Automatic Synthesis of Specialized Hash Functions

Renato Hoffmann renato.hoffmann@edu.pucrs.br
Leonardo Faé leonardo.fae@edu.pucrs.br
Dalvan Griebler dalvan.griebler@pucrs.br
Xinliang David Li davidxl@google.com
Fernando Pereira fernando@dcc.ufmg.br

Goal

1. Generate code for hash functions

Goal

DCC

1. Generate code for hash functions
2. Code is specialized for the keys

Goal

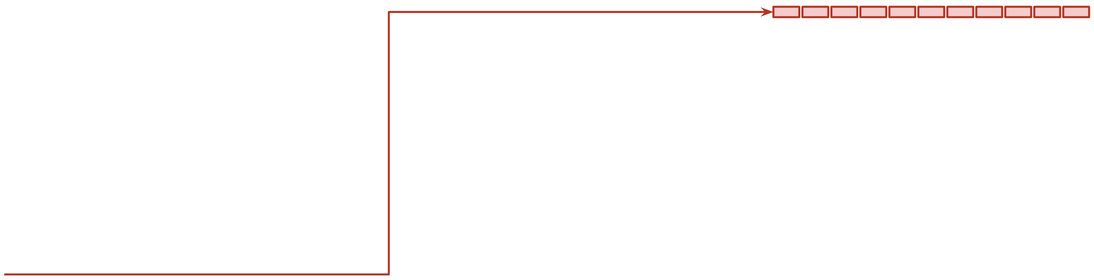
1. Generate code for hash functions
2. Code is specialized for the keys
    a. Constant length

Goal

1. Generate code for hash functions
2. Code is specialized for the keys
    a. Constant length

Examples: SSN

XXX-XX-XXXX



Eleven bytes

Goal

DCC

1. Generate code for hash functions
2. Code is specialized for the keys
    a. Constant length
    b. Constant subsequence

XXX-XX-XXXX

Goal

DCC

1. Generate code for hash functions
2. Code is specialized for the keys
   a. Constant length
   b. Constant subsequence

Examples: SSN

XXX-XX-XXXX

Constant '_'

Goal

DCC

1. Generate code for hash functions
2. Code is specialized for the keys
   a. Constant length
   b. Constant subsequence
   c. Constrained byte ranges

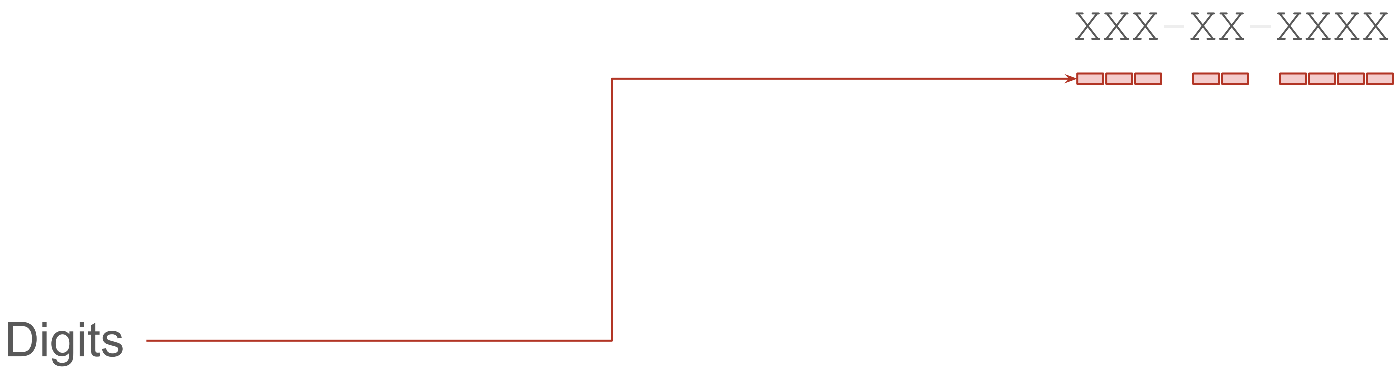
XXX-XX-XXXX

Goal

DCC

1. Generate code for hash functions
2. Code is specialized for the keys
   a. Constant length
   b. Constant subsequence
   c. Constrained byte ranges

Examples: SSN



Goal

1. Generate code for hash functions
2. Code is specialized for the keys
   a. Constant length
   b. Constant subsequence
   c. Constrained byte ranges
3. Specialized hash is faster

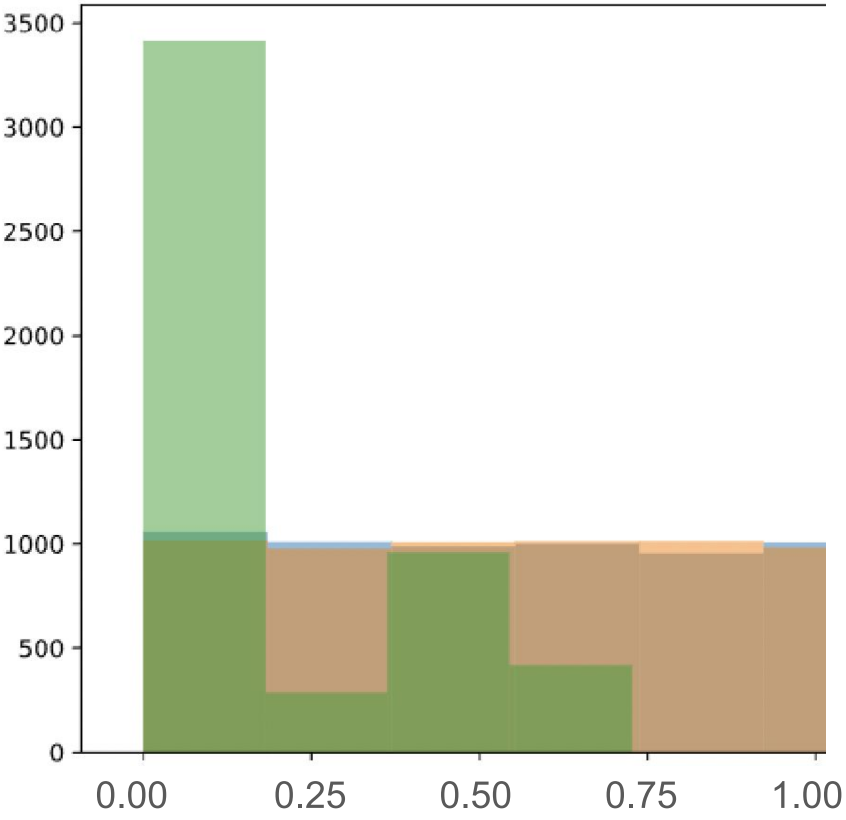
Goal

DCC

1. Generate code for hash functions
2. Code is specialized for the keys
    a. Constant length
    b. Constant subsequence
    c. Constrained byte ranges
3. Specialized hash is faster

