Hashing

The most important techniques behind Yahoo! are hashing, hashing and hashing! – Udi Manber

Example 1: path finding

- Assume you want to implement BFS on a graph of 1000 nodes. You can allocate an array of size 1000 to store the node information (‘visited’ flag)
- What about search on Rubik's cube graph (order of 10^6 for $2\times2\times2$ cube, 10^{19} for $3\times3\times3$ cube)?

Example 2: data bases

- Maintain a set of employees (students, messenger users, …), each identified by a social security number (student ID, phone number, …)

Example 3: deduplication

- In a programming language compiler, how to store user-declared identifiers?

- Index *k-mers* in a genomic sequence or words in a text

Hash tables: suppored operations

- A generalization of arrays (“direct addressing”)
- *Goal*: maintain a (possibly evolving) set of objects belonging to a large “universe” (e.g. configurations, ID numbers, words, etc.)
- *Applications*: deduplication, indexing, path finding, file integrity test (checksum), etc.

Hash tables: suppored operations

- A generalization of arrays (“direct addressing”)
- *Goal*: maintain a (possibly evolving) set of objects belonging to a large “universe” (e.g. configurations, ID numbers, words, etc.)
- *Applications*: deduplication, indexing, path finding, file integrity test (checksum), etc.

- INSERT: add a new object
- DELETE: delete existing object
- LOOKUP: check for an object

$\Big\}$ “Dictionary” data structure

- possibly specified by a *key* "associative array"

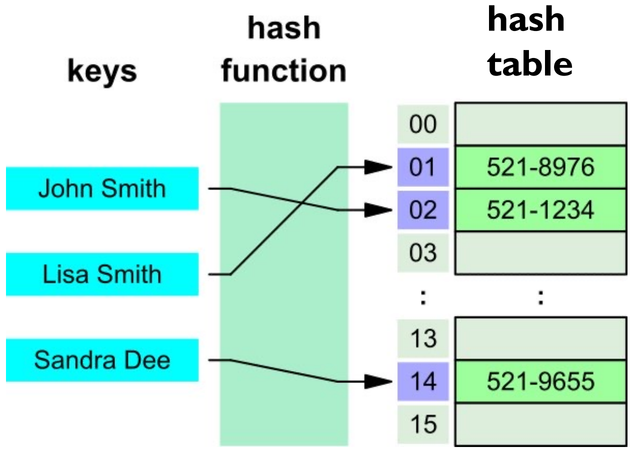
Naive solutions

- Bit array (bitmap)
  - still too big for huge applications
  - does not support access to objects
  - BUT … (cf Bloom filters at the end of this lecture)
- Linked list
  - look-up too slow
- Search trees
  - better but still slow and memory demanding

Hash tables

- Notation
  - U : universe of all possible keys (*Ex*: strings, IP addresses, game configurations, …)
  - K : subset of keys (actually stored in the dictionary), $|K| \ll |U|$
  - $|K| = n$

- Use a table of size proportional to $|K|$: hash table
  - we lose the direct-addressing ability
  - hash function maps keys to entries of the hash table (**buckets** or slots)



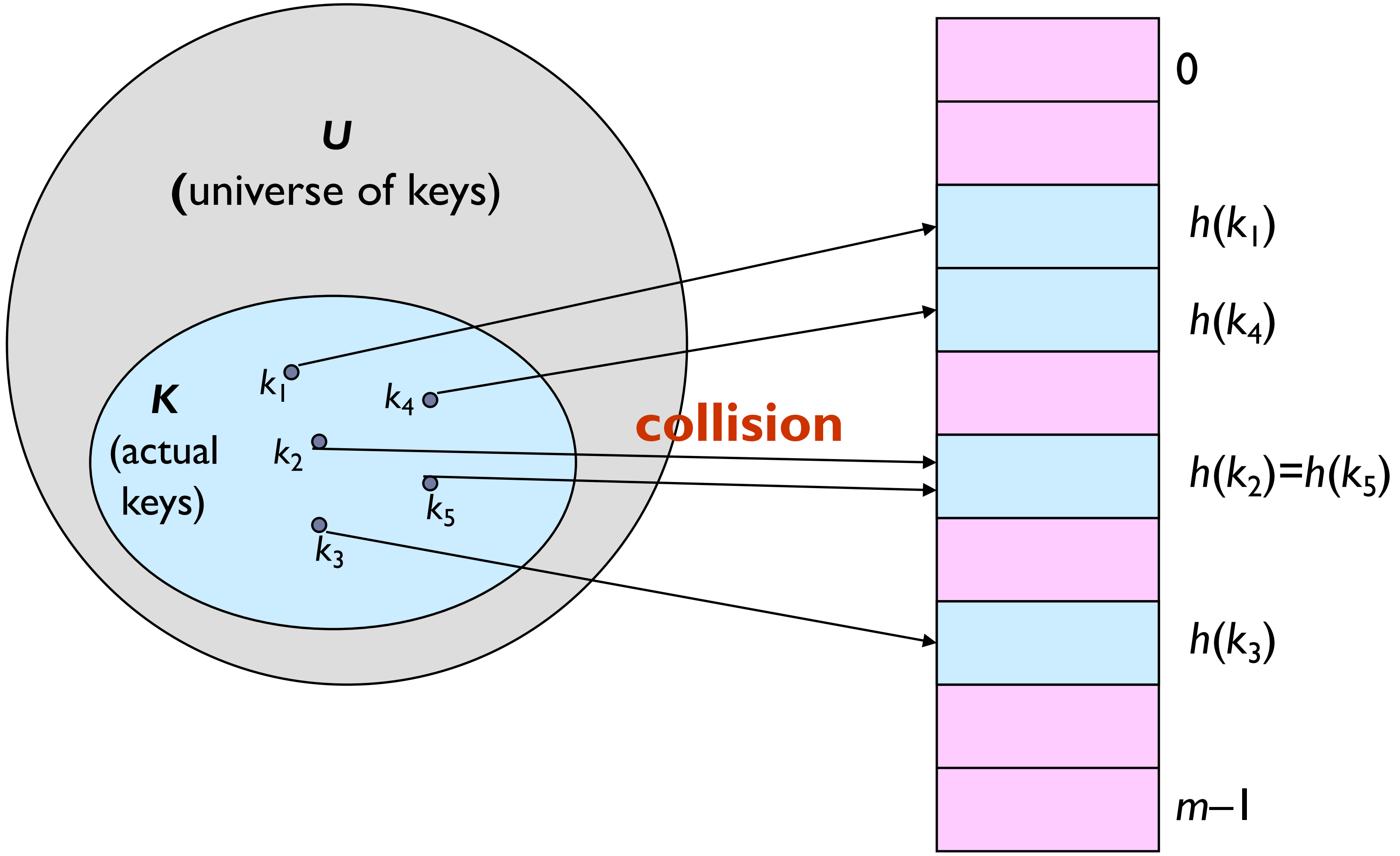
Hash functions

- Hash function *h*: Mapping from *U* to the slots of a hash table $T[0..m-1]$.

$$h : U \to \{0, 1, \ldots, m-1\}$$

- With direct addressing, key k maps to slot $A[k]$
- With hash tables, key k maps or "hashes" to bucket $T[h[k]]$
- $h[k]$ is the *hash value* (or simpy *hash*) of key k

Hashing and collisions



Collisions: birthday "paradox"

- What is the probability that two people from a class of 40 students have their birthday the same day?

