Formalization of 3-independence of simple tabulation hashing

Wei De Leong and Yong Kiam Tan and Seng Joe Watt

July 14, 2026

Abstract

Simple tabulation hashing [2] is a computationally-efficient hashing algorithm to get values that behave independently and are uniformly distributed for any 3 distinct keys. This entry formalizes the 3-independence and non-4-independence of simple tabulation hashing, based on the written proofs provided by [1, Solutions 8][4]

Contents

1 Preliminaries **2**
1.1 The XOR operator . . . 2
1.1.1 Definition of XOR . . . 3
1.1.2 Standard Instantiations . . . 4
1.1.3 Syntactic sugar . . . 4
1.2 Intro rules, contrapositives and bijections . . . 9
1.2.1 Application . . . 11

2 Definitions **12**

3 Proofs **14**
3.1 Proof of 1-independence and uniformity . . . 14
3.2 Proof of two-independence . . . 14
3.3 Proof of 3-independence . . . 14
3.4 Proof of 3-universal . . . 15
3.5 Proof of non-4-universal . . . 15

4 Appendix **16**
4.1 Utility . . . 16
4.2 Alternate proof of non-4-universal . . . 16
4.2.1 Preliminaries . . . 16
4.2.2 Proofs . . . 17

theory *Xor*
imports
Main
begin

This theory abstractly defines

- the binary operator *xor* (aliases (XOR) ($\oplus$))
- the n-ary version *xor-fold*
- and the Big version *xor-sum* (alias $\bigoplus i{\in}J.\ f\ i$).

An implementation of xor should abide by the following laws:

- identity element
- commutative
- associative
- left/right neutrality
- self-inverse

which constitutes an abelian group.
Instantiations of abel_group_xor have been provided for bool, nat and int.
Other theories relating to xor:

- *abstract-boolean-algebra-sym-diff*
- *xor*
- (AFP) `Approximate_Model_Counting.RandomXOR` [3]

1 Preliminaries

1.1 The XOR operator

lemma (**in** *monoid-list*) *list-conv-take-drop*:
shows $F\ xs = F\ (take\ i\ xs) * F\ (drop\ i\ xs)$
⟨*proof*⟩

lemma (**in** *monoid-list*) *list-conv-take-nth-drop*:
assumes $i < length\ xs$
shows $F\ xs = F\ (take\ i\ xs) * xs\ !\ i * F\ (drop\ (Suc\ i)\ xs)$
⟨*proof*⟩

lemma (**in** *comm-monoid-list*) *absorb-right*:
  assumes $i <$ *length xs*
  shows $F\ xs * v = F\ (xs[i := xs\ !\ i * v])$
  ⟨*proof*⟩

lemma (**in** *semiring-bit-operations*) *eq-iff*:
  shows *semiring-bit-operations-class.xor* $a\ b = 0 \longleftrightarrow a = b$
  ⟨*proof*⟩

1.1.1 Definition of XOR

class *xor* =
  fixes *xor* :: ‹$'a \Rightarrow 'a \Rightarrow 'a$› (**infixr** ‹*XOR*› *59*) — Definition of 2-ary xor

class *abel-group-xor* = *zero* + *xor* +
  assumes *xor-commute*: a *XOR* $b = b$ *XOR* a
  assumes *xor-assoc*: $(a$ *XOR* $b)$ *XOR* $c = a$ *XOR* b *XOR* c
  assumes *xor-right-neutral*: a *XOR* $0 = a$
  assumes *eq-iff* [*iff*]: a *XOR* $b = 0 \longleftrightarrow a = b$
begin

lemma *self-inv* [*simp*]:
  shows a *XOR* $a = 0$
  ⟨*proof*⟩

The above properties show that XOR forms an abelian group

sublocale *group xor 0 id*
  ⟨*proof*⟩

sublocale *abel-semigroup xor* ⟨*proof*⟩

sublocale *comm-monoid xor 0* ⟨*proof*⟩

sublocale *xor-sum*: *comm-monoid-set xor 0*
  defines *xor-sum* = *xor-sum.F* ⟨*proof*⟩

abbreviation *Xor-Sum* ≡ *xor-sum* $(\lambda x.\ x)$

sublocale *xor-list*: *monoid-list xor 0*
  defines *xor-fold* = *xor-list.F* ⟨*proof*⟩

sublocale *xor-comm-list*: *comm-monoid-list xor 0*
  rewrites *monoid-list.F xor 0* = *xor-fold*
⟨*proof*⟩

sublocale *xor-list*: *comm-monoid-list-set xor 0*
  rewrites *monoid-list.F xor 0* = *xor-fold* **and** *comm-monoid-set.F xor 0* = *xor-sum*
⟨*proof*⟩

lemma *xor-cong*:
  assumes *a XOR b = 0*
  shows *a = b*
  ⟨*proof*⟩

lemma *inj-on-UNIV*:
  shows *inj-on xor UNIV*
  ⟨*proof*⟩

lemma *map-indices-conv-list-update-conv*:
  shows *map* (*λj. if j = i then g j else xs ! j*) [*0..<length xs*] = *xs*[*i := g i*]
  ⟨*proof*⟩

