

# UMASH: a fast almost universal 64-bit string hash

Paul Khuong, [Backtrace I/O](#)

2022-02-23

UMASH is a string hash function with throughput—22 GB/s on a 2.5 GHz Xeon 8175M—and latency—9-20 ns for input sizes up to 64 bytes on the same machine—comparable to that of performance-optimised hashes like [MurmurHash3](#), [XXH3](#), or [farmhash](#). Its 64-bit output is *almost universal* (see below for collision probability), and it, as well as both its 32-bit halves, satisfies [Reini Urban’s fork of SMHasher](#) and [Yves Orton’s extended version](#).

The universal collision bounds for UMASH hold under the assumption that keys are fully re-generated with random bytes, independently of the hash function’s inputs. While UMASH accepts a 64-bit “seed” value, it is merely a hint that we would like to see different hash values, without any guarantee. Nevertheless, this suffices to pass SMHasher’s seed-based collision tests. For real guarantees, expand short keys into full UMASH keys with a stream cipher like [Salsa20](#).

UMASH should not be used for cryptographic purposes, especially not against adversaries that can exploit side-channels or adapt to the hashed outputs, but it does provide worst-case pair-wise collision bounds. For any two strings of  $s$  bytes or fewer, the probability of them hashing to the same value satisfies  $\varepsilon < \lceil s/4096 \rceil \cdot 2^{-55}$ , as long as the independently generated key is chosen uniformly at random. However, once a few collisions have been identified (e.g., through a timing side-channel in a hash table), the linearity of the hash function makes it trivial to combine them into exponentially more colliding values.

UMASH may also generate a 128-bit fingerprint by computing a secondary hash value concurrently with the usual (primary) UMASH value; the combination of the two 64-bit values guarantees a collision probability less than  $2^{-83}$  for input size up to 64 MB, and less than  $2^{-70}$  at 5 GB. The secondary hash function mostly reuses work performed in the primary one, so this additional collision resistance comes at a reasonable overhead.

## Overview of the UMASH hash function

When hashing strings of 8 or fewer bytes, UMASH converts them to 64-bit integers and passes those to a mixing routine based on [SplitMix64](#), with an additional random parameter that differs for each input size  $s \in [0, 8]$ . The result is universal, never collides values of the same size, and otherwise collides values with probability  $\varepsilon_{\text{short}} \approx 2^{-64}$ .

There’s nothing special about SplitMix64, except that it’s well known, satisfies SMHasher’s bias and avalanche tests, and is invertible. Similarly, we use an invertible but otherwise arbitrary xor-rot finaliser in the general case to satisfy SMHasher without impacting the collision bounds.

The structure of UMASH on strings of 9 or more bytes (Figure 1) closely follows that of [Dai and Krovetz’s VHASH](#), with parameters weakened to improve speed at the expense of collision rate, and the [NH block compressor](#) replaced with a (unfortunately) bespoke hybrid compressor OH.<sup>1</sup>

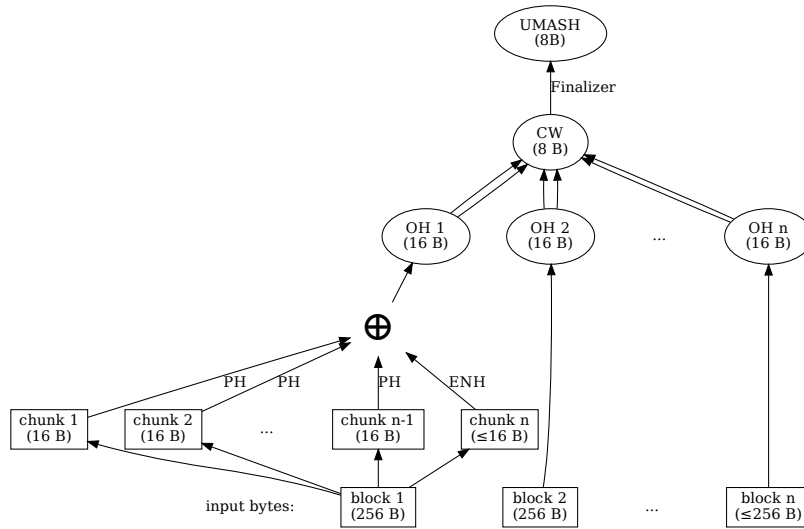


Figure 1: The two-level (OH / polynomial string hash) structure of UMASH

UMASH splits its input string into a sequence of blocks  $\mathbf{M}$ ; the first-level OH compressor further decomposes each block  $\mathbf{M}_i$  of 256 bytes into  $m$ , a sequence of 16-byte chunks. The final block may be short, in which case the blocking logic may include redundant data to ensure its size is a multiple of 16 bytes.

<sup>1</sup>OH because it’s a hybrid of PH and NH, and O is the letter between N and P.

Each block of up to 256 bytes is compressed to 16 bytes with OH. The resulting outputs are then fed to a Carter-Wegman string polynomial hash function (with a standard Horner evaluation algorithm), and the accumulator finalised with a reversible function to obtain a UMASH value.

Although the analysis below assumes a modulus of  $M_{61} = 2^{61} - 1$  for the polynomial hash, the implementation actually works in  $2^{64} - 8 = 8M_{61}$ . This does not worsen the collision probability, but obviously affects the exact hash values computed by the algorithm.

The first-level compression function is sampled from the OH family with word size  $w = 64$  bits and a block size of 256 bytes.

The OH compression function relies on PH, the equivalent of NH in 128-bit carry-less arithmetic for all but the last iteration, which instead mixes the remaining 16-byte chunk with NH.

The core PH construction has been re-derived and baptised multiple times, recently as CLNH by Lemire and Kaser; Bernstein dates it back to 1968. Compared to NH, PH offers better throughput on higher-end contemporary cores, but worse latency. That’s why OH switches to NH for the last iteration.

For a given 16-byte chunk  $m_i$ , we parameterise PH on a 16-byte key  $k_i$  as

$$\text{PH}_{k_i}(m_i) = (m_i^{hi} \oplus k_i^{hi}) \odot (m_i^{lo} \oplus k_i^{lo}),$$

where  $\cdot^{hi}$  is the high 8-byte (64-bit) half of a 16-byte value,  $\cdot^{lo}$  the low 8-byte half, and  $\odot$  is a multiplication in 128-bit carry-less arithmetic. This family of hash functions mixes pairs of  $w = 64$  bit values into  $2w$ -bit hashes that are  $(2^{-w} = 2^{-64})$ -almost-xor-universal.

This last property means that, not only are two different 16-byte chunks unlikely to collide (i.e.,  $P[\text{PH}_{k_i}(x) = \text{PH}_{k_i}(y)] \leq 2^{-64}$  for  $x \neq y$ ), but in fact any specific difference  $\Delta_{\text{xor}}$  is just as unlikely:

$$P[\text{PH}_{k_i}(x) \oplus \text{PH}_{k_i}(y) = \Delta_{\text{xor}}] \leq 2^{-64}.$$

Almost-xor-universality lets us combine multiple PH-mixed values with bitwise xor ( $\oplus$ ) while preserving  $2^{-64}$ -almost-xor-universality for the final result.

