UMASH: a fast almost universal 64-bit string hash

Paul Khuong, Backtrace I/O

2020-08-27

UMASH is a string hash function with throughput—22 GB/s on a 2.5 GHz Xeon 8175M—and latency—9-22 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 (after expanding each seed to a full 320-byte key for the latter).

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, we recommend the use of a stream cipher like Salsa20 to expand short seeds into full UMASH keys.

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 $\varepsilon < \lceil s/2048 \rceil \cdot 2^{-56}$, as long as the key is generated 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.

Overview of the UMASH hash function

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 `PH`, its equivalent in 128-bit carry-less arithmetic. This construction has been re-derived and baptised multiple times, recently as `CLNH` by Lemire and Kaser; Bernstein dates it back to 1968.

UMASH splits an input string into a sequence of blocks $\mathbf{M}$; the first-level `PH` hash further decomposes each block $\mathbf{M}_i$ of 256 bytes into m, a sequence of 64-bit integers. 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 (two 64-bit integers).

Note that, while the analysis below assumes a modulus of $M_{61} = 2^{61} - 1$ for the polynomial hash, the implementation actually works in $2^{64} - 8 = 8 \cdot (2^{61} - 1)$. 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 `PH` family with word size $w = 64$ bits and a block size of 256 bytes. The `PH` family of function is parameterised on k, an array of randomly chosen w-bit words, and implements the following sum in $2w$-bit carry-less arithmetic, for even input size $|m| \leq |k|$:

$$\texttt{PH}_k(m) = \bigoplus_{i=0}^{|m|/2-1} (m_{2i} \oplus k_{2i}) \cdot (m_{2i+1} \oplus k_{2i+1}).$$

For notational convenience, we will assume that the result of `PH` is recast from a bitvector to a $2w$-bit unsigned integer. Note how the ring of polynomials over GF(2), while similar to GF(2^{2w}), does not ever reduce the result of a multiplication: we only multiply w-bit values, so the final result always fits in $2w$ bits.

When the key words k_i are uniformly chosen from $[0, 2^w)$ at random, the probability that two different blocks m and m' yield the same $2w$-bit `PH` hash value is at most 2^{-w}. In UMASH, `PH` works with $|k| = 32$ words of 64 bit each, yielding a collision probability of 2^{-64}.

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

Let $n = |\mathbf{M}|$ be the number of `PH` blocks in the input, and y_j, for $j < d = 2n$, be the stream of alternating low ($\texttt{PH}_k(\mathbf{M}_{j/2}) \bmod 2^{64}$) and high ($\lfloor \texttt{PH}_k(\mathbf{M}_{(j-1)/2})/2^{64} \rfloor$) 64-bit halves from `PH`'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 `PH` blocks.

Two streams of at most d half-values y and y' that differ in at least one place will evaluate to the same hash $CW_f(y) = CW_f(y')$ with probability at most

$d/(2^{61}-1)$, as long as the multiplier f is uniformly chosen at random. In practice, we will forbid the weak multiplier 0, bringing the collision probability to $\varepsilon_{\mathbb{F}} < d/(2^{61}-2)\ldots$ pretty much the same thing.

We will also simplify the software implementation by actually evaluating that polynomial in the ring $\mathbb{Z}/(2^{64}-8)\mathbb{Z}$, which contains $\mathbb{F}$ as a subfield: $2^{64}-8 = 8\cdot(2^{61}-1)$. This cannot worsen the collision probability, since we could always reduce the result mod $2^{61}-1$ after the fact.

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?

Strings of 8 or fewer bytes are converted to 64-bit integers and passed to a mixing routine based on SplitMix64, with the addition of a secret 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.

Longer strings of 9 or more bytes feed the result of `PH` to the polynomial hash in $\mathbb{F} = \mathbb{Z}/(2^{61}-1)\mathbb{Z}$. When the input is of medium size $s \in [9, 16)$, it is expanded to 16 bytes by redundantly reading the first and last 8 bytes. In any case, the small modulus means polynomial hashing disregards slightly more than 6 bits off each `PH` output. This conservatively yields a collision probability less than 2^{-57} from `PH`; we will prevent extension attacks by modifying `PH`'s output with the block's original byte length.

