Cuckoo Filter: Practically Better Than Bloom

Bin Fan (CMU/Google)
David Andersen (CMU)
Michael Kaminsky (Intel Labs)
Michael Mitzenmacher (Harvard)

Carnegie Mellon
Parallel Data Laboratory

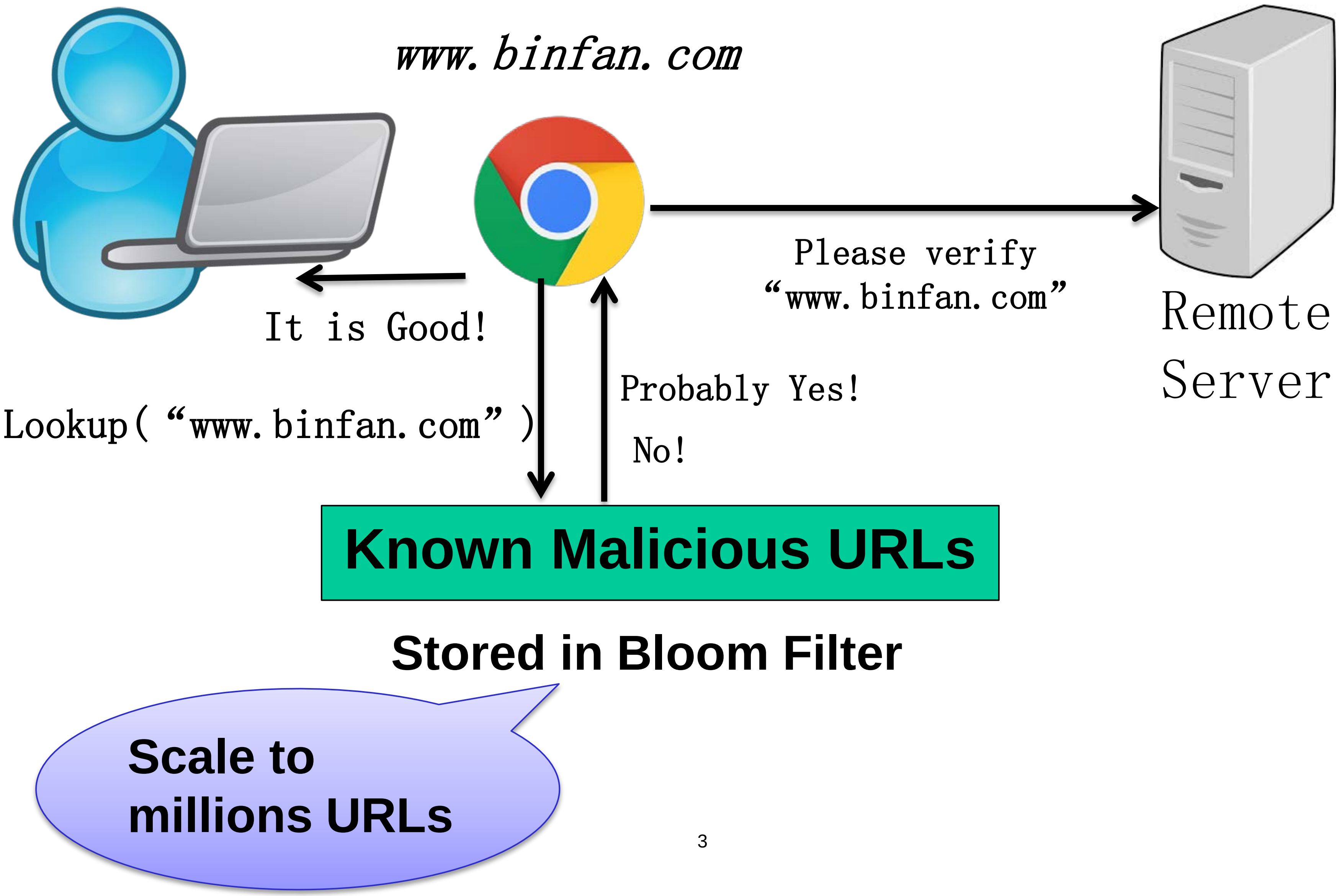
What is Bloom Filter? A Compact Data Structure Storing Set-membership

- Bloom Filters answer "is item ***x*** in set ***Y*** " by:
  - "**definitely no**", or
  - "**probably yes**" with probability ***ε*** to be wrong

false positive rate

- Benefit: not always precise but highly compact
  - Typically a few bits per item
  - Achieving lower ***ε*** (more accurate) requires spending more bits per item

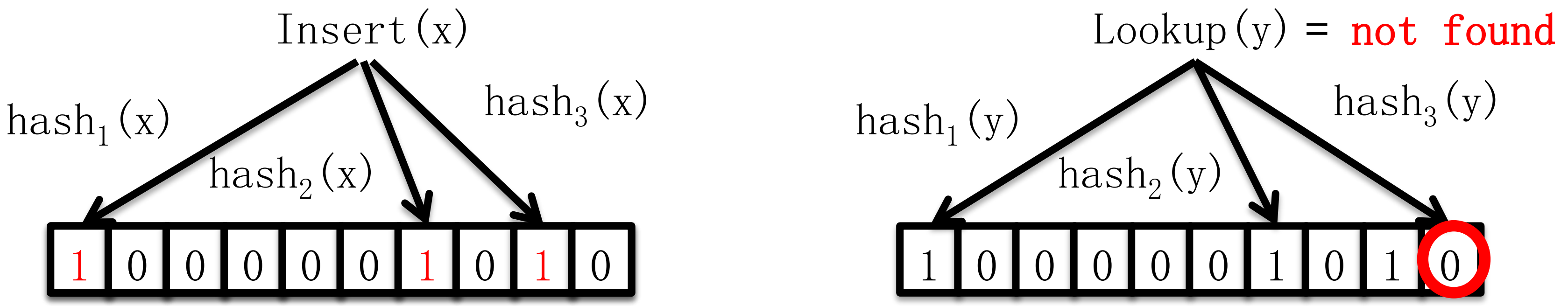
Example Use: Safe Browsing



Bloom Filter Basics

A Bloom Filter consists of ***m*** bits and ***k*** hash functions

Example: ***m*** = 10, ***k*** = 3



Succinct Data Structures for Approximate Set-membership Tests

	High Performance	Low Space Cost	Delete Support
Bloom Filter	✔	✔	✗
Counting Bloom Filter	✔	✗	✔
Quotient Filter	✗	✔	✔

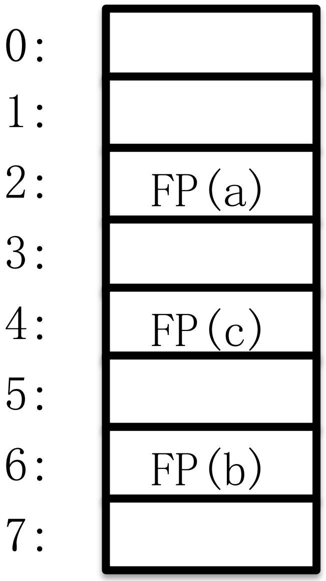
Can we achieve all three in practice?

Outline

- Background
- **Cuckoo filter algorithm**
- Performance evaluation
- Summary

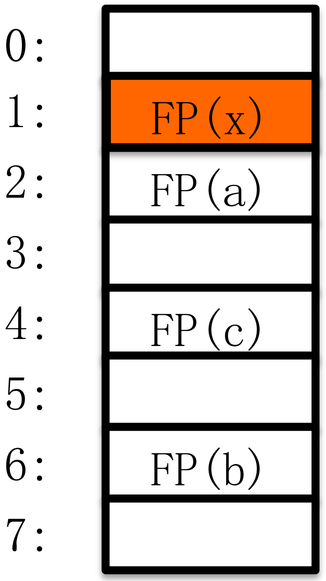
Basic Idea: Store Fingerprints in Hash Table

- **Fingerprint**(x): A hash value of x
  - Lower false positive rate ε, longer fingerprint



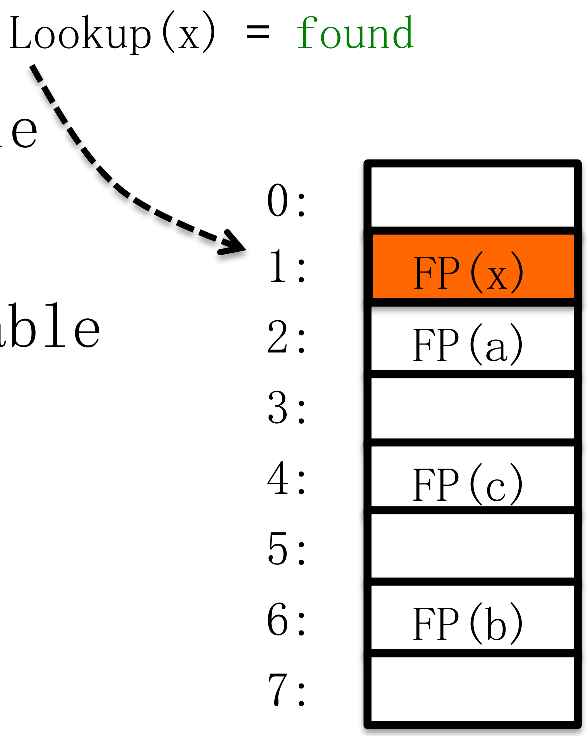
Basic Idea: Store Fingerprints in Hash Table

- **Fingerprint**(x): A hash value of x
  - Lower false positive rate ε, longer fingerprint
- **Insert**(x):
  - add **Fingerprint(x)** to hash table



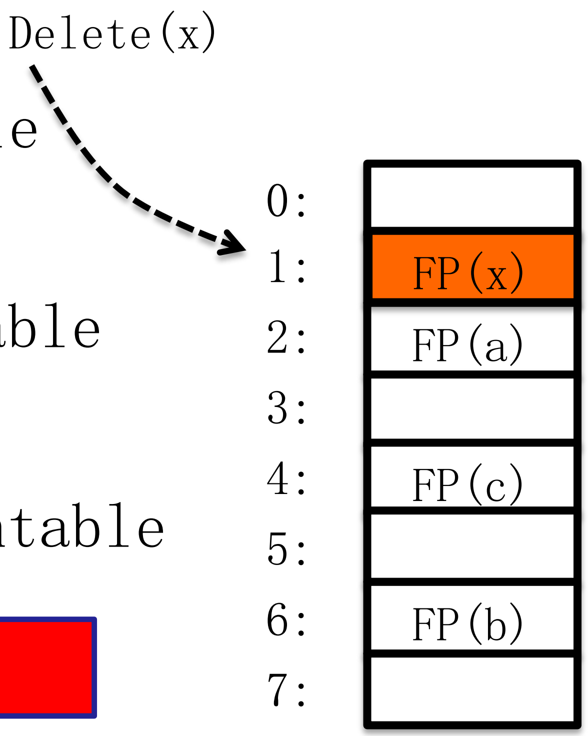
Basic Idea: Store Fingerprints in Hash Table

- **Fingerprint**(x): A hash value of x
  - Lower false positive rate ε, longer fingerprint
- **Insert**(x):
  - add **Fingerprint(x)** to hash table
- **Lookup**(x):
  - search **Fingerprint**(x) in hashtable



Basic Idea: Store Fingerprints in Hash Table

- **Fingerprint**(x): A hash value of x
  - Lower false positive rate ε, longer fingerprint
- **Insert**(x):
  - add **Fingerprint(x)** to hash table
- **Lookup**(x):
  - search **Fingerprint**(x) in hashtable
- **Delete**(x):
  - remove **Fingerprint**(x) from hashtable

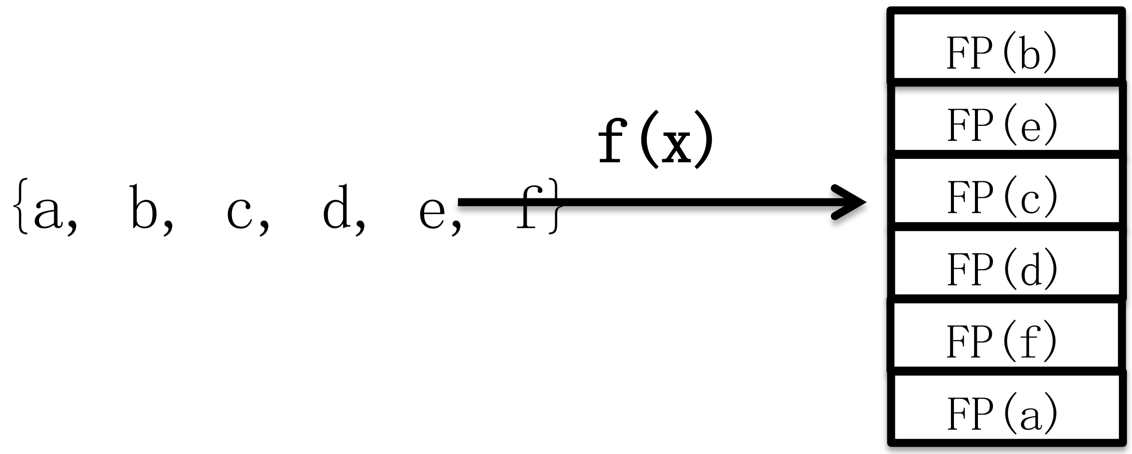


How to Construct Hashtable?

