J Cryptogr Eng (2016) 6:171–185
DOI 10.1007/s13389-015-0110-5

REGULAR PAPER

Faster 64-bit universal hashing using carry-less multiplications

Daniel Lemire[1] · Owen Kaser[2]

Received: 27 December 2014 / Accepted: 14 August 2015 / Published online: 4 September 2015

Abstract Intel and AMD support the carry-less multiplication (CLMUL) instruction set in their x64 processors. We use CLMUL to implement an almost universal 64-bit hash family (CLHASH). We compare this new family with what might be the fastest almost universal family on x64 processors (VHASH). We find that CLHASH is at least 60 % faster. We also compare CLHASH with a popular hash function designed for speed (Google's CityHash). We find that CLHASH is 40 % faster than CityHash on inputs larger than 64 bytes and just as fast otherwise.

Keywords Universal hashing · Carry-less multiplication · Finite field arithmetic

1 Introduction

Hashing is the fundamental operation of mapping data objects to fixed-size hash values. For example, all objects in the Java programming language can be hashed to 32-bit integers. Many algorithms and data structures rely on hashing: e.g., authentication codes, Bloom filters and hash tables. We typically assume that given two data objects, the probability that they have the same hash value (called a *collision*) is low. When this assumption fails, adversaries can negatively impact the performance of these data structures or even create denial-of-service attacks. To mitigate such problems, we can pick hash functions at random (henceforth called *random hashing*).

Random hashing is standard in Ruby, Python and Perl. It is allowed explicitly in Java and C++11. There are many fast random hash families—e.g., MurmurHash, Google's CityHash [35], SipHash [3] and VHASH [12]. Cryptographers have also designed fast hash families with strong theoretical guarantees [6,18,24]. However, much of this work predates the introduction of the CLMUL instruction set in commodity x86 processors. Intel and AMD added CLMUL and its `pclmulqdq` instruction to their processors to accelerate some common cryptographic operations. Although the `pclmulqdq` instruction first became available in 2010, its high cost in terms of CPU cycles—specifically an eight-cycle throughput on pre-Haswell Intel microarchitectures and a seven-cycle throughput on pre-Jaguar AMD microarchitectures—limited its usefulness outside of cryptography. However, the throughput of the instruction on the newer Haswell architecture is down to two cycles, even though it remains a high latency operation (7 cycles) [16,21].[1] See Table 1. Our main contribution is to show that the `pclmulqdq` instruction can be used to produce a 64-bit string hash family that is faster than known approaches while offering stronger theoretical guarantees.

2 Random hashing

In random hashing, we pick a hash function at random from some family, whereas an adversary might pick the data inputs. We want distinct objects to be unlikely to hash to the same value. That is, we want a low collision probability.

✉ Daniel Lemire
lemire@gmail.com

Owen Kaser
o.kaser@computer.org

[1] LICEF Research Center, TELUQ, Université du Québec, Montreal, QC, Canada

[2] Department of CSAS, University of New Brunswick, Saint John, NB, Canada

[1] The low-power AMD Jaguar microarchitecture does even better with a throughput of one cycle and a latency of three cycles.

Table 1 Relevant SIMD intrinsics and instructions on *Haswell* Intel processors, with latency and reciprocal throughput in CPU cycles per instruction [16,21]

Intrinsic	Instruction	Description	Latency	Rec. thr.
_mm_clmulepi64_si128	`pclmulqdq`	64-bit carry-less multiplication	7	2
_mm_or_si128	`por`	Bitwise OR	1	0.33
_mm_xor_si128	`pxor`	Bitwise XOR	1	0.33
_mm_slli_epi64	`psllq`	Shift left two 64-bit integers	1	1
_mm_srli_si128	`psrldq`	Shift right by x bytes	1	0.5
_mm_shuffle_epi8	`pshufb`	Shuffle 16 bytes	1	0.5
_mm_cvtsi64_si128	`movq`	64-bit integer as 128-bit reg.	1	–
_mm_cvtsi128_si64	`movq`	64-bit integer from 128-bit reg.	2	–
_mm_load_si128	`movdqa`	Load a 128-bit reg. from memory (aligned)	1	0.5
_mm_lddqu_si128	`lddqu`	Load a 128-bit reg. from memory (unaligned)	1	0.5
_mm_setr_epi8	–	Construct 128-bit reg. from 16 bytes	–	–
_mm_set_epi64x	–	Construct 128-bit reg. from two 64-bit integers	–	–

Table 2 Notation and basic definitions

$h : X \to \{0, 1, \ldots, 2^L - 1\}$	L-bit hash function
Universal	$P(h(x) = h(x')) \leq 1/2^L$ for $x \neq x'$
ϵ-Almost universal	$P(h(x) = h(x')) \leq \epsilon$ for $x \neq x'$
XOR-universal	$P(h(x) = h(x') \oplus c) \leq 1/2^L$ for any $c \in [0, 2^L)$ and distinct $x, x' \in X$
ϵ-Almost XOR-universal	$P(h(x) = h(x') \oplus c) \leq \epsilon$ for any integer $c \in [0, 2^L)$ and distinct $x, x' \in X$

We consider hash functions from X to $[0, 2^L)$. An L-bit family is *universal* [10,11] if the probability of a collision is no more than 2^{-L}. That is, it is universal if

$$P(h(x) = h(x')) \leq 2^{-L}$$

for any fixed $x, x' \in X$ such that $x \neq x'$, given that we pick h at random from the family. It is ϵ-*almost universal* [36] (also written ϵ-AU) if the probability of a collision is bounded by ϵ. I.e., $P(h(x) = h(x')) \leq \epsilon$, for any $x, x' \in X$ such that $x \neq x'$. (See Table 2.)

2.1 Safely reducing hash values

Almost universality can be insufficient to prevent frequent collisions, since a given algorithm might only use the first few bits of the hash values. Consider hash tables. A hash table might use as a key only the first b bits of the hash values when its capacity is 2^b. Yet even if a hash family is ϵ-almost universal, it could still have a high collision probability on the first few bits.

For example, take any 32-bit universal family $\mathcal{H}$ and derive the new 64-bit $1/2^{32}$-almost universal 64-bit family by taking the functions from $\mathcal{H}$ and multiplying them by 2^{32}: $h'(x) = h(x) \times 2^{32}$. Clearly, all functions from this new family collide with probability 1 on the first 32 bits, even though the collision probability on the full hash values is low $(1/2^{32})$. Using the first bits of these hash functions could have disastrous consequences in the implementation of a hash table.

Therefore, we consider stronger forms of universality.

- A family is Δ-*universal* [14,37] if

  $$P(h(x) = h(x') + c \bmod 2^L) \leq 2^{-L}$$

  for any constant c and any $x, x' \in X$ such that $x \neq x'$. It is ϵ-almost Δ-universal if $P(h(x) = h(x') + c \bmod 2^L \leq \epsilon$ for any constant c and any $x, x' \in X$ such that $x \neq x'$.
- A family is ϵ-almost XOR-universal if

  $$P(h(x) = h(x') \oplus c) \leq \epsilon$$

  for any integer constant $c \in [0, 2^L)$ and any $x, x' \in X$ such that $x \neq x'$ (where $\oplus$ is the bitwise XOR). A family that is $1/2^L$-almost XOR-universal is said to be XOR-universal [37].

Given an ϵ-almost Δ-universal family $\mathcal{H}$ of hash functions $h : X \to [0, 2^L)$, the family of hash functions

$$\{h(x) \bmod 2^{L'} \mid h \in \mathcal{H}\}$$

from X to $[0, 2^{L'})$ is $2^{L-L'} \times \epsilon$-almost Δ-universal [12]. The next lemma shows that a similar result applies to almost XOR-almost universal families.

Lemma 1 *Given an ϵ-almost XOR-universal family $\mathcal{H}$ of hash functions $h : X \to [0, 2^L)$ and any positive integer $L' < L$, the family of hash functions $\{h(x) \bmod 2^{L'} \mid h \in \mathcal{H}\}$ from X to $[0, 2^{L'})$ is $2^{L-L'} \times \epsilon$-almost XOR-universal.*

Proof For any integer constant $c \in [0, 2^L)$, consider the equation $h(x) = (h(x') \oplus c) \mod 2^{L'}$ for $x \neq x'$ with h picked from $\mathcal{H}$. Pick any positive integer $L' < L$. We have

$$\begin{aligned} P(h(x) = (h(x') \oplus c \mod 2^{L'})) \\ = \sum_{z \mid z \bmod 2^{L'}=0} P(h(x) = h(x') \oplus c \oplus z), \end{aligned}$$

where the sum is over $2^{L-L'}$ distinct z values. Because $\mathcal{H}$ is ϵ-almost XOR-universal, we have that $P(h(x) = h(x') \oplus c \oplus z) \leq \epsilon$ for any c and any z. Thus, we have that $P(h(x) = h(x') \oplus c \mod 2^{L'}) \leq 2^{L-L'}\epsilon$, showing the result. □

It follows from Lemma 1 that if a family is XOR-universal, then its modular reductions are XOR-universal as well.

As a straightforward extension of this lemma, we could show that when picking any L' bits (not only the least significant), the result is $2^{L-L'} \times \epsilon$-almost XOR-universal.

2.2 Composition

It can be useful to combine different hash families to create new ones. For example, it is common to compose hash families. When composing hash functions ($h = g \circ f$), the universality degrades linearly: if g is picked from an ϵ_g-almost universal family and f is picked (independently) from an ϵ_f-almost universal family, the result is $\epsilon_g + \epsilon_f$-almost universal [36].