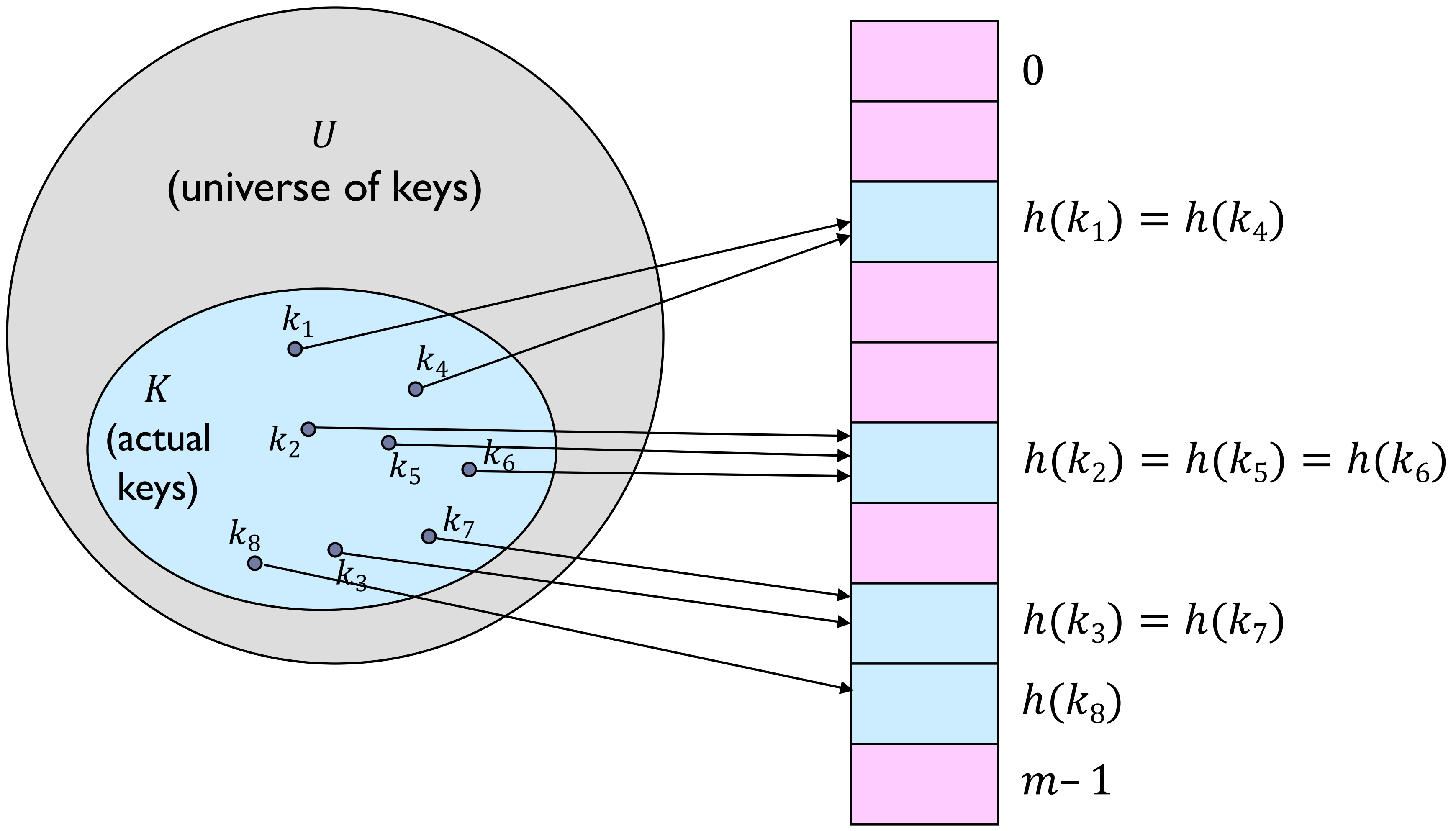
Collisions: birthday "paradox"

- What is the probability that two people from a class of 40 students have their birthday the same day?
- Answer: ≈ 0.89

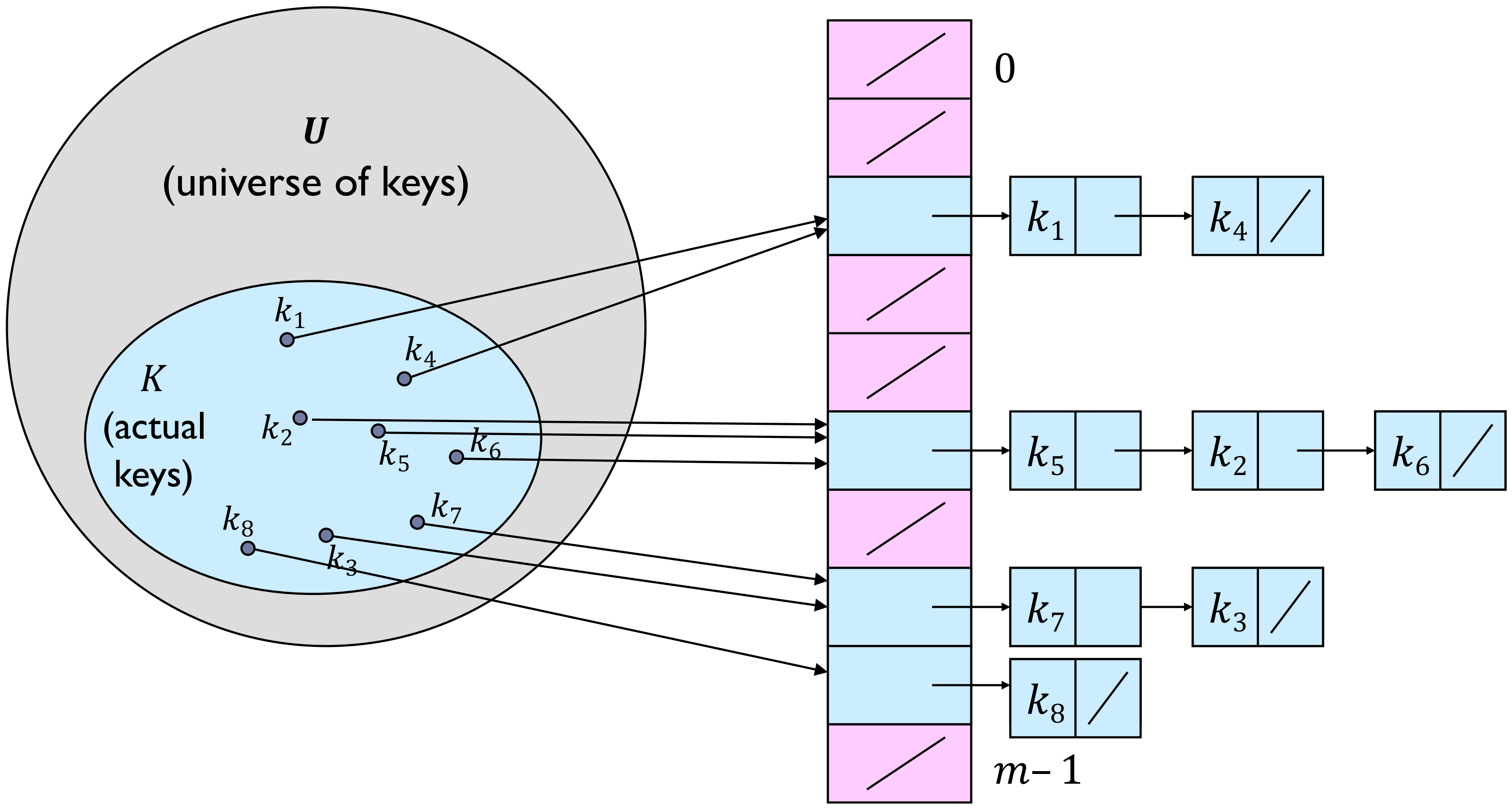
- Birthday paradox: in a group of 23 people, there is about 50% chance that two people have the same bithday

