# Regular and almost universal hashing: an efficient implementation

D. Ivanchykhin[1], S. Ignatchenko[1], D. Lemire[2]

[1] *OLogN Technologies AG, Triesen, Liechtenstein*
[2] *LICEF Research Center, TELUQ, Montreal, QC, Canada*

## SUMMARY

Random hashing can provide guarantees regarding the performance of data structures such as hash tables—even in an adversarial setting. Many existing families of hash functions are universal: given two data objects, the probability that they have the same hash value is low given that we pick hash functions at random. However, universality fails to ensure that all hash functions are well behaved. We might further require regularity: when picking data objects at random they should have a low probability of having the same hash value, for any fixed hash function. We present the efficient implementation of a family of non-cryptographic hash functions (PM+) offering good running times, good memory usage as well as distinguishing theoretical guarantees: almost universality and component-wise regularity. On a variety of platforms, our implementations are comparable to the state of the art in performance. On recent Intel processors, PM+ achieves a speed of 4.7 bytes per cycle for 32-bit outputs and 3.3 bytes per cycle for 64-bit outputs. We review vectorization through SIMD instructions (e.g., AVX2) and optimizations for superscalar execution.

KEY WORDS: performance; measurement; random hashing, universal hashing, non-cryptographic hashing, avalanche effect

## 1. INTRODUCTION

Hashing is ubiquitous in software. For example, most programming languages support hash tables, either directly, or via libraries. However, while many computer science textbooks consider random hashing, most software libraries use deterministic (i.e., non-random) hashing.

A hash function maps data objects, such as strings, to fixed-length values (e.g., 64-bit integers). We often expect data objects to be mapped evenly over the possible hash values. Moreover, we expect collisions to be unlikely: there is a *collision* when two objects are mapped to the same hash value. Hash tables can only be expected to offer constant time query performance when collisions are infrequent.

When hashing is deterministic, we pick one hash function once and for all. It is even customary for this hash function to be common knowledge. Moreover, the objects being hashed are typically not random, they could even be provided by an adversary. Hence, an adversary can cause many collisions that could translate into a denial-of-service (DoS) attack [1, 2, 3].

In random hashing, we regularly pick a new hash function at random from a family of hash functions. With such random hashing, we can bound the collision probability between two objects, even if the objects are chosen by an adversary. By using random hashing, programmers might produce more secure software and avoid DoS attacks. Maybe for this reason, major languages

*Correspondence to: Daniel Lemire, LICEF Research Center, TELUQ, Université du Québec, 5800 Saint-Denis, Office 1105, Montreal (Quebec), H2S 3L5 Canada. Email: lemire@gmail.com

Contract/grant sponsor: Natural Sciences and Engineering Research Council of Canada; contract/grant number: 261437

have adopted random hashing. Python (as of version 3.3), Ruby (as of version 1.9) and Perl (as of version 5.18) use random hashing by default [4]. Unfortunately, these languages fail to offer a theoretical guarantee regarding collision probabilities.

A family of hash functions having a low collision probability given two objects chosen by an adversary may contain terrible hash functions: e.g., hash functions mapping all objects to the same value (see § 3). In practice, such bad hash functions can be reused over long periods of time: e.g., a Java program expects the hash value of a given object to remain the same for the duration of the program. An adversary could detect that a bad hash function has been selected and launch a successful attack [5, 6]. Thus we should ensure that all hash functions in a family can be safely used. We believe that regularity might help in this regard: a function is regular if all hash values are equally likely given that we pick the data objects at random. Indeed, regularity implies that the probability that two objects picked at random have the same hash value is low. We generalize regularity to component-wise regularity by considering same-length strings that differ by one character. We want to minimize the probability that any two such strings have the same hash value.

In the following sections, we describe a practical approach toward generating non-cryptographic hash functions for arbitrary objects that have low probabilities of collision when picking hash values at random (i.e., universality) and low probabilities when picking data objects at random (i.e., regularity).

It is not difficult to construct such families: e.g., the family $h(x) = ax \bmod p$ for $p$ prime and an integer $a$ picked randomly in $[1, p)$ is regular and almost universal over integers in $[0, p)$. However, our objective is to implement a practical solution in software that provides competitive speed. In particular, we wish to hash arbitrarily long strings of bytes, not just integers in $[0, p)$, and in doing so, we wish to make the best possible use of current processors. To achieve our goals, we use affine functions over a suitable finite field to hash blocks of machine words. We choose the finite field so as to make the operations efficient. We then use a tree construction to hash long strings. The resulting family of hash functions is called PM+. We establish universality and regularity properties.

We run performance experiments using a variety of platforms such as Intel, AMD and ARM processors, with both Microsoft and GNU compilers. On recent Intel processors, our proposal (PM+) hashes long strings at a rate of 4.7 bytes per cycle for 32-bit outputs 3.3 bytes per cycle for 64-bit outputs. Generally, our functions match the speed of state-of-the-art hash functions: they are as fast as MurmurHash [7] on shorter segments and comparable to VHASH [8] on longer data segments. However, PM+ has distinguishing theoretical guarantees: MurmurHash is not claimed to be universal and VHASH offers poor regularity.

We also present the optimization methods used to achieve the good performance of PM+ (see § 6). For example, we optimize the computation of a scalar product in a finite field. Such optimizations might be useful for a variety of functions.

## 2. RANDOM HASHING

Good hash functions are such that hash values are random in some sense. To achieve randomness, we pick a hash function $h : X \to Y$ at random in a family of hash functions. (For our notation, see Table I.) For practical reasons, we assume $Y$ to be an interval of integers starting at zero, e.g., $Y = [0, 2^{32})$.

A family is *uniform* if $P(h(x) = c) = 1/|Y|$ for any constant $c \in Y$ and any $x \in X$ where $|Y|$ is the cardinality of $Y$ [9]. Uniformity is a weak property: let $\mathcal{H}$ be the family of hash functions of the form $h(x) = c$ for some $c \in Y$, then $\mathcal{H}$ is uniform even though each hash function maps all values to the same constant $c$. If $P(h(x) = c) \leq \epsilon$ for all $x$ and all $c$, then we say that it is $\epsilon$-almost uniform.

A family is *universal* [10, 11] if the probability of a collision is no larger than if the hash values were random: $P(h(x) = h(x')) \leq 1/|Y|$ for any $x, x' \in X$ such that $x \neq x'$. It is *$\epsilon$-almost universal* [12] (also written $\epsilon$-AU) if the probability of a collision is bounded by $\epsilon < 1$. Informally, we say that a family has *good universality* if it is $\epsilon$-almost universal for a small $\epsilon$. Universality does not imply uniformity.

Table I. Notation

| | |
|---|---|
| $h, f, g, f_i$ | hash functions |
| $\mathcal{H}, \mathcal{F}, \mathcal{G}$ | families of hash function |
| $X, Y, Z$ | sets of integer values |
| $x \in X$ | value $x$ in $X$ |
| $\lvert X \rvert$ | cardinality of the set $X$ |
| $p$ | prime number |
| $\kappa$ | parameter of the PM+ family |
| $s$ | string |
| $s_i$ | value of the $i^{\text{th}}$ character |
| $m$ | number of characters in a block |
| $n$ | number of bits |
| $L$ | number of levels |

For example, consider Carter-Wegman polynomial hashing [10]. It is given by the family of functions $h : Y^m \to Y$ of the form $h(s_1, s_2, \ldots, s_m) = \sum_{i=1}^{m} t^{n-i} s_i$ where the computation is executed over a finite field of cardinality $|Y|$. The value $t$ is picked in $Y$. It is $(m-1)/|Y|$-almost universal but not uniform (even when $m = 1$) [13].

### 2.1. $\Delta$-universality

A family is $\Delta$-*universal* ($\Delta$U) [14] if $P(h(x) = h(x') + c \bmod |Y|) \leq 1/|Y|$ for any constant $c$ and any $x, x' \in X$ such that $x \neq x'$. Moreover, it is $\epsilon$-almost $\Delta$-universal ($\epsilon$-A$\Delta$U or $\epsilon$-ADU) if $P(h(x) = h(x') + c \bmod |Y|) \leq \epsilon$ for any constant $c$ and any $x, x' \in X$ such that $x \neq x'$. $\Delta$-universality implies universality but not uniformity.

It is necessary sometimes to take an $L$-bit hash value and hash it down to $[0, m)$. It is common to simply apply a modulo operation to achieve the desired result. As long as the original hash family is $\Delta$-universal, the modulo operation is a sound approach as the next lemma shows.

*Lemma 1*
(Dai and Krovetz [8, Lemma 4]) Given an $\epsilon$-almost $\Delta$-universal family $\mathcal{H}$ of hash functions $h : X \to Y$, the family of hash functions $\{h(x) \bmod M \mid h \in \mathcal{H}\}$ from $X$ to $[0, M)$ is $\left\lceil \frac{2|Y|-1}{M} \right\rceil \times \epsilon$-almost $\Delta$-universal. Moreover, if $M$ divides $|Y|$, then the result is an $\frac{|Y|}{M} \times \epsilon$-almost $\Delta$-universal family.

Lemma 1 encourages us to seek low collision probabilities if we expect users to routinely rely on only a few bits of the hash result. For example, let us consider Bernstein's [15] state-of-the-art 128-bit Poly1305 family. It is $\epsilon$-almost $\Delta$-universal with $\epsilon = 8\lceil L/16 \rceil / 2^{106}$ where $L$ is the size of the input in bytes. For all but very large values of $L$, $\epsilon$ is very small. However, if we reduce Poly1305 to 32 bits by a modulo operation, the result is $\epsilon$-almost $\Delta$-universal with $\epsilon = 8\lceil L/16 \rceil / 2^{10}$ by Lemma 1. In other words, it might be possible to find two 2040-byte strings that always collide on their first 32 bits when using the Poly1305 hash family. Though this is not a problem in a cryptographic setting where a collision requires all 128 bits to be equal, it can be more of a concern with hash tables.

### 2.2. Strong universality

A family is *strongly universal* [16] (or pairwise independent) if given 2 distinct values $x, x' \in X$, their hash values are independent: $P(h(x) = y \wedge h(x') = y') = \frac{1}{|Y|^2}$ for any hash values $y, y' \in Y$. Strong universality implies uniformity, $\Delta$-universality and universality. Intuitively, strong universality means that given $h(x) = y$, we cannot tell anything about the value of $h(x')$ when $x' \neq x$.

When $M$ divides $|Y|$, if $\mathcal{H}$ is strongly universal then so is $\{h(x) \bmod M \mid h \in \mathcal{H}\}$. To put it another way, if $\mathcal{H}$ is strongly universal with $|Y|$ a power of two, then selecting the first few bits preserves strong universality.

The MULTILINEAR hash family is a strongly universal family [10, 17]. It is the addition of a constant value with the scalar product between random values (sometimes called *keys*) and the input data represented as vectors components $(s_1, \ldots, s_m)$, where operations and values are over a finite (or Galois) field: $h(s1, s_2, \ldots, s_m) = a_0 + \sum_{i=1}^{m} a_i s_i$. The hash function $h$ is specified by the randomly generated values $a_0, a_1, a_2, \ldots, a_m$. In practice, we often pick a *finite field* $\mathbb{F}_p$ having prime cardinality ($p$). Computations in $\mathbb{F}_p$ are easily represented using ordinary integer arithmetic on a computer: values are integers in $[0, p)$, whereas additions and multiplications are followed by a modulo operation ($x \times_{\mathbb{F}_p} y = xy \bmod p$ and $x +_{\mathbb{F}_p} y = x + y \bmod p$).

