NIST SHA-3 Competition

Waterfall Hash

Algorithm Specification and Analysis

Version 1.0
15 October 2008

Revision History

Version	Date	Author	Change
1.0	15 Oct 08	Bob Hattersley	First formal release

Contents

Tables

Figures

1 Introduction

1.1 Preamble

This document forms parts 2.B.1, 2.B.2, 2.B.4, 2.B.5 and 2.B.6 of the required contents of the submission package for a candidate algorithm to the NIST cryptographic hash algorithm (SHA-3) competition. It specifies the Waterfall Hash algorithm, estimates its computational efficiency and memory requirements, provides a security analysis including expected strength, and lists advantages and disadvantages of the algorithm.

The name Waterfall is chosen as a reflection of the main part of the algorithm, which involves "streams" flowing down (one way) into "pools".

1.2 Design Concepts

The Waterfall Hash is based on a novel multi-stream approach, unrelated to the Merkle-Damgård family. It carries a larger state than a classic block-cipher approach, balanced by less initialisation and key data than many hash algorithms. The size of the state combined with a carefully chosen updating mechanism allows demonstrations of security against all security-compromising attacks considered with a large margin of confidence.

A classic block-cipher approach proceeds in a series of steps, passing an amount of information equal (or similar) to the final size of the message digest between steps. Each step involves an expensive one-way function combining a block of input data into the current digest data. Effectively the digest data is encrypted using the block of input data as a key. The security strength of the algorithm is equivalent to the strength of the cipher operation, which at any rate is limited by the size of the chaining data.

Design of a block cipher hash is a balance between performance and strength. A perceived weakness may be overcome by increasing the number of rounds within the processing of one block, for example.

The primary idea behind the Waterfall Hash is to carry a much larger state forward while message data is passed into the algorithm. In the algorithm described here, approximately 2990 bits of information are maintained throughout Update processing, in contrast to the 512 bits of information expected in any SHA-3 candidate block-cipher algorithm. At the end, in Final processing, the state is combined into the message digest.

It would be computationally prohibitive to update the whole larger state with each new message block. Instead, each 32-bit word of message data is injected at a small number (three in this implementation) of locations, which vary cyclically. The message data is diffused, step by step, throughout the state, as the injection locations advance.

Each updating step is reversible: given the before and after state of any of various subsets of the state, it is possible to determine the value of the message word. As a result, each step is relatively inexpensive, though strongly non-linear. (Reversible non-linear functions can be much more cheaply designed than a function which is intended to resist reversal.) Information is lost as slowly as possible.

On the face of it, small amounts of reversible processing in each step might seem to allow easy insertion of modified data resulting in a collision. The second principal idea behind the Waterfall Hash is that with careful design of the diffusion process, it is possible to provide guarantees about the number of updating steps required for a viable attack, and about the likelihood of an attack succeeding. For example, we can determine a hard (non-probabilistic) lower bound on the number of updating steps required from the time that two message sequences initially diverge to the time when part of the state data can be made to converge again. Bounds of this kind can be used to determine a minimum number of non-linear function applications and the degree of mixing across the state involved, which can be combined with the properties of the non-linear updating function to determine work factor bounds.

Final processing, after the whole message has been read and processed, involves collapsing the large algorithm state into a message digest. This function is necessarily one-way in the sense that it is many-to-one (the ratio being close to 2^{2048} in this case), which offers most of the properties that we need. We also choose the mapping to be highly non-linear and difficult to reverse given any 2048-bit subset of its input; this improves the statistical properties of the algorithm as well as providing a security back-stop.

The algorithm can be simply extended (via configurable parameters) to use a larger state and in that case to output a larger digest if required in future. The security guarantees can be raised without significant additional effort for long messages – only memory usage and Final processing are affected.

There is a structural separation between the algorithm framework (the sequence of updates and the diffusion process) and the non-linear updating function which maps a 32-bit word reversibly to a 32-bit word. The framework therefore represents a Waterfall Family of hash algorithms in which alternative 32x32 substitution functions can be inserted.

It is hoped that even if this specific design is not selected, the ideas it contains may inspire future hash algorithm design.

1.3 Acknowledgements

I am grateful to Simon Hattersley of Michelson Diagnostics Ltd. and Jack Hitchens of Aspen Technology Ltd. for their help with performance testing, and to Linda White of Linda White Consulting for review of this document.

2 Algorithm Specification

2.1 Definitions and Conventions

In the following, the term "word" refers to a 32-bit unsigned integer value and "byte" refers to an 8-bit unsigned integer value.

The descriptive text assumes little-endian word order, so the value of the "first" byte (byte 0) of a word equals the value of the word (mod 256).

The symbol $\oplus$ is used for bitwise exclusive-or. The symbol $\wedge$is used for bitwise logical And, and the symbol $\vee$for bitwise logical Or. The symbol $\langle\langle\langle$ is used for bit-rotation towards higher powers (also known as left rotation): $81_{16} \langle\langle\langle\ 3 = 0c_{16}$, and the symbol $\rangle\rangle\rangle$ for bit-rotation towards lower powers (or right rotation): $0c_{16} \rangle\rangle\rangle\ 3 = 81_{16}$. The symbol $\langle\langle$ is used for bit-shift towards higher powers with overflow: $81_{16} \langle\langle\ 3 = 08_{16}$, and the symbol $\rangle\rangle$ for bit-shift towards lower powers: $0c_{16} \rangle\rangle\ 3 = 01_{16}$.

This description relates to the unoptimised reference version. There are obvious short-cuts that can be taken in any optimised version.

2.2 Algorithm Overview

The processing falls into three parts:

- Initialisation, performed once for each message, to prepare the algorithm state;
- Update, which proceeds as a series of steps, one step per input word plus one "timer" step after each 16 input words, updating the algorithm state;
- Final, performed once for each message, with no additional input, returning a digest.

Dynamic data structures, updated during each step of Update, consist of:

- an input buffer used to hold up to 64 input bytes known as the Input Buffer and associated count information – bits currently in buffer (Input Bit Count), number of completed blocks of 16 words (mod 2^{32}) (Input Block Count), number of times that the Input Block Count has cycled through zero (mod 2^{32}) (Input Slab Count);
- 3 arrays of words, known as Streams 1 to 3, of lengths 16, 7 and 6 respectively, which are each updated cyclically by one word in each step of Update;
- current indexes into Streams 1 to 3, known as Stream Indexes 1 to 3;
- 2 arrays of word of length 32, known as Pools 2 and 3, which are updated cyclically by one word using updated values from Streams 2 and 3 respectively in each step of Update;
- current index into both Pools, known as Pool Index.

Constant data consists of:

- an S-box array of 256 bytes, holding Rijndael S-box values.

Static data, set up only during Initialisation, consists of:

- an eXtended S-box array of 256 words, known as the X-box, used to perform a bijective non-linear word-to-word mapping in such a way as to mix the bytes of the input word into each byte of the output.

2.3 High-Level Processing Description

2.3.1 Initialisation

Initialisation consists of the following:

- Streams and Pools are initialised to zeros; the digest length is copied into one word of Stream 2.
- The X-box is set up to contain a different 1-1 mapping in each byte across all words, except the low-order byte which contains the Rijndael S-box exclusive-or the index.
- The digest length is saved.

2.3.2 Update

Update consists of:

- Message data is buffered in the Input Buffer (updating Input Bit Count) until 16 words are accumulated.
- One Update Step (see next) is performed for each of the 16 message words.
- The Input Block Count is incremented modulo 2^{32}, and if the result is zero, the Input Slab Count is incremented modulo 2^{32}.
- One Update Step is performed using Input Block Count as input.

2.3.3 Update Step

An Update Step takes one word as input, and updates the Streams and Pools.

- For each Stream, the Stream Indexes 1 to 3 are incremented cyclically, the input word is combined with the previously indexed word in the Stream and the current word in the Stream, the X-box applied, and the result stored in the current word in the Stream.
- The new values in Streams 2 and 3 are used to update Pools 2 and 3 respectively.

2.3.4 Final

Final consists of:

- If the Input Buffer is not empty, it is filled with zeros from the end of the message, and 17 Update Steps are performed as above consuming the Input Buffer and the Input Block Count.
- One Update Step is performed with the Input Bit Count as input.

- One Update Step is performed with the Input Slab Count as input.
- 16 Update Steps are performed with fixed input.
- A copy of Stream 1 is taken in a work array.
- For each half of each Pool, one Final Step (see below) is performed, mixing the copy of Stream 1 with the Pool section.
- 4 Final Steps are performed without additional input.
- The result is exclusive-ored with the original Stream 1 and with each half of each Pool.
- The requested number of words from the result are returned as the message digest.

2.3.5 Final Step

Each Final Step takes a work array of 16 words and optionally half of one of the Pool arrays (16 words) as input. It outputs a work array of 16 words (which becomes the input into the next Final Step).

- The 32 bits of each input word are divided between 4 words (8 bits each) of the output work array.
- The X-box substitution is applied to each word of the output array.
- The output array is exclusive-ored with the Pool section, if supplied.

2.4 Initialisation of the Streams

2.4.1 Specification

The Streams are initialised to all zeros, except for the first word in Stream 2, which is set to the requested digest length. Stream Indexes are arbitrarily initialised to the last word of each array.

2.4.2 Design Notes

No extra strength would be created by more complex initialisation. Note that the different bit switches used while updating each Stream ensure they diverge rapidly in any case.

The digest length is placed in Stream 2, since in a possible extension to the algorithm for use as a keyed hash, we may initialise Stream 1 to the key.

The lengths of the Stream arrays were chosen by analysis of multiple alternatives. The analysis was supported by considering second preimage resistance, and determining bounds on:

- the proportion of possible input states for which a short attack (fewer than 64 input words) could succeed;
- the work factor involved in a long attack (64 input words or more).

Further details are contained in the discussion of second preimage resistance.

The length of Stream 1 is required to be equal to the maximum digest length of 16 words. There are no special dependencies on the lengths of Streams 2 and 3. It may be that other lengths – for example 9 and 7 words respectively – could produce even better security bounds. Computational resource constraints prevented other options from being tested.

The lengths of the Streams are configurable constants STREAM1, STREAM2 and STREAM3 in the supplied code. See Section 5 for details of configurable parameters.

2.5 Initialisation of the Pools

2.5.1 Specification

The Pools are initialised to all zeros. The Pool Index is arbitrarily initialised to the last word of each array.

2.5.2 Design Notes

No extra strength would be created by a more complex initialisation.

The length of the two Pool arrays was chosen to reach required security levels and to be convenient for combining with the message digest in Final processing. A greater length (for example 48 or 64 words) would allow the security guarantees to rely less on the degree of non-linearity applied in each Update step, at small additional processing cost.

The length of the Pools is defined by a configurable constant POOLFACTOR which is the length as an integer multiple of the length of Stream 1.

2.6 Initialisation and Usage of the X-box

2.6.1 Specification of Initialisation

The extended S-box, or X-box, provides a reversible substitution function for 32-bit words mapping to 32-bit words.