We sketch the proof. For $x \neq x'$, we have that $g(f(x)) = g(f(x'))$ collides if $f(x) = f'(x)$. This occurs with probability at most ϵ_f, since f is picked from an ϵ_f-almost universal family. If not, they collide if $g(y) = g(y')$ where $y = f(x)$ and $y' = f(x')$, with probability bounded by ϵ_g. Thus, we have bounded the collision probability by $\epsilon_f + (1 - \epsilon_f)\epsilon_g \leq \epsilon_f + \epsilon_g$, establishing the result.

By extension, we can show that if g is picked from an ϵ_g-almost XOR-universal family, then the composed result ($h = g \circ f$) is going to be $\epsilon_g + \epsilon_f$-almost XOR-universal. It is not required for f to be almost XOR-universal.

2.3 Hashing tuples

If we have universal hash functions from X to $[0, 2^L)$, then we can construct hash functions from X^m to $[0, 2^L)^m$ while preserving universality. The construction is straightforward: $h'(x_1, x_2, \ldots, x_m) = (h(x_1), h(x_2), \ldots, h(x_m))$. If h is picked from an ϵ-almost universal family, then the result is ϵ-almost universal. This is true even though a single h is picked and reused m times.

Lemma 2 *Consider an ϵ-almost universal family $\mathcal{H}$ from X to $[0, 2^L)$. Then consider the family of functions $\mathcal{H}'$ of the form $h'(x_1, x_2, \ldots, x_m) = (h(x_1), h(x_2), \ldots, h(x_m))$ from X^m to $[0, 2^L)^m$, where h is in $\mathcal{H}$. Family $\mathcal{H}'$ is ϵ-almost universal.*

The proof is not difficult. Consider two distinct values from X^m, $x_1, x_2, \ldots, x_m$ and $x'_1, x'_2, \ldots, x'_m$. Because the tuples are distinct, they must differ in at least one component: $x_i \neq x'_i$. It follows that $h'(x_1, x_2, \ldots, x_m)$ and $h'(x'_1, x'_2, \ldots, x'_m)$ collide with probability at most $P(h(x_i) = h(x'_i)) \leq \epsilon$, showing the result.

2.4 Variable-length hashing from fixed-length hashing

Suppose that we are given a family $\mathcal{H}$ of hash functions that is XOR universal over fixed-length strings. That is, we have that $P(h(s) = h(s') \oplus c) \leq 1/2^L$ if the length of s is the same as the length of s' ($|s| = |s'|$). We can create a new family that is XOR universal over variable-length strings by introducing a hash family on string lengths. Let $\mathcal{G}$ be a family of XOR universal hash functions g over length values. Consider the new family of hash functions of the form $h(s) \oplus g(|s|)$ where $h \in \mathcal{H}$ and $g \in \mathcal{G}$. Let us consider two distinct strings s and s'. There are two cases to consider.

- If s and s' have the same length so that $g(|s|) = g(|s'|)$, then we have XOR universality since

$$\begin{aligned} P(h(s) \oplus g(|s|) &= h(s') \oplus g(|s'|) \oplus c) \\ &= P(h(s) = h(s') \oplus c) \\ &\leq 1/2^L, \end{aligned}$$

  where the last inequality follows because $h \in \mathcal{H}$, an XOR universal family over fixed-length strings.
- If the strings have different lengths ($|s| \neq |s'|$), then we again have XOR universality because

$$\begin{aligned} P(h(s) \oplus g(|s|) &= h(s') \oplus g(|s'|) \oplus c) \\ &= P(g(|s|) \\ &= g(|s'|) \oplus (c \oplus h(s) \oplus h(s'))) \\ &= P(g(|s|) = g(|s'|) \oplus c') \\ &\leq 1/2^L, \end{aligned}$$

  where we set $c' = c \oplus h(s) \oplus h(s')$, a value independent of $|s|$ and $|s'|$. The last inequality follows because g is taken from a family $\mathcal{G}$ that is XOR universal.

Thus, the result $(h(s) \oplus g(|s|))$ is XOR universal. We can also generalize the analysis. Indeed, if $\mathcal{H}$ and $\mathcal{G}$ are ϵ-almost universal, we could show that the result is ϵ-almost universal. We have the following lemma.

Lemma 3 *Let $\mathcal{H}$ be an XOR universal family of hash functions over fixed-length strings. Let $\mathcal{G}$ be an XOR universal* *family of hash functions over integer values. We have that the family of hash functions of the form $s \to h(s) \oplus g(|s|)$ where $h \in \mathcal{H}$ and $g \in \mathcal{G}$ is XOR universal over all strings.*

Moreover, if $\mathcal{H}$ and $\mathcal{G}$ are merely ϵ-almost universal, then the family of hash functions of the form $s \to h(s) \oplus g(|s|)$ is also ϵ-almost universal.

2.5 Minimally randomized hashing

Many hashing algorithms—for instance, CityHash [35]—rely on a small random *seed*. The 64-bit version of CityHash takes a 64-bit integer as a seed. Thus, we effectively have a family of 2^{64} hash functions—one for each possible seed value.

Given such a small family (i.e., given few random bits), we can prove that it must have high collision probabilities. Indeed, consider the set of all strings of m 64-bit words. There are 2^{64m} such strings.

- Pick one hash function from the CityHash family. This function hashes every one of the 2^{64m} strings to one of 2^{64} hash values. By a pigeonhole argument [31], there must be at least one hash value where at least $2^{64m}/2^{64} = 2^{64(m-1)}$ strings collide.
- Pick another hash function. Out of the $2^{64(m-1)}$ strings colliding when using the first hash function, we must have $2^{64(m-2)}$ strings also colliding when using the second hash function.

We can repeat this process $m - 1$ times until we find 2^{64} strings colliding when using any of these $m - 1$ hash functions. If an adversary picks any two of our 2^{64} strings and we pick the hash function at random in the whole family of 2^{64} hash functions, we get a collision with a probability of at least $(m-1)/2^{64}$. Thus, while we do not have a strict bound on the collision probability of the CityHash family, we know just from the small size of its seed that it must have a relatively high collision probability for long strings. In contrast, VHASH and our CLHASH (see Sect. 5) use more than 64 random bits and have correspondingly better collision bounds (see Table 4).

3 VHASH

The VHASH family [12,25] was designed for 64-bit processors. By default, it operates over 64-bit words. Among hash families offering good almost universality for large data inputs, VHASH might be the fastest 64-bit alternative on x64 processors—except for our own proposal (see Sect. 5).

VHASH is ϵ-almost Δ-universal and builds on the 128-bit NH family [12]:

$$\mathrm{NH}(s) = \sum_{i=1}^{l/2} ((s_{2i-1} + k_{2i-1} \bmod 2^{64}) \times (s_{2i} + k_{2i} \bmod 2^{64})) \bmod 2^{128}. \qquad (1)$$

NH is $1/2^{64}$-almost Δ-universal with hash values in $[0, 2^{128})$. Although the NH family is defined only for inputs containing an even number of components, we can extend it to include odd numbers of components by padding the input with a zero component.

We can summarize VHASH (see Algorithm 1) as follows:

Algorithm 1 VHASH algorithm

```
Require: 16 randomly picked 64-bit integers $k_1, k_2, \ldots, k_{16}$ defining a 128-bit NH hash function (see Equation 1) over inputs of length 16
Require: $k$, a randomly picked element of $\{w2^{96} + x2^{64} + y2^{32} + z \mid \text{integers } w, x, y, z \in [0, 2^{29})\}$
Require: $k'_1, k'_2$, randomly picked integers in $[0, 2^{64} - 258]$
input: string $M$ made of $|M|$ bytes
Let $n$ be the number of 16-word blocks ($\lceil |M|/16 \rceil$).
Let $M_i$ be the substring of $M$ from index $i$ to $i+16$, padding with zeros if needed.
Hash each $M_i$ using the NH function, labelling the resulting 128-bit results $a_i$ for $i = 1, \ldots, n$.
Hash the resulting $a_i$ with a polynomial hash function and store the value in a 127-bit hash value $p$: $p = k^n + a_1 k^{n-1} + \cdots + a_n + (|M| \bmod 1024) \times 2^{64} \bmod (2^{127} - 1)$.
Hash the 127-bit value $p$ down to a 64-bit value: $z = (p_1 + k'_1) \times (p_2 + k'_2) \bmod (2^{64} - 257)$, where $p_1 = p \div (2^{64} - 2^{32})$ and $p_2 = p \bmod (2^{64} - 2^{32})$.
return the 64-bit hash value $z$
```

- NH is used to generate a 128-bit hash value for each block of 16 words. The result is $1/2^{64}$-almost Δ-universal on each block.
- These hash values are mapped to a value in $[0, 2^{126})$ by applying a modular reduction. These reduced hash values are then aggregated with a polynomial hash and finally reduced to a 64-bit value.

In total, the VHASH family is $1/2^{61}$-almost Δ-universal over $[0, 2^{64} - 257)$ for input strings of up to 2^{62} bits [12, Theorem 1].

For long input strings, we expect that much of the running time of VHASH is in the computation of NH on blocks of 16 words. On recent x64 processors, this computation involves 8 multiplications using the `mulq` instruction (with two 64-bit inputs and two 64-bit outputs). For each group of two consecutive words (s_i and s_{i+1}), we also need two 64-bit additions. To sum all results, we need seven 128-bit additions that can be implemented using two 64-bit additions (`addq` and `adcq`). All of these operations have a throughput of at least 1 per cycle on Haswell processors. We can expect NH and, by extension, VHASH to be fast.