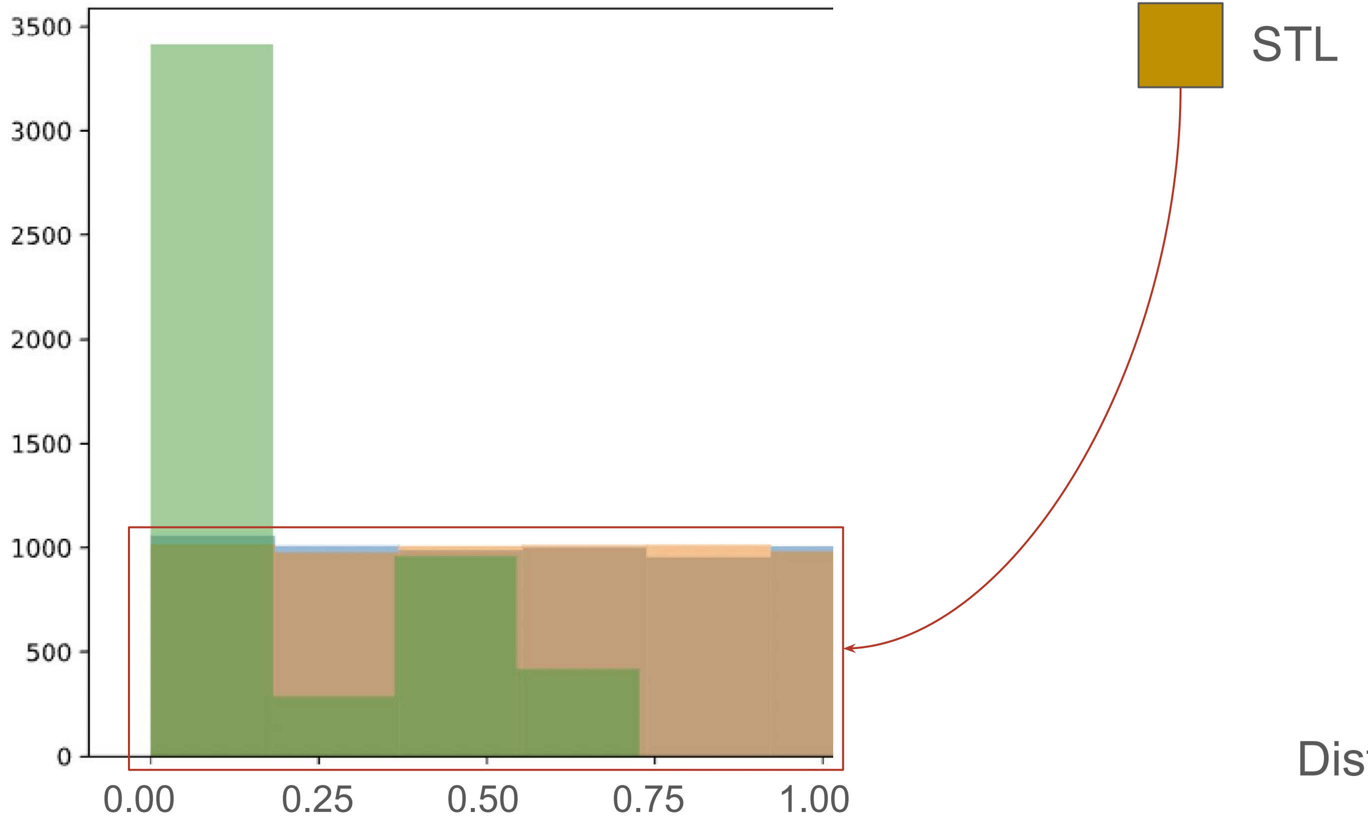
Distribution will be worse

Goal



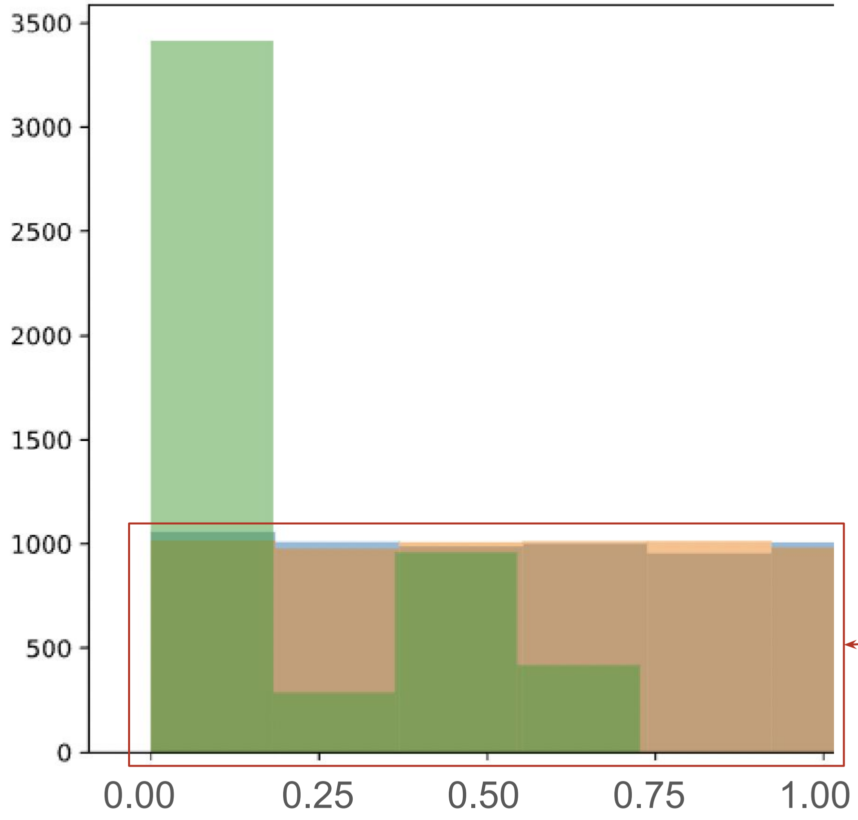
Distribution will be worse

Goal



Distribution will be worse

Goal

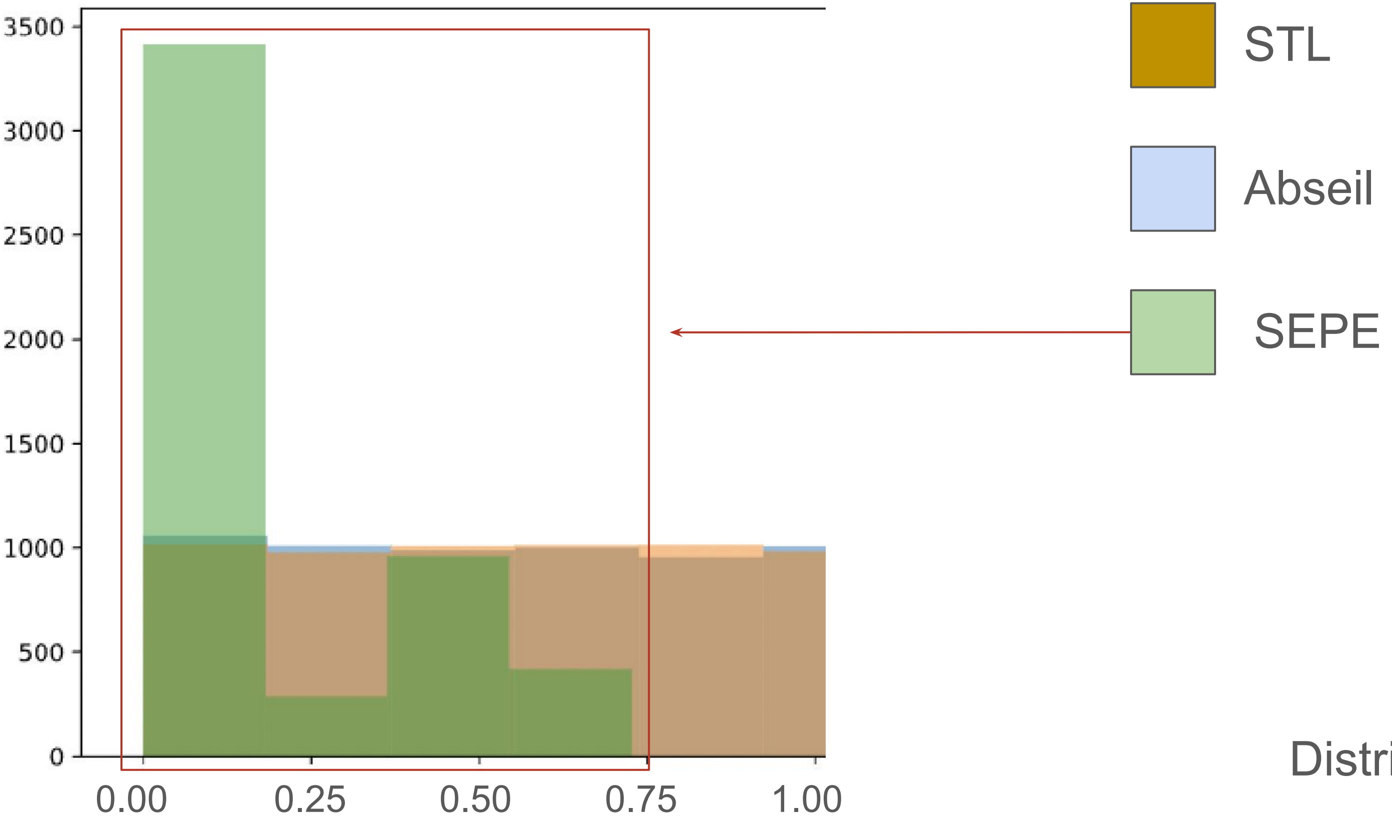


STL

Abseil

Distribution will be worse

Goal



Distribution will be worse

What is a Hash Function?

What is a Hash Function?

