	0	0.00006
20:	0	0	0	0	0.00005

Table 35: Empirical Results for Threefish-1024 with deprecated rotation constants, sample size 50,000,000 pairs.

References

[1] American Bankers Association, "Keyed Hash Message Authentication Code," ANSI X9.71, 2000.

[2] J. Aumasson, C. Calik, W. Meier, O. Ozen, R. Phan, and K. Varici, "Improved Cryptanalysis of Skein" `http://www.131002.net/papers.html`, submitted to the IACR eprint server, September 2009.

[3] E. Barker, D. Johnson, and M. Smid, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised)," NIST Special Publication SP 800-56A, Mar 2007.

[4] E. Barker and J. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators," NIST Special Publication SP 800-90, Mar 2007.

[5] M. Bellare, "New Proofs for NMAC and HMAC: Security without Collision-Resistance," *Advances in Cryptology—CRYPTO '06 Proceedings*, Springer-Verlag, 2006, pp. 602–619.

[6] M. Bellare, R. Canetti and H. Krawczyk, "Keying hash functions for message authentication," *Advances in Cryptology—CRYPTO '96 Proceedings*, Springer-Verlag, 1996 , pp. 1–15.

[7] M. Bellare, R. Canetti, and H. Krawczyk, "Pseudorandom Functions Revisited: The Cascade Construction and its Concrete Security," *Proceedings of the 37th Symposium on Foundations of Computer Science*, IEEE Press, 1996, pp. 514–523.

[8] M. Bellare, J. Kilian, and P. Rogaway. "The Security of Cipher Block Chaining," *Advances in Cryptology—CRYPTO '94 Proceedings*, Springer-Verlag, 1994, pp 341–358.

[9] M. Bellare, T. Kohno, S. Lucks, N. Ferguson, B. Schneier, D. Whiting, J. Callas, and J. Walker, "Provable Security Support for the Skein Hash Family," Version 1.0, Apr 2009, `http://www.skein-hash.info/sites/default/files/skein-proofs.pdf`.

[10] M. Bellare and T. Ristenpart, "Multi-Property-Preserving Hash Domain Extension and the EMD Transform," *Advances in Cryptology—ASIACRYPT '06 Proceedings*, Springer-Verlag, 2006, 299–314.

[11] M. Bellare and B. Yee, "Forward Security in Private Key Cryptography, " *Topics in Cryptology—CT-RSA*, Springer-Verlag, 2003, pp. 1–18.

[12] D.J. Bernstein, "Cache-Timing Attacks on AES," April 2005, `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[13] G. Bertoni, J. Daemen, M. Peeters, G. can Assche, "RadioGatún, a Belt-and-Mill Hash Function," *Second NIST Cryptographic Hash Workshop*, Santa Barbara, USA, 24–25 Aug 2006.

[14] E. Biham, "New Types of Cryptanalytic Attacks using Related Keys," *Journal of Cryptology*, v. 7, 1994, pp. 229–246.

[15] E. Biham and R. Chen, "Near-Collisions of SHA-0," *Advances in Cryptology - Crypto '04 Proceedings*, Springer-Verlag, 2004, pp. 290–305.

[16] E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*, Springer Verlag, 1993.

[17] A. Biryukov and D. Wagner, "Slide Attacks," *6th International Workshop on Fast Software Encryption*, Springer-Verlag, 1999, pp. 245–259.

[18] A. Biryukov and D. Wagner, "Advanced Slide Attacks," *Advances in Cryptology—EUROCRYPT '00 Proceedings*, Springer-Verlag, 2000, pp. 589–606.

[19] S. Micali and M. Blum, "How to Generate Cryptographically Strong Sequences of Pseudo-random Bits," *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science (FOCS '82)*, IEEE, 1982, pp. 112–117.

[20] J. Bonneau and I. Mironov, "Cache-Collision Timing Attacks Against AES," *Cryptographic Hardware and Embedded Systems–CHES 2006*, Springer-Verlag, 2006, pp. 201–215.