VHASH uses only 16 64-bit random integers for the NH family. As in Sect. 2.3, we only need one specific NH function irrespective of the length of the string. VHASH also uses a 128-bit random integer k and two more 64-bit random integers k'_1 and k'_2. Thus, VHASH uses slightly less than 160 random bytes.

3.1 Random bits

Nguyen and Roscoe showed that at least $\log(m/\epsilon)$ random bits are required [31],[2] where m is the maximal string length in bits and ϵ is the collision bound. For VHASH, the string length is limited to 2^{62} bits and the collision bound is $\epsilon = 1/2^{61}$. Therefore, for hash families offering the bounds of VHASH, we have that $\log m/\epsilon = \log(2^{62} \times 2^{61}) = 123$ random bits are required.

That is, 16 random bytes are theoretically required to achieve the same collision bound as VHASH, while many more are used (160 bytes) This suggests that we might be able to find families using far fewer random bits while maintaining the same good bounds. In fact, it is not difficult to modify VHASH to reduce the use of random bits. It would suffice to reduce the size of the blocks down from 16 words. We could show that it cannot increase the bound on the collision probability by more than $1/2^{64}$. However, reducing the size of the blocks has an adverse effect on speed. With large blocks and long strings, most of the input is processed with the NH function before the more expensive polynomial hash function is used. Thus, there is a trade-off between speed and the number of random bits, and VHASH is designed for speed on long strings.

4 Finite fields

Our proposed hash family (CLHASH, see Sect. 5) works over a binary finite field. For completeness, we review field theory briefly, introducing (classical) results as needed for our purposes.

The real numbers form what is called a *field*. A field is such that addition and multiplication are associative, commutative and distributive. We also have identity elements (0 for addition and 1 for multiplication). Crucially, all non-zero elements a have an inverse a^{-1} (which is defined by $a \times a^{-1} = a^{-1} \times a = 1$).

Finite fields (also called Galois fields) are fields containing a finite number of elements. All finite fields have cardinality p^n for some prime p. Up to an *algebraic isomorphism* (i.e., a one-to-one map preserving addition and multiplication), given a cardinality p^n, there is only one field (henceforth, $\mathrm{GF}(p^n)$). For any power of a prime, there is a corresponding field.

4.1 Finite fields of prime cardinality

It is easy to create finite fields that have prime cardinality ($\mathrm{GF}(p)$). Given p, an instance of $\mathrm{GF}(p)$ is given by the set of integers in $[0, p)$ with additions and multiplications completed by a modular reduction:

- $a \times_{\mathrm{GF}(p)} b \equiv a \times b \bmod p$
- and $a +_{\mathrm{GF}(p)} b \equiv a + b \bmod p$.

The numbers 0 and 1 are the identity elements. Given an element a, its additive inverse is $p - a$.

It is not difficult to check that all non-zero elements have a multiplicative inverse. We review this classical result for completeness. Given a non-zero element a and two distinct x, x', we have that $ax \bmod p \neq ax' \bmod p$ because p is prime. Hence, starting with a fixed non-zero element a, we have that the set $\{ax \bmod p \mid x \in [0, p)\}$ has cardinality p and must contain 1; thus, a must have a multiplicative inverse.

4.2 Hash families in a field

Within a field, we can easily construct hash families having strong theoretical guarantees, as the next lemma illustrates.

Lemma 4 *The family of functions of the form*

$$h(x) = ax$$

in a finite field ($\mathrm{GF}(p^n)$) *is* Δ*-universal, provided that the key* a *is picked from all values of the field.*

As another example, consider hash functions of the form $h(x_1, x_2, \ldots, x_m) = a^{m-1}x_1 + a^{m-2}x_2 + \cdots + x_m$ where a is picked at random (a *random input*). Such *polynomial hash functions* can be computed efficiently using Horner's rule: starting with $r = x_1$, compute $r \leftarrow ar + x_i$ for $i = 2, \ldots, m$. Given any two distinct inputs, $x_1, x_2, \ldots, x_m$ and $x'_1, x'_2, \ldots, x'_m$, we have that $h(x_1, \ldots, x_m) - h(x'_1, \ldots, x'_m)$ is a non-zero polynomial of degree at most $m - 1$ in a. By the fundamental theorem of algebra, we have that it is zero for at most $m - 1$ distinct values of a. Thus, we have that the probability of a collision is bounded by $(m - 1)/p^n$ where p^n is the cardinality of the field. For example, VHASH uses polynomial hashing with $p = 2^{127} - 1$ and $n = 1$.

We can further reduce the collision probabilities if we use m random inputs $a_1, \ldots, a_m$ picked in the field to compute a *multilinear* function: $h(x_1, \ldots, x_m) = a_1x_1 + a_2x_2 + \cdots + a_mx_m$. We have Δ-universality. Given two distinct inputs,

[2] In the present paper, $\log n$ means $\log_2 n$.

$x_1, \ldots, x_m$ and $x'_1, \ldots, x'_m$, we have that $x_i \neq x'_i$ for some i. Thus, we have that $h(x_1, \ldots, x_m) = c + h(x'_1, \ldots, x'_m)$ if and only if $a_i = (x_i - x'_i)^{-1}(c + \sum_{j \neq i} a_j(x'_j - x_j))$.

If m is even, we can get the same bound on the collision probability with half the number of multiplications [7,26,29]:

$$\begin{aligned} &h(x_1, x_2, \ldots, x_m) \\ &\quad = (a_1+x_1)(a_2+x_2)+\cdots+(a_{m-1}+x_{m-1})(a_m+x_m). \end{aligned}$$

The argument is similar. Consider that

$$\begin{aligned} &(x_i + a_i)(a_{i+1} + x_{i+1}) - (x'_i + a_i)(a_{i+1} + x'_{i+1}) \\ &\quad = a_{i+1}(x_i - x'_i) + a_i(x_{i+1} - x'_{i+1}) + x_{i+1}x_i + x'_i x'_{i+1}. \end{aligned}$$

Take two distinct inputs, $x_1, x_2, \ldots, x_m$ and $x'_1, x'_2, \ldots, x'_m$. As before, we have that $x_i \neq x'_i$ for some i. Without loss of generality, assume that i is odd; then we can find a unique solution for a_{i+1}: to do this, start from $h(x_1, \ldots, x_m) = c + h(x'_1, \ldots, x'_m)$ and solve for $a_{i+1}(x_i - x'_i)$ in terms of an expression that does not depend on a_{i+1}. Then use the fact that $x_i - x'_i$ has an inverse. This shows that the collision probability is bounded by $1/p^n$ and we have Δ-universality.

Lemma 5 *Given an even number m, the family of functions of the form*

$$\begin{aligned} h(x_1, x_2, \ldots, x_m) = (a_1 + x_1)(a_2 + x_2) \\ + (a_3 + x_3)(a_4 + x_4) \\ + \cdots + (a_{m-1} + x_{m-1})(a_m + x_m) \end{aligned}$$

in a finite field ($\mathrm{GF}(p^n)$) *is Δ-universal, provided that the keys $a_1, \ldots, a_m$ are picked from all values of the field. In particular, the collision probability between two distinct inputs is bounded by $1/p^n$.*

4.3 Binary finite fields

Finite fields having prime cardinality are simple (see Sect. 4.1), but we would prefer to work with fields having a power-of-two cardinality (also called binary fields) to match common computer architectures. Specifically, we are interested in $\mathrm{GF}(2^{64})$ because our desktop processors typically have 64-bit architectures.

We can implement such a field over the integers in $[0, 2^L)$ by using the following two operations. Addition is defined as the bitwise XOR ($\oplus$) operation, which is fast on most computers:

$$a +_{\mathrm{GF}(2^L)} b \equiv a \oplus b.$$

The number 0 is the additive identity element ($a \oplus 0 = 0 \oplus a = a$), and every number is its own additive inverse: $a \oplus a = 0$. Note that because binary finite fields use XOR as an addition, Δ-universality and XOR-universality are effectively equivalent for our purposes in binary finite fields.

Multiplication is defined as a carry-less multiplication followed by a reduction. We use the convention that a_i is the ith least significant bit of integer a and $a_i = 0$ if i is larger than the most significant bit of a. The ith bit of the carry-less multiplication $a \star b$ of a and b is given by

$$(a \star b)_i \equiv \bigoplus_{k=0}^{i} a_{i-k} b_k, \qquad (2)$$

where $a_{i-k}b_k$ is just a regular multiplication between two integers in $\{0, 1\}$ and $\bigoplus_{k=0}^{i}$ is the bitwise XOR of a range of values. The carry-less product of two L-bit integers is a $2L$-bit integer.

We can check that the integers with $\oplus$ as addition and $\star$ as multiplication form a *ring*: addition and multiplication are associative, commutative and distributive, and there is an additive identity element. In this instance, the number 1 is a multiplicative identity element ($a \star 1 = 1 \star a = a$). Except for the number 1, no number has a multiplicative inverse in this ring.

Given the ring determined by $\oplus$ and $\star$, we can derive a corresponding finite field. However, just as with finite fields of prime cardinality, we need some kind of modular reduction and a concept equivalent to that of prime numbers.[3]

Let us define degree(x) to be the position of the most significant non-zero bit of x, starting at 0 (e.g., degree$(1) = 0$, degree$(2) = 1$, degree$(2^j) = j$). For example, we have degree$(x) \leq 127$ for any 128-bit integer x. Given any two non-zero integers a, b, we have that degree$(a \star b) =$ degree$(a) +$ degree(b) as a straightforward consequence of Eq. 2. Similarly, we have that

$$\mathrm{degree}(a \oplus b) \leq \max(\mathrm{degree}(a), \mathrm{degree}(b)).$$

Not unlike regular multiplication, given integers a, b with $b \neq 0$, there are unique integers α, β (henceforth, the *quotient* and the *remainder*) such that

$$a = \alpha \star b \;\oplus\; \beta, \qquad (3)$$

where degree$(\beta) <$ degree(b).