Strawman

(Minimal) Perfect Hashing: No Collision but Update is Expensive

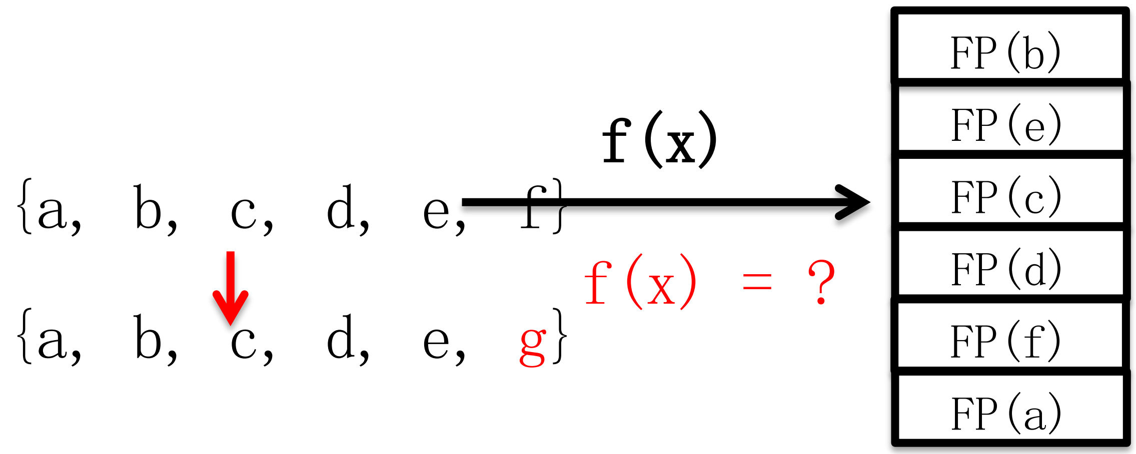
- Perfect hashing: maps all items with no collisions



Strawman

(Minimum) Perfect Hashing: No Collision but Update is Expensive

- Perfect hashing: maps all items with no collisions

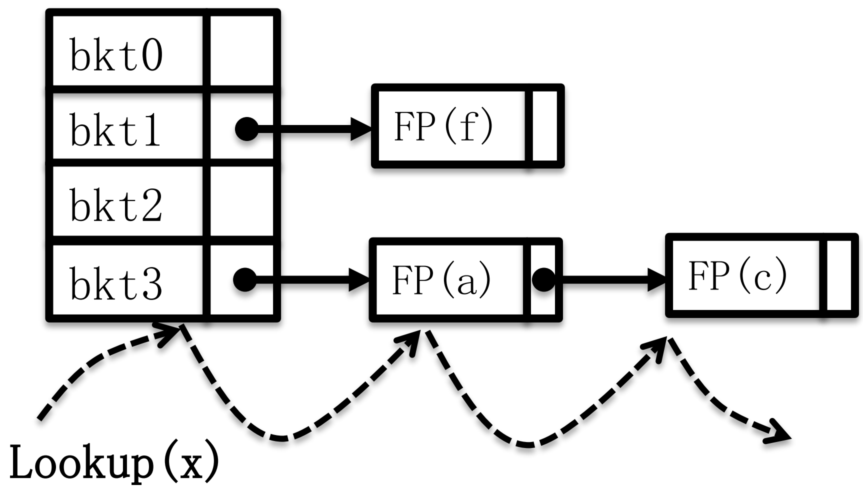


- Changing set must recalculate f ➔ high cost/bad performance of update

Strawman

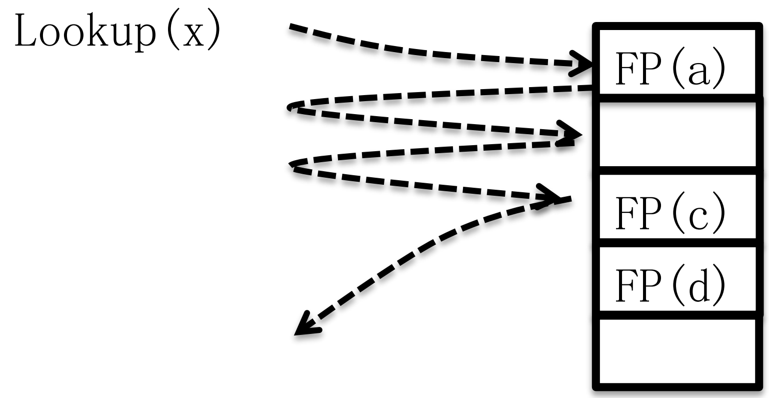
Convention Hash Table: High Space Cost

- Chaining :



- Pointers ➔
  low space utilization

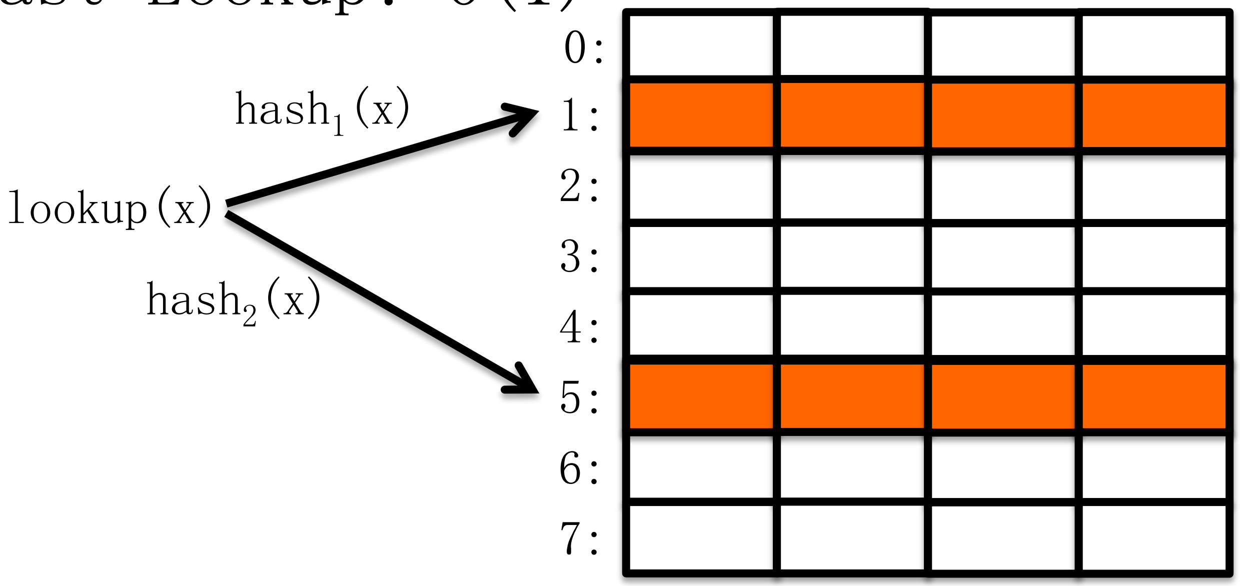
- Linear Probing



- Making lookups O(1) requires large % table empty ➔
  low space utilization
- Compare multiple fingerprints sequentially ➔
  more false positives

Cuckoo Hashing[Pagh2004] Good But ..

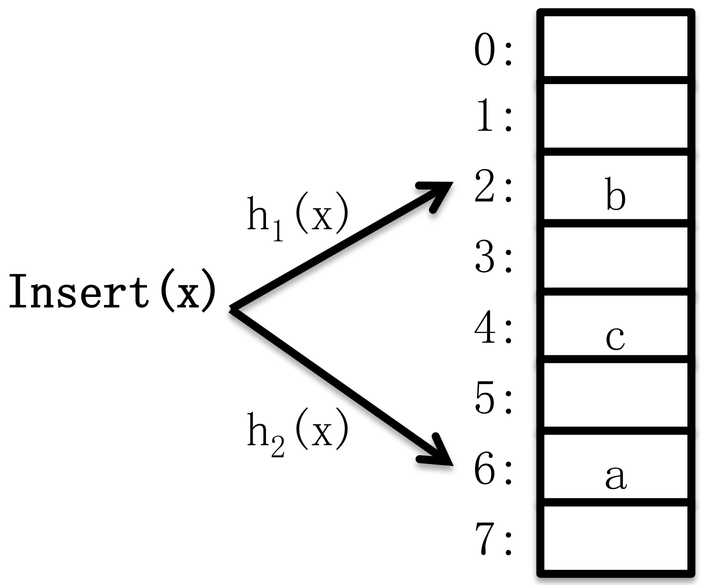
- High Space Utilization
  - 4-way set-associative table: >95% entries occupied
- Fast Lookup: O(1)



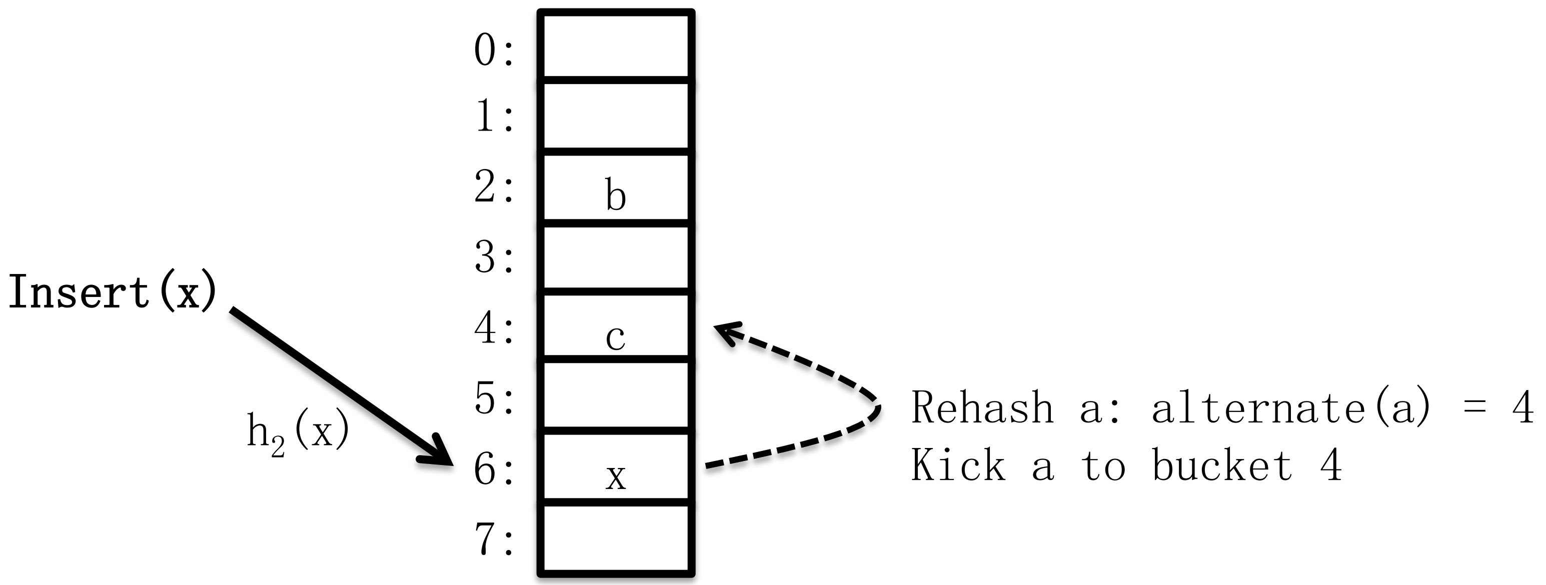
Standard cuckoo hashing doesn't work with

[Pagh2004] Cuckoo hashing.

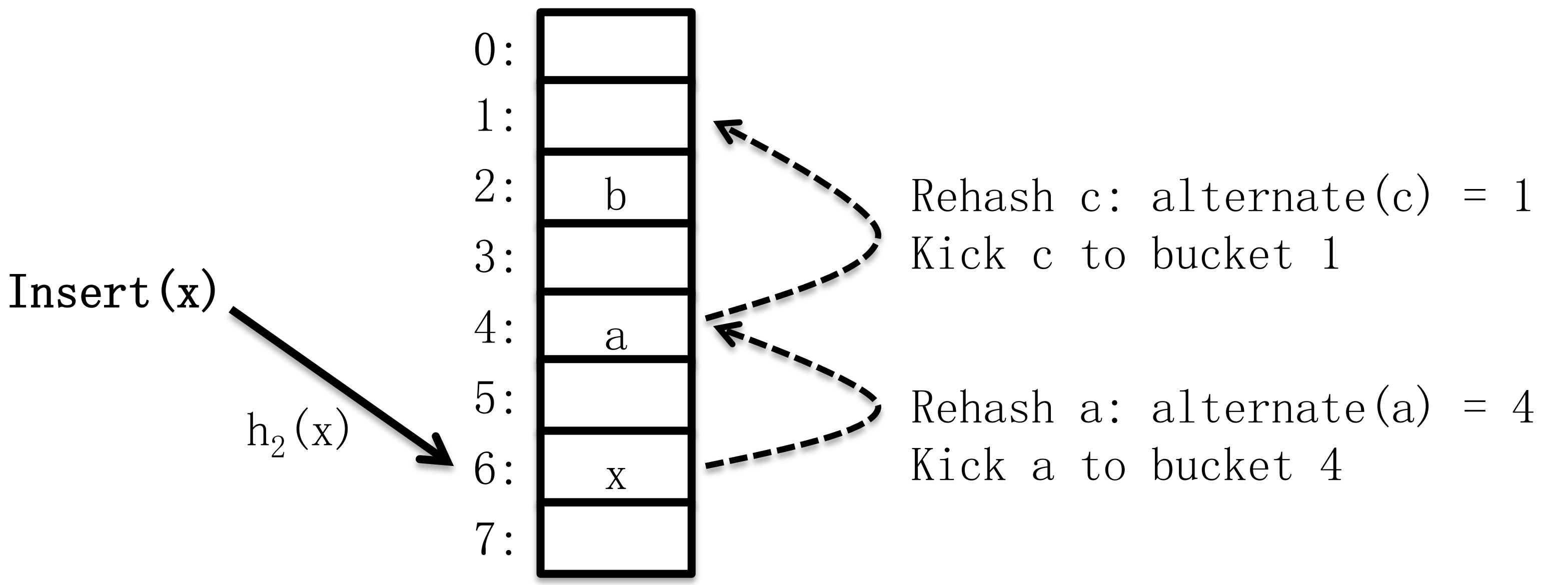
Standard Cuckoo Requires Storing Each Item



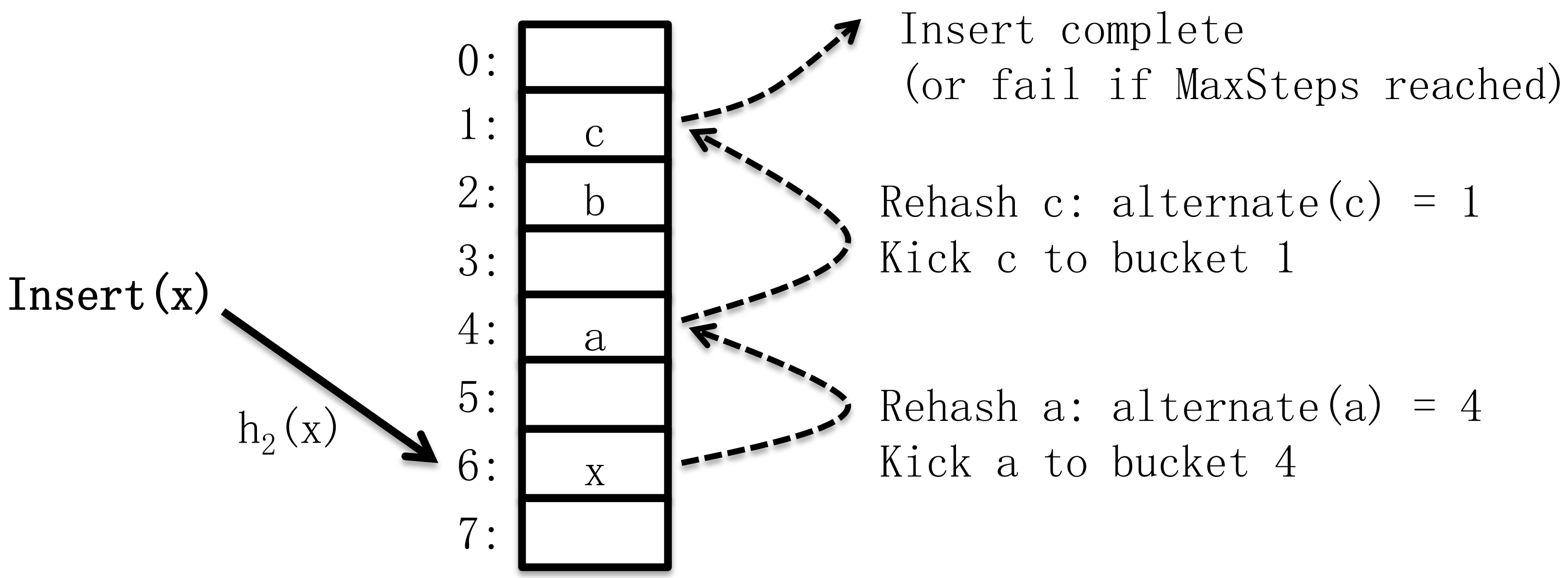
Standard Cuckoo Requires Storing Each Item



