# Automatic Synthesis of Specialized Hash Functions

Renato B. Hoffmann
PUC-RS
Brazil
renato.hoffmann@edu.pucrs.br

Leonardo G. Faé
PUC-RS
Brazil
leonardo.fae@edu.pucrs.br

Dalvan Grieber
PUC-RS
Brazil
dalvan.griebler@pucrs.br

Xinliang David Li
Google
USA
davidxl@google.com

Fernando Magno Quintão Pereira
Federal University of Minas Gerais
Brazil
fernando@dcc.ufmg.br

## Abstract

This paper introduces a technique for synthesizing hash functions specialized to particular byte formats. This code generation method leverages three prevalent patterns: (i) fixed-length keys, (ii) keys with common subsequences, and (iii) keys ranging on predetermined sequences of bytes. Code generation involves two algorithms: one identifies relevant regular expressions within key examples, and the other generates specialized hash functions based on these expressions. Comparative analysis demonstrates that the synthetic functions outperform the general-purpose hashes in the C++ Standard Template Library and the Google `Abseil` Library when keys are given in ascending, normal or uniform distribution. In applications where low-mixing hashes are acceptable, the synthetic functions achieve speedups ranging from 2% to 11% on full benchmarks, and speedups of almost 50x once only hashing speed is considered.

***CCS Concepts:*** • **Software and its engineering → Compilers**.

***Keywords:*** Hash, Synthesis, Specialization

## 1 Introduction

A hash function is a mathematical algorithm designed to convert variable-length inputs into fixed-length outputs [19]. Hash functions, along with their associated hash tables, are widely employed in data storage and retrieval. Depending on their properties, hash functions are classified as either cryptographic or non-cryptographic [3]. In both cases, there are many algorithms available to compute hashes of sequences of bytes [5]. Non-cryptographic hash functions, such as `FNV` [7] or `Murmur` [1], are designed for applications that do not require the security guarantees of cryptographic hashes. Consequently, they tend to offer faster performance and require fewer resources. Due to this efficiency, non-cryptographic functions are building blocks of a large number of computing applications. As illustrated by Kanev et al. [14]'s Figure 4, approximately 2% of all computational cycles in Google Datacenters worldwide are dedicated to hashing.

Typical hash functions are designed to handle arbitrary byte sequences. However, if constraints are applied to these sequences, then it might be possible to develop faster hashes while also maintaining or reducing collision rates. For instance, the `gperf` Perfect Hash Function Generator from GNU produces a perfect hash for a given set of user-provided keywords [23]. Nonetheless, as Section 4 will show, functions generated by `gperf` often exhibit slower performance compared to the default hash implementations [12] found in the C++ Standard Template Library (STL), which are general. Indeed, to the best of our knowledge, there is currently no code generator capable of producing specialized hash functions that can outperform the implementations of hash functions [11, 12] provided in C++ STL, whether in terms of computational speed or collision rate.

***Generation of Specialized Code.*** This paper describes a code generation methodology to produce specialized hash functions. Specializations are enabled by three patterns: sequences of bytes with fixed lengths; sequences of bytes with common subsequences; and sequences ranging on fixed byte intervals. Many applications use keys that meet these constraints: social security numbers, plate numbers, MAC addresses, etc. However, the presence of only one of these properties is already sufficient to enable specialization. Synthesis is parameterized by a set $\mathcal{S}$ of representative keys and a lattice $\mathcal{L}$ of byte intervals. Section 3.1 explains how we identify patterns of interest on $\mathcal{S}$. The byte that represents characters in the same index is the least upper bound of these characters in $\mathcal{L}$. Section 3.2 explains how these patterns are then used to guide code generation, which relies on machine-specific instructions to shuffle and compress bits.

***Summary of Results.*** The ideas described in this paper have been implemented in a library henceforth called Sepe. Sepe generates C++ functions that use either x86 or ARM-specific instructions. These functions are compatible with STL data structures such as `std::unordered_map` and `std::unorder ed_set`. Synthesis is guided either by examples of keys or by a user-provided regular expression describing the format of keys. Sepe functions trade dispersion by

performance. In this sense, they follow the design philosophy that Kraska et al. stress in their "*Case for Learned Index Structures*", e.g., "*If the goal is to build a highly-tuned system to store and query ranges of fixed-length records over a set of continuous integer keys, [...] the key itself can be used as an offset*" [15]. Thus, in scenarios where low-dispersion is acceptable, i.e., where an adversary is not expected to force collisions, Section 4.1 shows that these synthetic hash functions can be almost 50x faster than general functions taken from `Abseil` (implementations of `CityHash` [20] and `LLH` [21]) or the Standard Template Library (implementations from `murmur` [12] and `FNV` [11]). Additionally, Section 4.2 shows that these hashes match standard hash functions in terms of collision rate and bucket collisions (a measure of how many memory bins are created in data structures to store the hash keys), when given either uniformly or normally distributed keys. Code generation is also fast: Section 4.6 demonstrates that it is linearly proportional to the size of the largest key.

## 2 Hashing: an Overview

As mentioned in Section 1, hash functions fall into two categories: cryptographic and non-cryptographic. Cryptographic hash functions are tailored for security applications like digital signatures and password storage, embodying properties such as collision resistance, pre-image resistance, and the avalanche effect. Pre-image resistance ensures the difficulty of reverse engineering the function, collision resistance emphasizes the challenge of finding distinct inputs with identical outputs, and the avalanche effect indicates that a slight input change results in a significantly different output. In contrast, non-cryptographic hash functions prioritize speed over security. They are employed in data structures like sets and maps for rapid data retrieval and indexing. Desirable properties of non-cryptographic hashes include, in addition to collision resistance, determinism, and efficiency. Determinism ensures consistent outputs for identical inputs, and efficiency underscores the quick computation of hashes. This paper focuses specifically on non-cryptographic functions. Example 2.1 provides an example of such function.

**Example 2.1.** Figure 1 shows a slightly simplified implementation of murmur hash, as taken from the C++ Standard Template Library. This function processes arrays of characteres, eight characteres at a time. To facilitate processing data as 64-bit integers within the main loop, Line 4 removes any bytes that are not divisible by `sizeof(size_t)`; hence, allowing the main loop to process the data as 64-bit integers.

### 2.1 Hash Specializations

The code in Figure 1 is short, and relatively simple; however, it is very efficient. An informal experiment discussed on Stack Overflow on April 23rd, 2012 [4] showed that this implementation outperformed eight different implementations of hash functions (`FNV-1a`, `FNV-1`, `DBJ2a`, `DJB2`, `SDBM`, `SuperFastHash`, `CRC32` and `LoseLose`) across three workloads. The running time of the implementation in Figure 1 is proportional to $n \times c_1 + c_2$, where $n$ is the size of the key in bytes, $c_1$ is a set of operations that run for every byte in the key (`ptr`), and $c_2$ represents the remaining constant operations. While $n$ depends on the problem, the constants $c_1$ and $c_2$ can be reduced if the hash function is specialized for particular inputs. Many distributions of industrial-quality hash functions implement some form of specialization. Example 2.2 illustrates one such case.

```
01 size_t _hash(void* ptr, size_t len, size_t seed) {
02   size_t mul = (0xc6a4a793UL << 32UL) + 0x5bd1e995UL;
03   char* buf = (char*)ptr;
04   size_t len_aligned = len & ~(size_t)0x7;
05   char* end = buf + len_aligned;
06   size_t hash = seed ^ (len * mul);
07   for (char* p = buf; p != end; p += 8) {
08     size_t data = shift_mix((size_t)p * mul) * mul;
09     hash ^= data;
10     hash *= mul;
11   }
12   if ((len & 0x7) != 0) {
13     size_t data = load_bytes(end, len & 0x7);
14     hash ^= data;
15     hash *= mul;
16   }
17   hash = shift_mix(hash) * mul;
18   hash = shift_mix(hash);
19   return hash;
20 }
```

**Figure 1.** Implementation of Murmur hash for 64-bit `size_t`, taken from the C++ Standard Template Library. Code available at `libstdc++-v3/libsupc++/hash_bytes.cc:138` on February 26, 2024 [12]. The type of variables in this code has been simplified for readability.

**Example 2.2.** Figure 2 illustrates a segment of the Polymur Hash implementation, which is a 64-bit non-cryptographic universal hash function. The function comprises three specializations, as highlighted in Figure 2. Each specialization addresses distinct key lengths. These specializations optimize the Polymur implementation for efficient processing of short inputs while maintaining practicality for longer inputs. Notice that this implementation is not a specialization of the code earlier seen in Figure 1. Hence, in our setting, in spite of the length-aware specializations, the hash functions available in the Standard Template Library still outperform the code in Figure 2 in terms of hashing speed.