The X-box array contains static data, defined using affine transforms of the Rijndael S-box. Let R_i be the Rijndael S-box element i, $i = 0, \ldots 255$, X_j be the X-box word j, $j = 0, \ldots 255$, and X_{jk} the kth byte of X_j, $k = 0, \ldots 3$. Then:

$$X_{i0} \leftarrow R_i \oplus i, \forall i. \tag{2.6.1}$$

$$X_{i1} \leftarrow \{[R_i \oplus (R_i\langle\langle(2) \oplus (R_i\langle\langle(3)] \langle\langle\langle 7\} \oplus 3b_{16}, \forall i. \tag{2.6.2}$$

$$X_{i2} \leftarrow R_i \oplus (R_i\langle\langle(5) \oplus (R_i\rangle\rangle 7) \oplus 95_{16}, \forall i. \tag{2.6.3}$$

$$X_{i3} \leftarrow R_i \oplus (R_i\langle\langle(7) \oplus (R_i\rangle\rangle 5) \oplus 6a_{16}, \forall i. \tag{2.6.4}$$

By definition, $X_i \equiv X_{i0} + 2^8.X_{i1} + 2^{16}.X_{i2} + 2^{24}.X_{i3}, \forall i.$

2.6.2 Specification of Usage

The X-box is applied to a word, returning a word, via four identical steps. In each step:

- the low order byte in the word is used as an index into the X-box;
- the indexed entry in the X-box is exclusive-ored with the word;
- the word is right-rotated by 8 bits (so that the new first byte value equals the previous second byte value).

Refer to Figure 2.6.1 below.

Writing X_i, $i = 0, 1, \ldots 255$, for the X-box entries, and W for the input word, then one X-box step *Xstep* is defined by:

$$Xstep(W) \equiv (W \oplus X_{(W \bmod 256)}) \rangle\rangle\rangle\ 8, \tag{2.6.5}$$

and the X-box mapping *Xbox* is defined by:

$$Xbox(W) \equiv Xstep(Xstep(Xstep(Xstep(W)))). \tag{2.6.6}$$

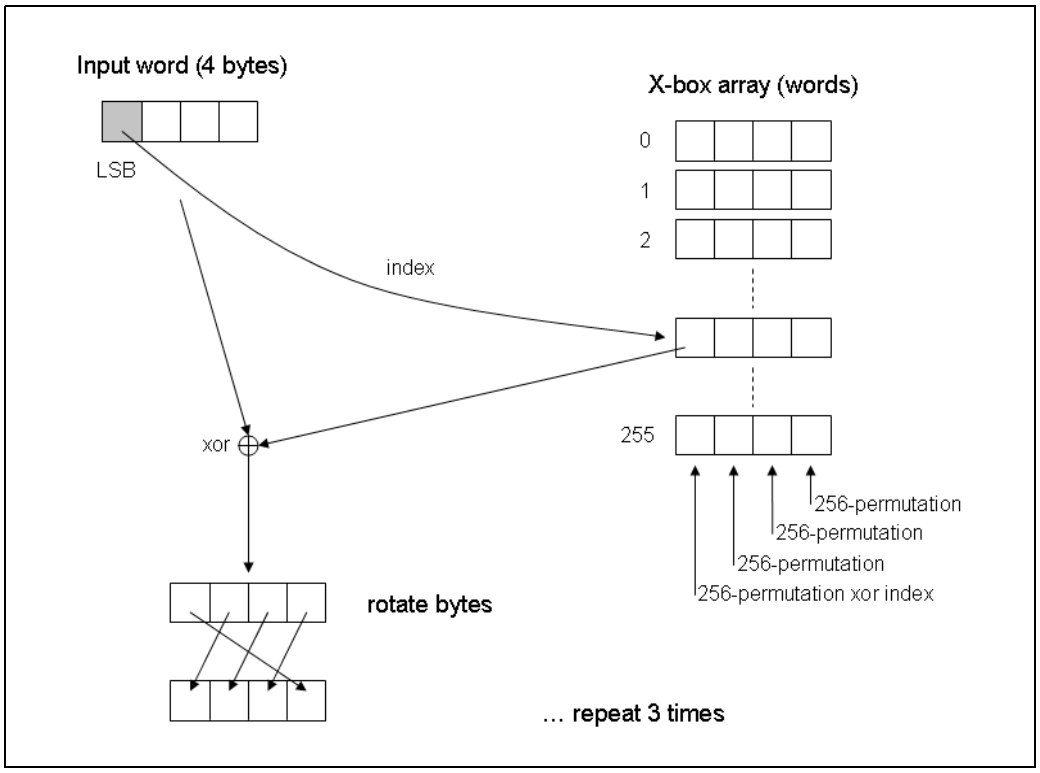

Figure 2.6.1 32x32 substitution using the X-box

2.6.3 Design Notes

The X-box data obeys the following rules:

- bytes 1 of each word form a permutation of the values 0 to 255;
- bytes 2 of each word similarly form a permutation;
- bytes 3 similarly;

- the zeroth (low-order) bytes, exclusive-ored with the index into the X-box, form a permutation.

Each X-box step is reversible (1-1), because the rotation is reversible and the step inside parentheses is a reversible substitution on the first byte (the Rijndael substitution) by choice of values. Having determined the original value of the first byte, the other X-box elements can be determined and thus the original values of the remaining input bytes. Therefore the complete application of the X-box is reversible.

The operation mixes the bytes of the word thoroughly. In each step, the value in the first byte is mixed into the remaining bytes, and the four steps with bit-rotation ensure that all bytes are mixed into all bytes.

The operation is non-linear because each step involves a non-linear substitution on the first byte. The degree of non-linearity is enhanced by the repeated application of the X-box in successive steps, so that any correlations between the Streams and between Stream and input data at different steps are quickly lost.

The operation is fast for the degree of mixing and non-linearity achieved, since it uses simple whole-word operations, and no 32-bit arithmetic.

The linear elements of the affine mappings used to define bytes 1 to 3 were chosen as follows. There are 30 invertible 8x8 linear functions L of the form (exclusive-or with a rotated copy and a shifted copy), that is:

$$L(r) \equiv r \oplus (r\langle\langle\langle k_1) \oplus (r\langle\langle k_2 \tag{2.6.7}$$

for rotations $1 \le k_1 \le 7$, and left or right shifts $-7 \le k_2 \le 7$, $k_2 \notin \{0, k_1, k_1-8\}$ (to exclude degenerate cases). Each ordered combination of three functions from this collection with mutually different shift and rotation amounts was tested. The test eliminated all cases in which changing a single bit in byte 0 could produce a zero difference in both bytes 2 and 3. A count was made of the number of times that a single bit change in byte 0 produced a zero difference in byte 3 after two sub-steps. The combination with the smallest count was chosen.

In subsequent testing it was found that the additional rotation in the definition of byte 1 could further improve maximum difference propagation, and the chosen rotation amount was determined by a randomised trial over 2^{27} inputs.

The constant terms in bytes 2 and 3 were chosen arbitrarily. The constant in byte 1 was chosen to avoid fixed points in the X-box function, and cycles of length of three or fewer, with a preference for a large minimum cycle length.

The choices were confirmed by testing for Guaranteed Avalanche Criterion, Strict Avalanche Criterion, Bit Independence Criterion, Difference Propagation and Linear Correlation. Further details are given in Section 4.3.

Note that the security of the whole hash algorithm does not depend critically on the qualities of this substitution step applied only a few times. The security proofs rely on the number of repeated applications of the X-box which must be made in the course of any viable attack.

If this function is found to have security weaknesses, then any stronger reversible 32-bit to 32-bit mapping could be used instead. On the other hand, if the function is considered excessively strong, a more efficiently calculated function could be substituted.

For optimised performance, the X-box will be defined in data.

2.7 Input Data Buffering

2.7.1 Specification

Processing is required to manage the arrival of bytes, and to zero the low-order bits of a partial byte in the last input of the message. Counters must be maintained.

When the Input Buffer is full, 16 words of input data (converted in little-endian form) are passed into the Update Step. Then the Input Block Count is used as an input in an additional Update Step.

2.7.2 Design Notes

The management of the Input Buffer has no particular security implications.

The use of the Input Block Count as a regular input prevents length-extension attacks, including those that combine a lengthened section with a shortened section (or more general combinations) to achieve a modified message length equal to the original message length.

No other message padding (with constant data) or block extension (with processed message data) is used. Block extension in particular creates the possibility of a collision in the short term. The Update phase of this algorithm has a provable minimum span of input over which a collision can be engineered, and the proof relies on the use of reversible operations.

Message padding could increase the security of Update, but no enhancement appears to be necessary. It would reduce performance.

In the optimised versions, data is processed directly from the input argument rather than passed via the buffer where possible.

2.8 Updating the Streams

2.8.1 Specification

In each Update Step, given one input word, for each Stream:

- the Stream Indexes are advanced cyclically;
- the contents of the currently indexed word are updated by combining the current value of the word, the current value of the previously indexed entry in the Stream, and the input word with exclusive-or;

- for Stream 2 the 2^0 bit is switched; for Stream 3 the 2^1 bit is switched;
- the result is passed through the X-box substitution;
- the result is stored in the currently indexed word.

Refer to Figure 2.8.1 below (which also includes the Pool updates).

Let the stream values be S_{si} for Streams $s = 1, 2, 3$ and entries $i = 0, 1, \ldots n_s-1$, let the initial Stream Index be $\hat{\imath}_s$, the message word M and the Stream-dependent mask b_s (= 0, 1 or 2), then for each s:

$$i \leftarrow (\hat{\imath}_s+1) \bmod n_s, \tag{2.8.1}$$

$$S_{si} \leftarrow Xbox(S_{s\hat{\imath}s} \oplus S_{si} \oplus M \oplus b_s), \tag{2.8.2}$$

$$\hat{\imath}_s \leftarrow i. \tag{2.8.3}$$

The X-box substitution operation *Xbox* is described above.