Standard Cuckoo Requires Storing Each Item

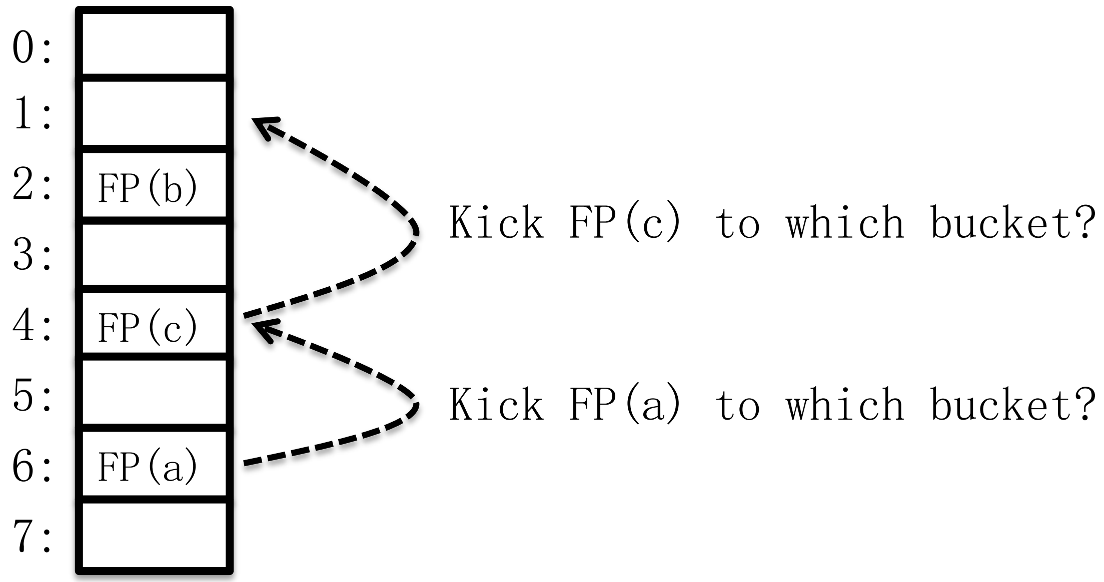


Standard Cuckoo Requires Storing Each Item



Challenge: How to Perform Cuckoo?

- Cuckoo hashing requires rehashing and displacing existing items



With only fingerprint, how to calculate item's alternate

Solution

We Apply Partial-Key Cuckoo

- Standard Cuckoo Hashing: **two independent hash functions for two buckets**

  $bucket1 = hash_1(x)$

  $bucket2 = hash_2(x)$

- Partial-key Cuckoo Hashing: **use one bucket and fingerprint to derive the other** [Fan2013]

  $bucket1 = hash(x)$

  $bucket2 = bucket1 \oplus hash(FP(x))$

To displace existing fingerprint:

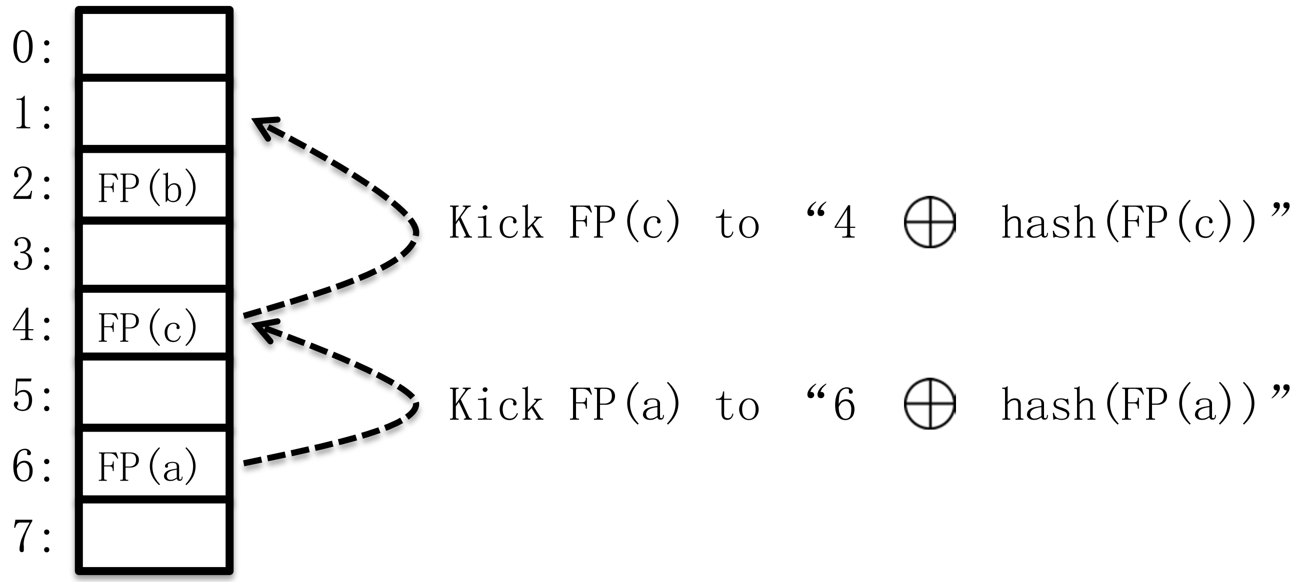
$$alternate(x) = current(x) \oplus hash(FP(x))$$

[Fan2013] MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing

Solution

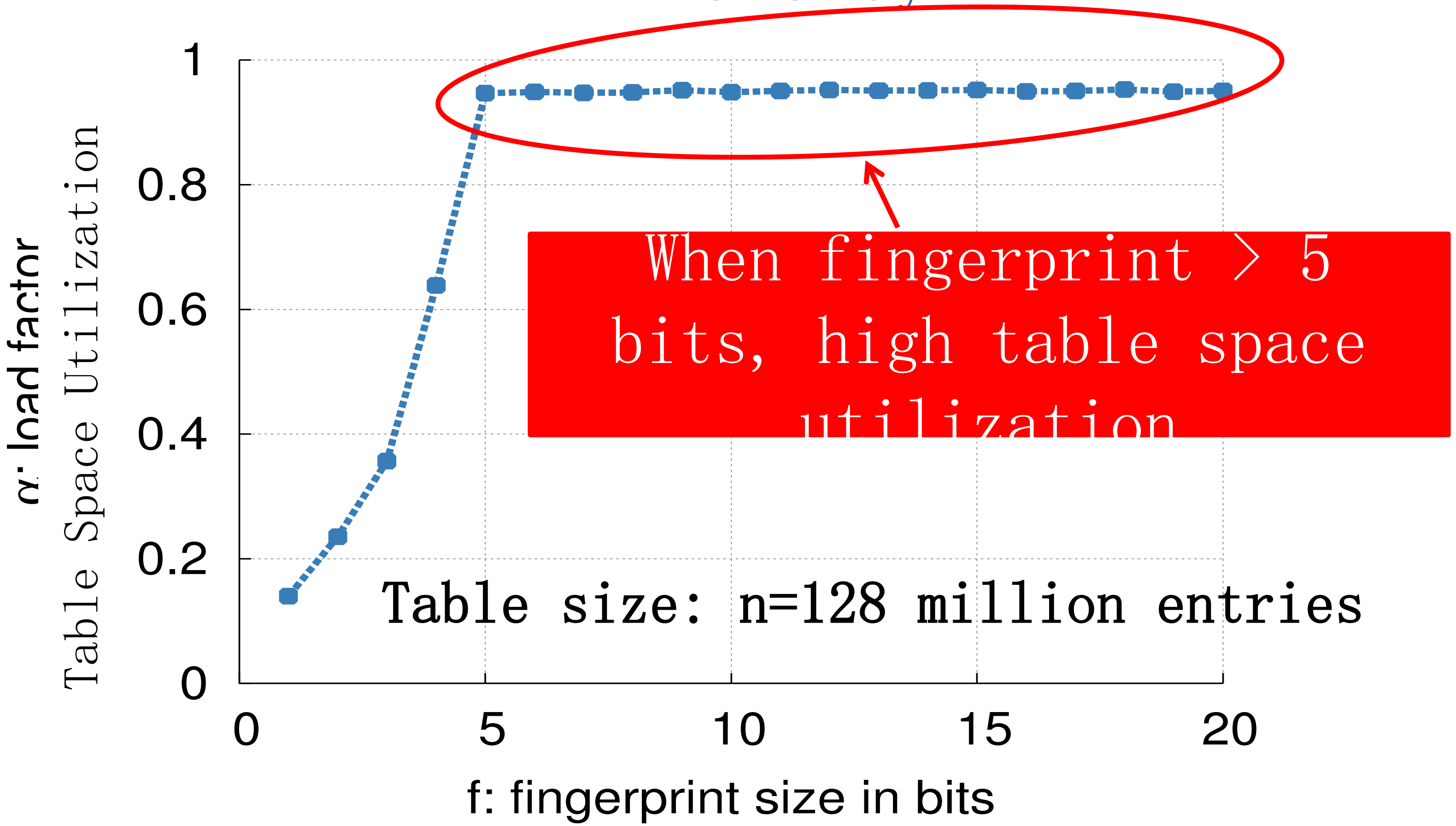
Partial Key Cuckoo Hashing

- Perform cuckoo hashing on fingerprints



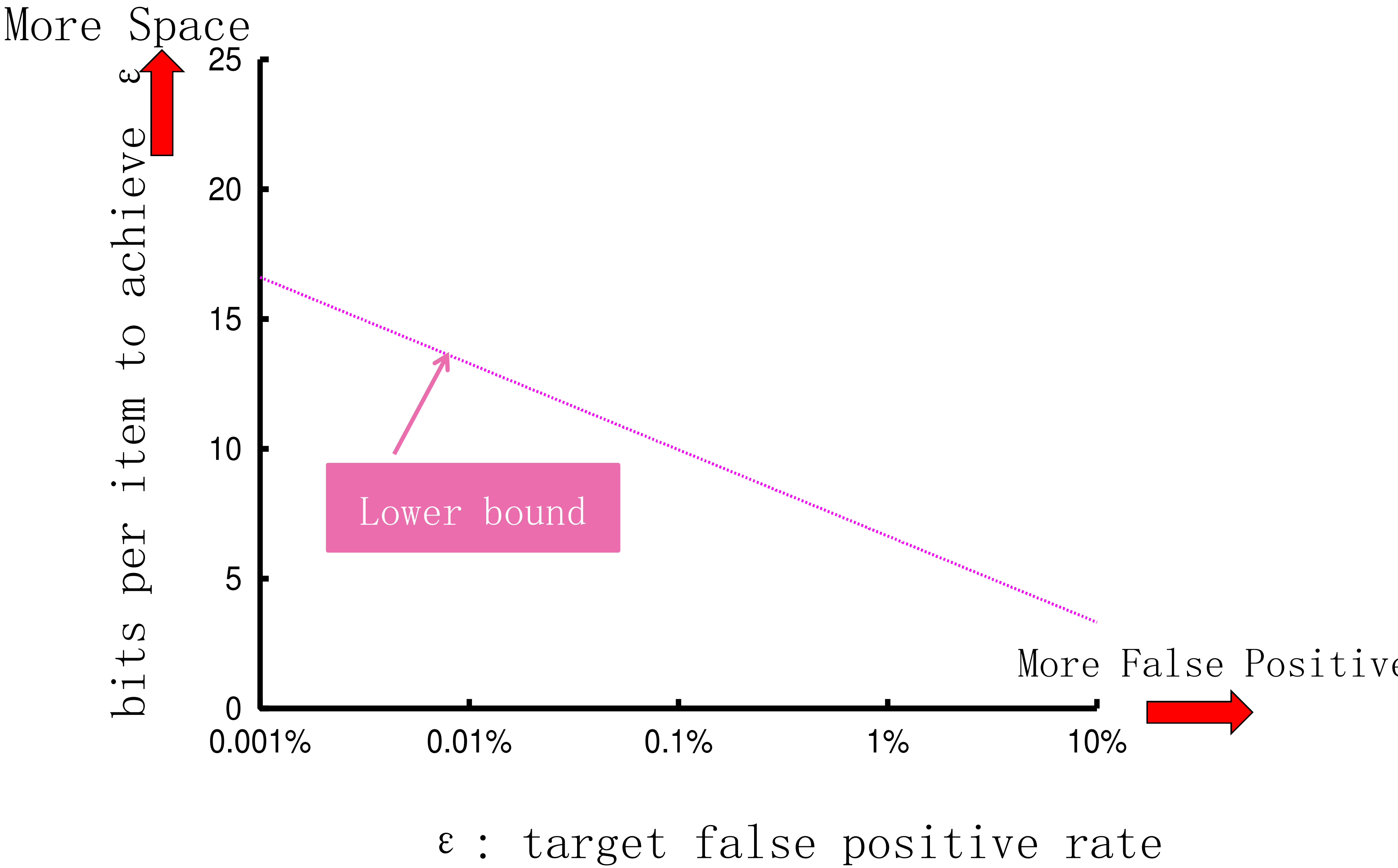
Can we still achieve high space utilization with partial-key cuckoo