***The Specialization Zoo.*** This paper considers three types of specializations on hash keys. Each type of specialization results from some constraint imposed on the byte sequence that forms the key. The following constraints are considered:

```
01 static inline uint64_t polymur_hash_poly611(
02   const uint8_t* buf,
03   size_t len,
04   const PolymurHashParams* p,
05   uint64_t tweak
06 ) {
07   ...
08   if (POLYMUR_LIKELY(len <= 7)) { ... }
09   if (POLYMUR_UNLIKELY(len >= 50)) { ... }
10   if (POLYMUR_LIKELY(len >= 8)) { ... }
11   ...
12 }
```

**Figure 2.** Snippet of the implementation of Polymur Hash, taken from https://github.com/orlp/polymur-hash on February 26, 2024. The implementation is specialized for three specific key lengths.

**length:** The hash is computed for sequences formed by a fixed number $n$ of bytes.
**range:** The hash is computed for sequences ranging over a subset of byte values.
**const:** The hash is computed for sequences that contain constant subsequences occurring at fixed positions.

Each one of the constraints mentioned above restricts the type of sequences. And each one of these restrictions leads to a different form of optimization, which Figure 3 highlights. These optimizations are the subject of Section 3. These specializations can be combined, as the next example explains. Example 2.3 discusses a handwritten implementation of a hash function; however, the techniques that Section 3 introduces generate functions that are equally as efficient.

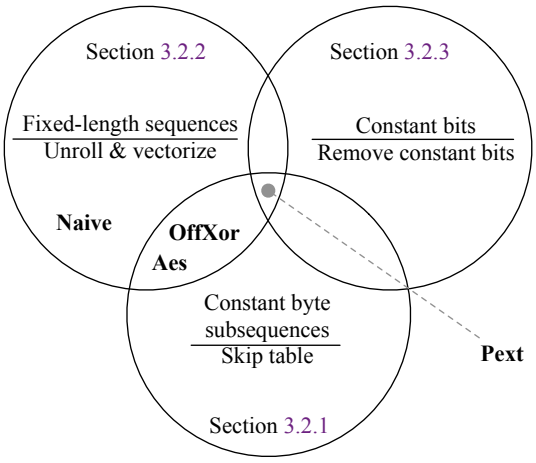



**Figure 3.** The three subsets of the byte sequences that enable the specializations discussed in this paper, with constraints above the line, and optimizations under it. Section 4 shall define the hash functions **Naive**, **Pext**, **OffXor** and **Aex**.

**Example 2.3.** Figure 4 shows a handwritten hash function specialized for US' Social Security Numbers (SSN). SSNs are 11-character strings following the format "xxx.xx.xxxx", where $x \in [0 \ldots 9]$. The function in Figure 4 was suggested in a reddit forum where the library introduced in this paper was been discussed [1]. The fixed length of SSNs allows us to write the function with just two loads, as seen in Lines 03 and 05 of Figure Figure 4. The fact that the digits can be represented with only four bits lets us have a bijection of 11-byte strings to 8-byte integers. The constant bits are discarded by the shift operation in Line 06. Finally, the fact that two of the characters (the dots) are constant substrings lets us disregard them when performing the shift and the addition in Line 06.

```
01 size_t _hash(char* ptr, size_t len, size_t seed) {
02   size_t hash1 = 0;
03   std::memcpy(&hash1, ptr, sizeof(size_t));
04   size_t hash2 = 0;
05   std::memcpy(&hash2, (char*)ptr + 3, sizeof(size_t));
06   size_t hash3 = hash2 << 4;
07   size_t hash4 = hash1 + hash3;
08   return hash4;
09 }
```

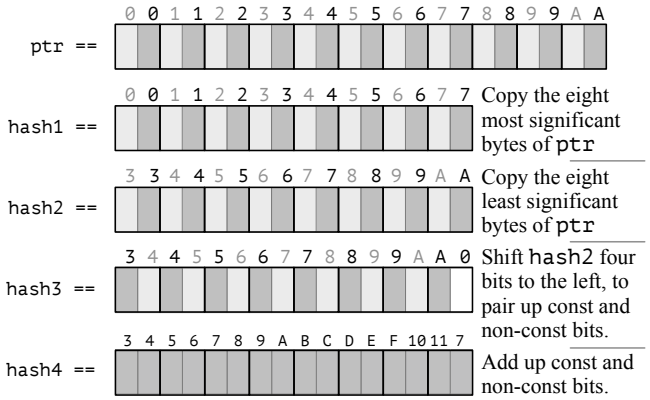



**Figure 4.** Handwritten version of a hash function specialized for US Social Security Numbers. The figure shows values separated in groups of four bits. Gray boxes represent groups of four constant bits (the upper four bits of ASCII digits). Dark gray boxes represent non-constant bits. To build a bijection between strings and 64-bit numbers, this hash function splits the key into two 64-bit words, shifts left four bits in one of these words, and adds them up; hence, summing up constant and non-constant bits.

## 3 Automatic Specialization

Sepe provides users with two different interfaces to create hash functions. Users can generate specialized hashes by feeding Sepe with regular expressions or with examples of keys. Example 3.1 illustrates these two approaches. As the example shows, users synthesize hash functions with only one line of command. Generation via examples produces a

[1] See discussion at https://www.reddit.com/r/Compilers/comments/1alc94w/automatic_specialization_of_hash_functions/.

regular expression; hence, Sepe's second approach is essentially an abstraction layer built over the first one. Section 3.1 explains how Sepe infers regular expressions out of examples of keys. Section 3.2, in turn, explains how these regular expressions guide Sepe's code generation engine.

**Example 3.1.** Figure 5 (a) shows the command line that we can use to produce hash functions for IPv4 keys of fixed length. The command in Figure 5 (b) will produce the same hash functions; however, instead of passing a regular expression to Sepe, this command feeds it with a file containing examples of keys. Either one of these two commands generates two hash functions. Figure 5 (c) shows the signature of one of these functions, and the body of the other. Finally, Figure 5 (d) shows how either of these two functions can be incorporated into typical STL code.

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

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

**(c)**
```
struct synthesizedPextHash {
    // Omitted for brevity in this example. See the
    // rest of this section for a complete example.
};

// 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;
  }
};
```

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

**Figure 5.** Sepe's getting started tutorial. See Example 3.1.

### 3.1 Creating Regular Expressions

The creation of hash functions from examples of keys requires converting these keys into a regular expression that recognizes them. The regular expression cannot be too conservative; otherwise, Sepe could simply output '.*' for any key. It cannot be too specific either, for we want to hash keys that are not in the set of examples. As a compromise, we identify regular expressions via a *semilattice of bit pairs*, which Definition 3.2 formalizes:

**Definition 3.2** (The Quad-Semilattice)**.** Let $\beta = \{00, 01, 10, 11\} \cup \{\top\}$ be the set of 4 quad numbers plus an extra element $\top$. If $b_0, b_1 \in \beta$, then the semilattice $\mathcal{L} = \langle \beta, \vee \rangle$ is such that $b_0 \vee b_1 = b_0$, if $b_0 = b_1$, or $\top$ otherwise.

**Theorem 3.3.** *$\langle \beta, \vee \rangle$ is a semilattice.*

**Proof:** We must show that (i) $\vee$ determines a partial order in $\beta$; and (ii) $b \vee \top = \top$.

- **i:** we define $\leq$ as follows: $b_0 \leq b_1$ if $b_0 \vee b_1 = b_1$. Thus, $b \leq \top$ and $b \leq b$, for any $b \in \beta$. If $b_0 \neq b_1 \neq \top$; then $b_0$ and $b_1$ are not comparable.
- **ii:** Follows from case analysis on the elements of $\beta$. □

Given a set of $m$ keys $\mathcal{S}$, we define the regular expression $r$ that recognizes every key in $\mathcal{S}$ as $f = c_0 c_1 \ldots c_{n-1}$, where $c_i = s_1[i] \vee s_2[i] \vee \ldots \vee s_m[i]$. Each $s_j \in \mathcal{S}, 1 \leq j \leq m$, such that $k_j[i]$ is the $i^{th}$ bit pair in $k_j$, and $n$ is the length, in bit pairs, of the largest key. If a given key $s_j$ contains fewer than $i$ bit pairs, we let $s_j[i] = \top$. Example 3.4 illustrates the process of producing a regular expression.

**Example 3.4.** Lets us assume we are building a regular expression that describes airport codes, such as those assigned by the International Air Transport Association (IATA). Figure 6 shows three keys, plus their ASCII and quad representations. Figure 6, on the bottom, also shows the least upper bound of these three keys, according to Definition 3.2. The least upper bound of eight of the bit pairs is the top element, whereas the remaining four are either `01` or `00`. Lets us assume that we are also hashing ICAO airport code, which have four letters. In this case, the missing fourth letter in the IATA code would be treated as four top elements. We would have, for instance, $JFK \vee LaX \vee GRu \vee RJTT = 0100\top^2 01\top^3 01\top^7$, where $\top^i$ is the sequence of $i$ occurrences of the top element.