- *Conclusion*: collisions are frequent

I. Collision Resolution by Chaining



Collision Resolution by Chaining



Hashing with chaining

- INSERT$(T.k) : O(1)$
- DELETE$(T.k)$, LOOKUP(k): $O(list\ length)$
- ⇒ a good hash function should distribute keys into buckets *as uniformly as possible*
- uniform hashing ⇒ expected list length is $\alpha = n/m$ (*load factor*)
- the *average* time of DELETE and LOOKUP is $O(1+\alpha) \Rightarrow O(1)$ if $n = O(m)$ (practical case)

Good hash functions

- Hash function should be easy to compute
- Desiging good hash functions is tricky. It is easy to design a bad hash function

- *Examples*: Phone numbers. Benford's law (e.g. prices, population sizes, …)

- Keys are usually considered as natural numbers
- *Example*: Interpret a character string as an integer expressed in some radix notation. Suppose the string is CLRS:
  - ASCII values: C=67, L=76, R=82, S=83.
  - There are 128 basic ASCII values.
  - So, CLRS = $67 \cdot 128^3 + 76 \cdot 128^2 + 82 \cdot 128^1 + 83 \cdot 128^0 = 141{,}764{,}947$

Division Method

- Map a key k into one of the m slots by taking the remainder of k divided by m. That is,

$$h(k) = k \bmod m$$

- *Example:* $m = 31$ and $k = 78 \Rightarrow h(k) = 16$

- *Advantage:* Fast, since requires just one division operation
- *Disadvantage:* Have to avoid certain values of m.
  - Don't pick certain values, such as $m = 2^p$ (as the hash won't depend on all bits of k)

- *Good choice for m:*
  - Primes, not too close to power of 2 (or 10) are good.

Multiplication Method

- If $0 < A < 1$, $h(k) = \lfloor m\ (kA \bmod 1) \rfloor = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ where $(kA \bmod 1) = kA - \lfloor kA \rfloor$: the fractional part of kA
- *Disadvantage:* Slower than the division method.
- *Advantage:* Value of m is not critical.
  - Typically chosen as a power of 2, i.e., $m = 2^p$, which makes the implementation easy

- *Example:* $m = 1000, k = 123, A = 0.6180339887\ldots$

$h(k) = \lfloor 1000(123 \cdot 0.6180339887 \bmod 1) \rfloor = \lfloor 1000 \cdot 0.018169 \ldots \rfloor = 18$

II. Collision Resolution by Open Addressing

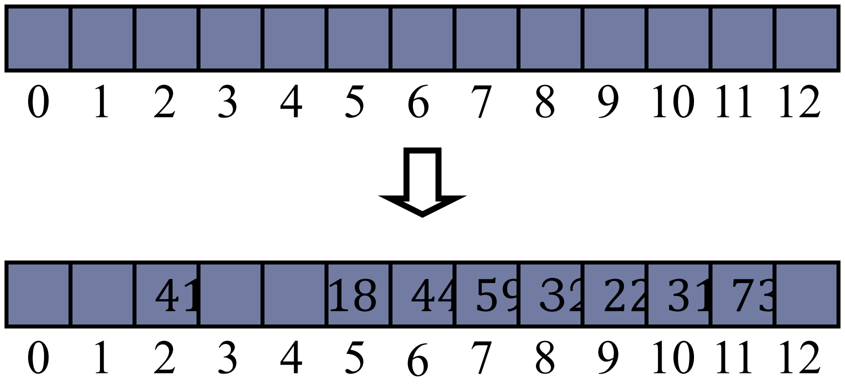
- All elements are stored in the hash table itself
- $\Rightarrow \quad n \leq m,$ no pointers

- hash function $h(k, i)$ where $i = 0,1,2, \dots, m-1,$ and $< h(k,0), h(k,1), \dots, h(k, m-1) >$ is a permutation
- when inserting/looking up k, probe $h(k,0), h(k,1), \dots$ (*probe sequence*) until
  - we find k, or
  - the bucket contains nil, or
  - m buckets have been unsuccessfully probed
- deletion is complicated, needs a special key "deleted", time may not be dependent on the load factor