Fingerprints Must Be "Long" for Space Efficiency

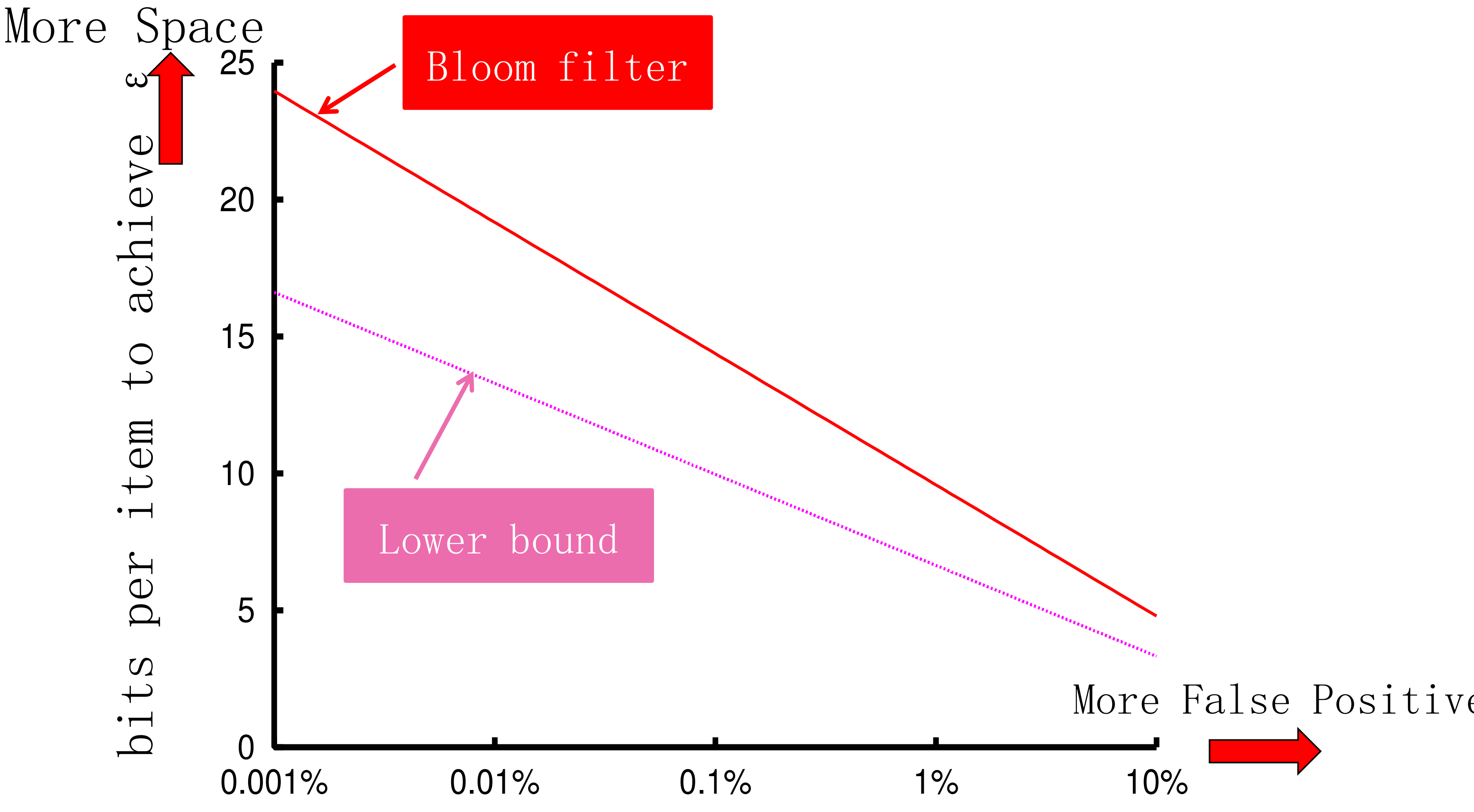


- Fingerprint must be Ω(logn/b) bits in theory
  - n: hash table size, b: bucket size
  - see more analysis in paper

Space Efficiency

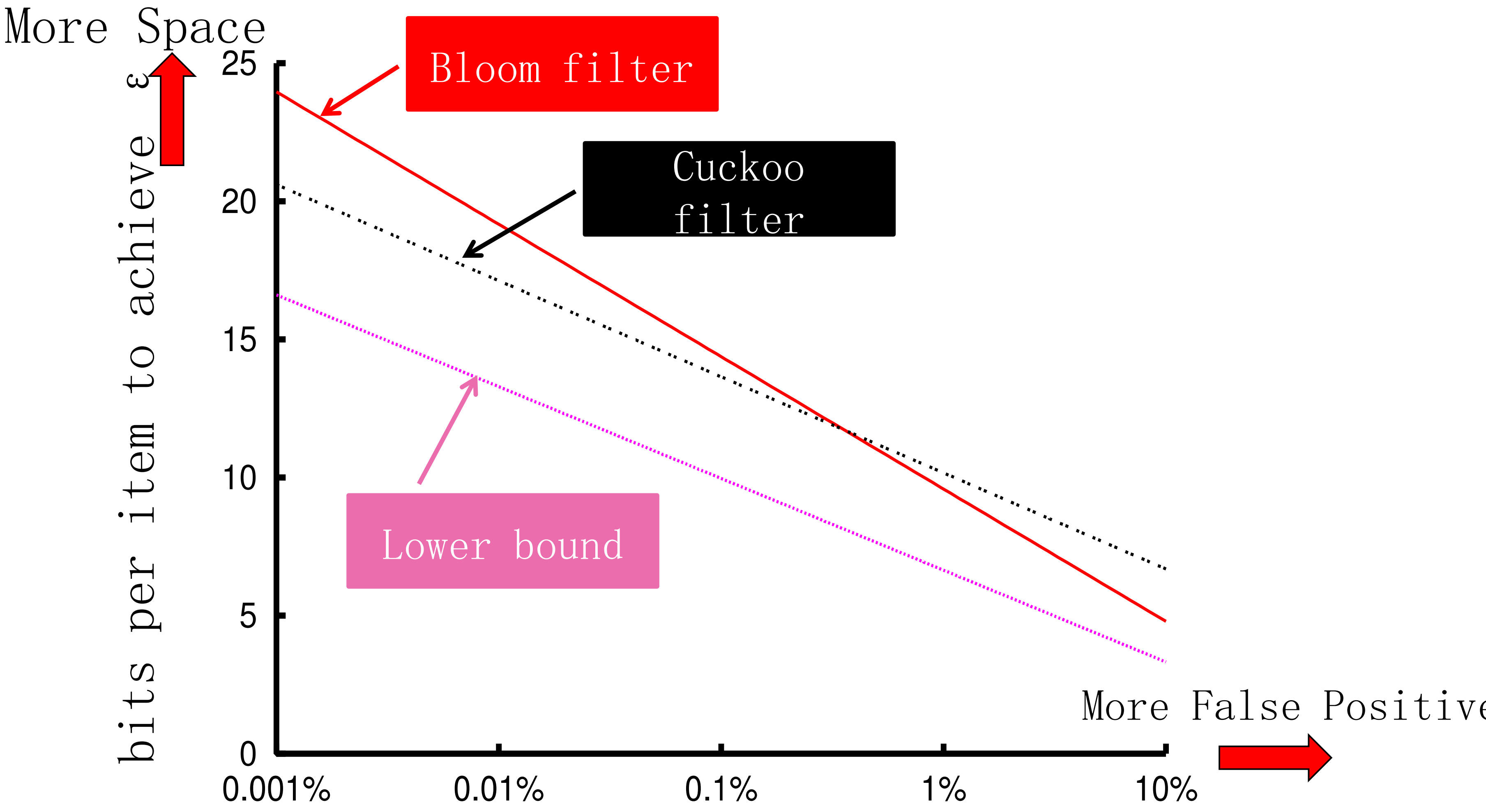


Space Efficiency



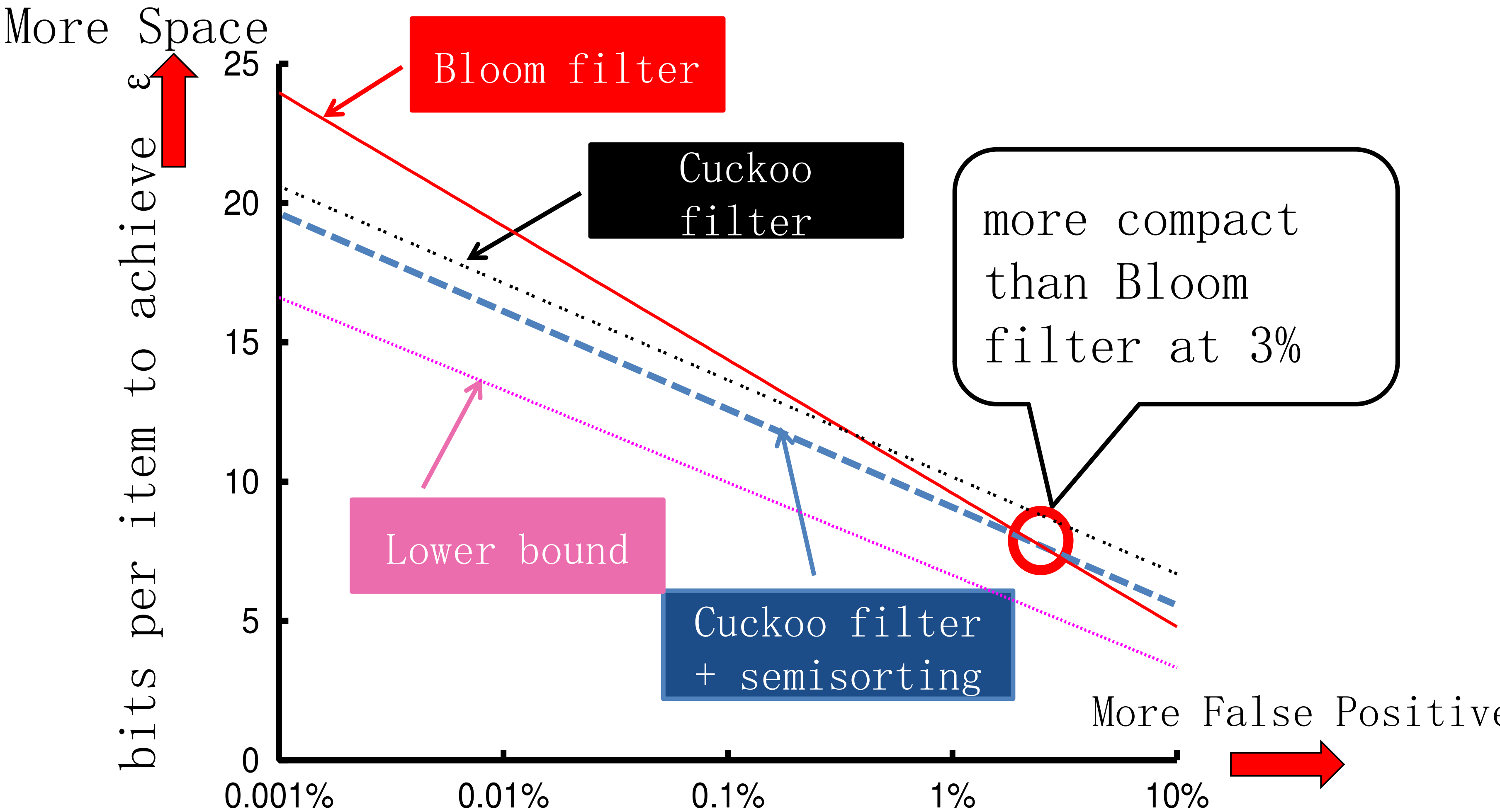
ε : target false positive rate

Space Efficiency



ε : target false positive rate

Space Efficiency



ε : target false positive rate

Outline

- Background
- Cuckoo filter algorithm
- Performance evaluation
- Summary

Evaluation

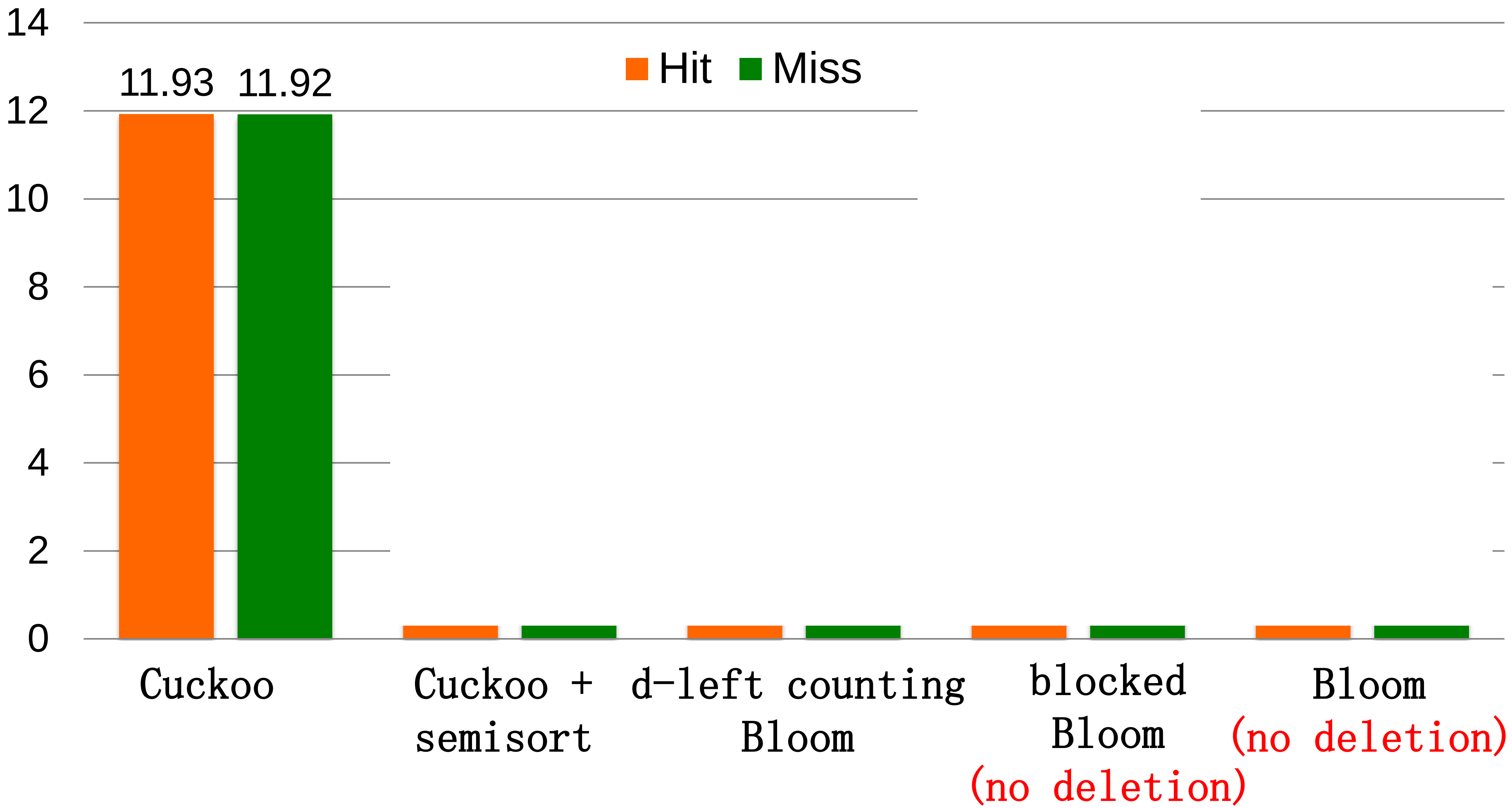
- Compare cuckoo filter with
  - Bloom filter (cannot delete)
  - Blocked Bloom filter [Putze2007] (cannot delete)
  - d-left counting Bloom filter [Bonomi2006]
  - Cuckoo filter + semisorting
  - More in the paper

