TABULATION BASED 5-INDEPENDENT HASHING WITH APPLICATIONS TO LINEAR PROBING AND SECOND MOMENT ESTIMATION[*]

MIKKEL THORUP[†] AND YIN ZHANG[‡]

Abstract. In the framework of Carter and Wegman, a k-independent hash function maps any k keys independently. It is known that 5-independent hashing provides good expected performance in applications such as linear probing and second moment estimation for data streams. The classic 5-independent hash function evaluates a degree 4 polynomial over a prime field containing the key domain $[n] = \{0, \ldots, n-1\}$. Here we present an efficient 5-independent hash function that uses no multiplications. Instead, for any parameter c, we make $2c-1$ lookups in tables of size $O(n^{1/c})$. In experiments on different computers, our scheme gained factors 1.8 to 10 in speed over the polynomial method. We also conducted experiments on the performance of hash functions inside the above applications. In particular, we give realistic examples of inputs that make the most popular 2-independent hash function perform quite poorly. This illustrates the advantage of using schemes with provably good expected performance for all inputs.

1. Introduction.

We consider "k-independent hashing" in the classic framework of Carter and Wegman [33]. For any $i \geq 1$, let $[i] = \{0, 1, \ldots, i-1\}$. We consider "hash" functions from "keys" in $[n]$ to "hash values" in $[m]$. A class $\mathcal{H}$ of hash functions is *k-independent* if for any distinct $x_0, \ldots, x_{k-1} \in [n]$ and any possibly identical $y_0, \ldots, y_{k-1} \in [m]$,

$$\mathrm{Pr}_{h \in \mathcal{H}} \{h(x_i) = y_i \text{ for all } i \in [k]\} = 1/m^k, \tag{1.1}$$

where "$\mathrm{Pr}_{h \in \mathcal{H}}$" means probability with respect to the uniform distribution. By a *k-independent hash function*, we mean a hash function that has been drawn from a k-independent class. A different way of formulating (1.1) is that the hash values should be uniform in $[m]$ and that any k keys are mapped independently.

We will often contrast k-independent hashing with the *universal hashing* from [6] which just requires low collision probability, that is, for any different $x, y \in [n]$, $\mathrm{Pr}_{h \in \mathcal{H}}\{h(x) = h(y)\} \leq 1/m$. Trivially, 2-independent hashing is universal, but universal hashing may have no randomness, *e.g.*, if $[n] = [m]$, the identity is universal.

As the concept of independence is fundamental to probabilistic analysis, k-independent hash functions are both natural and powerful in algorithm analysis. They allow us to replace the heuristic assumption of truly random hash functions with real (implementable) hash functions that are still "independent enough" to yield provable performance guarantees. This leads to an appealing framework for the design of efficient randomized algorithms:

1. Show that k-independence suffices for a given algorithm.
2. Use the best off-the-shelf method for k-independent hashing.

It is known that 5-independent hashing suffices for some classic applications discussed below. Motivated by this, we will present a principally different 5-independent class of hash functions which in experiments could be evaluated 1.8 to 10 times faster than functions in previously known 5-independent classes. We want to emphasize that our contribution is "theory for computing" rather than "theory of computing", the goal being to develop theoretical understanding leading to concrete improvements for real computing. Discussions of practical relevance is therefore an integral part of the paper, motivating the theoretical developments.

1.1. Applications.

We will now discuss classic applications of 4- and 5-independent hashing. We will also discuss the practical need for speed as well as the dangers of using faster schemes without the theoretical guarantees.

1.1.1. 4-independent hashing.

Hashing is typically applied to a set of $s \leq n$ keys from $[n]$ and often we consider collisions harmful. For any $k \geq 2$, with k-independent hashing from $[n]$ to $[m]$, or just universal hashing, the expected number of collisions is bounded by $\binom{s}{2} \cdot \frac{1}{m} < s^2/(2m)$. However, with 2-independent hashing, the variance in the number of collisions may be as large as $\Theta(s^4/m)$. The worst example is a 2-independent hash function that with probability $(m-1)/m$ picks a random 1-1 mapping, and with probability $1/m$ maps all keys to the same random hash value. On the other hand, as shown in [5, 10], with 4-independent hashing, the variance is no bigger than the expected number of collisions. As described in [10] this means that 4-independent hashing is more reliable in many algorithmic contexts.

[*]Preliminary parts of this work were presented at SODA'04 [30] and at ALENEX'10 [31].

[†]AT&T Labs—Research, 180 Park Avenue, Florham Park, New Jersey 07932, USA, `mthorup@research.att.com`

[‡]The University of Texas at Austin, 1616 Guadalupe, Suite 2.408, Austin, Texas 78701, USA, `yzhang@cs.utexas.edu`

Our special interest in fast 4-independent hashing is due to its applications in the processing of data streams, an area that has gathered intense interest both in the theory community and in many applied communities such as those of databases and the Internet (see [2, 20] and the references therein). A common data stream scenario is as follows. A large stream of items arrive one by one at a rapid pace. When an item arrives, we have very little time to update some small amount of local information in cache. This local information, often called a "sketch", is selected with a certain type of questions in mind. The item itself is lost before the next item arrives. When all the items have passed by, we have to answer questions about the data stream.

A classic example is the second moment computation of Alon *et al.* [1]. Each item has a weight and a key and we want to compute the second moment which is the sum over each key of the square of the total weight of items with that key. The generic method used in [1] is that when an item arrives, a hash function is computed of the key, and then the hash value is used to update the sketch. At any point in time, the sketch provides an unbiased estimator of the second moment of the data stream seen so far. In order to control the variance of the estimator, the hash function used in [1] has to be 4-independent.

1.1.2. 5-independent hashing. Stepping up from 4-independent to 5-independent hashing has proved relevant for some of the most popular randomized algorithms, leading to the same asymptotic time bounds as we get using unrealistic truly random functions. Hash tables are the most commonly used non-trivial data structures, and the most popular implementation on standard hardware uses linear probing which is both fast and simple. Pagh *et al.* [22] have shown that linear probing takes expected constant time per operation on any input if 5-independent hashing is used. They also show that some concrete 2-independent hash functions take expected logarithmic time for certain inputs. In the same vein, Karloff and Raghavan [15] considered a certain non-recursive variant of quicksort [14], proving it to take $O(n \log n)$ expected time when 5-independent hashing is used. They also demonstrated an expected quadratic time for certain 2-independent hash functions and inputs.

1.1.3. Time is critical. Motivating the need for speed, we note that time is critical for many streaming algorithms. For example, we may be analyzing packet headers coming through a high-end Internet router. Such a router uses fast forwarding technology, and is therefore hard to keep up with. If we cannot keep up, the information is lost. Asking the router to slow down for the sake of measurements is not an option. One might try investing in faster or specialized hardware for the measurements, but then we could also invest in faster routers.

Computing the hash function is a bottleneck in the streaming algorithm for second moment estimation. While 2-independent hashing may give less guarantees for the output, it had the advantage of being an order of magnitude faster than existing methods for 4-independent hashing. However, here we will improve the speed of 4-independent hashing, making it a more viable option in time critical scenarios.

To be very concrete, the application [18] that originally motivated us for this research in 2002 was to monitor packets passing through an OC48 Internet backbone link (2.48 gigabits per second). At this link speed, it is common for packets to arrive at a rate of more than a million per second, thus leaving us with less than a microsecond per packet. In the worst case, when all packets are of the minimum Internet packet size of 40 bytes (320 bits), the packet arrival rate can be as high as $2.48 \times 10^9/320 = 7.75 \times 10^6$ per second, leaving us with less than 130 nanoseconds per packet. A critical part of the application was to compute the second moment with the packet key being its single word 32-bit IP address, and the packet weight being its size in bytes. The speed-up achieved in this paper (down from 182 to 30 nanoseconds for the 4-independent hashing itself plus 20 nanoseconds to update the sketch) made the application possible. Since then, computers and routers have both become faster (the experiments reported in this paper are run on newer computers), but it remains true that time is critical if you want to keep up with a high-end router.

Linear probing, for which we use 5-independent hashing, is even more essential to the analysis of Internet traffic streams. The stream database Gigascope [8] uses linear probing for the initial classification of packets for different queries. Once again, we need fast hashing to keep up with the traffic.

1.1.4. On the practical danger of not having theoretical guarantees. In the above applications, theory has proved that certain degrees of independence suffice for provably good expected performance for any input, but should we worry about this in practice? Mitzenmacher and Vadhan [19] have shown that 2-independent hashing is as good as truly random as long as there is enough entropy in the data. We do have a very fast and simple implementation of 2-independent hashing from [9], and it would be tempting to just use this in practice.

For our two main applications, second moment estimation and linear probing, we will demonstrate simple structured inputs that could easily occur in practice, and where the fastest 2-independent hashing from [9] leads to bad performance. This illustrates the practical danger of not having proven good expected performance for all inputs.

1.2. k-independent hashing. We will describe in more detail the relevant known methods for k-independent hashing, and show how our new 5-independent hashing scheme fits in.

1.2.1. Limiting the key sizes. In this paper, the theoretical developments are geared towards efficient implementations. In the implementations we will mostly focus on the hashing of 32-bit and 64-bit keys. This means that standard arithmetic operations are straightforward to implement in C with 64-bit arithmetic. It also means that we are happy to consider an implementation based on a Mersenne prime larger than the largest key, *e.g.*, $p = 2^{89} - 1$. We are not concerned that we do not know if there are infinitely many Mersenne primes.

Note that if we want k-independent hashing of larger keys from a set S, then a standard trick is to first apply simple universal hashing h_u into an intermediate domain A of size $\omega(|S|^2)$, and then apply k-independent hashing h_k to the keys in A. The first hashing into A is expected to be collision free, and then the composite hash function $h_k \circ h_u$ is k-independent over S. In fact, for our prime applications of second moment estimation and linear probing, we do not need the first universal hashing h_u to be collision-free. Essentially it suffices to have a linear-sized intermediate domain A, $|A| = O(|S|)$. This was proved by Thorup [29] for linear probing, and we shall prove it in this paper for the case of second moment estimation. Today we would expect $|S| < 2^{64}$ and then 64-bit keys will do for the intermediate domain A. By the time larger data sets are considered, we will probably have processors doing 128-bit arithmetic as a standard.

1.2.2. Multiplication-based hashing. The classic implementation of k-independent hashing from $[n]$ to $[m]$ by Carter and Wegman [33] is to use a degree $k-1$ polynomial over some prime field:

$$h(x) = \left(\left(\sum_{i=0}^{k-1} a_i x^i \right) \bmod p \right) \bmod m \tag{1.2}$$

for some prime $p \geq n$ with each a_i picked randomly from $[p]$. If p is an arbitrary prime, this method is fairly slow because '$\mathrm{mod}\ p$' is slow. However, as pointed out by Carter and Wegman [6], we can get a fast implementation using shifts and bit-wise Boolean operations if p is a so-called Mersenne prime of the form $2^i - 1$. We shall refer to this as *CW-trick*. In the hashing of 32-bit integers, we can use $p = 2^{61} - 1$, and for 64-bit integers, we can use $p = 2^{89} - 1$. Refer to Appendix A.12–A.14 for the realization.

For the practical implementation of the above methods, recall that standard 64-bit multiplication on computers discards overflow beyond the 64 bits. For example, this implies that we may need four 64-bit multiplications just to implement a full multiplication of two numbers from $[2^{61} - 1]$.

Multiplication-shift based hashing with low independence. For the special case of 2-independent hashing, we have a much faster and more practical method from [9]. If we are hashing from ℓ_{in} to ℓ_{out} bits, for some $\ell \geq \ell_{in} + \ell_{out} - 1$, we pick two random numbers $a, b \in [2^\ell]$, and use the hash function $h_{a,b}$ defined by

$$h_{a,b}(x) = \left((ax + b) \bmod 2^\ell\right) \operatorname{div} 2^{\ell - \ell_{out}},$$

where div keeps only the integer portion of the result. This has a fast implementation on standard processors if ℓ is the machine word length:

$$h_{a,b}(x) = (ax + b) \gg (\ell - \ell_{out}),$$

where $\gg$ denotes a right shift and the multiplication assumes that the multiplication discards overflow beyond the ℓ bits, as is standard on most computers if, say, ℓ is 32 or 64.

In fact, if we are satisfied with plain universal hashing, then as shown in [11], it suffices that $\ell \geq \ell_{in}$ and to pick a single odd random number $a \in [2^\ell]$. We then use the hash function h_a defined by

$$h_a(x) = (ax) \gg (\ell - \ell_{out}).$$

As a typical example, if $\ell_{out} \leq \ell_{in} = 32$, then for 2-independent hashing, we would use a 64-bit integer a and 64-bit multiplication. But for plain universal hashing, it suffices to work with 32-bit integers, which is faster.

The above two schemes can be viewed as instances of multiplicative hashing [17] where the golden ratio of 2^ℓ is recommended as a concrete value of a (with such a fixed value, the schemes lose universality). We refer to them as "multiplication-shift" based methods.

We note that [9] presents generalizations of the above schemes to k-independent hashing for $k > 2$, but the generalizations rely on multiplication of very long integers, *e.g.*, 218 bits for 4-universal hashing, and these generalizations for $k > 2$, are not practical compared with CW-trick.

1.3. Tabulation based hashing. An alternative fast way of generating a 2-independent ℓ_{out}-bit string from a key is through *simple tabulation*: in this setup, a key $\vec{x} \in [n]$ is a sequence $(x_0, \dots, x_{c-1})$ of "characters" from a domain $[r]$ where $r = n^{1/c}$. In applications, r could be 2^8 corresponding to 8-bit characters. For each character position i, there is a hash function $g_i : [r] \to [2^{\ell_{out}}]$. The hash value is

$$h(x_0, x_1, \dots, x_{c-1}) = g_0[x_0] \oplus g_1[x_1] \oplus \cdots \oplus g_{c-1}[x_{c-1}].$$

Here $\oplus$ denotes bit-wise exclusive-or. Carter and Wegman [6] showed that if the g_i are independently chosen from a 2-independent class $\mathcal{H}$, then the combined hash function h will be 2-independent. We could think of this as a recursive formula for 2-independent hashing, and this is useful when the input key is a string that is much longer than the desired hash value. However, here we assume that the g_i are tabulated with precomputed values in arrays. Above we used '[]' around the arguments of the g_i to allude to this array representation. Each of these arrays has only $n^{1/c}$ entries. With standard 8-bit characters ($n^{1/c} = 2^8$) the arrays can fit in cache, and then the lookups are very fast. Typically, we will assume that the tables g_i are filled with independent random values (*e.g.*, from `random.org`).

Theoretically, the simple tabulation scheme is quite incomparable with the multiplication-shift based methods from §1.2.2. The multiplication-shift based methods store only a constant number of values. On the other hand, they use multiplication which is not in AC^0. In contrast, simple tabulation uses only AC^0 operations but c lookups in tables of size $O(n^{1/c})$. Which approach is faster depends entirely on the computer. Experiments in [28] compared different hashing methods on different computers, and found simple tabulation over 8-bit characters to be very competitive with the 2-independent multiplication-shift based hashing from [11].

For independence higher than 2, we have the general polynomial method from §1.2.2 which is much slower than the 2-independent multiplication-shift scheme. It is therefore natural to ask if tabulation can help with higher independence. It is quite easy to see that simple tabulation is 3-independent if each table is 3-independent (*cf.* Fact 2.5). Unfortunately the scheme breaks down for independence above 3. To see this, we XOR the hash values of four keys (a_0, a_1), (a_0, b_1), (b_0, a_1), and (b_0, b_1):

$$\begin{aligned} & h(a_0, a_1) \oplus h(a_0, b_1) \oplus h(b_0, a_1) \oplus h(b_0, b_1) \\ & = g_0[a_0] \oplus g_1[a_1] \oplus g_0[a_0] \oplus g_1[b_1] \oplus g_0[b_0] \oplus g_1[a_1] \oplus g_0[b_0] \oplus g_1[b_1] \\ & = 0. \end{aligned}$$

The sum is zero because each term appears exactly twice. We conclude that simple tabulation is not 4-independent.