**Figure 6.** The join operation on the quad-semilattice.

***Rationale.*** The quad-semilattice of Definition 3.2 groups bits in pairs, with the goal of identifying constant bits within keys. Any other granularity that is a power of two would fit the purpose of the optimizations to be discussed in Section 3.2. We opted to use bit pairs, because bit pairs are sufficiently expressive to identify constant bits in three important subsets of ASCII characters: digits, lower-case letters and upper-case letters. The lattice of bit pairs identifies four constant bits in digits, and two constant bits in letters (the first bit pair). Example 3.5 supports these observations.

**Example 3.5.** The keys used in Example 3.4 indicate that the four upper bits of keys (`0100`) are constants. These bits remain `0100` in Figure 6 because the examples of keys in that position use only upper-case ASCII characters ('J', 'L', 'G' and 'J'), which share that common four-bit prefix. However,

one lower-case ASCII character would show that only the first two bits (01) are invariant for keys involving lower and upper-case characters. In this example, the shortest sequence of constant bits is two.

Had we used a coarser granularity in Example 3.5, such as hexadecimals, to group bits, then we would miss the fact that the first quad of every ASCII upper case letter is invariant[2]. As explained in Section 3.1, users of Sepe can create hash functions from regular expressions, or from examples of keys. In the latter case, users must use good sets of examples. A good set of examples contain, for each quad, every possible combination of bits that is possible at that position of a key, as Example 3.6 explains.

**Example 3.6.** A set of good examples to describe fixed-length IPv4 keys in the *ddd.ddd.ddd.ddd* format requires each one of the *d* digits to change at least once. Two examples are already enough to meet this requirement, such as an IPv4 address with just 0s (ASCII('0')=00110000) and another with just 5s (00110101). For URLs containing upper/lower letters and digits, two examples would also be enough to exercise all the possible variations of quads, such as a sequence of 'E's (01000101), and a sequence of '0's (00110000).

### 3.2 Code Generation

The code generation phase of Sepe leverages the three constraints that Figure 3 highlights to synthesize optimized hash function, and follows the sequence of actions that Figure 7 specifies. These constraints do not need to occur together to enable specializations. The occurrence of any combination of them already permit the manifestation of the different optimizations that the rest of this section describes. The rest of this section discusses each part of Figure 7 in more details.

```
01 def synthesize(key):
02     ranges = parseRanges(key)
03   # Section 3.2.1
04     offsets = ignoreConstantSubsequences(ranges)
05   # Section 3.2.3
06     masks, shifts = calculateMasks(key,offsets)
07     hashables = removeConstBits(masks,shifts,offsets)
08   # Section 3.2.2
09     hashFunc = unrollSequences(hashables)
10     return hashFunc
```

**Figure 7.** Overview of the three phases that are present in Sepe's code generator.

#### 3.2.1 Constant Subsequences.

The regular expressions created after the Algorithm discussed in Section 3.1 might include constant words. We call a constant word a continuous sequence of quads that is equal or greater than the size of the minimum word that can be addressed in a given architecture. Constant words do not contribute to differentiating the hash codes of distinct keys. Sepe's code generator avoids loading constant sequences of word that exceed the size of the target architecture's vector lane. To this end, we build a *skip table*: an array with offsets to skip when computing the hash. Example 3.7 illustrates this approach.

**Example 3.7.** Function h1, in Figure 8 is a specialization of the general implementation hash pattern earlier seen in Figure 1. For simplicity, we have replaced hash-specific operations with two placeholders: initialize_hash and update_hash. In this example, we assume that keys follow the format "https://example.com/src?ssn=ddd.dd.dddd &name=..." Thus, keys contains only two parts that are not constant: the social security number (ddd.dd.dddd) and the name field. Figure 9 shows the words that considered and skipped in a particular example of key.

```
01 size_t h1(const char* ptr, size_t len, size_t seed,
02     const size_t* skip, size_t sk_len) {
03   size_t hash = initialize_hash(len, seed);
04   ptr = ptr + skip[0];
05   const char* end = ptr + len;
06   for(size_t c = 1; c <= sk_len; ++c) {
07     hash = update_hash(hash, load_u64(ptr));
08     ptr += skip[c];
09   }
10   while(ptr < end) {
11     hash = update_hash_u8(hash, ptr);
12     ptr++;
13   }
14   return hash;
15 }
```

*Use the skip table to jump over words that only contain constant quads*

**Figure 8.** Synthetic hash function that skip constant subsequences of bytes.

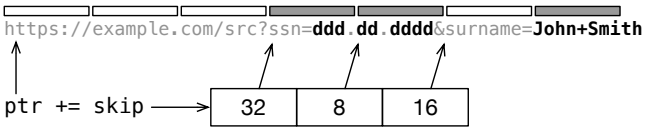



**Figure 9.** Example of skip table. The gray tabs mark the words that will be considered when building the hash code. The white tabs highlight words that will be skipped.

#### 3.2.2 Fixed-Length Sequences.

If keys have fixed length, we avoid iterations, and load words directly outside the main loop. If the key contains constant subsequences of bytes, then we can avoid loading these words without the need to use the skip table of Section 3.2.1. Therefore, this combination of constant subsequences and keys of fixed length essentially lets us apply partial evaluation on the code earlier discussed

[2] mischaracterizing variable bits as constants will not lead the code generator of Section 3.2 to produce incorrect hashes; however, mischaracterization might lead to the generation of hashes with higher collision rates.

in Example 3.7. Example 3.8 illustrates this process of code generation.

**Example 3.8.** Lets simplify the keys seen in Example 3.7 by removing the `name` field. In other words, lets assume keys with the format: `https://example.com/src?ssn=ddd.dd.dddd`. In this case, Figure 10 shows the hash function that would be synthesized by our code generation approach. Notice that function `h2` in Figure 10 loads overlapping bytes, e.g., `ptr` and `ptr + 3` share five bytes.

```
01 size_t h2(const char* ptr, size_t len, size_t seed) {
02   size_t hash = initialize_hash(9, seed);
03   hash = update_hash(hash, load_u64(ptr));
04   hash = update_hash(hash, load_u64(ptr+3));
05   return hash;
06 }
```

**Figure 10.** Synthetic hash function that processes fixed-length keys in the format `ddd.dd.dddd`.

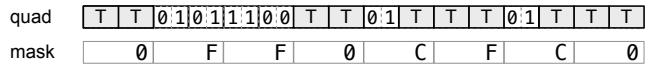
Example 3.8 shows that our synthetic hash functions might use words with overlapping bits when generating code for fixed-length sequences. Overlapping happens when the constant part of a sequence of bytes is not a multiple of the machine word (eight bytes in Example 3.8). We opted for loading only non-constant sequences when the length of keys is fixed. Thus, the last load of a non-constant sequence of $n$ bits always starts at position $n - 8$.

**3.2.3 Constant Bits.** Many computer architectures provide instructions that perform the parallel extraction of bits from a source operand. A typical example is the x86's `pext` instruction, or the Aarch64's `bext` instruction. This instruction employs a mask to selectively extract bits from the source operand. These bits—which can be non-contiguous—will be stored into the contiguous low-order bit positions in the destination register. The remaining upper bits of the destination register are zeroed. Figure 11 uses pseudo-code to explain the semantics of bit extraction. Bit extraction, as implemented via `pext`-like instructions is fast. For instance, `pext`'s latency varies from 3 to 20 cycles on modern x86 CPUs [6].

**Example 3.9.** Continuing Example 3.8, Figure 12 shows a hash function synthesized for keys that represent social security numbers. The two masks created at Lines 04 and 05 cover the two words necessary to load an 11-byte SSN number in the format `ddd.dd.dddd`. Because the non-constant part of the key is formed by numbers, only the less significant four bits of each digit vary. Consequently, `pext` is able to create a bijection from keys to 32 bit values.