[21] C. Burwick, D. Coppersmith, E. D'Avidnon, R. Gennaro, S. Halevi, C. Jutla, S.M. Matyas, L. O'Connor, M. Peyravian, D. Stafford, and N. Zunic, "MARS—A Candidate Cipher for AES," NIST AES Proposal, Jun 1998.

[22] F. Chabaud and A. Joux, "Differential Collisions in SHA-0," *Advances in Cryptology: Eurocrypt '98 Proceedings*, Springer-Verlag, 1998, pp. 56–71.

[23] L. Chen, "Recommendation for Key Derivation Using Pseudorandom Functions," NIST Special Publication SP 800-108, Apr 2008.

[24] J. Coron, Y. Dodis, C. Malinaud, P. Puniya, "Merkle–Damgård Revisited: How to Construct a Hash Function," *Advances in Cryptology: CRYPTO 05 Proceedings*, Springer-Verlag, 2005, 430–448.

[25] J. Daemen, R. Govaerts, and J. Vanderwalle, "Correlation Matrices," *Fast Software Encryption 1994*, Springer-Verlag, 1995, pp. 275–285.

[26] J. Daemen and V. Rijmen, *The Design of Rijndael: AES—The Advanced Encryption Standard*, Springer-Verlag, 2002.

[27] I. Damgård. "A Design Principle for Hash Functions," *Advances in Cryptology: Crypto '89 Proceedings*, Springer-Verlag, 1990, pp. 416–427.

[28] Q. Dang, "Randomized Hashing for Digital Signatures," NIST Special Publication SP 800-106, Aug 2008.

[29] M. Daum and S. Lucks, "The Story of Alice and her Boss," Eurocrypt 2005 rump session, 2005, `http://th.informatik.uni-mannheim.de/people/lucks/HashCollisions/`.

[30] H. Dobbertin, "Cryptanalysis of MD4," *Journal of Cryptology*, v 11, n. 4, 1998, pp. 253–271.

[31] Y. Dodis, R. Gennaro, J. Håstad, H. Krawczyk, and T. Rabin, "Randomness Extraction and Key Derivation Using the CBC, Cascade and HMAC Modes," *Advances in Cryptology: Crypto '04 Proceedings*, Springer-Verlag, 2004, pp 494–510.

[32] H. Feistel, "Cryptography and Computer Privacy," *Scientific American*, May 1973, pp. 15–23.

[33] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker, "The Skein Hash Function Family," Version 1.1, Nov 2008.

[34] N. Ferguson and B. Schneier, "A Cryptographic Evaluation of IPsec", Counterpane Internet Security, 1999, `http://www.schneier.com/paper-ipsec.pdf`.

[35] N. Ferguson and B. Schneier, *Practical Cryptography*, John Wiley & Sons, 2003.

[36] N. Ferguson, D. Whiting, B. Schneier, J. Kelsey, S. Lucks, and T. Kohno, "Helix: Fast Encryption and Authentication in a Single Cryptographic Primitive," *Fast Software Encryption 2003*, Springer-Verlag, 2003, pp. 330–346.

[37] M. Gebhardt, G. Illies, and W. Schindler, "A Note on the Practical Value of Single Hash Collisions for Special File Formats," *Sicherheit 2006*, pp. 333–344.

[38] B. Gladman, "SHA1, SHA2, HMAC and Key Derivation in C," `http://fp.gladman.plus.com/cryptography_technology/sha/index.htm`, accessed 27 Jun 2008.

[39] B. Gladman, personal communication, Aug 2008.

[40] M. Gorski, S. Lucks, and T. Peyrin, "Slide Attacks on a Class of Hash Functions," *Advances in Cryptology—ASIACRYPT '08 Proceedings*, Springer-Verlag, 2008, pp. 143–160.

[41] S. Gueron, "Advanced Encryption Standard (AES) Instructions Set," Intel, `http://softwarecommunity.intel.com/articles/eng/3788.htm`, accessed 25 Aug 2008.

[42] S. Halevi and H. Krawczyk, "Strengthening Digital Signatures via Randomized Hashing," *Advances in Cryptology: CRYPTO '06 Proceedings*, Springer-Verlag, 2006, pp. 41–59.