Open Addressing: Linear probing

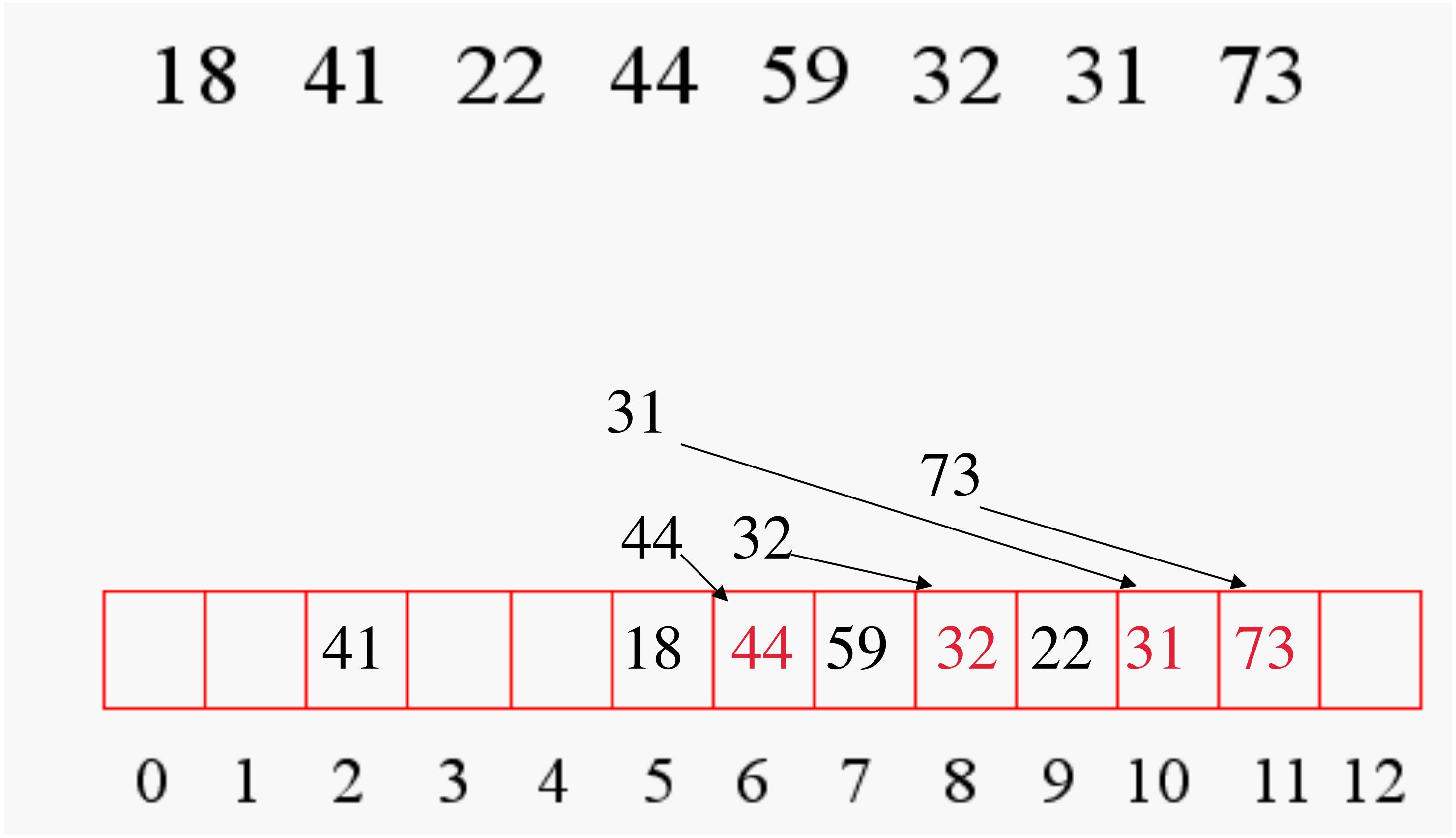
- The colliding item is placed in a different cell of the table
- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell $h(k,i) = (h'(k) + i) \bmod m$
- Each table cell inspected is referred to as a "probe"
- Colliding items clump together, causing future collisions to cause a longer sequence of probes

- *Example:*
  - $h'(k) = k \bmod 13$
  - Insert keys $18, 41, 22, 44, 59, 32, 31, 73,$ in this order

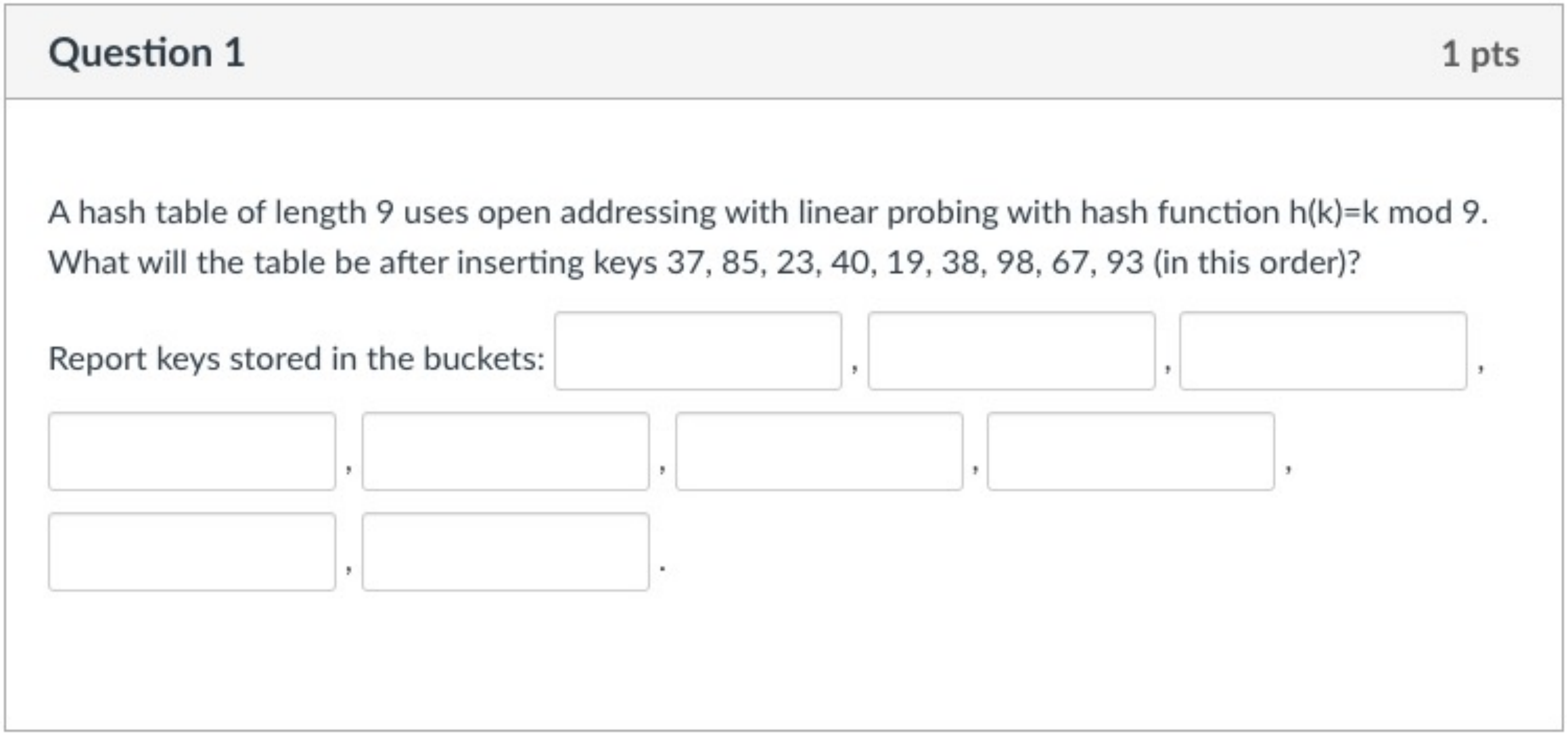


Example (cont.)

$$h'(k) = k \bmod 13$$



Quiz 5.1



Question 1 **1 pts**

A hash table of length 9 uses open addressing with linear probing with hash function h(k)=k mod 9. What will the table be after inserting keys 37, 85, 23, 40, 19, 38, 98, 67, 93 (in this order)?