The final NH step is similar, except in mixed-width modular arithmetic:

$$\text{NH}_{k_j}(m_j) = (m_j^{hi} +_{64} k_j^{hi}) \cdot_{128} (m_j^{lo} +_{64} k_j^{lo}),$$

where  $+_{64}$  denotes addition modulo  $2^{64}$ , and  $\cdot_{128}$  a full  $64 \times 64 \rightarrow 128$ -bit multiplication.

The NH family of hash function mixes 128-bit (16-byte) values into 128-bit hashes that are  $2^{-64}$ -almost- $\Delta$ -universal: for any  $x \neq y$ , every specific difference  $\Delta$  between  $\text{NH}_k(x)$  and  $\text{NH}_k(y)$  satisfies

$$P[\text{NH}_{k_j}(x) -_{128} \text{NH}_{k_j}(y) = \Delta] \leq 2^{-64},$$

where  $-_{128}$  denotes subtraction modulo  $2^{128}$ . This  $\Delta$ -almost-universality means that we can add an arbitrary 128-bit value to the output of NH and still obtain a function from the  $2^{-64}$ -almost-universal ENH family.

Let  $m$  be a block of  $n \geq 1$  16-byte chunks, and  $t$  an arbitrary 128-bit tag; the OH hash of  $m$  and  $t$  for a given parameter vector  $k$  is

$$\text{OH}_k(m) = \left( \bigoplus_{i=1}^{n-1} \text{PH}_{k_i}(m_i) \right) \oplus (\text{NH}_{k_n}(m_n) +_{128} t).$$

We will use the tag  $t$  to encode the initial block size (before expansion to an integral number of chunks) and thus prevent length extension attacks.

The second level is a [Carter-Wegman polynomial hash](#) in  $\mathbb{F} = \mathbb{Z}/(2^{61} - 1)\mathbb{Z}$  that consumes one 64-bit half of the OH output at a time, with a Horner update.<sup>2</sup>

In other words, given the  $i$ th OH-compressed value  $\text{OH}_k(\mathbf{M}_i)$ , we consume that 128-bit value by serially feeding  $\text{OH}_k(\mathbf{M}_i) \bmod 2^{64}$  and  $\lfloor \text{OH}_k(\mathbf{M}_i)/2^{64} \rfloor$  to the polynomial hash.

Let  $n = |\mathbf{M}|$  be the number of OH blocks in the input, and  $y_j$ , for  $0 < j \leq d = 2n$ , be the stream of alternating low ( $\text{OH}_k(\mathbf{M}_{(j+1)/2}) \bmod 2^{64}$ ) and high ( $\lfloor \text{OH}_k(\mathbf{M}_{j/2})/2^{64} \rfloor$ ) 64-bit halves from OH's outputs. The polynomial hash is parameterised on a single multiplier  $f \in \mathbb{F}$  and evaluates to

$$CW_f(y) = \left( \sum_{j=1}^d y_j \cdot f^{1+d-j} \right) \bmod 2^{61} - 1,$$

a polynomial of degree  $d = 2n$ , twice the number of OH blocks.

The last step is a finalizer that reversibly mixes the  $\bmod 2^{64} - 8$  polynomial hash value to spread variation across all the output bits and pass SMHasher. This finalizer is reversible, so it has no impact on collisions.

---

<sup>2</sup>The implementation in  $\bmod 2^{64} - 8$  actually precomputes the square of the evaluation point  $f \in \mathbb{F}$ , and performs two Horner updates in parallel. It thus computes a value that does not match the polynomial in  $\bmod 2^{64} - 8$  nor in  $\bmod M_{61}$ , but does match the latter once reduced to  $\bmod M_{61}$ .

## Where do collisions come from?

For strings of 8 or fewer bytes, UMASH uses a different parameterised mixing routine for each input size  $s \in [0, 8]$ . Each of these routines is invertible, and the random parameter ensures any input value can be mapped to any output value with probability  $2^{-64}$  (i.e., they’re universal hash functions). This means the short-string mixers never collide values of the same length, and otherwise collide with probability  $\varepsilon_{\text{short}} \approx 2^{-64}$ .

For strings of 9 or more bytes, we will show that the second-level Carter-Wegman polynomial quickly becomes the dominant source of collisions: each block of 256 bytes (except the last, which may be short) is compressed to 16 bytes, which increases the polynomial’s degree by *two*, one for each 64-bit half of the compressed output.

The polynomial is in  $\mathbb{F} = \mathbb{Z}/(2^{61} - 1)\mathbb{Z}$ , so the collision probability between two polynomials of degree at most  $d$  is  $\approx d \cdot 2^{-61}$ , i.e.,  $\lceil s/256 \rceil 2^{-60}$  for strings of  $s$  or fewer bytes.

We now have to show that the probability of collision when compressing two blocks, a constant, is much smaller than the polynomial’s, for reasonably sized  $s$  (asymptotically, this clearly holds).

Let’s do so by casting the first-level block compressor as the **xor** composition of independently sampled mixers, one for each 16-byte chunk. For any block, the last mixer is a member of the **ENH family**, and takes into account both the last 16-byte chunk, and the original (pre-extension) size of the block in bytes.

This ENH mixer is  $2^{-64}$ –almost-universal: the probability that two different tuples (**chunk**, **size**) mix to the same value is at most  $2^{-64}$ .

All other chunks are mixed with **functions from the PH family**, which is  $2^{-64}$ –almost-**xor**-universal: not only will two different chunks collide with probability at most  $2^{-64}$ , but, in fact, the bitwise **xor** of their mixed values will not take any specific value with probability greater than  $2^{-64}$ .

We can show that **xoring** together the output of these PH and ENH mixers is  $2^{-63}$ –almost-universal by analysing three cases:

1. the two input blocks have the same length and different contents
2. the blocks are identical after expansion, and only differ in the length tag
3. the blocks differ in length.

We shall handle the first case, two different input blocks of identical length (same number of chunks), by induction on the number of chunks. Clearly, that’s the case for blocks of one chunk: the ENH mixer is  $2^{-64}$ –almost-universal. Assume two blocks  $x$  and  $y$  of  $n > 1$  chunks differ. If their first chunks are identical, they only collide if **xoring** the mixers for the remaining chunks (and length tag) collides, and the induction hypothesis says that happens with probability at most  $2^{-64}$ . If their first chunks differ, they’ll be mixed with a PH function  $g$  and

will only collide if  $g(x_1) \oplus g(y_1)$  matches the difference between the hash for the remainder of the chunks. We know PH is  $2^{-64}$ -almost-xor-universal, so that happens with probability at most  $2^{-64}$ .

If the blocks are identical except for the length tag, the inputs to the ENH mixer still differ, so  $2^{-64}$ -almost-universality holds.