1.3.1. Tabulation for higher independence. Siegel [27] has shown how to use tabulation to reach much higher independence. The general framework is that we have d functions $f_0, \dots, f_{d-1}$ from the key domain $[n]$ to "derived characters" from a domain $[r']$. Given independent fully-random hash functions $g_0, \dots, g_{d-1}$ from $[r']$ to ℓ_{out}-bit strings, the hash value is

$$h(x) = \bigoplus_{i=0}^{d-1} g_i[f_i(x)].$$

Let $c' = (\log n)/\log_{r'}$, that is, $n^{1/c'} = r'$. Simple tabulation is the special case where the derived characters are the input characters and $c' = c$. However, using $d = O(c')$ derived characters, Siegel shows that if the f_i are fixed randomly, the resulting function is $n^{\Omega(1/c')}$-independent with high probability. The question now is whether we can find concrete f_i that can be represented and computed efficiently. Siegel shows probabilistically that there exists f_i that can be represented in space $O(n^{1/c'})$ so that each f_i can be computed with $c'^{\Theta(c')}$ lookups. As stated by Siegel, the construction is not practical.

Dietzfelbinger and Woelfel [12, §5] use Siegel's framework from above, but to get something more practical, they let the functions $f_0, \dots, f_{d-1}$ be independent 2-independent functions. They show for any distinct input keys $x_0, \dots, x_{d-1}$ and output values $y_0, \dots, y_{d-1}$

$$\Pr\{h(x_i) = y_i \text{ for all } i \in [k]\} = 1/m^k + (k/(d+1))(k/n^{1/c'})^d.$$

Above, the additive term $(k/(d+1))(k/n^{1/c'})^d$ is an error term relative to k-independence as defined in (1.1). For typical hash table applications we have $m \leq n$, so if we do not want the error term to dominate completely, we need $d > kc$ derived characters.

Dietzfelbinger and Woelfel [12, §5.2] mention that they can implement their k-independent hashing with a single multiplication, but this would be over numbers that are c' times as long as the input keys.

1.3.2. A new tabulation method for limited independence. In this paper we show that for limited independence there are much faster deterministic methods for computing the f_i, leading to exact k-independence with no error term in the probabilities.

As a striking example, consider the case where keys are divided into two characters from $[r]$, $r = n^{1/2}$. We will show that

$$h(x, y) = g_0[x] \oplus g_1[y] \oplus g_2[x + y]$$

is 5-independent if g_0, g_1, and g_2 are independent 5-independent hash functions into strings of the same length. In Siegel's framework, we have derived characters $f_0(x, y) = x$, $f_1(x, y) = y$, and $f_2(x, y) = x+y$. As the joint derived character domain, we could use $[r'] = [2r]$, but when storing g_0 and g_1 we should exploit that they only have entries from $[r]$. To improve the space for g_2, we will show that we can do the addition $x+y$ in any odd prime field containing the domain for input characters. For $r = 2^8, 2^{16}$, we will exploit that $p = r + 1$ is prime and do the addition in $\mathbb{Z}_p$. Then $x + y \in [r + 1]$.

To get 5-independent hashing for $c > 2$ characters, we could apply the above scheme recursively, but then we would end up using $c^{\log_2 3}$ derived characters. We will show that we can get down to $2c - 1$ derived characters from $[cn^{1/c}]$, or even from $[n^{1/c}+c]$ if $r = n^{1/c} \in \{2^8, 2^{16}\}$. The scheme is much more intricate than the above 2-character scheme, but it is efficient to implement, computing a 5-independent hash value with $2c-1$ lookups and $O(c)$ additions, shifts, and bit-wise Boolean operations.

We will also present a scheme for general k that gives k-independent hashing using $(k - 1)(c - 1) + 1$ k-independent hash functions. For $k = 5$, this is not as good as the 5-independent scheme just described, but it does have the advantage that the derived characters are from exactly the same domain as the input characters.

1.4. Comparison with the previous methods. Let us compare our new k-independent scheme with the above mentioned method of Dietzfelbinger and Woelfel [12] which needs $d \geq kc'$ randomly derived characters for any reasonable error bound. We use slightly fewer derived characters for the same space. More importantly, we derive them deterministically, which means that we avoid the error term from [12]. Moreover, we avoid the multiplications that [12] uses to compute their derived characters. Our construction only uses $O(kc)$ AC^0 operations: additions, shifts, and bit-wise Boolean operations plus memory look-ups from tables of $O(kcn^{1/c})$ total size.

Theoretically, our k-independent tabulation scheme is quite incomparable with the use of a degree $k - 1$ polynomial from §1.2.2. The polynomial methods store only k coefficients. On the other hand, they use $k - 1$ multiplications which are not in AC^0. In contrast, our tabulation scheme uses only AC^0 operations including $(k - 1)(c - 1) + 1$ lookups in tables of size $O(cn^{1/c})$.

In the extreme case where multiplication is not available, we could, of course, have a multiplication table of size $O(n^{1/c})$ for the multiplication of numbers in $[n^{1/(2c)}]$, and use it to simulate multiplication of numbers from $[n]$ using $4c^2$ lookups. In theory we might even get down to $\Theta(c \log c)$ lookups for such simulated multiplication [26]. For a degree k polynomial, we would end up with $\Theta(kc \log c)$ lookups which is not competitive with our less than kc lookups.

Here we are interested in the implementation on real computers offering both multiplication and memory. The relative speeds of multiplication versus memory depends entirely on the computer. Our particular interest is 4- and 5-independent hashing. We ran experiments on several computers, and found our new 5-independent tabulation to be significantly faster than both 4- and 5-independent hashing based on polynomials.

1.5. Contents. The remainder of this paper is organized as follows. In § 2, we present a general framework for tabulation based hashing along with some simple lemmas. In § 3, we present our schemes for 5-independent hashing. First we present the simple 2-character case, and then we move to the general case with c input characters and $2c - 1$ derived characters. In § 4, we present a scheme for general k that gives k-independent hashing on c input characters using $(k - 1)(c - 1) + 1$ table lookups. In § 5, we compare the speeds of different hashing schemes. In § 6, we first present a new second moment estimator and then demonstrate the effect of different hashing schemes on its accuracy.

In § 7, we demonstrate the effect of different hashing schemes on the performance of linear probing. We conclude in § 8. The full documentation of the code is in Appendix A. The source code of the C implementation is available online [32].

2. General framework. Our framework for tabulation based hashing with derived characters is essentially taken from Siegel [27]:

1. A *derived tabulation hash function* with d derived characters is defined in terms of d functions $f_0, \ldots, f_{d-1}$ mapping input keys $\vec{x}$ to *derived characters* $z_j = f_j(\vec{x})$. The domains $[r_j]$ of different derived characters z_j need not be the same. We refer to the vector $\vec{z} = (z_0, \ldots, z_{d-1})$ as the *derived key*.
2. For an ℓ_{out}-bit hash value, we assume we have d independent fully-random hash functions $g_0, \ldots, g_{d-1}$ where g_j maps the derived character z_j uniformly into $[2^{\ell_{out}}]$. The derived tabulation hash function h from keys $\vec{x}$ into $[2^{\ell_{out}}]$ is now defined by

$$h(\vec{x}) = g_0[z_0] \oplus \cdots \oplus g_{d-1}[z_{d-1}]. \tag{2.1}$$

Often we will view the input key as a vector $\vec{x} = (x_0, \ldots, x_{c-1})$ of c *input characters*. The input characters may be used directly as some of the derived characters. In that case, the other derived characters are referred to as *new*. For example, with simple tabulation, we have no new characters, and in the new 5-independent 2-character scheme we mentioned in §1.3.2, the only new derived character is $x_0 + x_1$.

As noted by Siegel in his proof of [27, Corollary 2.11], we have

OBSERVATION 2.1. *If the derived tabulation hash function h from (2.1) is k-independent when the g_j are fully random, then h is also k-independent if each g_j is only k-independent.*

The point in the observation is that for k-independence we only consider k keys at the same time, hence at most k characters from each table g_j. With Observation 2.1 in mind, as stated in our general framework, we always assume that the $g_0, ..., g_{d-1}$ are independent fully random hash functions.

We will now define the notion of a "derived key matrix" along with some basic algebraic properties. Consider $k' \leq k$ distinct keys $\vec{x}_i = (x_{i,0}, \ldots, x_{i,c-1})$, $i \in [k']$, with derived keys $\vec{z}_i = (z_{i,0}, \ldots, z_{i,d-1})$. We then define the *derived key matrix* as

$$D = \begin{bmatrix} z_{0,0} & z_{0,1} & \cdots & z_{0,d-1} \\ z_{1,0} & z_{1,1} & \cdots & z_{1,d-1} \\ \vdots & & & \vdots \\ z_{k'-1,0} & z_{k'-1,1} & \cdots & z_{k'-1,d-1} \end{bmatrix}$$

With derived key matrices, it is always understood that they are non-empty with $k' > 0$. We say that a derived key matrix has *an odd occurrence* if it has a column in which some character appears an odd number of times. From linear algebra, we will derive the following sufficient and necessary condition for k-independence:

PROPOSITION 2.2. *The following statements are equivalent:*

(i) Any matrix derived from $k' \leq k$ distinct keys has an odd occurrence.

(ii) The derived tabulation hash function h is k-independent

The proposition relies on our assumption that $g_0, ..., g_{d-1}$ are independent fully random hash functions.

The proof of Proposition 2.2 is deferred to §2.1. Proposition 2.2 provides a strengthening of Siegel's "peeling lemma" [27, Lemma 2.6]. We will say that a derived key matrix has *a unique occurrence* if it has a column in which some character appears exactly once. Trivially this implies an odd occurrence. Translated into our terminology, the peeling lemma says that we get k-independence if every matrix derived from $k' \leq k$ keys has a unique occurrence. This is stronger than our odd occurrence condition and it is not a necessary condition for k-independence. On the other hand, Proposition 2.2 is only true because we XOR the random ℓ_{out}-bit from the g_j in $h(\vec{x}) = \bigoplus_{j \in [s]} g_j[z_j]$, that is, hash values are taken from and added in $\mathbb{GF}(2^{\ell_{out}})$. Siegel is also working in $\mathbb{GF}(2^{\ell_{out}})$, but his peeling lemma holds true in the more general case where the hash values are taken from and added in any field. The useful fact given next illustrates the added advantage of Proposition 2.2.

FACT 2.3 (odd number of keys). *A matrix derived from an odd number of keys has an odd occurrence.*

Proof. We can consider any column. Since the number of rows is odd, there must be some character appearing an odd number of times in the column. □

Proposition 2.2 together with Fact 2.3 implies that the maximal independence is always odd.

COROLLARY 2.4. *Suppose the derived tabulation hash function h defined in (2.1) is k-independent. If k is even, then h is also $(k+1)$-independent.*

Finally, as our base case we have

FACT 2.5 (base). *The key matrix derived from ≤ 3 keys has an odd occurrence.*

Proof. By Fact 2.3 we only have to consider the key matrix derived from two keys. Since they are distinct, there must be a column in which each key has its own unique character. ☐

As a trivial case, we apply Fact 2.5 to simple tabulation hashing where the input keys are used directly as derived keys. In conjunction with Proposition 2.2, we get

COROLLARY 2.6. *Simple tabulation hashing is 3-independent.*

Finally, the fact given next introduces some flexibility that is useful in efficient implementations (*cf.* §3.4 below).

FACT 2.7. *Suppose we have functions $f_0, \ldots, f_{d-1}$ providing derived characters $z_j = f_j(\vec{x}) \in [r_j]$ such that the derived tabulation hash function h is k-independent. For $j \in [d]$, let τ_j be a function from a character domain $[r'_j]$ to $[r_j]$ and f'_j be a function from input keys to $[r'_j]$ such that $\tau_j(f'_j(\vec{x})) = f_j(\vec{x})$. Then with the derived characters $z'_j = f'_j(\vec{x})$, the derived tabulation hash function h' is k-independent.*

As a useful example, for some constants $\alpha_0, \ldots, \alpha_{d-1}$, we can let $f'_0, \ldots, f'_{d-1}$ be such that $f'_j(\vec{x}) \equiv f_j(\vec{x}) + \alpha_j \pmod{r_j}$ for any input key $\vec{x}$. Then $\tau_j(z'_j) \equiv (z'_j - \alpha_j) \bmod r_j = z_j$. So with the derived characters $z'_j = f'_j(\vec{x})$, the derived tabulation hash function h' is k-independent.

Proof. By Proposition 2.2 it suffices to show that if for some set X of keys, the derived key matrix D based on the f_j has a column j where some character a appears an odd number of times, then this is also the case for the matrix D' derived with the f'_j. We have an odd number of rows i such that $z_{i,j} = a$. These are exactly the rows such that $\tau_j(z'_{i,j}) = z_{i,j}$. There must therefore be a concrete value a' of $z'_{i,j}$ with $\tau_j(a') = a$ that appears an odd number of times in column j of D'. ☐

2.1. Proof of Proposition 2.2. We will now prove Proposition 2.2 which says that the following two statements are equivalent:

(i) Any matrix derived from $k' \leq k$ distinct keys has an odd occurrence.

(ii) The derived tabulation hash function h is k-independent.

First, assume that (i) is false, hence that there is a set of $k' \leq k$ distinct keys $\vec{x}_i$, $i \in [k']$, such that the derived key matrix

$$D = \begin{bmatrix} z_{0,0} & z_{0,1} & \cdots & z_{0,d-1} \\ z_{1,0} & z_{1,1} & \cdots & z_{1,d-1} \\ \vdots & & & \vdots \\ z_{k'-1,0} & z_{k'-1,1} & \cdots & z_{k'-1,d-1} \end{bmatrix}$$

satisfies that each character appears an even number of times in each column of D. We want to argue that $\bigoplus h(\vec{x}_i) = 0$. To see this, we note that

$$\bigoplus_{i \in [k']} h(\vec{x}_i) = \bigoplus_{i \in [k'], j \in [s]} g_j[z_{i,j}] = \bigoplus_{(j,a) \in [s] \times [r_j]} \left(\bigoplus_{i : z_{i,j} = a} g_j[a] \right).$$

Each $\bigoplus_{i:z_{i,j}=a} g_j[a]$ XOR the same value an even number of times, so $\bigoplus_{i:z_{i,j}=a} g_j[a] = 0$. Therefore $\bigoplus h(\vec{x}_i) = 0$, contradicting (ii). Thus (ii) implies (i).

The other, more interesting, direction follows by a simple translation to linear algebra. The translation itself is implicit in Siegel's work [27], but he did not take advantage of the special properties of $\mathbb{GF}(2^{\ell_{out}})$. First note that all output bits are independent, so it suffices to consider the case where $\ell_{out} = 1$, hence that the hash values are single bits from $\mathbb{GF}(2)$.

We now combine all the g_j into a single column vector G over $\mathbb{GF}(2)$ indexed by $\Sigma^* = \{(j,a) | j \in [d], a \in [r_j]\}$. We also translate each derived key $\vec{z} = (z_0, \ldots, z_{d-1})$ into a row vector z^* over $\mathbb{GF}(2)$ indexed by Σ^* where $z^*_{(j,a)} = 1$ if and only if $z_j = a$. Now we get the hash value as the dot-product $h(\vec{x}) = z^* \cdot G$.

Assume that (i) is true and consider k given keys $\vec{x}_i$, $i \in [k]$. We want to argue that they have independent and uniformly distributed hash values. Following the above pattern, the derived key matrix is translated into a matrix D^*

over $\mathbb{GF}(2)$ with rows indexed by $[k]$ and columns indexed by Σ^* and $D^*_{i,(j,a)} = 1$ if and only if $z_{i,j} = 1$. Finally, let $H = (h(\vec{x}_0), \ldots, h(\vec{x}_{k-1}))^\intercal$ be the row vector of the hash values of the $\vec{x}_i$. Then

$$H = D^* G.$$

Here D^* provides a fixed linear map defined from the given keys $\vec{x}_i$, $i \in [k]$. We want to argue that H is uniformly distributed over $\left(\mathbb{GF}(2)^k\right)^\intercal$ if G is uniformly distributed over $\left(\mathbb{GF}(2)^d\right)^\intercal$. This is the same as saying that all $H \in \left(\mathbb{GF}(2)^k\right)^\intercal$ have a pre-image of the same size.