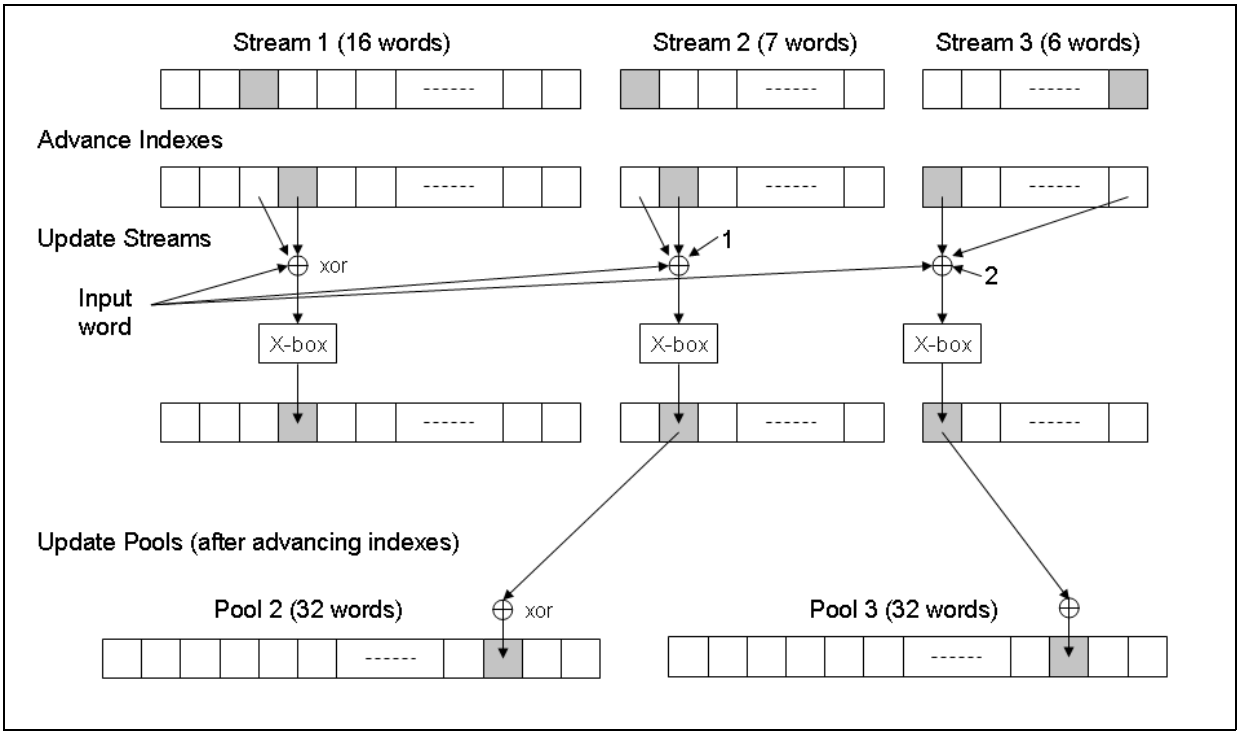

Figure 2.8.2 Update Step with one word of input

2.8.2 Design Notes

The design aim is to ensure that any change to an input word necessarily affects all three Streams, and given the successive non-linear updates of the Streams, in an uncorrelated way. Any message change is fed forward through the Streams, until further changes in input data reverse the change. (Note that the previous value $S_{s\hat{\imath}-1}$ is separated from the initial current value $S_{s\hat{\imath}}$ by (n_s-1) X-box operations, and therefore any correlation between them is small, so we expect a change to "cancel itself" in any given word, in exactly 1 out of every 2^{32} initial states.)

The different lengths of the Streams force an attacker attempting to "correct" an original message change (to create a collision) to make adjusting message changes at multiple subsequent steps. The result is that it is provably hard to reverse all the propagated changes in the Streams; and at the same time Pool data is inevitably changed. Details of the proofs are contained in Section 4.2.

The combination of values using exclusive-or is cheap, but satisfies the principal requirement of being reversible given a known Stream state.

The different bit-switches further reduce any correlation between the Streams.

The number of operations is quite small for each input word.

The updating of each Stream is independent, and so a multi-threaded implementation can use separate processor cores for maximum performance.

2.9 Updating the Pools

2.9.1 Specification

In each Update Step:

- the Pool Index is advanced cyclically;
- the current entry in Pool 2 is updated by exclusive-or with the updated Stream 2 entry;
- the current entry in Pool 3 is updated by exclusive-or with the updated Stream 3 entry.

Refer to Figure 2.8.1 above (which also includes the Stream updates).

Let the Pool values be P_{sj} for Pools $s = 2, 3$ and entries $j = 0, 1, \ldots 31$, let the current Pool Index be $\hat{\jmath}$, and the Stream entries be as defined above, then for $s = 2, 3$:

$$\hat{\jmath} \leftarrow (\hat{\jmath}+1) \bmod 32, \tag{2.9.1}$$

$$P_{s\hat{\jmath}} \leftarrow P_{s\hat{\jmath}} \oplus S_{s\hat{\imath}s}. \tag{2.9.2}$$

2.9.2 Design Notes

The purpose of the Pools is to provide a reservoir of variability, extending the algorithm state to 93 words in the Streams and Pools together (plus indexes and block counts).

A more complex update into each Pool would not contribute to the security proofs, since updates to a single entry occur 32 steps apart and there is minuscule correlation between the values. Essentially the security proofs only require that there exists exactly one updating 32-bit value (and thus message word) out of 2^{32} possible values which will make a single entry equal to a required value.

The two Pools are updated independently, so in a multi-threaded implementation, the update of Stream 2 and Pool 2 can be performed in one thread, and Stream 3 and Pool 3 in another.

2.10 Final Message Completion

2.10.1 Specification

Final Message Completion is the first part of processing in Final.

The Input Buffer is padded with zeros (including the low-order bits of the last input byte as necessary). The padded Input Buffer and the Input Block Count are fed into 17 Update Steps in the usual way.

One Update Step is performed with the Input Bit Count as input. One Update Step is performed with the Input Slab Count as input. 16 additional Update Steps are carried out with fixed input values 0, 1, … 15.

The number of additional Update Steps is a configurable constant FINALUPDATES in the supplied code.

2.10.2 Design Notes

Padding the Input Buffer and passing in the Input Bit Count ensure that a message ending in zero bits does not have the same message digest as the same message with some of the zero bits truncated.

Passing in the Input Slab Count means that message lengths of up to 2^{73} bits can be distinguished. If even longer messages were likely, it would be straightforward and cheap to add an additional counter extending the distinguished message length to 2^{105}. In any case no limit is implied on the length of the message itself.

The additional Update Steps with fixed input ensure that the final values of the message and the length counts are mixed with all entries in Stream 1 and half the Pool entries.

2.11 Final Steps

2.11.1 Specification

Following Final Message Completion, Stream 1 is copied into a work array, and four Final Steps are called, once for each half of each Pool, in the order: Pool 2 first half, Pool 2 second half, Pool 3 first half, Pool 3 second half. This is followed by four Final Steps without additional input.

Two working arrays are used; their roles (input and output) are switched in each successive Final Step.

The number of additional Final Steps is a configurable constant FINALSTEPS in the supplied code.

Each Final Step takes a work array of 16 words (the current mangled copy of Stream 1) and an optional 16-word section from one of the Pools as input, and outputs a work array of 16 words, as follows:

- The input array is transferred into the output array with the 32 bits from each word divided between 4 output words (8 bits each, 1 bit from each nybble) at different offsets applied cyclically. The bits are divided using the masks 14141414_{16}, 28282828_{16}, 42424242_{16} and 81818181_{16}, and are copied into words offset earlier in the output array by 0, 3, 7 and 12 respectively.
- The X-box substitution is applied to each word in the output work array.
- If a Pool section has been passed in, each word in the output work array is exclusive-ored with a word from that Pool section.

Refer to Figure 2.11.1 below.

Let the input working array be A_i, $i = 0, 1, \ldots 15$, the Pool section entries be Q_i, $i = 0, 1, \ldots 15$, and the output working array be B_i, $i = 0, 1, \ldots 15$. Then each Final Step consists of:

$$B_i \leftarrow (A_i \wedge 14141414_{16}) \vee (A_{(i+3) \bmod 16} \wedge 28282828_{16}) \vee (A_{(i+7) \bmod 16} \wedge 42424242_{16}) \vee (A_{(i+12) \bmod 16} \wedge 81818181_{16}), \forall i, \quad (2.11.1)$$

$$B_i \leftarrow Xbox(B_i), \forall i, \quad (2.11.2)$$

$$B_i \leftarrow B_i \oplus Q_i, \forall i. \quad (2.11.3)$$

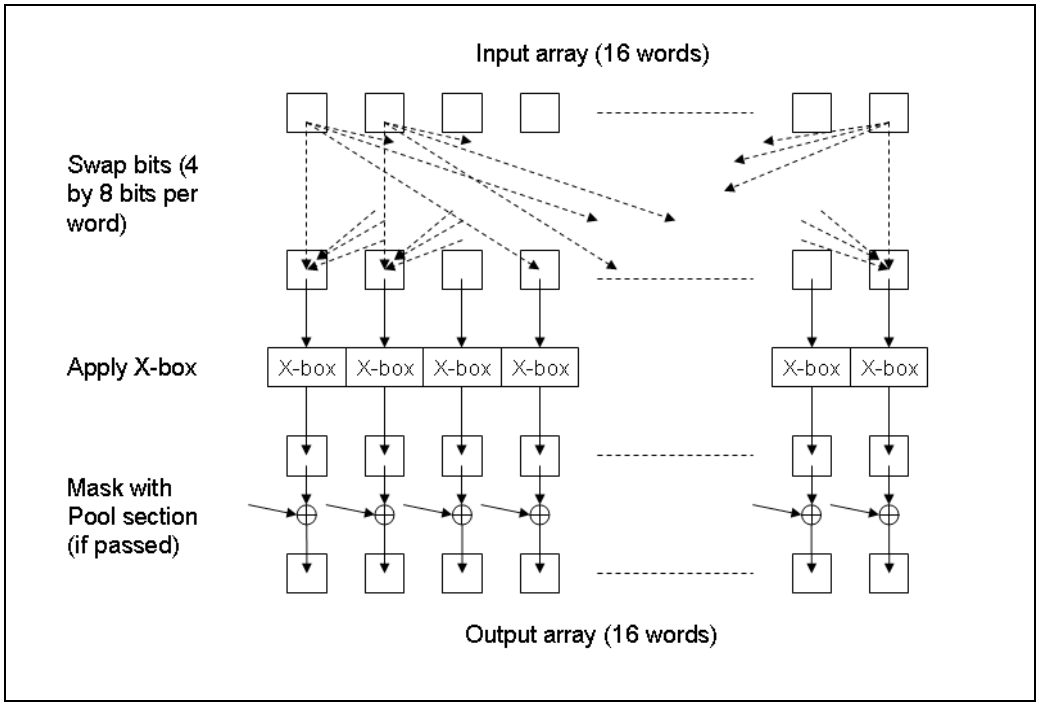

Figure 2.11.3 A Final Step

2.11.2 Design Notes

Exclusive-or with the section of Pool ensures that all the data in the Pools is used symmetrically and there are no weak Pool values.

The mixing of bits and the X-box operation are reversible, so the whole Final Step is a reversible operation on the Stream 1 values given the Pool state. Also, given the values in the input and output work arrays, the values in the section of the Pool can be determined, so it is also a reversible operation on each section of the Pool independently.

The masked bits are chosen arbitrarily but so as to split each byte into equal parts. The word offsets are chosen to rapidly spread the influence of any word across the whole array. Each bit influences every other within 3 Final Steps. Use of the same mask in each byte simplifies coding for 8-bit processors.

2.12 Final Completion

2.12.1 Specification

The output from the Final Steps is exclusive-ored, word by word, with Stream 1, and each 16-word section of each Pool.

A number of words from the result are returned as an array of bytes, low-order byte of word 0 first, according to the required message digest length.

2.12.2 Design Notes

Each Final Step is reversible on both of its 16-word inputs, and so the sequence of 8 Final Steps is reversible on each of its 5 16-word inputs (in the sense that given any 4 out of the 5 16-word inputs and the 16-word output, we can determine the remaining 16-word input), but non-linear.