Report keys stored in the buckets: , , , , , , , .

Quadratic probing

- $h(k, j) = (h'(k) + c_1 \cdot j + c_2 \cdot j^2) \bmod m$

- for example, $h(k, j) = (h'(k) + \frac{j(j+1)}{2}) \bmod m$

  $h(k, 0), h(k, 1), \ldots, h(k, m-1)$ is a permutation if m is a power of 2

Probing:

```
$j = 0; i = h'(k)$
**while** bucket $j$ is not empty AND $j < m - 1$
    $j = j + 1; i = i + j$
```

- quadratic probing works better than linear probing (less clumping)

Double Hashing

- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
  $$h(k,j) = (h(k) + jd(k)) \bmod m$$
  for $j = 0,1,\ldots,m-1$

- The secondary hash function $d(k)$ cannot have zero values

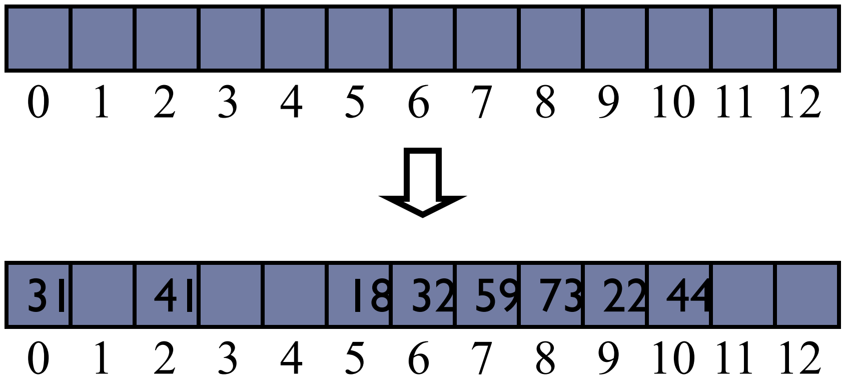
- m should be relatively prime to $d(k)$, e.g. $m = 2^q$ and $d(k)$ odd, or m is prime and $d(k) < m$

- Double hashing is usually more efficient than linear and quadratic probing

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $m = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys $18, 41, 22, 44, 59, 32, 31, 73$, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



Quiz 5.2

Question 1

1 pts

A hash table of length 8 uses open addressing with hash function h(k)=k mod 8, and double hashing with the second hash function $d(k) = 2\lfloor(k \mod 8)/2\rfloor + 1$. What will the table be after inserting keys 18, 37, 82, 20, 56, 73, 26, 41 (in this order)?

Report keys stored in the buckets: ____ , ____ , ____ , ____ , ____ , ____ , ____ , ____ .

Performance of Open Addressing

- Assuming that $< h(k,0), h(k,1), \dots, h(k,m-1) >$ is a random permutation (uniformly drawn), the expected number of probes in an insertion (or unsuccessful search) with open addressing is
  $$1/(1-\alpha),$$
  where $\alpha = n/m$ the load factor

- *Explanation*:
  let $p_i = \mathrm{P}[i \text{ first buckets are full}] = \alpha^i \quad (p_0 = 0)$
  $E[\text{number of probes}] = 1 + \sum_{i=1}^{m-1} i \cdot P[i-1 \text{ full buckets followed by an empty one}] = \sum_{i=1}^{m-1} i \cdot (p_{i-1} - p_i) = 1 + \sum_{i=1}^{m-1} p_i \approx$
  $$1 + \alpha + \alpha^2 + \alpha^3 + \cdots = 1/(1-\alpha)$$

Performance of Open Addressing

- Assuming that $< h(k,0), h(k,1), \dots, h(k,m-1) >$ is a random permutation (uniformly drawn), the expected number of probes in an insertion (or unsuccessful search) with open addressing is

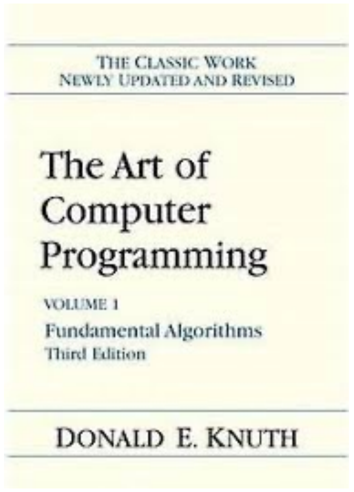
  $$1/(1-\alpha),$$

  where $\alpha = n/m$ the load factor

- The expected number of probes for a successful search is

  $$\frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{1-\frac{i}{m}} \le (1/\alpha)\ln(1/(1-\alpha))$$

Historical remarks


THE CLASSIC WORK
NEWLY UPDATED AND REVISED

The Art of
Computer
Programming

VOLUME 1
Fundamental Algorithms
Third Edition

DONALD E. KNUTH

Hashing by open addressing:
Analysed by Donald Knuth in 1962 (invention attributed to Andrei Ershov)

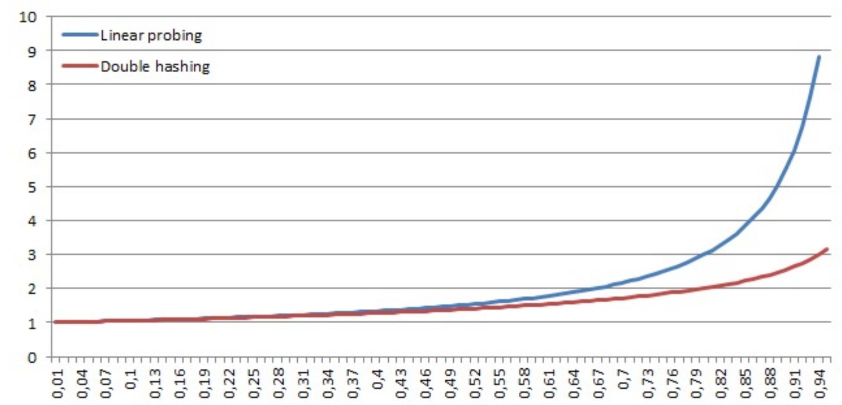
Hashing: some conclusions

- Chaining:
  - easy implementation
  - fast in practice
  - uses more memory
- Open addressing:
  - uses less memory
  - more complex removals