Because the mapping is linear, we know that the pre-image of any element in the image has exactly the same size. It remains to show that the mapping is surjective. This follows if D^* has full row rank. Working over $\mathbb{GF}(2)$ we have that D^* has full row rank if and only if there is no subset of the rows such that each column has an even number of ones in these rows. Suppose for a contradiction that we had such a subset of rows. Then the original derived key matrix obtained from these keys would have each character appearing an even number of times in each column, and this contradicts (i). Thus (ii) follows from (i), completing the proof of Proposition 2.2.

3. Tabulation-based 5-independent hashing.

We will now show how to derive some characters so that the derived tabulation hash function h defined in (2.1) becomes 5-independent. From Proposition 2.2, Fact 2.3, and Fact 2.5, we immediately get

LEMMA 3.1. *The derived tabulation hash function is 5-independent if and only if the matrix derived from any 4 keys has an odd occurrence.*

Recall, as stated in our general framework, that the $g_0, \ldots, g_{d-1}$ are assumed to be independent fully random hash functions.

3.1. 5-independent hashing with 2 characters.

THEOREM 3.2. *In the case of two-character keys (x, y), if we use x, y, and $x + y$ as derived characters, then the derived tabulation hash function*

$$h(x, y) = g_0[x] \oplus g_1[y] \oplus g_2[x + y]$$

is 5-independent.

Proof. Consider 4 distinct keys (x_i, y_i), $i \in [4]$, with new derived character $z_i = x_i + y_i$. By Lemma 3.1 it suffices to show that the derived key matrix

$$D = \begin{bmatrix} x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix}$$

has an element that is unique in its column. Without loss of generality, we may assume that each element appears at least twice in the input columns $\{x_i\}$ and $\{y_i\}$. Since the four keys (x_i, y_i) are distinct and D has only four rows, it is easy to see that each x_i and y_i must appear exactly twice in its column. Without loss of generality, we can assume the four distinct keys are (a_0, b_0), (a_0, b_1), (a_1, b_0), and (a_1, b_1) where $a_0 < a_1$ and $b_0 < b_1$. This implies $a_0 + b_0 < a_0 + b_1 < a_1 + b_1$ and $a_0 + b_0 < a_1 + b_0 < a_1 + b_1$. So both $a_0 + b_0$ and $a_1 + b_1$ are unique in column $\{z_i\}$. $\square$

A slight caveat of the above scheme is that the derived character $x + y$ requires one more bit than x and y, hence that g_2 needs to be over a domain that is twice as large as that of the input characters. It would have been nicer if we could just apply g_2 to $x \oplus y$ instead of $x + y$, but as we shall see in §3.2, the derived tabulation hash function h' would not be 4-independent.

However, the following theorem shows that we can still achieve 4-independent hashing if implement $x + y$ with addition over an *odd* prime field $\mathbb{Z}_p$ containing the domain for input characters. With b-bit characters for $b = 8, 16$, we can exploit that $2^b + 1$ is prime. Then the domain of the derived character $x + y$ is only one bigger than that of the input characters.

THEOREM 3.3. *Theorem 3.2 also holds true if the derived character $a + b$ is computed using addition over an odd prime field $\mathbb{Z}_p$ including the domain of the input characters. That is, if keys are divided into two characters from $[r]$ and $p \geq r$ is an odd prime, then the derived tabulation hash function*

$$h(x, y) = g_0[x] \oplus g_1[y] \oplus g_2[x + y \mod p]$$

is 5*-independent.*

Proof. For this proof, we use '$+$' and '$-$' to denote addition and subtraction in the field $\mathbb{Z}_p$. As in the proof of Theorem 3.2, we may assume that the 4 distinct keys (x_i, y_i), $i \in [4]$, are (a_0, b_0), (a_0, b_1), (a_1, b_0), and (a_1, b_1), respectively, where $a_0 \neq a_1$ and $b_0 \neq b_1$. With $z_i = x_i + y_i$, we need to show that some z_i is unique. Clearly, $z_0 = a_0 + b_0 \neq a_0 + b_1 = z_1$ (because $b_0 \neq b_1$). Similarly, $z_0 = a_0 + b_0 \neq z_2 = a_1 + b_0$ (because $a_0 \neq a_1$). Hence, if z_0 is not unique, we must have $z_0 = z_3$. Similarly, if z_1 is not unique, we must have $z_1 = z_2$. But these equalities imply $z_0 + z_1 = z_2 + z_3$. That is, $a_0 + b_0 + a_0 + b_1 = a_1 + b_0 + a_1 + b_1$, or $a_0 + a_0 = a_1 + a_1$. Therefore, there exists an element $e = a_0 - a_1 \neq 0$ such that $e + e = 0$. This is impossible in an odd prime field, so we conclude that some z_i is unique. □

3.2. Limitations. We now note that Theorem 3.3 does not hold if $a + b$ is derived by addition in an even field, (*e.g.*, if $\oplus$ is used for '$+$'). This is because in any even field $\mathbb{F}$, there exists an element $e \neq 0$ such that $e + e = 0$, where 0 is the zero element of $\mathbb{F}$. More precisely, consider now the four keys $(0,0)$, $(0,e)$, $(e,0)$, and (e,e). The derived key matrix

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & e & e \\ e & 0 & e \\ e & e & 0 \end{bmatrix}$$

does not have any character appearing an odd number of times in any column. Therefore, by Proposition 2.2, the derived tabulation hashing is not 4-independent.

Next we note that our 5-independent schemes from Theorem 3.2 and 3.3 are also not 6-independent. Assume that $0, 1, 2, 3$ are in the character domains, consider the six keys $(0,1), (0,2), (1,0), (1,2), (2,0), (2,1)$. The derived key matrix

$$\begin{bmatrix} 0 & 1 & 1 \\ 0 & 2 & 2 \\ 1 & 0 & 1 \\ 1 & 2 & 3 \\ 2 & 0 & 2 \\ 2 & 1 & 3 \end{bmatrix}$$

does not have any character appearing an odd number of times in any column. Therefore, by Proposition 2.2, the derived tabulation hashing is not 6-independent.

3.3. 5-independent hashing with c characters. For 5-independent hashing with more than two input characters from $[r]$, we can recursively apply the two-character scheme. But then, for c characters, we would end up using $c^{\log_2 3}$ derived characters. In this section we show how we can get down to $d = 2c - 1$ derived characters and yet get 5-independent derived tabulation hashing.

We use a finite field $\mathbb{F}$ of *odd* cardinality containing the input characters, *e.g.*, $\mathbb{Z}_p$ for a prime number p with $p \geq r$. All linear algebra notions like "full rank" refer to this field, and calculations (vector additions and vector-matrix multiplications) are carried out in this field. Given a key of c input characters $\vec{x} = (x_0, x_1, \ldots, x_{c-1})$, $x_i \in [r]$, we derive $d = 2c - 1$ characters by including the c input characters themselves together with $c - 1$ new characters. The vector $\vec{y} = (y_0, y_1, \ldots, y_{c-2})$ of new characters is computed as $\vec{y} = \vec{x}\, G$, where G is a $c \times (c-1)$ generator matrix with the property that any square submatrix of G has full rank. As an example of G, we can use the $c \times (c-1)$ Cauchy matrix below over $\mathbb{Z}_p$ (assuming that ı $\geq d = 2c - 1$):

$$\mathcal{C}^{c\times(c-1)} = \left[\frac{1}{u_i - v_j}\right]_{i\in[c],\ j\in[c-1]} = \begin{bmatrix} \frac{1}{u_0-v_0} & \frac{1}{u_0-v_1} & \cdots & \frac{1}{u_0-v_{c-2}} \\ \frac{1}{u_1-v_0} & \frac{1}{u_1-v_1} & \cdots & \frac{1}{u_1-v_{c-2}} \\ \vdots & & & \vdots \\ \frac{1}{u_{c-1}-v_0} & \frac{1}{u_{c-1}-v_1} & \cdots & \frac{1}{u_{c-1}-v_{c-2}} \end{bmatrix},$$

where $u_0, u_1, \ldots, u_{c-1}, v_0, v_1, \ldots, v_{c-2}$ are $(2c-1)$ distinct, but otherwise arbitrary, elements of $\mathbb{Z}_p$ (so $u_i - v_j \neq 0$ for all $i \in [c]$ and $j \in [c-1]$).

THEOREM 3.4. *We consider c-character keys $\vec{x} = (x_j)_{j \in [c]}$, $x_j \in [r]$. Let G be a $c \times (c-1)$ generator matrix with the property that any square submatrix of G has full rank over a prime field $\mathbb{Z}_p$, where $p \geq \max\{r, 2c-1\}$ is an odd prime. Let $\vec{y} = (y_j)_{j \in [c-1]}$ be the vector of $c-1$ new characters computed by $\vec{y} = \vec{x}\,G$ over $\mathbb{Z}_p$. If the derived key $\vec{z} = \vec{x}\vec{y}$ concatenates the input and the new characters, then the derived tabulation hash function*

$$h(\vec{x}) = g_0[x_0] \oplus \cdots \oplus g_{c-1}[x_{c-1}] \oplus g_c[y_0] \oplus \cdots \oplus g_{2c-2}[y_{c-2}]$$

is 5-independent.

With the above scheme, we only need $2c-1$ table lookups to compute the hash value for c input characters. However, to make the scheme useful in practice, we still need to compute $\vec{y} = \vec{x}\,G$ very efficiently. The schoolbook implementation would require $O(cd) = O(c^2)$ multiplications and additions over $\mathbb{Z}_p$. However, in § 3.4, we show efficient techniques to compute a $\vec{y}\,' \approx \vec{x}\,G$ in $O(c)$ time in a way that preserves 5-independent hashing.

Proof. [of Theorem 3.4] Consider 4 distinct c-character keys $\vec{x}_i = (x_{i,0}, \ldots, x_{i,c-1})$, $i \in [4]$. Let $\vec{y}_i = (y_{i,0}, \ldots, y_{i,c-2}) = \vec{x}_i G$ be the vector of new characters and consider the derived key matrix

$$\begin{aligned} D &= \begin{bmatrix} x_{0,0} & \cdots & x_{0,c-1} & y_{0,0} & \cdots & y_{0,c-2} \\ x_{1,0} & \cdots & x_{1,c-1} & y_{1,0} & \cdots & y_{1,c-2} \\ x_{2,0} & \cdots & x_{2,c-1} & y_{2,0} & \cdots & y_{2,c-2} \\ x_{3,0} & \cdots & x_{3,c-1} & y_{3,0} & \cdots & y_{3,c-2} \end{bmatrix} \\ &= \begin{bmatrix} X & Y \end{bmatrix} \end{aligned}$$

where X and Y are the submatrices formed by vectors $\vec{x}_i$ and $\vec{y}_i$, respectively. Clearly, $Y = XG$. By Lemma 3.1 it suffices to show that there is a column in which some character appears exactly once.

Assume by way of contradiction that each element of D appears at least twice in its column. Then each column $D[\cdot, j]$ must be either of type 0: $(a\;a\;a\;a)^T$, in which all 4 elements of the column are equal, or one of the three proper types in which each element appears exactly twice: type α: $(a\;a\;b\;b)^T$, type β: $(a\;b\;a\;b)^T$, and type γ: $(a\;b\;b\;a)^T$.

For $t = 0, \alpha, \beta, \gamma$, let X_t be the possibly empty submatrix of X that consists of all columns of type t. Also, let G_t be the submatrix of G consisting of the rows j such that column j of X is of type t. Finally, we define $Y_t = X_t G_t$. Then $Y = \sum_{t=0,\alpha,\beta,\gamma} Y_t$.

Consider a specific new column $Y[\cdot, j]$, $j \in [c-1]$. For $t = 0, \alpha, \beta, \gamma$, $Y_t[\cdot, j]$ is of type 0 or type t as it is a linear combination of input columns of type t. We say type $t \neq 0$ is *present* in new column $Y[\cdot, j]$ if $Y_t[\cdot, j]$ is of type t.

CLAIM 3.5. *If no column of Y contains a unique character, then at most one type can be present in each new column $Y[\cdot, j]$, $j \in [r]$.*

Proof. Suppose for a contradiction that at least two types are present in column $Y[\cdot, j]$. By symmetry, we may assume that α and β are present. Then $Y_\alpha[\cdot, j] = (a_0\;a_0\;b_0\;b_0)^T$ and $Y_\beta[\cdot, j] = (a_1\;b_1\;a_1\;b_1)^T$, so from the proof of Theorem 3.3, we know that $(Y_\alpha[\cdot, j] + Y_\beta[\cdot, j])$ has a unique character. This is the place where it is used that the field has odd cardinality. If γ is not present, $Y_0[\cdot, j] + Y_\gamma[\cdot, j]$ is of type 0, and then we know that $Y[\cdot, j]$ has a unique character. Otherwise, all three types $\neq 0$ are present and symmetric in that regard. If we don't have a unique character in $Y[\cdot, j]$, it is of some type, say 0 or γ. This implies that $Y_\alpha[\cdot, j] + Y_\beta[\cdot, j] = Y[\cdot, j] - Y_0[\cdot, j] + Y_\gamma[\cdot, j]$ is of type γ contradicting that $Y_\alpha[\cdot, j] + Y_\beta[\cdot, j]$ has a unique character. □

CLAIM 3.6. *Let n_t be the number of columns in X_t. If $n_t > 0$ and $t \neq 0$, then G has at most $n_t - 1$ new columns $Y[\cdot, j]$ where type t is not present.*

Proof. Assume by contradiction that there are n_t or more new columns where n_t is not present. We can then find an $n_t \times n_t$ submatrix G_t^0 of G_t consisting of columns j such that type t is not present in new column $Y[\cdot, j]$. Then, for any two rows $\vec{a}$ and $\vec{b}$ of X_t, $\vec{a}G_t^0 = \vec{b}G_t^0$. However, in X_t there are at least two different rows $\vec{a}$ and $\vec{b}$. So G_t^0 cannot have full rank. This contradicts the fact that all square submatrices of G have full rank. □

Now we are ready to prove Theorem 3.4. Since the input keys are distinct, there have to be at least two proper types t with $n_t > 0$. Without loss of generality, assume $n_\alpha > 0$ and $n_\beta > 0$. For every new column $Y[\cdot, j]$, from Claim 3.5, either α or β is not present. From Claim 3.6, there are at most $(n_\alpha - 1)$ new columns in which α is not present, and at most $(n_\beta - 1)$ new columns in which β is not present. As a result, the total number of new columns is at most $(n_\alpha - 1) + (n_\beta - 1) \leq (\sum_{t=0,\alpha,\beta,\gamma} n_t) - 2 = c - 2$. However, G has $c-1$ columns, so this gives us the desired contradiction. □

3.4. Relaxed and efficient computation of $\vec{x}\,G$. We would now like an efficient computation of the new characters $\vec{y} = (y_0, \ldots, y_{c-2}) = \vec{x}\,G$ from Theorem 3.4. Then the derived key $\vec{x}\vec{y}$ gives 5-independent tabulation hashing. The product in Theorem 3.4 is over $\mathbb{Z}_p$, so $y_j \in [p]$. Hence, by the example in Fact 2.7, for some constants $\alpha_0, \ldots, \alpha_{c-2}$, it suffices to compute some $\vec{y}\,' = (y'_0, \ldots, y'_{c-2})$ such that $y'_j \equiv y_j + \alpha_j \pmod p$ for $j \in [c-1]$. Using $\vec{x}\vec{y}\,'$ as derived key would still lead to a 5-independent class. Below we shall use this freedom to add a constant and an arbitrary multiple of p in the y'_j.

Multiplication through tabulation. Let $\vec{G}_i$, $i \in [c]$, be the c rows of the generator matrix G from Theorem 3.4. Then

$$\vec{y} = \vec{x}\,G = (x_0\ \ldots\ x_{c-1}) \begin{bmatrix} \vec{G}_0 \\ \vdots \\ \vec{G}_{c-1} \end{bmatrix} = \sum_{i \in [c]} x_i\, \vec{G}_i$$

Therefore, we can avoid all the multiplications by storing with each x_i not only the hash value $g_i[x_i]$, but also the above vector $x_i\,\vec{G}_i$, which we denote $G_i[x_i]$. Then $\vec{y}$ is the sum $\sum_{i\in[c]} G_i[x_i]$ over $\mathbb{Z}_p$ of these tabulated vectors. We stipulate that the values $g_i[x_i]$ and $G_i[x_i]$ are stored as pairs in a single table indexed by x_i, hence that they are found together in a single lookup.