DCC

1. Byte sequence of arbitrary length → fixed-size value

What is a Hash Function?

1. Byte sequence of arbitrary length → fixed-size value
2. Deterministic algorithm

What is a Hash Function?

1. Byte sequence of arbitrary length → fixed-size value
2. Deterministic algorithm
3. Indexing, data retrieval, encryption, etc.

What is a Hash Function?

1. Byte sequence of arbitrary length → fixed-size value
2. Deterministic algorithm
3. Indexing, data retrieval, ~~encryption~~, etc.

Goals

What is a **General** Hash Function?

DCC

```
size_t _hash(void* ptr, size_t len, size_t seed) {
  size_t mul = (0xc6a4a793UL << 32UL) + 0x5bd1e995UL;
  char* buf = (char*)ptr;
  size_t len_aligned = len & ~(size_t)0x7;
  char* end = buf + len_aligned;
  size_t hash = seed ^ (len * mul);
  for (char* p = buf; p != end; p += 8) {
      size_t data = shift_mix((size_t)p * mul) * mul;
      hash ^= data;
      hash *= mul;
  }
  if ((len & 0x7) != 0) {
      size_t data = load_bytes(end, len & 0x7);
      hash ^= data;
      hash *= mul;
  }
  hash = shift_mix(hash) * mul;
  hash = shift_mix(hash);
  return hash;
}
```

libstdc++-v3/
libsupc++/
hash_bytes.cc

The Standard Template Library (STL)

What is a **General** Hash Function?

```
size_t _hash(void* ptr, size_t len, size_t seed) {
  size_t mul = (0xc6a4a793UL << 32UL) + 0x5bd1e995UL;
  char* buf = (char*)ptr;
  size_t len_aligned = len & ~(size_t)0x7;
  char* end = buf + len_aligned;
  size_t hash = seed ^ (len * mul);
  for (char* p = buf; p != end; p += 8) {
      size_t data = shift_mix((size_t)p * mul) * mul;
      hash ^= data;
      hash *= mul;
  }
  if ((len & 0x7) != 0) {
      size_t data = load_bytes(end, len & 0x7);
      hash ^= data;
      hash *= mul;
  }
  hash = shift_mix(hash) * mul;
  hash = shift_mix(hash);
  return hash;
}
```

General types

What is a **General** Hash Function?

```
size_t _hash(void* ptr, size_t len, size_t seed) {
  size_t mul = (0xc6a4a793UL << 32UL) + 0x5bd1e995UL;
  char* buf = (char*)ptr;
  size_t len_aligned = len & ~(size_t)0x7;
  char* end = buf + len_aligned;
  size_t hash = seed ^ (len * mul);
  for (char* p = buf; p != end; p += 8) {
      size_t data = shift_mix((size_t)p * mul) * mul;
      hash ^= data;
      hash *= mul;
  }
  if ((len & 0x7) != 0) {
      size_t data = load_bytes(end, len & 0x7);
      hash ^= data;
      hash *= mul;
  }
  hash = shift_mix(hash) * mul;
  hash = shift_mix(hash);
  return hash;
}
```

General operations

What is a **General** Hash Function?

```
size_t _hash(void* ptr, size_t len, size_t seed) {
  size_t mul = (0xc6a4a793UL << 32UL) + 0x5bd1e995UL;
  char* buf = (char*)ptr;
  size_t len_aligned = len & ~(size_t)0x7;
  char* end = buf + len_aligned;
  size_t hash = seed ^ (len * mul);
  for (char* p = buf; p != end; p += 8) {
      size_t data = shift_mix((size_t)p * mul) * mul;
      hash ^= data;
      hash *= mul;
  }
  if ((len & 0x7) != 0) {
      size_t data = load_bytes(end, len & 0x7);
      hash ^= data;
      hash *= mul;
  }
  hash = shift_mix(hash) * mul;
  hash = shift_mix(hash);
  return hash;
}
```

General loops

What is a **Specialized** Hash Function?

Ex.: Length is always 11 bytes

What is a **Specialized** Hash Function?

DCC

```
for (char* p = buf; p != end; p += 8) {
  size_t data = shift_mix((size_t)p * mul) * mul;
  hash ^= data;
  hash *= mul;
}
if ((len & 0x7) != 0) {
  size_t data = load_bytes(end, len & 0x7);
  hash ^= data;
  hash *= mul;
}
```

XXX-XX-XXXX

Ex.: Length is always 11 bytes

What is a **Specialized** Hash Function?

DCC

```
static inline size_t load_3_bytes(const char* ptr) {
    return ((size_t)ptr[0] << 16)
         | ((size_t)ptr[1] << 8)
         | (size_t)ptr[2];
}

size_t _hash(void* ptr, size_t seed) {
    size_t mul = (0xc6a4a793UL << 32UL) + 0x5bd1e995UL;
    char* buf = (char*)ptr;
    // Load first 8 bytes
    size_t data1 = *(uint64_t*)buf;
    size_t hash = seed ^ (11 * mul);
    // Process first 8 bytes
    hash ^= shift_mix(data1 * mul) * mul;
    hash *= mul;
    // Process remaining 3 bytes
    size_t data2 = load_3_bytes(buf + 8);
    hash ^= shift_mix(data2 * mul) * mul;
    hash *= mul;
    // Final mixing
    hash = shift_mix(hash) * mul;
    hash = shift_mix(hash);
    return hash;
}
```

```
for (char* p = buf; p != end; p += 8) {
  size_t data = shift_mix((size_t)p * mul) * mul;
  hash ^= data;
  hash *= mul;
}
if ((len & 0x7) != 0) {
  size_t data = load_bytes(end, len & 0x7);
  hash ^= data;
  hash *= mul;
}
```

XXX-XX-XXXX

Ex.: Length is always 11 bytes

What is a **Specialized** Hash Function?

```
static inline size_t load_3_bytes(const char* ptr) {
    return ((size_t)ptr[0] << 16)
         | ((size_t)ptr[1] << 8)
         | (size_t)ptr[2];
}

size_t _hash(void* ptr, size_t seed) {
    size_t mul = (0xc6a4a793UL << 32UL) + 0x5bd1e995UL;
    char* buf = (char*)ptr;
    // Load first 8 bytes
    size_t data1 = *(uint64_t*)buf;
    size_t hash = seed ^ (11 * mul);
    // Process first 8 bytes
    hash ^= shift_mix(data1 * mul) * mul;
    hash *= mul;
    // Process remaining 3 bytes
    size_t data2 = load_3_bytes(buf + 8);
    hash ^= shift_mix(data2 * mul) * mul;
    hash *= mul;
    // Final mixing
    hash = shift_mix(hash) * mul;
    hash = shift_mix(hash);
    return hash;
}
```

✅ **No Loop**: Directly processes 11 bytes in **two memory loads**

XXX-XX-XXXX

Ex.: Length is always 11 bytes

What is a **Specialized** Hash Function?

```
static inline size_t load_3_bytes(const char* ptr) {
    return ((size_t)ptr[0] << 16)
         | ((size_t)ptr[1] << 8)
         | (size_t)ptr[2];
}

size_t _hash(void* ptr, size_t seed) {
    size_t mul = (0xc6a4a793UL << 32UL) + 0x5bd1e995UL;
    char* buf = (char*)ptr;
    // Load first 8 bytes
    size_t data1 = *(uint64_t*)buf;
    size_t hash = seed ^ (11 * mul);
    // Process first 8 bytes
    hash ^= shift_mix(data1 * mul) * mul;
    hash *= mul;
    // Process remaining 3 bytes
    size_t data2 = load_3_bytes(buf + 8);
    hash ^= shift_mix(data2 * mul) * mul;
    hash *= mul;
    // Final mixing
    hash = shift_mix(hash) * mul;
    hash = shift_mix(hash);
    return hash;
}
```

✅ **No Loop**: Directly processes 11 bytes in **two memory loads**

✅ **No Conditionals**: We remove all checks (`if ((len & 0x7) != 0)`).

XXX-XX-XXXX

Ex.: Length is always 11 bytes

What is a **Specialized** Hash Function?