lemma *fold-absorb*:
  assumes *i < length xs*
  shows *xor-fold xs XOR v =*
    xor-fold (*map* (*λj. if j = i then xs ! j XOR v else xs ! j*) [*0..<length xs*])
  ⟨*proof*⟩
end

hide-fact *xor-commute xor-assoc xor-right-neutral*

1.1.2 Standard Instantiations

instantiation *bool* :: *abel-group-xor* **begin**
definition *zero-bool ≡ False*
definition *xor-bool* :: *bool ⇒ bool ⇒ bool* **where** *xor-bool a b ≡* (*a ≠ b*)
instance ⟨*proof*⟩
end

instantiation *nat* **and** *int* :: *abel-group-xor* **begin**

definition *xor-nat* :: *nat ⇒ nat ⇒ nat* **where** *xor-nat ≡ semiring-bit-operations-class.xor*
definition *xor-int* :: *int ⇒ int ⇒ int* **where** *xor-int ≡ semiring-bit-operations-class.xor*
instance ⟨*proof*⟩
end

1.1.3 Syntactic sugar

open-bundle *xor-syntax* **begin**

notation *xor* (**infixr** ‹⊕› *59*)
notation *Xor-Sum* (‹⨁›)

Now: lots of fancy syntax. First, *xor-sum* (*λx. e*) *A* is written ⨁*x∈A. e*.

syntax (*ASCII*)
  -xor-sum :: *pttrn ⇒ 'a set ⇒ 'b ⇒ 'b::abel-group-xor* (‹(‹*indent=3 notation=*‹*binder XORSUM*››*XORSUM* (-/:-)./ -)› [*0*, *51*, *10*] *10*)
syntax

-xor-sum :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b::abel-group-xor* (‹(‹indent=2 notation=‹binder ⨁››⨁ (-/∈-)./ -)› [0, 51, 10] 10)

syntax-consts
-xor-sum ⇌ *xor-sum*

translations — Beware of argument permutation!
⨁ *i*∈*A*. *b* ⇌ *CONST xor-sum* (λ*i*. *b*) *A*
end

unbundle *no xor-syntax*
end
theory *Dependent-Product*
imports *HOL−Probability.Probability*
begin

The following lemma was reproduced from [3, Lemma *measure_pmf_prob_dependent_product_bound_eq*] The AFP imports `Monad_Normalisation.Monad_Normalisation` which alters some Isabelle syntax, which we want to avoid by copy pasting the lemmas here

lemma *measure-pmf-prob-dependent-product-bound-eq*:
assumes *countable A* ⋀*i*. *countable* (*B i*)
assumes ⋀*a*. *a* ∈ *A* ⟹ *measure-pmf.prob N* (*B a*) = *r*
shows *measure-pmf.prob* (*pair-pmf M N*) (*Sigma A B*) = *measure-pmf.prob M A* ∗ *r*
(**is** *?L* = *?R*)
⟨*proof*⟩

The following lemma was reproduced from [3, Lemma *measure_pmf_prob_dependent_product_bound_eq'*] The AFP imports `Monad_Normalisation.Monad_Normalisation` which alters some Isabelle syntax, which we want to avoid by copy pasting the lemmas here

lemma *measure-pmf-prob-dependent-product-bound-eq'*:
— N is the pmf we want to fix depending on values from M
assumes ⋀*a*. *a* ∈ *A* ∩ *set-pmf M* ⟹ *measure-pmf.prob N* (*B a*) = *r*
shows *measure-pmf.prob* (*pair-pmf M N*) (*Sigma A B*) = *measure-pmf.prob M A* ∗ *r*
(**is** *?L* = *?R*)
⟨*proof*⟩

end
theory *Xor-Inst*
imports
Xor
Fixed-Length-Vector.Fixed-Length-Vector
begin

More instances for *abel-group-xor*

unbundle *Fixed-Length-Vector.vec-syntax*

Sequences containing a base type that can be xor'd can itself also be xor'd, via element-wise xor

instantiation *vec* :: (*abel-group-xor*,*index1*) *abel-group-xor* **begin**

XORing two vectors is defined as element-wise XOR

definition *xor-vec* :: $('a,\ 'b)$ *vec* ⇒
$('a,\ 'b)$ *vec* ⇒
$('a,\ 'b)$ *vec* **where**
xor-vec a b ≡ *map-vec* $(\lambda(a',b').\ a'\ XOR\ b')$ (*zip-vec a b*)

The identity vector is a vector where every element is zero

definition *zero-vec* ≡ *replicate-vec 0* :: $('a,'b)$ *vec*

instance ⟨*proof*⟩
end

lemma *nth-conv* [*simp*]:
shows $(a\ XOR\ b)\ \$\ i = a\ \$\ i\ XOR\ b\ \$\ i$
⟨*proof*⟩

unbundle *no Fixed-Length-Vector.vec-syntax*
end
theory *Vec-Extras*
imports
HOL−Analysis.Finite-Cartesian-Product
Fixed-Length-Vector.Fixed-Length-Vector
begin

unbundle *no Finite-Cartesian-Product.vec-syntax*
unbundle *Fixed-Length-Vector.vec-syntax*

setup-lifting *type-definition-vec*

The most important lemmas to know when working with *Fixed-Length-Vector.vec* are

