Network Working Group
Internet-Draft
Intended status: Informational
Expires: 14 April 2026

Y. Collet
Meta
S. Josefsson, Ed.

11 October 2025

xxHash fast digest algorithm
draft-josefsson-xxhash-00

Abstract

Description of the xxHash fast digest algorithm.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-josefsson-xxhash/.

Source for this draft and an issue tracker can be found at https://gitlab.com/jas/ietf-xxhash.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 14 April 2026.

Copyright Notice

Table of Contents

1. Introduction

This document describes the xxHash digest algorithm for both 32-bit and 64-bit variants, named XXH32 and XXH64. The algorithm takes an input a message of arbitrary length and an optional seed value, then produces an output of 32 or 64-bit as "fingerprint" or "digest".

xxHash is primarily designed for speed. It is labeled non-cryptographic, and is not meant to avoid intentional collisions (same digest for 2 different messages), or to prevent producing a message with a predefined digest.

XXH32 is designed to be fast on 32-bit machines. XXH64 is designed to be fast on 64-bit machines. Both variants produce different output. However, a given variant shall produce exactly the same output, irrespective of the cpu / os used. In particular, the result remains identical whatever the endianness and width of the cpu is.

1.1. Operation notations

All operations are performed modulo {32,64} bits. Arithmetic overflows are expected. XXH32 uses 32-bit modular operations. XXH64 and XXH3 use 64-bit modular operations. When an operation ingests input or secret as multi-bytes values, it reads it using little-endian convention.

* +: denotes modular addition

* -: denotes modular subtraction

* *: denotes modular multiplication

- *Exception:* In XXH3, if it is in the form (u128)x * (u128)y, it denotes 64-bit by 64-bit normal multiplication into a full 128-bit result.

* X <<< s: denotes the value obtained by circularly shifting (rotating) X left by s bit positions.

* X >> s: denotes the value obtained by shifting X right by s bit positions. Upper s bits become 0.

* X << s: denotes the value obtained by shifting X left by s bit positions. Lower s bits become 0.

* X xor Y: denotes the bit-wise XOR of X and Y (same width).

* X | Y: denotes the bit-wise OR of X and Y (same width).

* ~X: denotes the bit-wise negation of X.

2. XXH32 Algorithm Description

2.1. Overview

We begin by supposing that we have a message of any length L as input, and that we wish to find its digest. Here L is an arbitrary nonnegative integer; L may be zero. The following steps are performed to compute the digest of the message.

The algorithm collect and transform input in _stripes_ of 16 bytes. The transforms are stored inside 4 "accumulators", each one storing an unsigned 32-bit value. Each accumulator can be processed independently in parallel, speeding up processing for cpu with multiple execution units.

The algorithm uses 32-bits addition, multiplication, rotate, shift and xor operations. Many operations require some 32-bits prime number constants, all defined below:

```
  // 0b10011110001101110111100110110001
  static const u32 PRIME32_1 = 0x9E3779B1U;
  // 0b10000101111010111100101001110111
  static const u32 PRIME32_2 = 0x85EBCA77U;
  // 0b11000010101100101010111000111101
  static const u32 PRIME32_3 = 0xC2B2AE3DU;
  // 0b00100111111010100111010110010111
  static const u32 PRIME32_4 = 0x27D4EB2FU;
  // 0b00010110010101100110011110110001
  static const u32 PRIME32_5 = 0x165667B1U;
```

These constants are prime numbers, and feature a good mix of bits 1 and 0, neither too regular, nor too dissymmetric. These properties help dispersion capabilities.

2.1.1. Step 1. Initialize internal accumulators

Each accumulator gets an initial value based on optional seed input. Since the seed is optional, it can be 0.

```
u32 acc1 = seed + PRIME32_1 + PRIME32_2;
u32 acc2 = seed + PRIME32_2;
u32 acc3 = seed + 0;
u32 acc4 = seed - PRIME32_1;
```

2.1.1.1. Special case: input is less than 16 bytes

When the input is too small (< 16 bytes), the algorithm will not process any stripes. Consequently, it will not make use of parallel accumulators.

In this case, a simplified initialization is performed, using a single accumulator:

```
u32 acc  = seed + PRIME32_5;
```

The algorithm then proceeds directly to step 4.

2.1.2. Step 2. Process stripes

A stripe is a contiguous segment of 16 bytes. It is evenly divided into 4 _lanes_, of 4 bytes each. The first lane is used to update accumulator 1, the second lane is used to update accumulator 2, and so on.

Each lane read its associated 32-bit value using *little-endian* convention.

For each {lane, accumulator}, the update process is called a _round_, and applies the following formula:

```
accN = accN + (laneN * PRIME32_2);
accN = accN <<< 13;
accN = accN * PRIME32_1;
```

This shuffles the bits so that any bit from input _lane_ impacts several bits in output _accumulator_. All operations are performed modulo 2^32.