[43] P. Hawkes, M. Paddon, and G. Rose, "On Corrective Patterns for the SHA-2 Family," Cryptology ePrint Archive, Report 2004/207.

[44] A. Joux, "Multicollisions in Iterated Hash Functions: Applications to Cascaded Constructions," *Advances in Cryptology: CRYPTO '04 Proceedings*, Springer-Verlag, 2004, pp. 306–316.

[45] B. Kaliski, "PKCS #5: Password-Based Cryptography Specification Version 2.0," RFC 2898, Sep 2000.

[46] D. Kaminski, "MD5 to be Considered Harmful Someday," Dec. 2004, `http://www.doxpara.com/md5\_someday.pdf`.

[47] M. Kausche, *Master's Thesis*, Bauhaus-Universität Weimar, 2008 (in preparation).

[48] J. Kelsey and T. Kohno, "Herding Hash Functions and the Nostradamus Attack," *Advances in Cryptology: EUROCRYPT '06 Proceedings*, Springer-Verlag, 2006, pp. 183–200.

[49] J. Kelsey and B. Schneier, "Second Preimages on n-bit Hash Functions for Much Less than $2n$ Work," *Advances in Cryptology: EUROCRYPT 2005 Proceedings*, Springer-Verlag, 2005, pp. 474–490.

[50] J. Kelsey, B. Schneier, and N. Ferguson, "Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator," *Sixth Annual Workshop on Selected Areas in Cryptography*, Springer Verlag, 1999, pp. 13–33.

[51] J. Kelsey, B. Schneier, and D. Wagner, "Key-Schedule Cryptanalysis of 3-WAY, IDEA, G-DES, RC4, SAFER, and Triple-DES," *Advances in Cryptology–CRYPTO '96 Proceedings*, Springer-Verlag, 1996, pp. 237–251.

[52] J. Kelsey, B. Schneier, and D. Wagner, "Related-Key Cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA," *ICICS '97 Proceedings*, Springer-Verlag, November 1997, pp. 233–246.

[53] J. Kelsey, B. Schneier, and D. Wagner, "Protocol Interactions and the Chosen Protocol Attack," *Security Protocols, 5th International Workshop April 1997 Proceedings*, Springer-Verlag, 1998, pp. 91–104.

[54] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side Channel Cryptanalysis of Product Ciphers," *Journal of Computer Security*, v. 8, n. 2–3, 2000, pp. 141–158.

[55] G. Kim and E. Spafford, "The Design and Implementation of Tripwire: a File System Integrity Checker," *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, 1994, pp. 18–29.

[56] J. Kim, A. Biryukov, B. Preneel, and S. Lee, "On the Security of Encryption Modes of MD4, MD5 and HAVAL," Cryptology ePrint Archive, report 2005/327.

[57] V. Klima, "Finding MD5 Collisions—a Toy For a Notebook," Cryptology ePrint Archive, Report 2005/075.

[58] V. Klima, "Finding MD5 Collisions on a Notebook PC Using Multi-message Modifications," Cryptology ePrint Archive, Report 2005/102.

[59] V. Klima, "Tunnels in Hash Functions: MD5 Collisions Within a Minute," Cryptology ePrint Archive, Report 2006/105.

[60] L. Knudsen, C. Rechberger, and S. Thomsen, "Grindahl—A Family of Hash Functions," *Fast Software Encryption 2007*, Springer-Verlag, 2007, pp. 39–57.

[61] P. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," *Advances in Cryptology—CRYPTO '96 Proceedings*, Springer-Verlag, 1996, pp. 104–113.

[62] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," *Advances in Cryptology—CRYPTO '99 Proceedings*, Springer-Verlag, 1999, pp. 388–397.

[63] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for Message Authentication," RFC 2104, 1997.

[64] A. Lenstra, B. de Weger, "On the Possibility of Constructing Meaningful Hash Collisions for Public Keys," *ACISP 2005*, pp. 267–279.

[65] H. Lipmaa, S. Moriai, "Efficient Algorithms for Computing Differential Properties of Addition", *Fast Software Encryption—FSE 2001*, Springer-Verlag, pp. 336–350.

