MeshHash2

Submitter's Name: Björn Fay
E-Mail: mail@bfay.de

Tel.: +49 6404 661672
Mobile: +49 177 6179340
Fax: +49 721 151-465799

Postal address:
Licher Pforte 27
35423 Lich
Germany

Name(s) of auxiliary submitter(s): None
Name of the algorithm inventor/developer: Björn Fay
Name of the owner of the algorithm: Björn Fay

Signature

Contents

1 Introduction and History

This specification describes a modification of a candidate for SHA-3, named MeshHash. The first version had a flaw in it, it was possible to mount a second preimage attack [Tho08]. So MeshHash has not fulfilled the requirements for SHA-3 anymore and hence was conceded broken. Furthermore there was a bug in the reference implementation: The macro for rotation of a word computed an undefined value if it should rotate a word by 0 bit.

But since the flaw can be easily fixed, which was already implemented in a preliminary version, it seems to be a good idea to publish MeshHash2 as a patch and see if it might be useful for further research or even usage. The patch uses a feedback, which increases the memory usage, but doesn't give more security against a straight forward collision attack, which was the reason it has been dropped from the preliminary version of MeshHash.

The following specification is the patched version of MeshHash, named MashHash2. It is a very flexible but conservative design with primarily security in mind and only secondarily speed. But it achieves about the same speed as the SHA-2 family and security up to 16320 bit. It can also be used in a keyed version as PRF or PRG and hence build a stream-cipher of it.

2 Algorithm specification

2.1 Notation

This specification uses the following terms, notations and operations:

Bit is a binary digit having a value of 0 or 1.

Byte is a group of eight bits having values from 0 to 255.

Word is a group of 64 bits or 8 bytes having values from 0 to $2^{64} - 1$.

PRF is the abbreviation of pseudo random function.

PRG is the abbreviation of pseudo random generator.

0^r is a group, string or array of r bits with value 0.

h is used as an index for hexadecimal numbers. For example $\texttt{1a}_h$ is the number 26.

b is used as an index for binary numbers. For example $\texttt{0101}_b$ is the number 5.

mod, % is the modulo operator, which finds the remainder of division of one number by another. For example:

$$8 \bmod 3 = 8 \,\%\, 3 = 2.$$

$\oplus$ is bitwise XOR of words or bytes. For example:

$$\texttt{0f}_h \oplus \texttt{55}_h = \texttt{00001111}_b \oplus \texttt{01010101}_b = \texttt{01011010}_b = \texttt{5a}_h.$$

$\varobar$ is bitwise OR of words or bytes. For example:

$$\texttt{0f}_h \varobar \texttt{55}_h = \texttt{00001111}_b \varobar \texttt{01010101}_b = \texttt{01011111}_b = \texttt{5f}_h.$$

$\oslash$ is bitwise AND of words or bytes. For example:

$$\texttt{0f}_h \oslash \texttt{55}_h = \texttt{00001111}_b \oslash \texttt{01010101}_b = \texttt{00000101}_b = \texttt{05}_h.$$

$\mathsf{RotR}^i(w)$ is a rotation of a word w to the right by i mod 64 bits. For example:

$$\mathsf{RotR}^{76}(\texttt{123456789abcdef}_h) = \texttt{def123456789abc}_h.$$

$w << b$ is a shift of a word w to the left by b bits. For example:

$$\texttt{ffffffff00000000}_h << 5 = \texttt{ffffffe000000000}_h.$$

$w >> b$ is a shift of a word w to the right by b bits. For example:

$$\texttt{ffffffff00000000}_h >> 5 = \texttt{07fffffff8000000}_h.$$

$\boxplus$ is addition modulo 2^{64}, that is $x \boxplus y = (x + y) \bmod 2^{64}$. For example:

$$\texttt{8765432112345678}_h \boxplus \texttt{9876543287654321}_h = \texttt{1fdb975399999999}_h.$$

$\boxdot$ is multiplication modulo 2^{64}, that is $x \boxdot y = (x \cdot y) \bmod 2^{64}$. For example:

$$\begin{aligned}&\texttt{4000000320000001}_h \boxdot \texttt{7000000000000007}_h\\ &= \texttt{c0000015e0000007}_h \boxplus \texttt{7000000000000000}_h\\ &= \texttt{30000015e0000007}_h.\end{aligned}$$

$|$ concatenates two groups, strings, or arrays of bits, bytes, or words.

Throughout this specification, the "big-endian" convention is used when expressing words or bytes, so that the most significant byte or bit is stored in the left-most position.

2.2 Overview

The MeshHash2-Algorithm is a mixture of the Merkle-Damgård model and the sponge model [BDPA07, BDPA08]. The algorithm has an internal state, which primarily consists of some so called pipes. The number of these pipes (we call it P) depends on the length of the hash value. The algorithm works in blocks and each block consists of P rounds. In each round one message word is processed and the internal state is updated. At the end of each block there is a special final round to update the internal state including the processing of a block counter and processing a given key if applicable. Before starting the processing the key is inserted at the beginning of the message.