- *vec-of-list-inject*: $[\![x \in \{xs.\ length\ xs = CARD('b)\};\ y \in \{xs.\ length\ xs = CARD('b)\}]\!] \Longrightarrow (vec\text{-}of\text{-}list\ x = vec\text{-}of\text{-}list\ y) = (x = y)$
- *vec-of-list-inverse*: $y \in \{xs.\ length\ xs = CARD('b)\} \Longrightarrow list\text{-}of\text{-}vec\ (vec\text{-}of\text{-}list\ y) = y$
- *nth-vec.rep-eq*: $(\$)\ x = (!)\ (list\text{-}of\text{-}vec\ x) \circ from\text{-}index$

Tip: When using *vec-of-list* to create a vector v from xs, it is essential to separately prove that $xs \in \{xs.\ length\ xs = CARD('b)\}$ for above lemmas to apply

lift-definition *vec-of-fcp* :: ('*a*, '*b*::{*finite*, *index1*}) *Finite-Cartesian-Product.vec* ⇒
('*a*, '*b*) *Fixed-Length-Vector.vec* **is**
‹λ*x*. *map* (λ*i*. *Finite-Cartesian-Product.vec-nth* *x* *i* :: '*a*) (*indexes* :: '*b* *list*)›
⟨*proof*⟩

definition *fcp-of-vec* :: ('*a*, '*b*::{*finite*, *index1*}) *Fixed-Length-Vector.vec* ⇒
('*a*, '*b*) *Finite-Cartesian-Product.vec* **where**
‹*fcp-of-vec* ≡ λ*x*. *Finite-Cartesian-Product.vec-lambda* (λ*i*. *x* $ *i*)›
lift-definition *enumerate-vec* :: *nat* ⇒ ('*a*, '*b*) *Fixed-Length-Vector.vec* ⇒
(*nat* × '*a*, '*b*) *Fixed-Length-Vector.vec* **is**
‹*List.indexed-from*› ⟨*proof*⟩

lemma *infinite-UNIV-vec*:
assumes *infinite* (*UNIV* :: '*a* *set*) *finite* (*UNIV* :: '*b* *set*)
shows *infinite* (*UNIV* :: ('*a*,'*b*) *Fixed-Length-Vector.vec* *set*)
⟨*proof*⟩

lemma *infinite-index-UNIV-vec*:
assumes *infinite* (*UNIV* :: '*b* *set*)
shows (*UNIV* :: ('*a*,'*b*) *Fixed-Length-Vector.vec* *set*) = {*vec-of-list* []}
⟨*proof*⟩

lemma *card-UNIV-vec*:
shows *CARD*(('*a*, '*b*) *Fixed-Length-Vector.vec*) = *CARD*('*a*) ^ *CARD*('*b*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma (**in** *index1*) *from-index-neq* [*simp*]:
assumes $a \neq b$
shows *from-index* *a* ≠ *from-index* *b*
⟨*proof*⟩

lemma *list-of-vec-not-empty* [*simp*]:
fixes *xs* :: ('*a*, '*b* :: *index1*) *Fixed-Length-Vector.vec*
shows *list-of-vec* *xs* ≠ []
⟨*proof*⟩

lemma *id-take-nth-drop-vec*:
fixes *xs* :: ('*a*, '*b* :: *index1*) *Fixed-Length-Vector.vec*
assumes *i* < *CARD*('*b*)
shows
xs = *vec-of-list* (
take *i* (*list-of-vec* *xs*) @
list-of-vec *xs* ! *i* #
drop (*Suc* *i*) (*list-of-vec* *xs*))
⟨*proof*⟩

lemma *id-take-nth-drop-vec'*:
fixes *xs* :: ('*a*, '*b* :: *index1*) *Fixed-Length-Vector.vec*
assumes $i < CARD('b)$
shows
xs = *vec-of-list* (
take i (*list-of-vec xs*) @
xs $ *to-index i* #
drop (*Suc i*) (*list-of-vec xs*))
⟨*proof*⟩

lemma *nth-not-mem-enumerate*:
assumes $y\ !\ i \neq x\ !\ i$
shows $(i,\ y\ !\ i) \notin$ *set* (*List.indexed-from 0 x*)
⟨*proof*⟩

lemma *nth-not-mem-enumerate-take*:
assumes $y\ !\ j \neq x\ !\ j$
shows $(j,\ y\ !\ j) \notin$ *set* (*List.indexed-from 0* (*take i x*))
⟨*proof*⟩

lemma *nth-not-mem-enumerate-drop*:
assumes $y\ !\ j \neq x\ !\ j$
shows $(j,\ y\ !\ j) \notin$ *set* (*List.indexed-from* (*Suc i*) (*drop* (*Suc i*) *x*))
⟨*proof*⟩

lemma *nth-enumerate-eq-vec*:
fixes *xs* :: ('*a*, '*b* :: *index1*) *Fixed-Length-Vector.vec*
assumes $m < CARD('b)$
shows *enumerate-vec n xs* $ *to-index m* = (*n* + *m*, *xs* $ *to-index m*)
⟨*proof*⟩