DCC

```
static inline size_t load_3_bytes(const char* ptr) {
    return ((size_t)ptr[0] << 16)
         | ((size_t)ptr[1] << 8)
         | (size_t)ptr[2];
}

size_t _hash(void* ptr, size_t seed) {
    size_t mul = (0xc6a4a793UL << 32UL) + 0x5bd1e995UL;
    char* buf = (char*)ptr;
    // Load first 8 bytes
    size_t data1 = *(uint64_t*)buf;
    size_t hash = seed ^ (11 * mul);
    // Process first 8 bytes
    hash ^= shift_mix(data1 * mul) * mul;
    hash *= mul;
    // Process remaining 3 bytes
    size_t data2 = load_3_bytes(buf + 8);
    hash ^= shift_mix(data2 * mul) * mul;
    hash *= mul;
    // Final mixing
    hash = shift_mix(hash) * mul;
    hash = shift_mix(hash);
    return hash;
}
```

✅ **No Loop**: Directly processes 11 bytes in **two memory loads**

✅ **No Conditionals**: We remove all checks (`if ((len & 0x7) != 0)`).

✅ **Efficient Memory Access**: `*(uint64_t*)buf` loads **8 bytes** in one instruction.

XXX-XX-XXXX

Ex.: Length is always 11 bytes

What is a **Specialized** Hash Function?

DCC

```
size_t _hash(void* ptr, size_t seed) {
    size_t mul = (0xc6a4a793UL << 32UL) + 0x5bd1e995UL;
    char* buf = (char*)ptr;
    // Load first 8 bytes
    size_t data1 = *(uint64_t*)buf;
    size_t hash = seed ^ (11 * mul);
    // Process first 8 bytes
    hash ^= shift_mix(data1 * mul) * mul;
    hash *= mul;
    // Process remaining 3 bytes
    size_t data2 = load_3_bytes(buf + 8);
    hash ^= shift_mix(data2 * mul) * mul;
    hash *= mul;
    // Final mixing
    hash = shift_mix(hash) * mul;
    hash = shift_mix(hash);
    return hash;
}
```

XXX-XX-XXXX

Ex.: Key has constant subparts

What is a **Specialized** Hash Function?

DCC

```
size_t _hash(void* ptr, size_t seed) {
    size_t mul = (0xc6a4a793UL << 32UL) + 0x5bd1e995UL;
    char* buf = (char*)ptr;
    // Load first 8 bytes
    size_t data1 = *(uint64_t*)buf;
    size_t hash = seed ^ (11 * mul);
    // Process first 8 bytes
    hash ^= shift_mix(data1 * mul) * mul;
    hash *= mul;
    // Process remaining 3 bytes
    size_t data2 = load_3_bytes(buf + 8);
    hash ^= shift_mix(data2 * mul) * mul;
    hash *= mul;
    // Final mixing
    hash = shift_mix(hash) * mul;
    hash = shift_mix(hash);
    return hash;
}
```

```
// Extract digits, skipping `-` at positions 3 and 6
uint64_t data1 = ((uint64_t)buf[0] << 56) |
                 ((uint64_t)buf[1] << 48) |
                 ((uint64_t)buf[2] << 40) |
                 ((uint64_t)buf[4] << 32) |  // Skip '-'
                 ((uint64_t)buf[5] << 24) |
                 ((uint64_t)buf[7] << 16) |  // Skip '-'
                 ((uint64_t)buf[8] << 8)  |
                 ((uint64_t)buf[9]);

uint16_t data2 = ((uint16_t)buf[10] << 8)
                  | (uint16_t)buf[11];
```

XXX-XX-XXXX

Ex.: Key has constant subparts

What is a **Specialized** Hash Function?

```
// Extract digits, skipping `-` at positions 3 and 6
uint64_t data1 = ((uint64_t)buf[0] << 56) |
                 ((uint64_t)buf[1] << 48) |
                 ((uint64_t)buf[2] << 40) |
                 ((uint64_t)buf[4] << 32) |  // Skip '-'
                 ((uint64_t)buf[5] << 24) |
                 ((uint64_t)buf[7] << 16) |  // Skip '-'
                 ((uint64_t)buf[8] << 8)  |
                 ((uint64_t)buf[9]);

uint16_t data2 = ((uint16_t)buf[10] << 8)
               | (uint16_t)buf[11];
```

✅ **Skip - Characters**:

- We only process **9 bytes** instead of 11.
- The **3rd and 6th bytes (-) are ignored** via selective bit shifts.

XXX-XX-XXXX

Ex.: Key has constant subparts

What is a **Specialized** Hash Function?

```
// Extract digits, skipping `-` at positions 3 and 6
uint64_t data1 = ((uint64_t)buf[0] << 56) |
                 ((uint64_t)buf[1] << 48) |
                 ((uint64_t)buf[2] << 40) |
                 ((uint64_t)buf[4] << 32) |  // Skip '-'
                 ((uint64_t)buf[5] << 24) |
                 ((uint64_t)buf[7] << 16) |  // Skip '-'
                 ((uint64_t)buf[8] << 8)  |
                 ((uint64_t)buf[9]);

uint16_t data2 = ((uint16_t)buf[10] << 8)
                  | (uint16_t)buf[11];
```

DDD-DD-DDDD

Ex.: Key ranges on subset of bytes

What is a **Specialized** Hash Function?

DCC

```
// Extract digits, skipping `-` at positions 3 and 6
uint64_t data1 = ((uint64_t)buf[0] << 56) |
                 ((uint64_t)buf[1] << 48) |
                 ((uint64_t)buf[2] << 40) |
                 ((uint64_t)buf[4] << 32) |  // Skip '-'
                 ((uint64_t)buf[5] << 24) |
                 ((uint64_t)buf[7] << 16) |  // Skip '-'
                 ((uint64_t)buf[8] << 8)  |
                 ((uint64_t)buf[9]);

uint16_t data2 = ((uint16_t)buf[10] << 8)
               | (uint16_t)buf[11];
```

```
size_t data1 = ((size_t)(buf[0] & 0x0F) << 32) |
               ((size_t)(buf[1] & 0x0F) << 28) |
               ((size_t)(buf[2] & 0x0F) << 24) |
               ((size_t)(buf[4] & 0x0F) << 20) |
               ((size_t)(buf[5] & 0x0F) << 16) |
               ((size_t)(buf[7] & 0x0F) << 12) |
               ((size_t)(buf[8] & 0x0F) << 8)  |
               ((size_t)(buf[9] & 0x0F) << 4)  |
               ((size_t)(buf[10] & 0x0F));
```

DDD-DD-DDDD

Ex.: Key ranges on subset of bytes

What is a **Specialized** Hash Function?

✅ **Lower 4 Bits Only (`& 0x0F`):** We eliminate redundant upper bits (`0b0011xxxx`).

```
// Extract digits, skipping `-` at positions 3 and 6
uint64_t data1 = ((uint64_t)buf[0] << 56) |
                 ((uint64_t)buf[1] << 48) |
                 ((uint64_t)buf[2] << 40) |
                 ((uint64_t)buf[4] << 32) |  // Skip '-'
                 ((uint64_t)buf[5] << 24) |
                 ((uint64_t)buf[7] << 16) |  // Skip '-'
                 ((uint64_t)buf[8] << 8)  |
                 ((uint64_t)buf[9]);

uint16_t data2 = ((uint16_t)buf[10] << 8)
               | (uint16_t)buf[11];
```

```
size_t data1 = ((size_t)(buf[0] & 0x0F) << 32) |
               ((size_t)(buf[1] & 0x0F) << 28) |
               ((size_t)(buf[2] & 0x0F) << 24) |
               ((size_t)(buf[4] & 0x0F) << 20) |
               ((size_t)(buf[5] & 0x0F) << 16) |
               ((size_t)(buf[7] & 0x0F) << 12) |
               ((size_t)(buf[8] & 0x0F) << 8)  |
               ((size_t)(buf[9] & 0x0F) << 4)  |
               ((size_t)(buf[10] & 0x0F));
```

DDD-DD-DDDD

Ex.: Key ranges on subset of bytes

What is a **Specialized** Hash Function?

✅ **Lower 4 Bits Only (`& 0x0F`):** We eliminate redundant upper bits (`0b0011xxxx`).