After the message is processed some final rounds are processed including processing the number of message bits and the number of hash bits. Thereafter the hash value is computed one byte per round similar to the processing of the message (see figure 1).

```
init internal state
data_stream = key | message | pad
while data left do
{
    normal round with next data word
    after P rounds do final block round
}
do some final rounds
while hash value is shorter than hash length do
{
    normal round with 0 as data
    compute and append next byte to hash value
    after P rounds do final block round
}
```

Figure 1: Overview in pseudocode

2.3 Details

2.3.1 Parameters

The algorithm works for messages of length up to 2^{256} bit. A key as array of bytes can be given whose length in bytes must be a multiple of 8 and smaller than $2^{15} = 32768$. But normal keys should have a length in the range up to 2048 byte (16384 bit). The output is a hash value as array of bytes whose length in bits must be a multiple of 8 and shorter than $2^{15} = 32768$, but should be shorter than or equal to 16320 bit, because there is no security gain for longer hash values. In addition MeshHash2 can be used as PRG (and create a stream-cipher) and hence a pseudo-random byte sequence of indefinite length can be produced. In this case a key should be given as seed, a message can be given as further input, and the number of pipes P has to be chosen according to security needs.

2.3.2 The internal state

The internal state consists of the following components:

number of pipes named P, which stores the number of pipes.

P pipes named `pipe[i]`, for $i = 0, \ldots, P-1$, each containing a word.

2**P feedback-words** named `feedback[i][j]`, for $i = 0, 1$ and $j = 0, \ldots, P-1$, representing the feedback of the actual block and the last block.

block round counter named `block_round_counter`, which counts the processed rounds per block.

key length named key_length, which is the length of the key in words. If no key is used then this value is 0.

key named key, which is an array of words key[i] for $i = 0, \ldots, \texttt{key\_length} - 1$ containing the key.

key counter named key_counter, which counts the uses of the key and is also an offset in the key.

bit counter named bit_counter, which counts the message bits. It consists of 4 words bit_counter[i] for $i = 0, \ldots, 3$, where the value of bit_counter is $\sum_{i=0}^{3} 2^{64i} \cdot \texttt{bit\_counter[i]}$.

block counter named block_counter, which counts the number of processed blocks. It consists of 4 words block_counter[i] for $i = 0, \ldots, 3$, where the value of block_counter is $\sum_{i=0}^{3} 2^{64i} \cdot \texttt{block\_counter[i]}$.

hash length named hashbitlen, which is the length of the hash value in bits.

The counters and length for which nothing is said about the type have to be integers which can handle numbers between 0 and $2^{15} - 1$ and can be easily converted to words (or are words). At the beginning the components of the internal state are set as follows:

- The number of pipes P is computed as the smallest integer greater than or equal to $\texttt{hashbitlen}/64 + 1$. But at least 4 and at most 256.
- Every pipe and feedback-word is set to 0.
- The value of bit_counter is set to the number of bits in the given message. The counter can also be updated while processing the message, if the message length is not known at the beginning.
- Every other counter is set to 0.
- The given key as array of bytes is transformed to an array of words, where the first of eight bytes is the most significant one and the last is the least significant one (big-endian) and is stored in key.
- The given key length in bytes is divided by 8 to give the key length in words and is stored in key_length.

2.3.3 Preprocessing

After setting the internal state and before processing the message the input data for the rounds is prepared as

$$\texttt{data\_stream} = \texttt{key} \mid \texttt{message} \mid 0^{r+64 \cdot P},$$

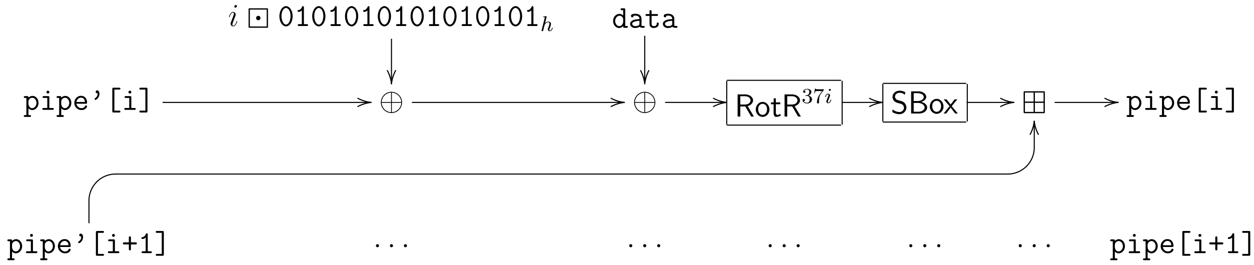

Figure 2: Normal Round

where r is the least integer so that key_length $\cdot 64 +$ bit_counter $+ r + 64 \cdot P$ is a multiple of $64 \cdot P$. Note that neither the length of the key nor "$r + 64 \cdot P$" are counted by bit_counter.

The key can already be processed directly after the initialization (this can also be seen as part of the initialization). The concatenation of $0^{r+64 \cdot P}$ and its processing can be done just before doing the final rounds, if the length of the message is not known at the beginning (this can also bee seen as part of the final rounds).