lemma *in-set-enumerate-eq-vec*:
fixes *xs* :: ('*a*, '*b* :: *index1*) *Fixed-Length-Vector.vec*
shows $p \in$ *set-vec* (*enumerate-vec n xs*) $\longleftrightarrow$
$n \leq$ *fst p* $\wedge$ *fst p* $< CARD('b) + n \wedge$ *xs* $ *to-index* (*fst p* − *n*) = *snd p*
⟨*proof*⟩

lemma *inj-vec-of-fcp*:
shows *inj vec-of-fcp*
⟨*proof*⟩

lemma *inj-fcp-of-vec*:
shows *inj fcp-of-vec*
⟨*proof*⟩

lemma *fcp-of-vec-inverse*:

shows *vec-of-fcp* (*fcp-of-vec* *x*) = *x*
⟨*proof*⟩

lemma *vec-of-fcp-inverse*:
shows *fcp-of-vec* (*vec-of-fcp* *x*) = *x*
⟨*proof*⟩

lemma *surj-vec-of-fcp*:
shows *surj vec-of-fcp*
⟨*proof*⟩

lemma *surj-fcp-of-vec*:
shows *surj fcp-of-vec*
⟨*proof*⟩

lemma *bij-vec-of-fcp*:
shows *bij vec-of-fcp*
⟨*proof*⟩

lemma *bij-fcp-of-vec*:
shows *bij fcp-of-vec*
⟨*proof*⟩

unbundle *no Fixed-Length-Vector.vec-syntax*
end
theory *Simple-Tabulation-Hashing*
imports
Universal-Hash-Families.Universal-Hash-Families-More-Product-PMF
Dependent-Product
Xor-Inst
Vec-Extras
begin

unbundle *Xor.xor-syntax*
unbundle *Fixed-Length-Vector.vec-syntax*
unbundle *no Finite-Cartesian-Product.vec-syntax*
hide-type *Finite-Cartesian-Product.vec*

1.2 Intro rules, contrapositives and bijections

lemma (**in** *prob-space*) *k-wise-indep-varsI*:
assumes $\bigwedge a\ J.\ [\![\ J \subseteq I;\ card\ J \leq k;\ finite\ J\ ]\!] \Longrightarrow$
$prob\ \{\omega.\ \forall\, i{\in}J.\ X\ i\ \omega = a\ i\} = (\prod i{\in}J.\ prob\ \{\omega.\ X\ i\ \omega = a\ i\})$
$M = measure\text{-}pmf\ p$
shows *k-wise-indep-vars* *k* (λ-. *count-space UNIV*) *X I*
⟨*proof*⟩

lemma (**in** *prob-space*) *indep-vars-pmf-contrapos*:

assumes *prob* $\{\omega.\ \forall x{\in}J.\ P\ x\ (X'\ x\ \omega)\} \neq (\prod x{\in}J.\ prob\ \{\omega.\ P\ x\ (X'\ x\ \omega)\})$
finite J $M = measure\text{-}pmf\ p$
shows $\neg$ *indep-vars* (λ-. *count-space UNIV*) $X'\ J$
⟨*proof*⟩

locale *bij-betw-funcsetE* **begin**

These lemmas prove bijections between sets of 2-arity functions and their decompositions

For example, a function $f : [0, 100] \rightharpoonup \{A, B, C\} \rightharpoonup \{True, False\}$ can be decomposed into 3 parts:

$$
\begin{aligned}
&[0, 100] \rightharpoonup \{A, B, C\} \rightharpoonup \{True, False\} \\
\rightleftharpoons\ &[1, 100] \rightharpoonup \{A, B, C\} \rightharpoonup \{True, False\}\times \\
&\quad \{A, C\} \rightharpoonup \{True, False\}\times \\
&\qquad \{True, False\}
\end{aligned}
$$

- base: $[1, 100] \rightharpoonup \{A, B, C\} \rightharpoonup \{True, False\}$ (split the 0-index, leaving 1-100 intact)
- z_0: $\{A, C\} \rightharpoonup \{True, False\}$ (represents the 0-index, but further split the B-index)
- z_b: $B \in \{True, False\}$ (decides the value for $f\ 0\ B$)

f is equivalent to its reconstruction, $f = base(0 := z_0(B := z_b))$

context
fixes α **and** $A\ B$ :: - *set*
assumes α-*in*-A: $\alpha \in A$
begin

lemma *bij-PiE-remove-point*:
bij-betw $(\lambda\ (f,\ v).\ f(\alpha := v))\ ((A - \{\alpha\} \rightarrow_E B) \times B)\ (A \rightarrow_E B)$
⟨*proof*⟩

Helper: "Absorb" a point into a function space bijection. This allows chaining: if we have a bijection F for a smaller domain $S \rightarrow (A - \{x\} \rightarrow B)$, we can automatically get a bijection for $S\ x\ B \rightarrow (A \rightarrow B)$.

lemma *bij-absorb-point*:
assumes *bij-betw* $F\ S\ (A - \{\alpha\} \rightarrow_E B)$
shows *bij-betw* $(\lambda\ (s,\ v).\ (F\ s)(\alpha := v))\ (S \times B)\ (A \rightarrow_E B)$
⟨*proof*⟩

Helper 2: "Lift Value" Attaches a computed value V to the function at point x. Used for the outer function (e.g., $a(\alpha := V)$).