Using regular addition. Since we are free to add arbitrary multiples of p in the y'_j, we can compute $\vec{y}\,' = \sum_{i\in[c]} G_i[x_i]$ using regular integer addition rather than addition over $\mathbb{Z}_p$.

Parallel additions. To make the additions efficient, we can exploit bit-level parallelism by packing the $G_i[x_i]$ into bit-strings with $\lceil \log_2 c \rceil$ bits between adjacent elements. Then we can add the vectors by adding the bit strings as regular integers. By Bertrand's Postulate, we can assume $p < 2^{b+1}$, hence that each element of $G_i[x_i]$ uses $b+1$ bits. Consequently, we use $b' = b + 1 + \lceil \log_2 c \rceil$ bits per element.

For any application we are interested in, $1 + \lceil \log_2 c \rceil \leq b$, and then $b' \leq 2b$. This means that our vectors are coded in bit-strings that are at most twice as long as the input keys. We have assumed our input keys are contained in a word. Hence, we can perform each vector addition with two word additions. Consequently, we can perform all $c-1$ vector additions in $O(c)$ time.

Compression. With regular addition in $\sum_{i\in[c]} G_i[x_i]$, the derived characters may end up as large as $c(p-1)$, which means that tables for derived characters need this many entries. If memory becomes a problem, we could perform the $\bmod p$ operation on the derived characters after we have done all the additions, thus placing the derived characters in $[p]$. This can be done in $O(\log c)$ total time using bit-level parallelism like in the vector additions.

However, for b-bit characters with $b = 8, 16$, we can do even better exploiting that $p = 2^b + 1$ is a prime. We are then going to place the new derived characters in $[2^b + c + 1]$. Consider one of the current derived characters $z = y'_j$. We know that $z \leq c\,p$. Let $z^* = (z \wedge (2^b - 1)) + c - ((z \gg b))$, where $\gg$ denotes a right shift and $\wedge$ denotes bit-wise AND. It is easily seen that $0 \leq z^* \leq 2^b + c$ and $z^* \equiv z + c \pmod p$. It follows that $y'^{*}_j \equiv y_j + c \pmod p$, and that we can use y'^{*}_j as new derived character.

The transformation from z to z^* can be performed in parallel for a vector of derived characters. With appropriate pre-computed constants, the compression is performed efficiently in one line of C code.

4. k-independent derived tabulation hashing. Here we present a scheme for general k-independent derived tabulation hashing. With c input characters we use $d = (k-1)(c-1)+1$ derived characters. By Corollary 2.4, when k is even, we get $(k+1)$-independence from k-independence, so we can assume that k is even.

For 5-independent hashing with $k = 4$, this is not as good as the results above, but it does have the small advantage that if the input consists of b-bit characters, it allows derived characters that are also b-bit characters.

We will work with a finite field $\mathbb{F}$ containing the input characters. Contrasting our previous construction for 5-independent hashing, the cardinality of $\mathbb{F}$ may be either odd or even, where even means that the cardinality is some power of 2. All linear algebra notions like "full rank" refer to this field, and calculations (vector additions and matrix-vector multiplications) are carried out in this field. Note that in finite fields with even cardinality, matrix-vector multiplications can be carried out in an efficient way since the XOR-operation on words is provided by the hardware. Given c input characters $\vec{x} = (x_i)_{i\in[c]}$, we derive $d = (k-1)(c-1)+1$ characters $\vec{z} = (z_j)_{j\in[d]}$, using

$$\vec{z} = \vec{x}\,H$$

where H is a $c \times d$ generator matrix with the property that any $c \times c$ submatrix of H has full rank. Matrices with this property are commonly used in coding theory to generate *erasure resilient codes* [4, 25]. For example, we can choose $H = [I_c\ G]$ where I_c is the $c \times c$ identity matrix and G is a $c \times (d-c)$ matrix with full rank of all square submatrices. Then we get a scheme just like in § 3.3 where G can be the Cauchy matrix $\mathcal{C}^{c\times(d-c)}$. However, this time we may use an even field such as $\mathbb{GF}(2^b)$ transforming b-bit input characters into b-bit derived characters. Assuming that the field has cardinality above d, another possible choice of H is a $c \times d$ Vandermonde matrix:

$$\mathcal{V}^{c\times d} = \left[v_j^i\right]_{i\in[c],\ j\in[d]} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ v_0 & v_1 & v_2 & \dots & v_{d-1} \\ \vdots & & & & \vdots \\ v_0^{c-1} & v_1^{c-1} & v_2^{c-1} & \dots & v_{d-1}^{c-1} \end{bmatrix},$$

where $v_0, v_1, \dots, v_{d-1}$ are d distinct, but otherwise arbitrary, elements of $\mathbb{GF}(2^b)$.

THEOREM 4.1. *Let H be a $c \times d$ matrix with the property that any $c \times c$ submatrix of H has full rank over a field containing the input characters. For any c-character key $\vec{x} = (x_i)_{i\in[c]}$, let $\vec{z} = (z_j)_{j\in[d]}$, be the d-character derived key computed by $\vec{z} = \vec{x}\,H$. Then the derived tabulation hash function*

$$h(\vec{x}) = g_0[z_0] \oplus \cdots \oplus g_{d-1}[z_{d-1}]$$

is k-independent.

The theorem relies on our assumption that $g_0, ..., g_{d-1}$ are independent fully random hash functions.

Proof. By Proposition 2.2 it suffices to prove for any $k' \le k$ distinct keys $\vec{x}_i$, $i \in [k']$, some element of the derived key matrix D is unique in its column. Suppose for a contradiction that every element appears at least twice in its column. For each column $D[\cdot, j]$, define $E_j = \{(a, b) \mid a, b \in [k'] \text{ s.t. } a < b \wedge D[a, j] = D[b, j]\}$. Then we have $|E_j| \ge k'/2$. As the result, $\sum_{j\in[s]} |E_j| \ge s \cdot k'/2 = ((k-1)(c-1)+1)\cdot k'/2 > (c-1)\binom{k'}{2}$. But there are only $\binom{k'}{2}$ different (a, b) pairs ($a, b \in [k']$, $a < b$). Therefore, there must exist a pair (a_0, b_0) that appears in at least c different E_j, $j \in [s]$.

Let D' and H' be the $c \times c$ submatrices of D and H consisting of columns j with $(a_0, b_0) \in E_j$. In D', rows a_0 and b_0 are identical, so $\vec{x}_{a_0}\,H' = \vec{x}_{b_0}\,H'$. However, $\vec{x}_{a_0} \ne \vec{x}_{b_0}$, so this contradicts the fact that H' is a $c \times c$ matrix with full rank. □

4.1. Efficient computation of $\vec{z}$ over $\mathbb{GF}(2^b)$. As mentioned above, we can pick $H = [I_q\ G]$ and thus get a scheme like in § 3.3 with $\vec{z}$ being the concatenation of $\vec{x}$ and $\vec{y} = \vec{x}\,G$. With the notation of § 3.4, we compute $\vec{y}$ as the sum $\sum_{i\in[c]} G_i[x_i]$ of the tabulated vectors $G_i[x_i]$. However, this time, we may work in the even field $\mathbb{GF}(2^b)$. Consequently, the elements of $g_i(x_i)$ are in $[2^b]$ and addition over $\mathbb{GF}(2^b)$ is just XOR as supported by the hardware without any need for carry bits. Thus, each vector $G_i[x_i]$ is represented as a bit-string of length $rc = (k-2)(c-1)c$, and we can compute $\vec{y} = \bigoplus_{i\in[c]} \vec{g}_i(x_i)$. The resulting derived characters are all in $[2^b]$. This scheme is thus in many ways simpler than our specialized scheme for 5-independent hashing over $\mathbb{Z}_p$. However, for $k = 4$, our general scheme performs worse because it uses $3c - 2$ derived characters, whereas our specialized scheme only uses $2c - 1$ derived characters. So far, we do not understand if this is an inherent advantage of $\mathbb{Z}_p$ over $\mathbb{GF}(2^b)$, or if there is a smarter way of exploiting $\mathbb{GF}(2^b)$.

5. Experimental evaluation of hash functions. In this section, we report on experiments in which the speed of different hashing schemes based on implementations in C was compared. In later sections, we will further evaluate their performance in the context of some concrete applications.

The hashing schemes. For $w = 32, 48, 64$, the common goal for the different hashing schemes is to produce w-bit hash values from w-bit keys. Recall that:

Univ and TwoIndep are the fast multiplication-shift based methods from § 1.2.2 for universal hashing and 2-independent hashing, respectively. If w is not clear from the context, we will use w as a suffix label as in Univ32 and TwoIndep32 for $w = 32$. The actual code for Univ$w$ and TwoIndep$w$ can be found in § A.3–A.5. We note that for TwoIndep64 in § A.5 we employ a cross product trick from [3].

CWtrick32, CWtrick48 and CWtrick64 are CW-trick schemes as described in § 1.2.2 with Mersenne primes $2^{61} - 1$, $2^{61} - 1$, and $2^{89} - 1$, respectively. The actual code for CWtrick32, CWtrick48, and CWtrick64 can be found in § A.12–A.14. All the code has been carefully optimized for evaluation speed.

ShortTable and CharTable are instances of the new tabulation based 5-independent hashing schemes from § 3.3 with 16-bit and 8-bit input characters, respectively. We consider them both with and without the compression in § 3.4. The code for ShortTable and CharTable with different key lengths can be found in § A.6–A.11. We note that the code presented does not include the initialization of tables, but only the code using the tables to compute the hash values.

Experimental setup. To evaluate the performance of the different schemes, we recorded the total running time over 10 million hash computations. In our original extended abstract [30], we simply used $1, 2, \ldots, 10^7$ as the sequence of input keys, but now we realized that this gives tabulation-based hashing an unfair advantage. Specifically, since the key is incremented by 1 each time, the high order character only changed very rarely. For a fairer comparison, we first filled a table with a million pseudo-random keys. Next we timed cycling through the array 10 times, resulting in a total of 10 million hash computations. The sequential reading of keys from the array is included in the timing, but this is very fast compared with the hash computation. Table 5.2 compares the different algorithms in terms of average hashing overhead (in nanoseconds) on three different computers (summarized in Table 5.1) using exactly the same code, data, and random numbers.

Name	Processor Type	Clock Speed	Number of Cores	Architecture	Cache Size	Launch Year
Computer A	Intel® Xeon™	3.2 GHz	1	32-bit	2 MB	2005
Computer B	AMD® Dual-Core Opteron™	2 GHz	2	64-bit	2 MB	2006
Computer C	Intel® Core™ i7-640M	2.8 GHz	2	64-bit	4 MB	2010

TABLE 5.1
Computers used in the experiments.

Computer A. Computer A is a single-core Intel® Xeon™ 3.2 GHz 32-bit processor with 2 MB cache initially launched in 2005. The 32-bit architecture shows up clearly in the performance difference between Univ32 and TwoIndep32. The difference between the two schemes is that Univ32 does a 32-bit multiplication while TwoIndep32 does a 64-bit multiplication. We have that Univ32 is almost three times faster than TwoIndep32. This could indicate that Computer A uses its fast 32-bit multiplication to simulate 64-bit multiplication.

We now consider the hashing of 32-bit keys with higher independence. We see that the tabulation based methods are significantly faster than the polynomial methods in CWtrick. Somewhat surprisingly, we see that CharTable is faster than ShortTable despite the fact that ShortTable uses the much simpler 2-character code. However, ShortTable uses tables of size 2^{16} whereas CharTable only uses tables of size 2^8. The smaller tables are in faster memory leading to faster execution even though the code with CharTable is more complex and uses 7 instead of 3 lookups. It is also worth noting that compression does not make any big difference. The code is a bit more complicated, but the space is smaller. Compared with the 5-independent CWtrick32, CharTable gains a factor of 10 in speed.

We now consider what happens when we move to larger keys and hash values. For the multiplication based methods, we generally expect a quadratic slow down. The point is that all multiplications are implemented with 64-bit multiplication which in C only gives a 64 bit output. This means that all exact multiplications are simulated by the 64-bit multiplications of 32 bit numbers. However, when we move to 48 bits, we can still use the same Mersenne prime $2^{61} - 1$ as with 32 bits while we for 64 bits need to step up to $2^{89} - 1$. This could explain why CWtrick48 is comparatively close in performance to CWtrick32.

For the tabulation based methods, recall that it is only ShortTable32 which benefits from the simple 2-character scheme. It is therefore not surprising that with longer keys we see a much clearer advantage of CharTable over ShortTable. Generally, the number of lookups grow linearly as $2c - 1$. However, the work we do to compute the derived keys is quadratic. More precisely, in §3.4 we did $O(c)$ operations on the vectors in G_j that themselves are twice as long as the keys. It is a bit surprising that CharTable48 is five times slower than CharTable32, but because of differences in compilation and pipelining, we can never hope for an exact understanding of the performance based on the speed of individual operations, *e.g.*, sometimes adding operations can even reduce the running time. One can try to measure individual operations using the cycle counter, but this interrupts and slows down the flow of the overall hash computation. We see that CharTable48 gains nearly a factor of 3 in speed over the 5-independent CWtrick48 while CharTable64 gains more than a factor of 4 over the 5-independent CWtrick64.

Computer B. Computer B is an AMD® Dual-Core Opteron™ 270 / 2 GHz 64-bit processor with 2 MB cache initially launched in 2006. The advantage of a 64-bit processor shows clearly in TwoIndep32 which is the first scheme

		hashing time (nanoseconds)		
bits	algorithm	computer A	computer B	computer C
32	Univ32	1.87	2.79	1.04
32	TwoIndep32	5.77	2.95	1.26
32	CWtrick32 (4-independent)	80.22	17.44	11.11
32	CWtrick32 (5-independent)	100.05	22.91	14.10
32	ShortTable32 (without compression)	18.84	23.90	6.24
32	ShortTable32 (with compression)	17.36	19.38	6.07
32	CharTable32 (without compression)	**9.48**	**11.12**	**5.77**
32	CharTable32 (with compression)	10.21	12.66	6.67
48	Univ48	5.32	5.25	1.54
48	TwoIndep48	20.77	6.99	3.54
48	CWtrick48 (4-independent)	100.93	31.19	17.26
48	CWtrick48 (5-independent)	139.64	40.28	23.08
48	ShortTable48 (without compression)	225.85	183.57	46.76
48	ShortTable48 (with compression)	219.12	178.57	42.58
48	CharTable48 (without compression)	**47.90**	**17.17**	**11.84**
48	CharTable48 (with compression)	50.86	17.41	11.89
64	Univ64	7.16	5.15	1.28
64	TwoIndep64	22.95	6.06	2.46
64	CWtrick64 (4-independent)	245.29	46.28	27.47
64	CWtrick64 (5-independent)	324.96	59.06	34.62
64	ShortTable64 (without compression)	294.93	226.19	85.87
64	ShortTable64 (with compression)	282.71	218.62	77.55
64	CharTable64 (without compression)	**74.26**	53.56	20.17
64	CharTable64 (with compression)	76.21	**22.04**	**19.05**

TABLE 5.2
Average time per hash computation for 10 million hash computations. For each key length and computer, we highlight the best performance on each computer. Note that CharTable and ShortTable are both 5-independent.

where we use 64-bit multiplication. On Computer A, Univ32, which only uses 32-bit multiplication, was much faster than TwoIndep32. But on Computer B, Univ32 is only marginally faster. So with the 64-bit architecture, we gain very little by working on 32-bit numbers. Generally it appears that all schemes based on 64-bit multiplications run almost a factor of 3–5 faster on Computer B than on Computer A.

For hashing with higher independence, the advantage of CharTable over CWtrick is smaller on Computer B whose fast 64-bit arithmetic is a big win for CWtrick, yet the advantage is always more than a factor of 2. It is also worth noting that CharTable64 with compression runs a factor of 2.5 faster than CharTable64 without compression. This indicates that without compression, Computer B does not have enough fast memory to hold all the tables used by CharTable64.