Finally, when one block contains more chunks than the other, the two blocks only collide if the longer block's final ENH hash matches exactly the difference between the two compressed outputs so far. This happens for only one NH value  $x$ , and any one value occurs with probability at most  $2^{-63}$  ( $x = 0$ , the worst case). Two blocks with different chunk counts thus collide with probability at most  $2^{-63}$ .

In all cases, the probability of collisions between different blocks is at most  $2^{-63}$ . However, we feed the 128-bit compressed output to the polynomial hash as two values in  $\mathbb{F} = \mathbb{Z}/(2^{61} - 1)\mathbb{Z}$ , which loses slightly more than 6 bits of entropy. In total, the first-level block compressor introduces collisions with probability less than  $\approx 2^{-57}$ .

The second-level polynomial hash starts with collision probability  $\approx 2^{-60}$ , but, already at 16 blocks (4096 bytes), we find  $d = 32$ , and thus a collision probability  $\varepsilon_{poly} = \frac{32}{2^{61}-1} \approx 2^{-56}$ ; let's conservatively bump that to  $\varepsilon < 2^{-55}$  to take into account the block compressor's collisions and the rounding error from  $2^{61} - 1 < 2^{61}$ . Longer strings will follow the same progression, so we can claim a collision probability  $\varepsilon < \lceil s/4096 \rceil \cdot 2^{-55}$ .

## What if that's not enough?

It's always possible to compute a second independent hash value by passing a fresh set of parameters to the same UMASH code. In fact, we could take a page from [UMAC and recycle most of the parameters](#). However, we may not actually need double the collision resistance: for many applications (e.g., in-memory caches in front of complex computations), there is a threshold (e.g.,  $2^{-80}$ — $2^{-70}$ , or  $10^{-24}$ — $10^{-21}$ ) after which collision rates are dominated by the probability of hardware failure or data center incineration.

For such applications, UMASH's fingerprinting mode wrings a few more independent bits out of the PH and ENH values already computed for each chunk in a block, merges them into a secondary compressed value for each block, and feeds that to an independent Carter-Wegman polynomial string hash function to obtain a secondary 64-bit hash value. Combined with the primary UMASH hash value, the probability of collisions for the resulting 128-bit fingerprint shrinks to  $\varepsilon_{fp} < \lceil s/2^{26} \rceil^2 \cdot 2^{-83}$ : less than  $2^{-83}$  for strings up to 64 MB, and less than  $2^{-70}$  at 5 GB.

In fingerprinting mode, UMASH generates inputs for a secondary Carter-Wegman polynomial hash function by deriving an additional chunk for each block as the

**xor** of the input chunks and their corresponding **OH** parameters, and by applying nearly-reversible *public* shufflers  $s_{n,i}$  to each **PH** output  $\text{PH}_{k_i}(m_i)$ . In other words, we derive an additional “checksum” chunk for each block as  $\bigoplus_{i=1}^n m_i \oplus k_i$ , and modify the primary UMASH function’s mixed **PH** values before **xoring** everything them together.

The checksum ensures that any two blocks that differ now differ in at least two chunks. Before **xoring** it in the secondary **OH'** compressor’s output, it is mixed with  $\text{PH}_{\text{checksum}}$  (parameterised with an independently sampled 128-bit value).

With the additional checksum chunk, it becomes very unlikely that two blocks that differ only differ in one mixed value. The shufflers  $s_{n,i}$  help reduce the probability that differences in the mixed values for two pairs of chunks fully cancel each other out: they now have to simultaneously cancel out in the primary UMASH value (i.e., when **xored** together), and in the secondary one (**xored** after applying  $s_{n,i}$ ).

The idea of deriving an additional checksum chunk and shuffling mixed values to make cancellation unlikely came from discussion with Jim Apple, [who fully exploits the concept in Halftime Hash](#). Apple builds on [Nandi’s](#) and [Ishai et al’s](#) analyses of “Encode, Hash, Combine” and “Expand, Randomise, Shrink.” Unfortunately for us, UMASH’s secondary hash is too simplistic for these analyses to apply directly. We have to analyse our weaker construction the hard way.

We only shuffle the **PH** values inherited from the primary UMASH computation. For the last **PH** value in a block of  $n$  chunks,  $s_{n,n-1}(x) = x \ll 1$ , where the bit shift is computed independently over  $x$ ’s two 64-bit halves. For other **PH** values,  $s_{n,i}(x) = (x \ll 1) \oplus (x \ll (n - i))$ ; again, the left shift  $\ll$  operates on the two 64-bit lanes independently.

The shuffling functions are linear over bitvectors and easy to implement with vectorised additions or shifts that operate on independent 64-bit lanes. The computational core of the compression function **OH** consists of the **PH** and **ENH** mixers. Computing a fingerprint only adds one **PH** per block to that core cost (compare Figure 2 with Figure 1): UMASH’s block compressor

$$\text{OH}_k(m) = \left( \bigoplus_{i=1}^{n-1} \text{PH}_{k_i}(m_i) \right) \oplus (\text{NH}_{k_n}(m_n) +_{128} t)$$

turns into

$$\text{OH}'_k(m) = \left( \bigoplus_{i=1}^{n-1} s_{n,i}(\text{PH}_{k_i}(m_i)) \right) \oplus (\text{NH}_{k_n}(m_n) +_{128} t) \oplus \text{PH}_{\text{checksum}} \left( \bigoplus_{i=1}^n k_i \oplus m_i \right),$$

where the additional **PH** mixes the checksum derived from all input chunks.

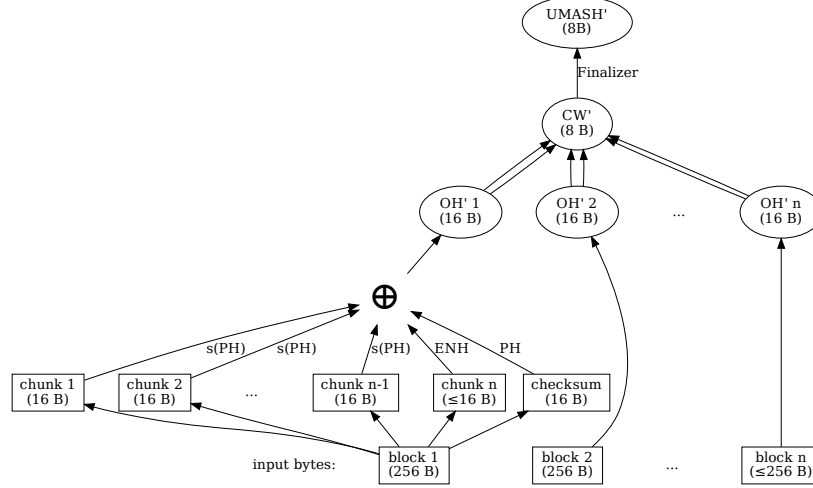


Figure 2: The structure of UMASH’s secondary hash function

Sharing this core between the primary and secondary hash functions means that the overhead of computing both their values, compared to computing only the primary 64-bit hash value, consists of one more PH plus some vectorisable left shifts (or additions) and **xors** for each block.