## 4 Empirical Evaluation

This section evaluates the following research questions:




```
01 size_t _pext_u64(size_t src, size_t mask) {
02   size_t dst = 0;
03   for (u8 m = 0, k = 0; m < 64; ++m) {
04     if (mask[m]) dst[k++] = src[m];
05   }
06   return dst;
07 }
```

**Figure 11.** Parallel bit extraction. We use the quads produced in Section 3.1 to guide the generation of the extraction mask, as shown on the top of the figure.

```
01 size_t h3(const char* ptr, size_t len, size_t seed) {
02   size_t hash = initialize_hash(9, seed);
03   Step 1: define the mask based on the constant quads:
04   size_t mk0 = 0x0f000f0f000f0f0f;
05   size_t mk1 = 0x0f0f0f0000000000;
06   Step 2: compress bits using masks and selective loading:
07   size_t hashable0 = _pext_u64(load_u64(ptr), mk0);
08   size_t hashable1 = _pext_u64(load_u64(ptr+3), mk1);
09   Step 3: shift significant bits as far to the left as possible:
10   size_t shift1 = hashable1 << 52;
11   Step 4: calculate and return the final hash code:
12   hash = update_hash(hash, hashable0);
13   hash = update_hash(hash, shift1);
14   return hash;
15 }
```

**Figure 12.** A bijection from SSN strings to 32-bit values created via x86's `pext` instruction.

**RQ1:** How does the running time of the synthetic hash functions compare with the running time of functions typically used in standard libraries?
**RQ2:** How do the synthetic hash functions compare with standard functions in terms of collision count?
**RQ3:** What is the distribution of hash codes of the synthetic functions, and how does this distribution compare with the distribution of standard hash functions?
**RQ4:** How does the behavior of the hash functions vary across architectures considering x86 and aarch64, in terms of running time and code size?
**RQ5:** How does the keys' distribution impact the hash function's behavior?
**RQ6:** What is the asymptotic complexity of the synthesis of the optimized hash function?
**RQ7:** How does Sepe behave with hashing structures that stress out its worst-case scenarios, namely, structures indexed by the most-significant key bits?

These questions are evaluated in a commodity environment, which we describe in the rest of this section.
**Hardware:** Experiments evaluated in this section were performed on a multicore server containing two Intel(R) Xeon(R)

Silver 4210 2.20GHz processors, totaling 20 physical cores with 40 threads and 140 GB of RAM.

**Software:** The experimental setup runs on Linux Ubuntu Server 20.04 (kernel 5.4.0-174-generic). Programs are written in C++17 and are compiled with `gnu compiler` version 10.5 with `-O2` optimization flag. The choice of optimization level, in this case, is arbitrary: the results reported in this section remain the same if we compile the codes with clang `-O3`.

**Synthetic Hash Functions:** We have implemented four families of hash functions that use the strategies discussed in this paper. These functions gradually incorporate the three techniques mentioned in Section 3.2. In increasing order of complexity, the functions are:

- **Naive:** Applies an `xor`-based hash operation on all the key's bytes, in chunks of 8 bytes at a time. This function explores the fixed-length constraint to exercise the optimizations mentioned in Section 3.2.2.
- **OffXor:** Same as `naive`, but only loads the bytes that do not repeat among the keys. In other words, these functions implement the remove-common-characters strategy of Section 3.2.1.
- **Aes:** Same as `offxor`, but combines the bytes using an AES encode instruction (`aesenc` in x86 and `AESE` in aarch64) instead of `xor`. The instruction is slower, but, being a cryptographic function, shows better distribution properties.
- **Pext:** Same as `offxor`, but removes common bits, as explained in Section 3.2.3. After extracting the relevant bits, it shifts them to ensure that the entirety of the 64-bit range is used. The final hash is computed similarly to **OffXor**.

**Baseline Hash Functions:** We compare the synthetic functions with the following implementations of hash functions:

- **STL:** the default hash function used in the implementation of the Standard Template Library [12].
- **Abseil:** the hash function used in the Google Abseil Library [21].
- **FNV:** another hash function implemented in the Standard Template Library [11].
- **City:** the hash function from Google specialized for string keys [20].
- **Gpt:** the hash function generated by ChatGPT 3.5 using a specific prompt for each key. The prompts are available in this paper's repository (removed due to blind review)[3].
- **Gperf:** function synthesized by GNU perfect hash function generator (`gperf`) using 1000 random keys [23].

**Benchmarks:** Experiments are evaluated by a driver: a program that generates keys and operates on them, using some data structure. An "experiment" is a parameterization of the driver. The driver supports the following parameters:

- **Structure:** Operations can be performed on the following data structures:
  - `std::unordered_map`;
  - `std::unordered_set`;
  - `std::unordered_multimap`;
  - `std::unordered_multiset`.
- **Distribution:** Keys are generated either on ascending order, or following a uniform or normal distribution.
- **Affectation:** An "affectation" refers to the process of generating a key and subsequently performing an insertion, removal, or search operation on it. Experiments always run 10000 affectations.
- **Spread:** Experiments use either 500, 2000 or 10000 keys.
- **Mode:** The execution mode is either batched or interweaved. The batched execution mode runs all the operations in batches, starting with insertion, followed by searches and eliminations. The interweaved execution first runs 50% of the insertions, and then randomly interweaves searches, insertions and eliminations in the following way:
  1. Generate a key $k$, drawn from a given distribution;
  2. With probability $P_i$, inserts $k$ into the hash table;
  3. With probability $P_s$, searches $k$ into the hash table;
  4. With probability $1 - (P_i + P_s)$, removes $k$ from the hash table.

  Only three combinations of probabilities are allowed: $(P_i, P_s) \subset \{(0.7, 0.2), (0.6, 0.2), (0.4, 0.3)\}$.
- **Keys:** we work with eight different types of keys:
  - **SSN** (`\d{3}-\d{2}-\d{4}`);
  - **CPF** (`\d{3}\.\d{3}\.\d{3}-\d{2}`);
  - **MAC** (`([0-9a-fA-F]{2}-){5}[0-9a-fA-F]{2}`);
  - **IPv4** (`(([0-9]{3})\.){3}[0-9]{3}`);
  - **IPv6** (`([0-9a-f]{4}:){7}[0-9a-f]{4}`);
  - **INTS** (`[0-9]{100}`);
  - **URL1** constant URL with 23 characteres, plus the suffix `[a-z0-9]{20}\.html`;
  - **URL2** constant URL with 36 characteres, plus the suffix `[a-z0-9]{20}\.html`;

By varying these parameters, we obtain 144 experiments, each consisting of 10000 affectations. We calculate geometric means and Mann-Whitney P-tests using ten samples—no sample is discarded, e.g., to warm the driver up. "Aggregate results" consist of averages of every possible parameterization of the driver for a given hash function for a total of 144 experiments; each experiment sampled ten times; each sample consisting of 10000 affectations. Thus, each one of the ten hash functions considered in this section runs $10000 \times 10 \times 144$

[3] An example of prompt used to synthesize a hash for MAC addresses is: "For a hash function, assume that keys are MAC, always 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`. Assume you do not need to assert key format. Produce an optimized hash function for this specific case with an unrolled for loop, and also consider that the constant character is always the same and in the same position."

times in order to produce an aggregate result. While fiddling with this experimental setup, we have tried more affectations (up to one million); however, we have not observed any difference from the results that we report using only 10000.

### 4.1 RQ1 – Running Time

Section 3 describes code-generation techniques that seek to produce fast implementations of hash functions. This section evaluates the quality of these functions in terms of their running speed. We measure speed via two methodologies, which we call **B-Time** and **H-Time**. The latter considers only the running time of the hash function, whereas the former also considers the time it takes to actuate on the data structure, either via insertions, searches, or eliminations. Thus, while **H-Time** is only measuring the relative speed of hash functions, **B-Time** is also measuring the impact of the hash on the overall behavior of data structures such as `std::unordered_set` and `std::unordered_map`. Throughout this Section, we use geometric means to obtain a value representing an entire group of experiments.

***Discussion.*** Figure 13 shows aggregate results produced with eight different hash functions. Each box plot contains 1440 experiments. Table 1 shows absolute geometric mean values of these experiments. The `Gperf` function, which appears in Table 1, has been excluded from Figure 13 because it is two orders of magnitude slower than all other functions.

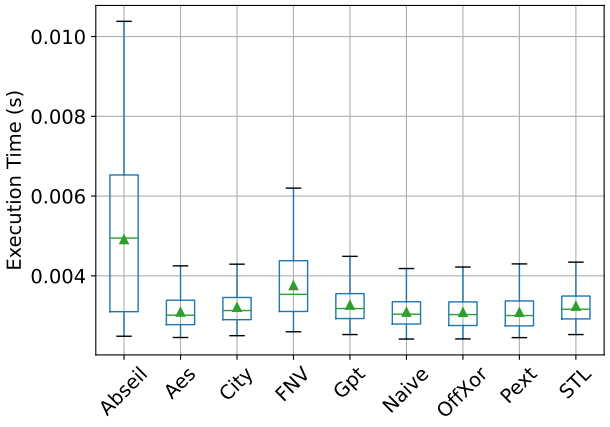



**Figure 13.** Boxplot of the execution time of different hash functions using 1440 experiments. Green triangles represent averages; middle lines, medians. Considering STL hashes, the slowest experiment runs on approximately 0.0044 seconds; the fastest runs in approximately 0.0026 seconds.