There are weak versions of strong universality that are stronger than $\epsilon$-almost universality. E.g., we say that the family is $\epsilon$-almost strongly universal if it is uniform and if

$$P\left(h(x) = y \mid h(x') = y'\right) \leq \epsilon$$

for any distinct $x, x'$. It is $\epsilon$-variationally universal if it is uniform and if

$$\sum_{y \in Y} \left| P\left(h(x) = y | h(x') = c\right) - 1/|Y| \right| \leq 2\epsilon$$

for all distinct $x, x'$ and for any $c$ [18]. There are also stronger versions of strong universality such as $k$-wise independence [19, 13]. For example, Zobrist hashing [20, 21, 22] is 3-wise independent (and therefore strongly universal). It is defined as follows. Consider the family $\mathcal{F}$ of all possible functions $X \to Y$. There are $|Y|^{|X|}$ such functions, so that they can each be represented using $|X| \log |Y|$ bits. Given strings of characters from $X$ of length up to $N$, pick $N$ functions from $\mathcal{F}$, $f_1, f_2, \ldots, f_N$ using $N|X| \log |Y|$ bits. The hash function is given by $s \to f_1(s_1) \veebar \cdots \veebar f_{|s|}(s_{|s|})$ where $\veebar$ is the bitwise exclusive or. Though Zobrist hashing offers strong universality, it may require a lot of memory. Setting aside the issue of cache misses, current x64 processors cannot sustain more than two memory loads per cycle which puts an upper bound on the speed of Zobrist hashing. In an exhaustive experimental evaluation of hash-table performance, Richter et al. [23] found that Zobrist hashing produces a low throughput. Consequently, the authors declare it to be "less attractive in practice" than its strong randomness properties would suggest.

## 2.3. *Composition and concatenation of families*

There are two common ways to combine families of hash functions: composition ($h(x) = g \circ f(x) \equiv g(f(x))$) and concatenation ($h(x) = (g(x), f(x))$ or $h = (g, f)$).[†] For completeness, we review important results found elsewhere [12]. Under composition uniformity is preserved, but universality tends to degrade linearly in the sense that the bounds on the collision probability add up (see Lemma 2).

*Lemma 2*
Let $\mathcal{F}$ and $\mathcal{G}$ be $\epsilon_{\mathcal{F}}$-almost and $\epsilon_{\mathcal{G}}$-almost universal families of hash functions $f : X \to Y$ and $g : Y \to Z$. Let $\mathcal{H}$ be the family of hash functions $h : X \to Z$ made of the functions $h = g \circ f$ where $f \in \mathcal{F}$ and $g \in \mathcal{G}$.

- Then $\mathcal{H}$ is $\epsilon_{\mathcal{F}} + \epsilon_{\mathcal{G}}$-almost universal.
- Moreover, if $\mathcal{G}$ is $\epsilon_{\mathcal{G}}$-almost $\Delta$-universal, then $\mathcal{H}$ is $\epsilon_{\mathcal{F}} + \epsilon_{\mathcal{G}}$-almost $\Delta$-universal.
- If $\mathcal{G}$ is uniform then so is $\mathcal{H}$.

*Lemma 3*
Universality is preserved under concatenation. That is, let $\mathcal{F}$ be a family of hash functions $f : X \to Y$, then the family made of the concatenations $(f, f) : X \times X \to Y \times Y$ is $\epsilon$-almost universal if $\mathcal{F}$ is $\epsilon$-almost universal.

[†]Some authors might refer to a concatenation as a cartesian product or a juxtaposition.

## 3. REGULARITY

Though we can require families of hash functions to have desirable properties such as uniformity or universality, we also want individual hash functions to have reasonably good properties. For example, what if a family contains the hash function $h(x) = c$ for some constant $c$? This particular hash function is certainly not desirable! In fact, it is the worst possible hash function for a hash table. Yet we can find many such hash functions in a family that is otherwise strongly universal. Indeed, Dietzfelbinger [9] proposed a strongly universal family made of the hash functions

$$h_{A,B}(x) = \left(Ax + B \bmod 2^K\right) \div 2^{n-1}$$

with integers $A, B \in [0, 2^K)$. It is strongly universal over the domain of integers $x \in [0, 2^n)$. However, one out of $2^K$ hash functions has $A = 0$. That is, if you pick a hash function at random, the probability that you have a constant function ($h_{0,B}(x) = B \div 2^{L-1}$) is $1/2^K$. Though this probability might be vanishingly small, many of the other hash functions have also poor distributions of hash values. For example, if one picks $A = 2^{K-1}$, then any two hash values ($h_{A,B}(x)$ and $h_{A,B}(x')$) may only differ by one bit, at most. Letting $A$ be odd also does not solve the problem: e.g., $A = 1, B = 0$ gives the hash function $x \div 2^{n-1}$ which is either 1 or 0.

Such weak hash functions are a security risk [5, 6]. Thus, we require as much as possible that hash functions be *regular* [24, 25].

*Definition 1*
A hash function $h : X \to Y$ is regular if for every $y \in Y$, we have that $|\{x \in X : (h(x) = y)\}| \leq \lceil |X|/|Y| \rceil$. Further, a family $\mathcal{H}$ of hash functions is regular if every $h \in \mathcal{H}$ is regular.

We stress that this regularity property applies to individual hash functions.[‡] However, we can still give a probabilistic interpretation to regularity: if we pick any two values $x_1$ and $x_2$ at random, the probability that they collide $h(x_1) = h(x_2)$ should be minimal ($|Y|/|X|$) if $h$ is regular.

As an example, consider the case where $X = Y = \{0, 1\}$. There are only two regular hash functions $h : X \to Y$. The first one is the identity function ($h_I(0) = 0, h_I(1) = 1$) and the second one is the negation function ($h_N(0) = 1, h_N(1) = 0$). The family $\{h_I, h_N\}$ is uniform and universal: the collision probability between distinct values is zero.

More generally, whenever $X = Y$, a function $h : X \to Y$ is regular if and only if it is a permutation. This observation suffices to show that it is not possible to have strong universality and regularity in general. Indeed, suppose that $X = Y$, then all hash functions $h$ must be permutations. Meanwhile, strong universality means that given that we know the hash value $y$ of the element $x$ (i.e., $h(x) = y$), we still known nothing about the hash value of $x'$ for $x' \neq x$. But if $h$ is a permutation, we know that the hash values differ ($h(x') \neq h(x)$)—contradicting strong universality. More formally, if $h$ is a permutation, we have that $h(x) \neq h(x')$ for $x \neq x'$ which implies that $P(h(x) = h(x')) = 0$ whereas $P(h(x) = h(x')) = 1/|Y|$ is required by strong universality. Thus, while we can have both universality and regularity, we cannot have both strong universality and regularity.

The next two lemmas state that regularity is preserved under composition and concatenation.

*Lemma 4*
(Composition) Assume that $|Y|$ divides $|X|$ and $|Z|$ divides $|X|$. Let $f : X \to Y$ and $g : Y \to Z$ be regular hash functions then $f \circ g : X \to Z$ is also regular.

*Lemma 5*
(Concatenation) Let $f : X_1 \to Y_1$ and $g : X_2 \to Y_2$ be regular hash functions then the function $h : X_1 \times X_2 \to Y_1 \times Y_2$ defined by $h(x_1, x_2) = (f(x_1), g(x_2))$ is also regular.

[‡]In contrast, Fleischmann et al. [26] used the term $\epsilon$-almost regular to indicate that a family is almost uniform: $P(h(x) = y) \leq \epsilon$ for all $x$ and $y$ given that $h$ is picked in $\mathcal{H}$.

### *3.1. Component-wise regularity*

We can also consider stronger forms of regularity. Consider hash functions of the form $f : X_1 \times X_2 \times \cdots \times X_m \to X$, then the hash function is *component-wise regular* if we can arbitrarily fix all input components but one and still generate all hash values fairly, that is

$$|\{x \in X_i : (h(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_m) = y)\}| \\ \leq \lceil |X_i|/|Y| \rceil$$

for any $i$, any $y$ and any values

$$x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_m.$$

Intuitively, component-wise regularity ensures that if we pick two same-length strings at random that differ in only one pre-determined component, the collision probability is minimized. By inspection, component-wise regularity is preserved under composition and concatenation.

Of particular interest are the hash functions of the form $f : X \times X \times \cdots \times X \to X$. In this case, component-wise regularity implies that the restriction of the function to one component (setting all other components to constants) is a permutation of $X$. Clearly, if $f_1$ and $f_2$ are two such functions then their concatenation $((f_1, f_2))$ is also component-wise regular, and if $g$ is itself component-wise regular, then the composition of $g$ with the concatenation $(f_1, f_2)$, written $g(f_1, f_2)$, is again component-wise regular. The following lemma formalizes this result.

*Lemma 6*
Let $f_i : X^m \to X$ be component-wise regular hash functions for $i = 1, \ldots, m$. Let $g : X^m \to X$ be a component-wise regular hash function. Then the composition and concatenation $g(f_1, f_2, \ldots, f_m)$ is component-wise regular.

### *3.2. $K$-regularity*

Regularity is not always reasonable: for example, regularity implies that $|Y|$ divides $|X|$. Naturally, we can weaken the definition of regularity: we say that hash function is *$K$-regular* if $h(x) = y$ is true for at most $K\lceil |Y|/|X| \rceil$ values $x$ given a fixed $y$. A 1-regular function is simply regular. We define component-wise $K$-regularity in the obvious manner. Our objective is to achieve $K$-regularity for a small value of $K$.

Though regularity is preserved under composition, $K$-regularity is not. Indeed, consider the 4-regular function $h : \{0, 1, \ldots, 2^n - 1\} \to \{0, 1, \ldots, 2^n - 1\}$ given by $h(x) = \lfloor x/4 \rfloor$. Composing $h$ with itself, we get a 16-regular function $h'(x) = h(h(x)) = \lfloor x/16 \rfloor$. The example illustrates the following lemma.

*Lemma 7*
Let $f : X \to Y$ and $g : Y \to Z$ be $K_1$-regular and $K_2$-regular hash functions then $f \circ g : X \to Z$ is $(K_1 \times K_2)$-regular if $|Y|$ divides $|X|$ and $|Y|$ divides $|Z|$.

*Proof*
Given $z \in Z$, we have that $g(y) = z$ is true for at most $K_2|Z|/|Y|$ values $y \in Y$. In turn, we have that $h(x) = y$ for at most $K_2|Y|/|X|$ values $x \in X$. Thus, given $z \in Z$, we have that $g(f(x)) = z$ is true for at most $K_2|Z|/|Y| \times K_2|Y|/|X| = K_1K_2|Z|/|X|$, completing the proof. □

Thus, in general, regularity degrades exponentially under composition. In contrast, universality degrades only linearly under composition: an $K_1/2^n$-almost universal family composed with another $K_2/2^n$-almost universal is at least $(K_1 + K_2)/2^n$-almost universal (by Lemma 2).

To achieve strong regularity, a good strategy might be to only compose functions that are 1-regular. Of course, we might still need to reduce the hash values to a useful range. Thankfully, regularity merely degrades to 2-regularity under modulo operations.

*Lemma 8*
Given a regular hash function $h : X \to Y$, we have that the hash function $h'$ defined by

$$h'(x) = h(x) \bmod M$$

for $M \leq |Y|$ is regular if $M$ divides $|Y|$ and 2-regular otherwise.