Input is consumed one full stripe at a time. Step 2 is looped as many times as necessary to consume the whole input, except for the last remaining bytes which cannot form a stripe (< 16 bytes). When that happens, move to step 3.

2.1.3. Step 3. Accumulator convergence

All 4 lane accumulators from the previous steps are merged to produce a single remaining accumulator of the same width (32-bit). The associated formula is as follows:

```
acc = (acc1 <<< 1) + (acc2 <<< 7) + (acc3 <<< 12) + (acc4 <<< 18);
```

2.1.4. Step 4. Add input length

The input total length is presumed known at this stage. This step is just about adding the length to accumulator, so that it participates to final mixing.

```
acc = acc + (u32)inputLength;
```

Note that, if input length is so large that it requires more than 32-bits, only the lower 32-bits are added to the accumulator.

2.1.5. Step 5. Consume remaining input

There may be up to 15 bytes remaining to consume from the input. The final stage will digest them according to following pseudo-code:

```
while (remainingLength >= 4) {
    lane = read_32bit_little_endian(input_ptr);
    acc = acc + lane * PRIME32_3;
    acc = (acc <<< 17) * PRIME32_4;
    input_ptr += 4; remainingLength -= 4;
}

while (remainingLength >= 1) {
    lane = read_byte(input_ptr);
    acc = acc + lane * PRIME32_5;
    acc = (acc <<< 11) * PRIME32_1;
    input_ptr += 1; remainingLength -= 1;
}
```

This process ensures that all input bytes are present in the final mix.

2.1.6. Step 6. Final mix (avalanche)

The final mix ensures that all input bits have a chance to impact any bit in the output digest, resulting in an unbiased distribution. This is also called avalanche effect.

```
acc = acc xor (acc >> 15);
acc = acc * PRIME32_2;
acc = acc xor (acc >> 13);
acc = acc * PRIME32_3;
acc = acc xor (acc >> 16);
```

2.1.7. Step 7. Output

The XXH32() function produces an unsigned 32-bit value as output.

For systems which require to store and/or display the result in binary or hexadecimal format, the canonical format is defined to reproduce the same value as the natural decimal format, hence follows *big-endian* convention (most significant byte first).

3. XXH64 Algorithm Description

3.1. Overview

XXH64’s algorithm structure is very similar to XXH32 one. The major difference is that XXH64 uses 64-bit arithmetic, speeding up memory transfer for 64-bit compliant systems, but also relying on cpu capability to efficiently perform 64-bit operations.

The algorithm collects and transforms input in _stripes_ of 32 bytes. The transforms are stored inside 4 "accumulators", each one storing an unsigned 64-bit value. Each accumulator can be processed independently in parallel, speeding up processing for cpu with multiple execution units.

The algorithm uses 64-bit addition, multiplication, rotate, shift and xor operations. Many operations require some 64-bit prime number constants, all defined below:

```
// 0b1001111000110111011110011011000110000101111010111100101010000111
static const u64 PRIME64_1 = 0x9E3779B185EBCA87ULL;
// 0b1100000101011001010101110001111010010011111010100111010110100111
static const u64 PRIME64_2 = 0xC2B2AE3D27D4EB4FULL;
// 0b0001011001010110011001111011000110011110001101110111100111111001
static const u64 PRIME64_3 = 0x165667B19E3779F9ULL;
// 0b1000010111101011110010100111011111000010101100101010111001100011
static const u64 PRIME64_4 = 0x85EBCA77C2B2AE63ULL;
// 0b0010011111101010011101011001011110001011001010110011001111000101
static const u64 PRIME64_5 = 0x27D4EB2F165667C5ULL;
```

These constants are prime numbers, and feature a good mix of bits 1 and 0, neither too regular, nor too dissymmetric. These properties help dispersion capabilities.

3.1.1. Step 1. Initialize internal accumulators

Each accumulator gets an initial value based on optional seed input. Since the seed is optional, it can be 0.

```
u64 acc1 = seed + PRIME64_1 + PRIME64_2;
u64 acc2 = seed + PRIME64_2;
u64 acc3 = seed + 0;
u64 acc4 = seed - PRIME64_1;
```

3.1.1.1. Special case: input is less than 32 bytes

When the input is too small (< 32 bytes), the algorithm will not process any stripes. Consequently, it will not make use of parallel accumulators.

In this case, a simplified initialization is performed, using a single accumulator:

```
u64 acc  = seed + PRIME64_5;
```

The algorithm then proceeds directly to step 4.

3.1.2. Step 2. Process stripes

A stripe is a contiguous segment of 32 bytes. It is evenly divided into 4 _lanes_, of 8 bytes each. The first lane is used to update accumulator 1, the second lane is used to update accumulator 2, and so on.

Each lane read its associated 64-bit value using *little-endian* convention.