✅ **Processes Only 9 Nibbles (4-bit values):** No need to store/process full bytes.

```
// Extract digits, skipping `-` at positions 3 and 6
uint64_t data1 = ((uint64_t)buf[0] << 56) |
                 ((uint64_t)buf[1] << 48) |
                 ((uint64_t)buf[2] << 40) |
                 ((uint64_t)buf[4] << 32) |  // Skip '-'
                 ((uint64_t)buf[5] << 24) |
                 ((uint64_t)buf[7] << 16) |  // Skip '-'
                 ((uint64_t)buf[8] << 8)  |
                 ((uint64_t)buf[9]);

uint16_t data2 = ((uint16_t)buf[10] << 8)
               | (uint16_t)buf[11];
```

```
size_t data1 = ((size_t)(buf[0] & 0x0F) << 32) |
               ((size_t)(buf[1] & 0x0F) << 28) |
               ((size_t)(buf[2] & 0x0F) << 24) |
               ((size_t)(buf[4] & 0x0F) << 20) |
               ((size_t)(buf[5] & 0x0F) << 16) |
               ((size_t)(buf[7] & 0x0F) << 12) |
               ((size_t)(buf[8] & 0x0F) << 8)  |
               ((size_t)(buf[9] & 0x0F) << 4)  |
               ((size_t)(buf[10] & 0x0F));
```

DDD-DD-DDDD

Ex.: Key ranges on subset of bytes

What is a **Specialized** Hash Function?

```
// Extract digits, skipping `-` at positions 3 and 6
uint64_t data1 = ((uint64_t)buf[0] << 56) |
                 ((uint64_t)buf[1] << 48) |
                 ((uint64_t)buf[2] << 40) |
                 ((uint64_t)buf[4] << 32) |  // Skip '-'
                 ((uint64_t)buf[5] << 24) |
                 ((uint64_t)buf[7] << 16) |  // Skip '-'
                 ((uint64_t)buf[8] << 8)  |
                 ((uint64_t)buf[9]);

uint16_t data2 = ((uint16_t)buf[10] << 8)
               | (uint16_t)buf[11];
```

```
size_t data1 = ((size_t)(buf[0] & 0x0F) << 32) |
               ((size_t)(buf[1] & 0x0F) << 28) |
               ((size_t)(buf[2] & 0x0F) << 24) |
               ((size_t)(buf[4] & 0x0F) << 20) |
               ((size_t)(buf[5] & 0x0F) << 16) |
               ((size_t)(buf[7] & 0x0F) << 12) |
               ((size_t)(buf[8] & 0x0F) << 8)  |
               ((size_t)(buf[9] & 0x0F) << 4)  |
               ((size_t)(buf[10] & 0x0F));
```

Ex.: Architecture is Intel

What is a **Specialized** Hash Function?

```
// Extract digits, skipping `-` at positions 3 and 6
uint64_t data1 = ((uint64_t)buf[0] << 56) |
             ((uint64_t)buf[1] << 48) |
             ((uint64_t)buf[2] << 40) |
             ((uint64_t)buf[4] << 32) |   // Skip '-'
             ((uint64_t)buf[5] << 24) |
             ((uint64_t)buf[7] << 16) |   // Skip '-'
             ((uint64_t)buf[8] << 8)  |
             ((uint64_t)buf[9]);

uint16_t data2 = ((uint16_t)buf[10] << 8)
               | (uint16_t)buf[11];
```

```
size_t data1 = ((size_t)(buf[0] & 0x0F) << 32) |
               ((size_t)(buf[1] & 0x0F) << 28) |
               ((size_t)(buf[2] & 0x0F) << 24) |
               ((size_t)(buf[4] & 0x0F) << 20) |
               ((size_t)(buf[5] & 0x0F) << 16) |
               ((size_t)(buf[7] & 0x0F) << 12) |
               ((size_t)(buf[8] & 0x0F) << 8)  |
               ((size_t)(buf[9] & 0x0F) << 4)  |
               ((size_t)(buf[10] & 0x0F));
```

```
// PEXT extracts only the 9 relevant nibbles,
// skipping `-` at pos 3 and 6.
// First, a mask selects 6 digits:
size_t data1 = _pext_u64(buf, 0x0F0F0F00F0F0F0F0ULL);
// Then, a mask selects last 3 digits:
size_t data2 = _pext_u64(tail, 0x00000F0F0F000000ULL);
// Merge extracted nibbles into a single integer
size_t data = (data1 << 12) | data2;
```

PEXT
(Parallel Bit Extract)

Ex.: Architecture is Intel

What is a **Specialized** Hash Function?

```
// PEXT extracts only the 9 relevant nibbles,
// skipping `-` at pos 3 and 6.
// First, a mask selects 6 digits:
size_t data1 = _pext_u64(buf, 0x0F0F0F00F0F0F0F0ULL);
// Then, a mask selects last 3 digits:
size_t data2 = _pext_u64(tail, 0x00000F0F0F000000ULL);
// Merge extracted nibbles into a single integer
size_t data = (data1 << 12) | data2;
```

```
size_t data1 = ((size_t)(buf[0] & 0x0F) << 32) |
               ((size_t)(buf[1] & 0x0F) << 28) |
               ((size_t)(buf[2] & 0x0F) << 24) |
               ((size_t)(buf[4] & 0x0F) << 20) |
               ((size_t)(buf[5] & 0x0F) << 16) |
               ((size_t)(buf[7] & 0x0F) << 12) |
               ((size_t)(buf[8] & 0x0F) << 8)  |
               ((size_t)(buf[9] & 0x0F) << 4)  |
               ((size_t)(buf[10] & 0x0F));
```

🚀 **Single PEXT Instruction** instead of multiple shifts and ORs

PEXT
(Parallel Bit Extract)

Ex.: Architecture is Intel

What is a **Specialized** Hash Function?

```
// PEXT extracts only the 9 relevant nibbles,
// skipping `-` at pos 3 and 6.
// First, a mask selects 6 digits:
size_t data1 = _pext_u64(buf, 0x0F0F0F00F0F0F0F0ULL);
// Then, a mask selects last 3 digits:
size_t data2 = _pext_u64(tail, 0x00000F0F0F000000ULL);
// Merge extracted nibbles into a single integer
size_t data = (data1 << 12) | data2;
```

```
size_t data1 = ((size_t)(buf[0] & 0x0F) << 32) |
               ((size_t)(buf[1] & 0x0F) << 28) |
               ((size_t)(buf[2] & 0x0F) << 24) |
               ((size_t)(buf[4] & 0x0F) << 20) |
               ((size_t)(buf[5] & 0x0F) << 16) |
               ((size_t)(buf[7] & 0x0F) << 12) |
               ((size_t)(buf[8] & 0x0F) << 8)  |
               ((size_t)(buf[9] & 0x0F) << 4)  |
               ((size_t)(buf[10] & 0x0F));
```

- 🚀 **Single PEXT Instruction** instead of multiple shifts and ORs
- 🔥 **Loads Only Relevant Bytes** rather than processing all 11 bytes separately

PEXT
(Parallel Bit Extract)

Ex.: Architecture is Intel

What is a **Specialized** Hash Function?

```
// PEXT extracts only the 9 relevant nibbles,
// skipping `-` at pos 3 and 6.
// First, a mask selects 6 digits:
size_t data1 = _pext_u64(buf, 0x0F0F0F00F0F0F0F0ULL);
// Then, a mask selects last 3 digits:
size_t data2 = _pext_u64(tail, 0x00000F0F0F000000ULL);
// Merge extracted nibbles into a single integer
size_t data = (data1 << 12) | data2;
```

```
size_t data1 = ((size_t)(buf[0] & 0x0F) << 32) |
               ((size_t)(buf[1] & 0x0F) << 28) |
               ((size_t)(buf[2] & 0x0F) << 24) |
               ((size_t)(buf[4] & 0x0F) << 20) |
               ((size_t)(buf[5] & 0x0F) << 16) |
               ((size_t)(buf[7] & 0x0F) << 12) |
               ((size_t)(buf[8] & 0x0F) << 8)  |
               ((size_t)(buf[9] & 0x0F) << 4)  |
               ((size_t)(buf[10] & 0x0F));
```

- 🚀 **Single PEXT Instruction** instead of multiple shifts and ORs
- 🔥 **Loads Only Relevant Bytes** rather than processing all 11 bytes separately
- ⚡ **Less Bitwise Arithmetic**, because `PEXT` does it all in hardware

PEXT
(Parallel Bit Extract)

Ex.: Architecture is Intel

Can we use it?

Can we use it?

DCC

Project Sepe

Automatic Synthesis of Hash Functions

Can we use it?