2.3.4 Normal round and SBox

A normal round processes one word (of data_stream, which consists of the key, the message, and $0^{r+64 \cdot P}$), named data, and the internal state as follows. Let pipe'[i] be pipe[i] at the beginning of the round. Then at the end of the round it is (see also figure 2)

$$\texttt{pipe[i]} = \mathsf{SBox}(\mathsf{RotR}^{37i}(\texttt{pipe'[i]} \oplus (i \boxdot 0101010101010101_h) \oplus \texttt{data})) \boxplus \texttt{pipe'[(i+1)\%P]}$$

where $\mathsf{SBox}(w)$ is computed according to the pseudocode:

```
SBox(w)
{
    w = w ⊡ 9e3779b97f4a7bb9_h
    w = w ⊞ 5e2d58d8b3bcdef7_h
    w = RotR^37(w)
    w = w ⊡ 9e3779b97f4a7bb9_h
    w = w ⊞ 5e2d58d8b3bcdef7_h
    w = RotR^37(w)
    return w
}
```

In round j (= block_round_counter) of the block the result pipe[j] is stored in a feedback-word, depending on block_counter (see also figure 3):

$$\texttt{feedback[block\_counter[0]} \owedge 0000000000000001_h\texttt{][j]} = \texttt{pipe[j]}.$$

After that increment the value of block_round_counter.

2.3.5 Final block round

After P normal rounds a final block round is done as follows.

Reset block round counter: Set `block_round_counter` to 0 for the next block.

Process block counter: The pipes are updated with `block_counter` by

$$\texttt{pipe[i]} = \mathsf{SBox}(\texttt{pipe[i]} \oplus \texttt{block\_counter[i\%4]})$$

for $i = 0, \ldots, P-1$. Increase `block_counter` by 1.

Process key: Let k be the least multiple of P that is greater than or equal to `key_length`. For $i = 0, \ldots, k-1$ do

$$\texttt{pipe[i\%P]} = \mathsf{SBox}(\texttt{pipe[i\%P]} \oplus \texttt{key[(i + key\_counter)\%key\_length]}).$$

After that set `key_counter` to $(\texttt{key\_counter} + 1) \bmod \texttt{key\_length}$ and for $i = 0, \ldots, P-1$ do

$$\texttt{pipe[i]} = \mathsf{SBox}(\texttt{pipe[i]} \oplus \texttt{key\_length} \oplus (i \boxdot 0101010101010101_h)),$$

where `key_length` is interpreted as a word.

If no key is given and `key_length` is 0, then this whole step is skipped.

Process feedback: Let $b \in \{0, 1\}$ be the least significant bit of `block_counter` (which is already updated for the next block), that is $b = \texttt{block\_counter[0]} \owedge 0000000000000001_h$. The pipes are updated with `feedback` by

$$\begin{aligned}\texttt{pipe[i]} &= \mathsf{SBox}(\texttt{pipe[i]} \oplus \texttt{feedback[b][i]})\\ \texttt{pipe[i]} &= \mathsf{SBox}(\texttt{pipe[i]} \oplus \texttt{feedback[1-b][i]})\end{aligned}$$

for $i = 0, \ldots, P-1$ (see also figure 3).

2.3.6 Final rounds

After all bits (words) of `data_stream` are processed some final rounds are done as follows:

- For $i = 0, \ldots, 3$ and $j = 0, \ldots, P-1$ do

$$\texttt{pipe[j]} = \mathsf{SBox}(\texttt{pipe[j]} \oplus \texttt{bit\_counter[i]} \oplus (j \boxdot 0101010101010101_h)).$$

- For $i = 0, \ldots, P-1$ do

$$\texttt{pipe[i]} = \mathsf{SBox}(\texttt{pipe[i]} \oplus \texttt{hashbitlen} \oplus (i \boxdot 0101010101010101_h)),$$

where `hashbitlen` is interpreted as a word.

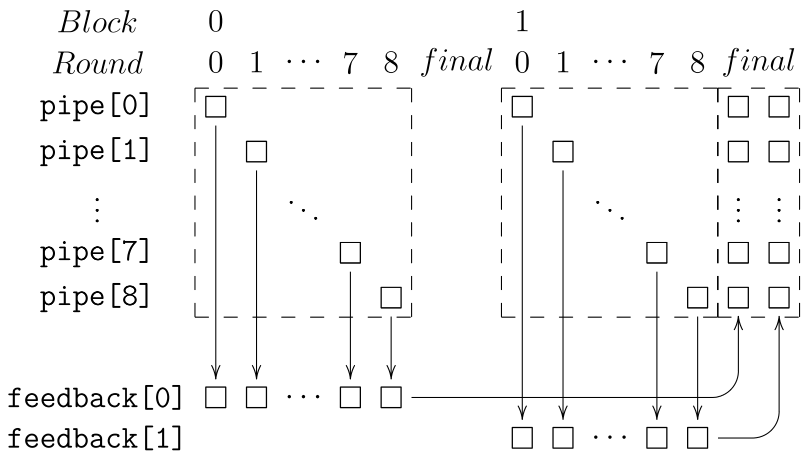

Figure 3: Feedback for $P = 9$

2.3.7 Compute the hash value by squeezing the sponge

The hash value `hashval` (as array of bytes) is computed byte by byte, similar to the processing of the message. For $i = 0, \ldots, \texttt{hashbitlen}/8 - 1$ do:

- Process a normal round with $\texttt{data} = 0$ (as in 2.3.4).
- Let k be the greatest even integer smaller than P and compute

  $$\texttt{hashval[i]} = (\texttt{pipe[0]} \oplus \texttt{pipe[2]} \oplus \ldots \oplus \texttt{pipe[k]}) \owedge \texttt{00000000000000ff}_h,$$

  where the result is interpreted as byte (there are only values between 0 and 255).
- If $i \bmod P = P - 1$ then process a final block round (as in 2.3.5).