Exclusive-or with the original input values makes the combined function one-way on each of those inputs. So given a Pool state, determining a Stream 1 state at the end of Final Message Completion which generates a given message digest is hard. Given a Stream 1 state and 48 words of Pool state, it is hard to determine the remaining 16 Pool words.

The exclusive-or also creates the possibility of collisions, so entropy is reduced slightly (less than 1 bit).

The algorithm does not permit reduction of the dimension of Stream 1 without other substantial changes, so we cannot take advantage of a smaller message digest size. Given the symmetry and thoroughness of the mixing in the sequence of Final Steps, we know that if the 512-bit digest has the required strength, any subsequence of the digest has a strength corresponding to its size.

3 Computational Efficiency and Memory Requirements

3.1 Performance Tests

3.1.1 Tested Software

The performance of five versions of the algorithm was tested, as shown in Table 3.1.1.

Table 3.1.1 Implementation versions

Version	Description
Reference	The reference version of the code without optimisations. Included in the supplied optical media under \Reference Implementation.
Optimised32	The version of the code optimised for 32-bit processors. Included in the optical media under \Optimized_32 bit.
Multithreaded	A version of the code based on Optimised32, but taking advantage of basic multithreading during Update. Included in the optical media under \Multithreaded.
8bit	A version based on Optimised32, but with almost all data stored and manipulated at a byte level, for estimation of performance on an 8bit processor. Included in the optical media under \8bit.
8bit_small	A version based on 8bit, but with program memory usage reduced by calculating X-box entries as needed instead of during initialisation. Included in the optical media under \8bit_small.

The version of code optimised for 64-bit processors, included in the optical media under \Optimized_64 bit, is identical to the version Optimised32, and so no tests were performed for this version. There is almost no processing carried out in 64-bit long words.

The multithreaded implementation uses the following Microsoft extensions to ANSI-C:

- `_beginthreadex`, called by the master thread to launch a slave thread;
- `WaitForMultipleObjects`, called by the master thread to wait for all slave threads to complete before exiting the Update routine;
- `_endthreadex`, called by a slave thread to signal completion to the master thread;
- `CloseHandle`, called by the master thread to release resources associated with the slave threads.

The two 8-bit versions were created in order to estimate performance and memory usage on an 8-bit platform, not as an indication of how such an implementation should be approached.

3.1.2 Test Platforms

The performance of the algorithms was tested on the platforms listed in Table 3.1.2.

Table 3.1.2 Test platforms

Platform	Description
A	Pentium M 750 1.86GHz 2GB RAM Windows XP Professional Service Pack 3
B	Core 2 Duo T7700 2.40GHz 3GB RAM Windows XP Professional Service Pack 3
C	Core 2 Quad Q9550 2.83GHz 2GB RAM Windows XP Professional Service Pack 3

Platform B matches the specification of the NIST SHA-3 Reference Platform closely enough to be regarded as identical.

In all cases, the test code was compiled and linked using Microsoft Visual C/C++ 2005 Express Edition. The following non-default compiler and linker options were used for the Reference, Optimised32 and Multithreaded versions:

Table 3.1.3 Compiler options for speed

Feature	Option
Optimization	Maximize Speed (/O2)
Inline Function Expansion	Any Suitable (/Ob2)
Favor Size or Speed	Favor Fast Code (/Ot)

The `__forceinline` compiler directive was used in the definition of `Init`, `Update` and `Final` in the header file (Waterfall.h) for the Optimised32 and Multithreaded versions.

The following non-default compiler and linker options were used for the 8bit and 8bit_small versions:

Table 3.1.4 Compiler options for small memory

Feature	Option
Optimization	Minimize Size (/O1)
Inline Function Expansion	Only __inline (/Ob1)
Favor Size or Speed	Favor Small Code (/Os)

3.1.3 Limiting Performance Test

This test used the `Update` routine to process a large memory-resident message array repeatedly to determine the limiting performance of the whole algorithm with large messages.

A 16MB array of bytes was filled with values extracted from the C library `rand` function reduced modulo 256. The `Init` routine was called, timing initiated, and then the `Update` routine was called 256 times for the whole array, and then CPU time and elapsed time measured and reported. A total of 4GB of message data was processed in each run.

Timing results used the `clock` function for elapsed time and `GetProcessTimes` for user and kernel CPU time.

The recorded times include processor task-switching times and (for elapsed times) other background processes, so are realistic, but show a slight disadvantage compared to tests run in more controlled environments or over tiny time-slices.

3.1.4 Limiting Performance Results

The following results were obtained as a minimum of 10 runs each. All times are in seconds. Cycles per byte values are calculated from user plus kernel time using the nominal clock speed of the processor. Mbytes per second values are calculated from elapsed time.

Table 3.1.5 Results for limiting performance tests

Platform	Version	User Time	Kernel Time	Elapsed Time	Cycles/ Byte	MB/s
A	Reference	108.94	0.00	108.94	47.18	37.60
A	Optimised32	50.22	0.00	50.27	21.75	81.49
A	Multithreaded	57.97	0.10	58.14	25.15	70.45
A	8bit	186.89	0.00	190.39	80.94	21.51
A	8bit_small	277.31	0.00	282.14	120.09	14.52
B	Reference	64.16	0.00	64.28	35.85	63.72
B	Optimised32	29.22	0.00	29.25	16.33	140.03
B	Multithreaded	46.13	0.02	29.95	25.78	136.75
C	Reference	57.95	0.00	57.95	38.19	70.68
C	Optimised32	26.70	0.00	26.72	17.59	153.30
C	Multithreaded	39.30	0.05	13.44	25.93	304.83

The Optimised32 version shows a satisfactory improvement over the Reference version. It is expected that assembler coding could enable some additional time savings, since only a limited number of (rather obvious) optimisations were implementable in C. It seems likely that a highly optimised assembler version of the algorithm would run as fast as a highly optimised version of SHA-1.

The poor relative performance of the Multithreaded version on platform B (with two processor cores) suggests that excessive time is taken scanning the message data three times as often.

Three slave threads are created in that version, and platform C (with 4 cores) runs close to its maximum of 75% utilisation, while achieving a speed-up of almost a factor of 2 over the single-threaded version. The addition of extra cores will not allow further speed improvements. (Note that the reported user and kernel times are the sum of the times on the separate processor cores used.)

A hardware implementation could take advantage of processing the three streams in parallel to achieve a factor of three improvement over serial processing.

The results for 8bit show approximately 4 times the number of cycles compared to Optimised32, which is in line with expectations, given that no 32-bit arithmetic operations are used in the algorithm.

3.1.5 Overhead Performance Test

This test used the `Init` and `Final` routines to determine the overhead for processing a single message. Timing was initiated, then `Init` and `Final` called 1 million times, then CPU and elapsed times measured and recorded.

Tests were not run for the Multithreaded version, which uses the same code in those two routines. The results for the Reference version were not felt to be of interest.

There is no significant difference in processing and therefore performance with different message digest lengths.

3.1.6 Overhead Performance Results

The following results were obtained as a minimum of 10 runs each. All times are in seconds. Cycles per message values are calculated from user plus kernel time using the nominal clock speed of the processor. Messages per second are calculated from elapsed time.

Table 3.1.6 Results for overhead performance tests

Platform	Version	User Time	Kernel Time	Elapsed Time	Cycles/ Message	Messages/ Second
A	Optimised32	3.09	0.00	3.09	5754	323310
A	8bit	13.44	0.00	13.66	24993	73227
A	8bit_small	18.44	0.00	18.70	34293	53467
B	Optimised32	1.84	0.00	1.84	4425	542594
C	Optimised32	1.67	0.00	1.67	4731	598444

We see that overhead processing for each message is comparable to around 270 bytes of message data, which will not be significant in most applications, though implies a factor of 4.2 performance hit for minimal length messages (16 words or fewer).

3.2 Memory Usage Tests

All the amounts of memory discussed here are insignificant for a modern 32-bit or 64-bit system. The memory estimates are really of interest only for 8-bit processors and other small platforms.

Direct observation of memory usage under Windows is complicated by the fact that memory is allocated to processes in increments of a minimum of 4096 bytes (1 memory page). This is therefore the limit of accuracy we can reach by observation.

An alternative approach is to analyse machine code listings and accumulate instruction and data bytes. This approach does not take account of padding due to alignment of routines, and it cannot take account of link-time code generation. However, it provides a better basis for estimates of memory usage on 8-bit processors.

3.2.1 Observed Memory Tests

A base version of the performance test program was created for comparison purposes (referred to as Base). This program included the same message array and message digest array, but not the hash state data. It did not call the hash routines, but performed time-wasting operations instead.

The memory usage of each program was observed using the Windows performance monitor utility PerfMon, measuring the Virtual Bytes counter of the Process performance object.

Memory usage was only observed on Platform A. It is expected that results for the NIST SHA-3 Reference Platform would be identical.

3.2.2 Observed Memory Results

All values are measured in bytes. In the Multithreaded version, the Virtual Bytes value varied over time: the maximum is given here.

Stack allocations were reduced to 4092 (one memory page minus the guard word) to avoid confusion with large multiple stack allocations for the three threads in the Multithreaded version. Note that the memory allocated to stack has no effect on performance unless it is used.

Table 3.2.7 Observed memory usage

Platform	Version	Virtual Bytes	Increment over Base
A	Base	22974464	
A	Reference	22974464	0
A	Optimised32	23003136	28672
A	Multithreaded	23203840	229376
A	8bit	22978560	4096
A	8bit_small	22978560	4096

The value of zero for the Reference version indicates that the incremental memory used by the algorithm is less than 4096 bytes, and did not require an additional memory page.

The additional memory consumed by the Optimised32 version is due to loop unrolling and in-lining of functions calls.

The increase in memory usage in the Multithreaded version is presumably due to additional run-time library code that is loaded with the executable.

3.2.3 Machine Code Listing Analysis

Machine code listings for the hash algorithm compilation units (not including the test calling program) were analysed, and total code bytes, constant data, public data bytes, and stack allocated bytes were accumulated. Note that stack bytes used during the three API routines `Init`, `Update` and `Final` are overlaid, so only the maximum (in `Final`) is reported

The bytes required for the hash state structure are added. (16 additional bytes will be required for pointers in the hash state structure in the optimised version in a 64-bit architecture.)

All values are in bytes. Code bytes and constant data are combined into total static bytes, which is the minimum number of bytes required to store the executable code. Public data, stack data and hash state are combined into total dynamic bytes, which are required in addition when the code is executed (RAM).

In the two 8-bit versions, usages of 32-bit addresses were counted, and the code bytes total reduced under the assumption that addresses would be 16 bits at most on an 8-bit platform.

Table 3.2.8 Calculated memory usage

Version	Code Bytes	Constant Data	Public Data	Stack Data	Hash State	Total Static	Total Dynamic
Reference	1405	256	1024	160	468	1661	1674
Optimised32	20574	1024	0	148	472	21598	620
Multithreaded	27770	1024	0	468	472	28794	940
8bit	1537	1024	0	37	460	2561	497
8bit_small	1785	256	0	41	460	2041	501

These values are generally consistent with the observed memory usage, except for the Multithreaded version, but are at a higher level of precision so are used in the summary.