The shuffling functions ( $s_{n,n-1}(x) = x \ll 1$ , and  $s_{n,i}(x) = (x \ll 1) \oplus (x \ll (n-i))$  when  $0 < i < n-1$ ) exhibit three crucial properties for our collision bounds:

1. for all  $0 < i < n$ ,  $s_{n,i}(x) \oplus s_{n,i}(y) = s_{n,i}(x \oplus y)$
2. for all  $0 < i < n$ ,  $f(x) = x \oplus s_{n,i}(x)$  is invertible
3. for any two distinct  $0 < i < j < n$ , **xoring** the two shufflers  $s_{n,i}$  and  $s_{j,n}$  together yields a function  $h(x) = s_{n,i}(x) \oplus s_{j,n}(x)$  with a small (low rank) null space. In other words, while  $h$  isn’t invertible, it’s “pretty close.”

### What’s the quality of this secondary compressor?

The shufflers  $s_{n,i}$  are *nearly* reversible: they only lose two bits to  $\ll 1$  (one for each 64-bit half, the remaining identity or **xor**-shift is reversible). In fact, PH’s output always fits in 127 bits (the carryless multiplication of two 64-bit values always has a 0 in the most significant bit of the 128-bit result), so the composition of  $s$  and PH is  $2^{-63}$ -almost-**xor**-universal. The checksum’s plain PH is  $2^{-64}$ -almost-**xor**-universal, and the ENH mixer is still  $2^{-64}$ -almost-universal.

After accounting for the bits lost when converting each 64-bit half to  $\mathbb{F}$  for the polynomial hash, the new block compressor collides different blocks with



probability  $< 2^{-56}$ . Combined with an independently sampled second-level polynomial string hash, we find an end-to-end collision probability  $\varepsilon_{\text{secondary}} < \lceil s/8192 \rceil \cdot 2^{-54}$ . For long enough strings, the new secondary hash is just as good as the original UMASH.

The real question is what happens to the probability of collision for the secondary UMASH hash function when the primary one collides? The two functions clearly aren't independent: the secondary one reuses all the primary's PH and ENH outputs, and merely shuffles some of them *deterministically* before **xoring** everything together and with the new checksum chunk's PH value.

In the common case, the collision in the primary UMASH function is introduced by the polynomial hash (i.e., the block compressor doesn't collide). Since the secondary function's polynomial is completely independent, the probability of a collision in the secondary hash function is still  $\varepsilon_{\text{secondary}} < \lceil s/8192 \rceil \cdot 2^{-54}$ .

Now, what's the probability that both block compressors collide simultaneously, for a given pair of distinct blocks? We will examine three cases in order to show that this happens with probability at most  $2^{-99}$ :

1. the additional checksum chunks differ between the two blocks
2. the chunk counts differ
3. the chunk counts and the checksum chunks are both identical, so the blocks must differ in at least two original (non-checksum) chunks.

The first two cases are easy. We shall further split the last in two subcases:

- a. The blocks are identical except for two chunks, one of which is the final ENH-mixed chunk
- b. The blocks differ in at least two PH-mixed chunks.

### The two easy cases

When the derived checksums differ, PH's  $2^{-64}$ -almost-**xor**-universality means that the secondary compressor collides with probability at most  $2^{-64}$ , regardless of what happens in the primary compressor.

For blocks of different chunk counts, the checksums match with probability at most  $2^{-128}$ : the checksums  $\bigoplus m_i \oplus k_i$  include different random 128-bit parameters  $k_i$  (the longer block includes more random parameters). This leaves a collision probability  $\approx 2^{-64}$  for the secondary compressor, even when the primary compressor collides.

In both easy cases, we find a probability of simultaneous collisions in the primary and second block compressors less than  $2^{-126}$ .

### The last, harder case (subcase a)

Otherwise, we have two blocks with identical chunk count  $n$ , and matching checksums. This can only happen if at least *two* original chunks differ. We must

analyse two subcases: either they differ in exactly two original chunks, one of which is the last chunk (mixed with ENH), or they differ in at least two original chunks that are mixed with PH.

Let's handle the first subcase, where the two blocks differ in the  $j$ th and the  $n$ th chunks. Let  $h_j$  be the PH output for the  $j$ th chunk, and  $h'_j$  that for the second block, and let  $h_n$  be the ENH output for the first block, and  $h'_n$  that for the second.

What's the probability that both compressors collide on that input? Assume the primary compressor collides, i.e.  $h_j \oplus h_n = h'_j \oplus h'_n$ . The secondary compressor collides iff  $s_{n,j}(h_j) \oplus h_n = s_{n,j}(h'_j) \oplus h'_n$ .

Xoring the two equations together shows that a collision in both compressors implies

$$s_{n,j}(h_j) \oplus h_j = s_{n,j}(h'_j) \oplus h'_j.$$

The structure of the shufflers  $s$  is such that  $s_{n,j}(h_j) \oplus h_j$  is invertible (equivalent to a xor-shift-shift expression). The two compressors thus collide only when  $h_j = h'_j$ . Since  $h_j \oplus h_n = h'_j \oplus h'_n$  (we assumed the primary compressor collided), this also implies  $h_n = h'_n$ .

By  $2^{-64}$ -almost-universality, both equalities happen simultaneously with probability at most  $2^{-128}$ . In this penultimate subcase, the probability of simultaneous collisions in the primary and secondary block compressors is  $2^{-128}$ .

### The last, harder case (subcase b)

In this final subcase, we have two blocks that differ in at least two PH-mixed chunks  $i$  and  $j$  ( $0 < i < j < n$ ). Let's find the probability that both compressors collide in this manner.

We can simplify the collision conditions to

$$h_i \oplus h_j = h'_i \oplus h'_j \oplus y,$$

and

$$s_{n,i}(h_i) \oplus s_{n,j}(h_j) = s_{n,i}(h'_i) \oplus s_{n,j}(h'_j) \oplus z,$$

where  $y$  and  $z$  are constants generated adversarially, but without any knowledge of the random parameters.

Let  $\Delta_i = h_i \oplus h'_i$ , and  $\Delta_j = h_j \oplus h'_j$ . We have

$$\Delta_i \oplus \Delta_j = y,$$

and

$$s_{n,i}(\Delta_i) \oplus s_{n,j}(\Delta_j) = z.$$

Applying the linear function  $s_{n,i}$  to the first equation yields the weaker

$$s_{n,i}(\Delta_i) \oplus s_{n,i}(\Delta_j) = s_{n,i}(y).$$

After **xoring** that with the second equation,

$$s_{n,i}(\Delta_i) \oplus s_{n,j}(\Delta_j) = z,$$

we find

$$s_{n,i}(\Delta_j) \oplus s_{n,j}(\Delta_j) = s_{n,i}(y) \oplus z.$$

We can expand the left-hand side to

$$\begin{aligned} (\Delta_j \ll (n-i)) \oplus (\Delta_j \ll (n-j)) & \quad \text{if } j < n-1, \text{ or} \\ \Delta_j \ll (n-i) & \quad \text{if } j = n-1. \end{aligned}$$

In both cases, we find a linear function of  $\Delta_j$  with a small null space (at most  $2^{2(n-i)}$ , because the shifts operate on independent 64-bit lanes). Since  $i > 0$  and  $n = 16$ , there are at most  $2^{30}$  values in that null space, so at most  $2^{30}$  values of  $\Delta_j$  can satisfy

$$s_{n,i}(\Delta_j) \oplus s_{n,j}(\Delta_j) = s_{n,i}(y) \oplus z,$$

a necessary condition for a collision in both compressors.