DCC

Project Sepe

Automatic Synthesis of Hash Functions

https://github.com/lac-dcc/sepe

How do we use SEPE?

```
(([0-9]{3})\.){3}[0-9]{3}
```

How do we use SEPE?

```
make_hash_from_regex.sh "(([0-9]{3})\.){3}[0-9]{3}"
```

```
// Simpler hash providing a baseline for efficiency:
struct synthesizedOffXorHash {
  size_t operator()(const std::string& key) const {
    const size_t h0 = load_u64_le(key.c_str());
    const size_t h1 = load_u64_le(key.c_str() + 7);
    return h0 ^ h1;
  }
};
```

How do we use SEPE?

```
make_hash_from_regex.sh "(([0-9]{3})\.){3}[0-9]{3}"
```

```
// Simpler hash providing a baseline for efficiency:
struct synthesizedOffXorHash {
  size_t operator()(const std::string& key) const {
    const size_t h0 = load_u64_le(key.c_str());
    const size_t h1 = load_u64_le(key.c_str() + 7);
    return h0 ^ h1;
  }
};
...
struct synthesizedPextHash {
    // Omitted for brevity in this example. See the
    // rest of this section for a complete example.
};
```

How do we use SEPE?

```
make_hash_from_regex.sh "(([0-9]{3})\.){3}[0-9]{3}"

// Simpler hash providing a baseline for efficiency:
struct synthesizedOffXorHash {
  size_t operator()(const std::string& key) const {
    const size_t h0 = load_u64_le(key.c_str());
    const size_t h1 = load_u64_le(key.c_str() + 7);
    return h0 ^ h1;
  }
};
...
void yourCppCode(void){
  std::unordered_map <std::string, int, synthesizedOffXorHash> map;
  map["255.255.255.255"] = 42;
  // ...
}
```

How do we use SEPE?

```
make_hash_from_regex.sh "(([0-9]{3})\.){3}[0-9]{3}"
```

```
std::unordered_map;
std::unordered_set;
std::unordered_multimap;
std::unordered_multiset
```

```
void yourCppCode(void){
  std::unordered_map <std::string, int, synthesizedOffXorHash> map;
  map["255.255.255.255"] = 42;
  // ...
}
```

How do we use SEPE (from examples)?

DCC

~~`make_hash_from_regex.sh "(([0-9]{3})\.){3}[0-9]{3}"`~~

How do we use SEPE (from examples)?

DCC

```
make_hash_from_regex.sh "(([0-9]{3})\.){3}[0-9]{3}"

keysynth "$(./bin/keybuilder < file_with_keys.txt)"
```

How do we use SEPE (from examples)?

~~make_hash_from_regex.sh "(([0-9]{3})\.){3}[0-9]{3}"~~

```
keysynth "$(./bin/keybuilder < file_with_keys.txt)"
```

```
123.456.789.012

487.337.492.389

298.347.239.844
```

How do we use SEPE (from examples)?

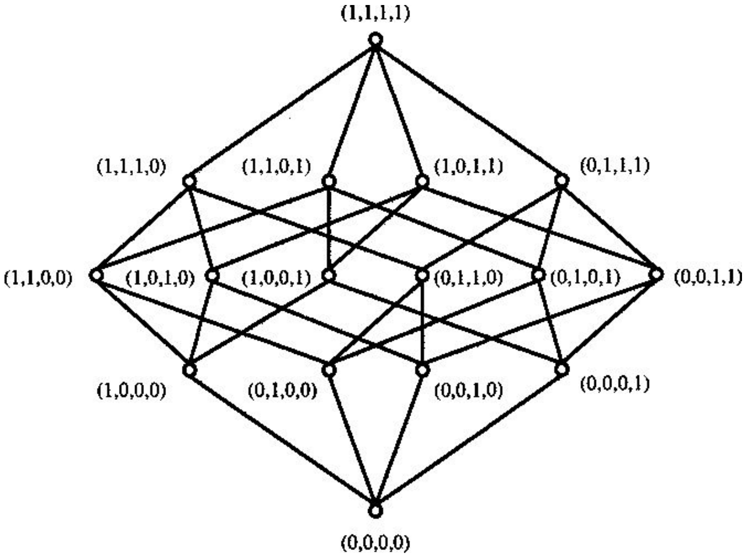
~~`make_hash_from_regex.sh "(([0-9]{3})\.){3}[0-9]{3}"`~~

```
keysynth "$(./bin/keybuilder < file_with_keys.txt)"
```

```
123.456.789.012

487.337.492.389

298.347.239.844
```



How do we use SEPE (from examples)?

~~make_hash_from_regex.sh "(([0-9]{3})\.){3}[0-9]{3}"~~

keysynth "$(./bin/keybuilder < **file_with_keys.txt**)"

Read the paper!

123.456.789.012

487.337.492.389

298.347.239.844

How Good is SEPE?

How Good is SEPE?

Function

STL

Abseil

City

FNV

Gpt

Naive

OffXor

Pext

Aes

How Good is SEPE?

Function

STL ⇐ Murmur Hash

Abseil

City

FNV ⇐ Fowler–Noll–Vo Hash

Gpt

Naive

OffXor

Pext

Aes

How Good is SEPE?

Function

STL

Abseil

City

FNV

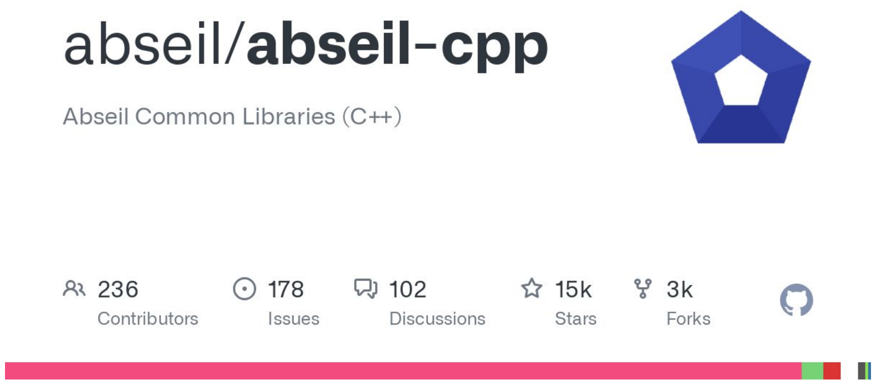
Gpt

Naive

OffXor

Pext

Aes



How Good is SEPE?

Function

STL

Abseil

City

FNV

Gpt

Naive

OffXor

Pext

Aes

How Good is SEPE?

DCC

- Function
- STL
- Abseil
- City
- FNV
- Gpt
- Naive
- OffXor
- Pext
- Aes

"Assume that keys are MAC addresses in the format `XX:XX:XX:XX:XX:XX', where all characters are hexadecimal. The `:' character is constant, so you can ignore them in your hash function. The fixed key size is 17 characters. The code is C++, and the keys are std::strings. Do not use std::hash. You do not need to assert key format. Produce an optimized hash function for these keys with an unrolled loop plus vectorization. Constant characters are always in the same position."

How Good is SEPE?