The 8-bit results demonstrate that the algorithm can be successfully implemented in a highly constrained environment. An implementation in assembler could expect to save additional code bytes (possibly allowing the more efficient 8bit version to fit into 2kB of static memory as well).

3.3 Summary

Estimates for the two platforms specified by NIST: the NIST SHA-3 Reference Platform (NIST_ref) and an 8-bit processor are given in Table 3.3.1. For the 8-bit processor, two sets of estimates are given: for platforms with more than 2kB of program storage available (8bit), and

for those with less (8bit_small) – though it is possible that more compact assembler coding could allow the more efficient 8bit algorithm to fit into 2kB in any case.

Estimates are also given for the quad-core processor identified as Platform C using the Multithreaded implementation (Platform C/MT). The cycles per byte figure is scaled by a factor of 3 to reflect the use of three cores in parallel (as observed in the elapsed time). Dynamic bytes includes the multithreading libraries loaded with the executable in this case.

The results for a 64-bit platform are expected to be identical to the NIST_ref figures given.

Note that there is no difference in performance or memory requirements with different digest lengths.

Table 3.3.9 Efficiency estimates

Platform	Cycles/ Byte	Cycles/ Message	Static Bytes	Dynamic Bytes
NIST_ref	16.33	4425	21598	620
8bit	80.94	24993	2561	497
8bit_small	120.09	34293	2041	501
Platform C/MT	8.64	4731	28794	200582

4 Security Analysis

4.1 Overview

The focus of this analysis is resistance to second preimage finding. Other properties derive from this, or are relatively straightforward to establish.

Second preimage resistance rests on three foundations:

- the properties of the updating scheme during message processing (Update), independent of the specific substitution mechanism used, which imply bounds on the numbers of message words required in any second preimage attack;
- the non-linear and differential properties of the X-box substitution function, which ensure that, when the function is composed sufficiently often, it resists linear and differential attacks;
- the multiple composing of the substitution function and the thorough mixing in the final phase (Final).

These aspects are discussed in detail below.

Other properties are discussed following the analysis of second preimage resistance.

4.2 Second Preimage Resistance

The algorithm has second preimage strength almost equal to the digest length, independent of the length of the message. This arises from the large algorithm state, and the way it is updated, which prevents collisions over short spans. The difficulty of finding even a single collision over the large state makes multi-collision attacks impractical.

4.2.1 Attacks During the Update Phase

In this section we consider attacks that create a collision entirely within the Update phase – that is the algorithm state immediately before Final is arranged to be the same with the alternative message as with the original message.

In any attack less than $64 \cdot 2^{32} = 2^{38}$ words, the use of the Input Block Count as a regular input to Update forces the alternative message section to be the same length as the original message section it replaces. A systematic calculated attack longer than this span clearly involves work greater than the much shorter attacks we consider now. (In any case, the use of the Input Slab Count as an input in Final forces the whole alternative message length to be the same as the original (modulo 2^{70} words). So a partial attack that matches the whole algorithm state except for the Input Slab Count must be balanced by another partial attack to restore it.)

A second preimage attack during Update is defined to start when the alternative message first diverges from the original message, and is defined to end at the earliest step (a step is associated with the input of a single word) at which the algorithm state is the same as if the original message had been input.

Second preimage attacks during Update can be divided into two:

- Those longer than 64 steps (64 input words) – call these "long attacks", and
- Those shorter than 64 steps – "short attacks".

4.2.2 Long Attacks During the Update Phase

In a long attack, each word in the Stream and Pool states is affected by most of the input words, so finding a long attack is equivalent to solving 93 simultaneous non-linear equations (the size of the Stream and Pool states) in as many variables as the length of the attack. Assuming for now that the X-box substitution can be treated as a random oracle, the equations do not decompose.

If the attack has length 64, then the first 32 words of the attack appear in every constraint, and the first 16 words appear as an argument to the X-box function compounded a minimum of 16 times in every constraint. Depending on the non-linearity of the X-box, this will make the equations hard to solve. But more than this we know that the chance of a second solution existing is tiny. (Second solution, since the first solution is just the original message.) From the given initial state, there are at most $2^{64 \cdot 32} = 2^{2048}$ final states, and we require a match with the original final state which is one amongst $2^{93 \cdot 32} = 2^{2976}$. Assuming only a small degree of smoothness in the distribution of final states, we can see that the chance of a second solution existing is around 1 in 2^{928}. The work factor in finding such an attack is at least of that order, since the attacker must search that order of initial states to have a good chance of finding one with two solutions.

To have a reasonable chance of a second solution existing, the attack must have length approaching 93. In the case of an attack of length 93, the first 61 words of the attack appear in every constraint, and the first 16 words appear in every constraint, as arguments to the substitution function composed a minimum of 47 times. This minimum occurs for the 15^{th} input word (zero-based), in the 29^{th} position in either Pool – that entry is the first to be touched by the alternative message only twice. (To be precise, 29 is the cyclical offset from the current Pool Index when the attack starts.)

We only require the substitution to possess moderately good non-linear and differential properties for an attempt to solve for the first 16 words to be no more successful than brute force, given the composing factor of 47.

Intermediate length attacks have a work factor intermediate between the two extremes of length 64 and length 93, and therefore no better than brute force. Attacks longer than 93 have the same bound on work factor, since in an attack of length n>93, the attacker can always choose arbitrary values for the first n–93 alternate message words, then attempt to solve for the remainder.

4.2.3 Short Attacks During the Update Phase

A broad-based short attack (fewer than 64 words) faces the same problem of imbalance between the number of constraints and the number of variables, but with even fewer variables. The attacker may attempt to avoid affecting all 93 words of the Stream and Pool states by design. We consider a class of structured attacks in which the choice of alternate message words is made so as to "cancel out" the effects of earlier variation from the original message, returning the value in selected words to same value as with the original message at that step, and thus to limit the diffusion of changes. We can call these "sparse attacks".

We can use an optimisation technique – specifically a shortest path algorithm – to demonstrate that any such sparse attack must involve an excess of constraints over variables of at least 17. This translates into a likelihood of the attack succeeding from any given initial state of 1 in $2^{17 \cdot 32} = 2^{544}$.

This shortest path algorithm is implemented in file TracePath.c supplied in the optical media under \Supporting_Code. The justification for this method is now given with some explanation of the implementation approach.

The Stream state at any step of the hash algorithm is represented by a single bit for each Stream word (29 bits). A bit value of 0 indicates that the word has the same value with the alternate message as with the original message at that step (converged), and a bit value of 1 indicates that the value is different (diverged). For convenience, we rotate the bits so that the latest updated position in each Stream is always in a specific position (the most significant bit). This means that the state is agnostic about where it occurs during an attack.

A given sparse attack maps onto a path in the space of Stream states represented in this way. We can derive rules about which states are reachable from which states, based on whether the current and previously updated entries in a given Stream are diverged or converged. (Recall that the new entry depends on an exclusive-or combination of each of the current entry, the previous entry and the input word, via the 1-1 substitution function.)

For example, if in Stream 1 just one of the current entry or the previous entry is converged and the other is diverged, while in each of Streams 2 and 3 both entries are converged, then there are exactly three possible outcomes of the step:

- the input word is the same as the original message word, in which case the new entry in Stream 1 is diverged, and the new entries in Streams 2 and 3 are converged;
- the input word is uniquely chosen so that the new entry in Stream 1 is converged, but then the new entries in Streams 2 and 3 must be diverged;
- any other choice of input word is made, in which case all new entries are diverged.

A full table of possible transitions is given in Table 4.2.1.

Table 4.2.10 Possible update transitions

Divergences in each Stream (current plus previous)			Input word	New word in each Stream (excl. symmetrical variants)		
0	0	0	0	0	0	0
			1	1	1	1
1	0	0	0	1	0	0
			1	1	1	1
			1	0	1	1
1	1	0	0	1	1	0
			1	1	1	1
			1	1	0	1
			1	0	0	1

Divergences in each Stream (current plus previous)			Input word	New word in each Stream (excl. symmetrical variants)		
1	1	1	0	1	1	1
			1	1	1	1
			1	1	1	0
			1	1	0	0
			1	0	0	0
2	0	0	0	1	0	0
			0	0	0	0
			1	0	1	1
2	1	0	0	1	1	0
			0	0	1	0
			1	1	1	1
			1	0	1	1
			1	1	0	1
			1	0	0	1
2	1	1	0	1	1	1
			0	0	1	1
			1	1	1	1
			1	1	1	0
			1	1	0	0
			1	0	0	0
2	2	0	0	1	1	0
			0	1	0	0
			0	0	0	0
			1	1	1	1
			1	1	0	1
			1	0	0	1
2	2	1	0	1	1	1
			0	1	0	1
			0	0	0	1
			1	1	1	1
			1	1	1	0
			1	1	0	0

Note that we include “unlikely” transitions, for example where both previous and current entries in a Stream are diverged, and the input word is the same as the original message, but the new Stream entry is converged. This will occur in exactly 1 in every 2^{32} pairs of previous and current values. (For each current value we know there is exactly one previous value out of 2^{32} possibilities that will result in convergence, since the X-box mapping is bijective.)

An attack must follow a path from the all-converged (zero) state back to the zero state, via a non-zero state, which we will call a “scheme”. By minimising the number of steps we can show that the minimum number of steps required to re-converge the Streams is 20. An example minimum length scheme is shown in Table 4.2.2 below. In each step the latest updated word is represented

by the right-most digit in each Stream. It depends on the previous updated entry (immediately above), the new current entry (the left-most entry above) and the input word above.

Table 4.2.11 Example minimum length scheme

Step	Input	Stream 1	Stream 2	Stream 3
0	1	0000000000000000	0000000	000000
1	1	0000000000000001	0000001	000001
2	1	0000000000000010	0000010	000010
3	1	0000000000000101	0000101	000101
4	1	0000000000001011	0001011	001011
5	0	0000000000010110	0010111	010111
6	0	0000000000101100	0101111	101111
7	0	0000000001011000	1011111	011111
8	0	0000000010110000	0111110	111111
9	0	0000000101100000	1111100	111111
10	0	0000001011000000	1111001	111110
11	0	0000010110000000	1110010	111101
12	0	0000101100000000	1100101	111010
13	0	0001011000000000	1001011	110101
14	0	0010110000000000	0010110	101011
15	0	0101100000000000	0101100	010110
16	1	1011000000000000	1011000	101100
17	0	0110000000000000	0110000	011000
18	1	1100000000000000	1100000	110000
19	1	1000000000000000	1000000	100000
20		0000000000000000	0000000	000000

In each step, the attacker may input the same word as the original message, or an alternative word. Each alternative word input represents the addition of a new variable – a choice between $2^{32}-1$ alternatives. Each transition may also impose one or more constraints, whenever the updated Stream word is required to be converged, and this will not automatically occur (as it will when both current and previous Stream words are converged and the input word is the same as the original message). Such a constraint reduces the solution space by a factor of exactly 2^{32} (since the X-box mapping is bijective).