Computer C. Computer C is a dual-core Intel® Core™ i7-640M 2.8 GHz 64-bit processors with 4 MB cache initially launched in 2010. All the hashing schemes run considerably faster on Computer C than on Computer B, but the relative ranking in the speeds of different schemes is fairly consistent with Computer B. Specifically, CharTable consistently performs the best and gains a factor of 1.8–2.4 in speed compared with the 5-independent CWtrick.

A general comment. We note that there is no reason to believe that computers of the future are moving particularly in the direction of Computer A or Computer B or Computer C. Sometimes the cache gets faster, sometimes the processor gets faster. As opposed to the multiplication based CWtrick, tabulation based hashing allows us to benefit from more fast cache, *e.g.*, with enough fast cache, ShortTable might outperform CharTable since it uses fewer operations. So far, CharTable has been the fastest algorithm for 4- and 5-independent hashing on every computer we tried.

Stepping a bit back, one could argue that we should not expect computers configured with much faster multiplication than cache. The basic point is that most data processing involves frequent cache and memory access. Therefore, even if it was technically possible, it would normally be wasteful to configure a computer with much faster multiplication than cache. Conversely, however, there is a lot of data processing that does not use multiplication, so it is easier to imagine real computers configured with faster cache than multiplication.

6. Second moment estimation. In this section, we are going to study how the different hashing schemes perform in the estimation of the second moment of a stream. The standard methods [1, 7] require 4-independent hashing to bound the variance of the estimates. On the other hand, we know from [19] that 2-independence suffices if the input has enough entropy. Moreover, it is common practice to ignore theoretical warnings about worst-case, and just use one's favorite hashing scheme. We use Univ and TwoIndep as typical representatives of such naïve "practical" choices. The question is if there are realistic inputs for which we get penalized for the naïve choice. A bad example will have to be highly structured so as to have low entropy.

More formally, let $\mathcal{S} = (a_1, w_1), (a_2, w_2), \ldots, (a_s, w_s)$ be a data stream, where each key a_i is a member of $[u]$. Let $v_a = \sum_{i:\ a_i = a} w_i$ be the total weight associated with key $a \in [u]$. Define, for each $j \geq 0$,

$$F_j = \sum_{a \in [u]} v_a^j.$$

Here we want to estimate the second moment F_2. The generic method used in [1, 7] is that when an item arrives, a hash function is computed of the key, and then the hash value is used to update a certain probabilistic summary data structure (often called a "sketch"). Note that keeping track of all the keys is out of the question. At any point in time, the sketch provides an unbiased estimator of the second moment of the data stream seen so far. In order to provably control the variance of the estimator, the hash function used has to be 4-independent. Below we will present a new estimator with the same variance guarantees as the previous methods, but which gains roughly a factor of 2 in speed. As discussed in the introduction, this speed was necessary for a real Internet application [18] that needed to keep up with a high-end router.

We are going to explore how the estimator performs with different hashing schemes, considering both speed and accuracy.

6.1. Second moment estimators.

Classic estimator. The classic method for estimating F_2 by Alon *et al.* [1] uses n counters c_i ($i \in [n]$) and n independent 4-independent hash functions s_i that map $[u]$ into $\{-1, 1\}$. When a new data item (a, w) arrives, for all $i \in [n]$, let $c_i = c_i + s_i(a) \cdot w$. After the stream has been processed, F_2 is estimated as $X_{\text{classic}} = \sum_{i \in [n]} c_i^2 / n$. Following the analysis in [1], we have $\mathrm{E}\left[X_{\text{classic}}\right] = F_2$ and $\mathrm{Var}\left[X_{\text{classic}}\right] = \sum_{a \neq b} 2\, v_a^2 v_b^2 = 2(F_2^2 - F_4)/n$.

Count sketch based estimator. Charikar *et al.* [7] described a data structure called *count sketch* for keeping track of frequent items in a data stream. We can adapt count sketch to support second moment estimation. Using this method, we need n counters c_i ($i \in [n]$) and two independent 4-independent hash functions $h : [u] \to [n]$ and $s : [u] \to \{-1, 1\}$. When a new data item (a, w) arrives, a single counter $c_{h(a)}$ is updated using $c_{h(a)} \mathrel{+}= s(a) \cdot w$. F_2 is then estimated as $X_{\text{count\_sketch}} = \sum_{i \in [n]} c_i^2$. We can prove that $\mathrm{E}\left[X_{\text{count\_sketch}}\right] = F_2$ and $\mathrm{Var}\left[X_{\text{count\_sketch}}\right] = 2(F_2^2 - F_4)/n$. Therefore, $X_{\text{count\_sketch}}$ achieves the same variance as X_{classic} with substantially lower update cost.

Fast count sketch based estimator. An alternative way of implementing the count sketch scheme is to use $2n$ counters c_i ($i \in [2n]$) and a 4-independent hash function $h : [u] \to [2n]$. When a new data item (a, w) arrives, w is directly added to the counter $c_{h(a)}$: $c_{h(a)} \mathrel{+}= w$. In the end F_2 is estimated using the alternating sum $X_{\text{fast\_count\_sketch}} = \sum_{i \in [n]} (c_{2i} - c_{2i+1})^2$. $X_{\text{fast\_count\_sketch}}$ achieves the same variance as $X_{\text{count\_sketch}}$, but is faster because the direct update of a counter based on a single hash value is much simpler. However, such simplicity comes at the cost of doubling the space.

A new estimator. Here we present a new estimator that achieves the same speed and variance as the fast count sketch based estimator without having to double the space. Instead of using $2n$ counters, our new method uses $m = n + 1$ counters c_i ($i \in [m]$), and a 4-independent hash function $h : [u] \to [m]$. The update algorithm is exactly the same as that of the fast count sketch based estimator: when a new data item (a, w) arrives, w is added to the counter $c_{h(a)}$: $c_{h(a)} \mathrel{+}= w$. But the estimation formula is quite different. We use

$$X_{\text{new}} = \frac{m}{m-1} \sum_{i \in [m]} c_i^2 - \frac{1}{m-1} \Big(\sum_{i \in [m]} c_i\Big)^2.$$

Note that we do not worry about the cost of adding up counters done in the end. Hence, it is not considered a problem to have a more complex sum for this. In §6.2, we prove

THEOREM 6.1. *If h is 2-independent,* $\mathrm{E}[X_{\text{new}}] = F_2$. *If h is 4-independent,* $\mathrm{Var}[X_{\text{new}}] = 2(F_2^2 - F_4)/(m-1) = 2(F_2^2 - F_4)/n$.

6.2. Analysis of the second moment estimator.

Notations:. For any possibly identical $a, b \in [u]$, let $a \sim b$ denote $h(a) = h(b)$, and $a \nsim b$ denote $h(a) \neq h(b)$. Let $a \underset{\neq}{\sim} b$ denote $a \sim b$ but $a \neq b$.

Clearly, we have

$$\begin{aligned} X_{\text{new}} &= \frac{m}{m-1} \sum_{a \sim b} v_a v_b - \frac{1}{m-1} \sum_{a,b} v_a v_b \\ &= \frac{m}{m-1} \left(\sum_a v_a^2 + \sum_{a \underset{\neq}{\sim} b} v_a v_b \right) - \frac{1}{m-1} \left(\sum_a v_a^2 + \sum_{a \underset{\neq}{\sim} b} v_a v_b + \sum_{a \nsim b} v_a v_b \right) \\ &= F_2 + \sum_{a \underset{\neq}{\sim} b} v_a v_b - \frac{1}{m-1} \sum_{a \nsim b} v_a v_b. \end{aligned} \tag{6.1}$$

Define

$$X_{a,b} = \begin{cases} 1 & \text{if } a \sim b; \\ -\frac{1}{m-1} & \text{otherwise.} \end{cases}$$

Then (6.1) becomes

$$X_{\text{new}} = F_2 + \sum_{a \neq b} v_a v_b X_{a,b}. \tag{6.2}$$

If h is 2-universal, then for any distinct $a, b \in [u]$, we have

$$\mathrm{E}[X_{a,b}] = 0, \tag{6.3}$$

$$\mathrm{E}\left[X_{a,b}^2\right] = \frac{1}{m-1}. \tag{6.4}$$

In addition, if h is 4-universal, then for any distinct $a, b \in [u]$ and distinct $c, d \in [u]$ such that $\{a, b\} \neq \{c, d\}$, we have

$$\mathrm{E}[X_{a,b} X_{c,d}] = \mathrm{E}[X_{a,b}]\,\mathrm{E}[X_{c,d}] =_{(6.3)} 0. \tag{6.5}$$

THEOREM 6.2. *If h is 2-universal, then*

$$\mathrm{E}[X_{\text{new}}] = F_2. \tag{6.6}$$

Proof.

$$\mathrm{E}[X_{\text{new}}] =_{(6.2)} F_2 + \mathrm{E}\left[\sum_{a \neq b} v_a v_b X_{a,b}\right] = F_2 + \sum_{a \neq b} v_a v_b \mathrm{E}[X_{a,b}] =_{(6.3)} F_2.$$

□

THEOREM 6.3. *If h is 4-universal, then*

$$\mathrm{Var}[X_{\text{new}}] = \frac{2}{m-1}(F_2^2 - F_4). \tag{6.7}$$

Proof.

$$\text{Var}\left[X_{\text{new}}\right] = \text{E}\left[(X_{\text{new}} - F_2)^2\right] =_{(6.2)} E\left[(\textstyle\sum_{a\neq b} v_a v_b X_{a,b})^2\right] \tag{6.8}$$

$$= 2\sum_{a\neq b} v_a^2 v_b^2 \text{E}\left[X_{a,b}^2\right] + \sum_{\substack{a\neq b\wedge c\neq d\\ \{a,b\}\neq\{c,d\}}} v_a v_b v_c v_d \text{E}\left[X_{a,b}X_{c,d}\right] \tag{6.9}$$

$$=_{(6.4)\ (6.5)} 2\sum_{a\neq b} v_a^2 v_b^2 \frac{1}{m-1} = \frac{2}{m-1}(F_2^2 - F_4). \tag{6.10}$$

□

6.3. Dealing with long hash keys. Our constructions of 4-universal hash functions are most efficient when the hash key length coincides with single or double machine word length. If the hash keys are longer, we show here that we can first hash them into a smaller domain $[2^{64}]$ using 2-universal hashing with little degradation on the accuracy of the final estimator.

THEOREM 6.4. *Let h be a 2-universal hash function from $[u]$ into $[u']$. Let $X = \sum_{a\sim b} v_a v_b$. We have*

$$\text{E}\left[|X - F_2|\right] \leq \frac{1}{u'}(\sum_a |v_a|)^2. \tag{6.11}$$

Proof. Let

$$Y_{a,b} = \begin{cases} 1 & \text{if } a\sim b; \\ 0 & \text{otherwise.}\end{cases}$$

Since h is 2-universal, for any distinct $a, b \in [u]$, we have

$$\text{E}\left[Y_{a,b}\right] = 1/u'. \tag{6.12}$$

Clearly,

$$X = \sum_a v_a^2 + \sum_{a \underset{\neq}{\sim} b} v_a v_b = F_2 + \sum_{a\neq b} v_a v_b Y_{a,b}. \tag{6.13}$$

Therefore,

$$\text{E}\left[|X - F_2|\right] \leq_{(6.13)} \sum_{a\neq b} |v_a v_b| \text{E}\left[Y_{a,b}\right] =_{(6.12)} \sum_{a\neq b} |v_a v_b| \frac{1}{u'} \leq \frac{1}{u'}(\sum_a |v_a|)^2.$$

□

If the mass of $\{|v_a|\}$ is distributed on $n' \ll 2^{64}$ keys, which is almost certainly the case in any foreseeable future, then we have

$$(\sum_a |v_a|)^2/F_2 \leq n' \ll 2^{64}.$$

Therefore, if we choose u' to be 2^{64}, then by (6.11) and Markov's Inequality, $|X - F_2|/F_2$ will be small with very high probability.

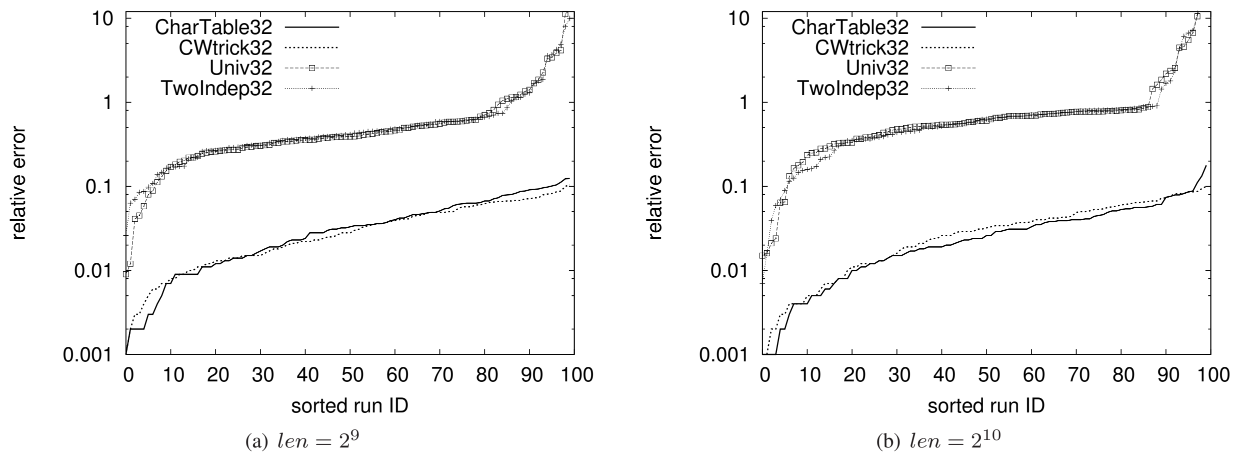
6.4. Performance evaluation. We now consider the effect of using different hash functions in the estimator. We know from Theorem 6.1 that the standard deviation is bounded if we use a 4-independent hash function, but we wonder if something can go wrong with a weaker hash function like Univ or TwoIndep. Recall that a bad example would have to be highly structured with low entropy. As an extreme but not unrealistic example, we consider intervals of unit weighted keys.

Experiments. The code for second moment estimation can be found in § A.15. We used $m = 2^{10}$, which gives a deviation bound of roughly $\sqrt{2/(m-1)} \approx 2^{-4.5} = 4.42\%$. Note that the counter space is less than that used for the tables of CharTable32. We therefore expect everything to fit in cache.

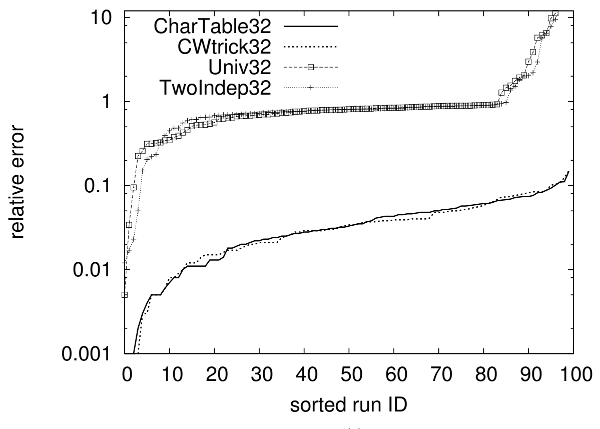
To illustrate the effect of different hashing schemes on the accuracy of the second moment estimator, we conducted the following simple experiment. First we chose a target interval length $len \in \{2^9, 2^{10}, 2^{11}\}$. The input keys are consecutive integers that form an interval $[i, j)$, where i is randomly chosen from $[0, len]$ and j is from $[len, 2len)$, Each input key is associated with a unit weight (*i.e.*, 1), so the true second moment is simply $j - i + 1$. Note that 2^9 is half the number of counters, so this could be thought of as representing a few same weight heavy items. Conversely, 2^{11} is twice the number of counters, so this is more like a large uniform distribution. For each $len \in \{2^9, 2^{10}, 2^{11}\}$, we consider 100 different sets of random seeds for the hash function (from `random.org`), and plot the relative error $\frac{|F_2^{\rm est} - F_2^{\rm real}|}{F_2^{\rm real}}$ from best to worst in Figure 6.1. We also compute the root mean squared relative error (RMSRE) across all the experiment runs, which is defined as the square root of the average squared relative errors. It is the empirical standard deviation and is presented in Table 6.1.