The uniqueness of the quotient and the remainder is easily shown. Suppose that there is another pair of values α', β' with the same property. Then $\alpha' \star b \;\oplus\; \beta' = \alpha \star b \;\oplus\; \beta$,

[3] The general construction of a finite field of cardinality p^n for $n > 1$ is commonly explained in terms of polynomials with coefficients from $\mathrm{GF}(p)$. To avoid unnecessary abstraction, we present finite fields of cardinality 2^L using regular L-bit integers. Interested readers can see Mullen and Panario [30], for the alternative development.

which implies that $(\alpha' \oplus \alpha) \star b = \beta' \oplus \beta$. However, since degree$(\beta' \oplus \beta) <$ degree(b), we must have that $\alpha = \alpha'$. From this, it follows that $\beta = \beta'$, thus establishing uniqueness.

We define $\div$ and mod operators as giving, respectively, the quotient $(a \div b = \alpha)$ and remainder $(a \bmod b = \beta)$ so that the equation

$$a \equiv a \div b \star b \quad \oplus \quad a \bmod b \tag{4}$$

is an identity equivalent to Eq. 3 (To avoid unnecessary parentheses, we use the following operator precedence convention: $\star$, mod and $\div$ are executed first, from left to right, followed by $\oplus$).

In the general case, we can compute $a \div b$ and $a \bmod b$ using a straightforward variation on the Euclidean division algorithm (see Algorithm 2) which proves the existence of the remainder and quotient. Checking the correctness of the algorithm is straightforward. We start initially with values α and β such that $a = \alpha \star b \oplus \beta$. By inspection, this equality is preserved throughout the algorithm. Meanwhile, the algorithm only terminates when the degree of β is less than that of b, as required. The algorithm must terminate, since the degree of q is reduced by at least one, each time it is updated (for a maximum of degree(a) − degree$(b) + 1$ steps).

Algorithm 2 Carry-less division algorithm

```
input: Two integers $a$ and $b$, where $b$ must be non-zero
output: Carry-less quotient and remainder: $\alpha = a \div b$ and $\beta = a \bmod b$, such that $a = \alpha \star b \oplus \beta$ and degree$(\beta) < b$
Let $\alpha \leftarrow 0$ and $\beta \leftarrow a$
while degree$(\beta) \geq$ degree$(b)$ do
    let $x \leftarrow 2^{\mathrm{degree}(\beta) - \mathrm{degree}(b)}$
    $\alpha \leftarrow x \oplus \alpha$, $\beta \leftarrow x \star b \oplus \beta$
end while
return $\alpha$ and $\beta$
```

Given $a = \alpha \star b \oplus \beta$ and $a' = \alpha' \star b \oplus \beta'$, we have that $a \oplus a' = (\alpha \oplus \alpha') \star b \oplus (\beta \oplus \beta')$. Thus, it can be checked that divisions and modular reductions are distributive:

$$(a \oplus b) \bmod p = (a \bmod p) \oplus (b \bmod p), \tag{5}$$

$$(a \oplus b) \div p = (a \div p) \oplus (b \div p). \tag{6}$$

Thus, we have $(a \oplus b) \bmod p = 0 \Rightarrow a \bmod p = b \bmod p$. Moreover, by inspection, we have that degree$(a \bmod b) <$ degree(b) and degree$(a \div b) =$ degree(a) − degree(b).

The carry-less multiplication by a power of two is equivalent to regular multiplication. For this reason, a modular reduction by a power of two (e.g., $a \bmod 2^{64}$) is just the regular integer modular reduction. Idem for division.

There are non-zero integers a such that there is no integer b other than 1 for which $a \bmod b = 0$; effectively, a is a prime number under the carry-less multiplication interpretation. These "prime integers" are more commonly known as *irreducible polynomials* in the ring of polynomials GF2$[x]$, so we call them *irreducible* instead of prime. Let us pick such an irreducible integer p (arbitrarily) such that the degree of p is 64. One such integer is $2^{64} + 2^4 + 2^3 + 2 + 1$. Then we can finally define the multiplication operation in GF(2^{64}):

$$a \times_{\mathrm{GF}(2^{64})} b \equiv (a \star b) \bmod p.$$

Coupled with the addition $+_{\mathrm{GF}(2^{64})}$ that is just a bitwise XOR, we have an implementation of the field GF(2^{64}) over integers in $[0, 2^{64})$.

We call the index of the second most significant bit the *subdegree*. We chose an irreducible p of degree 64 having minimal subdegree (4).[4] We use the fact that this subdegree is small to accelerate the computation of the modular reduction in the next section.

4.4 Efficient reduction in GF(2^{64})

AMD and Intel have introduced a fast instruction that can compute a carry-less multiplication between two 64-bit numbers, and it generates a 128-bit integer. To get the multiplication in GF(2^{64}), we must still reduce this 128-bit integer to a 64-bit integer. Since there is no equivalent fast modular instruction, we need to derive an efficient algorithm.

There are efficient reduction algorithms used in cryptography (e.g., from 256-bit to 128-bit integers [17]), but they do not suit our purposes: we have to reduce to 64-bit integers. Inspired by the classical Barrett reduction [5], Knežević et al. proposed a generic modular reduction algorithm in GF(2^n), using no more than two multiplications [22]. We put this to good use in previous work [26]. However, we can do the same reduction using a single multiplication. According to our tests, the reduction technique presented next is 30 % faster than an optimized implementation based on Knežević et al.'s algorithm.

Let us write $p = 2^{64} \oplus r$. In our case, we have $r = 2^4 + 2^3 + 2 + 1 = 27$ and degree$(r) = 4$. We are interested in applying a modular reduction by p to the result of the multiplication of two integers in $[0, 2^{64})$, and the result of such a multiplication is an integer x such that degree$(x) \leq 127$. We want to compute $x \bmod p$ quickly. We begin with the following lemma.

Lemma 6 *Consider any 64-bit integer $p = 2^{64} \oplus r$. We define the operations* mod *and* $\div$ *as the counterparts of the carry-less multiplication* $\star$ *as in Sect.* 4.3. *Given any x, we have that*

[4] This can be readily verified using a mathematical software package such as Sage or Maple.

$$x \bmod p = ((z \div 2^{64}) \star 2^{64}) \bmod p \oplus z \bmod 2^{64} \oplus x \bmod 2^{64},$$

where $z \equiv (x \div 2^{64}) \star r$.

Proof We have that $x = (x \div 2^{64}) \star 2^{64} \oplus x \bmod 2^{64}$ for any x by definition. Applying the modular reduction on both sides of the equality, we get

$$\begin{aligned} x \bmod p &= (x \div 2^{64}) \star 2^{64} \bmod p \oplus x \bmod 2^{64} \bmod p \\ &= (x \div 2^{64}) \star 2^{64} \bmod p \oplus x \bmod 2^{64} \\ &\quad \text{by Fact 1} \\ &= (x \div 2^{64}) \star r \bmod p \oplus x \bmod 2^{64} \\ &\quad \text{by Fact 2} \\ &= z \bmod p \oplus x \bmod 2^{64} \\ &\quad \text{by } z'\text{s def}. \\ &= ((z \div 2^{64}) \star 2^{64}) \bmod p \oplus z \bmod 2^{64} \\ &\quad \oplus x \bmod 2^{64} \\ &\quad \text{by Fact 3,} \end{aligned}$$

where Facts 1, 2 and 3 are as follows:

- (Fact 1) For any x, we have that $(x \bmod 2^{64}) \bmod p = x \bmod 2^{64}$.
- (Fact 2) For any integer z, we have that $(2^{64} \oplus r) \star z \bmod p = p \star z \bmod p = 0$ and therefore

  $$2^{64} \star z \bmod p = r \star z \bmod p$$

  by the distributivity of the modular reduction (Eq. 5).
- (Fact 3) Recall that by definition $z = (z \div 2^{64}) \star 2^{64} \oplus z \bmod 2^{64}$. We can substitute this equation in the equation from Fact 1. For any z and any non-zero p, we have that

  $$\begin{aligned} z \bmod p &= ((z \div 2^{64}) \star 2^{64} \oplus z \bmod 2^{64}) \bmod p \\ &= ((z \div 2^{64}) \star 2^{64}) \bmod p \oplus z \bmod 2^{64} \end{aligned}$$

  by the distributivity of the modular reduction (see Eq. 5).

Hence, the result is shown. □

Lemma 6 provides a formula to compute $x \bmod p$. Computing $z = (x \div 2^{64}) \star r$ involves a carry-less multiplication, which can be done efficiently on recent Intel and AMD processors. The computation of $z \bmod 2^{64}$ and $x \bmod 2^{64}$ is trivial. It remains to compute $((z \div 2^{64}) \star 2^{64}) \bmod p$. At first glance, we still have a modular reduction. However, we can easily memoize the result of $((z \div 2^{64}) \star 2^{64}) \bmod p$. The next lemma shows that there are only 16 distinct values to memoize (this follows from the low subdegree of p).