lemma *bij-lift-value*:
assumes *bij-betw* $V\ S\ B$

shows *bij-betw* $(\lambda(f,\ s).\ f(\alpha := V\ s))\ ((A - \{\alpha\} \rightarrow_E B) \times S)\ (A \rightarrow_E B)$
⟨*proof*⟩

end

Tuple Association Helper: $((A, B), C) \longleftrightarrow (A, B, C)$

lemma *bij-assoc-right-to-left*:
bij-betw $(\lambda\ (a,\ b,\ c).\ ((a,\ b),\ c))\ (A \times B \times C)\ ((A \times B) \times C)$
⟨*proof*⟩

lemma *bij-assoc-left-to-right*:
bij-betw $(\lambda((a,\ b),\ c).\ (a,\ b,\ c))\ ((A \times B) \times C)\ (A \times B \times C)$
⟨*proof*⟩

lemma *bij-betw-combine-right*:
assumes *bij-betw f B C*
shows *bij-betw* $(\lambda(a,\ b).\ (a,\ f\ b))\ (A \times B)\ (A \times C)$
⟨*proof*⟩

1.2.1 Application

lemma *remove1-remove1*:
assumes $\alpha \in A$ $\beta \in B$ *finite A finite B finite C* $C \neq \{\}$
shows *bij-betw*
$(\lambda(a,b,c).\ a(\alpha := b(\beta := c)))$
$((A - \{\alpha\} \rightarrow_E B \rightarrow_E C) \times (B - \{\beta\} \rightarrow_E C) \times C)$
$(A \rightarrow_E B \rightarrow_E C)$
⟨*proof*⟩

lemma *remove1-remove2*:
assumes $\beta 1 \neq \beta 2$ $\alpha \in A$ $\beta 1 \in B$ $\beta 2 \in B$
finite A finite B finite C $C \neq \{\}$
shows *bij-betw*
$(\lambda(a,b,c,d).\ a(\alpha := b(\beta 1 := c,\ \beta 2 := d)))$
$((A - \{\alpha\} \rightarrow_E B \rightarrow_E C) \times (B - \{\beta 1,\beta 2\} \rightarrow_E C) \times C \times C)$
$(A \rightarrow_E B \rightarrow_E C)$
⟨*proof*⟩

lemma *remove1-remove3*:
assumes $\beta 1 \neq \beta 2$ $\beta 2 \neq \beta 3$ $\beta 3 \neq \beta 1$ $\alpha \in A$ $\beta 1 \in B$ $\beta 2 \in B$ $\beta 3 \in B$
finite A finite B finite C $C \neq \{\}$
shows *bij-betw*
$(\lambda(a,b,c,d,e).\ a(\alpha := b(\beta 1 := c,\ \beta 2 := d,\ \beta 3 := e)))$
$((A - \{\alpha\} \rightarrow_E B \rightarrow_E C) \times (B - \{\beta 1,\beta 2,\beta 3\} \rightarrow_E C) \times C \times C \times C)$
$(A \rightarrow_E B \rightarrow_E C)$
⟨*proof*⟩

lemma *remove2-remove2'remove1*:
assumes $\alpha 1 \neq \alpha 2$ $\beta 1 \neq \beta 2$ $\alpha 1 \in A$ $\alpha 2 \in A$ $\beta 1 \in B$ $\beta 2 \in B$ $\beta 3 \in B$

finite A finite B finite C $C \neq \{\}$
shows *bij-betw*
$(\lambda(a,b,c,d,e,f).\ a(\alpha 1 := b(\beta 1 := d,\ \beta 2 := e),\ \alpha 2 := c(\beta 3 := f)))$
$((A - \{\alpha 1,\ \alpha 2\} \rightarrow_E B \rightarrow_E C) \times (B - \{\beta 1,\ \beta 2\} \rightarrow_E C) \times (B - \{\beta 3\} \rightarrow_E C) \times C \times C \times C)$
$(A \rightarrow_E B \rightarrow_E C)$
⟨*proof*⟩

end

2 Definitions

locale *simple-tabulation-hashing* =
fixes n :: *'n* :: {*index1*, *zero*} *itself* — key length, i.e. 4 for 4 'fragments
fixes r :: *'result* :: {*abel-group-xor*, *finite*} *itself* — bool or any type that supports xor
fixes q :: *'q* :: *index1 itself* — digest length, i.e. 3
fixes d :: *'fragment* :: *finite itself* — domain of input key fragments
assumes *n*: $n = TYPE('n)$
begin