That's quickly dominated by the collision probability for the polynomial hash, $\varepsilon_{\mathbb{F}} < d/(2^{-61}-2)$, where $d = 2\lceil s/256\rceil$. The worst-case collision probability is thus safely less than $\lceil s/2048\rceil \cdot 2^{-56}$. That's still universal with collision probability $\varepsilon_{\text{long}} < 2^{-40}$ for strings of 128 MB or less.

Applications that need stronger guarantees should compute two independent UMASH values (for a total of 128 bits), while recycling most of the `PH` key array with a Toeplitz extension. A C implementation that simply fuses the `PH` inner loops achieves a throughput of 10 GB/s and short-input latency of 10-28 ns on a 2.5 GHz Xeon 8175M, more than twice as fast as classic options like hardware-accelerated SHA-256, SipHash-2-4, or even SipHash-1-3.

Mapping from `NH` to `PH`

The design of VHASH relies on three important properties in `NH`:

1. `NH` is 2^{-64}-almost-universal. So is `PH`.
2. `NH` is actually 2^{-64}-almost-Δ-universal. `PH` is 2^{-64}-almost-XOR-universal.
3. Independent `NH`s can be computed by reusing key material with a "Toeplitz extension." `PH` offers the same capability: we can adapt Krovetz's proof to work with almost-XOR-universality.

As long as we remember to replace modular additions with bitwise `xor`, UMASH can easily steal small implementation tricks from VHASH.

Reference UMASH implementation in Python

```
from collections import namedtuple
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 `PH` words of 64 bit each.

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 `PH` keys avoid repeated values in order to protects against known weaknesses in the extremely similar `NH`.

```
UmashKey = namedtuple("UmashKey", ["poly", "ph"])


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)
    ph = []
    for _ in range(2 * BLOCK_SIZE):
        u64 = None
        while u64 is None or u64 in ph:
            u64 = random.getrandbits(64)
        ph.append(u64)
    return UmashKey(poly, ph)
```

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 `PH` key is injected in the process; while each step is trivial to invert, the additional `xor` is only invertible if we know the random `PH` 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 use a Toeplitz extension to generate a second `PH` key, collisions for the first and second keys 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 shift constant $S > 0$ (we use $S = 4$).

```
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 ‘PH‘ 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 `PH` 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 remembering the original byte size; this logic would also work on 8-byte inputs, but results in slower hashes for that common size. We then pass that block to a single eround of the `NH` compression function, and feed the result to the polynomial hash. Passing the original byte size down to the polynomial hash lets us defend against extension attacks.

We use `NH` for inputs of 9 to 16 bytes 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 `PH`. 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 with each block, for the outer polynomial hash to ingest.

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 last `PH` output before passing it to polynomial hashing. Since only the last block may span fewer than 256 bytes, this is equivalent to `xor`ing $|\mathbf{M}_i| \bmod 256$, a block's size modulo the maximum block size, to each block's PH output.

```
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 `PH` function defined by the `key`. This yields a stream of 128-bit `PH` outputs.

As a special case, we shuffle inputs of 9 to 16 bytes with a function from the lower-latency `NH` family. Although it doesn't seem that useful to compress 16 bytes to 16 bytes, `NH`'s output offers guarantees very similar to that of `PH`.

We propagate a `seed` argument all the way to the individual `PH` 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.

We protect against extension attacks by `xor`ing each block's `PH` value with that block's byte size modulo 256 (the maximum byte size), like VHASH does, but in GR(2^{128}). When two streams of blocks differ, the `xor` does not affect the collision probability: the `PH` bound already evaluates the probability that the `xor` of two compressed values matches any specific value (almost-XOR-universality). When the two streams of blocks are identical due to extension, the size of the last block differs, and the resulting compressed values definitely differ.

Similarly, in the `NH` medium input case, we add the block size modulo 256 to the high half of the `NH` result.