Lemma 7 *Given that x has degree less than 128, there are only 16 possible values of $(z \div 2^{64}) \star 2^{64} \bmod p$, where $z \equiv (x \div 2^{64}) \star r$ and $r = 2^4 + 2^3 + 2 + 1$.*

Proof Indeed, we have that

$$\text{degree}(z) = \text{degree}(x) - 64 + \text{degree}(r).$$

Because $\text{degree}(x) \leq 127$, we have that $\text{degree}(z) \leq 127 - 64 + 4 = 67$. Therefore, we have $\text{degree}(z \div 2^{64}) \leq 3$. Hence, we can represent $z \div 2^{64}$ using 4 bits: there are only 16 4-bit integers. □

Thus, in the worst possible case, we would need to memoize 16 distinct 128-bit integers to represent $((z \div 2^{64}) \star 2^{64}) \bmod p$. However, observe that the degree of $z \div 2^{64}$ is bounded by $\text{degree}(x) - 64 + 4 - 64 \leq 127 - 128 + 4 = 3$ since $\text{degree}(x) \leq 127$. By using Lemma 8, we show that each integer $((z \div 2^{64}) \star 2^{64}) \bmod p$ has degree bounded by 7, so that it can be represented using no more than 8 bits: setting $L = 64$ and $w \equiv z \div 2^{64}$, $\text{degree}(w) \leq 3$, $\text{degree}(r) = 4$ and $\text{degree}(w) + \text{degree}(r) \leq 7$.

Effectively, the lemma says that if you take a value of small degree w, you multiply it by 2^L and then compute the modular reduction on the result and a value p that is almost 2^L (except for a value of small degree r), then the result has small degree: it is bounded by the sum of the degrees of w and r.

Lemma 8 *Consider $p = 2^L \oplus r$, with r of degree less than L. For any w, the degree of $w \star 2^L \bmod p$ is bounded by* $\text{degree}(w) + \text{degree}(r)$.

Moreover, when $\text{degree}(w) + \text{degree}(r) < L$ *then the degree of $w \star 2^L \bmod p$ is exactly* $\text{degree}(w) + \text{degree}(r)$.

Proof The result is trivial if $\text{degree}(w) + \text{degree}(r) \geq L$, since the degree of $w \star 2^L \bmod p$ must be smaller than the degree of p.

So let us assume that $\text{degree}(w) + \text{degree}(r) < L$. By the definition of the modular reduction (Eq. 4), we have

$$w \star 2^L = w \star 2^L \div p \star p \oplus w \star 2^L \bmod p.$$

Let $w' = w \star 2^L \div p$, then

$$\begin{aligned} w \star 2^L &= w' \star p \oplus w \star 2^L \bmod p \\ &= w' \star r \oplus w' \star 2^L \oplus w \star 2^L \bmod p. \end{aligned}$$

The first L bits of $w \star 2^L$ and $w' \star 2^L$ are zero. Therefore, we have

$$(w' \star r) \bmod 2^L = (w \star 2^L \bmod p) \bmod 2^L.$$

Table 3 Values of $w \star 2^{64} \bmod p$ for $w = 0, 1, \ldots, 15$ given $p = 2^{64} + 2^4 + 2^3 + 3$

w		$w \star 2^{64} \bmod p$	
Decimal	Binary	Decimal	Binary
0	0000_2	0	00000000_2
1	0001_2	27	00011011_2
2	0010_2	54	00110110_2
3	0011_2	45	00101101_2
4	0100_2	108	01101100_2
5	0101_2	119	01110111_2
6	0110_2	90	01011010_2
7	0111_2	65	01000001_2
8	1000_2	216	11011000_2
9	1001_2	195	11000011_2
10	1010_2	238	11101110_2
11	1011_2	245	11110101_2
12	1100_2	180	10110100_2
13	1101_2	175	10101111_2
14	1110_2	130	10000010_2
15	1111_2	153	10011001_2

Moreover, the degree of w' is the same as the degree of w: $\text{degree}(w') = \text{degree}(w) + \text{degree}(2^L) + \text{degree}(p) = \text{degree}(w) + L - L = \text{degree}(w)$. Hence, we have $\text{degree}(w' \star r) = \text{degree}(w) + \text{degree}(r) < L$ and, of course, $\text{degree}(w \star 2^L \bmod p) < L$. Thus, we have that

$$w' \star r = w \star 2^L \bmod p.$$

Hence, it follows that $\text{degree}(w \star 2^L \bmod p) = \text{degree}(w' \star r) = \text{degree}(w) + \text{degree}(r)$. □

Thus, the memoization requires access to only 16 8-bit values. We enumerate the values in question ($w \star 2^{64} \bmod p$ for $w = 0, 1, \ldots, 15$) in Table 3. It is convenient that $16 \times 8 = 128$ bits: the entire table fits in a 128-bit word. It means that if the list of 8-bit values are stored using one byte each, the SSSE3 instruction pshufb can be used for fast look-up. (See Algorithm 3.)

Algorithm 3 Carry-less division algorithm

```
input: A 128-bit integer $a$
output: Carry-less modular reduction $a \bmod p$ where $p = 2^{64} + 27$
$z \leftarrow (a \div 2^{64}) \star r$
$w \leftarrow z \div 2^{64}$
Look up $w \star 2^{64} \bmod p$ in Table 3, store result in $y$
return $a \bmod 2^{64} \oplus z \bmod 2^{64} \oplus y$
```

Corresponding C implementation using x64 intrinsics:

```
uint64_t  modulo( __m128i a) {
        __m128i r = _mm_cvtsi64_si128(27);
        __m128i z =
          _mm_clmulepi64_si128( a, r, 0x01);
        __m128i table = _mm_setr_epi8(0, 27, 54,
          45, 108,119, 90, 65, 216, 195, 238,
          245,180, 175, 130, 153);
        __m128i y =
          _mm_shuffle_epi8(table
           ,_mm_srli_si128(z,8));
        __m128i temp1 = _mm_xor_si128(z,a);
        return _mm_cvtsi128_si64(
          _mm_xor_si128(temp1,y));
}
```

5 CLHASH

The CLHASH family resembles the VHASH family—except that members work in a binary finite field. The VHASH family has the 128-bit NH family (see Eq. 1), but we instead use the 128-bit CLNH family:

$$\text{CLNH}(s) = \bigoplus_{i=1}^{l/2} ((s_{2i-1} \oplus k_{2i-1}) \star (s_{2i} \oplus k_{2i})), \qquad (7)$$

where the s_i and k_i's are 64-bit integers and l is the length of the string s. The formula assumes that l is even: we pad odd-length inputs with a single zero word. When an input string M is made of $|M|$ bytes, we can consider it as string of 64-bit words s by padding it with up to 7 zero bytes so that $|M|$ is divisible by 8.

On x64 processors with the CLMUL instruction set, a single term $((s_{2i-1} \oplus k_{2i-1}) \star (s_{2i} \oplus k_{2i}))$ can be computed using one 128-bit XOR instructions (pxor in SSE2) and one carry-less multiplication using the pclmulqdq instruction:

- load (k_{2i-1}, k_{2i}) in a 128-bit word,
- load (s_{2i-1}, s_{2i}) in another 128-bit word,
- compute

  $$(k_{2i-1}, k_{2i}) \oplus (s_{2i-1}, s_{2i}) \equiv (k_{2i-1} \oplus s_{2i-1}, k_{2i} \oplus s_{2i})$$

  using one pxor instruction,
- compute $(k_{2i-1} \oplus s_{2i-1}) \star (k_{2i} \oplus s_{2i})$ using one pclmulqdq instruction (result is a 128-bit word).

An additional pxor instruction is required per pair of words to compute CLNH, since we need to aggregate the results.

We have that the family $s \to \text{CLNH}(s) \bmod p$ for some irreducible p of degree 64 is XOR universal over same-length strings. Indeed, Δ-universality in the field $GF(2^{64})$ follows from Lemma 5. However, recall that Δ-universality in a binary finite field (with operations $\star$ and $\oplus$ for multiplication and addition) is the same as XOR universality—addition is the XOR operation ($\oplus$). It follows that the CLNH family must be $1/2^{64}$-almost universal for same-length strings.

Given an arbitrarily long string of 64-bit words, we can divide it up into blocks of 128 words (padding the last block with zeros if needed). Each block can be hashed using CLNH and the result is $1/2^{64}$-almost universal by Lemma 2. If there is a single block, we can compute $\mathrm{CLNH}(s) \bmod p$ to get an XOR universal hash value. Otherwise, the resulting 128-bit hash values $a_1, a_2, \ldots, a_n$ can then be hashed once more. For this, we use a polynomial hash function, $k^{n-1}a_1 + k^{n-2}a_2 + \cdots + a_n$, for some random input k in some finite field. We choose the field $\mathrm{GF}(2^{127})$ and use the irreducible $p = 2^{127} + 2 + 1$. We compute such a polynomial hash function by using Horner's rule: starting with $r = a_1$, compute $r \leftarrow k \star r \oplus a_i$ for $i = 2, 3, \ldots, n$. For this purpose, we need carry-less multiplications between pairs of 128-bit integers: we can achieve the desired result with 4 `pclmulqdq` instructions, in addition to some shift and XOR operations. The multiplication generates a 256-bit integer x that must be reduced. However, it is not necessary to reduce it to a 127-bit integer (which would be the result if we applied a modular reduction by $2^{127} + 2 + 1$). It is enough to reduce it to a 128-bit integer x' such that $x' \bmod (2^{127} + 2 + 1) = x \bmod (2^{127} + 2 + 1)$. We get the desired result by setting x' equal to the *lazy* modular reduction [8] $x \bmod_{\mathrm{lazy}} (2^{127} + 2 + 1)$ defined as

$$\begin{aligned} & x \bmod_{\mathrm{lazy}} (2^{127} + 2 + 1) \\ & \quad \equiv x \bmod 2^{128} \oplus (x \div 2^{128}) \star 2 \oplus (x \div 2^{128}) \star 1. \end{aligned} \tag{8}$$

It is computationally convenient to assume that $\mathrm{degree}(x) \leq 256 - 2$ so that $\mathrm{degree}((x \div 2^{128}) \star 2) \leq 128$. We can achieve this degree bound by picking the polynomial coefficient k to have $\mathrm{degree}(k) \leq 128 - 2$. The resulting polynomial hash family is $(n-1)/2^{126}$-almost universal for strings having the same length where n is the number of 128-word blocks ($\lceil |M|/1024 \rceil$ where $|M|$ is the string length in bytes), whether we use the actual modular or the lazy modular reduction.