2.3.8 Usage as PRG

If MeshHash2 is used as PRG, then `hashbitlen` is set to 0 and the number of pipes P is explicitly given. The computation of the "hash value" (as in 2.3.7) has to be repeated as many times as desired and not just `hashbitlen`/8 times.

2.3.9 The whole algorithm

MeshHash2 computes a hash value of `hashbitlen` bits for `message` with `given_key` as follows (pseudocode):

```
if hashbitlen is no multiple of 8 or >= 2^15
    then exit with error
if length of given_key in bytes is no multiple of 8 or >= 2^15
    then exit with error
init internal state (as in 2.3.2)
```

```
data_stream = key | message | 0^(r+64·P) (as in 2.3.3)
while data_stream is not completely processed do
{
    data = next word in data_stream
    normal_round(data) (as in 2.3.4)
    if block_round_counter = P
        then final_block_round (as in 2.3.5)
}
final_rounds (as in 2.3.6)
compute_hash_value (as in 2.3.7)
```

3 Design rationale

The first design criterion (not necessarily the most important) was to allow parallel execution of MeshHash2. Therefore the algorithm is working in different pipes. But to prevent multi-collision-attacks [Jou04], which would make the whole construction just a little bit stronger than one pipe, these pipes have to be connected in some way. This is why there is an addition of the next pipe at the end of a round. Because this inter-pipe-communication may be slower than in-pipe-computation, the value to add is taken from the beginning of the round (end of previous round). This should give enough time for communication. In particular if there are 4 threads each with 2 pipes, then the value of the next thread (previous round but first pipe in that thread) is not needed until the second pipe in the actual thread is computed.

Each pipe has to compute different values otherwise we would not need more than one. So in each pipe the computation is altered slightly with operations that are easy to compute. This makes implementation in hardware easier since the greatest part of the pipes is identical and may be reused (highly optimized) or the circuits may simply be copied.

Each message-bit should influence each hash-bit or here every bit in each pipe. We easily see that every data-word influence every pipe, but for the diffusion of the message-bits a sbox (SBox) is used. Since many servers use a 64-bit operating system and more and more 64-bit workstations are in use the choice was to use a 64-bit sbox with simple operations that are typical for 64-bit systems but do a fair amount of diffusion. The choice for the operations was to use a multiplication, an addition, and a rotation. The integer for multiplication is the largest prime smaller than $\frac{2^{64}}{\varphi}$, where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. That way the multiplication is invertible modulo 2^{64}. The number for addition is the largest prime smaller than $\frac{2^{64}}{e}$. The number by which the word is rotated is the largest prime smaller than $\frac{64}{\varphi}$. That way rotations can be repeated 64 times before the same word emerges and the rotation is very asymmetric. Since the highest bit would only influence itself, this procedure is done two times so that every bit can influence 37 to 64 bits, depending on its position.

The number of pipes P was chosen to be large enough to take as many bits as the hash value plus one extra pipe since the value of one pipe can be easily controlled with the message.

To avoid extension attacks or a permutation of blocks, a counter for the message bits is processed after the message and a block counter is processed after each block. The size of the counter was chosen to be large enough to nearly take the number of atoms of the universe, which makes even theoretical attacks with counter overflows meaningless. So the final block rounds for each block are different as are the final rounds for different message lengths. The length of a block in words is equal to the number of pipes, so that for an "optimal round function" there would be as many pipe states as message blocks and no one-block collision could be found (but this does probably not hold for the real world).

The usage of the feedback is needed to prevent a second preimage attack as in [Tho08]. The feedback was chosen to be from two blocks to have an interleaving of blocks. This way a change in a message-word changes the result in the pipe and hence the feedback, which is used two times, at the end of the actual block and the next block. The message-words can be easily computed to produce a certain feedback word, so it does not help against simple search for collisions (in the pipes), but it should make "Inversion of the round-function" very difficult, because of the three points of influence of one message-word (two of them exactly one block apart).

Since in theory hash functions should be keyed, it should also be done in practice. Earlier implementations have taught us that it may be weak to use the key only at the beginning of a hash function. So it is processed after each block together with its length. To influence the normal rounds in the first block the key is also processed before the first block. Since the internal state has essentially $64 \cdot P$ (or `hashbitlen` for ease) bits, the key should not be longer than that. It would only slow down the algorithm and pretend to give more security than the algorithm actually could give. Only if used as a PRG or key generator it could be adequate to use more than $64 \cdot P$ bits for a key.