**B-Time Analysis:** The four synthetic hash functions (**Aes**, **Naive**, **OffXor**, and **Pext**) outperform the other implementations, including the standard STL hash. Mann–Whitney U tests show that there is a significant statistical difference between our synthetic functions and STL. Comparing synthesized functions, **OffXor** and **Naive** are statistically equivalent (p-value 0.51), and faster than **Pext**. `City` and STL are also statistically equivalent with p-value 0.44. `Gperf` is the outlier, running more slowly than the other functions. Feeding it with 1000 input keys causes it to generate a large lookup table, severely affecting its performance. If we take the geometric mean of all the aggregate results, then we improved performance over STL by 5.01%. If we examine each key type individually, we get performance improvements ranging from 3.78% to 9.5% for MAC/SSN and URL1, respectively. These numbers (**B-Time**) include not only hashing time, but also the time to update data structures; hence, these results are affected by three factors:

1. Hashing speed (**H-Time**), the time that the hash function takes to convert keys to 64-bit integers;
2. Collision rate, as more collisions slow down data retrieval;
3. Data structure implementation, which accounts for how the data structure is traversed, resized and updated with new keys.

**Table 1.** Performance comparison between different hash functions using a normal key distribution. **B-Time (*ms*):** Geometric mean execution time for benchmark, considering insertion, elimination, and search time. **H-Time (*ms*):** Total execution time of 10,000 activations of the hash function. **B-Coll:** Geometric mean collisions per bucket of the benchmark, considering 10,000 keys. **T-Coll:** Total number of collisions per hash function, considering 10,000 keys.

| Function | B-Time | H-Time | B-Coll | T-Coll |
|---|---|---|---|---|
| **Abseil** | 4.86 | 1.816 | 48.89 | 0 |
| **Aes** | 3.04 | 0.063 | 49.21 | 9 |
| **City** | 3.16 | 0.128 | 49.17 | 0 |
| **FNV** | 3.70 | 0.599 | 49.06 | 0 |
| **Gperf** | 12.71 | 0.045 | 55.87 | 55502 |
| **Gpt** | 3.22 | 0.171 | 49.47 | 7865 |
| **Naive** | 3.04 | 0.041 | 50.46 | 12 |
| **OffXor** | 3.03 | 0.037 | 50.67 | 12 |
| **Pext** | 3.03 | 0.050 | 49.42 | 0 |
| **STL** | 3.19 | 0.155 | 49.24 | 0 |

**H-Time Analysis:** In an STL Hash Map container, the time hashing a key is relatively small compared to all other operations and container-related logic. Analyzing **H-Time** allows for effectively observing the difference in purely hashing time. In **B-Time**, the difference between **OffXor** and STL is 5.01%, whereas the **H-Time** difference between the two is 418%. A similar pattern is seen for all of our synthesized functions. Comparing **Aes** with `City`, we get 203% performance improvements. Another observation is that `Gperf` has high

**B-Time** but low **H-Time** since the actual hashing operation is quite efficient. Still, collisions are frequent; hence, `Gperf`'s **B-Time** is high.

### 4.2 RQ2 – Collision Count

Collisions happen whenever two different strings yield the same hash code. The collision rate of a hash function is given by the ratio between collisions and hashes. Higher collision rates incur extra operations when interacting with the data. The collision rate depends on how the hash function distributes keys over hash codes and on how the data structure stores keys. In this section, we are concerned with the latter. In Section 4.3, we examine the hash function's distribution independently from the container. STL's hash-indexed containers divide data into buckets. Containers may place different keys inside the same bucket, even though they yield an entirely different hash. To count collisions in this case, we iterate over the buckets logging the number of keys inside the same bucket.

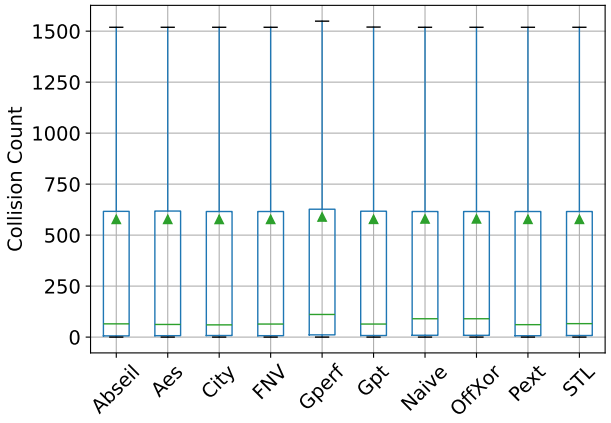



**Figure 14.** Collision count of the Hash Functions.

***Discussion.*** We distinguish two types of collisions: bucket and hash (total) collisions. In the former case, we are counting how often two keys ended up stored in the same bucket of a data structure. In the latter, we are counting how often two different keys generated the same 64-bit hash code.

**Bucket Collisions:** Figure 14 compares the collision count of the different hash functions. Column **B-Coll** in Table 1 reports the absolute number of bucket collisions. According to Mann-Whitney U tests, there is no meaningful statistical difference between our generated hash functions and STL's. Using STL containers, there is no statistically significant difference in bucket collisions (**B-Coll**) across hash functions, except for `GPerf`, which has a much higher collision rate.

**Total Collision Count:** Column **T-Coll** in Table 1 reports the real number of collisions generated by each Hash function. The `Gperf` function causes the highest number of collisions. This observation explains why it also has the highest **B-Time** despite having low **H-Time**. The high collision rate is due to the imperfect lookup table, which could be improved if the function is generated using every single possible input key. `Gpt`, in turn, despite having high **T-Coll**, has lower **B-Time** than `Gperf`. Although `Gpt` hashes cause 7865 collisions, 7857 of them are due to IPv4 keys. This concentration does not affect much `Gpt`'s geometric means. It is also worth mentioning that **Aes** has 9 **T-Coll** solely due to keys that are less than 16 bytes in size. Considering our synthetic hash functions, **Pext** always generates a bijection for key types that have equal or less than 64 relevant bits. For example, a 16 character integer in string format is a bijection with our **Pext** implementation. Even though we have key-types with 400 relevant bits (INTS), it still achieved zero **T-Coll** despite not being a bijection in these cases.

### 4.3 RQ3 – Hash Uniformity

The functions that Sepe generate are not cryptographic. This fact becomes apparent once we analyze the distribution of hashes over the domain of 64-bit integers. Some of our synthetic functions might end up representing the hash of a string that encodes a number as that number itself[4]. Such is the case, for instance, of **Pext**, when applied over social security numbers. Therefore, the distribution produced by **Pext** (or any other Sepe function) might be skewed if this function runs over non-uniform data. This section assesses the uniformity of our hash functions. To this end, we adopt the following methodology:

1. Generate 100,000 keys from one of the allowed distributions: incremental, uniform, or normal. The incremental distribution produces keys sorted by their ASCII values. For example, in case of SSNs, the keys would be, in ascending order: '`000-00-0000`', '`000-00-0001`', '`000-00-0002`', and so on.
2. Save all the hashes in a sorted vector $v$.
3. Build a histogram $h$ with the values stored in $v$.
4. Use the Chi-Square Goodness-of-Fit test to compare $h$ to a perfect distribution.

***Discussion.*** Table 2 summarizes the uniformity distribution analysis. The Chi-Test values are normalized with respect to the `STL` results. The closer the normalized values are to 0, the closer the distribution of hash values is to a perfect uniform distribution. Our synthetic hash functions are considerably less uniform than `STL`, `City`, `FNV`, and `Abseil`. Using the `pext` instruction to extract only the relevant bits

[4]Notice that functions like **Pext** embody the very nature of Kraska et al.'s learned index structures. Quoting Kraska et al.: "*If the goal is to build a highly-tuned system to store and query ranges of fixed-length records over a set of continuous integer keys one would not use a conventional B-Tree index over the keys since the key itself can be used as an offset, making it an $O(1)$ rather than $O(\ln n)$ operation to look-up any key.*" [15]

improves results for incremental key distributions. `Abseil`, `City`, `FNV`, and `STL` have similar uniformity between themselves and are always better than the synthesized versions. `Gpt` is only competitive with our functions for uniform key distributions; in particular, `Gpt` only achieved a good distribution for `MAC` address keys, where it achieved a statistically uniform distribution (p-value $>$ 0.05). If we exclude key types with fewer than 16 bytes (`CPF` and `SSN`), the Chi-test results for a normal distribution with **Aes** are 1.03; hence, very similar to `STL`. This behavior happens because **Aes** requires two 16 byte values; thus, we replicate the key to get a hash, which does not yield a good distribution.