It remains to reduce the final output $\mathcal{O}$ (stored in a 128-bit word) to a 64-bit hash value. For this purpose, we can use $s \to \mathrm{CLNH}(s) \bmod p$ with $p = 2^{64} + 27$ (see Sect. 4.4), and where k'' is a random 64-bit integer. We treat $\mathcal{O}$ as a string containing two 64-bit words. Once more, the reduction is XOR universal by an application of Lemma 5. Thus, we have the composition of three hash functions with collision probabilities $1/2^{64}$, $(n-1)/2^{126}$ and $1/2^{64}$. It is reasonable to bound the string length by 2^{64} bytes: $n \leq 2^{64}/1024 = 2^{54}$. We have that $2/2^{64} + (2^{54} - 1)/2^{126} < 2.004/2^{64}$. Thus, for same-length strings, we have $2.004/2^{64}$-almost XOR universality.

We further ensure that the result is XOR-universal over all strings: $P(h(s) = h(s') \oplus c) \leq 1/2^{64}$ irrespective of whether $|s| = |s'|$. By Lemma 3, it suffices to XOR the hash value with $k'' \star |M| \bmod p$ where k'' is a random 64-bit integer, $|M|$ is the string length expressed as a 64-bit integer, and $p = 2^{64} + 27$. The XOR universality follows for strings having different lengths by Lemma 4 and the equivalence between XOR-universality and Δ-universality in binary finite fields. As a practical matter, since the final step involves the same modular reduction twice in the expression $(\mathrm{CLNH}(s) \bmod p) \oplus ((k'' \star |M|) \bmod p)$, we can simplify it to $(\mathrm{CLNH}(s) \oplus (k'' \star |M|)) \bmod p$, thus avoiding an unnecessary modular reduction.

Table 4 Comparison between the two 64-bit hash families VHASH and CLHASH

	Universality	Input length
VHASH	$\frac{1}{2^{61}}$-almost Δ-universal	1–2^{59} bytes
CLHASH	XOR universal	1–1024 bytes
	$\frac{2.004}{2^{64}}$-almost XOR universal	1025–2^{64} bytes

Our analysis is summarized by the following lemma.

Lemma 9 CLHASH *is* $2.004/2^{64}$*-almost XOR universal over strings of up to* 2^{64} *bytes. Moreover, it is XOR universal over strings of no more than* 1 kB.

The bound of the collision probability of CLHASH for long strings ($2.004/2^{64}$) is four times lower than the corresponding VHASH collision probability ($1/2^{61}$). For short strings (1 kB or less), CLHASH has a bound that is eight times lower. See Table 4 for a comparison. CLHASH is given by Algorithm 4.

Algorithm 4 CLHASH algorithm: all operations are carry-less, as per § 4.3. The $\gg$ operator indicates a left shift: $\mathcal{O} \gg 33$ is the value $\mathcal{O}$ divided by 2^{33}.

```
Require: 128 randomly picked 64-bit integers $k_1, k_2, \ldots, k_{128}$
    defining a 128-bit CLNH hash function (see Equation 7) over in-
    puts of length 128
Require: $k$, a randomly picked 126-bit integer
Require: $k'$, a randomly picked 128-bit integer
Require: $k''$, a randomly picked 64-bit integer
input: string $M$ made of $|M|$ bytes
if $|M| \leq 1024$ then
    $\mathcal{O} \leftarrow \mathrm{CLNH}(M) \oplus (k'' \star |M|) \bmod (2^{64} + 27)$
    return $\mathcal{O}$
else
    Let $n$ be the number of 128-word blocks ($\lceil |M|/1024 \rceil$).
    Let $M_i$ be the substring of $M$ from index $128i$ to $128i + 127$
    inclusively, padding with zeros if needed.
    Hash each $M_i$ using the CLNH function, labelling the
    resulting 128-bit results $a_i$ for $i = 1, \ldots, n$. That is, $a_i \leftarrow$
    $\mathrm{CLNH}(M_i)$.
    Hash the resulting $a_i$ with a polynomial hash function and store
    the value in a 128-bit hash value $\mathcal{O}$: $\mathcal{O} \leftarrow a_1 \star k^{n-1} \oplus \cdots \oplus$
    $a_n \bmod_{\mathrm{lazy}} (2^{127} + 2 + 1)$ (see Equation 8).
    Hash the 128-bit value $\mathcal{O}$, treating it as two 64-bit words
    $(\mathcal{O}_1, \mathcal{O}_2)$, down to a 64-bit CLNH hash value (with the addition
    of a term accounting for the length $|M|$ in bytes)

        $z \leftarrow ((\mathcal{O}_1 \oplus k'_1) \star (\mathcal{O}_2 \oplus k'_2) \oplus (k'' \star |M|) \bmod (2^{64} + 27).$

    Values $k'_1$ and $k'_2$ are the two 64-bit words contained in $k'$.
    return the 64-bit hash value $z$
end if
```

5.1 Random bits

One might wonder whether using 1kB of random bits is necessary. For strings of no more than 1 kB, CLHASH is XOR universal. Stinson showed that in such cases, we need the number of random bits to match the input length [37]. That is, we need at least 1 kB to achieve XOR universality over strings having 1 kB. Hence, CLHASH makes nearly optimal use of the random bits.

6 Statistical validation

Classically, hash functions have been deterministic: fixed maps h from U to V, where $|U| \gg |V|$ and thus collisions are inevitable. Hash functions might be assessed according to whether their outputs are distributed evenly, i.e., whether $|h^{-1}(x)| \approx |h^{-1}(y)|$ for two distinct $x, y \in V$. However, in practice, the actual input is likely to consist of clusters of nearly identical keys [23]: for instance, symbol table entries such as `temp1`, `temp2`, `temp3` are to be expected, or a collection of measured data values is likely to contain clusters of similar numeric values. Appending an extra character to the end of an input string, or flipping a bit in an input number, should (usually) result in a different hash value. A collection of desirable properties can be defined, and then hash functions rated on their performance on data that is meant to represent realistic cases.

One common use of randomized hashing is to avoid DoS (denial-of-service) attacks when an adversary controls the series of keys submitted to a hash table. In this setting, prior to the use of a hash table, a random selection of hash function is made from the family. The (deterministic) function is then used, at least until the number of collisions is observed to be too high. A high number of collisions presumably indicates that the hash table needs to be resized, although it could indicate that an undesirable member of the family had been chosen. Those contemplating switching from deterministic hash tables to randomized hash tables would like to know that the typical performance would not degrade much. Yet, as carefully tuned deterministic functions can sometimes outperform random assignments for typical inputs [23], some degradation might need to be tolerated. Thus, it is worth checking a few randomly chosen members of our CLHASH families against statistical tests.

6.1 SMHasher

The SMHasher program [1] includes a variety of quality tests on a number of minimally randomized hashing algorithms, for which we have weak or no known theoretical guarantees. It runs several statistical tests, such as the following.

- Given a randomly generated input, changing a few bits at random should not generate a collision.
- Among all inputs containing only two non-zero bytes (and having a fixed length in [4, 20]), collisions should be unlikely (called the *TwoBytes* test).
- Changing a single bit in the input should change half the bits of the hash value, on average [13] (sometimes called the *avalanche effect*).

Some of these tests are demanding: e.g., CityHash [35] fails the *TwoBytes* test.

We added both VHASH and CLHASH to SMHasher and used the Mersenne Twister (i.e., MT19937) to generate the random bits [28]. We find that VHASH passes all tests. However, CLHASH fails one of them: the avalanche test. We can illustrate the failure. Consider that for short fixed-length strings (8 bytes or less), CLHASH is effectively equivalent to a hash function of the form $h(x) = a \star x \bmod p$, where p is irreducible. Such hash functions form an XOR universal family. They also satisfy the identity $h(x \oplus y) \oplus h(x) = h(y)$. It follows that no matter what value x takes, modifying the same ith bit modifies the resulting hash value in a consistent manner (according to $h(2^{i+1})$). We can still expect that changing a bit in the input changes half the bits of the hash value on average. However, SMHasher checks that $h(x \oplus 2^{i+1})$ differs from $h(x)$ in any given bit about half the time over many randomly chosen inputs x. Since $h(x \oplus 2^{i+1}) \oplus h(x)$ is independent of x for short inputs with CLHASH, any given bit is either always flipped (for all x) or never. Hence, CLHASH fails the SMHasher test.

Thankfully, we can slightly modify CLHASH so that all tests pass if we so desire. It suffices to apply an additional *bit mixing* function taken from MurmurHash [1] to the result of CLHASH. The function consists of two multiplications and three shifts over 64-bit integers:

$$\begin{aligned}
&x \leftarrow x \oplus (x \gg 33),\\
&x \leftarrow x \times 18397679294719823053,\\
&x \leftarrow x \oplus (x \gg 33),\\
&x \leftarrow x \times 14181476777654086739,\\
&x \leftarrow x \oplus (x \gg 33).
\end{aligned}$$

Each step is a bijection: e.g., multiplication by an odd integer is always invertible. A bijection does not affect collision bounds.