- C++ implementation, single threaded

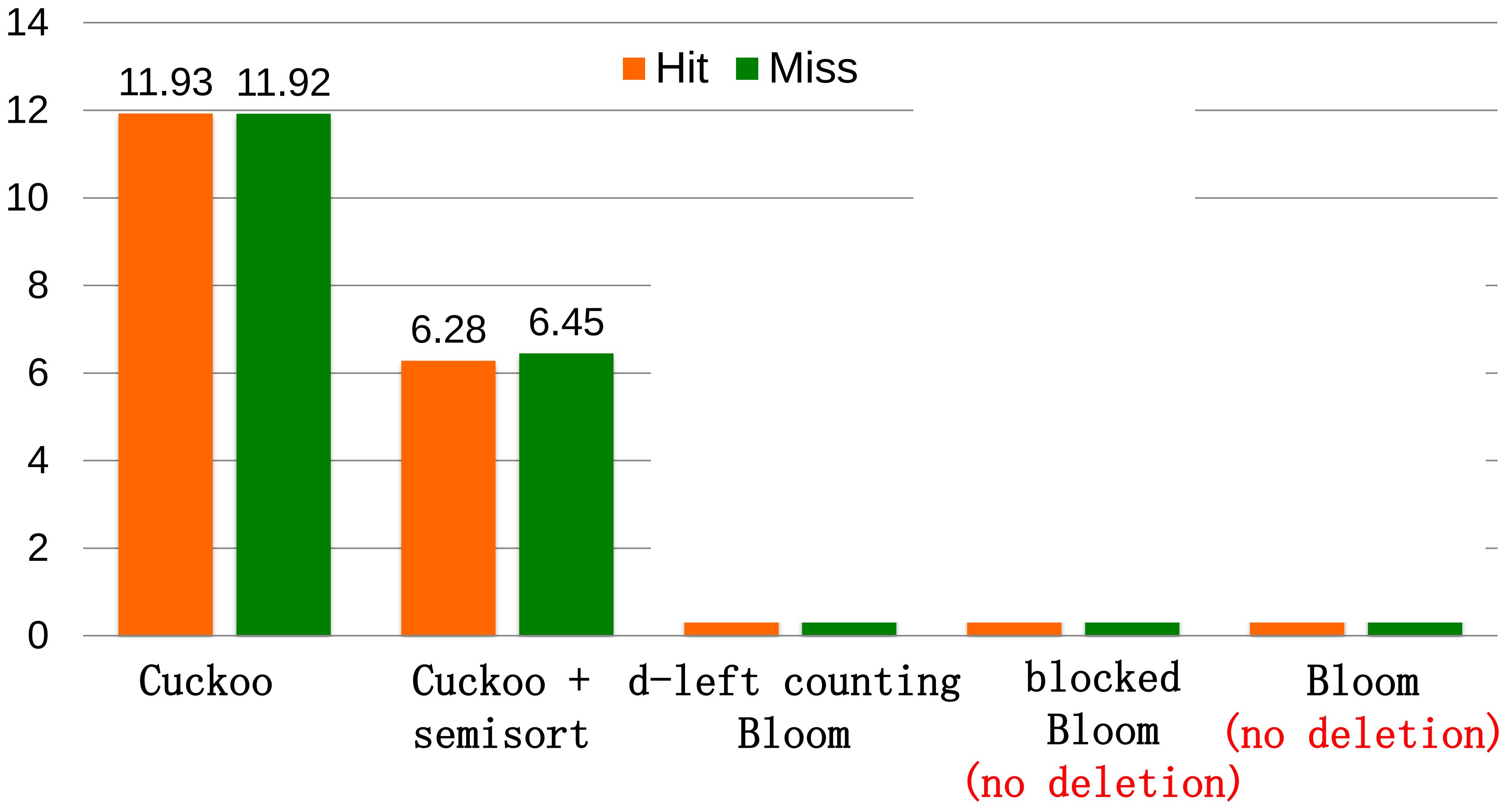
[Putze2007] Cache-, hash- and space- efficient bloom filters.

[Bonomi2006] Beyond Bloom filters: From approximate membership checks to approximate state machines.

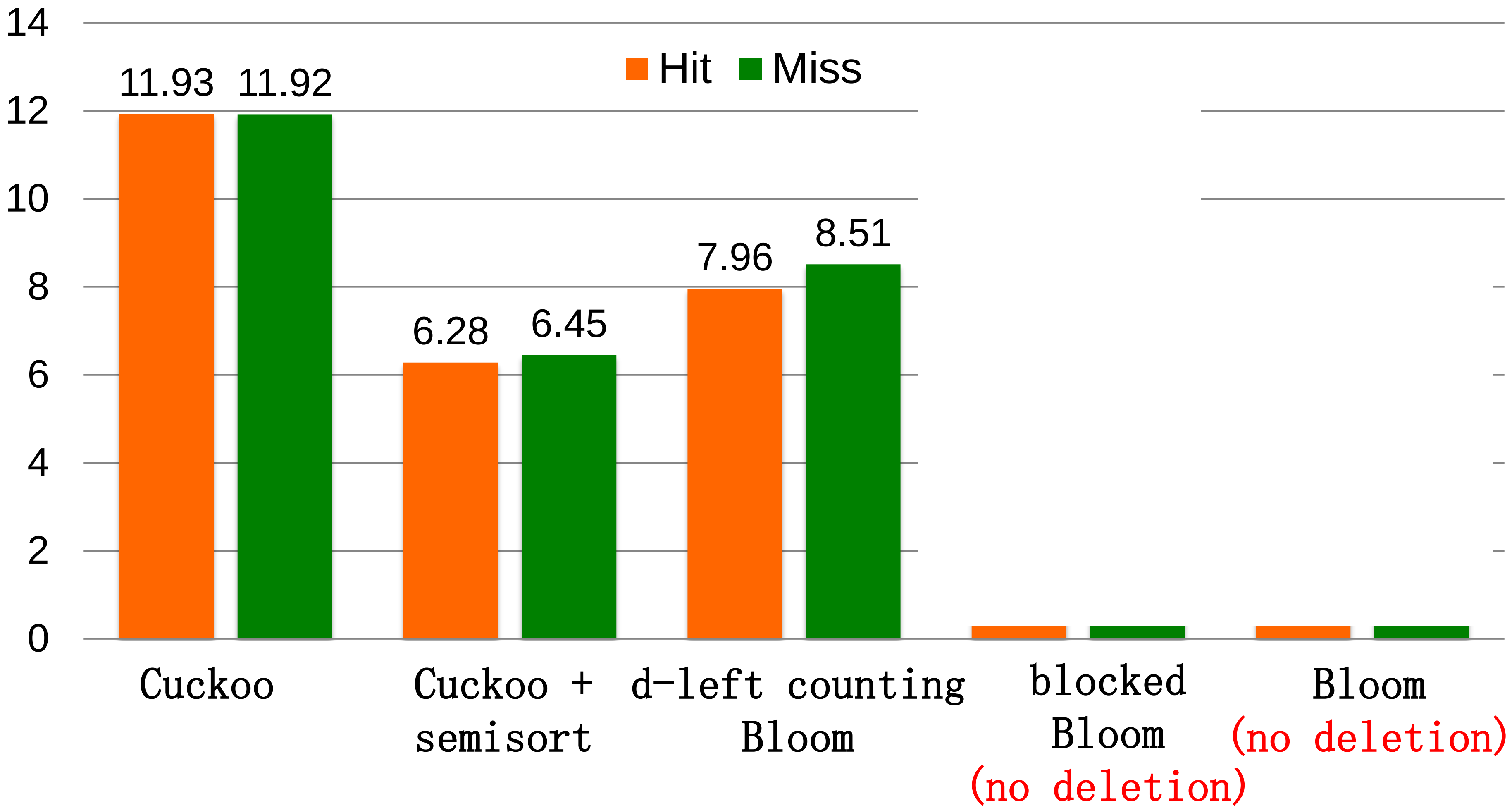
Lookup Performance (MOPS)



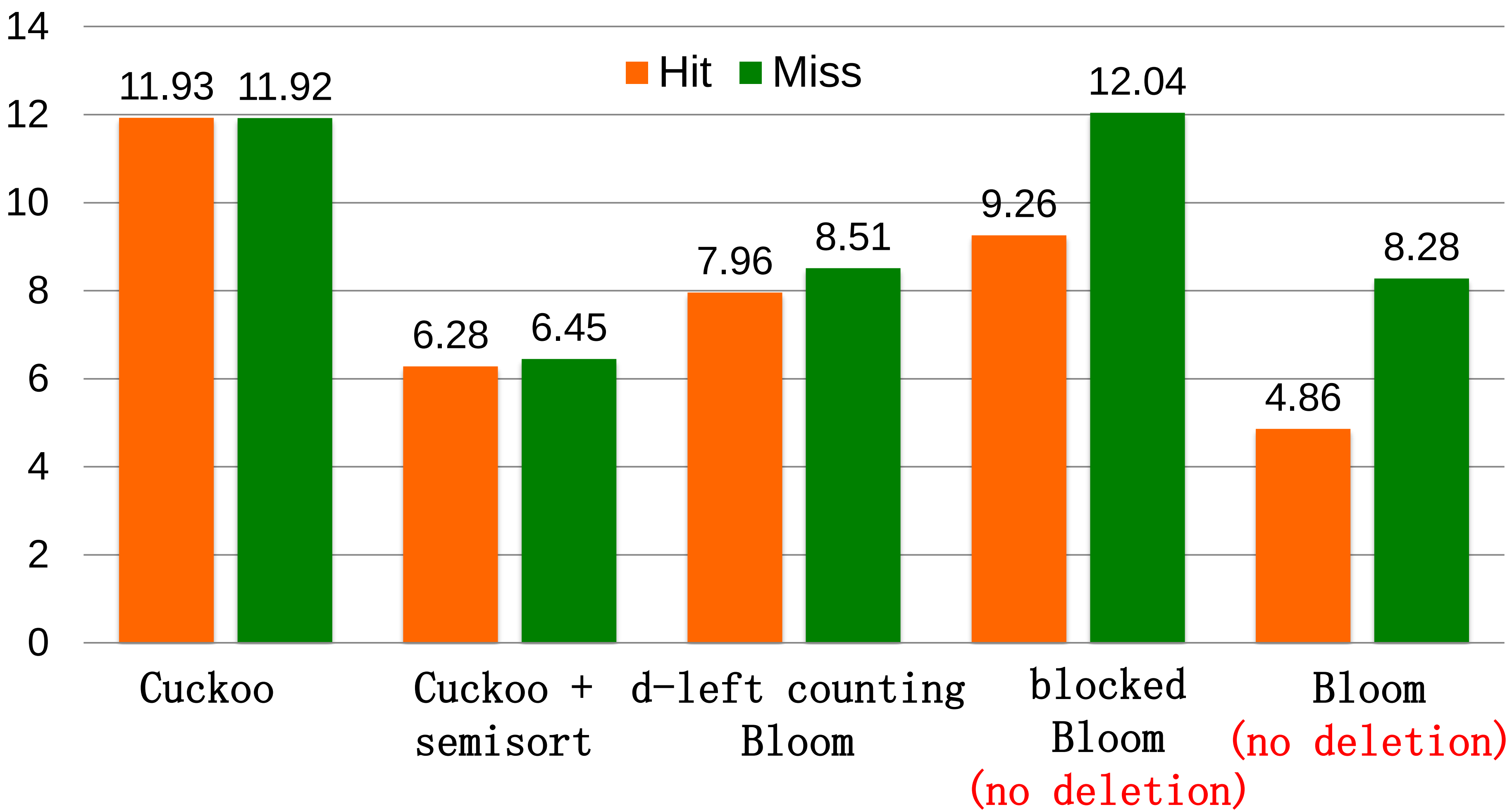
Lookup Performance (MOPS)



Lookup Performance (MOPS)