In tabulation hashing, keys are split into n fragments, each piece hashed individually, before being combined into a final digest with the XOR operator. i.e. $x \equiv [x_0, x_1, x_2]$, $h(x) \equiv h_0 x_0 \oplus h_1 x_1 \oplus h_2 x_2$
Therefore, the type variable 'n, CARD('n) and n must reflect the number of parts in a key. In the example above, 'n = 3 (which is an alias for num1 bit1)
The final digest length is indicated by the type variable 'q, and 'result is any type that supports xor. For example, 'result = bool. When using 'result = bool and 'q = 3, it can be interpreted as using a 3-bit word as the output, which can encode $2^3 = 8$ possibilities, e.g. [True, False, True]

abbreviation *cardn* :: *'n itself* ⇒ *nat* **where** *cardn* - ≡ *CARD('n)*

abbreviation *N* :: *'n itself* ⇒ *nat set* **where** *N type* ≡ {..<*cardn type*}

abbreviation *D* :: *'fragment set* **where** *D* ≡ *UNIV*

abbreviation *Dn* :: (*'fragment*, *'n*) *vec set* (‹D^n›) **where**
— domain of all possible input keys s.t. each key is composed of exactly *n* fragments
$D^n \equiv UNIV$

abbreviation *R* :: (*'result*, *'q*) *vec set* **where**
— range of all possible outputs, s.t. each output is composed of exactly *q* bits
R ≡ *UNIV*

abbreviation *H* :: *'n itself* ⇒ (*nat* ⇒ *'fragment* ⇒ (*'result*, *'q*) *vec*) *set* **where**

— set of all possible assignments and combinations for every column-item pair
$H\ type \equiv N\ type \rightarrow_E D \rightarrow_E R$

definition $tb\text{-}S$ :: $'n\ itself \Rightarrow (nat \Rightarrow 'fragment \Rightarrow ('result, 'q)\ vec)\ pmf$ **where**
— Generator of instances of H
$tb\text{-}S\ \text{-} \equiv pmf\text{-}of\text{-}set\ (H\ n)$

definition $tb\text{-}H$ :: $('fragment, 'n)\ vec \Rightarrow (nat \Rightarrow 'fragment \Rightarrow ('result, 'q)\ vec) \Rightarrow ('result, 'q)\ vec$ **where**
— Applies the tabulation hashing algorithm on *k* using a hash function from H
$tb\text{-}H\ k\ h \equiv xor\text{-}fold\ (map\ (\lambda i.\ h\ i\ (k\ \$\ to\text{-}index\ i))\ ([0..<cardn\ n]))$

lemma $tb\text{-}S\text{-}alt\text{-}def$:
shows $tb\text{-}S\ n = prod\text{-}pmf\ (N\ n)\ (\lambda\text{-}.\ prod\text{-}pmf\ UNIV\ (\lambda\text{-}.\ pmf\text{-}of\text{-}set\ R))$
$\langle proof \rangle$

lemma $tb\text{-}H\text{-}absorb$:
assumes $i < CARD('n)$
shows $tb\text{-}H\ x\ h \oplus c = tb\text{-}H\ x\ (h(i := (h\ i)(x\ \$\ to\text{-}index\ i := h\ i\ (x\ \$\ to\text{-}index\ i) \oplus c)))$
$\langle proof \rangle$

lemma $tb\text{-}H\text{-}absorb'$:
shows $tb\text{-}H\ x\ h \oplus c =$
$tb\text{-}H\ x\ (h(from\text{-}index\ i := (h\ (from\text{-}index\ i))(x\ \$\ i := h\ (from\text{-}index\ i)\ (x\ \$\ i) \oplus c)))$
$\langle proof \rangle$

lemma $tb\text{-}H\text{-}extract$:
assumes $i < CARD('n)$
shows $tb\text{-}H\ x\ (h(i := g(x\ \$\ to\text{-}index\ i := \alpha))) = tb\text{-}H\ x\ (h(i := g(x\ \$\ to\text{-}index\ i := 0))) \oplus \alpha$
$\langle proof \rangle$

lemma $tb\text{-}H\text{-}extract'$:
shows $tb\text{-}H\ x\ (h(from\text{-}index\ i := g(x\ \$\ i := \alpha))) = tb\text{-}H\ x\ (h(from\text{-}index\ i := g(x\ \$\ i := 0))) \oplus \alpha$
$\langle proof \rangle$

lemma $tb\text{-}H\text{-}exch$:
assumes $i < CARD('n)$
shows $tb\text{-}H\ x\ (a(i := aa(x\ \$\ to\text{-}index\ i := b))) = \alpha \longleftrightarrow$
$b = tb\text{-}H\ x\ (a(i := aa(x\ \$\ to\text{-}index\ i := \alpha)))$
$\langle proof \rangle$

lemma $tb\text{-}H\text{-}exch'$:
shows $tb\text{-}H\ x\ (a(from\text{-}index\ i := aa(x\ \$\ i := b))) = \alpha \longleftrightarrow$
$b = tb\text{-}H\ x\ (a(from\text{-}index\ i := aa(x\ \$\ i := \alpha)))$
$\langle proof \rangle$

lemma *tb-H-discard*:
  assumes $x \ \$ \ i \neq y \ \$ \ i$
  shows *tb-H* x (a(*from-index* i := b(y \$ i := c))) = *tb-H* x (a(*from-index* i := b))
  ⟨*proof*⟩

3 Proofs

3.1 Proof of 1-independence and uniformity

lemma *prob-tb-H-bin*:
  shows *measure-pmf.prob* (*tb-S* n) $\{h.$ *tb-H* $x\ h = \alpha\}$ = *1* / *real* (*card* R) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

theorem *uniform*:
  shows *prob-space.uniform-on* (*tb-S* n) (*tb-H* *key*) R
  ⟨*proof*⟩