So far we have not considered the Pools. Ideally, we would simultaneously track the state of each Pool word, but no computationally feasible way to do this has been found. However, we can derive (extremely) loose bounds that are sufficient to complete the proof.

Each Pool word that is affected by a diverged entry in its corresponding Stream must be affected by a diverged entry twice – otherwise it will remain diverged. It cannot be affected more than twice because in an attack of no more than 64 words the Pools are traversed at most twice. Each Pool word that is affected imposes an additional constraint, so we assume only that one constraint is added for every two diverged updates to a Pool. (This is a very weak assumption, since it does

not recognise that the diverged updates are required to affect the Pool in pairs of updates which affect the same word.) We weaken that bound still further to assume that each diverged update to a Pool adds "half" a constraint – that is, it reduces the size of the solution space by a factor of 2^{16}.

Now we are able to define a distance measure representing the number of constraints imposed on the initial state and the alternative message, minus the additional variables defined by divergent input words. Each additional constraint reduces the number of the solutions by a factor of 2^{32} and each additional variable increases the number of solutions by a factor of 2^{32}. (2^{32}–1 to be precise – we ignore the distinction.) We define the distance associated with each step as the logarithm base 2^{16} of this factor.

The distance for a step is then 2 for each additional constraint in the Streams, minus 2 for an additional variable, plus 1 for each new diverged entry in Stream 2 or 3 (which affect Pools 2 and 3 respectively). Solving the resulting shortest path problem shows that the minimum distance for any scheme is 34, which implies that any given sparse attack can succeed for at most one in every $2^{34 \cdot 16} = 2^{544}$ initial states. Identifying a suitable start point for an attack, or correspondingly which scheme to use given a start point, involves a work factor of the same order.

An example minimum distance scheme is shown in Table 4.2.3 below.

Table 4.2.12 Example minimum distance scheme

Step	Input	Stream 1	Stream 2	Stream 3	Step
0	1	0000000000000000	0000000	000000	0
1	1	0000000000000001	0000001	000001	0
2	1	0000000000000011	0000011	000011	4
3	0	0000000000000110	0000110	000110	0
4	0	0000000000001100	0001100	001100	0
5	0	0000000000011000	0011000	011000	0
6	1	0000000000110000	0110000	110000	1
7	1	0000000001100001	1100001	100000	0
8	1	0000000011000011	1000011	000001	4
9	0	0000000110000110	0000110	000010	0
10	0	0000001100001100	0001100	000100	0
11	0	0000011000011000	0011000	001000	0
12	0	0000110000110000	0110000	010000	0
13	1	0001100001100000	1100000	100000	1
14	1	0011000011000001	1000000	000001	4
15	0	0110000110000010	0000000	000010	0
16	1	1100001100000100	0000000	000100	2
17	1	1000011000001000	0000001	001001	3
18	0	0000110000010000	0000010	010011	1
19	0	0001100000100000	0000100	100111	2
20	0	0011000001000000	0001000	001110	0

Step	Input	Stream 1	Stream 2	Stream 3	Step
21	0	0110000010000000	0010000	011100	0
22	1	1100000100000000	0100000	111000	3
23	1	1000001000000000	1000001	110000	3
24	0	0000010000000000	0000010	100001	2
25	0	0000100000000000	0000100	000010	0
26	0	0001000000000000	0001000	000100	0
27	0	0010000000000000	0010000	001000	0
28	0	0100000000000000	0100000	010000	0
29	1	1000000000000000	1000000	100000	4
30		0000000000000000	0000000	000000	

Note that any attack following this scheme could not in fact succeed – since the length is less than 32, the 14 Pool entries affected in the scheme must remain diverged. It seems likely that the true minimum distance is substantially larger than the lower bound demonstrated here.

4.2.4 Attacks Completed During the Final Phase

In any attack which is not completed during the Update phase, the attacker must arrange for the state at the end of Update to be such that the fixed processing during the Final phase produces the desired digest.

Identifying such a state at the end of Update is hard. The processing during Final is strongly non-linear and thoroughly mixes all input values. The whole Final phase is made a one-way function of the original Stream 1 and Pools by exclusive-oring with them at the end.

So for any assumed values of the Pools, it is hard to find values for Stream 1 that will produce the required digest. Similarly, if the attacker assumes a value for Stream 1, and all but 16 words of the Pools, then the remaining Pool values are hard to determine.

However the proof does not rest on the strength of the Final phase. Having chosen a target for the algorithm state at the end of Update, the attacker is left with a problem which is effectively a first preimage attack during the Update phase, and thus more difficult than an attack completed during Update.

The attacker may construct a list of target states at the end of Update, each of which results in the same digest, and test multiple messages to find a collision with one of the listed states. However a collision is only likely if the product of the size of the list with the number of messages approaches the size of the state, or $2^{93 \cdot 32} = 2^{2976}$.

4.3 Properties of the X-box Substitution

4.3.1 Overview

To complete the analysis for second preimage resistance, we must show that the substitution function resists linear and differential analysis when composed 47 times. (This is the minimum

number of times that the function is applied to a value which depends on 16 input words in each of the 93 constraints that define a successful attack.)

It is difficult to test the properties of a 32x32 substitution fully. For example, to fully test resistance against linear analysis, it would be necessary to test all $2^{32}-1$ linear functions of the input against all $2^{32}-1$ linear functions of the output, and look for a bias over all 2^{32} input values. Simplified and randomised testing has been used where resources did not allow exhaustive tests.

The X-box function is a composition of 4 8x8 substitutions defined by the Rijndael S-box, with mixing in between steps defined by affine transforms of the Rijndael S-box. We refer to each of the four (identical) steps in the X-box function as an X-box step. When the X-box function is applied n times, each output byte is dependent on a minimum of $4n-3$ steps applied sequentially to values dependent on all input bytes. In 47 X-box passes, the Rijndael S-box is composed 185 times onto values depending on all input bytes, in each output byte. So we can rely to an extent on the well-known properties of that S-box. (Note that 14 successive applications of the S-box are considered to introduce sufficient non-linearity for AES with a 256-bit key.)

However we must confirm that the specific usage of the S-box in the X-box substitution does not undermine those properties. The following tests were applied to the whole X-box substitution: fixed points, Strict Avalanche Criterion (SAC), Bit Independence Criterion (BIC), Guaranteed Avalanche Criterion (GAC), linear correlation and difference propagation.

The X-box function is balanced since it is invertible.

4.3.2 Properties of the Rijndael S-box

The maximum linear correlation of the Rijndael S-box is 1/8 (the maximum bias over all linear functions of input and output together is 16 out of 256). Over 47 X-box passes this suggests that the maximum linear correlation with the original 16 words of input (over all possible subsequent input) to be of the order of $2^{-3 \cdot 185} = 2^{-555}$. This suggests that a linear analysis cannot be more effective than brute force.

The maximum difference propagation probability of the Rijndael S-box is 1/64 (the maximum number of occasions on which a difference in input produces a consistent difference on outputs is 4 out of 256). Over 47 X-box passes we expect the maximum difference propagation probability (over all possible subsequent input) to be of the order of $2^{-6 \cdot 185} = 2^{-1110}$. A differential attack does not appear viable.

4.3.3 Fixed Points

The X-box was tested for fixed points, and shown to have none. The code which demonstrates this is included in the optical media in file XboxCycle.c under \Supporting_Code.

The minimum cycle length is 11. Although this value is small, there does not appear to be any weakness associated with a short cycle, since the output from one application of the X-box is always exclusive-ored with message and other data before it is input to another application of the X-box.

4.3.4 Strict Avalanche Criterion

The Strict Avalanche Criterion (SAC) was tested by recording the number of times each bit in the output word changes with a change to each bit in byte 0, over all input words. The code to perform this test is included in the optical media in file XboxSAC.c under \Supporting_Code.

Note that the results for each input byte depend only on the number of steps applied to values dependent on the byte. After one application of the X-box, byte 0 has been an input into 4 steps, while byte 1 has been an input into only 3 steps. The SAC results for changes in input byte 1 match exactly after one additional step.

The minimum over 2^{31} input words (duplication of the input difference is avoided) was found with a change to bit 2 causing a change to bit 12 1073556720 times, which is 185104, or a fraction of 1.7×10^{-4}, below the ideal value of 2^{30}.

The maximum was found with a change to bit 7 causing a change to bit 13 1073908320 times, which is 166496 or a fraction of 1.6×10^{-4}, above the ideal value.

The tests were repeated for varying numbers of steps, and after two complete X-box passes it was found that the distribution of the number of changes for each (input bit, output bit) pair was indistinguishable from Normal with mean 2^{30} and standard deviation of $2^{14.5}$ (Chi-square p-value = 0.50). The minimum and maximum values lay within 0.37 standard deviations of the mean of the appropriate power distribution.

The X-box mapping clearly satisfies SAC.

4.3.5 Bit Independence Criterion

The Bit Independence Criterion (BIC) was tested under the assumption of good SAC properties, so the number of occasions on which each bit in byte 0 changed both bits in every pair of bits was recorded over all input words. (With perfect SAC results, this value would determine the number of occasions on which neither output bit changed, and those on which just one of them changed.) The test was run concurrently with the SAC test, and is defined in the same file XboxSAC.c under \Supporting_Code.

As for the SAC test, we only test for a single byte, since the results for other bytes will be identical after additional X-box steps.

The minimum over 2^{31} input words was found with a change in bit 2 causing bits 4 and 12 to change 536674760 times, which is 196152, or a fraction of 3.6×10^{-4}, below the ideal value of 2^{29}.

The maximum was found with a change to bit 7 causing bits 6 and 23 to change 537095632 times, which is 224720, or a fraction of 4.1×10^{-4}, above the ideal value.

After two X-box passes, the minimum and maximum values lay within 0.34 standard deviations of the mean of the appropriate power distribution.

The X-box mapping satisfies BIC.

4.3.6 Guaranteed Avalanche Criterion

The Guaranteed Avalanche Criterion (GAC) was tested by counting the number of output bits changed with any single input bit change in the first byte over all input words. The code used to perform this test is in file XboxGAC.c in folder \Supporting_Code.

The minimum number of bits changed over 8 input bit changes and 2^{31} inputs (duplicate differences were avoided) was found to be 1, attained in 24 out of 2^{34} trials, or a fraction of 1.4×10^{-9}. (A change in zero bits is not possible.)

A higher minimum number of bits changed is preferred. However, this result is a reflection of the fact that the distribution of number of bit changes very closely follows the binomial distribution expected of a random oracle. It is not clear how an attacker could make use of a low GAC measure.

4.3.7 Difference Propagation

Difference propagation was first tested between changes to byte 0 of the input word and each byte of output, over 2^{27} trials with pseudo-random input from an LFSR. The code to perform this test is included in the optical media in file XboxDiff.c under \Supporting_Code.

The maximum number of hits occurred with a change of $f4_{16}$ in byte 0 causing a change of df_{16} in byte 1 528258 times, which is 3970, or a fraction of 0.0076 above the ideal value of 2^{19}. This represents a difference propagation probability of 1/254.1.

