UMASH: a fast almost universal 64-bit string hash

Paul Khuong, Backtrace I/O

2021-06-29

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, passes both Reini Urban's fork of SMHasher and Yves Orton's extended version.

The universal collision bounds for UMASH hold under the assumption that independent keys are fully re-generated with random data. While UMASH accepts a regular 64-bit "seed" value, it is merely a hint that we would like to see a different set of 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 against adversaries that can exploit side-channels or adapt to the hashed outputs, but does guarantee 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 can also be used for fingerprinting, in conjunction with a secondary hash function; the combination provides a collision probability less than 2^{-70} for input size up to 1 GB. This secondary function reuses most of the work performed in the primary function, so the 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 a 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 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 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` (`OH` because `O` is the letter between `N` and `P`).

UMASH splits an input string into a sequence of blocks $\mathbf{M}$; the first-level `OH` hash 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 that the final block's size is a multiple of 16 bytes.

Each block is compressed to 16 bytes with `OH`. The resulting outputs are then fed to a Carter-Wegman polynomial hash function, and the accumulator reversibly finalised 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 function.

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

That 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 higher 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

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

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., $P[\texttt{PH}_{k_i}(x) = \texttt{PH}_{k_i}(y)] \leq 2^{-64}$ for $x \neq y$), but in fact any specific difference $\Delta_{\texttt{XOR}}$ is equally unlikely:

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

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:

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

where $+_{64}$ denotes 64-bit modular addition, and $\cdot_{128}$ a full $64 \times 64 \to 128$-bit multiplication.

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

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

where $-_{128}$ denotes modular 128-bit subtraction.

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

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

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. In other words, given the ith OH-compressed value $\texttt{OH}_k(\mathbf{M}_i)$, we consume its 128-bit integer value by consecutively feeding $\texttt{OH}_k(\mathbf{M}_i) \bmod 2^{64}$ and $\lfloor \texttt{OH}_k(\mathbf{M}_i)/2^{64} \rfloor$ to the polynomial hash.

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

$$CW_f(y) = \left(\sum_{j=0}^{d-1} y_j \cdot f^{d-j} \right) \bmod 2^{61} - 1,$$

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

The last step is a finalizer that reversibly mixes the mod $2^{64} - 8$ polynomial hash value to improve its distribution and pass SMHasher.

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 collides 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 in $\mathbb{F} = \mathbb{Z}/(2^{61} - 1)\mathbb{Z}$, so the collision probability for 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, that 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 $(\texttt{chunk}, \texttt{size})$ 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 snow show that xor-ing together the output of these mixers is $2^{-64}-$almost-universal, 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 chunk is 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 chunk differs, they'll be mixed with a `PH` function g, and they 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 the proof of $2^{-64}-$almost-universality holds.

Finally, if 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). Collisions thus occur between blocks with different chunk counts 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 less than 7 bits of entropy. In short, the first-level block compressor introduces collisions with probability less than 2^{-56}.

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 $\varepsilon_{poly} < 2^{-56}$; let's conservatively bump that to $\varepsilon < 2^{-55}$ to take the block compressor's collisions into account. 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?

We could use a Toeplitz extension to reuse most of the random parameters. However, we don't actually need double the entropy: in practice, anything stronger than $\approx 2^{-70}$ is dominated by hardware failure, incinerated datacenters, or war.

The plan is to apply nearly-reversible *public* shufflers xs_i to each `PH` output, and to generate an additional chunk for each block by xoring together the chunks and the corresponding `PH` parameters. In other words, we will derive an additional "checksum" chunk for each block with $\bigoplus m_i \oplus k_i$. This checksum ensures that any two blocks that differ now differ in at least two chunks, and is mixed with $\texttt{PH}_{\text{checksum}}$ (i.e., parameterised with an independently sampled 128-bit value) before xoring it in the secondary compressor's output.

The shuffler xs is the identity for the `ENH` value, and for the checksum `PH` mixed value. For the last `PH` value in a block of n chunks, $xs_{n-1}(x) = x{\ll}1$, where the bit shift is computed independently over x's two 64-bit halves. For other `PH` values, $xs_{n-i}(x) = (x{\ll}1) \oplus (x{\ll}i)$.

What's the quality of this secondary compressor?