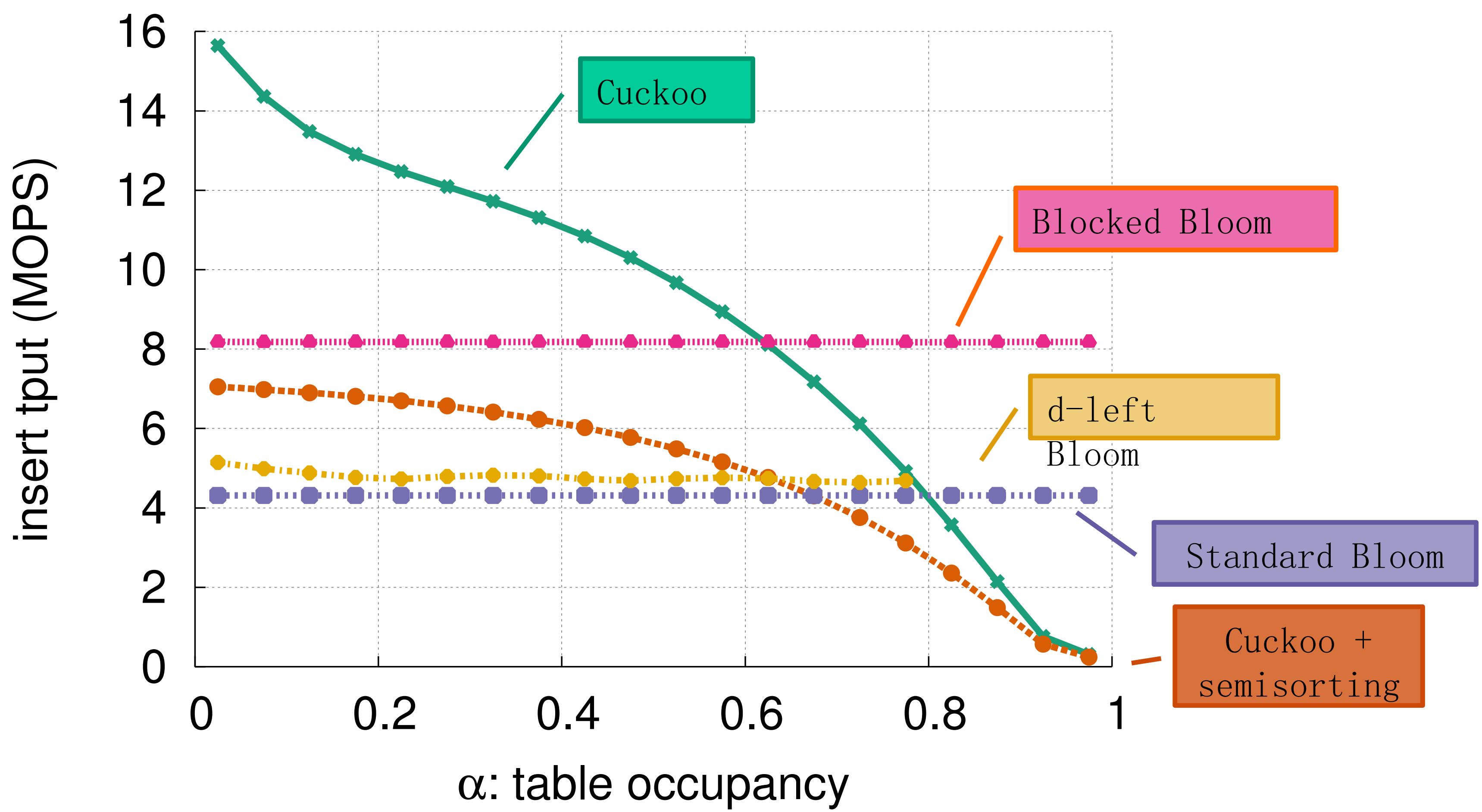


Lookup Performance (MOPS)



Cuckoo filter is among the fastest regardless

Insert Performance (MOPS)



Cuckoo filter has decreasing insert rate, but overall is only slower than blocked Bloom

Summary

- Cuckoo filter, a Bloom filter replacement:
  - Deletion support
  - High performance
  - Less Space than Bloom filters in practice
  - Easy to implement

- Source code available in C++:
  - https://github.com/efficient/cuckoofilter

Othello Hashing and Its Applications for Network Processing

Chen Qian

Department of Computer Engineering
qian@ucsc.edu
https://users.soe.ucsc.edu/~qian/

- Publications in *ICNP*'17, *SIGMETRICS*'17, *MECOMM*'17 and *Bioinformatics*

UNIVERSITY OF CALIFORNIA SANTA CRUZ

Background

- PhD in 2013 from UT Austin
  - with Simon Lam

- Research:
  - Computer networking
  - SDN/NFV
  - Internet of things
  - Security

Motivation

- Network algorithms always prefer small memory and fast processing speed.
  - Fast memory is precious resource on network devices
  - Needs to reach the line rate to avoid being a bottleneck, under large traffic volume
- More important in networks with layer-two semantics

Othello Hashing

- Essentially a key-value lookup structure
- Keys can be any names, addresses, identifiers, etc.
- Values should not be too long. At most 64 bit.

- For example
  - Key: network address; Value: link to forward a packet
  - Key: virtual IP; Value: direct IP

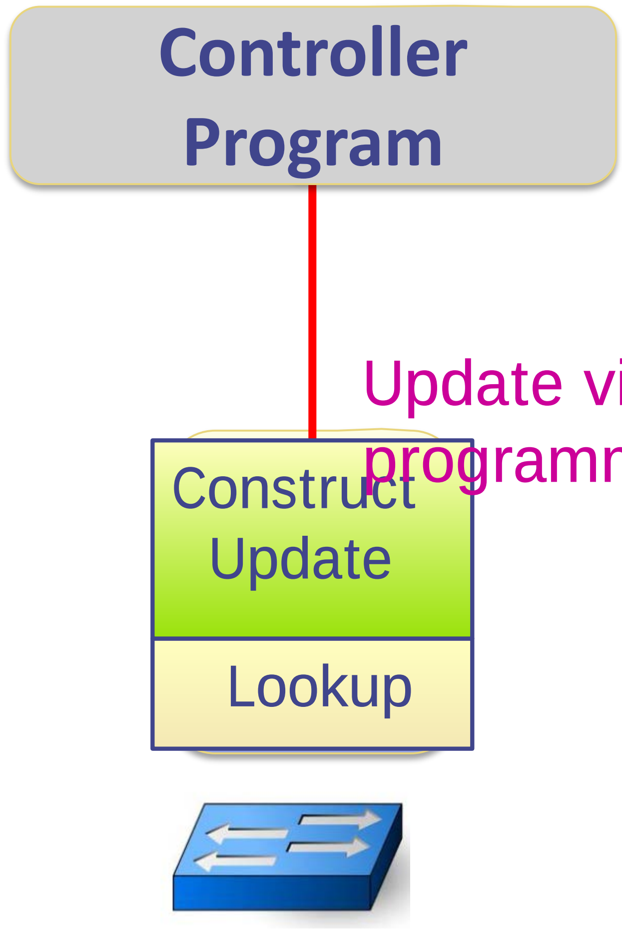
Why Othello is special

- Minimal query time: only two memory read operations (cachelines) per query.
- Minimal memory cost: 10%-30% of existing hash tables (e.g., Cuckoo).

Theoretical basis: **Minimal Perfect Hashing**

- Support dynamic updates: can be updated over a million times per second.

Idea of dynamic Othello lookups



Update via existing API of programmable networks

Optimize memory and query cost

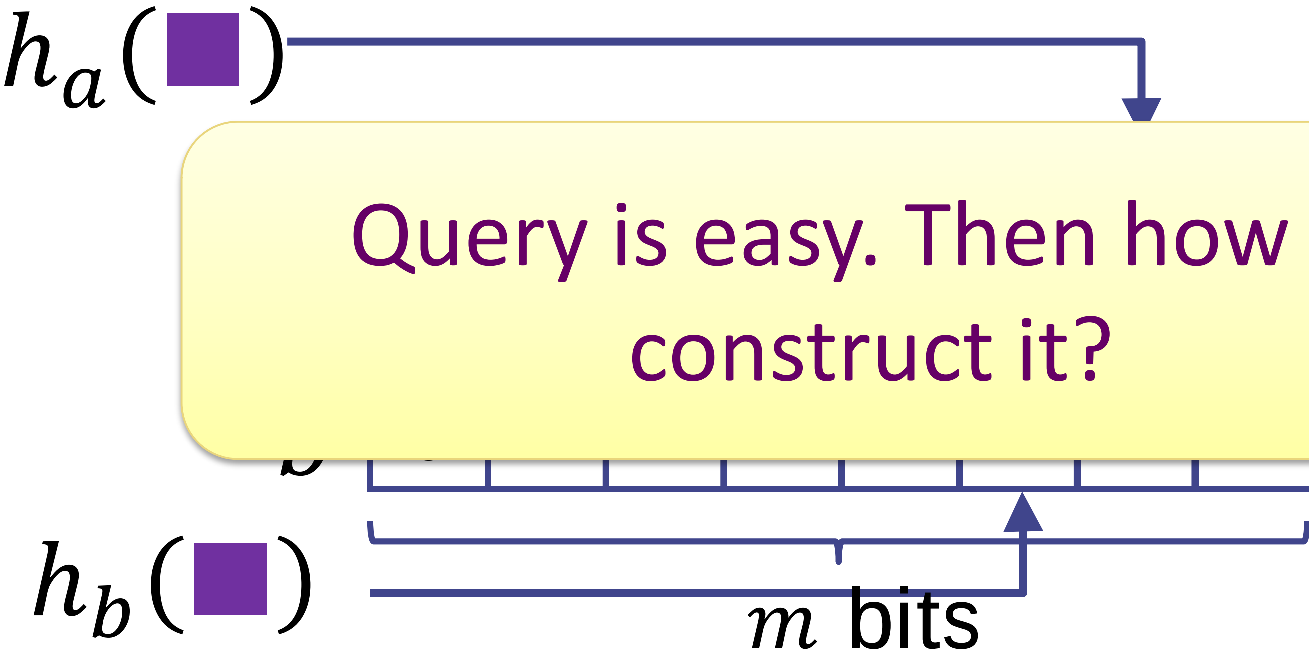
How Othello works

- **Basic version:** Classifies keys to **two sets** X and Y
  - Equivalent to key lookups for a 1-bit value
- Query result
  - $\tau(k) = 0 \Leftrightarrow k \in X$
  - $\tau(k) = 1 \Leftrightarrow k \in Y$
- **Advance version:** Classifies keys to 2^l sets
  - Equivalent to key lookups for a l-bit value

Othello Query Structure

n is # of keys

- Two bitmaps a, b with size m (m in $(1.33n, 2n)$)

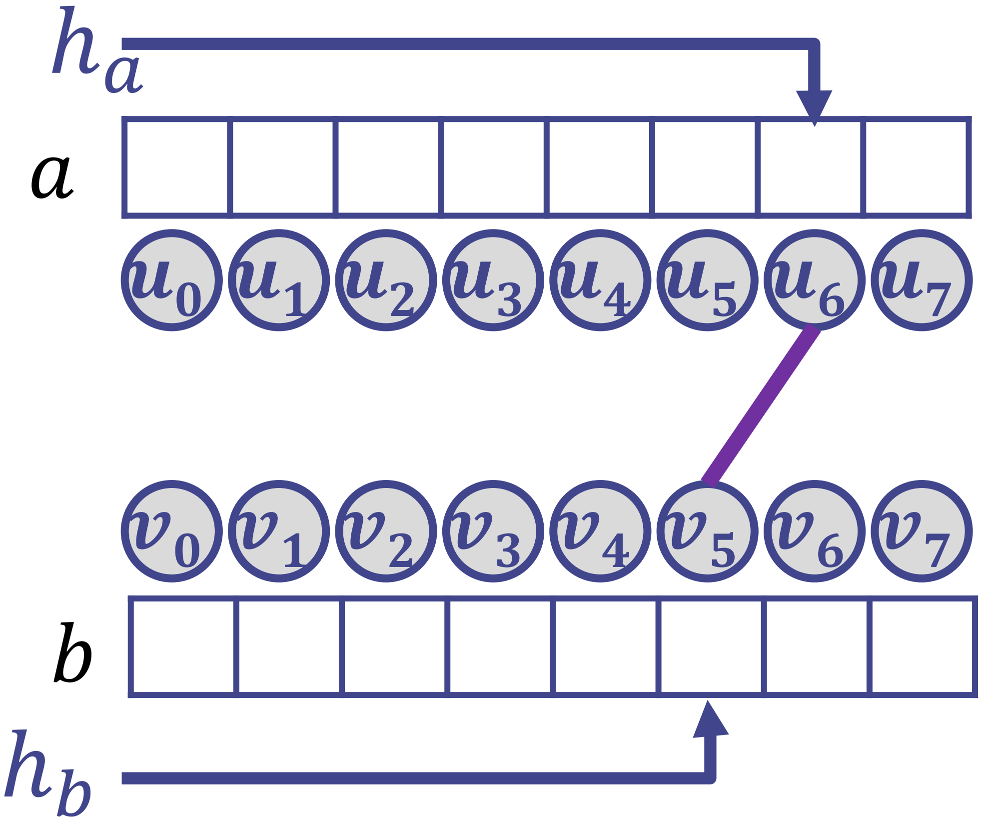


Query is easy. Then how to construct it?