Function

STL

Abseil

City

FNV

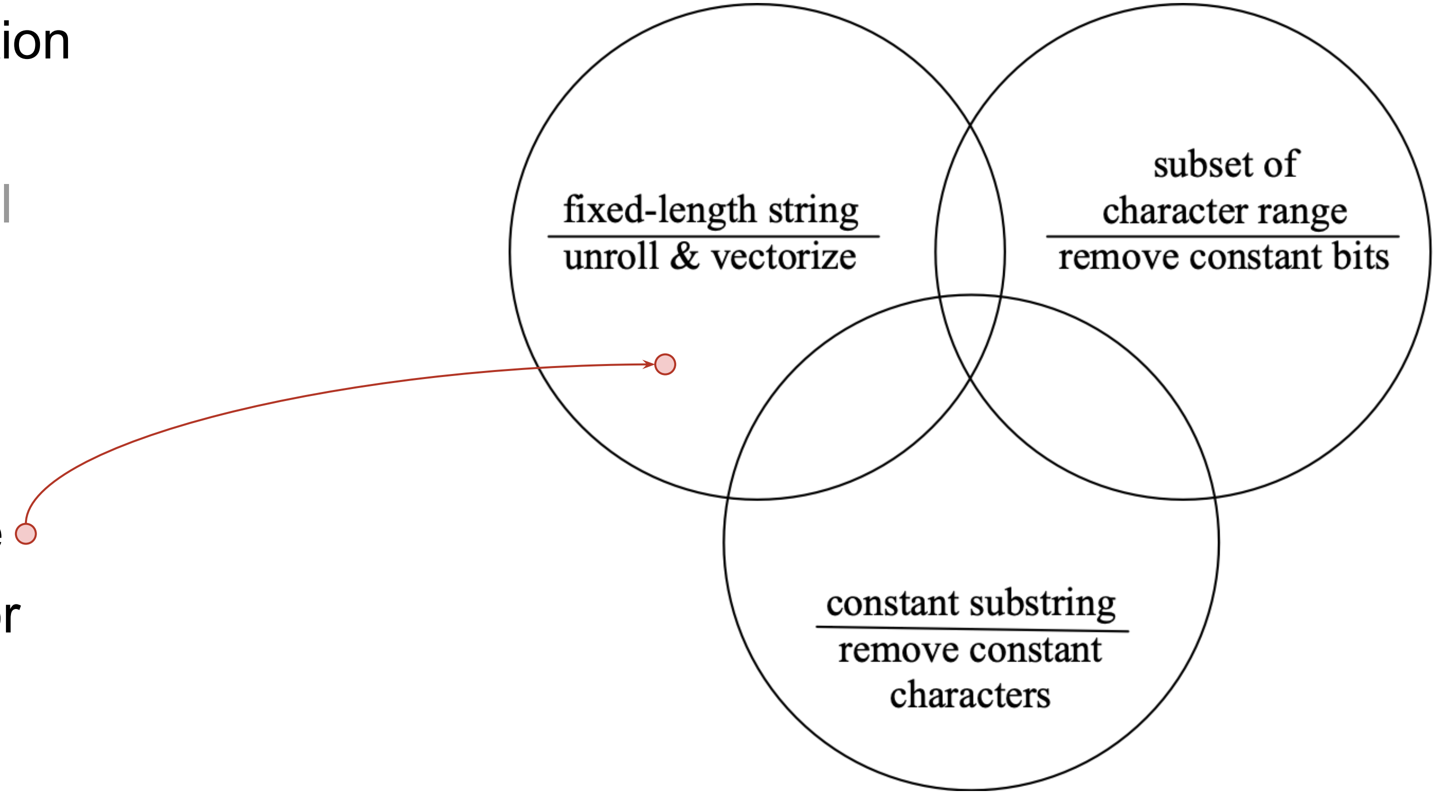
Gpt

Naive

OffXor

Pext

Aes



How Good is SEPE?

Function

STL

Abseil

City

FNV

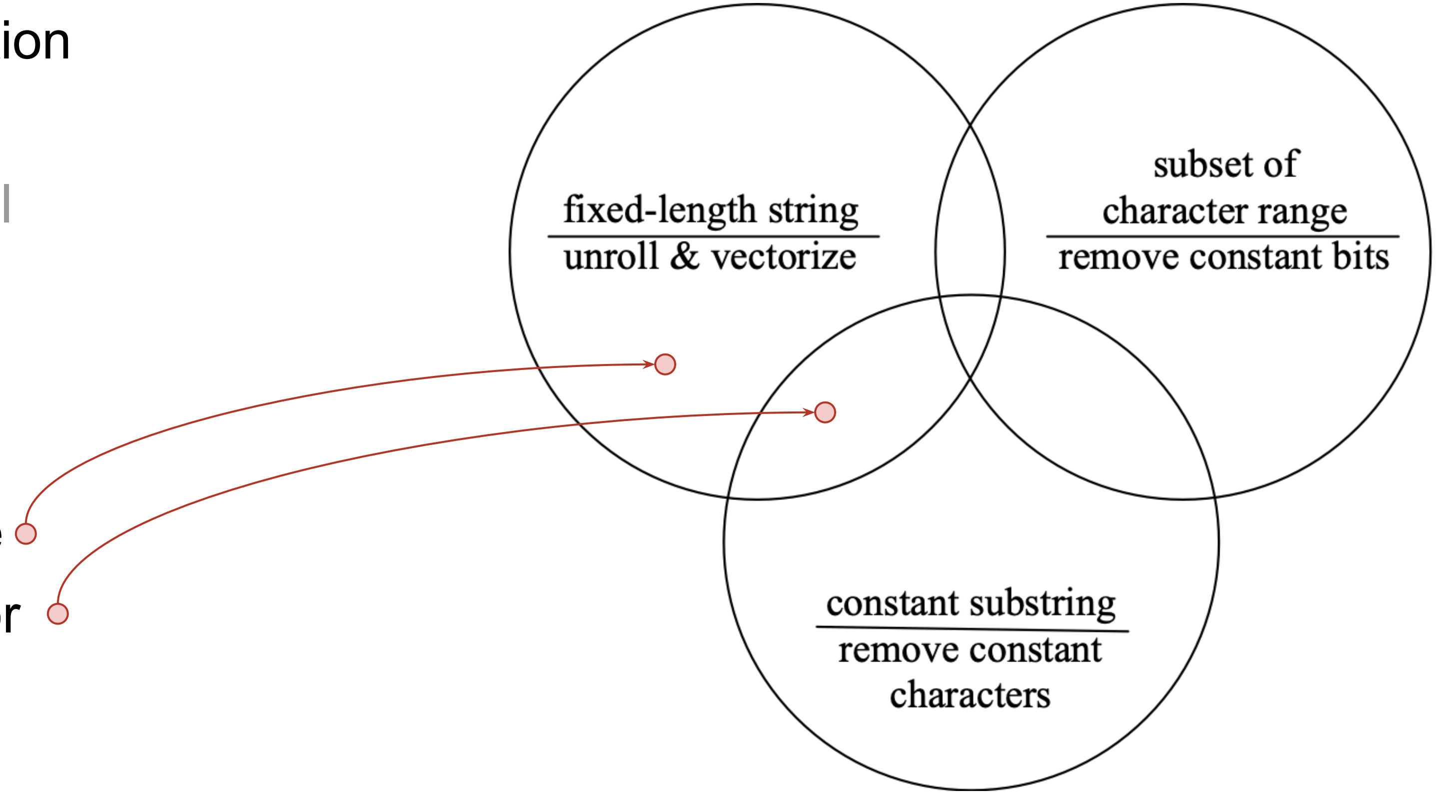
Gpt

Naive

OffXor

Pext

Aes



How Good is SEPE?

Function

STL

Abseil

City

FNV

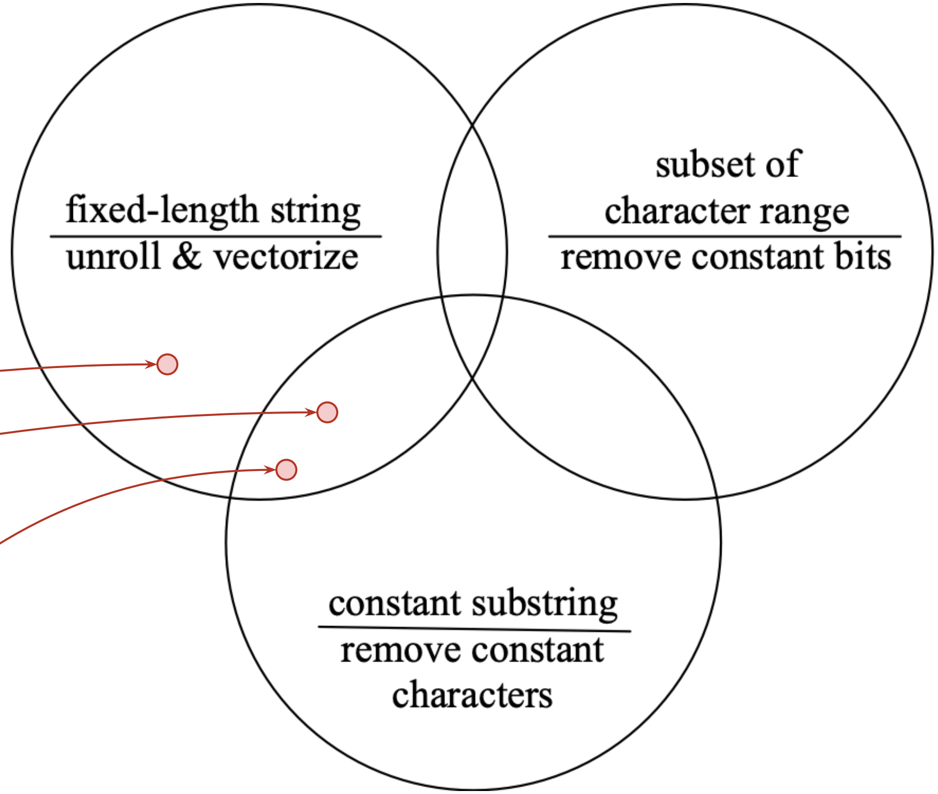
Gpt

Naive

OffXor

Pext

Aes



How Good is SEPE?