The xs shufflers are *nearly* reversible: they only lose two bits to the $\ll 1$ (a plain xor-shift is fully reversible). The composition of xs and `PH` is thus $2^{-62}-$almost-XOR-universal. The checksum's plain `PH` is $2^{-64}-$almost-XOR-universal, and the `ENH` mixer is still $2^{-64}-$almost-universal (and still collides blocks of different length with probability at most 2^{-63}).

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 at most 2^{-55}. 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 large enough strings, the new secondary hash is just as good as the original UMASH.

However, the real question is what's the probability of collision for the secondary UMASH hash when the original collides? The two hashes clearly aren't independent, since the secondary hash reuses *all* the `PH` and `ENH` outputs, and merely shuffles some of them before xoring with each other and with the new checksum chunk's own `PH` value.

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

Now, what's the probability that both the secondary block compressor collides when the primary compressor already collides?

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 happened to 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 less than 2^{-63} for the secondary compressor, even when the primary compressor collides.

Otherwise, we have two blocks with identical chunk count n, and matching checksums. That can only happen if at least *two* other chunks differ. We must analyse two cases: either they differ in exactly two (original) chunks, one of which is the last one, mixed with `ENH`, or they differ in at least two chunks that are mixed with `PH`.

In the first case, where the two blocks differ in the jth and the nth chunks, let h_j be the `PH` output for the jth 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? The primary compressor collides when $h_j \oplus h_n = h'_j \oplus h'_n$, and the secondary compressor when

$xs_j(h_j) \oplus h_n = xs_j(h'_j) \oplus h'_n$.

If we xor the two equations together, we find $xs_j(h_j) \oplus h_j = xs_j(h'_j) \oplus h'_j$,

The structure of xs_j is such that $xs_j(h_j) \oplus h_j$ is invertible (equivalent to a xor-shift-shift expression), and thus invertible. The two compressors collide only when $h_j = h'_j$; since $h_j \oplus h_n = h'_j \oplus h'_n$,, this also implies $h_n = h'_n$.

By $2^{-64}-$almost-universality, both equalityies happen simultaneously with probability at most 2^{-128}. Thus, even if we assume the primary compressor collides, the secondary collides with probability at most 2^{-64}, for this penultimate case.

In the last case, we have two blocks that differ in at least two PH-mixed chunks i and j ($i < j$). Let's find the condition that both compressors collide.

We can simplify the collision condition to $h_i \oplus h_j = h'_i \oplus h'_j \oplus y$, and $xs_i(h_i) \oplus xs_j(h_j) = xs_i(h'_i) \oplus xs_j(h'_j) \oplus z$, where y and z are constants generated adversarially, but without knowing anything about 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 $xs_i(\Delta_i) \oplus xs_j(\Delta_j) = z$.

Let's apply xs_i to the first condition: $xs_i(\Delta_i) \oplus xs_i(\Delta_j) = xs_i(y)$, since xs_i is a linear function.

After xoring that with the second condition $xs_i(\Delta_i) \oplus xs_j(\Delta_j) = z$, we find $xs_i(\Delta_j) \oplus xs_j(\Delta_j) = xs_i(y) \oplus z$.

We can expand the left-hand side to $(\Delta_j \ll (n-i)) \oplus (\Delta_j \ll (n-j))$ or $\Delta_j \ll (n-j)$ if $i = 1$. In both cases, we find an affine function of Δ_j with a small null space (at most 2^{2j}).

There are thus at most $2^{2j} \leq 2^{2(n-1)} = 2^{30}$ values in that null space, so at most 2^{30} values of Δ_j can satisfy $xs_i(\Delta_j) \oplus xs_j(\Delta_j) = xs_i(y) \oplus z$, a necessary condition for a collision in both compressors.

By $2^{-64}-$almost-XOR-universality of PH, this happens with probability at most 2^{-34}. Finally, for any given Δ_j, there is at most one Δ_i such that $\Delta_i \oplus \Delta_j = y$, so the combined collision probability for *both* compressors is at most 2^{-98}.

When the first compressor collides, the probability that the second collides as well is at most 2^{-34}.

The probability that the secondary compressor collides when the regular UMASH compressor does is thus at most 2^{-34}, or less than 2^{-27} once we take into account the entropy lost to $\mathbb{F}$.