■ is in set Y

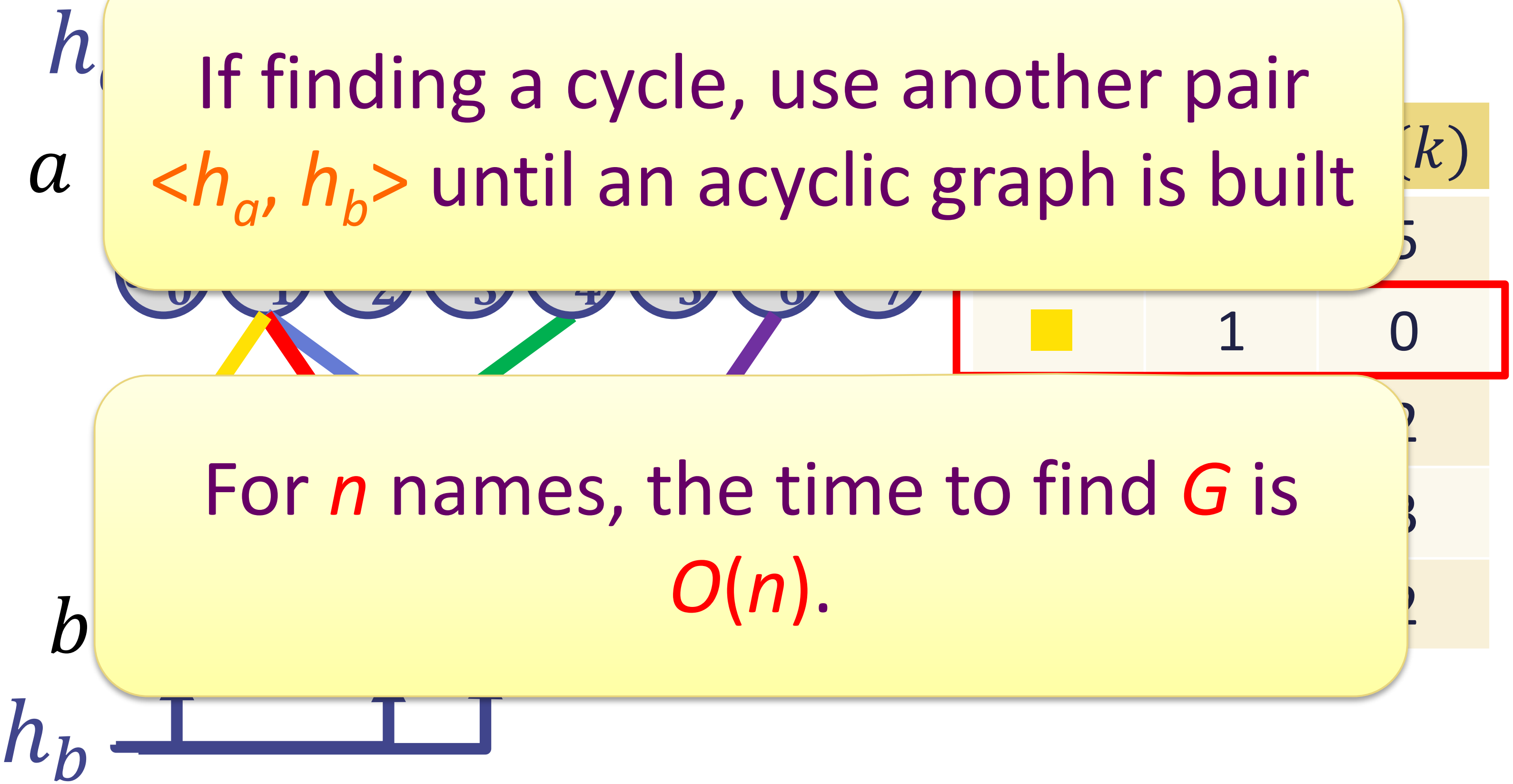
Othello Control Structure: Construct

- G: acyclic bipartite graph



k	$h_a(k)$	$h_b(k)$
■	6	5

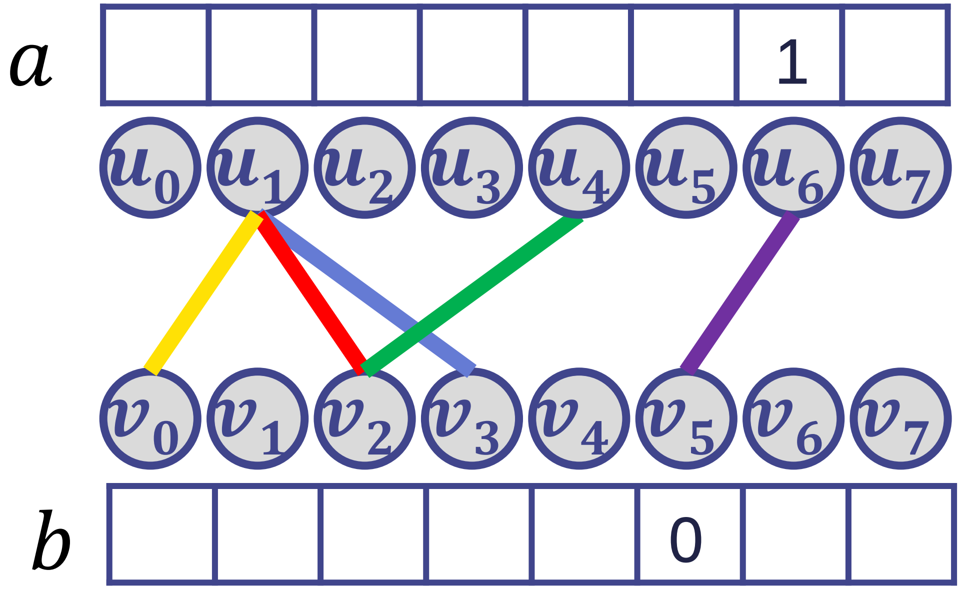
Othello Construct



If finding a cycle, use another pair $\langle h_a, h_b \rangle$ until an acyclic graph is built

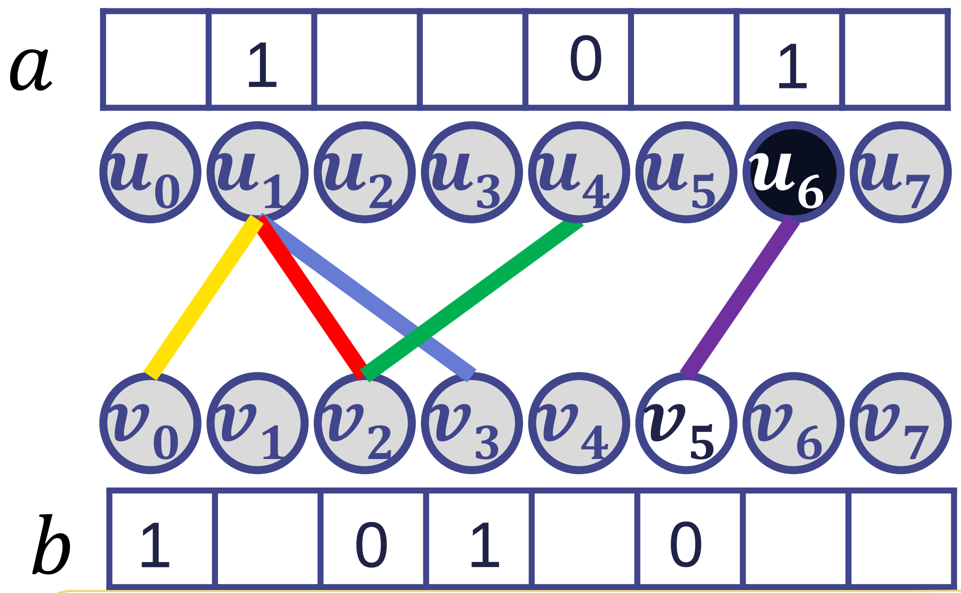
For n names, the time to find G is $O(n)$.

Compute Bitmap



k	$h_a(k)$	$h_b(k)$	set
■ (purple)	6	5	Y
■ (yellow)	1	0	
■ (red)	1	2	
■ (blue)	1	3	
■ (green)	4	2	

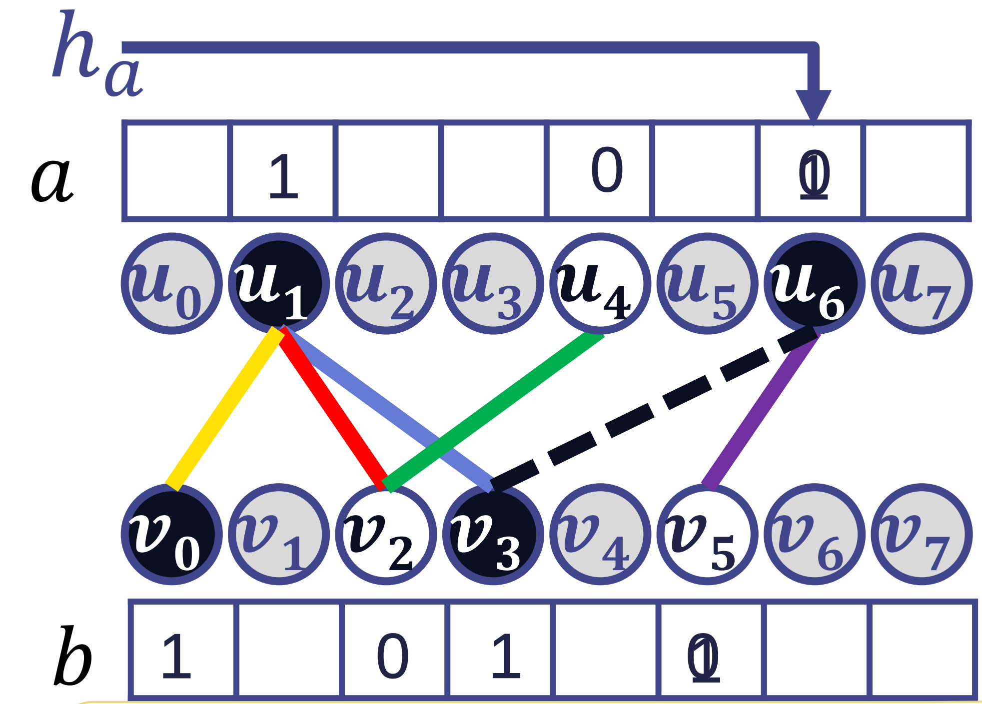
Compute Bitmap



k	$h_a(k)$	$h_b(k)$	set
(purple)	6	5	Y
(yellow)	1	0	X
(red)	1	2	Y
(blue)	1	3	X
(green)	4	2	X