*Proof*
Pick any value $y \in [0, M)$. If $h'(x) = y$ then $h(x) = y + kM \bmod |Y|$ for some integer $k$ such that $kM \in [0, |Y|)$. There are $\left\lceil \frac{|Y|}{M} \right\rceil$ such values for $k$:

$$0, 1, \ldots, \left\lceil \frac{|Y|}{M} \right\rceil M - 1.$$

Because $h$ is regular, the equation $h'(x) = y$ has at most $\left\lceil \frac{|Y|}{M} \right\rceil \times \frac{|X|}{|Y|}$ solutions for $x$. To determine the $K$-regularity, we have to divide this result by $\frac{|X|}{M}$: $K = \left\lceil \frac{|Y|}{M} \right\rceil \times \frac{|X|}{|Y|} \times \frac{M}{|X|} = \left\lceil \frac{|Y|}{M} \right\rceil \times \frac{M}{|Y|} \leq \left( \frac{|Y|}{M} + \frac{M-1}{M} \right) \frac{M}{|Y|} = 1 + \frac{M-1}{|Y|}$. We have that $K \leq 2$ in general and $K = 1$ if $M$ divides $|Y|$. $\square$

## 4. A TREE-BASED CONSTRUCTION FOR UNIVERSALITY AND REGULARITY

We want to address long objects, such as variable-length strings, while maintaining the collision probability as low as possible. Though we could get strong universality with the MULTILINEAR hash family, we would need as many random bits as there are bits in our longest object. What if some of our objects use gigabytes or more? It may simply not be practical to generate and store so many random bits. To alleviate this problem, it is common to use a tree-based approach [27, 16, 28]. Such an approach allows us to hash very long strings using hash functions that require only a few kilobytes for their description. It is a standard approach so we present it succinctly.

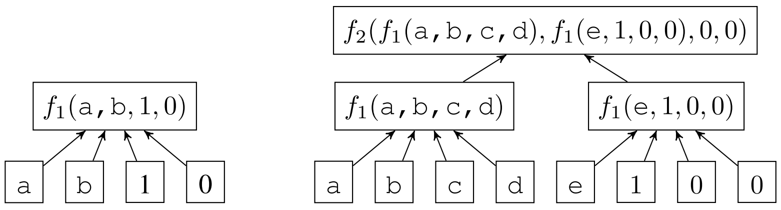
Let $X$ be a set of integers values containing at least the values 0 and 1. We pick $L$ hash functions $f_1, f_2, \ldots : X^m \to X$ from a family $\mathcal{H}$ (e.g., MULTILINEAR family from § 2.2). Take any string $s$ made of $N$ character values from $X$ and let $L = \lceil \log_m N + 1 \rceil$. Append the value 1 at the end of the string $s$ to create the new string $\sigma$ [8, 29, 30, 31]. If $L = 1$, simply return $f_1(\sigma)$ with the convention that we pad $\sigma$ with enough zeros that it has length $m$. If $L > 1$, split $\sigma$ into $m^{L-1}$ segments of length $m$ each (except for the last segment that might need padding) and apply $f_L$ on each segment: the result is a new string of length at most $m^{L-1}$. Split again the result into $m^{L-2}$ segments of length at most $m$ each and apply $f_{L-1}$ on each segment. Continue until a single value remains. See Fig. 1 for an illustration and see Algorithm 1 for the corresponding pseudocode.

If the family $\mathcal{H}$ is $\epsilon$-almost universal, then the family formed by the tree-based construction has to be $L\epsilon$-almost universal. Indeed, it can be viewed as the composition of $(f_L, f_L, \ldots)$, $(f_{L-1}, f_{L-1}, \ldots)$, $\ldots$, $f_1$. Each one is $\epsilon$-almost universal by Lemma 3. And the composition of $L$ $\epsilon$-almost universal functions is $L\epsilon$-almost universal by Lemma 2.

Moreover, because $f_1$ is $\epsilon$-almost $\Delta$-universal, and the composition of $(f_L, f_L, \ldots)$, $(f_{L-1}, f_{L-1}, \ldots)$, $\ldots$, $(f_2, f_2, \ldots)$ is $(L-1)\epsilon$-almost universal, we have that the final construction must be $L\epsilon$-almost $\Delta$-universal. Further, as long as $\mathcal{H}$ is a uniform family, the construction is uniform. Moreover, by Lemma 6, we have that if the family $\mathcal{H}$ is regular and component-wise regular, then the result from the construction is regular component-wise regular as well.

To achieve almost $\Delta$-universality, it is only required that the last of the hash functions applied come from an almost $\Delta$-universal family. Thus it is possible to use families with weaker universalities (e.g., merely $\epsilon$-almost universal) as part of the tree-based construction, while still offering almost $\Delta$-universality in the end. However, with regularity, we cannot as easily substitute potentially weaker hash families: we require that all hash functions being composed be regular.

For clarity, we described Algorithm 1 in such a way that the first level is computed entirely as a first step (using $f_1$), followed by a second pass at the second level (using $f_2$) and so on. This



$f_1(\texttt{a},\texttt{b},1,0)$
a
b
1
0
$f_2(f_1(\texttt{a},\texttt{b},\texttt{c},\texttt{d}), f_1(\texttt{e},1,0,0),0,0)$
$f_1(\texttt{a},\texttt{b},\texttt{c},\texttt{d})$
$f_1(\texttt{e},1,0,0)$
a
b
c
d
e
1
0
0

(a) String `ab` (b) String `abcde`

Figure 1. Simplified tree-based algorithm hashing the strings `ab` and `abcde` using hash functions $f_1, f_2, \ldots : X^4 \to X$

**Algorithm 1** Tree-based algorithm

```
Require: Set of integer values $X$ containing at least the values 0 and 1. {E.g., set of all 32-bit integers.}
Require: $L$ hash functions $f_1, f_2, \ldots, f_L$ of the form $X^m \to X$ for $m > 1$, picked independently from a
    family $\mathcal{H}$ that is uniform and $\epsilon$-almost $\Delta$-universal.
1: input: string $s$ made of $N$ character values from $X$ with $1 \leq N \leq m^L - 1$. {That is, $s \in X^N$ and
   $|s| = N$.}
2: Let $\sigma$ be the string of length $N+1$ that we get by appending the value 1 at the end of the string $s$. {We
   have that $|\sigma| \leq m^L$.}
3: $j \leftarrow 1$
4: while $\sigma$ contains more than one character value ($|\sigma| > 1$) do
5:     while the length $|\sigma|$ is not a multiple of $m$, append a zero to $\sigma$.
6:     $\sigma \leftarrow f_j(\sigma_1, \ldots, \sigma_m), f_j(\sigma_{m+1}, \ldots, \sigma_{2m}), \ldots,$
               $f_j(\sigma_{|\sigma|-m+1}, \ldots, \sigma_{|\sigma|})$
7:     $j \leftarrow j + 1$
8: end while
9: return the sole character value of $\sigma$ as the hash value of $s$
```

approach requires allocating dynamically a possibly large amount of memory. We compute the same result using a bounded and small amount of memory [29]: no more than $m(L-1)$ values from $X$. We first hash the first $m$ characters of the string that has been extended with an extra 1. The result is written at the first location in the second level. We repeat with the next $m$ elements. (Cases where we have fewer than $m$ characters left are also handled efficiently, avoiding copies and explicit zero-padding.) Once we have $m$ hash values stored in the second level, we hash them and store the result in the third level. After each chunk of $m$ characters is hashed, we push its hash value to a higher level. Once we are done hashing the input, we complete the computation.

Almost all data objects in modern computing can be represented as a string of bytes (8-bit words) so we assume that we accept strings of bytes for complete generality. Yet on 32-bit or 64-bit processors, it is not always desirable to process the inputs byte-by-byte: it is more natural and faster to process the data using 32-bit or 64-bit machine words. So our set of characters $X$ is made of all 32-bit or all 64-bit values. When appending the string with a value of 1 as in Algorithm 1, we actually pad with a 1-byte and zeros to the nearest machine word boundary. In software, we avoid creating a new extended string with padded bytes—as it would be inefficient. Instead we just compute the final machine word and use an optimized code path.

As pointed out by Halevi and Hugo [32], there is a downside to the tree-based approach: the universality degrades linearly with the height of the tree. We could solve this problem by hashing all but the last level of the tree to a larger domain (e.g., one of cardinality $L|X|$), as long as we could maintain regularity. Or, instead, we could use a two-level approach where only the first level uses MULTILINEAR, while the second level uses a polynomial hash family: VHASH described in Appendix F uses a similar approach [8]. We would need to ensure that we have good regularity in both levels. However, we can alleviate this degraded universality problem by using a tree of small height. That is, if we choose the family $\mathcal{H}$ of hash functions $h : X^m \to X$ with a relatively large integer $m$, we may never require a tall tree (e.g., one with more than $\approx 8$ levels). In this manner, the tree-based approach may still meet our goals by ensuring component-wise regularity while still achieving good universality.

## 5. UNIVERSALITY AND REGULARITY WITH PM+

To implement Algorithm 1, we need to select a family of hash functions $\mathcal{H}$. The MULTILINEAR family (see § 2.2) might fit our needs in mathematical terms since it is strongly universal: $h(s) = a_0 + \sum_{i=1}^{m} a_i s_i \bmod p$ for $p$ prime. By picking keys $a_1, a_2, \ldots$ as integers in $[1, p)$, we get a component-wise regular and almost universal family. However, the resulting hash family depends crucially on the choice of a prime number $p$. In related work, authors chose prime numbers smaller than a power of two [30, 8, 22] such as Mersenne primes or pseudo-Mersenne primes (primes of the form $2^n - k$ where $k$ is much smaller than $2^n$ in absolute value [33]). Such prime numbers enable fast modulo reduction algorithms. For example, $p = 2^{61} - 1$ is a Mersenne prime. Given a 64-bit integer $x$, we can compute $x \bmod p$ by first computing $(x \bmod 2^{61}) + (x \div 2^{61})$ and then subtracting $p$ from $x$ if $x$ exceeds $p$.

Of course, we do not hash strings of numbers in $[0, p)$ for $p$ prime, instead we hash strings of numbers in $[0, 2^n)$. Choosing $p < 2^n$ is not a problem to get almost universality [31, Section 4]. However, it makes it more difficult to achieve regularity. To illustrate the problem, consider once more the family $h(x) = ax \bmod p$ for $p$ prime and an integer $a$ picked randomly in $[1, p)$. This family is regular for inputs in $[0, p)$. Suppose however that $x \in [0, 2^n)$ for $2^n > p$, then the result is at most 2-regular. Because regularity degrades exponentially with composition, if we use a 2-regular function at each level in the a tree-based setting, the final result might only be $2^L$-regular for trees of height $L$. However, the problem goes away if we pick $2^n < p$, as $h(x) = ax \bmod p$ is then regular once more.

Hence, our selection of prime numbers $p$ is based on *two* requirements:

1. for a number $x$ that fits a single processor word, $x \bmod p$ should be equal to $x$, thus making it easy to achieve regularity, and

2. reduction modulo $p$ of numbers that do not fit to a single word should be expressed in terms of computationally inexpensive operations. In practice, this may be achieved by choosing $p$ close to a power of two matching the processor word size (such as $2^{64}$).

Thus, we use minimal primes that are greater than any number that fits in a single processor word, that is, for a 32-bit platform, $p = 2^{32} + 15$, and for 64-bit platform, $p = 2^{64} + 13$ (see Table II). We call primes of a form $2^n + k$ where $k$ is small PSEUDO+MERSENNE *primes* by analogy with pseudo-Mersenne primes. Table II gives several such primes, e.g., $2^{64} + 13$.