3.2 Proof of two-independence

lemma *prob-tb-H-bin1-bin2*:
  assumes $x \neq y$
  shows *measure-pmf.prob* (*tb-S* n) $\{h.$ *tb-H* $x\ h = \alpha \wedge$ *tb-H* $y\ h = \beta\}$ = *1* / *real* (*card* R $*$ *card* R)
  (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

3.3 Proof of 3-independence

lemma *prob-3same-conv*:
  assumes $x \ \$ \ i \neq y \ \$ \ i$ $y \ \$ \ i \neq z \ \$ \ i$ $z \ \$ \ i \neq x \ \$ \ i$
  shows *measure-pmf.prob* (*tb-S* n) $\{h.$ *tb-H* $x\ h = \alpha \wedge$ *tb-H* $y\ h = \beta \wedge$ *tb-H* $z\ h = \gamma\}$ =
    1 / (*real* (*card* R) $*$ *real* (*card* R) $*$ *real* (*card* R)) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *prob-2same-conv*:
  assumes $x \ \$ \ i \neq y \ \$ \ i$ $z \ \$ \ j \neq x \ \$ \ j$ $i \neq j$
    and $z \ \$ \ i = x \ \$ \ i$
  shows *measure-pmf.prob* (*tb-S* n) $\{h.$ *tb-H* $x\ h = \alpha \wedge$ *tb-H* $y\ h = \beta \wedge$ *tb-H* $z\ h = \gamma\}$ =
    1 / (*real* (*card* R) $*$ *real* (*card* R) $*$ *real* (*card* R)) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *prob-tb-H-bin1-bin2-bin3*:
  assumes $x \neq y$ $y \neq z$ $z \neq x$
  shows *measure-pmf.prob* (*tb-S* n) $\{h.$ *tb-H* $x\ h = \alpha \wedge$ *tb-H* $y\ h = \beta \wedge$ *tb-H* $z\ h = \gamma\}$ =

$1 / (real\ (card\ R) * real\ (card\ R) * real\ (card\ R))$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

3.4 Proof of 3-universal

lemma *three-indep-vars*:
shows *prob-space.k-wise-indep-vars* (*tb-S n*) *3* (λ-. *count-space R*) *tb-H* D^n
⟨*proof*⟩

theorem *three-universal*:
shows *prob-space.k-universal* (*tb-S n*) *3 tb-H* D^n *R*
⟨*proof*⟩

3.5 Proof of non-4-universal

lemma *prob-4-conv*:
fixes *u v* :: *'fragment*
assumes $W \oplus X \oplus Y \oplus Z = 0$ $\bigwedge h.\ P\ (h(0 := undefined,\ Suc\ 0 := undefined)) = P\ h$
$CARD('n) > Suc\ 0$ $u \neq v$
shows *measure-pmf.prob* (*tb-S n*) {*h*::*nat* ⇒ *'fragment* ⇒ (*'result*, *'q*) *vec*.
$h\ 0\ u \oplus h\ (Suc\ 0)\ u \oplus P\ h = W\ \wedge$
$h\ 0\ u \oplus h\ (Suc\ 0)\ v \oplus P\ h = X\ \wedge$
$h\ 0\ v \oplus h\ (Suc\ 0)\ u \oplus P\ h = Y\ \wedge$
$h\ 0\ v \oplus h\ (Suc\ 0)\ v \oplus P\ h = Z$ }
$= 1 / (real\ (card\ R) * real\ (card\ R) * real\ (card\ R))$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *not-four-indep*:
assumes $CARD('n) > 1$ — if n = 1, then tabulation hashing is 4-independent
$card\ D \geq 2$ — if D = or D = x, then it is impossible to obtain 4 distinct keys
$card\ R > 1$ — tabulation hashing is 4-independent otherwise
shows ¬ *prob-space.k-wise-indep-vars* (*tb-S n*) *4* (λ-. *count-space R*) *tb-H* D^n
⟨*proof*⟩

theorem *not-four-universal*:
assumes $CARD('n) > 1$ — if n = 1, then tabulation hashing is 4-independent
$card\ D \geq 2$ — if D = or D = x, then it is impossible to obtain 4 distinct keys
$card\ R > 1$ — tabulation hashing is 4-independent otherwise
shows ¬ *prob-space.k-universal* (*tb-S n*) *4 tb-H* D^n *R*
⟨*proof*⟩

end

4 Appendix

4.1 Utility

context *prob-space*
begin

context
  fixes $K\ D\ k\ p$
  assumes *assms'*: $K \subseteq D$ *card* $K \leq k$ *finite* K $M =$ *measure-pmf* p
begin

lemma *k-wise-indep-vars-probD*:
  assumes *k-wise-indep-vars* k (λ-. *count-space UNIV*) $X\ D$
  shows *prob* $\{\omega.\ \forall x \in K.\ P\ x\ (X\ x\ \omega)\} = (\prod x \in K.$ *prob* $\{\omega.\ P\ x\ (X\ x\ \omega)\})$
  ⟨*proof*⟩