For each {lane, accumulator}, the update process is called a _round_, and applies the following formula:

```
round(accN,laneN):
  accN = accN + (laneN * PRIME64_2);
  accN = accN <<< 31;
  return accN * PRIME64_1;
```

This shuffles the bits so that any bit from input _lane_ impacts several bits in output _accumulator_. All operations are performed modulo 2^64.

Input is consumed one full stripe at a time. Step 2 is looped as many times as necessary to consume the whole input, except for the last remaining bytes which cannot form a stripe (< 32 bytes). When that happens, move to step 3.

3.1.3. Step 3. Accumulator convergence

All 4 lane accumulators from previous steps are merged to produce a single remaining accumulator of same width (64-bit). The associated formula is as follows.

Note that accumulator convergence is more complex than 32-bit variant, and requires to define another function called _mergeAccumulator()_:

```
mergeAccumulator(acc,accN):
  acc  = acc xor round(0, accN);
  acc  = acc * PRIME64_1;
  return acc + PRIME64_4;
```

which is then used in the convergence formula:

```
  acc = (acc1 <<< 1) + (acc2 <<< 7) + (acc3 <<< 12) + (acc4 <<< 18);
  acc = mergeAccumulator(acc, acc1);
  acc = mergeAccumulator(acc, acc2);
  acc = mergeAccumulator(acc, acc3);
  acc = mergeAccumulator(acc, acc4);
```

3.1.4. Step 4. Add input length

The input total length is presumed known at this stage. This step is just about adding the length to accumulator, so that it participates to final mixing.

```
  acc = acc + inputLength;
```

3.1.5. Step 5. Consume remaining input

There may be up to 31 bytes remaining to consume from the input. The final stage will digest them according to following pseudo-code:

```
while (remainingLength >= 8) {
    lane = read_64bit_little_endian(input_ptr);
    acc = acc xor round(0, lane);
    acc = (acc <<< 27) * PRIME64_1;
    acc = acc + PRIME64_4;
    input_ptr += 8; remainingLength -= 8;
}

if (remainingLength >= 4) {
    lane = read_32bit_little_endian(input_ptr);
    acc = acc xor (lane * PRIME64_1);
    acc = (acc <<< 23) * PRIME64_2;
    acc = acc + PRIME64_3;
    input_ptr += 4; remainingLength -= 4;
}

while (remainingLength >= 1) {
    lane = read_byte(input_ptr);
    acc = acc xor (lane * PRIME64_5);
    acc = (acc <<< 11) * PRIME64_1;
    input_ptr += 1; remainingLength -= 1;
}
```

This process ensures that all input bytes are present in the final mix.

3.1.6. Step 6. Final mix (avalanche)

The final mix ensures that all input bits have a chance to impact any bit in the output digest, resulting in an unbiased distribution. This is also called avalanche effect.

```
acc = acc xor (acc >> 33);
acc = acc * PRIME64_2;
acc = acc xor (acc >> 29);
acc = acc * PRIME64_3;
acc = acc xor (acc >> 32);
```

3.1.7. Step 7. Output

The XXH64() function produces an unsigned 64-bit value as output.

For systems which require to store and/or display the result in binary or hexadecimal format, the canonical format is defined to reproduce the same value as the natural decimal format, hence follows *big-endian* convention (most significant byte first).

4. XXH3 Algorithm Overview

XXH3 comes in two different versions: XXH3-64 and XXH3-128 (or XXH128), producing 64 and 128 bits of output, respectively.

XXH3 uses different algorithms for small (0-16 bytes), medium (17-240 bytes), and large (241+ bytes) inputs. The algorithms for small and medium inputs are optimized for performance. The three algorithms are described in the following sections.

Many operations require some 64-bit prime number constants, which are mostly the same constants used in XXH32 and XXH64, all defined below:

```
// 0b10011110001101110111100110110001
static const u64 PRIME32_1 = 0x9E3779B1U;
// 0b10000101111010111100101001110111
static const u64 PRIME32_2 = 0x85EBCA77U;
// 0b11000010101100101010111000111101
static const u64 PRIME32_3 = 0xC2B2AE3DU;
// 0b1001111000110111011110011011000110000101111010111100101010000111
static const u64 PRIME64_1 = 0x9E3779B185EBCA87ULL;
// 0b1100001010110010101011100011110100100111110101001110101101001111
static const u64 PRIME64_2 = 0xC2B2AE3D27D4EB4FULL;
// 0b0001011001010110011001111011000110011110001101110111100111111001
static const u64 PRIME64_3 = 0x165667B19E3779F9ULL;
// 0b1000010111101011110010100111011111000010101100101010111001100011
static const u64 PRIME64_4 = 0x85EBCA77C2B2AE63ULL;
// 0b0010011111010100111010110010111100010110010101100110011111000101
static const u64 PRIME64_5 = 0x27D4EB2F165667C5ULL;
// 0b0001011001010110011001111001000110011110001101110111100111111001
static const u64 PRIME_MX1 = 0x165667919E3779F9ULL;
// 0b1001111110110010000111000110010100011110100110001101111100100101
static const u64 PRIME_MX2 = 0x9FB21C651E98DF25ULL;
```

The XXH3_64bits() function produces an unsigned 64-bit value. The XXH3_128bits() function produces a XXH128_hash_t struct containing low64 and high64 - the lower and higher 64-bit half values of the result, respectively.