The idea of using PSEUDO+MERSENNE primes for universal hashing is not new [15, 32].

- Our approach is similar to Multidimensional-Modular-Hashing (MMH) [32]. The MMH authors use $p = 2^{32} + 15$ for $n = 32$. They build their hash family on multilinear functions of the form $h(s) = (\sum_{i=1}^{m} a_i s_i \bmod 2^{2n}) \bmod p$ (as opposed to $h(s) = \sum_{i=1}^{m} a_i s_i \bmod p$). That is, they use only two $n$-bit words to compute the sum although more than $2n$ bits are required (e.g., 3 words) to compute the exact sum. They prove that their speed optimization only degrades the universality slightly (by a factor of 2). However, they also degrade the regularity. Because the regularity degrades exponentially with composition in the worst case (see § 3), we prefer to avoid non-regular functions for a tree-based construction. Moreover, we are able to produce fast code to compute the exact sum (see Appendix B). Since we benchmark our contributed functions against a family faster than MMH (VHASH [30, 8]), we do not consider MMH further.

- Our approach is also related to Bernstein's [15] cryptographic Poly1305 function that uses $p = 2^{130} - 5$ to generate 128-bit hash values. Bernstein reports choosing $p = 2^{130} - 5$ instead of a value closer to $2^{128}$ for computational convenience. Though it is possible that larger

Table II. Smallest primes larger than a power of two [34]

| Power of two | Smallest prime |
|---|---|
| $2^8$ | $2^8+1$ |
| $2^{16}$ | $2^{16}+1$ |
| $2^{32}$ | $2^{32}+15$ |
| $2^{64}$ | $2^{64}+13$ |
| $2^{128}$ | $2^{128}+51$ |

primes than the ones we choose could allow further speed optimizations, it may also degrade the universality slightly. Thus we do not consider the possibility further.

From this family of prime numbers, we define the PM+-MULTILINEAR family of hash functions.

*Definition 2*
Let $p$ be a prime, $X=[0,p)$ and $m$ a positive integer. Let $2^n$ be the largest power of two smaller than $p$. The PSEUDO+MERSENNE-MULTILINEAR family (or PM+-MULTILINEAR) is the set of functions from $X^m$ to $X$ of the form

$$f(s)\equiv f(s_1,\ldots,s_m)=\left(b+\sum_{i=1}^{m}a_is_i\right) \bmod p \qquad (1)$$

where $b$ is an integer in $[0,2^n)$ whereas $a_1,\ldots,a_m$ are non-zero integers in $(0,p-\kappa)$ for some integer $\kappa\geq 0$.

Observe that integers subject to additions and multiplications modulo $p$ for $p$ prime form a *finite field* $\mathbb{F}_p$. Thus we have that $ax \bmod p = a'x \bmod p$ implies $a=a'$ unless $x=0$ since $x$ is invertible (in $\mathbb{F}_p$). We have that PM+-MULTILINEAR is component-wise regular: we can solve the equation $f(s)=y$ for $s_i$ with exactly one value: $s_i=a_i^{-1}(y-a_1s_1-a_2s_2-\cdots-a_{i-1}s_{i-1}-a_{i+1}s_{i+1}-a_ms_m)$ in $\mathbb{F}_p$. That is, it suffices to require that the parameters $a_1,\ldots,a_m$ are non-zero to get regularity.

We can also show $\epsilon$-almost $\Delta$-universality as follows. Consider the equation

$$\left(b+\sum_{i=1}^{m}a_is_i\right)-\left(b+\sum_{i=1}^{m}a_is'_i\right) \bmod p = y$$

for some $y$ in $[0,p)$ for two distinct strings $s$ and $s'$. We have that $s_r\neq s'_r$ for some index $r$. Thus, fixing all other values, we can solve for exactly one value $a_r$ such that the equality holds. When picking the hash function at random, $a_r$ can have one of $p-\kappa-1$ different values (all integers in $[1,p-\kappa)$), thus the equation holds with probability at most $1/(p-1-\kappa)$.

Similarly, we can show that PM+-MULTILINEAR is $1/2^n$-almost uniform. Indeed, consider the equation

$$\left(b+\sum_{i=1}^{m}a_is_i\right) \bmod p = y$$

for some $y$ in $[0,p)$. Fixing the $a_i$'s, the $s_i$'s and $y$, there is exactly one value $b\in[0,p)$ solving this equation. Yet we have $2^n$ possible values for $b$, hence the result.

We have the following lemma.

*Lemma 9*
The family PM+-MULTILINEAR is $1/(p-1-\kappa)$-almost $\Delta$-universal, $1/2^n$-almost uniform and component-wise regular.

Though it may seem that the parameter $\kappa$ is superfluous as setting $\kappa = 0$ optimizes universality, we shall see that restricting the range of values with $\kappa > 0$ can ease computations. Similarly, it may seem wasteful to pick $b \in [0, 2^n)$ instead of picking it in $[0, p)$, but this is again done for computational convenience.

In what follows, we call PM+ the use Algorithm 1 with the hash family PM+-MULTILINEAR. The result is a hash family that is $L/(p-1-\kappa)$-almost $\Delta$-universal, uniform and component-wise regular over strings of length up to $m^L - 1$.

Naturally, we want the resulting hash values to fit in a more convenient range than $[0, p)$. So, we compute $h(s) \bmod 2^n$. We have that $\left\lceil \frac{p}{2^n} \right\rceil = 2$. As per Lemmas 1 and 8, the result is $3L/(p-1-\kappa)$-almost $\Delta$-universal and 2-regular.[§]

## 6. AN EFFICIENT IMPLEMENTATION OF PM+

The functions in the PM+-MULTILINEAR family make use of a modulo operation. On most modern processors, division and modulo operations are computationally expensive in comparison to addition, or even multiplication. Thankfully, equation 1 suggests a single modulo operation after a series of multiplications and additions that reduces the number of modulo operations to one per $m$ multiplications [32, 31], where $m$ is a parameter of our family. We can furthermore tune PM+ by optimizing the scalar product computations (see § 6.1) and replacing expensive modulo operation by a specialized routine (see § 6.2).

### *6.1. Scalar product computation*

Our data inputs are strings of $n$-bit characters. Two cases are important: $2^n = 2^{32}$ (particularly for 32-bit architectures) and $2^n = 2^{64}$ (mostly for 64-bit architectures). Where applicable we refer to them separately as PM+-MULTILINEAR32 (or PM+32 in the tree-based version) and PM+-MULTILINEAR64 (or PM+64 in the tree-based version).

Consider the scalar product between keys and components $\sum_i a_i s_i$. Recall that we pick the values $a_i$ in $(0, p-\kappa)$. As long as we choose $\kappa$ large enough so that $p - \kappa \leq 2^n$, we have that $a_i$ is also a machine-sized word (e.g., 64 bits on a 64-bit platform). For long data segments, we expect most of the running time to be due to the first level of the tree, and mostly due to the computation of the sum $\sum_{i=1}^{m} a_i s_i$. We describe our fast implementation of such computations on modern superscalar processors in Appendix B.

Though we hash strings of machine-sized words (so that $s_i \in [0, 2^n)$), in a tree-based setting (see Algorithm 1), we can no longer assume that $s_i$ fits in a single word—beyond the first level. Two words are required in general. The $s_i$'s are in $[0, 2^{32} + 15)$ for PM+32 and in $[0, 2^{64} + 13)$ for PM+64 at all but the first level in Algorithm 1. (We could reduce the hash values so that they fit in a single word, but it would degrade the regularity and universality of the result.) For speed and convenience, we still want the result of the multiplication to fit in two words. That is, we want that $a_i s_i \in [0, 2^{2n})$ or, more specifically,

$$(p - \kappa)(p - 1) < 2^{2n}.$$

For this purpose, we set $\kappa = 24$ for PM+64. That is, we pick the $a_i$'s in $(0, 2^{64} + 13 - 24) = (0, 2^{64} - 11)$. For PM+32, we set $\kappa = 28$ and pick $a_i$'s in $(0, 2^{32} + 15 - 28) = (0, 2^{64} - 13)$. See Table III for the parameters and Table IV for the properties of the resulting hash families.

For both the PM+32 and PM+64 cases, we use a maximum of 8 levels ($L = 8$). Yet we are unlikely to use that many levels in practice: e.g., if we assume that inputs fit in four gigabytes, then 4 levels are sufficient.

[§]Since $\frac{2p-1}{2^n} < 3$.

Table III. Parameters used by PM+

| Word size | $2^n$ | $p$ | $\kappa$ | $m$ | $L$ |
|---|---|---|---|---|---|
| 32 bits | $2^{32}$ | $2^{32}+15$ | 28 | 128 | 8 |
| 64 bits | $2^{64}$ | $2^{64}+13$ | 24 | 128 | 8 |

Table IV. Properties of Algorithm 1 applied with PM+-MULTILINEAR in a tree-based setting (PM+). String lengths are expressed in machine words (32 bits or 64 bits).

| Name | Word size | Hash interval | max. string length | universality | regularity |
|---|---|---|---|---|---|
| PM+32 | 32 bits | $[0, 2^{32})$ | $(2^{56}-1)$ words | $\frac{12}{2^{31}-7}$-A$\Delta$U | component-wise 2-regular |
| PM+64 | 64 bits | $[0, 2^{64})$ | $(2^{56}-1)$ words | $\frac{12}{2^{63}-6}$-A$\Delta$U | component-wise 2-regular |

## 6.2. *Efficient modulo reduction*

We have insured that the result of our multiplications fit in two words. Halevi and Krawczyk [32, Section 3.1] have derived an efficient modulo reduction in such cases. However, the sum of our multiplications requires more than two words since, unlike Halevi and Krawczyk, we compute an exact sum. Thus we need to derive an efficient routine to apply the modulo operation on three input words. We have that

$$S = \left(b + \sum_{i=1}^{m} a_i s_i\right) \leq (2^n - 1) + m(2^{2n} - 1)$$

Because we choose $m = 128$, we have that the result fits into three words $(w_0, w_1, w_2)$ (either 32-bit or 64-bit words) with a small value stored in the most significant word $w_2$ (no larger than $m$).

The modulo reduction uses the equalities (modulo $2^n + k$): $2^n = -k$ and $(2^n)^2 = (-k)^2 = k^2$. In our case, $k = 15$ or $k = 13$ depending on whether we use a 32-bit or 64-bit platform. Then, for any number $S$ that fits in 3 words $w_0, w_1, w_2$ (i.e., it is smaller than $2^{3n}$ where $n = 32$ or $n = 64$), we have

$$\begin{aligned} S &\equiv w_0 + w_1 \times 2^n + w_2 \times (2^n)^2 \\ &\equiv w_0 - k \times w_1 + k^2 \times w_2 \end{aligned}$$

modulo $2^n + k$. Let $u_0 = (k \times w_1) \bmod 2^n$ and $u_1 = (k \times w_1) \div 2^n$, then $k \times w_1 = u_1 \times 2^n + u_0$. By substitution, we further obtain

$$\begin{aligned} S &\equiv w_0 + k^2 \times w_2 - 2^n \times u_1 - u_0 \\ &\equiv w_0 + k^2 \times w_2 + k \times u_1 - u_0 \\ &\equiv (w_0 + k^2 \times w_2 + k \times u_1) + (2^n + k - u_0) \end{aligned}$$

modulo $2^n + k$.