A difference measured in only one byte of the output does not give a good indication of the difference propagation probability for a whole word – we expect differences in the other bytes to be fairly evenly distributed.

Therefore this one-byte difference was refined to a difference propagated from byte 0 to a full word, by testing over all input words with the given input difference, and checking only those that matched the given output difference. The code to perform this test is included in the optical media in file XboxDiffRef.c under \Supporting_Code.

The maximum number of hits over all inputs was 4096, occurring for each of 4 output differences.

Each of these one-byte-to-one-word differences was then extended to word-to-word differences, using the inverse of the X-box function to find all input differences which produced these high-occurrence output differences. The code to perform this test is included in the optical media in file XboxDiffExt.c under \Supporting_Code.

The maximum number of hits over all inputs was 8192, occurring in just 19 cases of the $4 \cdot 2^{32}$ cases tested. The difference propagation probability in each case is 2^{-19}. Over 47 passes, this implies a difference propagation probability bounded above by 2^{-893}.

It seems unlikely, given this evidence, that any word-to-word difference would occur as often as 2^{21} times, let alone a chain of 47 of them, allowing the possibility of a differential attack more efficient than brute force.

4.3.8 Linear Correlation

Linear correlation proved hard to test. In the first test, correlation between all linear functions of input byte 0 and output byte 0 were tested over 2^{24} pseudo-random inputs from an LFSR. The code to perform this test is included in the optical media in file XboxLin.c under \Supporting_Code.

The maximum bias was found to be 8944, corresponding to a correlation of 1.1×10^{-3}. However, when this relationship was tested over all 2^{32} possible inputs, the bias was exactly zero – the apparent bias was an artefact of the selection of input values. The same proved true of all other apparently high-bias pairs of functions tested.

Therefore a test was made for a randomly-selected set of 4096 pairs of linear functions defined across all bytes, over all 2^{32} input values. The code to perform this test is in file XboxLinFull.c.

The largest bias was found to be –301056 with input and output masks $7220c73d_{16}$ and $91270c59_{16}$, corresponding to a correlation of $1.4 \times 10^{-4} = 2^{-12.8}$. Since only 4096 out of 2^{64} pairs of linear functions were tested, we cannot argue from this result that there are no pairs with a higher correlation.

However, it was noted that of the 4096 pairs tested, only 22 pairs had a correlation higher than $2.8 \times 10^{-4} = 2^{-11.8}$. If a chain of 47 pairs of linear correlations with this correlation is found then the correlation across the chain will be no greater than $2^{-11.8 \cdot 47} = 2^{-555}$. But the chance of finding such a chain or a better one can be estimated at $(22/4096)^{47} = 2^{-354}$.

Finding a useful attack via linear correlation seems highly unlikely to be more effective than brute force.

4.4 First Preimage Resistance

First preimage resistance is a weaker criterion than second preimage resistance, for the same number of bits of strength. The algorithm has second preimage strength, and therefore first preimage strength, almost equal to the digest length, independent of message length.

4.5 Collision Resistance

4.5.1 Collisions in the Update Phase

A collision occurring in the Update phase requires 93 words (2976 bits) of the algorithm state to be matched. Any kind of full-state birthday attack will clearly take more effort than a birthday attack on the algorithm as a whole.

Length-extension and insertion attacks are prevented by the use of the Input Block Count.

The structure of the algorithm, with the three elements Stream 1, Stream 2/Pool 2 and Stream 3/Pool 3 being updated independently, suggests an attack similar to that against concatenated digests from different algorithms. That is:

- create a table of collisions (extended to multi-collisions) in one of these three elements;
- test for corresponding collisions in the second element;
- test within those matches for collisions in the third element.

Creating collisions in any one of the three elements is almost trivial. However, we know from the foregoing arguments that any short attack is very unlikely to succeed even if exhaustively pursued. Only attacks of close to 93 message words have a reasonable chance of success (even given unlimited time and computational resources). The attacker can rapidly generate collisions and multi-collisions in one element. However (with a mild assumption of independence), the number of tests required before expecting to find a collision in either of Stream 2/Pool 2 or Stream 3/Pool3 is at least approaching $2^{38.32}/2 = 2^{1215}$ divided by the number of prepared collisions.

4.5.2 Collisions in the Final Phase

A collision in the Final phase is subject to the same arguments as second preimage finding in the Final phase. Since the operation is atomic, no generic attacks better than a birthday attack exist.

The exclusive-or with the Stream 1 and Pool 2 and 3 states at the end of Final reduces the entropy by less than 1 bit.

4.6 Use as a Keyed Hash

The algorithm can be conveniently extended for use as a keyed hash or randomised hash for key length of 512 bits as follows:

- set Stream 1 to the key value during Init (zero-padded for short keys);
- exclusive-or the final digest with the key value in Final immediately before returning the digest.

Up until the exclusive-or at the end of Final, the key value is used only once, and in operations which are reversible. Therefore given the algorithm state before the final exclusive-or and the message, the key can be recovered. The intermediate processing is highly non-linear, so the process is a one-way function of the key and there are no weak keys. No information can be gathered about the key given message and digest other than by a brute force attack.

Collision resistance and second preimage resistance are unaffected by the use of an initial value in this way. These properties will be unchanged from use as an unkeyed hash.

We turn to the specific attack described in Federal Register notice [Docket no. 070911510-7512-01] section 4Aii bullet 3. Write $S = u_1(M, k)$ as the function representing the Update phase and mapping message M and key k to the Stream 1 state S immediately before the Final Steps, $P =$

$u_2(M)$ as the function mapping message M to the Pool state P at the same point, and $f(S,P)$ as the function representing the latter part of Final consisting of the Final Steps, but not including the final exclusive-or with Stream 1, Pools and key. The attacker must choose M_1, then given key k_1 and digest d where:

$$d = f(u_1(M_1, k_1), u_2(M_1)) \oplus u_1(M_1, k_1) \oplus u_2(M_1) \oplus k_1, \quad (4.6.1)$$

must find M_2 and k_2 such that:

$$f(u_1(M_2, k_2), u_2(M_2)) \oplus u_1(M_2, k_2) \oplus u_2(M_2) \oplus k_2 = d. \quad (4.6.2)$$

The only potentially useful key-independent relationship (for the attacker) between M_1 and M_2 would seem to be that they should produce the same Pool state immediately before the Final Steps within Final, that is:

$$u_2(M_1) = u_2(M_2) = P, \text{ say.} \quad (4.6.3)$$

The attacker's problem reduces to:

$$f(u_1(M_2, k_2), P) \oplus u_1(M_2, k_2) \oplus k_2 = f(u_1(M_1, k_1), P) \oplus u_1(M_1, k_1) \oplus k_1. \quad (4.6.4)$$

The attacker could choose a value of P since it is relatively easy to find a first preimage on the Pools alone.

A collision between M_1 and M_2 in the Pools alone is only possible with 38 Update Steps or more, in order for all differences in the Pools to be eliminated following the first divergence. (The shortest path to a collision in Stream 2 and 3 on their own is 12 steps, and the last diverged Pool update is 6 steps earlier, 6 steps from the start of the divergence.) However we should not rule out the (remote) possibility that total Pool convergence is only reached during the Update Steps within Final. Update Steps are performed in blocks of 17 during Update, and then a minimum of 19 steps during Final. A message of no more than 16 words will take 36 Update Steps, so the message must be at least 17 words and therefore there will be a total of at least 52 Update Steps.

No further choices relating to M_1 appear to help, since there are no weak messages, and M_1 appears only in combination with k_1. The attacker's problem becomes:

$$f(u_1(M_2, k_2), P) \oplus u_1(M_2, k_2) \oplus k_2 = d. \quad (4.6.5)$$

Each word k_2 (and M_2) appears within the first term on the left hand side within the X-box substitution function composed a minimum of $(52 - 15) + 8 = 45$ times. Given the properties of the X-box substitution, this is sufficient to prevent any solve technique for k_2 more efficient than brute force.

If the attacker does not make any special choice of M_1 and M_2, or makes a choice that does not enable the simplification above, then the problem is almost identical to finding a first preimage for the given message digest, and involves no less work.

4.7 Use as an HMAC

The algorithm may be used as the basis for an HMAC in the usual way:

$$HMAC_k(M) = hash[(k \oplus \text{opad}) \,||\, hash((k \oplus \text{ipad}) \,||\, M)], \qquad (4.7.1)$$

for key k (up to 16 words), the usual constants opad and ipad, and message M. There are no issues with the relative size of the message digest and the block length. To preserve uniqueness with short keys, the key k must be zero-padded to 16 words.

Note that the usual vulnerabilities associated with using the simpler keyed hash as a MAC do not appear to apply to the Waterfall Hash and the keyed hash construction described above. This is partly because the Final stage does not simply return the current digest value even in the non-keyed hash, and so length extension attacks do not apply, and partly because the final standard result is exclusive-ored with the key before the keyed digest is returned. It may be worth considering that construction for use as a MAC, for performance reasons.

4.8 Use as an HMAC-PRF

The algorithm may be used as the basis for an HMAC-PRF in the usual way, replacing the message in the HMAC construction with an iteration counter.

A number of tests of the statistical properties of the output of the underlying unkeyed hash algorithm have been carried out, limited by computational resources. No proof of resistance at the level of $2^{n/2}$ to a distinguishing attack is offered.

If the unkeyed hash algorithm alone performs well in statistical tests, then we can expect an HMAC to perform as well or better, given the additional non-linear and mixing steps.

Tests at the bit level (bit frequency, run of zeros/ones etc.) seem inappropriate for an algorithm that is strongly word- and byte-based, and less discriminatory over relatively small data sets. These form the majority of the tests in the NIST suite. The tests selected here are mostly simple frequency and correlation tests, but the Binary Matrix Rank Test was added from the NIST suite.

All the tests were run with a very high iteration count to give a high degree of confidence in the results.

In each case a Chi-square test was used to compare the observed frequencies with the random ideal. Counts with expected frequency less than 5 were grouped successively from either end of the distribution to make each expected frequency the smallest possible greater than or equal to 5. Extreme values were also recorded and used to provide a secondary test, based on the probability of occurrence of the recorded value or a more extreme one, over all the tests in a group.

All tests succeeded at the 5% level of confidence, except the auto-correlation test. For the auto-correlation test, the test failed using a single hash call, and was re-run using HMAC with a zero key. This test succeeded at the 1% level.

This slight weakness in the auto-correlation result suggests that it may be necessary to tune the algorithm parameters for this usage, if resistance to a distinguishing attack over more than 2^{32} output words is considered important.

We note in any case that a successful distinguishing attack may have no particular implications for security in practice. It is certainly possible to distinguish the output of an algorithm from a random oracle while making neglible gains in ability to predict future output. Moreover it is unlikely that any practical use of the system would continue to use the same key for more than 2^{32} outputs.