7 Speed experiments

We implemented a performance benchmark in C and compiled our software using GNU GCC 4.8 with the `-O2` flag. The benchmark program ran on a Linux server with an Intel i7-4770 processor running at 3.4 GHz. This CPU has 32 kB of L1 cache, 256 kB of L2 cache per core, and 8 MB of L3 cache shared by all cores. The machine has 32 GB of RAM (DDR3-1600 with double channel). We disabled Turbo Boost and set the processor to run only at its highest clock speed, effectively disabling the processor's power management. All timings are done using the time-stamp counter (`rdtsc`) instruction [34]. Although all our software[5] is single threaded, we disabled hyper-threading as well.

Our experiments compare implementations of CLHASH, VHASH, SipHash [3], GHASH [17] and Google's CityHash.

- We implemented CLHASH using Intel intrinsics. As described in Sect. 5, we use various *single instruction, multiple data* (SIMD) instructions (e.g., SSE2, SSE3 and SSSE3) in addition to the CLMUL instruction set. The random bits are stored consecutively in memory, aligned with a cache line (64 bytes).
- For VHASH, we used the authors' 64-bit implementation [25], which is optimized with inline assembly. It stores the random bits in a C `struct`, and we do not include the overhead of constructing this `struct` in the timings. The authors assume that the input length is divisible by 16 bytes, or padded with zeros to the nearest 16-byte boundary. In some instances, we would need to copy part of the input to a new location prior to hashing the content to satisfy the requirement. Instead, we decided to optimistically hash the data in-place without copy. Thus, we slightly overestimate the speed of the VHASH implementation—especially on shorter strings.
- We used the reference C implementation of SipHash [4]. SipHash is a fast family of 64-bit pseudorandom hash functions adopted, among others, by the Python language.
- CityHash is commonly used in applications where high speed is desirable [15,27]. We wrote a simple C port of Google's CityHash (version 1.1.1) [35]. Specifically, we benchmarked the `CityHash64WithSeed` function.
- Using Gueron and Kounavis' [17] code, we implemented a fast version of GHASH accelerated with the CLMUL instruction set. GHASH is a polynomial hash function over GF(2^{128}) using the irreducible polynomial $x^{128} + x^7 + x^2 + x + 1$: $h(x_1, x_2, \ldots, x_n) = a^n x_1 + a^{n-1} x_2 + \cdots + a x_n$ for some 128-bit key a. To accelerate computations, Gueron and Kounavis replace the traditional Horner's rule with an extended version that processes input words four at a time: starting with $r = 0$ and precomputed powers a^2, a^3, a^4, compute $r \leftarrow a^4(r + x_i) + a^3 x_{i+1} + a^2 x_{i+2} + a x_{i+3}$ for $i = 1, 4, \ldots, 4\lfloor m/4 \rfloor - 3$. We complete the computation with the usual Horner's rule when the number of input words is not divisible by four. In contrast with other hash functions, GHASH generates 128-bit hash values.

Table 5 A comparison of estimated CPU cycles per byte on a Haswell Intel processor using 4 kB inputs

Scheme	64 B input	4 kB input
VHASH	1.0	0.26
CLHASH	**0.45**	**0.16**
CityHash	0.48	0.23
SipHash	3.1	2.1
GHASH	2.3	0.93

All schemes generate 64-bit hash values, except that GHASH generates 128-bit hash values
Bold values indicate the fastest scheme

VHASH, CLHASH and GHASH require random bits. The time spent by the random number generator is excluded from the timings.

7.1 Results

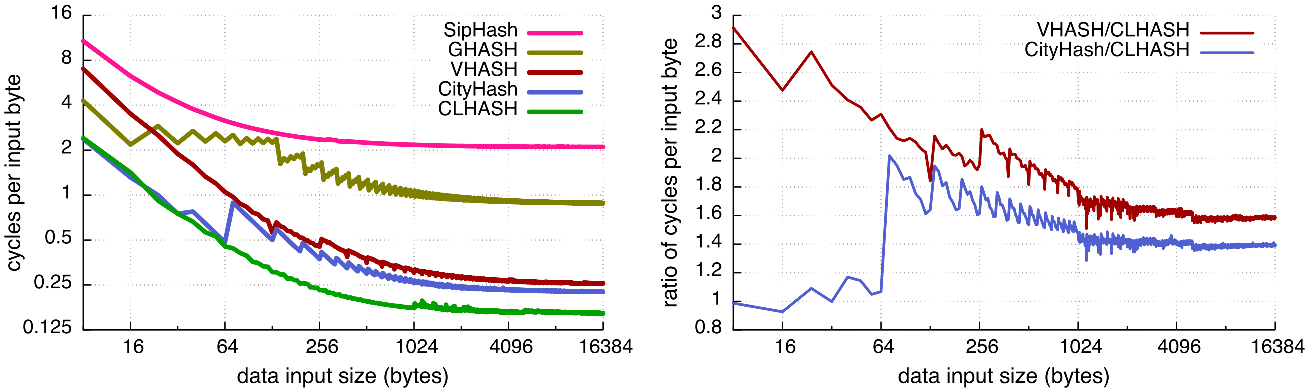
We find that the hashing speed is not sensitive to the content of the inputs—thus, we generated the inputs using a random number generator. For any given input length, we repeatedly hash the strings so that, in total, 40 million input words have been processed.

As a first test, we hashed 64 B and 4 kB inputs (see Table 5) and we report the number of cycles spent to hash one byte: for 4 kB inputs, we got 0.26 for VHASH,[6] 0.16 for CLHASH, 0.23 for CityHash and 0.93 for GHASH. That is, CLHASH is over 60 % faster than VHASH and almost 45 % faster than CityHash. Moreover, SipHash is an order of magnitude slower. Considering that it produces 128-bit hash values, the PCMUL-accelerated GHASH offers good performance: it uses less than one cycle per input byte for long inputs.

Of course, the relative speeds depend on the length of the input. In Fig. 1, we vary the input length from 8 bytes to 8 kB. We see that the results for input lengths of 4 kB are representative. Mostly, we have that CLHASH is 60 % faster than VHASH and 40 % faster than CityHash. However, CityHash and CLHASH have similar performance for small inputs (32 bytes or less), whereas VHASH fares poorly over these same small inputs. We find that SipHash is not competitive in these tests.

[5] Our benchmark software is made freely available under a liberal open-source license (https://github.com/lemire/StronglyUniversalStringHashing), and it includes the modified SMHasher as well as all the necessary software to reproduce our results.

[6] For comparison, Dai and Krovetz reported that VHASH used 0.6 cycles per byte on an Intel Core 2 processor (Merom) [25].



(a) Cycles per input byte

(b) Ratios of cycles per input byte

Fig. 1 Performance comparison for various input lengths. For large inputs, CLHASH is faster, followed in order of decreasing speed by CityHash, VHASH, GHASH and SipHash

7.2 Analysis

From an algorithmic point of view, VHASH and CLHASH are similar. Moreover, VHASH uses a conventional multiplication operation that has lower latency and higher throughput than CLHASH. The VHASH implementation relies on hand-tuned assembly code. Yet, CLHASH is 60 % faster.

For long strings, the bulk of the VHASH computation is spent computing the NH function. When computing NH, each pair of input words (or 16 bytes) uses the following instructions: one `mulq`, three `adds` and one `adc`. Both `mulq` and `adc` generate two micro-operations (μops) each, so without counting register loading operations, we need at least $3 + 2 \times 2 = 7$ μops to process two words [16]. Yet, Haswell processors, like other recent Intel processors, are apparently limited to a sustained execution of no more than 4μops per cycle. Thus we need at least 7/4 cycles for every 16 bytes. That is, VHASH needs at least 0.11 cycles per byte. Because CLHASH runs at 0.16 cycles per byte on long strings (see Table 5), we have that no implementation of VHASH can surpass our implementation of CLHASH by more than 35 %. Simply put, VHASH requires too many μops.

CLHASH is not similarly limited. For each pair of input 64-bit words, CLNH uses two 128-bit XOR instructions (`pxor`) and one `pclmulqdq` instruction. Each `pxor` uses one (fused) μop, whereas the `pclmulqdq` instruction uses 2 μops for a total of 4μops, versus the 7μops absolutely needed by VHASH. Thus, the number of μops dispatched per cycle is less likely to be a bottleneck for CLHASH. However, the `pclmulqdq` instruction has a throughput of only two cycles per instruction. Thus, we can only process one pair of 64-bit words every two cycles, for a speed of $2/16 = 0.125$ cycles per byte. The measured speed (0.16 cycles per byte) is about 35 % higher than this lower bound according to Table 5. This suggests that our implementation of CLHASH is nearly optimal—at least for long strings. We verified our analysis with the IACA code analyzer [19]. It reports that VHASH is indeed limited by the number of μops that can be dispatched per cycle, unlike CLHASH.

8 Related work

The work that led to the design of the `pclmulqdq` instruction by Gueron and Kounavis [17] introduced efficient algorithms using this instruction, e.g., an algorithm for 128-bit modular reduction in Galois Counter Mode. Since then, the `pclmulqdq` instruction has been used to speed up cryptographic applications. Su and Fan find that the Karatsuba formula becomes especially efficient for software implementations of multiplication in binary finite fields due to the `pclmulqdq` instruction [38]. Bos et al. [9] used the CLMUL instruction set for 256-bit hash functions on the Westmere microarchitecture. Elliptic curve cryptography benefits from the `pclmulqdq` instruction [32,33,39]. Bluhm and Gueron pointed out that the benefits are increased on the Haswell microarchitecture due to the higher throughput and lower latency of the instruction [8].