By the  $2^{-64}$ -almost-xor-universality of PH, this happens with probability at most  $2^{-34}$ . We can improve that bound by one bit because the output of PH only spans 127 bits: a carryless multiplication of 64-bit values never populates the most significant bit of its 128-bit output. The left shifts thus actually discard at most  $2^{29}$  bits of  $\Delta_j$  (half the null space can't be generated by PH) which means the equality above happens with probability at most  $2^{-35}$ .

Finally, for any given  $\Delta_j$ , there is one  $\Delta_i$  such that  $\Delta_i \oplus \Delta_j = y$ , so, by  $2^{-64}$ -almost-universality of PH, the combined collision probability for *both* compressors is at most  $2^{-99}$ .

### Collision bounds, assemble!

For short inputs (at most 8 bytes), the secondary SplitMix64s use a different set of random constants. Since each modified SplitMix64 is a universal bijection, there is exactly one value for the random parameter that will let the longer value collide in the first UMASH, and again exactly one value for the other random parameter in the secondary UMASH. This gets us a collision probability of  $2^{-128}$ .

For longer inputs (more than 8 bytes), we just showed the probability that both compressors collide is at most  $2^{-99}$ , or  $\approx 2^{-87}$  once we take into account the entropy lost to  $\mathbb{F}$ .

There are three scenarios under which collisions could happen for the combined 128-bit fingerprint of long inputs:

1. both compressors collide simultaneously,  $p \approx 2^{-87}$
2. one compressor collides ( $\approx 2^{-62}$ ), and the polynomial string hash collides for the other ( $\approx 2^{-25}$  for  $2^{36}$  blocks, i.e., 16 TB), so  $p \approx 2^{-87}$  for strings up to 16 TB
3. neither compressor collides, but both polynomial hashes do, the asymptotically more likely option by far. This case also covers collisions between long and short values.

The two independent string hashes collide with probability  $\approx d^2 \cdot 2^{-122}$ , for two polynomials of degree  $d$ , i.e.,  $\lceil s/256 \rceil^2 \cdot 2^{-120}$  for strings of  $s$  or fewer bytes.

At  $2^{18}$  blocks (i.e.,  $s = 2^{26}$ ), we find the two string hashes collide with probability at most  $\varepsilon_{poly^2} \approx 2^{-84}$ . Let's conservatively bump that to  $\varepsilon_{fp} < 2^{-83}$  to take into account the other collision scenarios and the rounding errors from  $2^{61} - 1 < 2^{61}$ . Longer strings will follow the same progression, so we can claim a combined collision probability  $\varepsilon_{fp} < \lceil s/2^{26} \rceil^2 \cdot 2^{-83}$ .

That's  $2^{-83}$  for strings up to 64 MB, and less than  $2^{-70}$  at 5 GB.

## Reference UMASH implementation in Python

```
from collections import namedtuple
from functools import reduce
import random
import struct

# We work with 64-bit words
W = 2 ** 64

# We chunk our input 16 bytes at a time
CHUNK_SIZE = 16

# We work on blocks of 16 chunks
BLOCK_SIZE = 16
```

### Carry-less multiplication

Addition in the ring of polynomials over GF(2) is simply bitwise `xor`. Multiplication is slightly more complex to describe in software. The code below shows a pure software implementation; in practice, we expect to use hardware carry-less multiplication instructions like [CLMUL](#) on x86-64 or [VMULL](#) on ARM.

While non-cryptographic hash functions have historically avoided this operation, it is now a key component in widely used encryption schemes like [AES-GCM](#). We can expect a hardware implementation to be available on servers and other higher end devices.

```
def gfmul(x, y):
```

```

    """Returns the 128-bit carry-less product of 64-bit x and y."""
    ret = 0
    for i in range(64):
        if (x & (1 << i)) != 0:
            ret ^= y << i
    return ret

```

## Key generation

A UMASH key consists of one multiplier for polynomial hashing in  $\mathbb{F} = \mathbb{Z}/(2^{61} - 1)\mathbb{Z}$ , and of 32 OH words of 64 bit each (16 chunks of 16 bytes each). The secondary hash appends two more “twisting” 64-bit words to the OH array, and another polynomial multiplier.

The generation code uses rejection sampling to avoid weak keys: the polynomial key avoids 0 (for which the hash function constantly returns 0), while the OH keys avoid repeated values in order to protect against [known weaknesses](#) in the extremely similar NH.

```

# We encode a OH key and a secondary twisting constant in one UmashKey.
TWISTING_COUNT = 2

```

```

# For short keys, the secondary hash's set of constants lives four
# u64s ahead: that's the value we used in the earlier more symmetric
# incarnation of UMASH, and it seemed to work well.
SHORT_KEY_SHIFT = 4

```

```

UmashKey = namedtuple("UmashKey", ["poly", "oh"])

```

```

def is_acceptable_multiplier(m):
    """A 61-bit integer is acceptable if it isn't 0 mod 2**61 - 1.
    """
    return 1 < m < (2 ** 61 - 1)

```

```

def generate_key(random=random.SystemRandom()):
    """Generates a UMASH key with `random`"""
    poly = 0
    while not is_acceptable_multiplier(poly):
        poly = random.getrandbits(61)
    oh = []
    for _ in range(2 * BLOCK_SIZE + TWISTING_COUNT):
        u64 = None
        while u64 is None or u64 in oh:
            u64 = random.getrandbits(64)
        oh.append(u64)

```

```
return UmashKey(poly, oh)
```

## Short input hash

Input strings of 8 bytes or fewer are expanded to 64 bits and shuffled with a modified [SplitMix64 update function](#). The modification inserts a 64-bit data-independent “noise” value in the SplitMix64 code; that noise differs unpredictably for each input length in order to prevent extension attacks.