Function

STL

Abseil

City

FNV

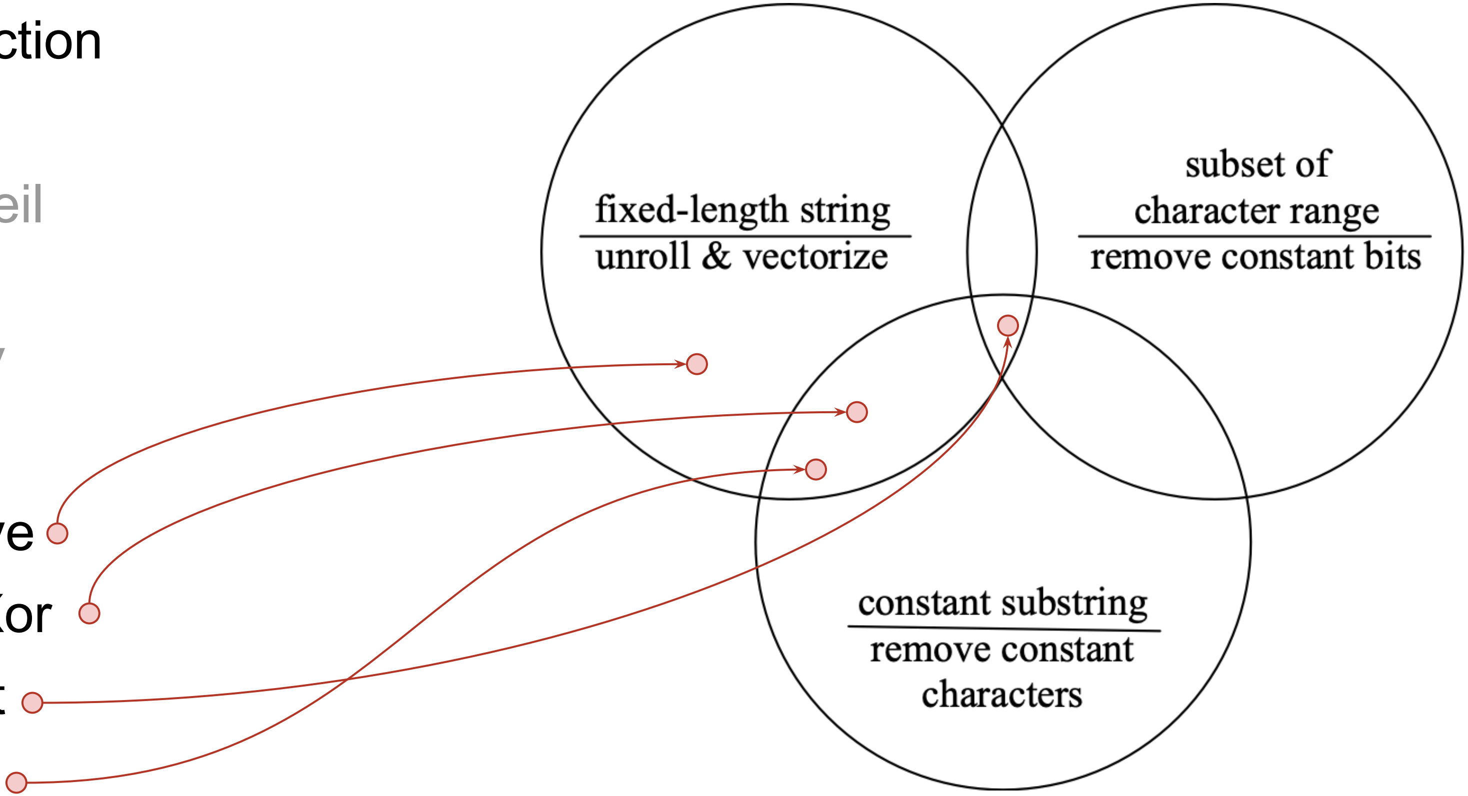
Gpt

Naive

OffXor

Pext

Aes



How Good is SEPE?

Function

STL

Abseil

City

FNV

Gpt

Naive

OffXor

Pext

Aes

Keys = 10,000 x

- SSN
- MAC
- IPv4 (same length with padding)
- IPv6
- URL with 43 bytes (20 constant)
- URL with 56 bytes (20 constant)
- ...

DCC

How Good is SEPE?

Function
STL
Abseil
City
FNV
Gpt
Naive
OffXor
Pext
Aes

Containers:

- `std::unordered_map`
- `std::unordered_set`
- `std::unordered_multimap`
- `std::unordered_multiset`

How Good is SEPE?

DCC

Function	B-Time
STL	3.19
Abseil	4.86
City	3.16
FNV	3.7
Gpt	3.22
Naive	3.04
OffXor	3.03
Pext	3.03
Aes	3.04

Bucket Time: Hashing Time + Data Structure Time

- Hashing speed
- Collision rate
- Data structure implementation

How Good is SEPE?

Function	B-Time
STL	3.19
Abseil	4.86
City	3.16
FNV	3.7
Gpt	3.22
Naive	3.04
OffXor	3.03
Pext	3.03
Aes	3.04

Bucket Time: Hashing Time + Data Structure Time

- Hashing speed
- Collision rate
- Data structure implementation

10,000
× 8 types of keys
× 4 containers

How Good is SEPE?

Function	B-Time	H-Time
STL	3.19	0.155
Abseil	4.86	1.816
City	3.16	0.128
FNV	3.7	0.599
Gpt	3.22	0.171
Naive	3.04	0.041
OffXor	3.03	0.037
Pext	3.03	0.05
Aes	3.04	0.063

Hashing Time: Total execution time of 10,000 activations of the hash function.

How Good is SEPE?

DCC

Function	B-Time	H-Time
STL	3.19	0.155
Abseil	4.86	1.816
City	3.16	0.128
FNV	3.7	0.599
Gpt	3.22	0.171
Naive	3.04	0.041
OffXor	3.03	0.037
Pext	3.03	0.05
Aes	3.04	0.063

Hashing Time: Total execution time of 10,000 activations of the hash function.

4-50x faster

How Good is SEPE?

Bucket Collision: Two keys point to the same bucket within the implementation of the STL container, considering 10,000 keys.

Function	B-Coll
STL	49.24
Abseil	48.89
City	49.17
FNV	49.06
Gpt	49.47
Naive	50.46
OffXor	50.67
Pext	49.42
Aes	49.21

How Good is SEPE?

DCC

Function	T-Coll
STL	0
Abseil	0
City	0
FNV	0
Gpt	7865
Naive	12
OffXor	12
Pext	0
Aes	9

Total Hash Collisions: Two keys are mapped to the same hash code, considering 10,000 keys.

How Good is SEPE?

Total Hash Collisions: Two keys are mapped to the same hash code, considering 10,000 keys.

Function	T-Coll
STL	0
Abseil	0
City	0
FNV	0
Gpt	**7865**
Naive	12
OffXor	12
Pext	0
Aes	9

Conclusion

Synthetic hash functions can be 4-50x faster

Conclusion

Synthetic hash functions can be 4-50x faster

Synthesis time is super fast (linear on key length)

Conclusion

DCC

Synthetic hash functions can be 4-50x faster

Synthesis time is super fast (linear on key length)

Good for indexing data (bad for crypto)

Conclusion

DCC

Synthetic hash functions can be 4-50x faster

Synthesis time is super fast (linear on key length)

Good for indexing data (bad for crypto)

Open question: Fast implementation with good distribution?

References & Contact

DCC

Fernando Pereira

fernando@dcc.ufmg.br

http://lac.dcc.ufmg.br

References & Contact

DCC

Fernando Pereira

fernando@dcc.ufmg.br

http://lac.dcc.ufmg.br

References

Automatic Synthesis of Specialized Hash Functions

https://homepages.dcc.ufmg.br/~fernando/publications/papers/CGO25_Hoffmann.pdf

References & Contact

DCC

Project Sepe

Automatic Synthesis of Hash Functions

https://github.com/lac-dcc/sepe

References

Automatic Synthesis of Specialized Hash Functions

https://homepages.dcc.ufmg.br/~fernando/publications/papers/CGO25_Hoffmann.pdf