For systems requiring storing and/or displaying the result in binary or hexadecimal format, the canonical format is defined to reproduce the same value as the natural decimal format, hence following *big-endian* convention (most significant byte first).

4.1. Seed and Secret

XXH3 provides seeded hashing by introducing two configurable constants used in the hashing process: the seed and the secret. The seed is an unsigned 64-bit value, and the secret is an array of bytes that is at least 136 bytes in size. The default seed is 0, and the default secret is the following 192-byte value:

```
static const u8 defaultSecret[192] = {
  0xb8, 0xfe, 0x6c, 0x39, 0x23, 0xa4, 0x4b, 0xbe,
  0x7c, 0x01, 0x81, 0x2c, 0xf7, 0x21, 0xad, 0x1c,
  0xde, 0xd4, 0x6d, 0xe9, 0x83, 0x90, 0x97, 0xdb,
  0x72, 0x40, 0xa4, 0xa4, 0xb7, 0xb3, 0x67, 0x1f,
  0xcb, 0x79, 0xe6, 0x4e, 0xcc, 0xc0, 0xe5, 0x78,
  0x82, 0x5a, 0xd0, 0x7d, 0xcc, 0xff, 0x72, 0x21,
  0xb8, 0x08, 0x46, 0x74, 0xf7, 0x43, 0x24, 0x8e,
  0xe0, 0x35, 0x90, 0xe6, 0x81, 0x3a, 0x26, 0x4c,
  0x3c, 0x28, 0x52, 0xbb, 0x91, 0xc3, 0x00, 0xcb,
  0x88, 0xd0, 0x65, 0x8b, 0x1b, 0x53, 0x2e, 0xa3,
  0x71, 0x64, 0x48, 0x97, 0xa2, 0x0d, 0xf9, 0x4e,
  0x38, 0x19, 0xef, 0x46, 0xa9, 0xde, 0xac, 0xd8,
  0xa8, 0xfa, 0x76, 0x3f, 0xe3, 0x9c, 0x34, 0x3f,
  0xf9, 0xdc, 0xbb, 0xc7, 0xc7, 0x0b, 0x4f, 0x1d,
  0x8a, 0x51, 0xe0, 0x4b, 0xcd, 0xb4, 0x59, 0x31,
  0xc8, 0x9f, 0x7e, 0xc9, 0xd9, 0x78, 0x73, 0x64,
  0xea, 0xc5, 0xac, 0x83, 0x34, 0xd3, 0xeb, 0xc3,
  0xc5, 0x81, 0xa0, 0xff, 0xfa, 0x13, 0x63, 0xeb,
  0x17, 0x0d, 0xdd, 0x51, 0xb7, 0xf0, 0xda, 0x49,
  0xd3, 0x16, 0x55, 0x26, 0x29, 0xd4, 0x68, 0x9e,
  0x2b, 0x16, 0xbe, 0x58, 0x7d, 0x47, 0xa1, 0xfc,
  0x8f, 0xf8, 0xb8, 0xd1, 0x7a, 0xd0, 0x31, 0xce,
  0x45, 0xcb, 0x3a, 0x8f, 0x95, 0x16, 0x04, 0x28,
  0xaf, 0xd7, 0xfb, 0xca, 0xbb, 0x4b, 0x40, 0x7e,
};
```

The seed and the secret can be optionally specified using the *_withSecret and *_withSeed versions of the hash function.

The seed and the secret cannot be specified simultaneously (*_withSecretAndSeed is actually *_withSeed for short and medium inputs <= 240 bytes, and *_withSecret for large inputs). When one is specified, the other one uses the default value. There is one exception though: when input is large (> 240 bytes) and a seed is given, a secret is derived from the seed value and the default secret using the following procedure:

```
deriveSecret(u64 seed):
  u64 derivedSecret[24] = defaultSecret[0:192];
  for (i = 0; i < 12; i++) {
    derivedSecret[i*2] += seed;
    derivedSecret[i*2+1] -= seed;
  }
  return derivedSecret; // convert to u8[192] (little-endian)
```

The derivation treats the secrets as 24 64-bit values. In XXH3 algorithms, the secret is always read similarly by treating a contiguous segment of the array as one or more 32-bit or 64-bit values. *The secret values are always read using little-endian convention*.