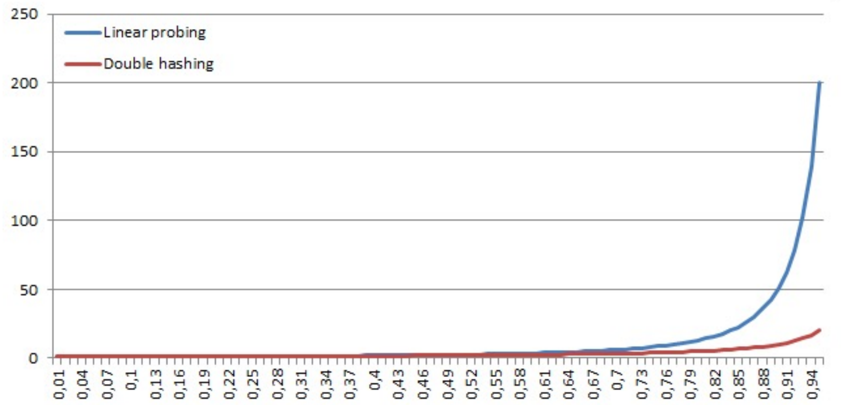
- Implemented in standard libraries, e.g. `std::unordered_map` in C++

Linear probing vs Double hashing

comparison of average number of operations



successful search



unsuccessful search

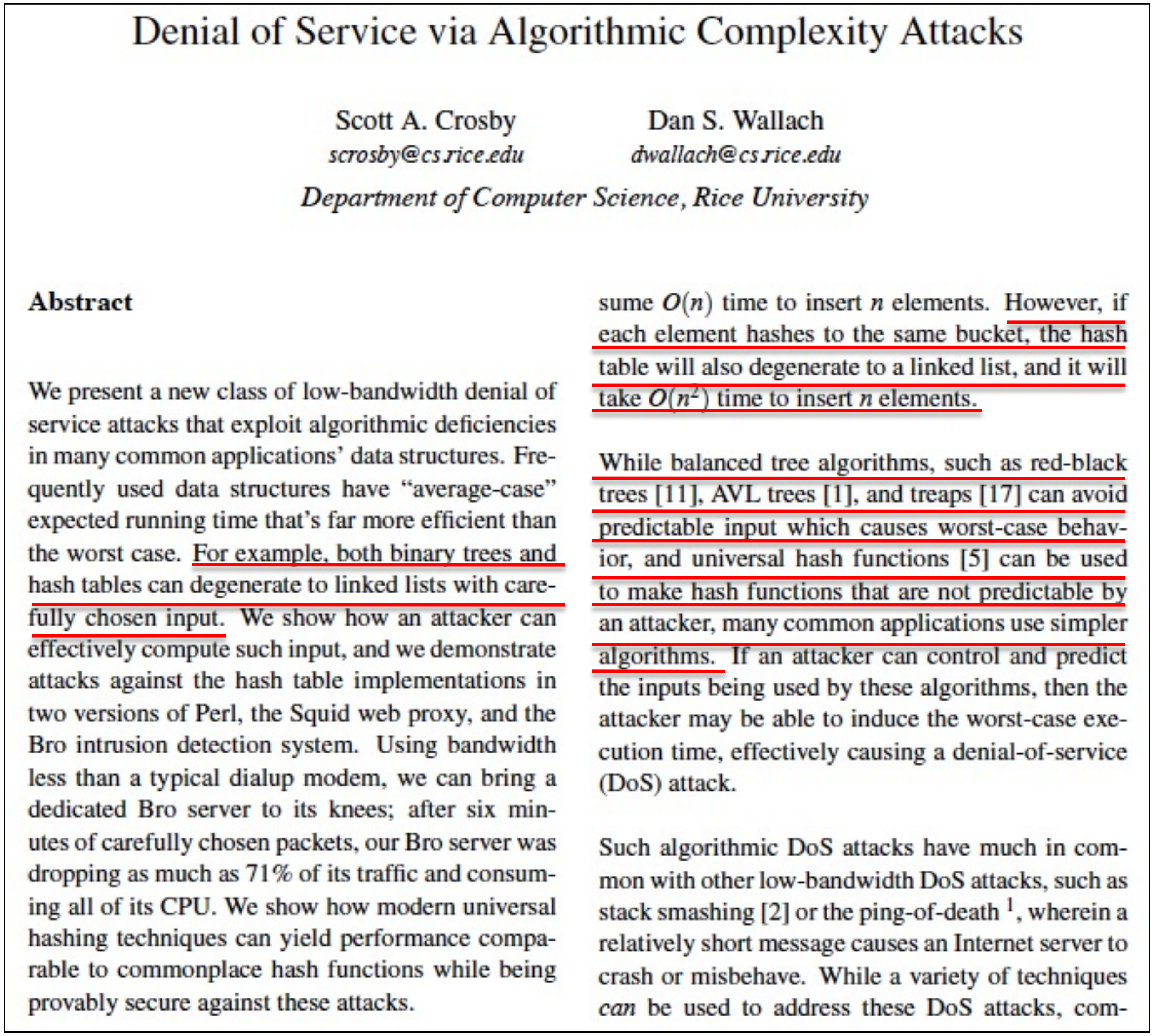
Universal hashing

(Variant of hashing with chaining)

Universal hashing (with chaining)

- *Motivation*: avoid pathological ("adversary") datasets
  - practical applications in avoiding DDoS attacks