**Table 2.** Hash uniformity distribution test according to different key distributions. All values are Chi-test normalized by the respective STL version. The closer to 0, the more uniform the distribution.

| Function | Inc | Normal | Uniform |
|---|---|---|---|
| **Abseil** | 1.02 | 0.99 | 1.01 |
| **Aes** | 1365.32 | 2506.61 | 2548.34 |
| **City** | 1.01 | 0.98 | 1.00 |
| **FNV** | 17.20 | 0.99 | 1.01 |
| **Gperf** | 2538.36 | 8638.28 | 8817.40 |
| **Gpt** | 2568.49 | 6205.49 | 3784.17 |
| **Naive** | 63.44 | 2531.19 | 3821.92 |
| **OffXor** | 59.29 | 2532.35 | 3822.70 |
| **Pext** | 7.63 | 2512.13 | 1303.73 |
| **STL** | 1.00 | 1.00 | 1.00 |

### 4.4 RQ4 – Architectural Impact

The standard library's hash functions run on many different computer architectures. Therefore, it is essential to consider more than one architecture when comparing them with our generated hash functions. This section analyzes running time and code size differences an aarch64 processor; hence, providing the reader with some perspective on how Sepe functions behave when running on an architecture different than x86. Results presented in the rest of this section were collected on a Jetson device with a quad-core Cortex-A57 CPU (1479 MHz), a 128-core NVIDIA Maxwell GPU, and 4 GB RAM. The machine does not have support for the `bext` instruction (the aarch64 equivalent to `pext`), and thus the **Pext** family of functions were left out.

***Discussion.*** The performance results for execution in an aarch64 machine are displayed in Figure 15. Comparing it to the x86 results in Figure 13, the slower hash functions (`Abseil` and `FNV`) have improved relatively to the others. Our synthetic functions, **Aes**, **Naive** and **OffXor**, remain the fastest. Overall, the tests have higher variability for different keys than their x86 equivalents. For all keys, Mann–Whitney U reveals no significant statistical difference between **Naive** and **OffXor**. Also for all keys, there is a significant difference between them and the other hash functions. **Aes** is sometimes equivalent to **Naive** and **OffXor**, and other times slower. These results show that using the specialization strategies presented in Section 3, we can implement more performant functions even on highly disparate hardware where we cannot use specialized hardware instructions.

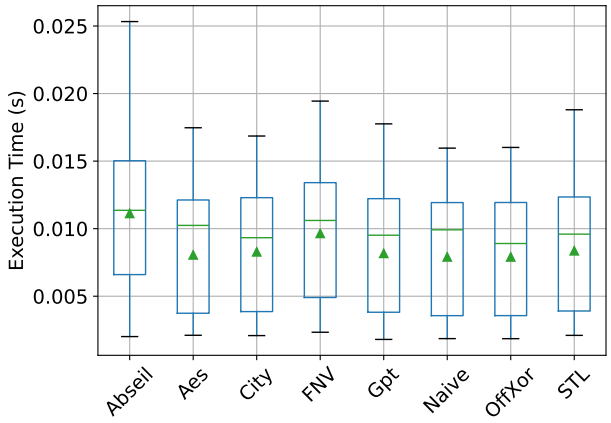



**Figure 15.** The execution time of different hash functions on an aarch64 architecture. Boxes aggregate the **B-Time** of 1440 experiments. Thus, these results include the time to affect the data structure, in addition to hashing time.

### 4.5 RQ5 – Key Distribution Impact

Compared to linear arrays that position subsequent data physically next to each other in the memory hierarchy, hash-indexed containers sacrifice data locality for fixed indexing time. As a result, subsequent keys may result in far-apart memory positions. Therefore, this RQ aims to analyze the impact of key distributions on performance.

***Discussion.*** Table 3 reveals that the key distribution impacts total execution time (**BT** – short for **Bucket Time**) and the total collision count (**TC**). Regarding collisions of the synthetic hash functions, only **Pext** achieved 0 collisions across all key distributions. Regarding bucket time, the Uniform key distribution yields the fastest execution times. Examining solely the hashing time, the difference between key distributions is equivalent to the standard deviation. Therefore, the difference in bucket time between distributions is due to the implementation of the STL container.

### 4.6 RQ6 – Synthesis Complexity

This paper proposes a technique to synthesize hash functions. General program synthesis is a hard problem: many instances of this problem are undecidable; whereas many others have exponential complexity [8, Ch.1]. However, this

**Table 3.** Performance comparison between different hash functions and Key distributions. **BT (*ms*):** Geometric mean execution time for benchmark, considering insertion, elimination, and search time. **TC:** Total number of collisions per hash function, considering 10,000 keys.

| Function | Inc | | Normal | | Uniform | |
|---|---|---|---|---|---|---|
| | BT | TC | BT | TC | BT | TC |
| **Abseil** | 4.86 | 0 | 4.86 | 0 | 3.11 | 0 |
| **Aes** | 5.67 | 6 | 3.04 | 9 | 1.35 | 9545 |
| **City** | 3.16 | 0 | 3.16 | 0 | 1.47 | 0 |
| **FNV** | 3.68 | 0 | 3.70 | 0 | 2.04 | 0 |
| **Gperf** | 62.46 | 54468 | 12.71 | 55502 | 30.59 | 79903 |
| **Gpt** | 6.68 | 7493 | 3.22 | 7865 | 1.49 | 17573 |
| **Naive** | 3.09 | 9 | 3.04 | 12 | 1.30 | 0 |
| **OffXor** | 3.06 | 9 | 3.03 | 12 | 1.30 | 0 |
| **Pext** | 3.00 | 0 | 3.03 | 0 | 1.32 | 0 |
| **STL** | 3.19 | 0 | 3.19 | 0 | 1.47 | 0 |

section presents empirical evidence that the code generation methodology described in this paper runs in linear time on the size of the largest key in the set of input examples. We shall demonstrate that this methodology handles efficiently keys with up to $2^{14}$ bytes.

***Discussion.*** Figure 16 shows the results of generating the **Pext** hashes for keys of size ranging from $2^4$ to $2^{14}$, where keys are sequences of digits without constant subsequences. Therefore, keys are entirely processed, as we cannot discard any part of the input using the skip table of Section 3.2.1. Figure 16 suggests that synthesis follows a linear asymptotic behavior. Indeed, the smallest Pearson correlation between synthesis time and problem size is 0.993 for **Aes**, whereas a coefficient of 1.0 indicates perfect linear correlation, and 0.0 indicates no correlation. This is within expectations since our code generation methods only require simple loops over the representation of a regular expression. The running time of synthesizing **Pext** functions grows faster—although still linearly—due to the cost of printing machine instructions, as the loop of each **Pext** function is fully unrolled.

### 4.7 RQ7 – Worst-Case Scenarios

Unordered STL containers determine the bucket where to store data by taking the modulo of the hash value with the current number of buckets. Therefore, even though Sepe functions might not produce well-distributed hash values, keys are still likely to be mapped into different buckets, as Example 4.1 will demonstrate.

**Example 4.1.** Sepe might generate a hash function for SSNs that uses the SSN itself as the hash value. Assuming 100 buckets, two successive SSNs will fall into different buckets, e.g., `123456789 % 100 = 89` and `123456790 % 100 = 90`.

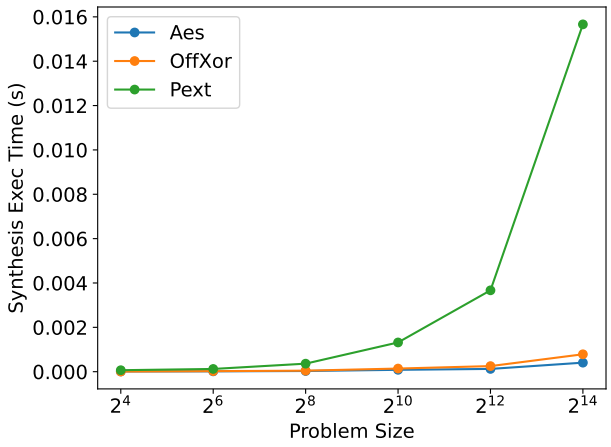



**Figure 16.** Synthesis time vs program size. The smallest value is 0.000069 seconds; the largest is approximately 0.016 seconds. The three curves show linear behavior (we remind the reader that the X-axis is logarithmic).

Consequently, despite the poor distribution of Sepe functions (as discussed in Section 4.3), they still exhibit low bucket collisions (as seen in Section 4.1). However, a different hashing structure could impose a significant burden on Sepe's functions. In this section, we explore two such scenarios: (i) containers that index buckets using the upper bits of the 64-bit hash value; and (ii) keys with fewer than 8 bytes[5].