The hash values of different length but for the same message should not have some easy detectable relation. So the length of the hash value is processed after the message in the final rounds. Similar things hold for the key length, but it is processed after each block since the key is processed there.

The computing of the hash value is done byte by byte as in the sponge model so that the algorithm is very flexible to use. For hash values the computation overhead is small (≤ 8 blocks), but the algorithm can also produce infinite long streams of bytes, one byte per round (P bytes per block). To use MeshHash2 as a PRG or key generator the output should not allow to make some meaningful statement about the internal state or even the key. That way no prediction of future values can be made. Hence each output byte gives nearly no information about the internal state, which is changed before the next byte is computed. This can be seen as a kind of extreme truncation or kind of hardcore function (but notice that the "round function" is no one-way function and hence there is no hardcore function, see [Gol03], but since the internal state is not outputted, this should not cause any problems). Some further research may be needed to give some more founded arguments here.

4 Special Features

There are three features that make MeshHash2 very flexible to use:

Key The algorithm is designed with key usage in mind. So there is no need for extra workarounds as in HMAC. Supported key lengths are from 0 to $64 \cdot P$ (a little bit more than `hashbitlen`) bits (with 64-bit steps). Technically even values up to $2^{15} - 1 = 32767$ bytes are possible, but there is no or just little gain in security above $64 \cdot P$ bits for use as hash function.

Hash length The possible length of the hash value vary from 8 bits to 16320 bits (in 8-bit steps). Technically even values up to $2^{15} - 8 = 32760$ bits are possible, but there is no or just little gain in security above 16320 bits.

PRG MeshHash2 might be used as a PRG (and for key derivation or for building a stream-cipher), where the key length might be even larger than $64 \cdot P$ bits to gain more entropy. The sequence of pseudo-random-bytes can be $P \cdot 2^{256}$ bytes long before a repetition of the inner state occurs and hence a repetition of the sequence before position $P \cdot 2^{256}$ is very unlikely.

5 Security

5.1 Sponge

MeshHash2 can be seen as sponge [BDPA07, BDPA08] with an adequate mapping p or padding pad, which maps any input word (or block, which ever fits best) to an input character unequal to 0. And/or the sponge model could be slightly altered. Hence the security proof of [BDPA08] or a slightly adapted version should hold for MeshHash2. As capacity we take the size of the internal state, but only the pipes since the others are just counters and feedback-words, which are not very randomly transformed. We even take only $P-1$ pipes, because one pipe can be easily controlled by selecting the message suitably. That gives us a security parameter of $64 \cdot (P-1)$ bits for usage as normal hash function (with `hashbitlen` $= 64 \cdot (P-1)$ for this security proof), which means that you have to compute about $2^{32 \cdot (P-1)}$ hash values to find a collision and $2^{64 \cdot (P-1)}$ to find a 2nd preimage. Further more we have that if an attacker computes at most $2^{16 \cdot (P-1)}$ rounds he can distinguish MeshHash2 with a maximal probability of about $2^{-32 \cdot (P-1)}$ from a random oracle. But these numbers would only hold if the "round function" would be totally random, which is surly not the case (especially not if used without a key, with key the round function would be more random but still not as random as a real random function). But on the other hand the block counter makes internal collisions only possible in the same round, so that the capacity would be 256 bits larger or the probability a bit smaller. So all things considered the numbers above are more or less only estimates, but well-founded.

5.2 Merkle-Damgård

MeshHash2 can also be seen as a Merkle-Damgård construction, at least the processing of a whole block. The only difference to MD is the computation of the hash value from the pipes round by round. But at least we can use statements about MD for the internal state.

It may be easy to find a collision of the compression function (with different internal states as input). This is not as bad as it sounds, it just means that we can not use the conclusion, that a collision resistant compression function leads to a collision resistant hash function.

One security feature is a kind of NMAC construction for MeshHash2, since before the computation of the hash value there are some special rounds and a totally different computation of the hash value than simply outputting the internal state. Also the computation of the hash value can be seen as a kind of extreme truncation (or chop solution as it is called in [CDMP05]). And if we treat the block counter as part of the message (at the end of each block), as well as the bit counter (at the end of the message, after a block with a "different encoding"), then we have a prefix-free input for MD. All three mentioned points would make MeshHash2 indifferentiable from a random oracle, if the compression function would be a random oracle as is shown in [CDMP05].

The usage of a block counter can also be seen as dithering [Riv05] and gives no fix-points. It also leads to a protection against message expansion (see [BD06])

5.3 Attacks

The following attacks have been considered:

Finding fix-points: Since there are no fix-points due to the block counter, no fix-points can be found.

Expandable messages: The block counter and bit counter make sure that every block is processed differently depending on its position and that messages with different length are processed differently at the end, so there are no expandable messages.

Second preimage: Since there are no expandable messages, the attacks [KS05, ABF$^+$08] to find 2nd preimages faster than 2^n can not be applied anymore. Also the attack of Thomsen [Tho08] can not be applied due to the feedback.

Differential cryptanalysis: Since each bit in a data word is processed in each pipe and different data words lead to different pipe states for each pipe after one round, the changing of one bit (or more) in a data word leads to a change in at least one bit per pipe, that are at least P bits but with high probability more than P bits. And one feedback-word has at least one bit changed, which is used again after the actual and the next block.