lemma
  assumes *k-universal* $k\ X\ D\ R$
  shows
    k-universal-probD:
      $\bigwedge P.$ *prob* $\{\omega.\ \forall x \in K.\ P\ x\ (X\ x\ \omega)\} = (\prod x \in K.$ *prob* $\{\omega.\ P\ x\ (X\ x\ \omega)\})$
      (**is** *PROP ?thesis-0*) **and**
    k-universal-probD':
      prob $\{\omega.\ \forall x \in K.\ X\ x\ \omega = P\ x\} = (\prod x \in K.$ *indicat-real* $R\ (P\ x)$ / *real* (*card*
R))
      (**is** *?thesis-1*)
⟨*proof*⟩

end
end

locale *simple-tabulation-hashing'* = *simple-tabulation-hashing* +
  assumes r: $r =$ *TYPE*('*result*::{*finite*,*abel-group-xor*})
  assumes q: $q =$ *TYPE*('*q*::*index1*)
  assumes d: $d =$ *TYPE*('*fragment*::*finite*)
begin

end

4.2 Alternate proof of non-4-universal

4.2.1 Preliminaries

lemma (**in** *prob-space*) *finite-if-k-universal*:
  assumes *k-universal* $k\ X\ D\ R\ D \neq \{\}$
  shows *finite* R

$\langle proof \rangle$

lemma *xor-sum-uniform*:
fixes α :: $'a$ :: $\{finite,abel\text{-}group\text{-}xor\}$
defines $R \equiv UNIV$
assumes [*simp*]: *finite J* $J \neq \{\}$
shows *measure-pmf.prob* (*pmf-of-set* ($J \rightarrow_E R$)) $\{f.\ (\bigoplus x{\in}J.\ f\ x) = \alpha\} = 1\ /$ *real* $CARD('a)$
(**is** *?lhs* = *?rhs*)
$\langle proof \rangle$

lemma *k-universal-conv-pmf-of-set*:
defines $R \equiv UNIV$
assumes *prob-space.k-universal* (*measure-pmf p*) *k X D R*
and $J \subseteq D$ *card* $J \leq k$ *finite J* $D \neq \{\}$
shows *map-pmf* ($\lambda\omega.\ \lambda x{\in}J.\ X\ x\ \omega$) p = *pmf-of-set* ($J \rightarrow_E R$)
$\langle proof \rangle$

lemma *k-universal-imp-xor-sum-uniform*:
fixes α :: $'a$ :: $\{finite,abel\text{-}group\text{-}xor\}$
defines $R \equiv UNIV$
assumes *prob-space.k-universal* (*measure-pmf p*) *k X D R*
and $J \subseteq D$ *card* $J \leq k$ *finite J* $J \neq \{\}$
shows *measure-pmf.prob* p $\{\omega.\ (\bigoplus x{\in}J.\ X\ x\ \omega) = \alpha\} = 1\ /$ *real* (*card R*)
(**is** *?lhs* = *?rhs*)
$\langle proof \rangle$

4.2.2 Proofs

context *simple-tabulation-hashing* **begin**

lemma *k-wise-indep-vars-imp-xor-sum-uniform*:
assumes *prob-space.k-wise-indep-vars* (*tb-S n*) *k* (λ-. *count-space R*) *tb-H* D^n
$J \neq \{\}$ *card* $J \leq k$
shows *measure-pmf.prob* (*tb-S n*) $\{h.\ (\bigoplus x{\in}J.\ tb\text{-}H\ x\ h) = \alpha\} = 1\ /$ *real* (*card R*)
$\langle proof \rangle$

lemma *not-k-wise-indep-vars-by-xor-sum*:
assumes *measure-pmf.prob* (*tb-S n*) $\{h.\ (\bigoplus x{\in}J.\ tb\text{-}H\ x\ h) = \alpha\} \neq 1\ /$ *real* (*card R*)
$J \neq \{\}$ *card* $J \leq k$
shows $\neg$ *prob-space.k-wise-indep-vars* (*tb-S n*) *k* (λ-. *count-space R*) *tb-H* D^n
$\langle proof \rangle$

lemma *not-four-indep'*:

assumes $CARD('n) > 1$ — if n = 1, then tabulation hashing is 4-independent
$card\ D \geq 2$ — if D = or D = x, then it is impossible to obtain 4 distinct keys
$card\ R > 1$ — tabulation hashing is 4-independent otherwise
shows $\neg$ *prob-space.k-wise-indep-vars* (*tb-S n*) *4* (λ-. *count-space R*) *tb-H* D^n
⟨*proof*⟩
end
end

References

[1] D. Mareek. NTIN066 Data Structures 1. https://ufal.mff.cuni.cz/courses/ntin066, Apr 2023. Solutions 8 is relevant for simple tabulation hashing.

[2] M. Patrascu and M. Thorup. The power of simple tabulation hashing. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, pages 1–10, New York, NY, USA, 2011. Association for Computing Machinery. numpages: 10, keywords: tabulation hashing, minwise independence, linear probing, independence, cuckoo hashing, concentration bounds, location: San Jose, California, USA.

[3] Y. K. Tan and J. Yang. Approximate model counting. *Archive of Formal Proofs*, March 2024. https://isa-afp.org/entries/Approximate_Model_Counting.html, Formal proof development.

[4] Wikipedia contributors. Tabulation hashing. https://en.wikipedia.org/wiki/Tabulation_hashing#Universality, Sep 2024. Accessed: 2026-03-28.