4.1.1. Final Mixing Step (avalanche)

To make sure that all input bits have a chance to impact any bit in the output digest (avalanche effect), the final step of the XXH3 algorithm is usually one of the two fixed operations that mix the bits in a 64-bit value. These operations are denoted avalanche() and avalanche_XXH64() in the following XXH3 description.

```
avalanche(u64 x):
  x = x xor (x >> 37);
  x = x * PRIME_MX1;
  x = x xor (x >> 32);
  return x;

avalanche_XXH64(u64 x):
  x = x xor (x >> 33);
  x = x * PRIME64_2;
  x = x xor (x >> 29);
  x = x * PRIME64_3;
  x = x xor (x >> 32);
  return x;
```

5. XXH3 Algorithm Description (for small inputs)

The algorithm for small inputs (0-16 bytes of input) is further divided into 4 cases: empty, 1-3 bytes, 4-8 bytes, and 9-16 bytes of input.

The algorithm uses byte-swap operations. The byte-swap operation reverses the byte order in a 32-bit or 64-bit value. It is denoted bswap32 and bswap64 for its 32-bit and 64-bit versions, respectively.

5.1. Empty input

The hash of empty input is calculated from the seed and a segment of the secret:

```
XXH3_64_empty():
  u64 secretWords[2] = secret[56:72];
  return avalanche_XXH64(seed xor secretWords[0] xor secretWords[1]);

XXH3_128_empty():
  u64 secretWords[4] = secret[64:96];
  return {avalanche_XXH64(seed xor secretWords[0] xor secretWords[1]), // lower half
          avalanche_XXH64(seed xor secretWords[2] xor secretWords[3])}; // higher half
```

5.1.1. 1-3 bytes of input

The algorithm starts from a single 32-bit value combining the input bytes and its length:

```
u32 combined = (u32)input[inputLength-1] | ((u32)inputLength << 8) |
               ((u32)input[0] << 16) | ((u32)input[inputLength>>1] << 24);
// LSB          8       16           24                     MSB
//  | last byte | length | first byte | middle-or-last byte |
```

Then the final output is calculated from the value and the first 8 bytes (XXH3-64) or 16 bytes (XXH3-128) of the secret to produce the final result. The secret here is read as 32-bit values instead of the usual 64-bit values.

```
XXH3_64_1to3():
  u32 secretWords[2] = secret[0:8];
  u64 value = ((u64)(secretWords[0] xor secretWords[1]) + seed) xor (u64)combined;
  return avalanche_XXH64(value);

XXH3_128_1to3():
  u32 secretWords[4] = secret[0:16];
  u64 low = ((u64)(secretWords[0] xor secretWords[1]) + seed) xor (u64)combined;
  u64 high = ((u64)(secretWords[2] xor secretWords[3]) - seed) xor (u64)(bswap32(combi
ned) <<< 13);
  // note that the bswap32(combined) <<< 13 above is 32-bit rotate
  return {avalanche_XXH64(low), // lower half
          avalanche_XXH64(high)}; // higher half
```

Note that the XXH3-64 result is the lower half of XXH3-128 result.

5.1.2. 4-8 bytes of input

The algorithm starts from reading the first and last 4 bytes of the input as little-endian 32-bit values, and a modified seed:

```
u32 inputFirst = input[0:4];
u32 inputLast = input[inputLength-4:inputLength];
u64 modifiedSeed = seed xor ((u64)bswap32((u32)lowerHalf(seed)) << 32);
```

Again, these values are combined with a segment of the secret to produce the final value.

```
XXH3_64_4to8():
  u64 secretWords[2] = secret[8:24];
  u64 combined = (u64)inputLast | ((u64)inputFirst << 32);
  u64 value = ((secretWords[0] xor secretWords[1]) - modifiedSeed) xor combined;
  value = value xor (value <<< 49) xor (value <<< 24);
  value = value * PRIME_MX2;
  value = value xor ((value >> 35) + inputLength);
  value = value * PRIME_MX2;
  value = value xor (value >> 28);
  return value;

XXH3_128_4to8():
  u64 secretWords[2] = secret[16:32];
  u64 combined = (u64)inputFirst | ((u64)inputLast << 32);
  u64 value = ((secretWords[0] xor secretWords[1]) + modifiedSeed) xor combined;
  u128 mulResult = (u128)value * (u128)(PRIME64_1 + (inputLength << 2));
  u64 high = higherHalf(mulResult); // mulResult >> 64
  u64 low = lowerHalf(mulResult); // mulResult & 0xFFFFFFFFFFFFFFFF
  high = high + (low << 1);
  low = low xor (high >> 3);
  low = low xor (low >> 35);
  low = low * PRIME_MX2;
  low = low xor (low >> 28);
  high = avalanche(high);
  return {low, high};
```