***Discussion – Low-Mixing Containers.*** . To evaluate Sepe's behavior with a low-mixing container[6], we applied it to a version of an unordered map where buckets are determined by $u\%B$, where $u$ counts the most significant bits of the hash value, and $B$ is the number of buckets. For instance, if $u$ is 16, then all 64-bit hash values from 0 to $2^{48} - 1$ will be mapped to the same bucket. Under such conditions, Figure 17 shows the BC (Bucket Collisions) and Figure 18 shows the TC (True Collisions) of different hash functions.

In this scenario, two of Sepe's functions, **Naive** and **OffXor**, experience an increasing number of bucket and true collisions. In contrast, the **Pext** and **Aes** variants demonstrate greater resistance. **Pext**-based functions left-shift up to five bits with the `0x07` mask. However, **Pext**-based hashing still shows 7.1x more true collisions than the equivalent STL hash. These results corroborate the findings from Section 4.3, which indicate that Sepe functions have poor distribution. Therefore, Sepe should not be used with containers that discard bits from the hash value when indexing buckets.

[5]The second scenario cannot occur in practice, as Sepe defaults to the standard STL function for keys with fewer than eight bytes.
[6]In the context of this work, a "*low-mixing container*" is one where the bucket indexing process relies on only a small portion of the hash value (e.g., the most significant bits) rather than using the full hash value.

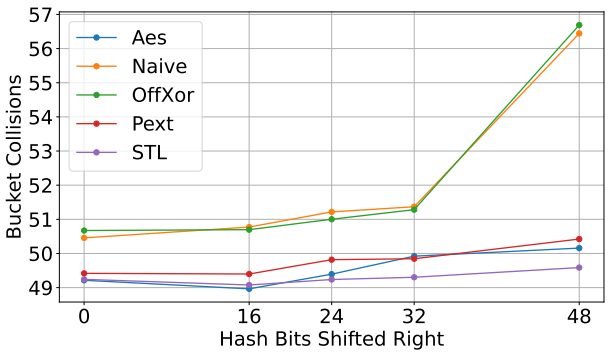



**Figure 17.** Aggregated Bucket Collisions (BC) when $64 - X$ most-significant bits are used to index buckets. The $X$ axis shows how many least-significant bits are discarded. For instance, $X = 48$ indicates that only the $64 - 48 = 16$ most significant bits were used to index buckets.

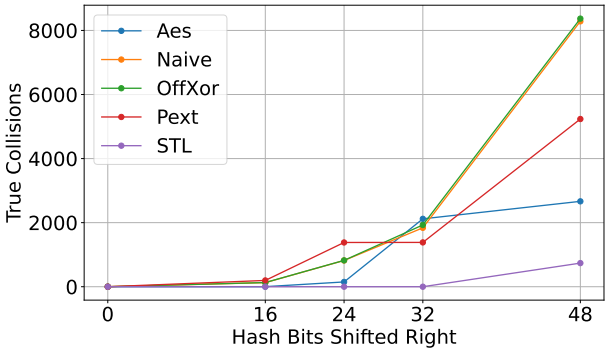



**Figure 18.** Impact of a low-mixing container on true collisions. The X-axis is defined as in Figure 17.

***Discussion – Four-Digit Integers.*** . The combination of a low-mixing container that uses most-significant bits with keys that contain less than eight bytes is the worst possible case for Sepe. To evaluate our functions under such constraints, Figure 18 reports bucket and true collisions using keys formed by four integer digits. If we use only the 32 most significant bits to choose the bucket, then **STL** has 18 bucket collisions and 5,786 true collisions, while **Pext** has 43 BCs and 9,999 TCs. However, if, instead, we use the 32 least significant bits, then these two functions behave similarly: **Pext** has 5,786 true collisions, the same as **STL**. Again, we emphasize that Sepe does not produce, by default, hash functions for keys that contain less than 64 bits.

***Discussion – Gradual Specialization.*** . This section illustrates one advantage of **Pext** (or **Aes**) over the other hash functions produced by Sepe. As seen in Figure 3 (Page 3), the sequence **Naive** → **OffXor** → **Pext** represent a gradual addition of constraints. Except for the low-mixing containers explored in this section, there is no performance benefit from using our most constrained function, **Pext**, over the simpler **OffXor** implementation.

## 5 Related Work

To our knowledge, there are no compilers that generate specialized code for hash functions tailored to specific input formats. However, numerous efforts have been made to specialize the hash indexing data structure (not the hash function) to allow fast storage and retrieval of elements from specific datasets. Additionally, the techniques presented in this paper are influenced by two significant subdomains of programming language research: program synthesis [16] and code specialization. In the latter case, our work fits into the general framework that Muth et al. [18] call "value specialization", where knowledge of the data determines the machine code that a compiler produces. This section provides an overview of this relevant literature, highlighting works that share similarities with our approach.

***Data Specialization.*** In 2018, Kraska et al. [15] introduced the notion of "Learned Index Structures (LIS)". A LIS is a combination of a data structure (e.g., a B-Tree or a Bloom Filter) plus a machine-learning guided model that predicts the position of keys within the structure. In contrast to Sepe, in the learned model there is no synthesis of hash functions. Rather, a neural network maps the input data (in Kraska et al.'s work a fixed-length string) to an interval within the data-structure where this key is guaranteed to be found. Kraska et al.'s indexing models have inspired much research; even allowing, for instance, the specialization of how hash tables are placed in memory [13]. Nevertheless, further studies have shown—in the words of Sabek et al. [22]—that: "*learned models can indeed outperform hash functions but only for certain data distributions and with a limited margin.*"

Within the domain of data structure specialization, the work that is the closest to Sepe is Hentschel et al. [10]'s Entropy-Learned Hashing. Hentschel et al. constrain hashing to only high-entropy parts of the data; that is, subsequences of the data whose contents tend to vary more. Additionally, Hentschel et al. produces hash tables and bloom filters that are specialized for these constrained hash functions (which Hentschel calls "*Cerebral Data Structures*" [9]). By observing the set of keys (which are assumed to have fixed length), they are able to determine subintervals of these byte sequences that present high and low entropy. Thus, they can hash only the high-entropy parts of a key, and determine the maximum capacity of the hash table beyond which they need to rehash the stored items into a new larger buffer. In contrast to Sepe, Hentschel et al. do not generate code for hash functions; rather, the beauty of their work resides on the fact that they can constrain any well-known hash function to only high entropy bits.

***Bitmask Specialization.*** There exist code-generation techniques that specialize cryptographic primitives using techniques similar to those that we have mentioned in Section 3.2.3. For instance, given a high-level description of a cryptographic routine, Usuba [17] and its extension, Tornado [2] produce a functionally equivalent implementation that uses bit masks to enhance its security guarantees against side-channel attacks. The code that Tornado generates is specialized for a given *masking order*, which is the number of masking operations that are applied onto sensitive data. Similarly, Lif [24] produces code that is secure against side-channels, specializing it to the set of inputs that are considered "public"; that is, non-sensitive.

## 6 Conclusion

This paper has described a code generation technique that produces hash functions specialized for particular types of data, namely sequences of bytes with fixed length, or common substrings or constant bits. These synthetic functions show very good performance when compared to standard hash functions available in libraries such as C++ STL or Google Abseil, sometimes showing speedups of almost 50x (as seen in column **H-Time** in Table 1). Nevertheless, although the functions run efficiently, we believe that there are still many directions along which the approach proposed in this paper can be improved. In particular, our techniques specialize hashing, but not storage and retrieval. Thus, we see room for generating code for specialized data structures. Furthermore, the the hash operations we synthesize lack properties such as pre-image and collision resistance, which are desirable in cryptographic hash functions. We leave the synthesis of efficient and secure cryptographic hash functions as future work.

***Software:*** The ideas presented in this work have been implemented as public software, released under the GPL 3.0 license, and available at https://github.com/lac-dcc/sepe.

## Acknowledgement

This project is supported by a grant that Google Inc. has generously donated to UFMG's Compilers Lab. Fernando Pereira also acknowledges support from the following research agencies: CNPq (grant 406377/2018-9), FAPEMIG (grant PPM-00333-18), and CAPES (Edital PrInt). The specialization in Figure 3 is based on a reddit comment by `u/moon-chilled`.

## References

[1] Austin Appleby. 2008. MurmurHash, First Announcement. https://tanjent.livejournal.com/756623.html. Accessed: 2024-03-21.

[2] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. 2020. Tornado: Automatic Generation of Probing-Secure Masked Bitsliced Implementations. In *EUROCRYPT* (Zagreb, Croatia). Springer-Verlag, Berlin, Heidelberg, 311–341. https://doi.org/10.1007/978-3-030-45727-3_11

[3] Ivan Bjerre Damgård. 1989. A design principle for hash functions. In *CRYPTO* (Santa Barbara, California, USA). Springer-Verlag, Berlin, Heidelberg, 416–427.