If *G* is acyclic, easy to find a coloring plan

Name Addition – color flip

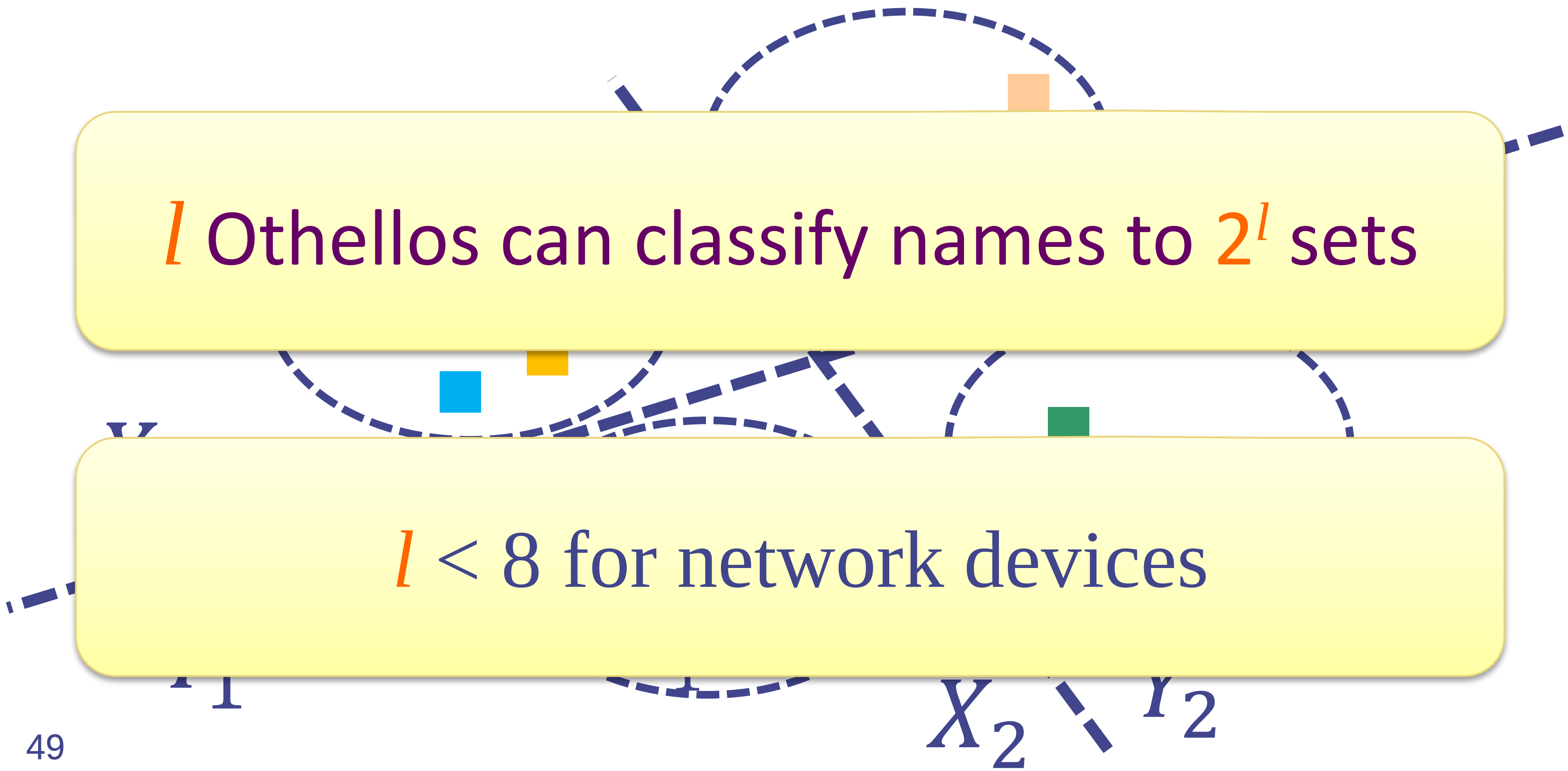


k	$h_a(k)$	$h_b(k)$	set
	6	5	Y
	1	0	X
	1	2	Y
	1	3	X
	4	2	X
	6	3	Y

If *G* is acyclic, flipping is trivial

L-Othello functionality

- Classifies names into 2^l sets: $Z_0, Z_1, \cdots, Z_{2^l-1}$

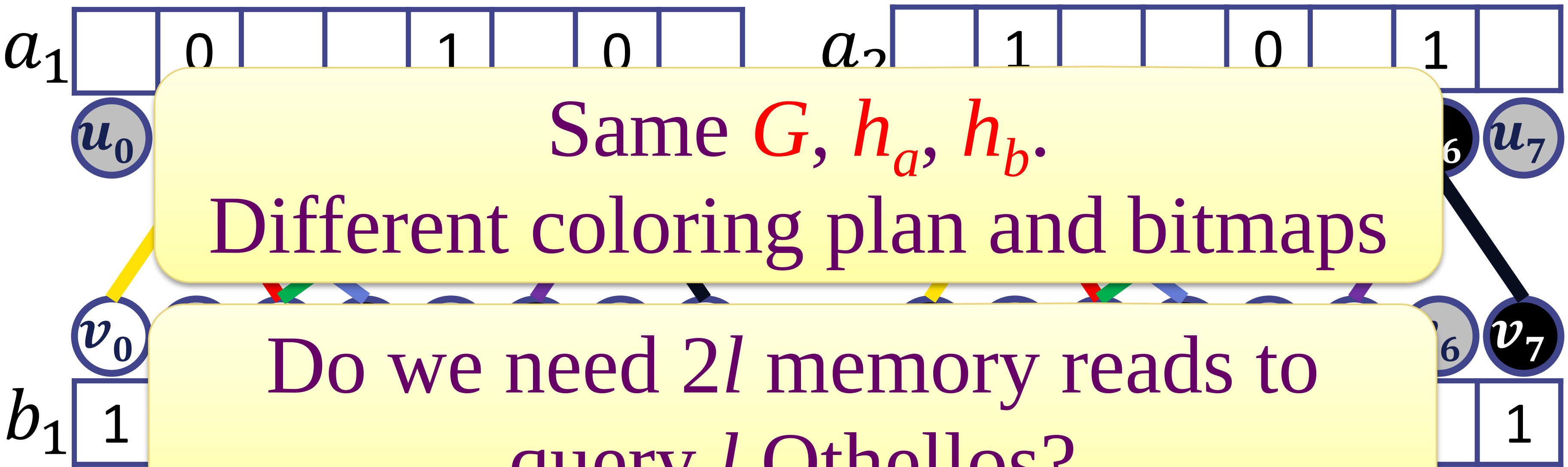


l Othellos can classify names to 2^l sets

$l < 8$ for network devices

Example

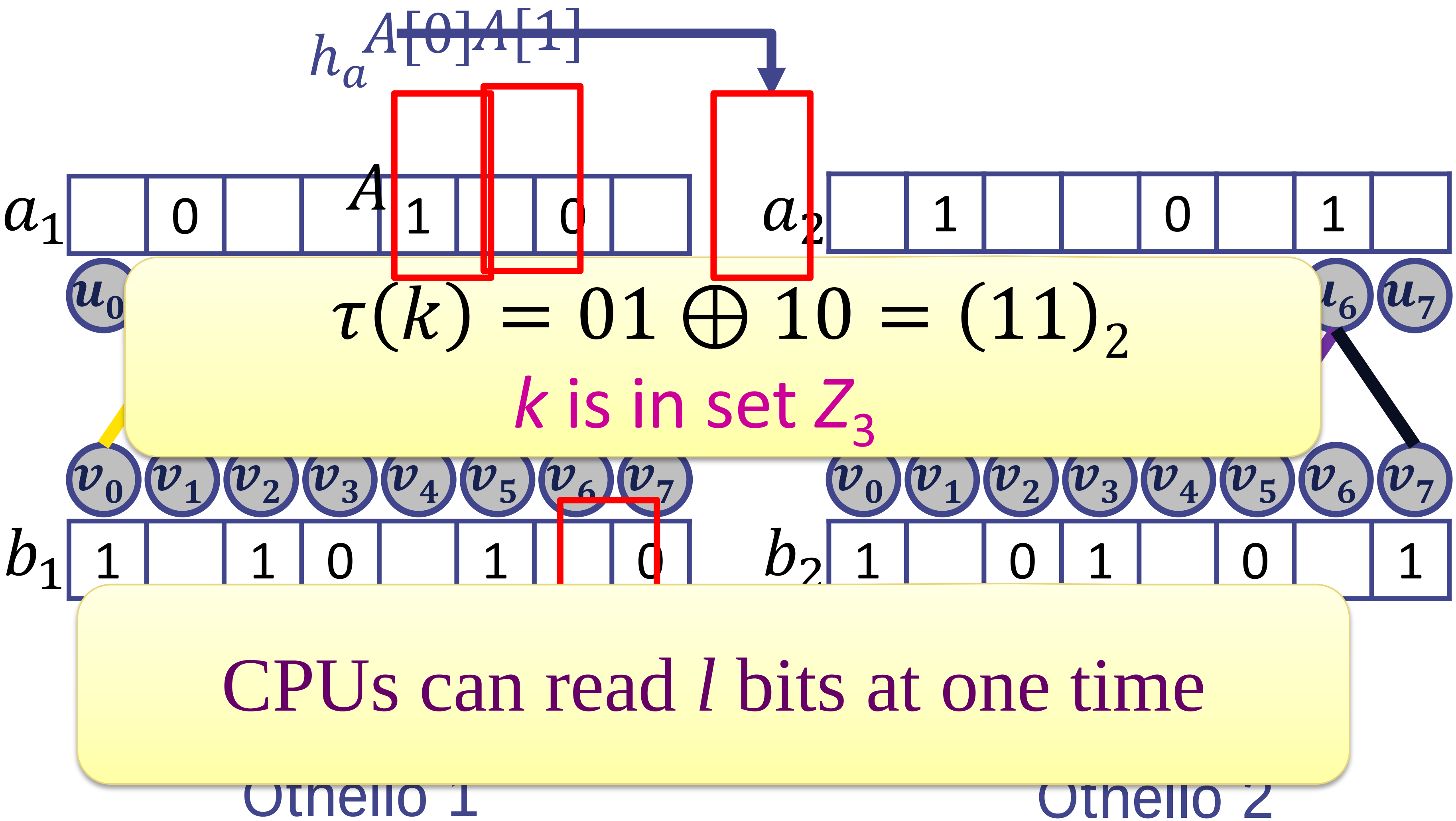
- Classify keys in 8 sets: $Z_0, Z_1, \cdots, Z_7$
- Orthogonal separation of sets
  - $X_3 = Z_0 \cup Z_1 \cup Z_2 \cup Z_3$; $Y_3 = Z_4 \cup Z_5 \cup Z_6 \cup Z_7$.
  - $X_2 = Z_0 \cup Z_1 \cup Z_4 \cup Z_5$; $Y_2 = Z_2 \cup Z_3 \cup Z_6 \cup Z_7$.
  - $X_1 = Z_0 \cup Z_2 \cup Z_4 \cup Z_6$; $Y_1 = Z_1 \cup Z_3 \cup Z_5 \cup Z_7$.
- **$6=(110)_2$** $k \in Y_3 \cap Y_2 \cap X_1 \Rightarrow k \in Z_6$
- l Othellos : classify keys in 2^l sets.



Same G, h_a, h_b.
Different coloring plan and bitmaps

Do we need $2l$ memory reads to query l Othellos?

Othello 1 Same $X \cup Y$ Othello 2



h_a
$A[0]$ $A[1]$
A
a_1
0
1
0
a_2
1
0
1
$\tau(k) = 01 \oplus 10 = (11)_2$
k is in set Z_3
v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7
v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7
b_1
1 1 0 1 0
b_2
1 0 1 0 1
CPUs can read *l* bits at one time
Othello 1
Othello 2

Alien keys

- What is $\tau(k) = a[h_a(k)] \oplus b[h_b(k)]$ when k is not in S?
  - An arbitrary value

- $\tau(k)$ return 1 with when
  - $a[i] = 1$ && $b[j] = 0$, or
  - $a[i] = 0$&& $b[j] = 1$

Applications of Othello

- 1. Forwarding Information Base (FIB)
- 2. Software load balancer
- 3. Data placement and lookup
- 4. Private queries
- 5. Genomic sequencing search
- And more…

A Concise FIB

- Resolving FIB explosion is crucial
  - For layer-two interconnected data centers
  - For OpenFlow-like fine-grained flow control
- Concise using *l*-Othello is a portable solution
  - In hardware devices
  - Or software switches

A Fast, Small, and Dynamic Forwarding Information Base, In ACM SIGMETRICS 2017
A Concise Forwarding Information Base for Scalable and Fast Name Switching, in IEEE ICNP 2017.

Network-wide updating

- If all devices share a same set of network names/addresses
  - Such as in layer-two Ethernet-based data centers
  - All Othellos will share a same *G*.
  - Hence network-wide updating is very efficient!
- Update consistency also provided

Implementation of three prototypes

- 1. Memory mode
  - Query and control structures running on different threads.
- 2. CLICK modular router
- 3. Intel Data Plane Development Kit (DPDK)

Comparison:

- Buffalo

Yu, Fabrikant, Rexford, in CoNEXT'09

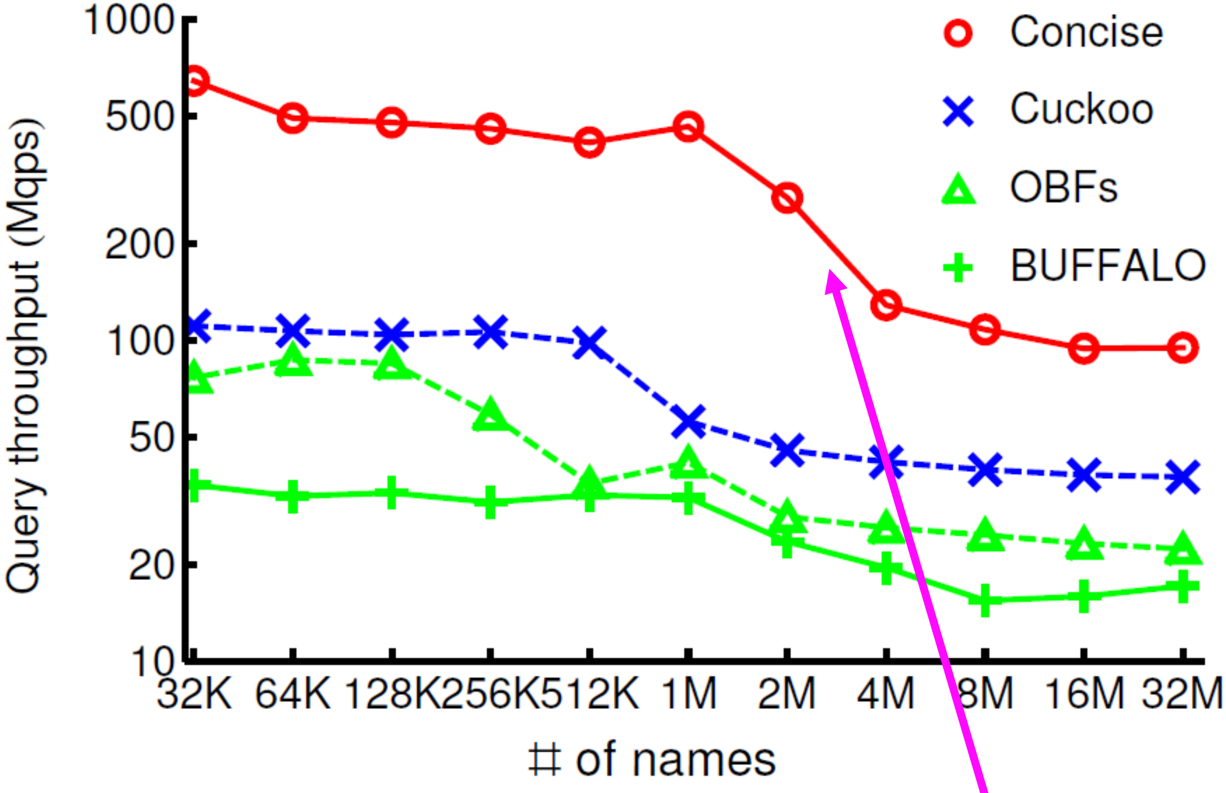
- Cuckoo hashing

Zhou, Fan, Lim, Kaminsky, Andersen,
in CoNEXT'13 and SIGCOMM'15

Comparison: Memory size

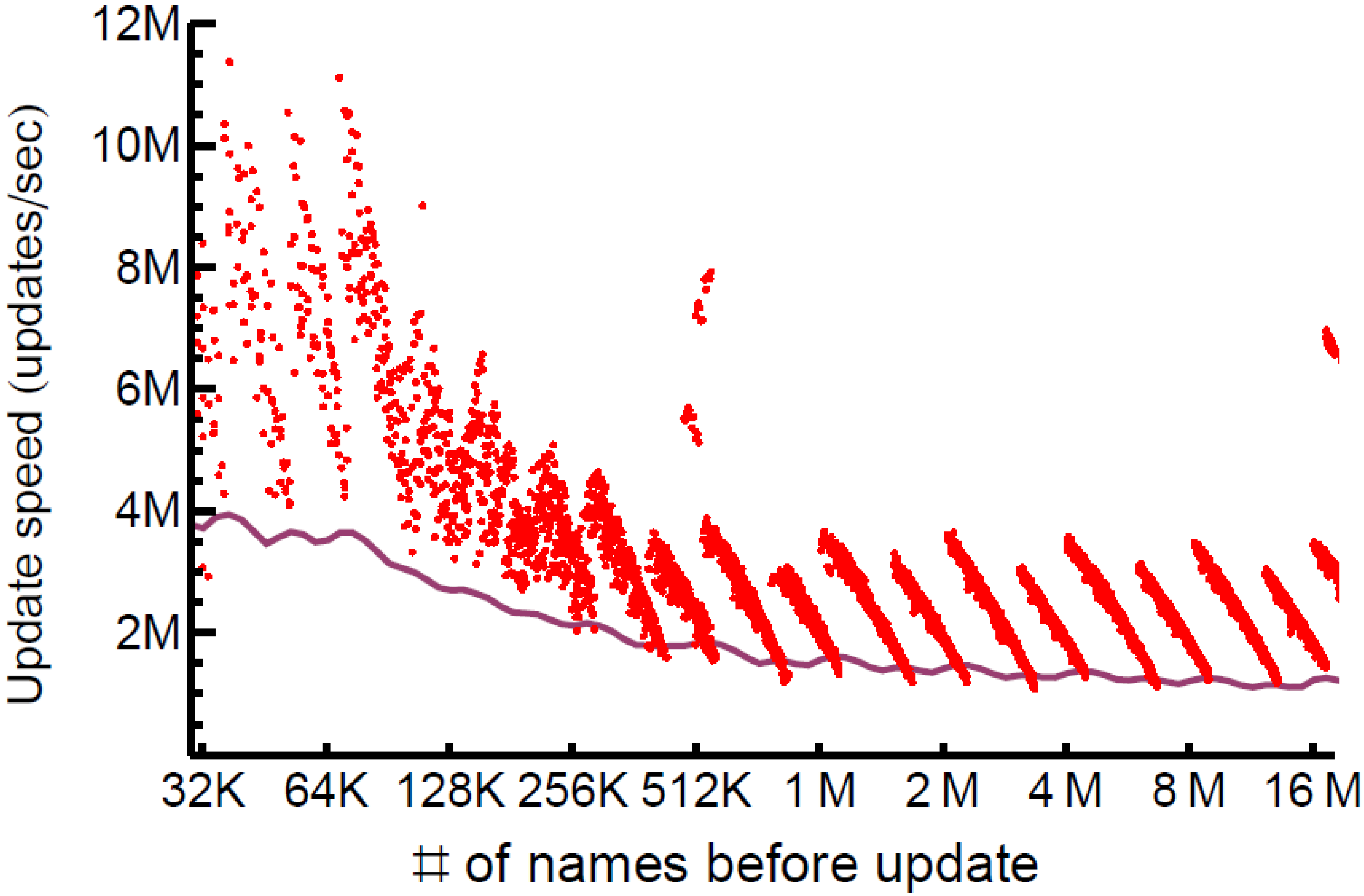
FIB Example			Memory Size		
Name Type	# Names	# Actions	Concise	Cuckoo	Buffalo
MAC (48 bits)	$7*10^5$	16	1M	5.62M	2.64M
MAC (48 bits)	$5*10^6$	256	16M	40.15M	27.70M
MAC (48 bits)	$3*10^7$	256	128M	321.23M	166.23M
IPv4 (32 bits)	$1*10^6$	16	2M	4.27M	3.77M
IPv6 (128 bits)	$2*10^6$	256	8M	34.13M	11.08M
OpenFlow (356 bits)	$3*10^5$	256	1M	14.46M	1.67M
OpenFlow (356 bits)	$1.4*10^6$	65536	8M	67.46M	18.21M
File name (varied)	359194	16	512K	19.32M	1.35M

Query speed



2x to 4x speed advantage

Update speed



For unknown network names

- 1. For data centers with most internal traffic
  - Such situation is rare
- 2. For networks with much incoming traffic
  - A filter can be installed at a firewall
- 3. Concise may include an *r*-bit checksum.
  - A lookup still requires 2 memory accesses in total, as long as $l + r <= 64$.

Thank You

Chen Qian

cqian12@ucsc.edu

https://users.soe.ucsc.edu/~qian/