We have thus reduced $S$ to a number smaller than $2^{2n}$ (modulo $2^n + k$) which fits in two $n$-bit words. We can therefore write $S \equiv v_0 + 2^n v_1 (\bmod 2^n + k)$ where $v_0, v_1 \in [0, 2^n)$ are easily computed.

We can also bound our representation of $S$ as follows:

- $w_0 \leq 2^n - 1$;
- $k^2 w_2 \leq k^2 m \leq 15^2 \times 128 = 28800$
- $k \times u_1 = k \times (k \times w_1 \div 2^n) \leq k \times ((k \times 2^n - 1) \div 2^n) \leq k(k-1) \leq 210$;
- $2^n + k - u_0 \leq 2^n + k - 1 \leq 2^n + 14$.

Thus we have a bound of $2 \times 2^n + 29023$. It follows that $v_1 \leq 2$.

To reduce $S$ to a number in $[0, p)$ requires branching (see Algorithm 2). The algorithm works as follows:

- If $k \times v_1 \leq v_0$, we exploit the fact that, modulo $2^n + k$, we have $2^n \times v_1 = -k \times v_1$ to return $v_0 - k \times v_1$.

  To accelerate this case in software, we can use the fact that $v_0 \geq 2k \Rightarrow k \times v_1 \leq v_0$. Since $v_0 \geq 2k$ is common and faster than checking that $k \times v_1 \leq v_0$, it is worth introducing an extra branch.

- If $k \times v_1 > v_0$, then we know that $v_1 > 0$. If $v_1 = 1$, then $v_0 < k$ and we can return $v_0 + 2^n$ without any reduction. Otherwise we have that $v_1 = 2$. In such a case, we use the fact that $2 \times 2^n = 2^n - k \bmod (2^n + k)$ to write $v_0 + 2 \times 2^n$ as $2^n - k + v_0$. This value is smaller than $2^n + k$ since $v_0 < 2k$.

**Algorithm 2** Reduction algorithm: find the integer $z \in [0, 2^n + k)$ such that $v_1 + 2^n v_2 = z \bmod 2^n + k$.

```
1: input: an integer $v_1 \in [0, 2^n)$ and an integer $v_2 \in \{0, 1, 2\}$ {Represents $v_1 + 2^n v_2$.}
2: if $k \times v_1 \leq v_0$ then
3:     return $v_0 - k \times v_1$          {Can use $v_0 \geq 2k \Rightarrow k \times v_1 \leq v_0$ to accelerate the check}
4: end if
5: if $v_1 = 1$ then
6:     return $v_0 + 2^n$
7: end if
8: return $v_0 - k$                          {$v_1 = 2$ in this case}
```

## 7. ACHIEVING THE AVALANCHE EFFECT

It is often viewed as desirable that a small change in the input should lead to a large change in the hash value. For example, we often check whether hash functions satisfy the *avalanche effect*: changing a single bit of the input should flip roughly half the bits of the output [35].

To improve our hash functions in such respect, we add an extra step to further *mix* the output bits. We borrowed these procedures from MurmurHash [7]. For PM+64 the step in C is

```
z = z ^ (z >> 33);
z = z * 0xc4ceb9fe1a85ec53;
z = z ^ (z >> 33);
```

and for PM+32 it is

```
z = z ^ (z >> 13);
z = z * 0xab3be54f;
z = z ^ (z >> 16);
```

where $\veebar$ is the bitwise exclusive or and $z$ is an unsigned integer of a respective size (32 bit for PM+32 and 64 bit for PM+64). These transformations are invertible for all integers that fit a single word and, therefore, they do not affect universality and regularity.

## 8. EXPERIMENTS

We implemented PM+ for the x64, x86 and ARM platforms in C++. On the x86 platform, we use the SSE2 instruction set for best speed. We make our software freely available under an open source license.¶

¶http://sourceforge.net/projects/hasher2/

To test the practical fitness of the PM+ schema, we chose the SMHasher [7] framework. It provides a variety of performance tests as well as several statistical tests. For comparison purposes, we used the same framework to test other hashes commonly used in industry.

1. The hash functions used in the C++ standard (`std`) library.
2. The hash functions used in the Boost library, a widely used C++ library.
3. MurmurHash 3A for 32-bit platforms and MurmurHash 3F for 64-bit platforms: a popular family of hash functions used by major projects such Apache Hadoop and Apache Cassandra.
4. VHASH [8, 30] (see Appendix F), one of the fastest hash families on 64-bit processors.
5. SipHash [36]: the 64-bit family hash functions used by the Python language.

Of course, there are many more fast hash functions (e.g., xxHash, CityHash [37, 38], SpookyHash [39], FarmHash [40], CLHASH [41] and tabulation-based or Zobrist hashing [20, 21, 22]). For a recent review of non-cryptographic hash functions, we refer the interested reader to Ahmad and Younis [42], Estébanez et al. [43] or Thorup [44]. We leave a more detailed comparison to future work.

We performed extra steps to ensure these functions work inside the SMHasher testing environment:

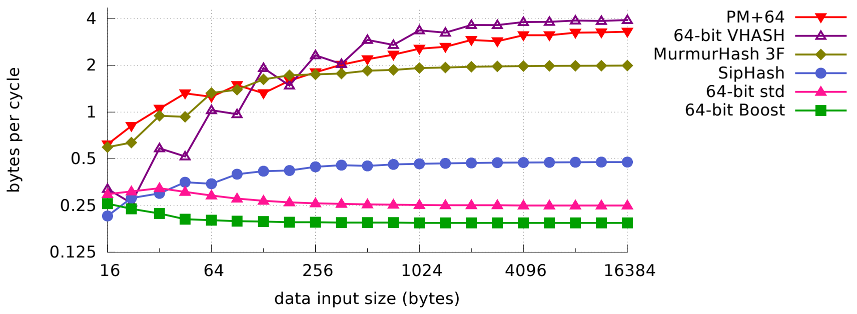
**Standard library** To hash an arbitrary length data segment we used the library function `hash <string>`: it takes a `string` object as a single parameter. The C++11 standard does not specify the implementation so it is vendor and even version specific. In practice, the hash value generated occupies 32 bits on 32-bit platforms and 64 bits on 64-bit platforms.

In the context of the SMHasher testing environment, we must first create a string object based on the data segment and its length to use this function. To exclude the time used for the creation of the `std::string` object, we added a separate method that does just the creation of the object itself, and nothing else. Hence, we were able to estimate the time required to create the object and deduct it from the whole processing time. We have observed that time spent on object preparation was roughly 10 % of the total processing time.

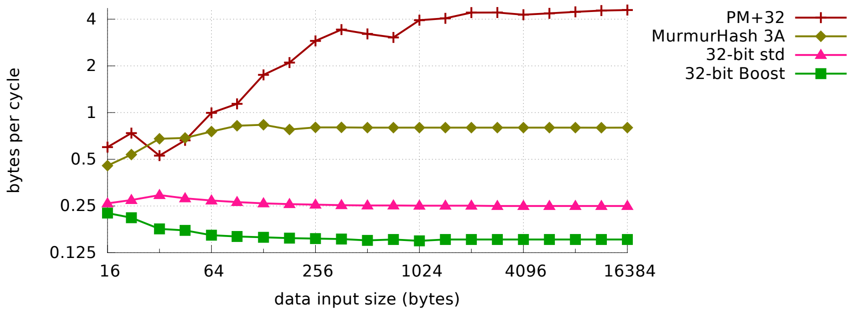
**Boost** Boost is a well regarded C++ library and it is likely that its hash functions are in common use. We tested the `hash_range( char*, char*)` function from version 1.5 of the Boost library. Like the standard library, the hash value generated occupies 32 bits on 32-bit platforms and 64 bits on 64-bit platforms.

**VHASH** We chose to compare against VHASH because it is one of the fastest families of hash functions on longer data sets: e.g., it is several times faster than high performance alternatives such as Poly1305 [30]. It is faster than UMAC [30] which has itself found to be twice as fast as MMH [27]. We used the most recent VHASH implementation made available by its authors [45]. It generates 32-bit hash values on 32-bit platforms whereas it generates 64-bit hash values on 64-bit platforms. On 64-bit platforms, it pads data with zeros if the input size is not a multiple of 16 bytes. Such padding results in copying up to 127 bytes to an intermediate buffer. This operation is done at most once per data segment, and, therefore, affects only relatively short segments. We believe that certain changes in the base implementation might potentially be more efficient than our approach with copying; correspondingly, we calculate and present optimistic estimates that do not include time spent on additional processing (similar to our approach with the standard library). We proceed similarly on the 32-bit ARM platform.

For the Intel 32-bit platform (x86), the authors' implementation [45] provides two options: one uses SSE2 instructions and another one is in pure C. The performance of the SSE2 implementation is more than two times higher, so the SSE2 option is used for testing. This particular SSE2 implementation does not require a particular memory alignment. However, it also assumes that data is processed in blocks of 16 bytes, so we use buffering as in the 64-bit platform.



(a) 64 or more bit outputs



(b) 32-bit outputs

Figure 2. Speed of the hash functions on random strings of various lengths on the recent Intel Haswell microarchitecture.

## 8.1. *Variable-length results on recent Intel processors*

In Fig. 2, we compare directly the speeds in bytes per CPU cycle of our hash families over random strings of various lengths. We use a recent Intel processor with the recent Haswell microarchitecture: an Intel i7-4770 processor running at 3.4 GHz. This processor has 32 kB of L1 cache per core, 256 kB of L2 cache per core and 8 MB of L3 cache. The software was compiled with GNU GCC 4.8 to a 64-bit Linux executable.

In this test, the 64-bit VHASH is capable of hashing 3.9 input bytes per cycle for long strings (4 kB or more). PM+64 is 15 % slower on such long strings at 3.3 bytes per cycle.

Our PM+32 can be 40 % faster than PM+64, reaching speeds of 4.7 bytes per cycle on long strings. Thus, if we only need 32-bit hash values, it could be preferable to use PM+32. The speed of MurmurHash 3A is disappointing at 0.8 bytes per cycle, whereas MurmurHash 3F does better at 2 bytes per cycle on long strings. SipHash reaches a speed of 0.5 bytes per cycle. The Boost and std hash functions are slower on long strings (less than 0.25 bytes per cycle).

On short strings, PM+64 and PM+32 are fastest followed by MurmurHash 3F.

## 8.2. *Multiplatform performance testing*

*8.2.1. Methodology* We compare the time used by all hashing methods using `std::hash` as a reference (setting `std::hash` to 1.0). Each single test is characterized by three "dimensions": (1) platform and compiler; (2) data; and (3) physical machine. We have reduced our analysis to the first two "dimensions" as follows: given a platform and data, tests were done on some number of physical machines, and respective normalized timings were averaged.

*8.2.2. Platforms, compilers, hardware, sample data* Information about platforms/compilers used for our tests is gathered in Appendix A.