12th USENIX Security Symposium, 2003



Denial of Service via Algorithmic Complexity Attacks

Scott A. Crosby
scrosby@cs.rice.edu

Dan S. Wallach
dwallach@cs.rice.edu

Department of Computer Science, Rice University

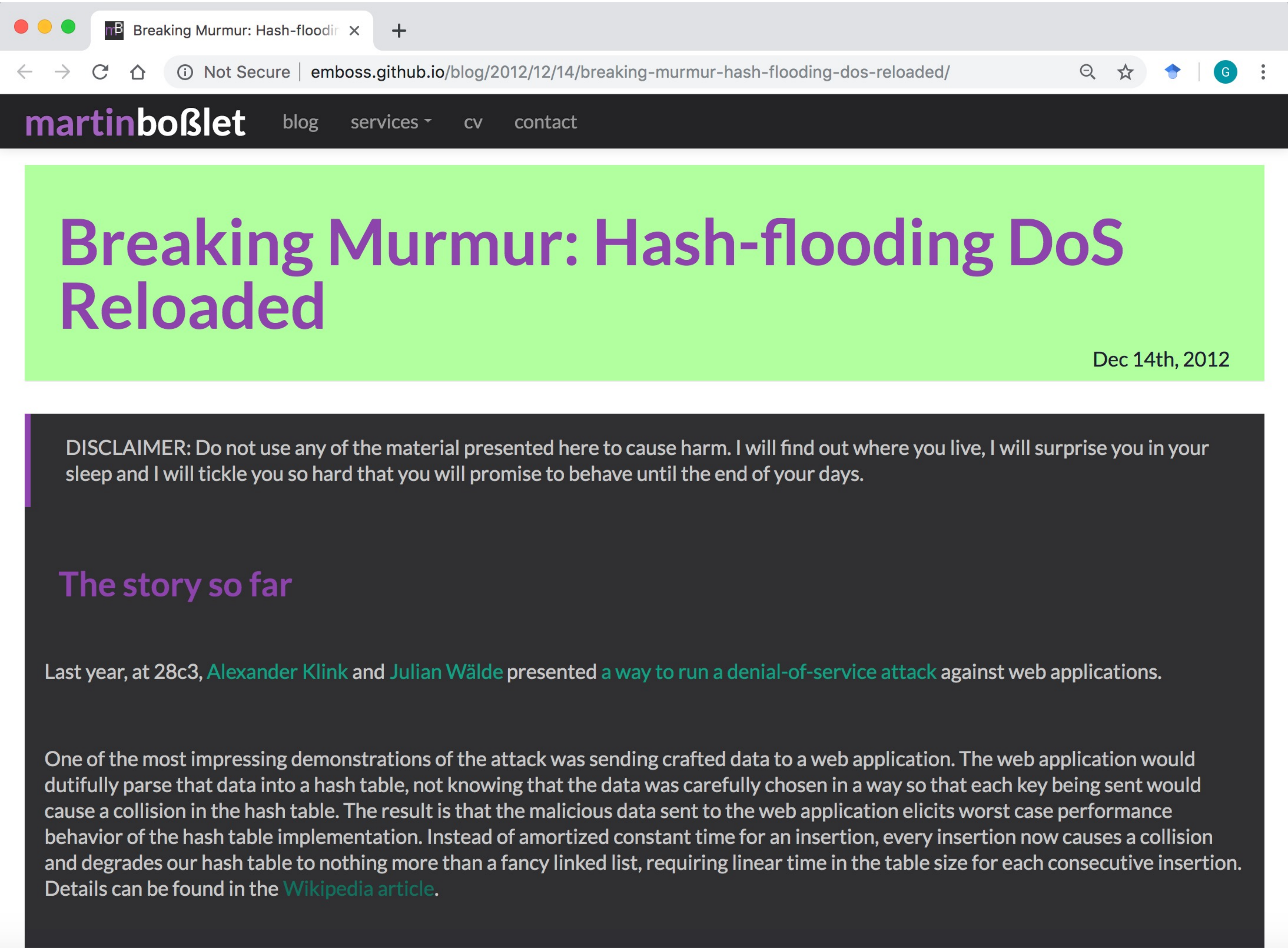
Abstract

We present a new class of low-bandwidth denial of service attacks that exploit algorithmic deficiencies in many common applications' data structures. Frequently used data structures have "average-case" expected running time that's far more efficient than the worst case. For example, both binary trees and hash tables can degenerate to linked lists with carefully chosen input. We show how an attacker can effectively compute such input, and we demonstrate attacks against the hash table implementations in two versions of Perl, the Squid web proxy, and the Bro intrusion detection system. Using bandwidth less than a typical dialup modem, we can bring a dedicated Bro server to its knees; after six minutes of carefully chosen packets, our Bro server was dropping as much as 71% of its traffic and consuming all of its CPU. We show how modern universal hashing techniques can yield performance comparable to commonplace hash functions while being provably secure against these attacks.

sume $O(n)$ time to insert n elements. However, if each element hashes to the same bucket, the hash table will also degenerate to a linked list, and it will take $O(n^2)$ time to insert n elements.

While balanced tree algorithms, such as red-black trees [11], AVL trees [1], and treaps [17] can avoid predictable input which causes worst-case behavior, and universal hash functions [5] can be used to make hash functions that are not predictable by an attacker, many common applications use simpler algorithms. If an attacker can control and predict the inputs being used by these algorithms, then the attacker may be able to induce the worst-case execution time, effectively causing a denial-of-service (DoS) attack.

Such algorithmic DoS attacks have much in common with other low-bandwidth DoS attacks, such as stack smashing [2] or the ping-of-death [1], wherein a relatively short message causes an Internet server to crash or misbehave. While a variety of techniques *can* be used to address these DoS attacks, com-


Breaking Murmur: Hash-floodi ×

Not Secure | emboss.github.io/blog/2012/12/14/breaking-murmur-hash-flooding-dos-reloaded/

martinboßlet blog services cv contact

Breaking Murmur: Hash-flooding DoS Reloaded

Dec 14th, 2012

> DISCLAIMER: Do not use any of the material presented here to cause harm. I will find out where you live, I will surprise you in your sleep and I will tickle you so hard that you will promise to behave until the end of your days.

The story so far

Last year, at 28c3, Alexander Klink and Julian Wälde presented a way to run a denial-of-service attack against web applications.

One of the most impressing demonstrations of the attack was sending crafted data to a web application. The web application would dutifully parse that data into a hash table, not knowing that the data was carefully chosen in a way so that each key being sent would cause a collision in the hash table. The result is that the malicious data sent to the web application elicits worst case performance behavior of the hash table implementation. Instead of amortized constant time for an insertion, every insertion now causes a collision and degrades our hash table to nothing more than a fancy linked list, requiring linear time in the table size for each consecutive insertion. Details can be found in the Wikipedia article.

Universal hashing (with chaining)

- introduced by Carter&Wegman (1979)

- *Definition*: A family H of hash functions is called *universal* iff for any pair of keys k, l,
$$\mathrm{P}_{h \in H}[h(k) = h(l)] \leq 1/m$$
(Equiv., the nb of hash functions h with $h(k) = h(l)$ is $\leq |H|/m$)

- *Theorem:* the expected time (over $h \in H$) of INSERT, DELETE, LOOKUP is $O(1 + \alpha)$

  NB: no assumption on the distribution of keys

Universal class of hash functions

- Choose a large prime p (larger than the maximum key)
- Let $0 \leq a, b \leq p - 1, a \neq 0$
- for a key k, define $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$
- $H_{p,m} = \{h_{a,b}\}$ is a universal family of hash functions
- *Proof idea*:
  - given a, b, values $((ak + b) \bmod p)$ are distinct for different k's
  - for $k_1 \neq k_2$ distinct pairs (a, b) yield distinct pairs $((ak_1 + b) \bmod p, (ak_2 + b) \bmod p)$
  - $\mathrm{P}[(ak_1 + b) \bmod p =_{\bmod m} (ak_2 + b) \bmod p] \leq 1/m$