5.1.3. 9-16 bytes of input

The algorithm starts from reading the first and last 8 bytes of the input as little-endian 64-bit values:

```
u64 inputFirst = input[0:8];
u64 inputLast = input[inputLength-8:inputLength];
```

Once again, these values are combined with a segment of the secret to produce the final value.

```
   XXH3_64_9to16():
     u64 secretWords[4] = secret[24:56];
     u64 low = ((secretWords[0] xor secretWords[1]) + seed) xor inputFirst;
     u64 high = ((secretWords[2] xor secretWords[3]) - seed) xor inputLast;
     u128 mulResult = (u128)low * (u128)high;
     u64 value = inputLength + bswap64(low) + high + (u64)(lowerHalf(mulResult) xor highe
rHalf(mulResult));
     return avalanche(value);

   XXH3_128_9to16():
     u64 secretWords[4] = secret[32:64];
     u64 val1 = ((secretWords[0] xor secretWords[1]) - seed) xor inputFirst xor inputLast
;
     u64 val2 = ((secretWords[2] xor secretWords[3]) + seed) xor inputLast;
     u128 mulResult = (u128)val1 * (u128)PRIME64_1;
     u64 low = lowerHalf(mulResult) + ((u64)(inputLength - 1) << 54);
     u64 high = higherHalf(mulResult) + ((u64)higherHalf(val2) << 32) + (u64)lowerHalf(va
l2) * PRIME32_2;
     // the above line can also be simplified to higherHalf(mulResult) + val2 + (u64)lowe
rHalf(val2) * (PRIME32_2 - 1);
     low = low xor bswap64(high);
     // the following three lines are in fact a 128x64 -> 128 multiplication ({low,high}
= (u128){low,high} * PRIME64_2)
     u128 mulResult2 = (u128)low * (u128)PRIME64_2;
     low = lowerHalf(mulResult2);
     high = higherHalf(mulResult2) + high * PRIME64_2;
     return {avalanche(low), // lower half
             avalanche(high)}; // higher half
```

6. XXH3 Algorithm Description (for medium inputs)

This algorithm is used for medium inputs (17-240 bytes of input). Its internal hash state is stored inside 1 (XXH3-64) or 2 (XXH3-128) "accumulators", each storing an unsigned 64-bit value.

6.1. Step 1. Initialize internal accumulators

The accumulator(s) are initialized based on the input length.

```
// For XXH3-64
u64 acc = inputLength * PRIME64_1;

// For XXH3-128
u64 acc[2] = {inputLength * PRIME64_1, 0};
```

6.2. Step 2. Process the input

This step is further divided into two cases: one for 17-128 bytes of input, and one for 129-240 bytes of input.

6.2.1. Mixing operation

This step uses a mixing operation that mixes a 16-byte segment of data, a 16-byte segment of secret and the seed into a 64-bit value as a building block. This operation treat the segment of data and secret as little-endian 64-bit values.

```
mixStep(u8 data[16], size secretOffset, u64 seed):
  u64 dataWords[2] = data[0:16];
  u64 secretWords[2] = secret[secretOffset:secretOffset+16];
  u128 mulResult = (u128)(dataWords[0] xor (secretWords[0] + seed)) *
                   (u128)(dataWords[1] xor (secretWords[1] - seed));
  return lowerHalf(mulResult) xor higherHalf(mulResult);
```

The mixing operation is always invoked in groups of two in XXH3-128, where two 16-byte segments of data are mixed with a 32-byte segment of secret, and the accumulators are updated accordingly.

```
mixTwoChunks(u8 data1[16], u8 data2[16], size secretOffset, u64 seed):
  u64 dataWords1[2] = data1[0:16]; // again, little-endian conversion
  u64 dataWords2[2] = data2[0:16];
  acc[0] = acc[0] + mixStep(data1, secretOffset, seed);
  acc[1] = acc[1] + mixStep(data2, secretOffset + 16, seed);
  acc[0] = acc[0] xor (dataWords2[0] + dataWords2[1]);
  acc[1] = acc[1] xor (dataWords1[0] + dataWords1[1]);
```

The input is split into several 16-byte chunks and mixed, and the result is added to the accumulator(s).