This conversion to 8 bytes is optimised for architectures where misaligned loads are fast, and attempts to minimise unpredictable branches.

```
def vec_to_u64(buf):
    """Converts a <= 8 byte buffer to a 64-bit integer. The conversion is
    unique for each input size, but may collide across sizes."""

    n = len(buf)
    assert n <= 8
    if n >= 4:
        lo = struct.unpack("<I", buf[0:4])[0]
        hi = struct.unpack("<I", buf[-4:])[0]
    else:
        # len(buf) < 4. Decode the length in binary.
        lo = 0
        hi = 0
        # If the buffer size is odd, read its first byte in `lo`
        if (n & 1) != 0:
            lo = buf[0]

        # If the buffer size is 2 or 3, read its last 2 bytes in `hi`.
        if (n & 2) != 0:
            hi = struct.unpack("<H", buf[-2:])[0]
        # Minimally mix `hi` in the `lo` bits: SplitMix64 seems to have
        # trouble with the top ~4 bits.
        return (hi << 32) | ((hi + lo) % (2 ** 32))
```

In addition to a full-blown **key**, UMASH accepts a **seed** parameter. The **seed** is used to modify the output of UMASH without providing any bound on collision probabilities: independent keys should be generated to bound the probability of two inputs colliding under multiple keys.

The short vector hash modifies the [SplitMix64 update function](#) by reversibly **xoring** the state with a data-independent value in the middle of the function. This additional **xor** lets us take the input **seed** into account, works around `SplitMix64(0) = 0`, and adds unpredictable variability to inputs of different lengths that `vec_to_u64` might convert to the same 64-bit integer. It comes in after one round of **xor-shift-multiply** has mixed the data around: moving it

earlier fails SMHasher’s Avalanche and Differential Distribution tests.

Every step after `vec_to_u64` is reversible, so inputs of the same byte length  $\leq 8$  will never collide. Moreover, a random value from the OH key is injected in the process; while each step is trivial to invert, the additional `xor` is only invertible if we know the random OH key.

Two values of the same size never collide. Two values  $x$  and  $y$  of different lengths collide iff  $k_{\text{len}(x)} \oplus k_{\text{len}(y)}$  has exactly the correct value to cancel out the difference between  $x$  and  $y$  after partial mixing. This happens with probability  $2^{-64}$ .

When we compute the secondary UMASH, collisions for the first and second hashes are independent:  $k_{\text{len}(x)} \oplus k_{\text{len}(y)}$  and  $k_{\text{len}(x)+S} \oplus k_{\text{len}(y)+S}$  are independent for chunk shift constant  $S \geq 2$  (we use  $S = 2$ , i.e., four 64-bit words).

```
def umash_short(key, seed, buf):
    """Hashes a buf of 8 or fewer bytes with a pseudo-random permutation."""
    assert len(buf) <= 8

    # Add an unpredictable value to the seed. vec_to_u64` will
    # return the same integer for some inputs of different length;
    # avoid predictable collisions by letting the input size drive the
    # selection of one value from the random `OH` key.
    noise = (seed + key[len(buf)]) % W

    h = vec_to_u64(buf)
    h ^= h >> 30
    h = (h * 0xBF58476D1CE4E5B9) % W
    h ^= h >> 27
    # Our only modification to SplitMix64 is this `xor` of the
    # randomised `seed` into the shuffled input. A real
    # implementation can expose marginally more instruction-level
    # parallelism with `h = (h ^ noise) ^ (h >> 27)`.
    h ^= noise
    h = (h * 0x94D049BB133111EB) % W
    h ^= h >> 31
    return h
```

## Long input hash

Inputs of 9 bytes or longer are first compressed with a 64-bit OH function (128-bit output), and then accumulated in a polynomial string hash.

When the input size is smaller than 16 bytes, we expand it to a 16-byte chunk by concatenating its first and last 8 bytes, while generating the “tag” by `xoring` the original byte size with the seed (this logic would also work on 8-byte inputs, but results in slower hashes for that common size). We pass this single input

chunk to the OH compression function, and feed the result to the polynomial hash. The tag protects against extension attacks, and the OH function reduces to the low latency [NH compression function](#) for single-chunk inputs.

We use NH for the last chunk in each OH block because, while throughput is challenging with NH compared to PH, the latency of NH's integer multiplications tends to be lower than that of PH's carry-less multiplications.

Longer inputs are turned into a stream of 16-byte chunks by letting the last chunk contain redundant data if it would be short, and passing blocks of up to 16 chunks to OH. The last block can still be shorter than 16 chunks of 16 bytes (256 bytes), but will always consist of complete 16-byte chunks. Again, we protect against extension collisions by propagating the original byte size in each compressed block.

This approach simplifies the end-of-string code path when misaligned loads are efficient, but does mean we must encode the original string length  $s$  somewhere. We use the [same trick as VHASH](#) and `xor ( $s \bmod 256$ )` with the seed to generate the last block's tag value. Since only the last block may span fewer than 256 bytes, this is equivalent to `xoring  $|M_i| \bmod 256$` , a block's size modulo the maximum block size, with the seed to generate each block's tag.

Tags are 128-bit values, and `xoring` a 64-bit seed with  $s \bmod 256$  yields a 64-bit value. We avoid a carry in the NH step by letting the tag be equal to the `xored` value shifted left by 64 bits (i.e., we only populate the high 64-bit half).

```
def chunk_bytes(buf):
    """Segments bytes in 16-byte chunks; generates a stream of (chunk,
    original_byte_size). The original_byte_size is always CHUNK_SIZE,
    except for the last chunk, which may be short.
```

```
    When the input is shorter than 16 bytes, we convert it to a
    16-byte chunk by reading the first 8 and last 8 bytes in the
    buffer. We know there are at least that many bytes because
    `umash_short` handles inputs of 8 bytes or less.
```

```
    If the last chunk is short (not aligned on a 16 byte boundary),
    yields the partially redundant last 16 bytes in the buffer. We
    know there are at least 16 such bytes because we already
    special-cased shorter inputs.
```

```
    """
    assert len(buf) >= CHUNK_SIZE / 2
    n = len(buf)
    if n < CHUNK_SIZE:
        yield buf[: CHUNK_SIZE // 2] + buf[-CHUNK_SIZE // 2 :], n
        return
```

```
    for i in range(0, len(buf), CHUNK_SIZE):
```



```

        if i + CHUNK_SIZE <= n:
            yield buf[i : i + CHUNK_SIZE], CHUNK_SIZE
        else:
            yield buf[n - CHUNK_SIZE :], n - i

def blockify_chunks(chunks):
    """Joins chunks in up to 256-byte blocks, and generates
    a stream of (block, original_byte_size)."""
    acc = []
    size = 0
    for chunk, chunk_size in chunks:
        assert len(chunk) == CHUNK_SIZE
        assert len(acc) <= BLOCK_SIZE
        if len(acc) == BLOCK_SIZE:
            # Only the last chunk may be short.
            assert size == CHUNK_SIZE * BLOCK_SIZE
            yield acc, size
            acc = []
            size = 0
        acc.append(chunk)
        size += chunk_size
    assert acc
    yield acc, size

```

Given a stream of blocks, the last of which may be partial, we compress each block of up to 256 bytes (but always a multiple of 16 bytes) with the `OH` function defined by the `key`. This yields a stream of 128-bit `OH` outputs.

We propagate a `seed` argument all the way to the individual `OH` calls. We use that `seed` to elicit different hash values, with no guarantee of collision avoidance. Callers that need collision probability bounds should generate fresh keys.