Table V. Relative time taken to hash (1–31 bytes) and long (256 kB) segments (standard library = 1). Best results are in bold.

| | Boost | Murmur-3[a] | SipHash | VHASH | PM+32 | PM+64[b] |
|---|---|---|---|---|---|---|
| *x86, GCC* | | | | | | |
| short | 1.03 | **0.50** | 3.94 | 2.64[c] | 0.59 | |
| long | 1.60 | 0.31 | 2.75 | 0.27 | **0.11** | |
| *x86, MSVC* | | | | | | |
| short | 1.23 | **0.63** | 5.70 | 4.28[c] | 0.76 | |
| long | 1.11 | 0.30 | 2.39 | 0.28 | **0.10** | |
| *x64, GCC* | | | | | | |
| short | 1.05 | 0.57 | 1.27 | 1.19[c] | **0.50** | 0.53 |
| long | 1.38 | 0.15 | 0.73 | **0.08** | 0.10 | 0.09 |
| *x64, MSVC* | | | | | | |
| short | 1.37 | 0.77 | 1.90 | 2.02[c] | 0.69 | **0.65** |
| long | 1.40 | 0.13 | 0.67 | **0.10** | **0.09** | **0.09** |
| *ARMv7* | | | | | | |
| short | 0.89 | **0.86** | –[d] | 1.22[c] | 0.87 | |
| long | 1.29 | 0.76 | –[d] | 0.81 | **0.49** | |

[a] MurmurHash 3A for 32-bit platforms and MurmurHash 3F for 64-bit platforms. As suggested by a comment in the MurmurHash3 code [46], MurmurHash 3A is best on 32-bit platforms, and MurmurHash 3F is best on 64-bit platforms (and this has been confirmed in our tests).
[b] PM+64 is implemented for 64-bit platforms only
[c] Optimistic estimate as described above; measured values were 30–40 % higher.
[d] Not tested

Performance tests were done for short (1–31 bytes) and long (256 kB) data segments. Results for both short and long data segments were averaged; all results were finally represented as ratios to the default `std::hash` function. For these tests, data segments were provided by the SMHasher testing framework. None of the methods is designed or optimized for a specific type of data, such as, for instance, text, and, therefore, none of the methods is put in explicit (dis)advantage by such data generation.

We expect our results to be independent from the number of CPU cores since none of our techniques are parallelized. Moreover, we also expect the RAM type to be insignificant since even our large segments fit in L3 processor cache.

*8.2.3. Multiplatform performance results* Results of performance testing are gathered in Table V. On x86 and x64 platforms time was measured in CPU clocks (`rdtsc`); and on ARM time values were collected in microseconds. While averaging over different physical machines, the greatest relative standard deviation among all entries except SipHash was 22 % (deviation of SipHash was up to 38 %), and for over 90 % of entries this value was less than 15 %. The relative standard deviations are sufficiently small to view the presented averages as representative and to provide some assurance that relative performance results of our algorithms can be expected on a variety of platforms. The results are summarized in Fig. 3.

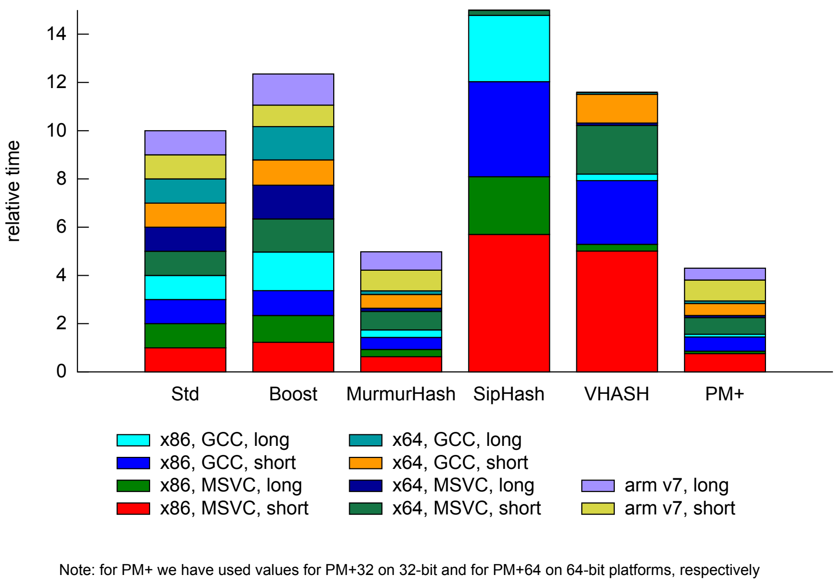


Figure 3. Performance summary

Our results show that the hash functions in the standard library can be slow on long segments: MurmurHash, VHASH and PM+ can be ten times faster. But even on short segments, PM+ can be twice as fast as the standard library (on x64 platforms).

The VHASH implementation is only competitive on long segments on x64. We are not surprised: it was designed specifically for 64-bit processors. On the x64-GCC platform, VHASH can be up to about 30 % faster than PM+. (The precise averages on long segments for VHASH and PM+ are 0.081 and 0.106.) This is consistent with earlier findings [17] (see Appendix F): VHASH is based on a function (NH) that is computationally inexpensive compared with MULTILINEAR—at the expense of regularity.

PM+ fares well on the ARM platform: PM+ is at least 50 % faster than the alternatives on long segments.

PM+ is faster than MurmurHash 3 on x64 platforms. MurmurHash 3 is only significantly faster (20 %) than PM+ on short segments on the x86-MSVC platform.

PM+32 and PM+64 have, in average, similar performance on 64-bit platforms. A closer examination reveals that PM+32 is faster than PM+64 on recent processors supporting AVX2 instruction set (as reported in § 8.1) while it is slower on older processors without support for AVX2.

## 9. CONCLUSION

We have described methods for constructing almost-universal hash function families for data strings of variable length. Our hash functions are suitable for use in common data structures such as hash tables. They offer strong theoretical guarantees against denial-of-service attacks:

- We have almost universality: given two distinct data objects chosen by an adversary, the probability that they have the same hash value, that is, the probability that they collide, is very low given that we pick the hash functions at random. Our families have lower collision bounds than the state-of-the-art VHASH.

- We have shown that these hash functions are regular and component-wise regular, that is, they make an even use of all possible hash values. In doing so, they minimize the collision probability between two data objects selected at random. Competitive alternatives such as VHASH are not regular which is a possible security risk [5, 6].

Further, we have shown that an implementation of these non-cryptographic hash functions offered competitive speed (as fast as MurmurHash), and were substantially faster than the implementations found in C++ standard libraries. Our approach is similar to previous work on fast universal hash families (e.g., MMH [32], CLHASH [41], UMAC [27], VHASH [30] and Poly1305 [15]), except that we get good regularity in addition to the high speed and universality. To promote the use of our hash functions among practitioners and researchers, our implementation is freely available as open source software.

In the future, it may be interesting to analyze the regularity of other universal hash families [15, 32, 31, 41], possibly improving it when possible. We could also seek faster families of hash functions that are both almost universal and regular.

## 10. ACKNOWLEDGEMENTS

We thank Ivan Kravets for his help with our testing framework, and more specifically with the ARM processors.

## REFERENCES

1. Crosby SA, Wallach DS. Denial of service via algorithmic complexity attacks. *Proc. 12th Conference on USENIX Security Symposium*, USENIX, Berkeley: Washington, DC, 2003; 3–3.
2. Klink A, Wälde J. Denial of service through hash table multi-collisions. *Technical Report oCERT-2011-003*, Open Source Computer Security Incident Response Team, Pittsburgh, PA 2011.
3. Oorschot P, Robert JM, Martin M. A monitoring system for detecting repeated packets with applications to computer worms. *International Journal of Information Security* 2006; **5**(3):186–199, doi:10.1007/s10207-006-0081-8.
4. Orton Y. Hardening Perl's hash function. http://blog.booking.com/hardening-perls-hash-function.html [last checked June 2016] 2013.
5. Handschuh H, Preneel B. Key-recovery attacks on universal hash function based mac algorithms. *Advances in Cryptology – CRYPTO 2008*, *Lecture Notes in Computer Science*, vol. 5157. Springer: Berlin Heidelberg, 2008; 144–161, doi:10.1007/978-3-540-85174-5_9.
6. Saarinen MJO. Cycling attacks on GCM, GHASH and other polynomial MACs and hashes. *Fast Software Encryption*, *Lecture Notes in Computer Science*, vol. 7549. Springer: Berlin Heidelberg, 2012; 216–225, doi:10.1007/978-3-642-34047-5_13.
7. Appleby A. SMHasher & MurmurHash. https://github.com/aappleby/smhasher [last checked June 2016] 2012.
8. Dai W, Krovetz T. VHASH security. *IACR Cryptology ePrint Archive* 2007; **338**.
9. Dietzfelbinger M. Universal hashing and k-wise independent random variables via integer arithmetic without primes. *Proc. 13th Symp. on Theoretical Aspects of Computer Science*, LNCS 1046, Springer-Verlag, Berlin: Grenoble, France, 1996; 569–580.
10. Carter JL, Wegman MN. Universal classes of hash functions. *Journal of computer and system sciences* 1979; **18**(2):143–154.
11. Cormen TH, Leiserson CE, Rivest RL, Stein C. *Introduction to Algorithms, Third Edition*. 3rd edn., The MIT Press: Cambridge, MA, 2009.
12. Stinson DR. Universal hashing and authentication codes. *Designs, Codes and Cryptography* 1994; **4**(4):369–380.
13. Lemire D. The universality of iterated hashing over variable-length strings. *Discrete Applied Mathematics* 2012; **160**(4–5):604–617, doi:10.1016/j.dam.2011.11.009.
14. Stinson DR. On the connections between universal hashing, combinatorial designs and error-correcting codes. *Congressus Numerantium* 1996; **114**:7–28.
15. Bernstein DJ. The Poly1305-AES Message-Authentication Code. *Fast Software Encryption*, *Lecture Notes in Computer Science*, vol. 3557. Springer: Berlin Heidelberg, 2005; 32–49, doi:10.1007/11502760_3.
16. Wegman MN, Carter JL. New hash functions and their use in authentication and set equality. *Journal of computer and system sciences* 1981; **22**(3):265–279.
17. Lemire D, Kaser O. Strongly universal string hashing is fast. *The Computer Journal* 2014; **57**(11):1624–1638, doi:10.1093/comjnl/bxt070.
18. Krovetz T, Rogaway P. Variationally universal hashing. *Information Processing Letters* 2006; **100**(1):36–39.
19. Lemire D, Kaser O. Recursive $n$-gram hashing is pairwise independent, at best. *Computer Speech and Language* Oct 2010; **24**(4):698–710, doi:10.1016/j.csl.2009.12.001.
20. Zobrist AL. A new hashing method with application for game playing. *Technical Report 88*, Computer Sciences Department, University of Wisconsin 1970. http://www.cs.wisc.edu/techreports/viewreport.php?report=88 [last checked June 2016].