```
def ph_compress_one_block(key, seed, block, block_size):
    """Applies the ‘PH‘ hash to compress a block of up to 256 bytes."""
    # Seed goes to the low half to avoid shuffling in SIMD implementations
    acc = seed % W
    # Only the last block may be partial; every other block will
    # always find ‘increment == 0‘.
    increment = block_size % (CHUNK_SIZE * BLOCK_SIZE)
    for i, chunk in enumerate(block):
        ka = key[2 * i]
        kb = key[2 * i + 1]
        xa, xb = struct.unpack("<QQ", chunk)
        acc ^= gfmul(xa ^ ka, xb ^ kb)
    return acc ^ increment


def nh_compress_one_medium_block(key, seed, block, block_size):
    """Applies the ‘NH‘ hash to shuffle a block of 16 bytes."""
    assert block_size <= 16
    increment = block_size % (CHUNK_SIZE * BLOCK_SIZE)
    # Seed and block size go in the high half to avoid carries.
    acc = ((seed + increment) % W) * W
    for i, chunk in enumerate(block):
        ka = key[2 * i]
        kb = key[2 * i + 1]
        xa, xb = struct.unpack("<QQ", chunk)
        xa = (xa + ka) % W
        xb = (xb + kb) % W
        acc += xa * xb
    # Mix the low half into the high bits
    acc ^= (acc % W) * W
    acc %= W ** 2
    return acc


def ph_compress(key, seed, blocks):
    """Applies the ‘PH‘ compression function to each block; generates
    a stream of compressed values"""
    first_block = True
    for block, block_size in blocks:
        if first_block and block_size <= 16:
            yield nh_compress_one_medium_block(key, seed, block, block_size)
        else:
            yield ph_compress_one_block(key, seed, block, block_size)
        first_block = False
```

PH 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 $\lceil 2^{64}/|\mathbb{F}| \rceil^2 \cdot 2^{-64} < 2^{-57}$. 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 PH, but having the last step be a multiplication improves distribution.

```
def poly_reduce(multiplier, input_size, compressed_values):
    """Updates a polynomial hash with the ‘PH‘ 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 always maps 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 `PH`. 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 $\mathbb{F}^2$, $\varepsilon_{\texttt{PH}} < 9^2 \cdot 2^{-64} \approx 0.64 \cdot 2^{-57}$, 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 $\varepsilon_{\text{long}} \approx 2\lceil s/256 \rceil \cdot 2^{-61} + 0.64 \cdot 2^{-57} < \lceil s/2048 \rceil \cdot 2^{-56}$.

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 $\varepsilon < \lceil s/2048 \rceil \cdot 2^{-56}$.

```
def umash_long(key, seed, buf):
    assert len(buf) >= CHUNK_SIZE / 2
    blocks = blockify_chunks(chunk_bytes(buf))
    ph_values = ph_compress(key.ph, seed, blocks)
    poly_acc = poly_reduce(key.poly, len(buf), ph_values)
    return finalize(poly_acc)


def umash(key, seed, buf):
    if len(buf) <= CHUNK_SIZE / 2:
        return umash_short(key.ph, seed, buf)
    return umash_long(key, seed, buf)
```

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 `PH` loop early, 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` 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 the `PH` loop body 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.

What if we need stronger guarantees?

We can reuse most of the `PH` key with a Toeplitz extension. We skip four `PH` parameters (two suffice for correctness), and obtain a second independent UMASH with five additional parameters (one multiplier, and four `PH` values) in the key; it's also easy to merge the two `PH` compression loops, especially for incremental hashing.

When the input is 8 bytes or shorter, we `xor` in a single length-dependent value from the `PH` key. Inputs of the same length never collide, so a Toeplitz extension offers independent collision probabilities. Two inputs of different length collide when $k_s \oplus k_{s'}$ equals one unlucky value; the same expression in the Toeplitz-extended key is $k_{s+4} \oplus k_{s'+4}$, with distribution independent of $k_s \oplus k_{s'}$.

Combining two UMASHes squares the collision probability. The resulting probability, $\varepsilon^2 < \lceil s/2048 \rceil^2 \cdot 2^{-112}$ is easily comparable to the uncorrectable DRAM error rates reported by Facebook: 0.03% per month $\approx 2^{-66}$ per CPU cycle.

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

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.