4.8.1 Word Frequency Test

The algorithm was called 2^{32} times with a 4-byte message taking values 0 to $2^{32}-1$ (treated as a word). Only the first word of output was considered. The distribution of the number of occasions each 4-byte value occurred was determined, and compared to the Poisson distribution with mean 1. The code used to perform this test is in file TestDigest32.c in folder \Supporting_Code in the optical media.

The Chi-square p-value was found to be 0.15, and we accept at the 5% level. The maximum frequency was 13, which we accept at 5%.

4.8.2 Word Correlation Test

The algorithm was called 2^{32} times with a 4-byte message taking values 0 to $2^{32}-1$. The first word was differenced with the second word using exclusive-or, and the frequency of each resulting 32-bit value recorded. File TestDigest32.c was extended with the addition of a constant flag PARALLEL for this test.

The Chi-square p-value was found to be 0.61, and the maximum frequency was 12, so we accept at the 5% level.

4.8.3 Byte Frequency Test

The algorithm was called 2^{16} times with a 2-byte message taking values 0 to $2^{16}-1$ (treated as a short word). The frequency of each byte value in the output was determined for each of the 64 byte positions, and compared to the Poisson distribution with mean 256 using a Chi-square test. The code used to perform this test is in file TestDigest8.c.

The Chi-square p-values for the 64 frequency distributions were sorted and the minimum p-value found to be 0.019. We accept at the 5% level, since the threshold for the minimum of 64 values is $0.00080 = 1-(1-0.05)^{1/64}$.

Minimum and maximum numbers of hits over all bytes were 197 and 314 respectively. We could reject at 5% only if the values were less than 187 or greater than 332 respectively.

4.8.4 Byte Correlation Test

The algorithm was called 2^{32} times with a 4-byte message taking values 0 to $2^{32}-1$. The correlation between neighbouring bytes in the output was tested by treating the 64 bytes of output as 62 (overlapping) clusters of three bytes taken together. The frequency distribution from each position in the array was accumulated separately and compared to the Poisson distribution with mean 256 using a Chi-square test for each position. The code used to perform this test is in file TestDigest8c.c.

The Chi-square p-values for the 62 frequency distributions were sorted, and the minimum p-value found to be 0.0099. We accept at the 5% level, since the threshold for the minimum of 62 values is 0.00083.

Any count in all 62 separate distributions less than 160 or greater than 366 would be rejected at the 5% level. No such outliers were found.

4.8.5 Auto-correlation Test

An auto-correlation sequence is defined by taking pairs of values at a fixed offset in the sequence of results and differencing with exclusive-or. The algorithm was called 2^{32} times with a 4-byte message taking values 0 to $2^{32}-1$, and the frequency of each differenced value from the first words of successive digests was recorded. TestDigest32.c was extended with the addition of a constant flag AUTOCORR for this test.

This test failed with a Chi-square statistic of 38.5 over 11 degrees of freedom, p-value = 6.3×10^{-5}.

Therefore the test was re-run using the HMAC construction with a zero key. The test succeeded at the 1% level with a Chi-square p-value of 0.027, and a maximum frequency of 12.

This rather marginal result suggests that a longer test run may be able to discriminate between the HMAC output and a random oracle. For usage as an HMAC-PRF, it may be necessary to increase the number of additional Final Steps (constant parameter FINALSTEP), for example from 4 to 8, which would increase computation time by around 40%.

4.8.6 Binary Matrix Rank Test

Two square binary matrices of size 16x16 can be extracted from each message digest, and the rank of each matrix determined. The algorithm was called 2^{28} times with a 4-byte message taking values 0 to $2^{28}-1$, and the number of matrices (in a total of 2^{29}) with each rank (0, … 16) was recorded. The recorded frequencies were compared against the distribution given in NIST SP800-22 (recalculated for 16x16) using a Chi-square test. The code to perform this test is TestDigestRank.c, which also relies on matrix.c, rank.c and several header files from the NIST statistical test suite.

The Chi-square p-value was 0.86 and we accept at the 5% level. The minimum rank was 11, for which value the expected frequency is 52.

5 Algorithm Parameters

The Waterfall Hash algorithm has several parameters which may be adjusted, for example in order to:

- increase confidence over security levels
- improve statistical measures
- increase speed of execution
- reduce memory usage
- reduce security for the purpose of analysis.

Each of the parameters described here is an explicitly declared constant in one or more of the C implementations provided in the optical media.

5.1 Stream Lengths

Stream lengths in words are defined by the positive integer constants STREAM1, STREAM2 and STREAM3 declared in the header file Waterfall.h with recommended values:

STREAM1 = 16
STREAM2 = 7
STREAM3 = 6.

The length of Stream 1 is an upper bound in words on the message digest length – so if its value is reduced, Init will return an error status to a request for a 512-bit digest.

Aside from that limitation, the Stream lengths may be chosen freely (as positive integers) within reasonable bounds. The effect on performance will be very small (until cache size becomes a limitation), and the effect on memory usage is insignificant (except perhaps for tightly-constrained 8-bit processors).

The relationship between Stream lengths and security is not a simple one. Larger values may produce worse outcomes. The key issues are the minimum possible length of any second pre-image collision attack, and the minimum "probability" of the existence of a collision within less than a given number of steps. To evaluate the effects of a different choice of values, an analysis of the type demonstrated by TracePath.c (under \Supporting_Code in the optical media) may be used.

5.2 Pool Lengths

Each Pool is constrained to be the same length, and to be a positive multiple of the length of Stream 1. This is indicated by defining constant POOLFACTOR to be the length of each Pool as a multiple of STREAM1 in Waterfall.h, with recommended value:

POOLFACTOR = 2.

Any positive integer value is valid. Note that the derived constant POOL should not be changed.

A larger value has a more substantial impact on memory use than changing Stream lengths, but still not significant for a 32- or 64-bit processor. Again processing effort is not affected until cache becomes limiting.

The security impact of a larger value is unambiguous: it makes collisions within the Update phase significantly harder, roughly in step with the number of bits of state added. Of course, the overall strength of the algorithm is limited by the digest size, so this may be of purely theoretical interest.

5.3 Update Steps in Final

The number of Update Steps called with fixed input values during the Final phase, after the last (zero-padded) block and message length indicators are processed, is controlled by the non-negative integer value FINALUPDATES, declared in Waterfall.c:

FINALUPDATES = 16.

This setting is chosen to ensure that the final data input is diffused throughout Stream 1 and half of the Pool entries.

A larger value may be used to increase the diffusion of the last few message words, or to increase the minimum number of function evaluations composed onto any message word. The performance impact is moderate, given that one Update Step is called for each message word during the Update phase.

5.4 Final Steps

The number of additional Final Steps called in the Final phase, once the Pools have been combined into Stream 1, is controlled by non-negative integer value FINALSTEPS, declared in Waterfall.c:

FINALSTEPS = 4.

A larger value may increase the difficulty of a collision attack (for example when used as a keyed hash), or improve the statistical behaviour (when used as a PRF).

The impact on performance of increasing FINALSTEPS is quite substantial where the message is short, as it is when used as an HMAC-PRF. (In processing of a moderately long message the impact is insignificant.) Each additional Final Step appears to cost approximately the same as 30 additional message bytes. (This is an estimate made without direct measurement.)

5.5 Byte Order

In the optimised and other implementations, byte ordering is defined with the constant LITTLEENDIAN declared in Waterfall.c. A non-zero value indicates that the lowest address byte in a word is the least-significant byte, and thus has the value of the word reduced modulo 256.

The interpretation of byte-based message data was chosen to follow the little-endian ordering for efficiency on Intel platforms.

5.6 Loop Unrolling

In the optimised implementations, the constant UNROLL controls the compilation of sections of code which either use a for-loop over Update Steps in a block (if UNROLL= 0) or unroll the loop into a sequence of 16 Update Steps (otherwise). On the tested platforms with the tested compiler, loop unrolling was found to improve performance slightly.

5.7 Multithreading

In the Multithreaded implementation, the constant THREADS defines whether multithreading should be used at all. (The setting of zero was only used to diagnose problems.) Given that multithreading is to be used, the constant MINBLOCKS sets a threshold for the number of blocks (16 words) of message, below which the processing is still performed sequentially.

If there are more than 16 . MINBLOCKS words of input, then three threads are created, one to update Stream 1 only, one to update Stream2 and Pool 2, and one to update Stream 3 and Pool3. They run independently over all available full blocks of message data, while the default thread waits for all to complete.

The current setting has not been validated as an optimal choice.

5.8 Intermediate Values

Intermediate values may be reported to standard output by setting the constant DEBUG, declared in Waterfall.c, to a non-zero value as follows:

DEBUG = 0 No output
DEBUG = 1 Calls to the API with reporting of arguments
DEBUG = 2 Report of hash state following principal steps (Update, following Update Steps in Final, following Final Steps in Final, and digest before truncation)
DEBUG >= 3 Detailed breakdown of the processing of each message word.

The output with higher values is cumulative on the output with lower values.

Example output is given under \KAT_MCT in the optical media for 512 and 1024 byte messages and.each digest size, in files IntermediateValues_<digest size>.txt. The routine used to produce the output, Intermediates.c, is included in the same folder.

6 Advantages and Limitations

6.1 Advantages

6.1.1 Security

The structure of the Waterfall Hash allows firm, high lower bounds to be established on the complexity of the problem that an attacker must solve, for all the security-compromising attacks considered.

Multi-collision and other types of long message attack are ruled out by the difficulty of finding even one collision during the Update phase.

Length-extension attacks are ruled out by additional processing in Final.

The statistical properties tested are good, with very long test sequences used for a high degree of discrimination.

6.1.2 Implementation

The implementation is compact and simple. There is a minimum of fixed data – no more than a few words. (We do not count the Rijndael S-box which may be easily calculated.)

The algorithm may be implemented in less than 2kB of program memory and 512 bytes of RAM.

The algorithm runs quickly on large messages compared to the SHA-2 family. A 2x speed-up is possible with three processor cores, and a hardware implementation not limited by a single memory bus could achieve a 3x speed-up.

6.1.3 Flexibility

The algorithm is highly extensible. In particular, the sizes of the Streams and Pools can be changed with little impact on performance (subject to cache sizes). Thus the digest size and security guarantees may be increased to meet future requirements, effectively without limit, as cache sizes increase.

If the X-box substitution is found lacking – too slow, insufficiently non-linear, vulnerable to side channel attacks – then it can be readily replaced by any bijective 32-bit to 32-bit mapping. The core security arguments do not depend on the specific details of the substitution step.

It is also an advantage that this substitution step can be examined and tested in isolation, to provide meaningful information about the whole algorithm, thus providing a high degree of confidence in the security claims made.

Any digest size which is a multiple of 32 bits and no more than 512 bits is permitted.

6.2 Limitations

Unlike most block cipher-based hash algorithms, which commonly use a Feistel network or an FFT-style butterfly, there would appear to be few opportunities for local parallelism, and this is likely to make hardware implementations less efficient.

The use of an array look-up for the X-box may make the algorithm vulnerable to side-channel attacks. For this reason an alternative substitution mechanism may be preferred.

The Final phase is somewhat slow relative to the Update Steps. This may be a disadvantage where many short messages are processed, or when the algorithm is used as DRGB.