In Table 6.2, we compared the processing times on different computers with the different hashing schemes applied to the keys in pseudo-random order. We report both the time to update the sketch, and the time to do the hashing alone.

Findings. A quick look at Figure 6.1 reveals that the accuracy with Univ and TwoIndep is orders of magnitude worse than that with the 4-independent schemes CWtrick and CharTable. Looking at Table 6.1 we see that the RMSRE of CharTable32 and CWtrick32 is pretty consistent with our upper bound $\sqrt{2/(m-1)} \approx 2^{-4.5} = 0.0442$ on the standard deviation.



(a) $len = 2^9$ (b) $len = 2^{10}$



(c) $len = 2^{11}$

FIG. 6.1. *Impact of hash functions on the accuracy of our second moment estimator. Each data point represents the result for one experiment run (with a different random seed).*

algorithm	RMSRE		
	$len = 2^9$	$len = 2^{10}$	$len = 2^{11}$
CharTable32	0.036	0.049	0.035
CWtrick32	0.052	0.046	0.047
Univ	1.693	3.472	6.906
TwoIndep	1.693	3.481	6.879

TABLE 6.1
RMSRE given by different hash functions.

bits	algorithm	sketch update time (ns)			hashing time (ns)		
		Computer A	B	C	Computer A	B	C
32	Univ	6.64	4.73	1.81	1.90	2.95	1.05
32	TwoIndep	12.93	6.09	2.13	5.78	3.13	1.27
32	CWtrick (4-independent)	89.70	22.48	11.60	80.12	18.34	11.83
32	ShortTable (without compression)	51.00	45.83	8.34	18.78	24.86	6.24
32	ShortTable (with compression)	47.48	30.03	8.36	17.09	18.94	6.12
32	CharTable (without compression)	27.56	**13.18**	**7.13**	**9.52**	**11.17**	**5.80**
32	CharTable (with compression)	**20.71**	15.37	8.77	10.51	13.05	6.66

TABLE 6.2
Average time per sketch update for 10 million updates on Computer A, B and C (summarized in Table 5.1). The sketch uses $m = 2^{10}$ counters. We highlight the best performance on each computer.

Table 6.2 summarizes the sketch update time with the different hash functions. The time for the hashing alone on the same keys is also reported, but we cannot conclude that the time to change the sketch is the total time minus the hashing time, for with pipelining, the total may be less than the sum of the parts.

As when doing the hashing alone, the sketch update is much faster with Univ32 and TwoIndep32. Among 4-independent hashing schemes, CharTable32 consistently outperforms the other on all three computers. In particular, it outperforms CWtrick which was the previously fastest 4-independent scheme. More precisely, on Computer A, CharTable32 is a factor of 4.33 faster than CWtrick32. On Computer B and C, the performance gap is smaller but still significant — CharTable32 is more than a factor of 1.6 faster than the 4-independent CWtrick32.

Summing up, we definitely get the fastest second moment estimation if we use weak hash functions like Univ or TwoIndep, but then there are inputs for which the estimates have no accuracy. If we want to enjoy the provable guarantees with 4-independent hashing, then our new tabulated CharTable is substantially faster than the classic CWtrick.

7. Linear probing.

Linear probing is one of the most popular implementations of dynamic hash tables storing all keys in a single array. Below we generally assume that the keys fill half the array. When we get a key, we first hash it to a location. Next we probe consecutive locations until the key or an empty location is found. Giving birth to the analysis of algorithms, Knuth [16] proved that linear probing uses an expected constant number of probes if the hash function is truly random. More recently, Pagh *et al.* [22] proved that 5-independent hashing suffices for an expected constant number of probes per update or search, and this is for any possible set of input keys. We also note that Thorup [29] has shown that we preserve this constant expected time if we first do universal hashing into a domain of the same size n as the number of keys. Thus if we can provide a fast 5-independent hashing for 32- or at most 64-bit keys, then we get a fast implementation of linear probing with provably good expected performance.

We note that this application is very different from that of second moment estimation. One difference is that high independence is now used to bound the running time rather than for accuracy of results. Another is that the code for linear probing is more complicated using a large array that does not fit in cache. This means that there will now be competition for the cache. However, we expect the small tables used by CharTable to reside in cache most of the time.

As in the second moment estimation, our next question is whether 5-independent hashing with its good theoretical performance would also perform better in practice than the naïve practical Univ and TwoIndep. We note that this choice is not that naïve, for universal hashing suffices for other implementations of hash tables like simple chaining, and it is only recently [22] that we have learned that the theoretical performance is so sensitive to the degree of independence.

From [19] we know that a bad case needs to have low entropy and again we study a dense interval. Note that a dense interval is a very realistic case for hash tables, *e.g.*, if dealing with ASCII characters, the alphabet forms a commonly used interval. In practice, we may have a concentration in a dense interval together with some outliers. An example could be a discrete heavy-tailed distribution hitting the small numbers densely, yet having a few much larger numbers.

It was already known from [13] that structured input can cause linear probing to be slower than other methods, but in [13] the slowness was largely due to the use of the old-fashioned direct 2-independent method from (1.2). This particular hash function was proved to be bad also for linear probing in [22]. Here we compete against the multiplication-shift based methods Univ and TwoIndep that are an order of magnitude faster. Also our experiments are special in that we do many experiments to study robustness.

Experiments. We conduct experiments to evaluate the impact of hash functions on the performance of linear probing. We construct a hash table with 2^{21} entries. We then insert entries into the hash table one by one until the array is half full. From then on we perform 10 million insertion/deletion cycles. During each cycle, we first insert a new key into the hash table, and then delete an old key from the hash table, which was inserted into the hash table a million steps back. We then compute the average amount of time spent for each update operation (*i.e.*, either insertion or deletion). In addition to such timing results, we report the average number of linear probes involved per update, which is independent of machine configurations.

For our experiments, we construct two very different key sequences:

- *A dense interval.* In this case, we use a random permutation of $[2^{20}]$ as the key sequence. To do so, we first generate 2^{20} distinct 32-bit random numbers 10-independently and then sort these numbers to obtain a permutation of their original indices. At any point, we have the last 1 million keys in the table, representing roughly 95% of $[2^{20}]$.
- *Random keys.* In this case, we generate 2^{20} distinct 32-bit random numbers 10-independently and use them as the key sequence. This is also the sequence used when we test the speed of the hash functions above.

We store the constructed input key sequence in an array (of size 2^{20}) and then cycle through the array to obtain the next key to insert or delete. For the same input key sequence, we repeat the experiment 100 times. In each experiment, we obtain a different set of random numbers (from `random.org`) and use them to initialize all the hash functions.

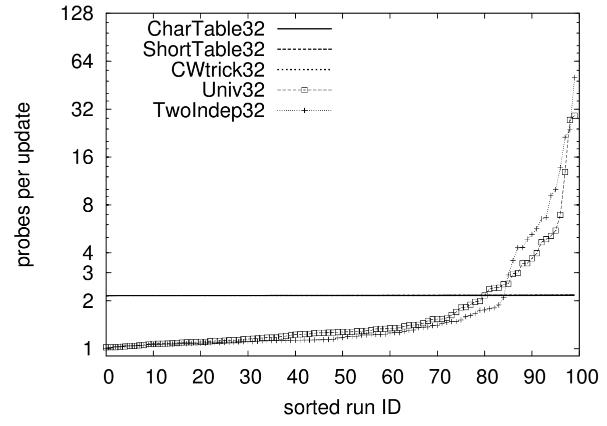
Findings. Figures 7.1–7.2 show the results. In each figure we have a particular input key sequence for which we study linear probing with different hash functions. For each hash function, we consider 100 different sets of random seeds, and plot the performance from best to worst. In (a) we have the average number of probes per operation over 10 million insert/delete cycles; in (b), (c) and (d) we have the average time per operation on Computer A, B and C, respectively. The random seeds are the same across (a)–(d), so the same probe count holds for all three computers.

We first consider the combinatorial probe count measure (a). In Figure 7.1 we have a dense interval. For such highly structured input, neither Univ nor TwoIndep is robust. For some experiments, the use of Univ and TwoIndep require a significant number of linear probes per insertion/deletion. In contrast, with our 5-independent schemes, CharTable32 and CWtrick32, we do not see any obvious difference between the best and the worst experiment. The average probe count ranges from 3.23 to 3.26. The much smaller variance for 5-independent hashing is expected, because the result from [22] also limits the variance on the probes per operation. More precisely, to bound the expected number of probes, they show in the proof of [21, Theorem 4.3] that the probability of doing more than ℓ probes is $O(1/\ell^2)$, hence that the expected number is $O(\sum_{\ell\in[n]} 1/\ell^2) = O(1)$. We can also use this to bound the variance, which within a factor of 2 can be computed by letting the ℓ-th probe pay ℓ, leading to a variance bound of $O(\sum_{\ell\in[n]} \ell/\ell^2) = O(\log n)$. Overall, for 10–20% experiments (which use different random seeds for initializing the hash function), CharTable32 and CWtrick32 significantly outperform Univ and TwoIndep in terms of the average probe count.

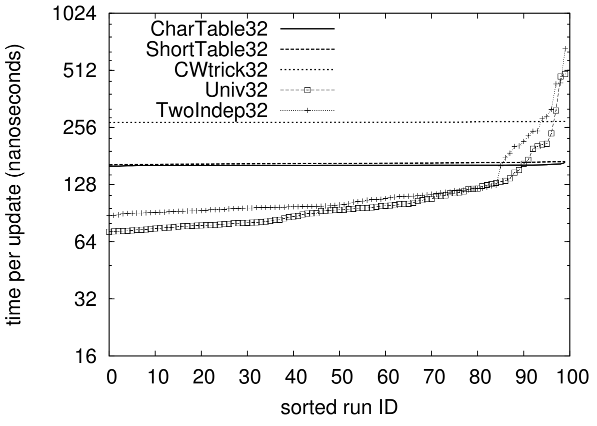
Now consider instead the average probe count in Figure 7.2 (a) for a random set of keys. These are 10-independent, so in fact, much more random than our 5-independent hash functions CharTable32 and CWtrick32. With so much randomness in the keys, the limited randomness in the hash functions has no impact, and now we see that all schemes have a robust average probe count of around 3.24. In particular, the heavy tails disappear for Univ and TwoIndep.

We now switch our attention to the average time spent per update on the two computers. In essence this is the cost of hash computation plus the cost of memory access for the probes in the table. The latter is dominated by the cache miss from the initial probe at the hash location. For the random keys in Figure 7.2, we essentially have the same number of probes with all the hash function, so the differences in time are due to the differences in hash computation. Thus it is not surprising that we see the same ordering as the one found for the hash computations alone in Table 5.2.

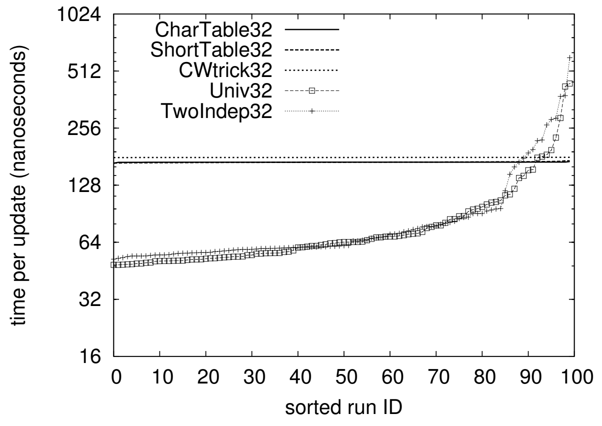
In Figure 7.2 (b) we have the results for Computer A. A slightly surprising finding is that the differences are bigger



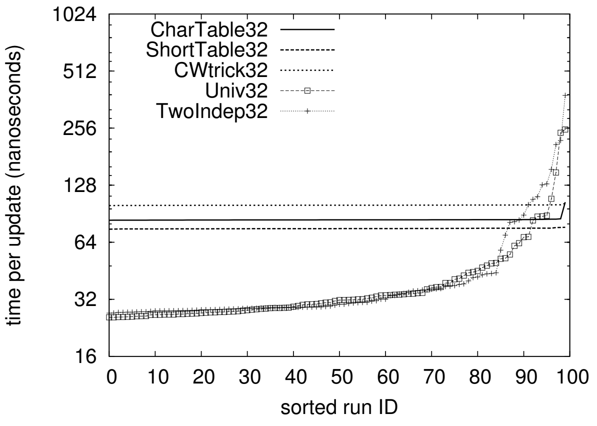
(a) Average probes per update



(b) Average time per update (Computer A)



(c) Average time per update (Computer B)



(d) Average time per update (Computer C)

FIG. 7.1. *Impact of hash functions on linear probing performance with permuted input sequence. Each data point represents the result for one experiment run.*

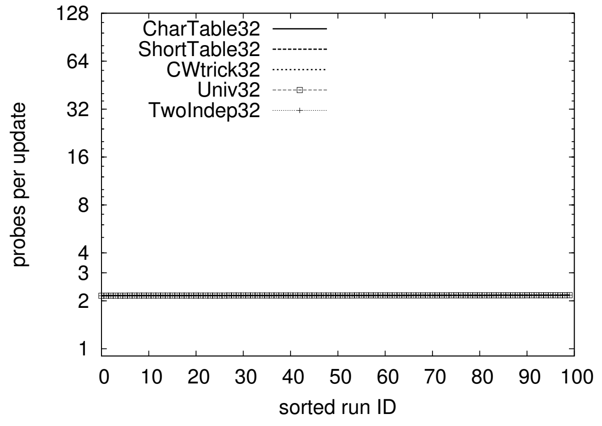
than those in Table 5.2. More precisely, in Figure 7.2 (b) we see that CharTable32 is about 25 nanoseconds slower than Univ, and CWtrick32 is about 130 nanoseconds slower. In both cases, this is much more than the cost of computing the hash function itself. If it had only been CharTable32 that had been slow, we would have expected this to be due to competition over the cache. However, CWtrick is just using a few registers for its seeds, so a more likely explanation is that the optimizing compiler does not do as good a job when faced with the more complicated hash functions.

Figure 7.2 (c) and (d) show the results for Computer B and C, respectively. Again we see that CharTable32 and CWtrick32 are slower than what we would expect from Table 5.2, and relatively speaking, the difference between CharTable32 and CWtrick32 is reduced, yet we still have CharTable32 coming out as the fastest 5-independent scheme.

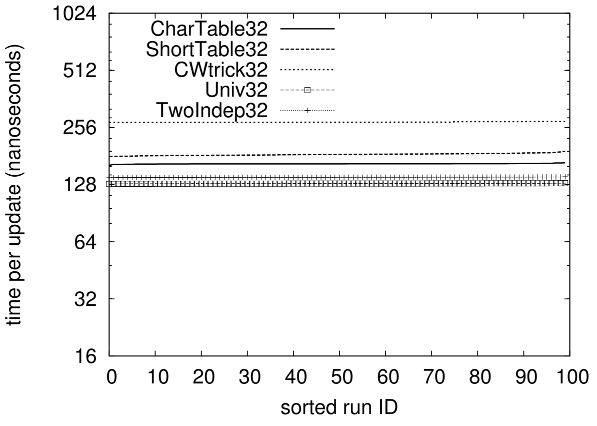
If we now consider Figure 7.1 (b), (c) and (d), we see the combined effect of differences in hashing speed and differences in number of probes. This gives Univ and TwoIndep an additive advantage compared with the pure probe count in (a), so now it is only for 10% of the cases that CharTable32 and CWtrick32 perform better than Univ and TwoIndep. Yet the heavy tails persist, so Univ and TwoIndep are still much less robust.