For long (more than 8 bytes) inputs, the combined collision probability is as follows.

The first block compression level collides with probability at most 2^{-83} (comfortably less than 2^{-70}).

The first hash collides end-to-end with probability $\varepsilon < \lceil s/4096 \rceil 2^{-55}$.

In the common case, that's because the compressed blocks differ, but the polynomial hash collided anyway. The secondary hash then collides with probability $\varepsilon_{\text{secondary}} < \lceil s/8192 \rceil \cdot 2^{-54}$.

Otherwise, this happens because the compressed blocks collide. The secondary block compressor then collides with probability at most 2^{-34}, and the whole thing thus collides with probability $\varepsilon_{\text{secondary}} \leq 2^{-33}$ for inputs shorter than 16 GB.

The probability that *both* compressors collide is thus less than $\lceil s/4096 \rceil 2^{-88}$ for input sizes up to 16 GB, or less than 2^{-70} for input size up to 1 GB.

As for short inputs, we will use a shifted set of constants for the random value injected in SplitMix64; 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}.

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 sofware. 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 protects 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 `xor`ing 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 ≤ 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 $k_{\texttt{len}(x)} \oplus k_{\texttt{len}(y)}$ 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: $k_{\texttt{len}(x)} \oplus k_{\texttt{len}(y)}$ and $k_{\texttt{len}(x)+S} \oplus k_{\texttt{len}(y)+S}$ 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 longers 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 `xor`ing 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 mod 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 $|\mathbf{M}_i|$ mod 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 mod 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 `xor`ing 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 shuffled versions of the original `PH` values.

Computing the secondary hsh on top of the primary one only incurs one more carryless multiplication (the 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 only use the checksum chunk in the secondary hash
        mixed.append(gfmul(lrc[0] ^ key[-2], lrc[1] ^ key[-1]))
    return mixed


def xs(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 ^= xs(mixed_chunk, i + 1, n)
    return acc


def oh_compress(key, seed, blocks, secondary):
    """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.

Note that 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 improves distribution.

```
def poly_reduce(multiplier, input_size, compressed_values):
```

```
    """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 = [0]

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

        # The real update is in (mod 2**64 - 8), with a multiplier^2
        # reduced to (mod 2**61 - 1).
        acc[0] = (mulsq * (acc[0] + y0) + multiplier * y1) % (W - 8)
        # Both values should be the same (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 `xor`ing a pair of bit-rotated values guarantees invertibility (`xor`ing 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 `(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.

Two short inputs never collide if they have the same length; in general the probability that they collide satisfies $\varepsilon_{\text{short}} \leq 2^{-64}$.

The probability of collision for longer inputs is the sum of the collision probability for one `PH` block after truncation to $\mathbb{F}^2$, $\varepsilon_{\texttt{PH}} < 9^2 \cdot 2^{-63} \approx 0.64 \cdot 2^{-56}$, and that of the polynomial hash, $\varepsilon_{\mathbb{F}} \approx d \cdot 2^{-61}$, where $d = 2\lceil s/256 \rceil$ is twice the number of `PH` blocks for an input length $s > 8$. Together, this yields $\varepsilon_{\text{long}} \approx 2\lceil s/256 \rceil \cdot 2^{-61} + 0.64 \cdot 2^{-56} < \lceil s/4096 \rceil \cdot 2^{-55}$.

The short input hash is independent of the polynomial hash multiplier, and can thus be seen as a degree-0 polynomial. Any collision between short and long inputs is subject to the same $\varepsilon_{\text{long}}$ collision bound. The total collision probability for strings of s bytes or fewer (with expectation taken over the generated key) thus adds up to $\varepsilon < \lceil s/4096 \rceil \cdot 2^{-55}$.

```
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-optimise 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 new block compressed values are fed to an independent polynomial hash, so we obtain a combined collision probability less than 2^{-70} for inputs up to 1 GB.

Acknowledgements

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

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`.

Change log

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"

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

2020-09-13: use an invertible xor-rot finalizer, instead of multiply xorshift.

2020-09-20: mix 9-16 byte inputs with `NH` instead of `PH`, before passing to the same polynomial hash and finalisation.

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.