- In practice p is often set to $2^{31} - 1$ for 32-bit numbers and to $2^{61} - 1$ for 64-bit numbers (Mersenne primes)

Example of universal hashing

- Assume we are hashing IP addresses $x_1.x_2.x_3.x_4$ with $0 \leq x_i \leq 255$
- Choose m a prime number
- Consider quadruples $a = (a_1, a_2, a_3, a_4)$ with $0 \leq a_i \leq m - 1$
- Define
  $h_a(x_1.x_2.x_3.x_4) = (a_1 \cdot x_1 + a_2 \cdot x_2 + a_3 \cdot x_3 + a_4 \cdot x_4) \bmod m$

- $H = \{h_a\}$ is a universal family (can be proved)

Remarks

- Other (more) efficient universal hashing schemes exist, such as Multiply-shift [Dietzfelbinger et al. A reliable randomized algorithm for the closest-pair problem. J. Algorithms, 25:19–51, 1997]

$h_a : [0..2^w - 1] \to [0..2^l - 1]$ defined by
$h_a(x) = \lfloor (ax \bmod 2^w)/2^{w-l} \rfloor$ for a random w-bit odd integer a

- for details see [M. Thorup, High Speed Hashing for Integers and Strings, arxiv:1504.06804, May 2019]

Perfect hashing

Motivation

- Can we guarantee a ***worst-case*** $O(1)$ time for hash table operations?

Motivation

- Can we guarantee a ***worst-case*** $O(1)$ time for hash table operations?

- **Yes** if the set of keys is *static*

Motivation

- Can we guarantee a ***worst-case*** $O(1)$ time for hash table operations?

- **Yes** if the set of keys is *static*

- Naive solution: enumerate keys ⇒ sorting!
  - construction $O(n \log n)$
  - storage (of the function) $O(n)$
  - function computation $O(\log n)$

Motivation

- Can we guarantee a ***worst-case*** $O(1)$ time for hash table operations?

- **Yes** if the set of keys is *static*

- However, the construction time is ***expected*** $O(n)$ (Las Vegas algorithm)

Collisions: analysis

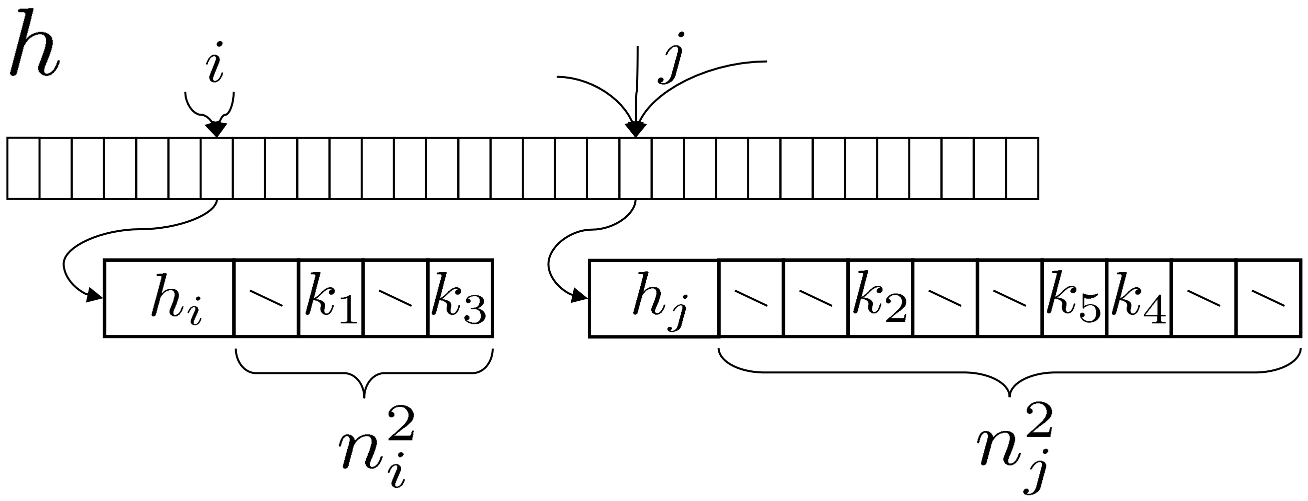
- What is the expected number of collisions?
  - i.e. number of pairs $(k, l), k \neq l$ and $h(k) = h(l)$
- $X_{kl} = 1$ iff $h(k) = h(l)$
- $E[\sum_{k \neq l} X_{kl}] = \sum_{k \neq l} E[X_{kl}] = \binom{n}{2} \frac{1}{m} \approx \frac{n^2}{2m}$

- Remarks:
  - if $m \approx n^2$, then we have $\frac{1}{2}$ expected collisions $\Rightarrow$ $P[\text{collision}] \leq \frac{1}{2}$ (cf birthday paradox)
  - by iterating, we can build a hash table with NO collision after $O(1)$ trials ***in expectation***

Markov inequality
$$P[X \geq a] \leq \frac{E[X]}{a}$$

Perfect hashing

- Fredman, Komlós, Szemerédi (1984)
- Guarantees $O(1)$ **worst-case** time of LOOKUP for a ***static*** set of keys. Solution uses universal hashing.
- 2-level hash scheme:



- LOOKUP: **worst-case** $O(1)$

Why $\sum n_i{}^2$ can be maid $\leq 2n$

- $\sum n_i{}^2 = n + [\text{nb of pairs which collide}]$
- $\mathrm{E}[\text{nb of pairs which collide}] = 2 \cdot \frac{n^2}{n} = n$
- $\Rightarrow \mathrm{E}[\sum n_i{}^2] = 2n \underset{\text{by Markov inequality}}{\Rightarrow} P[\sum n_i{}^2 > 4n] < 1/2$
- Algorithm (sketch)
  - hash to primary table of size $O(n)$ using a universal h.f. h
  - hash each non-empty bucket to a table of size $n_i{}^2$; if $\sum n_i{}^2 > 4n$, rehash
  - using a universal h.f. h_i; if collision, rehash until there is none (expected $O(1)$ time by birthday paradox)

Perfect hashing is practical!

- practical implementations exist, e.g. `gperf` in C++

Minimal Perfect Hash Functions (MPHF)

- MPHF: bijective perfect hash function, i.e. $h\colon S \to [1..|S|]$
- enumeration of keys of S
- efficient construction algorithms of MPFH exist, see e.g. https://blog.gopheracademy.com/advent-2017/mphf/
- space efficiency: a few bits per key
- lower bound ≈ 1.44 bits/key, cf [Mehlhorn 82], [Belazzougui, Botelho, Dietzfelbinger 09]