8. Concluding remarks.

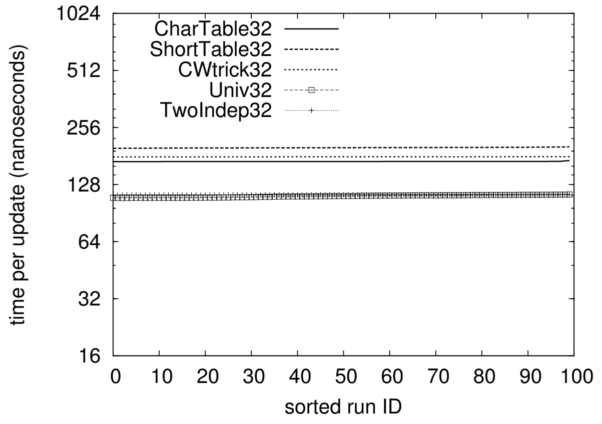
We developed simple tabulation based schemes for 4- and 5-independent hashing. In addition, k-independent schemes are constructed for $k > 5$. The idea in tabulation is to exploit the fact that tables over characters can be stored in fast memory. For keys divided into c characters, we get 5-independent hashing using only $2c - 1$ character table lookups. We saw that the version CharTable based on 8-bit characters was much faster than the classic polynomial based method CWtrick optimized using Mersenne primes as originally suggested by Carter and



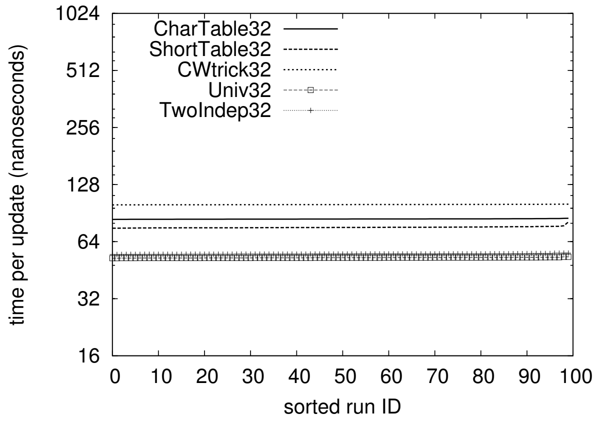
(a) Average probes per update



(b) Average time per update (Computer A)



(c) Average time per update (Computer B)



(d) Average time per update (Computer C)

FIG. 7.2. *Impact of hash functions on linear probing performance with random input sequence. Each data point represents the result for one experiment run.*

Wegman [6]. When the hashing is studied in isolation (*cf.* Table 5.2) the difference between CharTable and CWtrick is by a factor of 1.8 to 10. The smallest gain is on Computer B which has really fast 64-bit multiplications as needed by CWtrick, but seemingly slightly slower memory.

We also studied the hash functions inside applications of second moment estimation and linear probing. The advantage in speed was smaller inside the more complicated case of linear probing, but CharTable continued to outperform CWtrick in all scenarios considered.

Another aspect of our application experiments was to see if we really benefit from the performance guarantees we get with high independence. For second moment estimation, 4-independence is used to bound the variance of the estimates, and for linear probing, 5-independence is used to prove expected constant time for any operation. On the other hand, we know from [19] that 2-independence suffices if the input has enough entropy in the data. For both applications, we studied the quite realistic case where the input is a dense interval, and found that the fast practical 2-independent method TwoIndep failed miserably. Thus, for robustness, we recommend listening to the theory and go for hashing schemes with provably good performance. In the above applications, 4- and 5-independence has been proved to suffice, and our CharTable provides the most efficient implementation for this degree of independence.

Open problems. We mention some open problems:

- We conjecture that our $2c - 1$ lookups is best possible for tabulation based 5-independent hashing.
- Generally we wonder what the least number of lookups needed for k-independent hashing is. We get down to $(k-1)(c-1)+1$ lookups while for high independence, we have Siegel's method [27] getting $n^{\Omega(1/c)}$-

independence using $c^{\Theta(c)}$ lookups.

Recent developments. The work here has inspired some later developments. Inspired by the negative experimental findings for TwoIndep in this paper, [23] has proved some negative results; namely that TwoIndep has $\Theta(\log n)$ expected performance in linear probing when applied to a dense intervals, and that there exists 4-independent hash functions leading to $\Theta(\log n)$ search time for some input. We note that this does not preclude the existence of other hash function working well despite independence below 5. In fact, very recently [24], it has been shown that simple tabulation hashing suffices for our applications even though simple tabulation hashing is not 4-independent. On the one hand, this suggests that the concept of k-independence is less relevant for many applications. On the other hand, k-independence has a strong tradition within probabilistic analysis, and our 5-independent tabulation scheme still has some interesting relations to the work in [24]. Generally we would get better hidden constants with our derived keys. More fundamentally, our derived keys violate a negative example from [24] which shows that Cuckoo hashing fails with probability $1/n^{1/3}$ with simple tabulation hashing. It could be that our 5-independent tabulation got closer to the error probability of $1/n$ that we get with truly random hashing.

Acknowledgment. We would like to thank Martin Dietzfelbinger for suggesting Proposition 2.2, which is simple yet powerful in handling any odd number of keys. In our original submission, we had a specialized proof for 5 keys which was more complicated.

REFERENCES

[1] N. Alon, Y. Matias, and M. Szegedy, *The space complexity of approximating the frequency moments*, J. Comput. System Sci., 58 (1999), pp. 137–147.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, *Models and issues in data stream systems*, in Proc. 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 2002, pp. 1–16.

[3] J. R. Black, C. U. Martel, and H. Qi, *Graph and hashing algorithms for modern architectures: Design and performance*, in Proc. 2nd WAE, MPI Saarbrücken, 1998, pp. 37–48.

[4] J. Bloemer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman, *An XOR-based erasure-resilient coding scheme*, Technical Report TR-95-048, ICSI, Berkeley, California, Aug. 1995. http://www.icsi.berkeley.edu/~luby/PAPERS/cauchypap.ps.

[5] Andrei Z. Broder and Anna R. Karlin, *Multi-level adaptive hashing*, in Proc. 1st SODA, 1990, pp. 43–53.

[6] J.L. Carter and M.N. Wegman, *Universal classes of hash functions*, J. Comp. Syst. Sci., 18 (1979), pp. 143–154.

[7] M. Charikar, K. Chen, and M. Farach-Colton, *Finding frequent items in data streams*, in Proc. 29th ICALP, LNCS 2380, 2002, pp. 693–703.

[8] C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk, *Gigascope: A stream database for network applications*, in SIGMOD Conference, 2003, pp. 647–651.

[9] M. Dietzfelbinger, *Universal hashing and k-wise independent random variables via integer arithmetic without primes*, in Proc. 13th STACS, LNCS 1046, 1996, pp. 569–580.

[10] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger, *Polynomial hash functions are reliable*, in Proc. 19th ICALP, LNCS 623, 1992, pp. 235–246.

[11] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen, *A reliable randomized algorithm for the closest-pair problem*, J. Algorithms, 25 (1997), pp. 19–51.

[12] M. Dietzfelbinger and P. Woelfel, *Almost random graphs with simple hash functions*, in Proc. 35th STOC, 2003, pp. 629–638.

[13] G. L. Heileman and W. Luo, *How caching affects hashing*, in Proc. 7th ALENEX, 2005, pp. 141–154.

[14] C. A. R. Hoare, *Algorithm 64: Quicksort*, Commun. ACM, 4 (1961), p. 321.

[15] H.J. Karloff and P. Raghavan, *Randomized algorithms and pseudorandom numbers*, J. ACM, 40 (1993), pp. 454–476.

[16] D. E. Knuth, *Notes on "open" addressing*, 1963. Unpublished memorandum. Available at http://citeseer.ist.psu.edu/knuth63notes.html.

[17] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, Massachusetts, 1973. ISBN 0-201-03803-X.

[18] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, *Sketch-based change detection: Methods, evaluation, and applications*, in Proc. ACM Internet Measurement Conference (IMC-03), 2003, pp. 234–247.

[19] Michael Mitzenmacher and Salil P. Vadhan, *Why simple hash functions work: exploiting the entropy in a data stream*, in Proc. 19th SODA, 2008, pp. 746–755.

[20] S. Muthukrishnan, *Data streams: Algorithms and applications*, Foundations and Trends in Theoretical Computer Science, 1 (2005).

[21] Anna Pagh, Rasmus Pagh, and Milan Ruzic, *Linear probing with constant independence*, in Proc. 39th STOC, 2007, pp. 318–327.

[22] A. Pagh, R. Pagh, and M. Ruzic, *Linear probing with constant independence*, SIAM J. Computing, 39 (2009), pp. 1107–1120.

[23] M. Patrascu and M. Thorup, *On the k-independence required by linear probing and minwise independence*, in Proc. 37th ICALP, 2010, pp. 715–726.

[24] Mihai Patrascu and Mikkel Thorup, *The power of simple tabulation hashing*, CoRR, abs/1011.5200 (2010). To appear at STOC'11.

[25] Luigi Rizzo, *Effective erasure codes for reliable computer communication protocols*, ACM Computer Communication Review, 27 (1997), pp. 24–36.

[26] A. Schönhage and V. Strassen, *Schnelle muliplication großer zahlen*, Computing, 7 (1971), pp. 281–292.

[27] Alan Siegel, *On universal classes of extremely random constant-time hash functions*, SIAM J. Comput., 33 (2004), pp. 505–543.
[28] M. Thorup, *Even strongly universal hashing is pretty fast*, in Proc. 11th SODA, 2000, pp. 496–497.
[29] Mikkel Thorup, *String hashing for linear probing*, in Proc. 20th SODA, 2009, pp. 655–664.
[30] M. Thorup and Y. Zhang, *Tabulation based 4-universal hashing with applications to second moment estimation*, in Proc. 15th SODA, 2004, pp. 608–617.
[31] ———, *Tabulation based 5-universal hashing and linear probing*, in Proc. 12th ALENEX, 2010, pp. 62–76.
[32] Mikkel Thorup and Yin Zhang, *Source code for tabulation-based hashing*, 2011. `http://www.cs.utexas.edu/~yzhang/software/hash.tar.gz`.
[33] M.N. Wegman and J.L. Carter, *New hash functions and their use in authentication and set equality*, J. Comp. Syst. Sci., 22 (1981), pp. 265–279.

Appendix A. Code.

A.1. The C-level instruction count. Besides an experimental comparison, we are going to compare our tabulation based scheme with CW-trick using a coarse-grained analysis of C code. We assume we have a 64-bit processor, and we charge a unit cost for each instruction on one or two 64-bit double word. Of computational instructions we have standard arithmetic operation such as addition and multiplication. We note for both addition and multiplication that overflow beyond the 64 bits are discarded. In particular, this means that multiplication is only used to multiply integers below 2^{32}. With CW-trick we do not need modulus or division so we do not need to worry about the slowness of these operations. Of other computational instructions, we have regular and bit-wise Boolean operations and shifts. Finally, we may define a vector of characters over a double word and extract a character at unit cost.

We note that among the above operations, multiplication is typically the most expensive. Since multiplication is used by CW-trick and not by us, we are generous to the opponent when only charging one unit for multiplication.

We assume that our processor has a small number, say 30, of registers. Here registers are just thought of as memory that is so fast that copying between cells is almost free. Since the registers are controlled by the compiler, it is convenient to ignore it. When running CW-trick, we assume that all variables reside in registers. However, we do charge a unit cost for memory access beyond the registers. This means that our tabulation based methods are charged a unit for each look-up. Obviously, the unit cost is only fair if the tables are small enough to fit in reasonably fast memory. For example, if tables over 16-bit characters were too slow, we could switch to tables over 8-bit characters.

As a final cost, we charge a unit for a jump. For a conditional jump, we charge for the evaluation of the condition and for the jump if it is made. All our procedure calls are inline, so we do not need to charge for them.

We refer to the above cost as the *C-level instruction count*. Here "C-level" refers to the fact that a finer analysis would have to take the concrete machine and compiler into account. Our C-level analysis will be complemented with an experimental evaluation on two different computers, and as it turns out, the C-level instruction count does give a fairly accurate prediction of the actual running times.

Finally, the source code of the complete C implementation is available online [32].

A.2. Common data types and macros.
```
typedef unsigned char       INT8;
typedef unsigned short      INT16;
typedef unsigned int        INT32;
typedef unsigned long long  INT64;
typedef INT64               INT96[3];

// different views of a 64-bit double word
typedef union {
  INT64 as_int64;
  INT16 as_int16s[4];
} int64views;

// different views of a 32-bit single word
typedef union {
  INT64 as_int32;
  INT16 as_int16s[2];
  INT8  as_int8s[4];
} int32views;

typedef struct {
  INT64 h;
  INT64 u;
  INT32 v;
} Entry;

// extract lower and upper 32 bits from INT64
const INT64 LowOnes = (((INT64)1)<<32)-1;
const INT64 Ones48  = (((INT64)1)<<48)-1;
#define HIGH(x)  ((x)>>32)
#define LOW(x)   ((x)&LowOnes)
#define LOW48(x) ((x)&Ones48)
```

A.3. Multiplication-shift based hashing for 32-bit keys.

```
/* universal hashing for 32-bit key x. A is a random 32-bit odd number */
inline INT32 Univ32(INT32 x, INT32 A) {
  return (A*x);
} // 1 C-level instruction

/* 2-independent hashing for 32-bit key x. A and B are random 64-bit numbers */
inline INT32 TwoIndep32(INT32 x, INT64 A, INT64 B) {
  return (INT32) ((A*x + B) >> 32);
} // 3 C-level instructions
```

Univ32 uses 1 C-level instruction, whereas TwoIndep32 uses 3 C-level instructions. Both include one multiplication.

A.4. Multiplication-shift based hashing for 48-bit keys.

```
/* universal hashing for 48-bit key x. A is a random 48-bit odd number */
inline INT32 Univ48(INT64 x, INT64 A) {
  return LOW48(A*x);
} // 2 C-level instructions

/* 2-independent hashing for 48-bit key x.
   A1 and B1 are random 64-bit numbers;
   A0 and B0 are random 32-bit numbers
   stored as 64-bit integers (INT64). */
inline INT32 TwoIndep48(INT32 x, INT64 A0, INT64 A1, INT64 B0, INT64 B1) {
  INT64 H, carry, x0, x1, x0A0;
  x1 = HIGH(x);
  x0 = LOW(x);
  x1A0 = x1*A0;
  carry = HIGH(LOW(x0A0) + B0);
  H = x*A1 + x0*A0 + HIGH(x0A0) + B1 + carry;
  return H;
} // 13 C-level instructions
```

Univ48 uses 2 C-level instruction (which is a multiplication), whereas TwoIndep48 uses 13 C-level instructions (including 3 multiplications).

A.5. Multiplication-shift based hashing for 64-bit keys.

```
/* universal hashing for 64-bit key x. A is a random 64-bit odd number */
inline INT32 Univ64(INT64 x, INT64 A) {
  return (A*x);
} // 1 C-level instruction

/* 2-independent hashing for 64-bit key x.
   A00,A01,A10,A11, and B are random 64-bit numbers;*/
inline INT32 TwoIndep64(INT32 x,
                        INT64 A00, INT64 A1, INT64 A2,
                        INT64 B0, INT64 B1, INT64 B2)
{
  INT64 x0,x1,H;

  x1 = HIGH(x);
  x0 = LOW(x);
  H=(x0+A00)*(x1+A01);
  H>>=32;
  H+=(x0+A10)*(x1+A11);
  H+=B;
  return H;
} // 11 C-level instructions
```
Univ64 uses 1 C-level instruction (which is a multiplication), whereas TwoIndep64 uses 11 C-level instructions (including 2 multiplications).

A.6. Tabulation based hashing for 32-bit keys using 16-bit characters.
```
/* tabulation based hashing for 32-bit key x using 16-bit characters.
   T0, T1, T2 are pre-computed tables */
inline INT32 ShortTable32(INT32 x, INT32 T0[], INT32 T1[], INT32 T2[])
{
  INT32 x0, x1, x2;
  x0 = x&65535;
  x1 = x>>16;
  x2 = x0 + x1;
  x2 = compressShort32(x2);  // optional
  return T0[x0] ^ T1[x1] ^ T2[x2];
} // 8 + 4 = 12 C-level instructions

// optional compression
inline INT32 compressShort32(INT32 i) {
  return 2 - (i>>16) + (i&65535);
} // 4 instructions
```
The code uses 12 C-level instructions (8 without compression), including 3 lookups in 16-bit tables.