21. Zobrist AL. A new hashing method with application for game playing. *ICCA Journal* 1990; **13**(2):69–73.
22. Thorup M, Zhang Y. Tabulation-based 5-independent hashing with applications to linear probing and second moment estimation. *SIAM Journal on Computing* 2012; **41**(2):293–331.
23. Richter S, Alvarez V, Dittrich J. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proc. VLDB Endow.* Nov 2015; **9**(3):96–107, doi:10.14778/2850583.2850585.
24. Bellare M, Kohno T. Hash function balance and its impact on birthday attacks. *Advances in Cryptology-Eurocrypt 2004*, Springer: Berlin Heidelberg, 2004; 401–418.
25. Canetti R, Micciancio D, Reingold O. Perfectly one-way probabilistic hash functions (preliminary version). *Proc. 13th Annual ACM Symposium on Theory of Computing*, STOC '98, ACM: New York, NY, USA, 1998; 131–140, doi:10.1145/276698.276721.
26. Fleischmann E, Forler C, Lucks S. Γ-MAC[H, P]: A New Universal MAC Scheme. *Proc. 4th Western European Conference on Research in Cryptology*, WEWoRC'11, Springer-Verlag: Berlin, Heidelberg, 2012; 83–98, doi: 10.1007/978-3-642-34159-5_6.
27. Black J, Halevi S, Krawczyk H, Krovetz T, Rogaway P. UMAC: Fast and secure message authentication. *Proc. 19th Int. Cryptology Conf. on Advances in Cryptology*, LNCS 1666, Springer-Verlag, Berlin: Santa Barbara, CA, 1999; 216–233.
28. Sarkar P. A trade-off between collision probability and key size in universal hashing using polynomials. *Designs, Codes and Cryptography* 2011; **58**(3):271–278.
29. Boesgaard M, Christensen T, Zenner E. Badger – a fast and provably secure MAC. *Proc. 3rd Applied Cryptography and Network Security*, ACNS'05, Springer Berlin Heidelberg: Berlin, Heidelberg, 2005; 176–191, doi:10.1007/11496137_13.
30. Krovetz T. Message authentication on 64-bit architectures. *Selected Areas in Cryptography*, *Lecture Notes in Computer Science*, vol. 4356. Springer: Berlin Heidelberg, 2007; 327–341, doi:10.1007/978-3-540-74462-7_23.
31. Krovetz T, Rogaway P. Fast universal hashing with small keys and no preprocessing: The PolyR construction. *Information Security and Cryptology-—ICISC 2000*. Springer: Berlin Heidelberg, 2001; 73–89.
32. Halevi S, Krawczyk H. MMH: Software message authentication in the Gbit/second rates. *Fast Software Encryption*, *Lecture Notes in Computer Science*, vol. 1267, Biham E (ed.). Springer: Berlin Heidelberg, 1997; 172–189, doi: 10.1007/BFb0052345.
33. Nussbaumer HJ. Digital filtering using complex Mersenne transforms. *IBM Journal of Research and Development* Sep 1976; **20**(5):498–504, doi:10.1147/rd.205.0498.
34. Quet L. $a(n)$ = least prime $\geq 2^{2^n}$. http://oeis.org/A132198 [last checked June 2016] 2007.
35. Estébanez C, Hernandez-Castro JC, Ribagorda A, Isasi P. Evolving hash functions by means of genetic programming. *Proc. 8th annual conference on Genetic and evolutionary computation*, ACM: New York, NY, USA, 2006; 1861–1862.
36. Aumasson JP, Bernstein DJ. SipHash: a fast short-input PRF. *Progress in Cryptology-INDOCRYPT 2012*. Springer: Berlin Heidelberg, 2012; 489–508.
37. Pike G, Alakuijala J. Introducing CityHash. http://google-opensource.blogspot.com/2011/04/introducing-cityhash.html [last checked June 2016] 2011.
38. So W, Narayanan A, Oran D, Wang Y. Toward fast ndn software forwarding lookup engine based on hash tables. *Proc 8th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, ACM: New York, NY, USA, 2012; 85–86, doi:10.1145/2396556.2396575.
39. Jenkins B. SpookyHash: a 128-bit noncryptographic hash. https://github.com/centaurean/spookyhash [last checked June 2016] 2012.
40. FarmHash, a family of hash functions. https://github.com/google/farmhash [last checked June 2016].
41. Lemire D, Kaser O. Faster 64-bit universal hashing using carry-less multiplications. *Journal of Cryptographic Engineering* 2015; **6**(3):171–185, doi:10.1007/s13389-015-0110-5.
42. Ahmad T, Younis U. Randomness testing of non-cryptographic hash functions for real-time hash table based storage and look-up of URLs. *Journal of Network and Computer Applications* 2014; **41**(0):197–205, doi:10.1016/j.jnca.2013.11.007.
43. Estébanez C, Saez Y, Recio G, Isasi P. Performance of the most common non-cryptographic hash functions. *Software: Practice and Experience* 2014; **44**(6):681–698, doi:10.1002/spe.2179.
44. Thorup M. High speed hashing for integers and strings. http://arxiv.org/abs/1504.06804 [last checked June 2016] 2015.
45. Krovetz T, Dai W. VMAC and VHASH Implementation. http://fastcrypto.org/vmac/ [last checked June 2016] 2007.
46. Appleby A. MurmurHash3 code. https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp [last checked June 2016] 2012.
47. Intel Corporation. Intel IACA tool: A Static Code Analyser. https://software.intel.com/en-us/articles/intel-architecture-code-analyzer [last checked June 2016] 2012.
48. Intel Corporation. The Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/ [last checked June 2016] 2015.
49. Thorup M. Even strongly universal hashing is pretty fast. *Proc. 11th ACM-SIAM Symp. on Discrete Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia: San Francisco, CA, 2000; 496–497.
50. Babai L, Pomerance C, Vértesi P. The mathematics of Paul Erdös. *Notices of the AMS* 1998; **45**(1):19–31.
51. Lambert D. Number of distinct products $ij$ with $0 \leq i, j \leq 2^n - 1$. http://oeis.org/A027417 [last checked June 2016] 2012.

Table VI. Platforms used.

| Name | Processor | Bits | Compiler | Flags/Configuration |
|---|---|---|---|---|
| x64, GCC | AMD, Intel[a] | 64 | GNU GCC 4.8 | -O2 -march=x86-64 |
| x64, MSVC | Intel Core i7[b] | 64 | MSVS 2013 | Release |
| x86, GCC | AMD, Intel[a] | 32 | GNU GCC 4.8 | -O2 -march=i686 |
| x86, MSVC | Intel Core i7[b] | 32 | MSVS 2013 | Release |
| ARMv7 | ARM Cortex/Krait[c] | 32 | GNU GCC 4.6[d] | Release |

[a] Results have been averaged over AMD FX-8150 Eight-Core (Bulldozer, Desktop), Intel Core i7 620M (Westmere, Mobile), Intel Xeon E5-2630 (Sandy Bridge, Server), Intel Core i5-3230M (Ivy Bridge, Mobile), and Intel Core i7-4770 (Haswell, Desktop) with a maximum relative standard deviation of 0.20.

[b] Results have been averaged over Intel Core i7-2820QM (Sandy Bridge, Mobile), Intel Core i7-3667U (Ivy Bridge, Ultra-low power), Intel Core i7-3770 (Ivy Bridge, Desktop), Intel Core i7-4960X (Ivy Bridge, Extreme edition), and Intel Core i7-4700MQ (Haswell, Mobile) with a maximum relative standard deviation of 0.12.

[c] Results have been averaged over Exynos 3110 (Cortex A8), Qualcomm Snapdragon MSM8255 (Scorpion), dual-core Exynos 4210 (Cortex-A9), dual-core Exynos 4412 (Cortex-A9), and quad-core Qualcomm Snapdragon 600 (Krait 300) with a maximum relative standard deviation of 0.21.

[d] From the Android NDK, revision r9d.

## A. PLATFORMS USED

The platforms/compilers that we have used for testing are: x86/x64 with Microsoft Visual C++ 2013 compiler; x86/x64 with GCC compiler (version: 4.8); and ARMv7 with the Android NDK (revision 9c, December 2013) which uses the GCC compiler internally. For more details see Table VI.

## B. OPTIMIZATION TECHNIQUES FOR CALCULATING SCALAR PRODUCTS ON X86 AND X64 PROCESSORS

As mentioned in § 6.1, it is important to optimize the computation of the scalar product. Overall, for the computation of the scalar product on x86 processors, we found best to use vectorization in the 32-bit case presented in § B.1. In the 64-bit case, we present a thoroughly optimized use of conventional instructions in § B.2.

### *B.1. Vectorizing the Computation of the Scalar Product*

We can implement a scalar product over pairs of 32-bit integers using one multiplication per pair, as well as additions with carry bit (e.g., the `adc` x86 instruction) to generate the resulting 3-word (96-bit) result.

To achieve better speed, we use the fact that modern CPUs support vector computations through Single Instruction on Multiple Data (SIMD) instructions. For instance, the x86 architecture has Streaming SIMD Extensions (SSE) using 128-bit registers and the more recent Advanced Vector Extensions (AVX) using wider 256-bit registers.

Our fastest 32-bit scalar production implementation for recent Intel processors uses AVX2. AVX2 has a `vpmuludq` instruction (corresponding to the `_mm256_mul_epu32` Intel intrinsic) that can multiply four pairs of 32-bit integers, thus generating four 64-bit integers.

### B.2. *Faster sums using two sets of accumulators*

The standard instruction set may provide better for 64-bit outputs. That is, we can multiply two 64-bit integers, and then add the 128-bit result to three 64-bit words (representing a 192-bit sum) using a sequence of x64 instructions: `mulq` (multiplication), `addq` (addition), and two `adcq` (add with a carry bit). The steps can be described as follows:

1. We use three 64-bit registers as accumulators $c_1, c_2, c_3$ representing the total sum as a $3 \times 64 = 192$-bit integer. The registers are initialized with zeros.
2. For each input pair of 64-bit values, the `mulq` instruction multiplies them and stores the results in two 64-bit registers ($a$ and $d$). One register ($a$) contains the least significant 64 bits of the product, and the other ($d$) the most significant 64 bits.
3. The first accumulator, corresponding to the least significant 64 bits, is easily updated with a simple addition (`addq`): $c_1 = (c_1 + a) \bmod 2^{64}$. If the sum exceeds $2^{64} - 1$, the carry bit $b$ is set to 1. That is, we have that $b = (c_1 + a) \div 2^{64}$. Then we update the second accumulator using the add-with-carry instruction (`adcq`): $c_2 = (c_2 + d + b) \bmod 2^{64}$. We also update the third accumulator similarly.

This approach is efficient: we only use 4 arithmetic x64 instructions per input pair. Yet, maybe surprisingly, there is still room for optimization.

If their operands and output values are independent, modern processors may perform more than a single instruction at a time. Indeed, recent Intel processors can retire 4 instructions (or fused $\mu$ops) per cycle. We reviewed the initial version of our code with the IACA code analyzer [47] for the most recent Intel microarchitecture (Haswell). IACA revealed that the throughput was limited by data dependencies. Though the processor can execute one multiplication per cycle, it may sometimes have to wait for the accumulators to be updated. Thus we rewrote our code to use two sets of accumulators. Effectively, we sum the odd terms and the even terms separately ($\sum_{i=1}^{m/2} a_{2i}s_{2i}$ and $\sum_{i=1}^{m/2} a_{2i+1}s_{2i+1}$) and then we combine them. Respective code samples can be found in Appendix D (in x64 assembly) and Appendix E (in C++ with Intel intrinsics). A new analysis with IACA reveals that the throughput of this new code is then limited by the frontend of the processor (responsible for instruction decoding). On long strings, using a recent Haswell processor (Intel i7-4770 running at 3.4 GHz), we went from $\approx 1150$ million input pairs per second to $\approx 1350$ million input pairs per second (an 18 % gain).