The following attacks are still possible:

Multi-Collision [Jou04]: Since there is no "wide-range-feedback", finding a collision (for the internal state) can still be "amplified" to a multi-collision. But since finding collisions should be hard, this attack remains a theoretical one. It should be mentioned that these multi-collisions are for the whole internal state and not for each pipe as considered in the design.

Herding [KK06]: Still possible for the same reason as above.

5.4 Expected strength

After the considerations in the previous sections the expected strength of MeshHash2 is as follows:

Hash function If MeshHash2 is used as hash function with a hash value of length n bits where $8 \leq n \leq 16320$ and n is a multiple of 8, then it has the following expected strength against

- finding a preimage: 2^n,
- finding a 2nd preimage: 2^n,
- finding a collision: $2^{\frac{n}{2}}$.

This includes the truncation of longer hash values to n bits. For the four required lengths the expected strength is (in bits):

hash length	preimage	2nd preimage	collision
224	224	224	112
256	256	256	128
384	384	384	192
512	512	512	256

HMAC/PRF If MeshHash2 is used as HMAC/PRF like in FIPS 198-1 the block size and hence the key size is $64 \cdot P$ bits. But since MeshHash2 has native support for keys, this could also be used. It supports keys from 64 to $64 \cdot P$ bits (in 64-bit steps). The expected strength of this native HMAC/PRF (as for the approach in FIPS 198-1) depends on the goal to achieve. For n-bit hash values and k-bit keys this is:

- If finding a collision for a given key is sufficient (or a valid HMAC for another message, which could be achieved by guessing the key) then it is the minimum of $2^{\frac{n}{2}}$ and 2^k
- If finding a preimage or 2nd preimage for a given key is sufficient (or a valid HMAC for another message, which could be achieved by guessing the key) then it is the minimum of 2^n and 2^k
- If only finding a valid HMAC for another message is sufficient (which could be achieved by guessing the key or the HMAC) then it is the minimum of 2^n and 2^k.

Hence the key should have the same length as the hash value.

Randomized hashing for digital signatures If MeshHash2 is used for randomized hashing it can be done as in SP 800-106, but also the native support of MeshHash2 can be used, where the random value is used as key. As also stated in SP 800-107 the expected strength for this is the minimum of 2^n and $2^{\frac{n}{2}} + 2^k$, where the length of the hash value is n bits and a k-bit random value is used. This is because for a valid collision you have to guess also the right random value, which changes in every signature computation without influence of an attacker.

PRG If MeshHash2 is used as PRG the key is used as seed and additional input such as personalization string or nonce can be given as message. The expected strength is 2^k, where k is the length of the seed or the key in bits. The key can be up to 16320 bits long (technically even longer, up to $2^{15} - 8 = 32760$ bytes, with unknown security gain, but the entropy of the sequence should still increase). That means that you should not be able to distinguish the generated bit sequence from a real random sequence with a workload less than 2^k and the produced sequence (if long enough) should have an entropy of k bits. The number of pipes P needed to achieve this strength is the same as for a k-bit hash value. It may be smaller, but further research is needed here.

Key derivation This can be seen as a special use case of PRG. But even if you could derive a 2048-bit key out of a 512-bit seed, the derived key would only have an entropy of 512 bits and hence should not be used as a 2048-bit key. On the other hand you can derive numerous 2048-bit keys (as concatenated byte string) out of a 2048-bit seed for use in different applications or different time slices, where the system of all these applications together has a security level of 2048 bits. That means instead of breaking one part of the system one can guess the seed and hence break all parts. On the other hand if one part is broken due to some weakness and the attacker is able to derive one key with less than a 2^k-workload, he is not able to derive the other keys.

6 Tunable security parameter for testing

There is one parameter where the security could be tuned. This is the number of pipes and can be selected by use of an extra init function. In the header file of the reference implementation there could also be changed the minimum and maximum number of pipes as well as the number of extra pipes to be added for security reasons. There could also be changed the number of words for the bit and block counter. That way the algorithm could be crippled even to one pipe, which will give a very lousy hash function.

7 Efficiency and memory requirements

7.1 Memory

The amount of memory that MeshHash2 needs can be easily derived from the design of the internal state (in bytes):

P pipes		$P \cdot 8$
$2P$ feedback-words		$2P \cdot 8$
block round counter		2
key length		2
key	`key_length` $\cdot 8$	(if unkeyed) 0
key counter		2
bit counter	$4 \cdot 8$	32
block counter	$4 \cdot 8$	32
hash length		2
one temporary pipe per thread		8
actual data word		8
total		key + $3P \cdot 8 + 88$

Depending on the implementation another 3 counters (each of 2 bytes) and 2 words could be needed (another 22 bytes). Without a key, but plus these extra 22 bytes, we get the following memory usage for the required lengths of the hash value:

hash length	memory
224 bits	230 bytes
256 bits	230 bytes
384 bits	278 bytes
512 bits	326 bytes

7.2 Efficiency

Since the efficiency depends on the system, we look at 64-bit, 32-bit and 8-bit systems in different sections.