In order to prevent extension attacks, we generate each block’s tag by `xoring` the seed with that block’s byte size modulo 256 (the maximum byte size), and shifting the resulting 64-bit value left by 64 bits, much like [VHASH](#) does; this tag is added to the `NH`-mixed value, to form an `ENH` value.

For the secondary `UMASH`, we generate an additional “checksum” chunk by `xoring` together the input chunks and the corresponding `PH` and `NH` parameters. That last checksum chunk is `PH`-mixed with the additional “twisting” parameters. Finally, the original `ENH` value is `xored` in with the checksum’s `PH`, and with shuffled versions of the original `PH` values.

Computing this secondary hash on top of the primary one only incurs one more carryless multiplication (the computational core of the block compressor)!

```

def oh_mix_one_block(key, block, tag, secondary=False):
    """Mixes each chunk in block."""

```

```

mixed = list()
lrc = (0, 0) # we generate an additional chunk by xoring everything together
for i, chunk in enumerate(block):
    ka = key[2 * i]
    kb = key[2 * i + 1]
    xa, xb = struct.unpack("<QQ", chunk)
    lrc = (lrc[0] ^ (ka ^ xa), lrc[1] ^ (kb ^ xb))
    if i < len(block) - 1:
        mixed.append(gfmul(xa ^ ka, xb ^ kb))
    else:
        # compute ENH(chunk, tag)
        xa = (xa + ka) % W
        xb = (xb + kb) % W
        enh = ((xa * xb) + tag) % (W * W)
        enh ^= (enh % W) * W
        mixed.append(enh)
if secondary: # We use the checksum chunk only in the secondary hash
    mixed.append(gfmul(lrc[0] ^ key[-2], lrc[1] ^ key[-1]))
return mixed

def shuffle(x, i, n):
    """Computes our almost-xor-shift of x, on parallel 64-bit halves.

    If i == n, this function is the identity
    If i == n - 1, it returns (x << 1)
    Otherwise, it returns [x << (n - i)] ^ (x << 1).

    The reverse ordering in the shifts (smaller values of `i` shift
    more) makes it possible to implement the surrounding loop with
    accumulators that are shifted by one at each inner iteration.
    """

    def parallel_shift(x, s):
        lo, hi = x % W, x // W
        lo = (lo << s) % W
        hi = (hi << s) % W
        return lo + W * hi

    shift = n - i
    if shift == 0:
        return x
    if shift == 1:
        return parallel_shift(x, 1)
    return parallel_shift(x, shift) ^ parallel_shift(x, 1)

```

```

def oh_compress_one_block(key, block, tag, secondary=False):
    """Applies the `OH` hash to compress a block of up to 256 bytes."""
    mixed = oh_mix_one_block(key, block, tag, secondary)
    if secondary is False:
        # Easy case (fast hash): xor everything
        return reduce(lambda x, y: x ^ y, mixed, 0)

    acc = mixed[-1]
    n = len(mixed) - 1
    for i, mixed_chunk in enumerate(mixed[:-1]):
        acc ^= shuffle(mixed_chunk, i + 1, n)
    return acc

def oh_compress(key, seed, blocks, secondary=False):
    """Applies the `OH` compression function to each block; generates
    a stream of compressed values"""
    for block, block_size in blocks:
        size_tag = block_size % (CHUNK_SIZE * BLOCK_SIZE)
        tag = (seed ^ size_tag) * W
        yield oh_compress_one_block(key, block, tag, secondary)

```

OH is a fast compression function. However, it doesn't scale to arbitrarily large inputs. We split each of its 128-bit outputs in two 64-bit halves, and accumulate them in a single polynomial hash modulo  $2^{64} - 8 = 8 \cdot (2^{61} - 1)$ .

Modular multiplication in that ring is more efficient when one of the multipliers is known to be less than  $2^{61}$ . A random multiplier  $f \in \mathbb{F} = \mathbb{Z}/(2^{61} - 1)\mathbb{Z}$  naturally satisfies that constraint; so does its square  $f^2$ , once fully reduced to  $\mathbb{F}$ . We will use that to pre-compute  $f^2 \in \mathbb{F}$  and overlap most of the work in a double-pumped Horner update. The result will differ from a direct evaluation modulo  $2^{64} - 8$ , but not in  $\mathbb{F}$ , which is what matters for analyses.

Every time we feed one 64-bit half of a PH value to the polynomial in  $\mathbb{F} = \mathbb{Z}/(2^{61} - 1)\mathbb{Z}$ , we lose slightly more than 3 bits of data. The resulting collision probability for truncated PH is less than  $\lceil 2^{64}/|\mathbb{F}| \rceil^2 \cdot 2^{-63} < 2^{-56}$ . As  $s$ , the input size in bytes, grows, that's quickly dominated by the collision probability for the polynomial hash in  $\mathbb{F}$ ,  $\varepsilon_{\mathbb{F}} \leq d/|\mathbb{F}| = 2\lceil s/256 \rceil/(2^{61} - 2)$ , where the last decrement accounts for our disqualifying 0 from the set of multipliers.

The Horner update increments before multiplying: this does not impact the degree of the hash polynomial, nor its collisions probability since its inputs are randomised by OH, but having the last step be a multiplication helps satisfy SMHasher.

```

def poly_reduce(multiplier, input_size, compressed_values, initial=0):
    """Updates a polynomial hash with the `OH` compressed outputs."""

```

```

# Square the multiplier and fully reduce it. This does not affect
# the result modulo  $2^{61} - 1$ , but does differ from a
# direct evaluation modulo  $2^{64} - 8$ .
mulsq = (multiplier ** 2) % (2 ** 61 - 1)
acc = [initial]

def update(y0, y1):
    """Double-pumped Horner update (mostly) modulo  $8 * (2^{61} - 1)$ ."""
    # Perform a pair of Horner updates in  $(\text{mod } 2^{61} - 1)$ .
    reference = multiplier * (acc[0] + y0)
    reference = multiplier * (reference + y1)
    reference %= 2 ** 61 - 1

    # The real update is in  $(\text{mod } 2^{64} - 8)$ , with a multiplier2
    # reduced to  $(\text{mod } 2^{61} - 1)$ .
    acc[0] = (mulsq * (acc[0] + y0) + multiplier * y1) % (W - 8)
    # Both values should be the same  $(\text{mod } 2^{61} - 1)$ .
    assert acc[0] % (2 ** 61 - 1) == reference

for value in compressed_values:
    lo = value % W
    hi = value // W
    update(lo, hi)
return acc[0]

```

## Reversible finalisation

Mere universality does not guarantee good distribution; in particular, bit avalanche tests tend to fail. In the case of UMASH, the modulus,  $2^{64} - 8$  also creates obvious patterns: the low 3 bits end up being a mod 8 version of the mod  $2^{61} - 1$  polynomial hash.