In previous work, we used the `pclmulqdq` instruction for fast 32-bit random hashing on the Sandy Bridge and Bulldozer architectures [26]. However, our results were disappointing, due in part to the low throughput of the instruction on these older microarchitectures.

9 Conclusion

The `pclmulqdq` instruction on recent Intel processors enables a fast and almost universal 64-bit hashing family (CLHASH). In terms of raw speed, the hash functions from this family can surpass some of the fastest 64-bit hash functions on x64 processors (VHASH and CityHash). Moreover, CLHASH offers superior bounds on the collision probability. CLHASH makes optimal use of the random bits, in the sense that it offers XOR universality for short strings (less than 1 kB).

We believe that CLHASH might be suitable for many common purposes. The VHASH family has been proposed for cryptographic applications, and specifically message authentication (VMAC): similar applications are possible for CLHASH. Future work should investigate these applications.

Other microprocessor architectures also support fast carry-less multiplication, sometimes referring to it as *polynomial multiplication* (e.g., ARM [2] and Power [20]). Future work might review the performance of CLHASH on these architectures. It might also consider the acceleration of alternative hash families such as those based on Toeplitz matrices [37].

Acknowledgments This work was supported by the National Research Council of Canada, under Grant 26143.

References

1. Appleby, A.: SMHasher & MurmurHash (2012). http://code.google.com/p/smhasher. Last checked March 2015
2. ARM Limited: ARMv8 architecture reference manual (2014). http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/. Last checked March 2015
3. Aumasson, J.P., Bernstein, D.J.: SipHash: a fast short-input PRF. In: Galbraith, S., Nandi, M. (eds.) Progress in Cryptology (INDOCRYPT 2012). Lecture Notes in Computer Science, vol. 7668, pp. 489–508. Springer, Berlin (2012). doi:10.1007/978-3-642-34931-7_28
4. Aumasson, J.P., Bernstein, D.J.: SipHash: high-speed pseudorandom function (reference code) (2014). https://github.com/veorq/SipHash. Last checked Nov 2014
5. Barrett, P.: Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In: Odlyzko, A.M. (ed.) Advances in Cryptology (CRYPTO' 86). Lecture Notes in Computer Science, vol. 263, pp. 311–323. Springer, Berlin (1987). doi:10.1007/3-540-47721-7_24
6. Bernstein, D.J.: The Poly1305-AES message-authentication code. In: Fast Software Encryption. Lecture Notes in Computer Science, vol. 3557, pp. 32–49. Springer, Berlin (2005). doi:10.1007/11502760_3
7. Black, J., Halevi, S., Krawczyk, H., Krovetz, T., Rogaway, P.: UMAC: fast and secure message authentication. In: Wiener, M. (ed.) Advances in Cryptology (CRYPTO' 99). Lecture Notes in Computer Science, vol. 1666, pp. 216–233. Springer, Berlin (1999). doi:10.1007/3-540-48405-1_14
8. Bluhm, M., Gueron, S.: Fast software implementation of binary elliptic curve cryptography. Tech. rep, Cryptology ePrint Archive (2013)
9. Bos, J.W., Özen, O., Stam, M.: Efficient hashing using the AES instruction set. In: Proceedings of the 13th International Conference on Cryptographic Hardware and Embedded Systems (CHES'11), pp. 507–522. Springer, Berlin (2011)
10. Carter, J.L., Wegman, M.N.: Universal classes of hash functions. J. Comput. System Sci. **18**(2), 143–154 (1979). doi:10.1016/0022-0000(79)90044-8
11. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn, 3rd edn. The MIT Press, Cambridge (2009)
12. Dai, W., Krovetz, T.: VHASH security. Tech. Rep. 338, IACR Cryptology ePrint Archive (2007)
13. Estébanez, C., Hernandez-Castro, J.C., Ribagorda, A., Isasi, P.: Evolving hash functions by means of genetic programming. In: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, pp. 1861–1862. ACM, New York (2006)
14. Etzel, M., Patel, S., Ramzan, Z.: Square hash: fast message authentication via optimized universal hash functions. In: Wiener, M. (ed.) Advances in Cryptology (CRYPTO' 99). Lecture Notes in Computer Science, vol. 1666, pp. 234–251. Springer, Berlin (1999). doi:10.1007/3-540-48405-1_15
15. Fan, B., Andersen, D.G., Kaminsky, M., Mitzenmacher, M.D.: Cuckoo filter: practically better than Bloom. In: Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT '14), pp. 75–88. ACM, New York (2014). doi:10.1145/2674005.2674994
16. Fog, A.: Instruction tables: lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. Tech. rep., Copenhagen University College of Engineering (2014). http://www.agner.org/optimize/instruction_tables.pdf. Last checked March 2015
17. Gueron, S., Kounavis, M.: Efficient implementation of the Galois Counter Mode using a carry-less multiplier and a fast reduction algorithm. Inf. Process. Lett. **110**(14), 549–553 (2010). doi:10.1016/j.ipl.2010.04.011
18. Halevi, S., Krawczyk, H.: MMH: software message authentication in the Gbit/second rates. In: Biham, E. (ed.) Fast Software Encryption. Lecture Notes in Computer Science, vol. 1267, pp. 172–189. Springer, Berlin (1997). doi:10.1007/BFb0052345
19. Intel Corporation: Intel IACA tool: a static code analyser (2012). https://software.intel.com/en-us/articles/intel-architecture-code-analyzer. Last checked March 2015
20. Intel Corporation: Power ISA Version 2.07 (2013). https://www.power.org/wp-content/uploads/2013/05/PowerISA_V2.07_PUBLIC.pdf. Last checked March 2015
21. Intel Corporation: Power ISA Version 2.07 (2014). https://software.intel.com/sites/landingpage/IntrinsicsGuide/. Last checked March 2015
22. Knežević, M., Sakiyama, K., Fan, J., Verbauwhede, I.: Modular reduction in $GF(2^n)$ without pre-computational phase. In: von zur Gathen, J., Imaña, J.L., Koç, C.K. (eds.) Arithmetic of Finite Fields. Lecture Notes in Computer Science, vol. 5130, pp. 77–87. Springer, Berlin (2008). doi:10.1007/978-3-540-69499-1_7
23. Knuth, D.E.: Searching and Sorting. The Art of Computer Programming, vol. 3. Addison-Wesley, Reading (1997)
24. Krovetz, T.: Message authentication on 64-bit architectures. In: Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 4356, pp. 327–341. Springer, Berlin (2007). doi:10.1007/978-3-540-74462-7_23
25. Krovetz, T., Dai, W.: VMAC and VHASH implementation (2007). http://fastcrypto.org/vmac/. Last checked March 2015
26. Lemire, D., Kaser, O.: Strongly universal string hashing is fast. Comput. J. **57**(11), 1624–1638 (2014). doi:10.1093/comjnl/bxt070
27. Lim, H., Han, D., Andersen, D.G., Kaminsky, M.: Mica: a holistic approach to fast in-memory key-value storage. In: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14), pp. 429–444. USENIX Association, Berkeley (2014)
28. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. **8**(1), 3–30 (1998). doi:10.1145/272991.272995
29. Motzkin, T.S.: Evaluation of polynomials and evaluation of rational functions. Bull. Am. Math. Soc. **61**(9), 163 (1955)

30. Mullen, G.L., Panario, D.: Handbook of Finite Fields, 1st edn. Chapman & Hall/CRC, London (2013)
31. Nguyen, L.H., Roscoe, A.W.: New combinatorial bounds for universal hash functions. Tech. Rep. 153, Cryptology ePrint Archive (2009)
32. Oliveira, T., Aranha, D.F., López, J., Rodríguez-Henríquez, F.: Fast point multiplication algorithms for binary elliptic curves with and without precomputation. In: Joux, A., Youssef, A. (eds.) Selected Areas in Cryptography (SAC 2014). Lecture Notes in Computer Science, pp. 324–344. Springer International Publishing, Switzerland (2014). doi:10.1007/978-3-319-13051-4_20
33. Oliveira, T., López, J., Aranha, D.F., Rodríguez-Henríquez, F.: Two is the fastest prime: lambda coordinates for binary elliptic curves. J. Cryptogr. Eng. **4**(1), 3–17 (2014). doi:10.1007/s13389-013-0069-z
34. Paoloni, G.: How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. Intel Corporation, Santa Clara (2010)
35. Pike, G., Alakuijala, J.: The CityHash family of hash functions (2011). https://code.google.com/p/cityhash/. Last checked March 2015
36. Stinson, D.R.: Universal hashing and authentication codes. Des. Codes Cryptogr. **4**(4), 369–380 (1994). doi:10.1007/BF01388651
37. Stinson, D.R.: On the connections between universal hashing, combinatorial designs and error-correcting codes. Congr. Numer. **114**, 7–28 (1996)
38. Su, C., Fan, H.: Impact of Intel's new instruction sets on software implementation of $GF(2)[x]$ multiplication. Inf. Process. Lett. **112**(12), 497–502 (2012). doi:10.1016/j.ipl.2012.03.012
39. Taverne, J., Faz-Hernández, A., Aranha, D.F., Rodríguez-Henríquez, F., Hankerson, D., López, J.: Speeding scalar multiplication over binary elliptic curves using the new carry-less multiplication instruction. J. Cryptogr. Eng. **1**(3), 187–199 (2011). doi:10.1007/s13389-011-0017-8