7.2.1 64 Bit

Because the trend is to use 64-bit systems MeshHash2 is designed with these systems in mind. Probably the best way to say something about efficiency is to do some tests. Therefore a comparison of the SHA family is done with some variants of MeshHash2.

For this tests (64 bit only) a slightly optimized version is used, where assembler code is used for the rotation (can be switched in the source code). Without this optimization it would take about 9 – 20 % more time.

The testing system was:

Operating system:	Ubuntu 9.04-amd64 with kernel 2.6.28-11-generic
Processor:	Intel(R) Core(TM)2 Duo T5600 @ 1.83GHz
Memory:	2 GB
Compiler:	gcc version 4.3.3 (Ubuntu 4.3.3-5ubuntu4)
Compiler options:	-std=c99 -pedantic -O2 -march=native
Used libraries:	crypto (libssl 0.9.8g-15ubuntu3)
	mhash (libmhash2 0.9.9-1)

And the results for processing 100 MiB ($100 \cdot 2^{20}$ byte) in 1 KiB (2^{10} byte) blocks are (in ms and cycles per byte):

SHA1 (crypto)	560 ms	9.8 cpb
SHA1 (mhash)	630 ms	11.0 cpb
SHA256 (mhash)	1280 ms	22.3 cpb
SHA512 (mhash)	890 ms	15.5 cpb
MeshHash2 (160 bits)	790 ms	13.8 cpb
MeshHash2 (160 bits + key)	970 ms	16.9 cpb
MeshHash2 (224 bits)	830 ms	14.5 cpb
MeshHash2 (224 bits + key)	1020 ms	17.8 cpb
MeshHash2 (256 bits)	840 ms	14.7 cpb
MeshHash2 (256 bits + key)	1010 ms	17.6 cpb
MeshHash2 (384 bits)	950 ms	16.6 cpb
MeshHash2 (384 bits + key)	1110 ms	19.4 cpb
MeshHash2 (512 bits)	1060 ms	18.5 cpb
MeshHash2 (512 bits + key)	1230 ms	21.5 cpb
MeshHash2 (1024 bits)	1580 ms	27.6 cpb
MeshHash2 (1024 bits + key)	1730 ms	30.2 cpb
MeshHash2 (2048 bits)	2600 ms	45.4 cpb
MeshHash2 (2048 bits + key)	2790 ms	48.7 cpb
plain write	110 ms	1.9 cpb

The used keys have the length of $(P-1) \cdot 64$ bits, that is the smallest multiple of 64 that is greater than or equal to the hash length. The "plain write" is the time to write the 100 MiB in one-byte steps without processing by a hash function.

For the computation of $100000 = 10^5$ hash values of a one-block message we get the following times and hence clock cycles per hash value:

224 bits	270 ms	4941 clock cycles
224 bits + key	370 ms	6771 clock cycles
256 bits	300 ms	5490 clock cycles
256 bits + key	400 ms	7320 clock cycles
384 bits	470 ms	8601 clock cycles
384 bits + key	570 ms	10431 clock cycles
512 bits	650 ms	11895 clock cycles
512 bits + key	850 ms	15555 clock cycles

To set up the algorithm all counters, pipes, and the feedback-words have to be set to zero. Additionally the key has to be copied and processed if given. A test with $10000000 = 10^7$ calls of the init procedure has been done and the following times (total) and clock cycles (per init) were needed:

224 bits	1840 ms	337 clock cycles
224 bits + key	4180 ms	765 clock cycles
256 bits	1840 ms	337 clock cycles
256 bits + key	4180 ms	765 clock cycles
384 bits	1890 ms	346 clock cycles
384 bits + key	5740 ms	1050 clock cycles
512 bits	2040 ms	373 clock cycles
512 bits + key	7710 ms	1411 clock cycles

There are no significant tradeoffs between speed and memory.

7.2.2 32 Bit

As with the 64-bit system the same tests were done on the same machine, but this time without any assembler-optimization.

The testing system was:

Operating system:	Ubuntu 9.04-i386 with kernel 2.6.28-11-generic
Processor:	Intel(R) Core(TM)2 Duo T5600 @ 1.83GHz
Memory:	2 GB
Compiler:	gcc version 4.3.3 (Ubuntu 4.3.3-5ubuntu4)
Compiler options:	-std=c99 -pedantic -O2 -march=native
Used libraries:	crypto (libssl 0.9.8g-15ubuntu3)
	mhash (libmhash2 0.9.9-1)

And the results for processing 100 MiB ($100 \cdot 2^{20}$ byte) in 1 KiB (2^{10} byte) blocks are (in ms and cycles per byte):