6.2.2. 17-128 bytes of input

The input is read as _N_ 16-byte chunks starting from the beginning and _N_ chunks starting from the end, where _N_ is the smallest number that these 2_N_ chunks cover the whole input. These chunks are paired up and mixed, and the results are accumulated to the accumulator(s).

```
   // the loop variable ‘i‘ should be signed to avoid underflow in implementation
   processInput_XXH3_64_17to128():
     u64 numRounds = ((inputLength - 1) >> 5) + 1;
     for (i = numRounds - 1; i >= 0; i--) {
       size offsetStart = i*16;
       size offsetEnd = inputLength - i*16 - 16;
       acc += mixStep(input[offsetStart:offsetStart+16], i*32, seed);
       acc += mixStep(input[offsetEnd:offsetEnd+16], i*32+16, seed);
     }

   processInput_XXH3_128_17to128():
     u64 numRounds = ((inputLength - 1) >> 5) + 1;
     for (i = numRounds - 1; i >= 0; i--) {
       size offsetStart = i*16;
       size offsetEnd = inputLength - i*16 - 16;
       mixTwoChunks(input[offsetStart:offsetStart+16], input[offsetEnd:offsetEnd+16], i*3
2, seed);
     }
```

6.2.3. 129-240 bytes of input

The input is split into 16-byte (XXH3-64) or 32-byte (XXH3-128) chunks. The first 128 bytes are first mixed chunk by chunk, followed by an intermediate avalanche operation. Then the remaining full chunks are processed, and finally the last 16/32 bytes are treated as a chunk to process.

```
   processInput_XXH3_64_129to240():
     u64 numChunks = inputLength >> 4;
     for (i = 0; i < 8; i++) {
       acc += mixStep(input[i*16:i*16+16], i*16, seed);
     }
     acc = avalanche(acc);
     for (i = 8; i < numChunks; i++) {
       acc += mixStep(input[i*16:i*16+16], (i-8)*16 + 3, seed);
     }
     acc += mixStep(input[inputLength-16:inputLength], 119, seed);

   processInput_XXH3_128_129to240():
     u64 numChunks = inputLength >> 5;
     for (i = 0; i < 4; i++) {
       mixTwoChunks(input[i*32:i*32+16], input[i*32+16:i*32+32], i*32, seed);
     }
     acc[0] = avalanche(acc[0]);
     acc[1] = avalanche(acc[1]);
     for (i = 4; i < numChunks; i++) {
       mixTwoChunks(input[i*32:i*32+16], input[i*32+16:i*32+32], (i-4)*32 + 3, seed);
     }
     // note that the half-chunk order and the seed is different here
     mixTwoChunks(input[inputLength-16:inputLength], input[inputLength-32:inputLength-16]
, 103, (u64)0 - seed);
```

6.3. Step 3. Finalization

The final result is extracted from the accumulator(s).

```
   XXH3_64_17to240():
     return avalanche(acc);

   XXH3_128_17to240():
     u64 low = acc[0] + acc[1];
     u64 high = (acc[0] * PRIME64_1) + (acc[1] * PRIME64_4) + (((u64)inputLength - seed)
* PRIME64_2);
     return {avalanche(low), // lower half
             (u64)0 - avalanche(high)}; // higher half
```

7. XXH3 Algorithm Description (for large inputs)

This algorithm is used for inputs larger than 240 bytes. The internal hash state is stored inside 8 "accumulators", each one storing an unsigned 64-bit value.

7.1. Step 1. Initialize internal accumulators

The accumulators are initialized to fixed constants:

```
u64 acc[8] = {
  PRIME32_3, PRIME64_1, PRIME64_2, PRIME64_3,
  PRIME64_4, PRIME32_2, PRIME64_5, PRIME32_1};
```

7.2. Step 2. Process blocks

The input is consumed and processed one full block at a time. The size of the block depends on the length of the secret. Specifically, a block consists of several 64-byte stripes. The number of stripes per block is floor((secretLength-64)/8) . For the default 192-byte secret, there are 16 stripes in a block, and thus the block size is 1024 bytes.

```
secretLength = lengthInBytes(secret);    // default 192; at least 136
stripesPerBlock = (secretLength-64) / 8; // default 16; at least 9
blockSize = 64 * stripesPerBlock;        // default 1024; at least 576
```

The process of processing a full block is called a _round_. It consists of the following two sub-steps:

7.2.1. Step 2-1. Process stripes in the block

A stripe is evenly divided into 8 lanes, of 8 bytes each. In an accumulation step, one stripe and a 64-byte contiguous segment of the secret are used to update the accumulators. Each lane reads its associated 64-bit value using little-endian convention.

The accumulation step applies the following procedure:

```
accumulate(u64 stripe[8], size secretOffset):
  u64 secretWords[8] = secret[secretOffset:secretOffset+64];
  for (i = 0; i < 8; i++) {
    u64 value = stripe[i] xor secretWords[i];
    acc[i xor 1] = acc[i xor 1] + stripe[i];
    acc[i] = acc[i] + (u64)lowerHalf(value) * (u64)higherHalf(value);
                      // (value and 0xFFFFFFFF) * (value >> 32)
  }
```

The accumulation step is repeated for all stripes in a block, using different segments of the secret, starting from the first 64 bytes for the first stripe, and offset by 8 bytes for each following round:

```
round_accumulate(u8 block[blockSize]):
  for (n = 0; n < stripesPerBlock; n++) {
    u64 stripe[8] = block[n*64:n*64+64]; // 64 bytes = 8 u64s
    accumulate(stripe, n*8);
  }
```