## C. CODE SAMPLE TO SUM 64-BIT PRODUCTS OF 32-BIT INTEGERS

The following C++ code computes the 96-bit integer representing the sum of 128 products between pairs of 32-bit integers using AVX2 intrinsics ($m = 128, n = 32$). See § B.1 for an analysis. For the description of the intrinsics, we refer the reader to Intel's documentation [48].

```
// input: two arrays of 32-bit integers
// const uint32_t* coeff;
// const uint32_t* x;

// output parameters:
uint64_t low_bits;
uint32_t high_bits;

__m256i ctr0, ctr1;
__m256i a, data, product, temp;
uint64_t temp_fin;

// Set accumulators to zero
ctr0 = _mm256_setzero_si256 ();
ctr1 = _mm256_setzero_si256 ();
```

```
// process the loop (unrolling may help)
for ( int i=0; i<128; i+=8 )
{
  // Load 256-bit value (eight ints)
  a = _mm256_loadu_si256
      ((__m256i *)(coeff+i));
  data = _mm256_loadu_si256
      ((__m256i *)(x+i));
  // multiply ints at even positions
  product = _mm256_mul_epu32 ( data, a);
  temp = _mm256_srli_epi64
      ( product, 32 );
  ctr1 = _mm256_add_epi64
      ( ctr1, temp );
  ctr0 = _mm256_add_epi64
      ( ctr0, product );
  // exchange even-odd
  // note: 0xb1 =  1*1+0*4+3*16+2*64
  a = _mm256_shuffle_epi32
      ( a, 0xb1);
  data = _mm256_shuffle_epi32
      ( data, 0xb1 );
  // multiply ints at even positions
  // (former odd positions)
  product = _mm256_mul_epu32 ( data, a);
  temp = _mm256_srli_epi64
      ( product, 32 );
  ctr1 = _mm256_add_epi64
      ( ctr1, temp );
  ctr0 = _mm256_add_epi64
      ( ctr0, product );
}

// finalize

// desired results are in c0 and c1
// we interleave the sums and add them
temp = _mm256_unpackhi_epi64
       ( ctr0, ctr1 );
data = _mm256_unpacklo_epi64
       ( ctr0, ctr1 );
ctr1 = _mm256_add_epi64
       ( data, temp );
// extract  a 64+32 bit number
// (low_bits, high_bits)
uint64_t lo = *(uint64_t*)(&ctr1) +
            ((uint64_t*)(&ctr1))[2];
uint64_t hi = ((uint64_t*)(&ctr1))[1] +
            ((uint64_t*)(&ctr1))[3];
uint32_t lohi = lo >> 32;
uint32_t hilo = hi;
uint32_t diff = lohi - hilo;
hi += diff;
lo = (uint32_t)lo +
       (((uint64_t)(uint32_t)hi)<<32);

// answer:
low_bits = lo;
high_bits = hi >> 32;
```

## D. CODE SAMPLE TO SUM 128-BIT PRODUCTS OF 64-BIT INTEGERS (ASSEMBLER)

The following assembly code computes the 192-bit integer representing the sum of 128 products between pairs of 64-bit integers using the standard x64 instruction set ($m = 128, n = 64$).

```
// input: pointers to 64-bit arrays
// rbx: address of the start of
//      the 1st array
// rcx: address of the start of
//      the 2nd array
// 1st accumulator:
// r10: least significant 64 bits
// r11: mid 64 bits
// r12: most significant 64 bits
// 2nd accumulator:
// r13: least significant 64 bits
// r14: mid 64 bits
// r15: most significant 64 bits

// add 1st product to 1st accumulator
movq 0(%rbx),%%rax\n
mulq 0(%rcx)\n
addq %%rax,  %%r10\n
adcq %%rdx,  %%r11\n
adcq $0,  %%r12\n

// add 1st product to 1st accumulator
movq 8(%rbx),%%rax\n
mulq 8(%rcx)\n
addq %%rax,  %%r13\n
adcq %%rdx,  %%r14\n
adcq $0,  %%r15\n

// ... repeat as necessary

// merge accumulators:
movq 8(%rbx),%%rax\n
mulq 8(%rcx)\n
addq %%rax,  %%r13\n
adcq %%rdx,  %%r14\n
adcq $0,  %%r15\n

// the sum of products is now at
// (r10, r11, r12)
```

## E. CODE SAMPLE TO SUM 128-BIT PRODUCTS OF 64-BIT INTEGERS USING INTEL INTRINSICS

The following C++ code computes the 192-bit integer representing the sum of 128 products between pairs of 64-bit integers using Intel intrinsics ($m = 128, n = 64$). Such code is well suited for the Microsoft Visual C++ compiler.

```
// 1st accumulator:
uint64_t low1   = 0; // least sign. 64 bits
uint64_t high1  = 0; // next 64 bits
uint64_t vhigh1 = 0; // most significant
// 2nd accumulator
uint64_t low2   = 0; // least sign. 64 bits
uint64_t high2  = 0; // next 64 bits
uint64_t vhigh2 = 0; // most significant
// intermediates:
```

```
uint64_t mulLo, mulHi;
unsigned char c;
for(size_t i = 0; i<128; i+=2)
{
  // process even pair
  // _umul128 is Microsoft-specific
  mulLo = _umul128(a[i],s[i],&mulHi);
  // _addcarry_u64 is an Intel intrinsic
  // supported by Microsoft
  c = _addcarry_u64
     (0, mulLo, low1, &low1);
  c = _addcarry_u64
     (c, mulHi, high1, &high1);
  _addcarry_u64(c, vhigh1, 0, &vhigh1);

  // process odd pair
  mulLo = _umul128
         (a[i+1],s[i+1],&mulHi);
  c = _addcarry_u64
         (0, mulLo, low2, &low2);
  c = _addcarry_u64
         (c, mulHi, high2, &high2);
  _addcarry_u64(c, vhigh2, 0, &vhigh2);
}

c = _addcarry_u64(0, low1, low2, &low1);
c = _addcarry_u64
    (c, high1, high2, &high1);
_addcarry_u64
    (c, vhigh1, vhigh2, &vhigh1);

// result is at (low1, high1, vhigh1)
```

## F. NON-REGULARITY OF THE VHASH FAMILY

The effort to design practical universal random hash functions with good properties has a long history. Thorup [49] showed that strongly universal hashing could be very fast. Crosby and Wallach [1] showed that almost universal hashing could be as fast as common deterministic hash functions. Their conclusion was that while universal hash functions were not standard practice, they should be. In particular, they got good experimental results with UMAC [27].

More recently, Krovetz proposed the VHASH family [30]. On 64-bit processors, it is faster than the hash functions from UMAC.

Like UMAC, VHASH is $\epsilon$-almost $\Delta$-universal and builds on the NH family:

$$\mathrm{NH}(s) = \left( \sum_{i=1}^{l/2} \left( ((s_{2i-1} + k_{2i-1}) \bmod 2^n) \right. \right.$$
$$\left. \left. \times ((s_{2i} + k_{2i}) \bmod 2^n) \right) \right) \bmod 2^{2n}.$$

NH is fast in part due to the fact that it uses one multiplication per pair of input words. In contrast, MMH or PM+, as derivatives of MULTILINEAR, use at least one multiplication per input word. However, the number of multiplications is not necessarily a performance bottleneck: recent Intel processors can execute one multiplication per cycle. We should not expect hash functions with half the number of multiplications to be twice as fast [17]. For example, a fast hash function might be limited by the number of micro-operations that the processor can retire per cycle (4 on recent Intel processors) rather than by the number of multiplications.

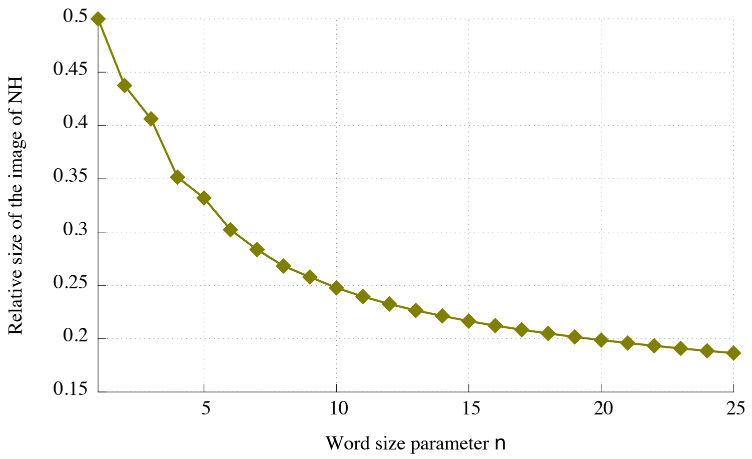


Figure 4. Fraction of all $2n$-bit integers that are the product of two $n$-bit integers.

Like MULTILINEAR, NH is $1/2^n$-almost $\Delta$-universal, but MULTILINEAR generates values in $[0, 2^n)$ whereas NH generates values in $[0, 2^{2n})$. (For this reason, NH might not be well suited for a tree-based approach as described in § 4.)

Krovetz reports that VHASH is twice as fast as UMAC (0.5 CPU cycle per input byte vs. 1 CPU cycle per input byte on an AMD Athlon processor). For long strings on 64-bit processors, we expect VHASH to be one of the fastest universal hash families.

The updated VHASH [8] family is $\epsilon$-almost universal over $[0, 2^{64} - 257)$ with $\epsilon = \frac{1}{2^{61}}$ for strings of length up to $2^{62}$ bits. In contrast, PM+ produces hash values in $[0, 2^{64})$ with $\epsilon = \frac{8}{2^{63}-6}$ for strings of length up to $(2^{62} - 64)$ bits.

We can describe the 64-bit VHASH as follows: NH is used with $n = 64$ to generate 128-bit hash values on 128-byte blocks. The result is $1/2^{64}$-almost $\Delta$-universal on each block. In turn, the result is mappend to the interval $[0, 2^{126})$ by applying a modulo reduction $(\bmod 2^{126})$: the family is then $1/2^{62}$-almost $\Delta$-universal on each block. The hashed values over each block are then aggregated using a polynomial hash family computed over $[0, 2^{127} - 1)$. The result is finally reduced to $[0, 2^{64} - 257)$ with modulo operations and divisions.

The NH family is not regular. For instance, consider values of $s$ where $s = (s_1, s_2)$ and at least one of $s_1$ and $s_2$ is even, which is $\frac{3}{4}$ of all possible values. If both $k_1$ and $k_2$ are even, then $\mathrm{NH}(s)$ is even, too, and, therefore, $\frac{3}{4}$ of all values are mapped to only $\frac{1}{2}$ of all values.

To make matters worse, the NH family is never regular for any choice of keys $(k_i)$ and it has "very little regularity" as $n$ grows in the following sense. For any given integers $k_1, k_2 \in [0, 2^n)$, consider the map from $[0, 2^n) \times [0, 2^n) \to [0, 2^{2n})$ given by $\mathrm{NH}(x, y) = ((x + k_1 \bmod 2^n)((y + k_2 \bmod 2^n) \bmod 2^{2n}$. Because we pick $x, y \in [0, 2^n)$, we can choose $k_1 = k_2 = 0$ without loss of generality. We can then ask about the size of the image of $\mathrm{NH}(x, y)$. That is, which fraction of all integers in $[0, 2^{2n})$ are the product of two numbers in $[0, 2^n)$? Erdös showed that this ratio goes to zero as $n$ becomes large [50]. Though we do not know of an exact formula, we plot the relative size of the image of NH in Fig. 4: already at $n = 20$ only about one integer out of five in $[0, 2^{40})$ can be generated by the product of two integers in $[0, 2^{20})$ [51]. We expect that for $n = 64$, the ratio is considerably less than 20 %. Note that keeping only, say, the least significant $2n - 2$ bits (e.g., applying $\bmod 2^{2n-2}$) or most significant $2n - 2$ bits (e.g., applying $\div 2^2$) does not change the core result: the relative size of the image still goes to zero as $n$ becomes large.

Hence NH is not even 5-regular. The issue is more dramatic if we consider component-wise regularity. Indeed, consider $\mathrm{NH}(s)$ over 2-character strings $(s_1, s_2)$. If $s_2 + k_2 \bmod 2^n = 0$, we have that $\mathrm{NH}(s) = 0$ for all values of $s_1$, which is the worst possible case. NH and VHASH are not at all component-wise regular.