A.7. Tabulation based hashing for 32-bit keys using 8-bit characters.
```
/* tabulation based hashing for 32-bit key x using 8-bit characters.
   T0, T1 ... T6 are pre-computed tables */
inline INT32 CharTable32(int32views x,
                         INT32 *T0[], INT32 *T1[], INT32 *T2[], INT32 *T3[],
                         INT32 T4[], INT32 T5[], INT32 T6[])
{
  INT32 *a0, *a1, *a2, *a3, c;

  a0 = T0[x.as_int8s[0]]; a1 = T1[x.as_int8s[1]];
  a2 = T2[x.as_int8s[2]]; a3 = T3[x.as_int8s[3]];

  c  = a0[1] + a1[1] + a2[1] + a3[1];
  c  = compressChar32(c);  // optional

  return a0[0] ^ a1[0] ^ a2[0] ^ a3[0] ^
         T4[c&1023] ^ T5[(c>>10)&1023] ^ T6[c>>20];
} // 32 + 5 = 37 C-level instructions

// optional compression
inline INT32 compressChar32(INT32 i) {
  const INT32 Mask1 = (((INT32)3)<<20) + (((INT32)3)<<10) + 3;
  const INT32 Mask2 = (((INT32)255)<<20) + (((INT32)255)<<10) + 255;
  const INT32 Mask3 = (((INT32)3)<<20) + (((INT32)3)<<10) + 3;
  return Mask1 + (i&Mask2) - ((i>>8)&Mask3);
} // 5 C-level instructions
```
The code uses 37 C-level instructions (32 without compression), including 7 lookups in 8-bit tables.

A.8. Tabulation based hashing for 48-bit keys using 16-bit characters.
```
/* tabulation based hashing for 48-bit key x using 16-bit characters.
   T0, T1 ... T4 are pre-computed tables */
inline INT64 ShortTable48(int64views x,
                          INT64 *T0[], INT64 *T1[], INT64 *T2[],
                          INT64 T3[], INT64 T4[])
{
  INT64 *a0, *a1, *a2, c;

  a0 = T0[x.as_int16s[0]];
  a1 = T1[x.as_int16s[1]];
  a2 = T2[x.as_int16s[2]];

  c = a0[1] + a1[1] + a2[1];
  c = compressShort48(c);   // optional

  return a0[0] ^ a1[0] ^ a2[0] ^ T3[c&262143] ^ T4[c>>18];
} // 22 + 5 = 27 C-level instructions

// optional compression
inline INT32 compressShort48(INT64 i) {
  const INT64 Mask1 = (((INT64)3)<<18) + 3;
  const INT64 Mask2 = (((INT64)65535)<<18) + 65535;
  const INT64 Mask3 = (((INT64)3)<<18) + 3;
  return Mask1 + (i&Mask2) - ((i>>16)&Mask3);
} // 5 C-level instructions
```
The code uses 27 C-level instructions (22 without compression), including 5 lookups in 16-bit tables.

A.9. Tabulation based hashing for 48-bit keys using 8-bit characters.

```
/* tabulation based hashing for 48-bit key x using 8-bit characters.
   T0, T1 ... T10 are pre-computed tables */
inline INT64 CharTable48(int64views x,
                         INT64 *T0[], INT64 *T1[], INT64 *T2[],
                         INT64 *T3[], INT64 *T4[], INT64 *T5[],
                         INT64  T6[], INT64  T7[], INT64  T8[],
                         INT64  T9[], INT64 T10[])
{
  INT64 *a0, *a1, *a2, c;

  a0 = T0[x.as_int8s[0]]; a1 = T1[x.as_int8s[1]];
  a2 = T2[x.as_int8s[2]]; a3 = T2[x.as_int8s[3]];
  a4 = T2[x.as_int8s[4]]; a5 = T2[x.as_int8s[5]];

  c = a0[1] + a1[1] + a2[1] + a3[1] + a4[1] + a5[1];
  c = compressChar48(c);   // optional

  return
    a0[0] ^ a1[0] ^ a2[0] ^ a3[0] ^ a4[0] ^ a5[0] ^
    T6[(c&2047)] ^ T7[((c>>11)&2047)] ^ T8[((c>>22)&2047)] ^
    T9[((c>>33)&2047)] ^ T10[c>>44];
} // 52 + 5 = 57 C-level instructions

// optional compression
inline INT32 compressChar48(INT64 i) {
  const INT64 Mask1 = 5 +   (((INT64)5)<<11) + (((INT64)5)<<22) +
                            (((INT64)5)<<33) + (((INT64)5)<<44);
  const INT64 Mask2 = 255 + (((INT64)255)<<11) + (((INT64)255)<<22) +
                            (((INT64)255)<<33) + (((INT64)255)<<44);
  const INT64 Mask3 = 7 +   (((INT64)7)<<11) + (((INT64)7)<<22) +
                            (((INT64)7)<<33) + (((INT64)7)<<44);
  return Mask1 + (i&Mask2) - ((i>>8)&Mask3);
} // 5 C-level instructions
```
The code uses 57 C-level instructions (52 without compression), including 11 lookups in 8-bit tables.

A.10. Tabulation based hashing for 64-bit keys using 16-bit characters.

```
/* tabulation based hashing for 64-bit key x using 16-bit characters.
   T0, T1 ...  T6 are pre-computed tables */
inline INT64 ShortTable64(int64views x,
                          INT64 *T0[], INT64 *T1[],
                          INT64 *T2[], INT64 *T3[],
                          INT64 T4[], INT64 T5[], INT64 T6[])
{
  INT64 *a0, *a1, *a2, *a3, c;

  a0 = T0[x.as_int16s[0]]; a1 = T1[x.as_int16s[1]];
  a2 = T2[x.as_int16s[2]]; a3 = T3[x.as_int16s[3]];

  c = a0[1] + a1[1] + a2[1] + a3[1];
  c = compressShort64(c);   // optional

  return a0[0] ^ a1[0] ^ a2[0] ^ a3[0] ^ T4[c&2097151] ^
         T5[(c>>21)&2097151] ^ T6[c>>42];
} // 32 + 5 = 37 C-level instructions

// optional compression
inline INT64 compressShort64(INT64 i) {
  const INT64 Mask1 = 4 + (((INT64)4)<<21) + (((INT64)4)<<42);
  const INT64 Mask2 = 65535 + (((INT64)65535)<<21) + (((INT64)65535)<<42);
  const INT64 Mask3 = 31 + (((INT64)31)<<21) + (((INT64)31)<<42);
  return Mask1 + (i&Mask2) - ((i>>16)&Mask3);
} // 5 C-level instructions
```
The code uses 37 C-level instructions (32 without compression), including 7 lookups in 16-bit tables.

A.11. Tabulation based hashing for 64-bit keys using 8-bit characters.
```
/* tabulation based hashing for 64-bit key x using 8-bit characters.
   T0, T1 ... T14 are pre-computed tables */
inline INT64 CharTable64(int64views x,
                         Entry T0[], Entry T1[], Entry T2[], Entry T3[],
                         Entry T4[], Entry T5[], Entry T6[], Entry T7[],
                         INT64 T8[], INT64 T9[], INT64 T10[], INT64 T11[],
                         INT64 T12[], INT64 T13[], INT64 T14[])
{
  Entry *a0, *a1, *a2, *a3, *a4, *a5, *a6, *a7;
  INT64 c0;
  INT32 c1;

  a0 = &T0[x.as_int16s[0]]; a1 = &T1[x.as_int16s[1]];
  a2 = &T2[x.as_int16s[2]]; a3 = &T3[x.as_int16s[3]];
  a4 = &T4[x.as_int16s[4]]; a5 = &T5[x.as_int16s[5]];
  a6 = &T6[x.as_int16s[6]]; a7 = &T7[x.as_int16s[7]];

  c0  = a0->u + a1->u + a2->u + a3->u + a4->u + a5->u + a6->u + a7->u;
  c1  = a0->v + a1->v + a2->v + a3->v + a4->v + a5->v + a6->v + a7->v;

  c0 = compressChar64_0(c0);   // optional
  c1 = compressChar64_1(c1);   // optional

  return
    a0->h ^ a1->h ^ a2->h ^ a3->h ^ a4->h ^ a5->h ^ a6->h ^ a7->h ^
    T8[(c0&2043)] ^ T9[((c0>>11)&2043)] ^ T10[((c0>>22)&2043)] ^
    T11[((c0>>33)&2043)] ^ T12[(c0>>44)] ^ T13[(c1&2043)] ^ T14[(c1>>11)];
} // 85 + 10 = 95 C-level instructions

// optional compression
inline INT64 compressChar64_0(INT64 i) {
  const INT64 Mask1 = 7   + (((INT64)7)<<11) + (((INT64)7)<<22) +
                              (((INT64)7)<<33) + (((INT64)7)<<44);
  const INT64 Mask2 = 255 + (((INT64)255)<<11) + (((INT64)255)<<22) +
                              (((INT64)255)<<33) + (((INT64)255)<<44);
  const INT64 Mask3 = 7   + (((INT64)7)<<11) + (((INT64)7)<<22) +
                              (((INT64)7)<<33) + (((INT64)7)<<44);
  return Mask1 + (i&Mask2) - ((i>>8)&Mask3);
} // 5 C-level instructions

// optional compression
inline INT32 compressChar64_1(INT32 i) {
  const INT64 Mask1 = (((INT64)7)<<11) + 7;
  const INT64 Mask2 = (((INT64)255)<<11) + 255;
  const INT64 Mask3 = (((INT64)7)<<11) + 7;
  return Mask1 + (i&Mask2) - ((i>>8)&Mask3);
} // 5 C-level instructions
```
The code uses 95 C-level instructions (85 without compression), including 15 lookups in 8-bit tables.

A.12. CW trick for 32-bit keys with prime $2^{61}-1$.

```
const INT64 Prime = (((INT64)1)<<61) - 1;

/* Computes ax+b mod Prime, possibly plus 2*Prime, exploiting the structure of Prime. */
inline INT64 MultAddPrime32(INT32 x, INT64 a, INT64 b) {
  INT64 a0,a1,c0,c1,c;
  a0 = LOW(a)*x;
  a1 = HIGH(a)*x;
  c0 = a0+(a1<<32);
  c1 = (a0>>32)+a1;
  c  = (c0&Prime)+(c1>>29)+b;
  return c;
} // 12 C-level instructions

// CWtrick for 32-bit key x (Prime = 2^61-1)
inline INT64 CWtrick32(INT32 x, INT64 A, INT64 B, INT64 C, INT64 D, INT64 E) {
  INT64 h;
  h = MultAddPrime32(x,MultAddPrime32(x,MultAddPrime32(x,MultAddPrime32(x,A,B),C),D),E);
  h = (h&Prime)+(h>>61);
  if (h>=Prime) h-=Prime;
  return h;
} // 12*4 + 5 = 53 C-level instructions
```
The code uses 53 C-level instructions, including 8 multiplications. With 4-independence only, the code uses 41 C-level instructions, including 8 multiplications.

A.13. CW trick for 48-bit keys with prime $2^{61}-1$.

```
/* Computes ax+b mod Prime, possibly plus 2*Prime, exploiting the structure of Prime. */
inline INT64 MultAddPrime(INT64 x, INT64 a, INT64 b) {
  INT64 x0, x1, c0, c1, c, a0, a1, ax00, ax01_10, ax11;

  x0 = LOW(x); x1 = HIGH(x); a0 = LOW(a); a1 = HIGH(a);
  ax00 = a0*x0; ax11 = a1*x1; ax01_10 = a0*x1 + a1*x0;

  c0 = ax00 + (ax01_10<<32);
  c1 = (ax00>>61) + (ax01_10>>29) + (ax11<<3);
  c  = (c0&Prime61) + c1 + b;
  c  = (c&Prime61) + (c>>61);

  return c;
} // 22 C-level instructions

// CWtrick for 48-bit key x (Prime = 2^61-1)
inline INT64 CWtrick48(INT32 x, INT64 A, INT64 B, INT64 C, INT64 D, INT64 E) {
  INT64 h;
  h = MultAddPrime(MultAddPrime(MultAddPrime(MultAddPrime(x,A,B),x,C),x,D),x,E);
  h = (h&Prime)+(h>>61);
  if (h>=Prime) h-=Prime;
  return h;
} // 22*4 + 5 = 93 C-level instructions
```
The code uses 93 C-level instructions, including 16 multiplications. With 4-independence only, the code uses 71 C-level instructions, including 12 multiplications.

A.14. CW trick for 64-bit keys using prime $2^{89}-1$.
```
const INT64 Prime89_0  = (((INT64)1)<<32)-1;
const INT64 Prime89_1  = (((INT64)1)<<32)-1;
const INT64 Prime89_2  = (((INT64)1)<<25)-1;
const INT64 Prime89_21 = (((INT64)1)<<57)-1;

/* Computes (r mod Prime89) mod 2^64,
   exploiting the structure of Prime89 */
inline INT64 Mod64Prime89(INT96 r) {
  INT64 r0, r1, r2;

  // r2r1r0 = r&Prime89 + r>>89
  r2 = r[2];
  r1 = r[1];
  r0 = r[0] + (r2>>25);
  r2 &= Prime89_2;

  return (r2 == Prime89_2 && r1 == Prime89_1 && r0 >= Prime89_0) ?
         (r0 - Prime89_0) : (r0 + (r1<<32));
} // 7 C-level instructions (worst case)

/* Computes a 96-bit r such that r mod Prime89 == (ax+b) mod Prime89
   exploiting the structure of Prime89. */
inline void MultAddPrime89(INT96 r, INT64 x, INT96 a, INT96 b) {
  INT64 x1, x0, c21, c20, c11, c10, c01, c00;
  INT64 d0, d1, d2, d3, s0, s1, carry;

  x1 = HIGH(x);  x0 = LOW(x);

  c21 = a[2]*x1;  c20 = a[2]*x0;
  c11 = a[1]*x1;  c10 = a[1]*x0;
  c01 = a[0]*x1;  c00 = a[0]*x0;

  d0 = (c20>>25)+(c11>>25)+(c10>>57)+(c01>>57);
  d1 = (c21<<7);
  d2 = (c10&Prime89_21) + (c01&Prime89_21);
  d3 = (c20&Prime89_2) + (c11&Prime89_2) + (c21>>57);

  s0 = b[0] + LOW(c00) + LOW(d0) + LOW(d1);
  r[0] = LOW(s0);  carry = HIGH(s0);

  s1 = b[1] + HIGH(c00) + HIGH(d0) + HIGH(d1) + LOW(d2) + carry;
  r[1] = LOW(s1);
  carry = HIGH(s1);

  r[2] = b[2] + HIGH(d2) + d3 + carry;
} // 59 C-level instructions

// CWtrick for 64-bit key x (Prime = 2^89-1)
inline INT64 CWtrick64(INT64 x, INT96 A, INT96 B, INT96 C, INT96 D, INT96 E) {
  INT96 r;
  MultAddPrime89(r,x,A,B); MultAddPrime89(r,x,r,C);
  MultAddPrime89(r,x,r,D); MultAddPrime89(r,x,r,E);
  return Mod64Prime89(r);
} // 59*4 + 7 = 243 C-level instructions
```
The code uses 243 C-level instructions, including 24 multiplications. With 4-independence only, the code uses 184 C-level instructions, including 18 multiplications. Note that `Mod64Prime89` only produces the 64 least significant bits of the answer.

A.15. Second moment estimation.
```
#define NumCounters 32768      // (1<<15)
INT64 Counters[NumCounters];

/* precomputed tables whose hash strings
 *  only use 15 least significant bits */
INT64 *T0, *T1, *T2;

inline void StreamUpdateSecond(INT32 ipaddr, INT32 size) {
  Counters[ShortTable32(ipaddr,T0,T1,T2)]+=size;
} // 3 instructions plus those in ShortTable32.

double StreamEstimateSecond() {
  int i;
  INT64 c;
  double sum = 0, sqsum = 0;
  for (i = 0; i < NumCounters; i++) {
    c = Counters[i];
    sum += c;
    sqsum += c*c;
  }
  return sqsum + (sqsum-sum*sum)/(NumCounters-1);
}
```
The code for updating the second moment counters when the hash function is ShortTable32. It adds 3 instructions to those in ShortTable32, including one additional lookup and update.