7.2.2. Step 2-2. Scramble accumulators

After the accumulation steps are finished for all stripes in the block, the accumulators are scrambled using the last 64 bytes of the secret.

```
round_scramble():
  u64 secretWords[8] = secret[secretLength-64:secretLength];
  for (i = 0; i < 8; i++) {
    acc[i] = acc[i] xor (acc[i] >> 47);
    acc[i] = acc[i] xor secretWords[i];
    acc[i] = acc[i] * PRIME32_1;
  }
```

A round is thus a round_accumulate followed by a round_scramble:

```
round(u8 block[blockSize]):
  round_accumulate(block);
  round_scramble();
```

Step 2 is looped to consume the input until there are less than or equal to blockSize bytes of input left. Note that we leave the last block to the next step even if it is a full block.

7.2.3. Step 3. Process the last block and the last 64 bytes

Accumulation steps are run for the stripes in the last block, except for the last stripe (whether it is full or not). After that, run a final accumulation step by treating the last 64 bytes as a stripe. Note that the last 64 bytes might overlap with the second-to-last block.

```
// len is the size of the last block (1 <= len <= blockSize)
lastRound(u8 block[], size len, u64 lastStripe[8]):
  size nFullStripes = (len-1)/64;
  for (n = 0; n < nFullStripes; n++) {
    u64 stripe[8] = block[n*64:n*64+64];
    accumulate(stripe, n * 8);
  }
  accumulate(lastStripe, secretLength - 71);
```

7.3. Step 4. Finalization

In the finalization step, a merging procedure is used to extract a single 64-bit value from the accumulators, using an initial seed value and a 64-byte segment of the secret.

```
   finalMerge(u64 initValue, size secretOffset):
     u64 secretWords[8] = secret[secretOffset:secretOffset+64];
     u64 result = initValue;
     for (i = 0; i < 4; i++) {
       // 64-bit by 64-bit multiplication to 128-bit full result
       u128 mulResult = (u128)(acc[i*2] xor secretWords[i*2]) *
                        (u128)(acc[i*2+1] xor secretWords[i*2+1]);
       result = result + (lowerHalf(mulResult) xor higherHalf(mulResult));
                         // (mulResult and 0xFFFFFFFFFFFFFFFF) xor (mulResult >> 64)
     }
     return avalanche(result);
```

XXH3-128 runs the merging procedure twice for the two halves of the result, using different secret segments and different initial values derived from the total input length. The XXH3-64 result is just the lower half of the XXH3-128 result.

```
   XXH3_64_large():
     return finalMerge((u64)inputLength * PRIME64_1, 11);

   XXH3_128_large():
     return {finalMerge((u64)inputLength * PRIME64_1, 11), // lower half
             finalMerge(~((u64)inputLength * PRIME64_2), secretLength - 75)}; // higher h
alf
```

8. Performance considerations

The xxHash algorithms are simple and compact to implement. They provide a system independent "fingerprint" or digest of a message of arbitrary length.

The algorithm allows input to be streamed and processed in multiple steps. In such case, an internal buffer is needed to ensure data is presented to the algorithm in full stripes.

On 64-bit systems, the 64-bit variant XXH64 is generally faster to compute, so it is a recommended variant, even when only 32-bit are needed.

On 32-bit systems though, positions are reversed: XXH64 performance is reduced, due to its usage of 64-bit arithmetic. XXH32 becomes a faster variant.

Finally, when vector operations are possible, XXH3 is likely the faster variant.

9. Reference Implementation

A reference library written in C is available at https://www.xxhash.com [XXHASH]. The web page also links to multiple other implementations written in many different languages. It links to the github project page (https://github.com/Cyan4973/xxHash) where an issue board (https://github.com/Cyan4973/xxHash/issues) can be used for further public discussions on the topic.

10. IANA Considerations

None

11. Security Considerations

xxHash is not a cryptographic hash function, and has not been designed for use in any cryptographic setting.

Implementations has to follow best practices to avoid security concerns, and users needs to continously re-evaulate implementations for security vulnerabilities.

12. Notices

This document is derived from https://github.com/Cyan4973/xxHash/blob/dev/doc/xxhash_spec.md commit d66a9cb6f223b1f15cceebee39bc07ed0bd4ad3d with notice below, with all changes are tracked at https://gitlab.com/jas/ietf-xxhash using Git.

Version 0.2.0 (29/06/23)

Version changes

v0.2.0: added XXH3 specification, by Adrien Wu
v0.1.1: added a note on rationale for selection of constants
v0.1.0: initial release

13. Informative References

[XXHASH] "xxHash", n.d., <https://xxhash.com/>.

Authors' Addresses

Yann Collet
Meta
Email: cyan@meta.com

Simon Josefsson (editor)
Email: simon@josefsson.org