[4] Jordan Earls and Sazzad Hissain Khan. 2011. Which hashing algorithm is best for uniqueness and speed? https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed. Accessed: 2024-03-21.

[5] César Estébanez, Yago Saez, Gustavo Recio, and Pedro Isasi. 2014. Performance of the most common non-cryptographic hashfunctions. *Softw. Pract. Exper.* 44, 6 (jun 2014), 681–698. https://doi.org/10.1002/spe.2179

[6] Agner Fog. 2022. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs. https://www.agner.org/optimize/instruction_tables.pdf. Last updated on 2022-11-04.

[7] Glenn Fowler, Landon Curt Noll, and Kiem-Phong Vo. 1994. FNV hash history. http://www.isthe.com/chongo/tech/comp/fnv/index.html. Accessed: 2024-03-21.

[8] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Found. Trends Program. Lang.* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

[9] Brian Hentschel. 2022. *Cerebral Data Structures: Integrating Context into Data Structure Design and Implementation.* Ph. D. Dissertation. Harvard University.

[10] Brian Hentschel, Utku Sirin, and Stratos Idreos. 2022. Entropy-Learned Hashing: Constant Time Hashing with Controllable Uniformity. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1640–1654. https://doi.org/10.1145/3514221.3517894

[11] Jakub Jelinek. 2024. Implementation of the FNV hash in the Standard Template Library. https://github.com/gcc-mirror/gcc/blob/ee0717da1eb5dc5d17dcd0b35c88c99281385280. File /libstdc++-v3/libsupc++/hash_bytes.cc, Line 123, Accessed: 2024-03-21.

[12] Jakub Jelinek. 2024. Implementation of the murmur hash in the Standard Template Library. https://github.com/gcc-mirror/gcc/blob/ee0717da1eb5dc5d17dcd0b35c88c99281385280. File /libstdc++-v3/libsupc++/hash_bytes.cc, Line 138, Accessed: 2024-03-21.

[13] Aarati Kakaraparthy, Jignesh M. Patel, Brian P. Kroth, and Kwanghyun Park. 2022. VIP hashing: adapting to skew in popularity of data on the fly. *Proc. VLDB Endow.* 15, 10 (jun 2022), 1978–1990. https://doi.org/10.14778/3547305.3547306

[14] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News* 43, 3S (jun 2015), 158–169. https://doi.org/10.1145/2872887.2750392

[15] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. arXiv:1712.01208 [cs.DB]

[16] Zohar Manna and Richard J. Waldinger. 1971. Toward automatic program synthesis. *Commun. ACM* 14, 3 (mar 1971), 151–165. https://doi.org/10.1145/362566.362568

[17] Darius Mercadier and Pierre-Évariste Dagand. 2019. Usuba: high-throughput and constant-time ciphers, by construction. In *PLDI* (Phoenix, AZ, USA). Association for Computing Machinery, New York, NY, USA, 157–173. https://doi.org/10.1145/3314221.3314636

[18] Robert Muth, Scott A. Watterson, and Saumya K. Debray. 2000. Code Specialization Based on Value Profiles. In *SAS*. Springer-Verlag, Berlin, Heidelberg, 340–359.

[19] M. Naor and M. Yung. 1989. Universal one-way hash functions and their cryptographic applications. In *STOC* (Seattle, Washington, USA). Association for Computing Machinery, New York, NY, USA, 33–43. https://doi.org/10.1145/73007.73011

[20] Geoff Pike and Jyrki Alakuijala. 2024. Implementation of the CityHash64 hash function in the Abseil Library. https://github.com/abseil/abseil-cpp/blob/master/absl/hash/internal/city.cc. Accessed: 2024-03-21.
[21] Nafi Rouf. 2024. Implementation of Low-Level Hash in the Abseil Library. https://github.com/abseil/abseil-cpp/blob/master/absl/hash/internal/low_level_hash.cc. Accessed: 2024-03-21.
[22] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, and Tim Kraska. 2021. When Are Learned Models Better Than Hash Functions? arXiv:2107.01464 [cs.DB]
[23] Douglas C. Schmidt, Keith Bostic, and Bruno Haible. 2007. The GNU perfect hash function generator. https://github.com/rurban/gperf. Accessed: 2024-03-21.
[24] Luigi Soares, Michael Canesche, and Fernando Magno Quintão Pereira. 2023. Side-channel Elimination via Partial Control-flow Linearization. *ACM Trans. Program. Lang. Syst.* 45, 2, Article 13 (jun 2023), 43 pages. https://doi.org/10.1145/3594736

# A Further Analyses

This section explores two extra research questions, using the same experimental methodology discussed in Section 4. The two new research questions that we explore are:

**RQ8:** What is the asymptotic complexity of hashing, considering our synthetic functions and the different baseline functions?

**RQ9:** How does the hash function impact the running-time behavior of different data structures?

## A.1 RQ8 – Hash Functions Complexity

All hash functions generated using the techniques outlined in Section 3 process inputs one word at a time without backtracking. The number of operations performed per word remains constant. Consequently, we expect that the asymptotic complexity of these functions will be linear relative to the length of the key. This section presents empirical evidence supporting this expectation.

***Discussion.*** Figure 19 shows **Pext**'s runtime behavior alongside the other standard hashing functions, as the input size increases in powers of two. The inputs are the same used in Section 4.6. All the hash functions exhibit linear behavior, with the smallest Pearson correlation between problem size and hashing speed being 0.9979 for **Pext**. We omit the running time of the other synthetic functions to avoid cluttering Figure 19, as these functions are strictly faster than **Pext**.

## A.2 RQ9 – Data Structure Impact

Data structures manipulate and use the generated hashes differently. Furthermore, they may be more or less sensitive to variations in the hash functions' properties. Thus, this section examines in more detail the impact of the hash functions on four data structures from the C++ Standard Template Library: `unordered_map`, `unordered_set`, `unordered_multimap` and `unordered_mul- tiset`.

***Discussion.*** For the sake of space, we only show in this section the relative performance between the different data

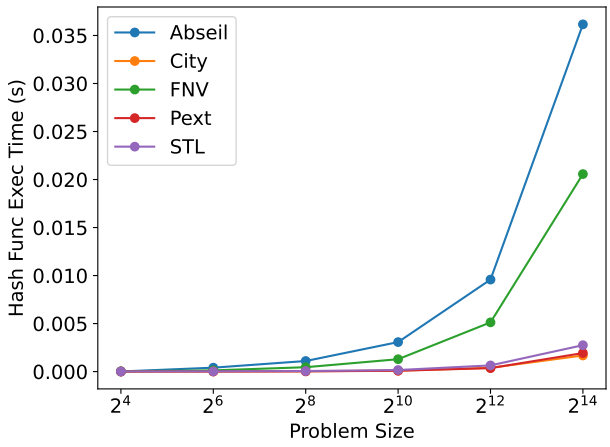



**Figure 19.** Hash Functions execution time with increasing problem sizes. The smallest value is 0.000059 seconds.

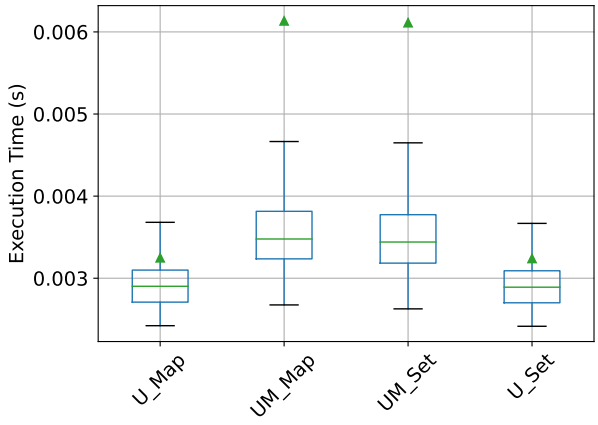



**Figure 20.** Boxplot of the execution time grouped by Container. U represents Unordered, and UM represents Unordered Map.

structures. Figure 20 summarizes these results. The figure shows that the MultiMap and MultiSet data structures have higher execution times than Map and Set. The *Multi* variants allow multiple elements mapping to the same key, which requires another layer of indirection to operate on the data. Although we do not show it in Figure 20, we have not observed that any particular synthetic hash function benefits more from specific data structures. Our conclusion is that the relative performance of these synthetic hash functions does not depend on the data structure that is used to organize data, assuming standard implementations of maps and sets. In other words, the same trends reported in Figures 13 or Table 1 remain true once we analyze them across separate data structures.