[66] M. Liskov, R. Rivest, and D. Wagner, "Tweakable Block Ciphers," *Advances in Cryptology—CRYPTO 2002 Proceedings*, Springer-Verlag, 2002, pp. 31–46.

[67] J. Lu and J. Kim, "Attacking 44 Rounds of the SHACAL-2 Block Cipher Using Related-Key Rectangle Cryptanalysis," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences 2008*, E91-A(9), pp. 2588–2596.

[68] S. Lucks, "Two-Pass Authenticated Encryption Faster Than Generic Composition," *Fast Software Encryption 2005*, Springer-Verlag, 2005, pp. 284–298.

[69] S. Lucks, "A Failure-Friendly Design Principle for Hash Functions," *Advances in Cryptology: ASIACRYPT '05 Proceedings*, Springer-Verlag, 2005, pp. 474–494.

[70] S.M. Matyas, C.H. Meyer, and J. Oseas, "Generating strong one-way functions with cryptographic algorithms," *IBM Technical Disclosure Bulletin*, Vol. 27, No. 10A, 1985, pp. 5658–5659.

[71] M. Matsui, "Linear Cryptanalysis Method for DES Cipher," *Advances in Cryptology—EUROCRYPT '93 Proceedings*, Springer-Verlag, 1993, pp. 386–397.

[72] U. Maurer, R. Renner, and C. Holenstein, "Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology," Theory of Cryptography Conference (TCC), 2004.

[73] R. Merkle, "A Digital Signature Based on Conventional Encryption Functions," *Advances in Cryptology—CRYPTO '87 Proceedings*, Springer-Verlag, 1988, pp. 369–378.

[74] R. Merkle, "A Certified Digital Signature Scheme." *Advances in Cryptology—CRYPTO '89 Proceedings*, Springer-Verlag, 1990, pp. 218–238.

[75] R. Merkle, "One way hash functions and DES," *Advances in Cryptology—CRYPTO '89 Proceedings*, Springer-Verlag, 1990, pp. 428–446.

[76] O. Mikle, "Practical Attacks on Digital Signatures Using MD5 Message Digest," Cryptology eprint archive report 2004/356, `http://eprint.iacr.org/2004/356/`.

[77] C. Mitchell, F. Piper, and P. Wild, "Digital signatures," in *Contemporary Cryptology: The Science of Information Integrity*, G.J. Simmons, Ed., IEEE Press, 1991, pp. 325–378.

[78] F. Muller, "Differential Attacks against the Helix Stream Cipher," *Fast Software Encryption 2004*, Springer-Verlag, 2004, pp. 94–108.

[79] National Bureau of Standards, NBS FIPS PUB 46, "Data Encryption Standard," U.S. Department of Commerce, Jan 1977.

[80] National Institute of Standards and Technology, "Secure Hash Standard," FIPS 180, 11 May 1993.

[81] National Institute of Standards and Technology, "Announcing the Standard for Secure Hash Standard," FIPS 180-1, 17 Apr 1995.

[82] National Institute of Standards and Technology, "Announcing the Advanced Encryption Standard," FIPS 197, 26 Nov 2001.

[83] National Institute of Standards and Technology, "Specification for the Secure Hash Standard," FIPS 180-2, 1 Aug 2002.

[84] National Institute of Standards and Technology, "Digital Signature Standard (DSS)," FIPS 186-2, 27 Jan 2000.

[85] National Institute of Standards and Technology, "The Keyed-Hash Message Authentication Code (HMAC)," FIPS 198, 6 Mar 2002.

[86] National Institute of Standards and Technology, "Announcing The Development of New Hash Algorithm(s) for the Revision of Federal Information Processing Standard (FIPS) 180-2, Secure Hash Standard," *Federal Register*, v. 72, n. 14, 23 Jan 2007, pp. 2861–2863.

[87] National Institute of Standards and Technology, "Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family," *Federal Register*, v. 72, n. 212, 2 Nov 2007, pp. 62212–62220.

[88] National Security Agency, "Skipjack and KEA Algorithm Specification," Version 2.0, May 1998.