SHA1 (crypto)	480 ms	8.4 cpb
SHA1 (mhash)	940 ms	16.4 cpb
SHA256 (mhash)	1830 ms	31.9 cpb
SHA512 (mhash)	4370 ms	76.3 cpb
MeshHash2 (160 bits)	2520 ms	44.0 cpb
MeshHash2 (160 bits + key)	3140 ms	54.8 cpb
MeshHash2 (224 bits)	2860 ms	49.9 cpb
MeshHash2 (224 bits + key)	3450 ms	60.2 cpb
MeshHash2 (256 bits)	2860 ms	49.9 cpb
MeshHash2 (256 bits + key)	3450 ms	60.2 cpb
MeshHash2 (384 bits)	3500 ms	61.1 cpb
MeshHash2 (384 bits + key)	4100 ms	71.6 cpb
MeshHash2 (512 bits)	4140 ms	72.3 cpb
MeshHash2 (512 bits + key)	4740 ms	82.7 cpb
MeshHash2 (1024 bits)	6990 ms	122.0 cpb
MeshHash2 (1024 bits + key)	7590 ms	132.5 cpb
MeshHash2 (2048 bits)	12530 ms	218.7 cpb
MeshHash2 (2048 bits + key)	13130 ms	229.1 cpb
plain write	50 ms	0.9 cpb

The used keys have the length of $(P-1) \cdot 64$ bits, that is the smallest multiple of 64 that is greater than or equal to the hash length. The "plain write" is the time to write the 100 MiB in one-byte steps without processing by a hash function.

For the computation of $100000 = 10^5$ hash values of a one-block message we get the following times and hence clock cycles per hash value:

224 bits	1010 ms	18483 clock cycles
224 bits + key	1330 ms	24339 clock cycles
256 bits	1090 ms	19947 clock cycles
256 bits + key	1450 ms	26535 clock cycles
384 bits	1890 ms	34587 clock cycles
384 bits + key	2210 ms	40443 clock cycles
512 bits	2720 ms	49776 clock cycles
512 bits + key	3450 ms	63135 clock cycles

To set up the algorithm all counters, pipes, and the feedback-words have to be set to zero. Additionally the key has to be copied and processed if given. A test with $10000000 = 10^7$ calls of the init procedure has been done and the following times (total) and clock cycles (per init) were needed:

224 bits	3380 ms	619 clock cycles
224 bits + key	11100 ms	2031 clock cycles
256 bits	3380 ms	619 clock cycles
256 bits + key	11560 ms	2115 clock cycles
384 bits	3440 ms	630 clock cycles
384 bits + key	17900 ms	3276 clock cycles
512 bits	4760 ms	871 clock cycles
512 bits + key	27860 ms	5098 clock cycles

There are no significant tradeoffs between speed and memory.

7.2.3 8 Bit

As there has neither been announced a reference platform nor does the author has an 8-bit system to do some tests, the estimates for 8-bit systems are only very rough.

As a raw estimate you need for all operations without the multiplication in the sbox 8-times as many steps on an 8-bit system than on a 64-bit system. For some extra logic we should perhaps blow up the factor to 16-times. But on the other side most of these operations can be done in parallel, so if you can afford to implement these steps in a parallel manner, you are not so much slower. The bottleneck is the multiplication in the sbox. Since there are many possibilities for optimization (one factor is constant) and the possibilities of 8-bit processors vary, there could only be a rough estimate. Since it is sufficient to shift the factor 26 times and add it to or subtract it from a temporary value which has to be done for a decreasing number of bytes this gives about $26 \cdot 8 \cdot 3/2 = 312$ byte-operations. The 8 is for the 8 bytes in a word, the 3 is for the addition or subtraction, another one for a carry bit and a shift and the 2 is for the decreasing number of bytes. But this also could be done in parallel and there is still much room for optimization. Perhaps if optimized and parallelized it could be done with the same speed as the other operations, namely 8 to 16 times slower than on a 64-bit system (with the same clock-speed). Since multiplication is an old and well known problem there are presumably some very tricky and efficient solutions.

The overall estimation then is somewhere between 8-times and about 300-times slower than on a 64-bit system (with the same clock-speed). For exact estimates it seems one has to implement it on different platforms.

References

[ABF+08] Elena Andreeva, Charles Bouillaguet, Pierre-Alain Fouque, Jonathan J. Hoch, John Kelsey, Adi Shamir, and Sébastien Zimmer. Second preimage attacks on dithered hash functions. In Smart [Sma08], pages 270–288.

[BD06] Eli Biham and Orr Dunkelman. A framework for iterative hash functions – haifa, 2006. second NIST hash workshop.

[BDPA07] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Sponge functions, 2007. ECRYPT Hash Workshop 2007.

[BDPA08] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In Smart [Sma08], pages 181–197.

[CDMP05] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-damgård revisited: How to construct a hash function. In Victor Shoup, editor, *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer, 2005.

[Gol03] Oded Goldreich. *Foundations of Cryptography*, volume I, Basic Tools. Cambridge University Press, reprinted and corrected edition, 2003.

[Jou04] Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In Matthew K. Franklin, editor, *CRYPTO*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.

[KK06] John Kelsey and Tadayoshi Kohno. Herding hash functions and the nostradamus attack. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2006.

[KS05] John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005.

[Riv05] Ronald L. Rivest. Abelian square-free dithering for iterated hash functions, 2005. ECRYPT Hash Function Workshop 2005.

[Sma08] Nigel P. Smart, editor. *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13–17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*. Springer, 2008.

[Tho08] Søren S. Thomsen. Second preimage attack on MeshHash, 2008. `http://www.mat.dtu.dk/people/S.Thomsen/meshhash/2ndpreimage.pdf`.