This predictability does not impact our collision bounds (they already assume the hash outputs are modulo  $2^{61} - 1$ ), but could create clumping in data structures. We address that with an invertible [xor-rotate](#) transformation. Rotating before [xor](#) mixes both the low and high bits around, and [xoring](#) a pair of bit-rotated values guarantees invertibility ([xoring](#) a single rotate would irreversibly map both 0 and -1 to 0).

The pair of rotation constants in the finalizer, 8 and 33, was found with an exhaustive search: they're good enough for SMHasher. In theory, this is a bad finalizer, for all constants. The rotation counts are likely tied to the  $(\text{mod } 2^{64} - 8)$  polynomial hash.

```

def rotl(x, count):
    """Rotates the 64-bit value `x` to the left by `count` bits."""

```

```

ret = 0
for i in range(64):
    bit = (x >> i) & 1
    ret |= bit << ((i + count) % 64)
return ret

def finalize(x):
    """Invertibly mixes the bits in x."""
    return x ^ rotl(x, 8) ^ rotl(x, 33)

```

## Putting it all together

UMASH is the combination of `umash_short` for inputs shorter than 9 bytes, and `umash_long` for everything else. The latter hash function chops up its input in 16-byte chunks (where the last chunk is never partial, but potentially redundant), groups these chunks in blocks of up to 256 bytes, and compresses each block down to 128 bits with OH. A polynomial hash function in  $\mathbb{F} = \mathbb{Z}/(2^{61} - 1)\mathbb{Z}$  reduces that stream of compressed outputs to a single machine word.

```

def umash_long(key, seed, buf, secondary):
    assert len(buf) >= CHUNK_SIZE / 2
    blocks = blockify_chunks(chunk_bytes(buf))
    oh_values = oh_compress(key.oh, seed, blocks, secondary)
    poly_acc = poly_reduce(key.poly, len(buf), oh_values)
    return finalize(poly_acc)

def umash(key, seed, buf, secondary):
    if len(buf) <= CHUNK_SIZE / 2:
        return umash_short(
            key.oh if secondary is False else key.oh[SHORT_KEY_SHIFT:], seed, buf
        )
    return umash_long(key, seed, buf, secondary)

```

## Implementation tricks

UMASH is structured to minimise the amount of work for mid-sized inputs composed of a handful of chunks. The last 16-byte chunk stops the OH loop early to divert to a low-latency NH in (64-bit) general purpose registers, and the chunking logic avoids complex variable-length `memcpy`.

The multipliers and modulus for the polynomial hashes were chosen to simplify implementation on 64-bit machines: assuming the current hash accumulator is in  $[0, 2^{64} - 8)$ , we can increment it by a 64-bit value in three x86-64 instructions,

multiply the sum by a 61-bit multiplier in five instructions, and re-normalize the accumulator to the modulus range with two instructions followed by a predictable branch.

It’s also convenient that the range for fast multipliers,  $[0, 2^{61})$ , is as wide as the theoretical modulus: we can precompute the square of the polynomial multipliers and perform a double Horner update with two independent multiplications, which exposes more instruction-level parallelism than individual Horner updates.

For longer inputs, we can micro-optimize the PH bulk of the OH inner loop by pre-loading the key in SIMD registers (e.g., in 8 AVX-256 registers), when vectorising the  $m_i \oplus k_i$  step. That step isn’t particularly slow, but PH is so simple that a decent implementation tends to be backend-bound, so minimising the number of hardware micro-ops helps.

We only consider the size of the input at the very end of the `umash_long` function; this makes it possible to implement an incremental hashing interface. The outer polynomial hash also enables parallel out-of-order computation of the hash, as long as the I/O is delivered at 256 byte boundaries (in practice, the I/O granularity is at least 512 bytes, often 4KB or more), a useful property when hashing larger data files.

Finally, when a stronger “fingerprint” is called for, we can compute a secondary hash by reusing all the PH and ENH values, for each block, and only computing one more PH value based on a [LRC checksum](#). The secondary block compressor can be computed with accumulators, one of which is shifted for each chunk. The additional block-compressed values are fed to an independent polynomial hash, and yields a combined collision probability less than  $2^{-70}$  for inputs up to 5 GB.

## Acknowledgements

Any error in the analysis or the code is mine, but a few people helped improve UMASH and its presentation.

Thomas Ahle pointed out issues in the presentation of the new fingerprinting algorithm, and rightfully called for more attribution of prior work.

Colin Percival scanned an earlier version of this document for obvious issues, encouraged me to simplify the parameter generation process, and prodded us to think about side channels, even in data structures.

Joonas Pihlaja helped streamline my initial attempt while making the reference implementation easier to understand.

Jacob Shufro independently confirmed that he too found the reference implementation understandable, and tightened the natural language.

Phil Vachon helped me gain more confidence in the implementation tricks borrowed from VHASH after replacing the NH compression function with PH.

## References

- Apple. [HalftimeHash: Modern Hashing without 64-bit Multipliers or Finite Fields](#), 2021.
- Bernstein. [Polynomial evaluation and message authentication](#), 2007.
- Black, Halevi, Krawczyk, Krovetz, and Rogaway. [UMAC: Fast and Secure Message Authentication](#), 1999.
- Boesgaard, Scavenius, Pedersen, Christensen, and Zenner. [Badger - A Fast and Provably Secure MAC](#), 2004.
- Carter and Wegman. [Universal Classes of Hash Functions](#), 1979.
- Dai and Krovetz. [VHASH Security](#), 2007.
- Handschuh and Preneel. [Key-Recovery Attacks on Universal Hash Function based MAC Algorithms](#), 2008.
- Ishai, Kushilevitz, Ostrovsky, and Sahai. [Cryptography with Constant Computational Overhead](#), 2008.
- Lemire and Kaser. [Faster 64-bit universal hashing using carry-less multiplications](#), 2015.
- Nandi. [On the Minimum Number of Multiplications Necessary for Universal Hash Functions](#), 2013.
- Reynolds. [Basic XOR-rotates and their inverse](#), 2017.
- Yuksel, Kaps, and Sunar. [Universal Hash Functions for Emerging Ultra-Low-Power Networks](#), 2004.

## Change log

- 2022-02-23: [Link to the blog posts in this section](#).
- 2022-02-14: Clarify the presentation of the fingerprinting algorithm, and tighten its analysis.
- 2021-06-29: simplify secondary OH compression loop.
- 2021-06-28: actually document the new fingerprinting algorithm inspired by [Nandi’s “On the Minimum Number of Multiplications Necessary for Universal Hash Functions”](#)
- 2020-10-13: replace PH with OH, a variant that uses NH for the last 16-byte chunk. Analysis shows we can make every block benefit from the improved latency of NH, not just short 9-16 byte blocks. [Also described on pvk.ca](#).
- 2020-09-20: mix 9-16 byte inputs with NH instead of PH, before passing to the same polynomial hash and finalisation.

2020-09-13: use an [invertible xor-rot](#) finalizer, instead of multiply xorshift.

2020-08-27: use  $s$  for the input byte length parameter, which renders less confusingly than  $l$  (ell) in the face of font issues.

2020-08-24: [Initial public release](#).