[89] S. Paul and B. Preneel, "Solving Systems of Differential Equations of Addition," *Information Security and Privacy, 10th Australasian Conference, ACISP 2005*, Springer-Verlag, 2005, pp. 75–88.

[90] S. Paul, B. Preneel, "Near Optimal Algorithms for Solving Differential Equations of Addition With Batch Queries," *Progress in Cryptology - INDOCRYPT 2005*, Springer-Verlag, 2005, pp. 75–88.

[91] C. Percival, "Cache Missing for Fun and Profit," BSDCan 2005, 2005, `http://www.daemonology.net/papers/htt.pdf`.

[92] J. J. Quisquater and M. Girault, "2n-bit Hash-Functions Using n-bit Symmetric Block Cipher Algorithms," *Advances in Cryptology: EUROCRYPT '89 Proceedings*,Springer-Verlag, 1990, pp. 102–109.

[93] R. Rivest, "The MD4 Message Digest Algorithm," *Advances in Cryptology: CRYPTO '90 Proceedings*, Springer-Verlag, 1990, pp. 303–311.

[94] R. Rivest, "The MD5 Message Digest Algorithm," RFC 1321, 1992.

[95] R. Rivest, M. Robshaw, R. Sidney, and Y.L. Yin, "The RC6 Block Cipher," NIST AES Proposal, Jun 98.

[96] P. Rogaway, "Formalizing Human Ignorance," *VietCrypt 2006 Proceedings*, pp. 211–228.

[97] P. Rogaway, M. Bellare, and J. Black, "OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption," *ACM Transactions on Information and System Security (TISSEC)*, v. 6, n. 3, Aug 2003, pp. 365–403.

[98] S.K. Sanadhya and P. Sarkar, "Some Observations on Strengthening the SHA-2 Family," Cryptology ePrint Archive: Report 2008/272, 9 May 2008.

[99] S. Sanadhya and P. Sarkar, "New Collision attacks Against Up To 24-step SHA-2," Cryptology ePrint Archive: Report 2008/270, 22 Sep 2008.

[100] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, *The Twofish Encryption Algorithm*, John Wiley and Sons, 1999.

[101] B. Schneier and D. Whiting, "Fast Software Encryption: Designing Encryption Algorithms for Optimal Software Speed on the Intel Pentium Processor," *Fast Software Encryption, Fourth International Workshop Proceedings (January 1997)*, Springer-Verlag, 1997, pp. 242–259.

[102] G. Sevestre, private communication.

[103] M. Stevens, "Fast Collision Attack on MD5," Cryptology ePrint Archive, report 2006/104.

[104] M. Stevens, A. Lenstra, and B. de Weger, "Predicting the Winner of the 2008 US Presidential Elections using a Sony PlayStation 3," Nov 2007, `http://www.win.tue.nl/hashclash/Nostradamus/`.

[105] D. Whiting, B. Schneier, S. Lucks, and S. Muller, "Phelix: Fast Encryption and Authentication in a Single Cryptographic Primitive," ECRYPT Stream Cipher Project Report 2005/027.

[106] X. Wang, D. Feng, X. Lai, and H. Yu, "Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD," Cryptology ePrint Archive, Report 2004/199.

[107] X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu, "Cryptanalysis of the Hash Functions MD4 and RIPEMD," *Advances in Cryptology—EUROCRYPT '05 Proceedings*, Springer-Verlag, 2005, pp. 1–18.

[108] X. Wang and H. Yu, "How to Break MD5 and Other Hash Functions," *Advances in Cryptology—EUROCRYPT '05 Proceedings*, Springer-Verlag, 2005, pp. 19–35.

[109] X. Wang, Y.L. Yin, and H. Yu, "Collision Search Attacks on SHA1," research summary, 2005.

[110] H. Wu and B. Preneel, "Differential-Linear Attacks against the Stream Cipher Phelix," *Proceedings of Fast Software Encryption 2007*, Springer-Verlag, 2007, pp. 87–100.

[111] A. Yao, "Theory and Applications of Trapdoor Functions," *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science (FOCS '82)*, IEEE, 1982, pp. 80–91.