Cuckoo Filter: Practically Better Than Bloom

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

CONEXT
nference

Carnegie Mellon
Parallel Data Laboratory

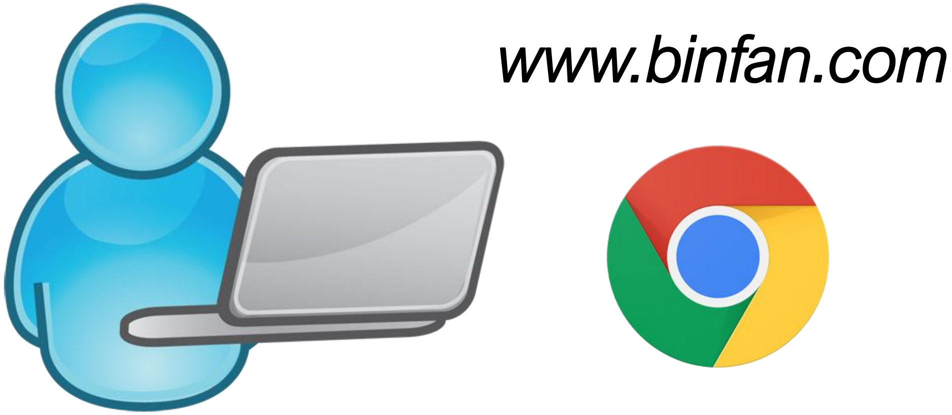
What is Bloom Filter? A Compact Data Structure Storing Set-membership

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

false positive rate

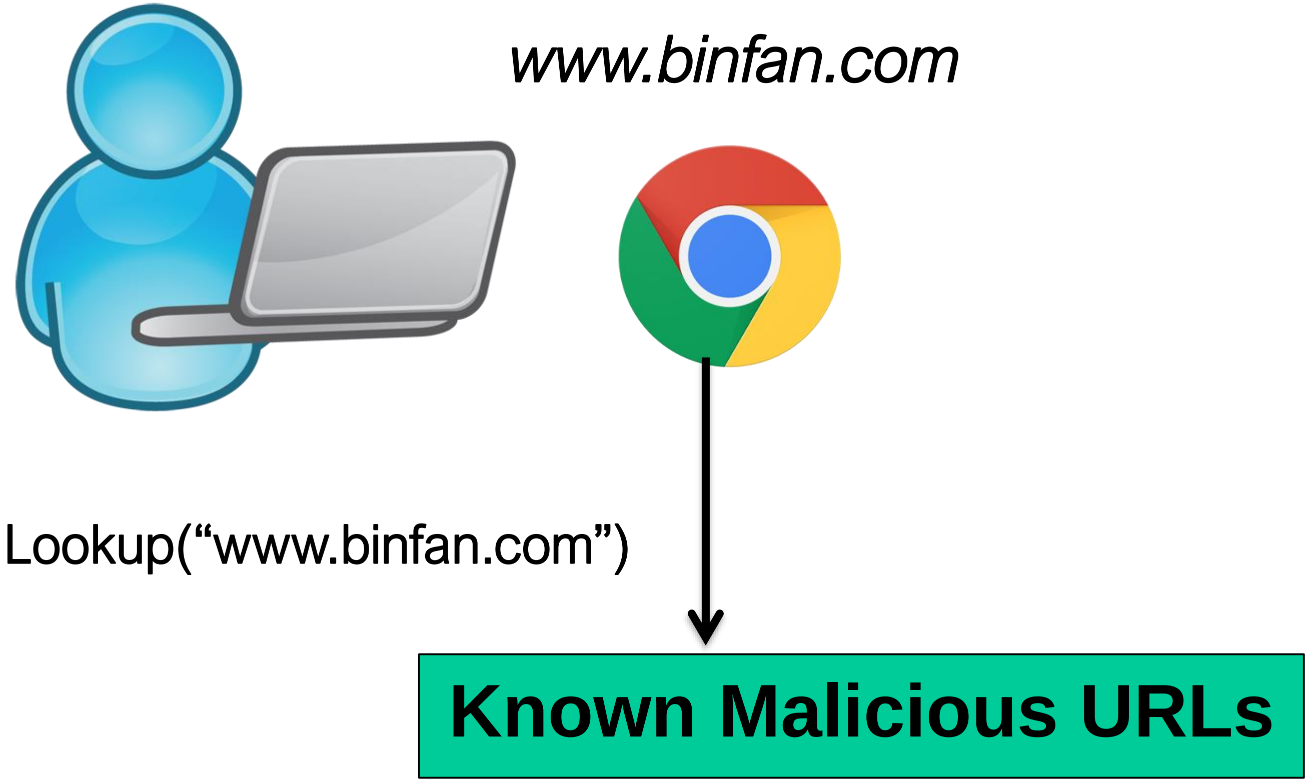
- Benefit: not always precise but highly compact
  - Typically a few bits per item
  - Achieving lower ***ε*** (more accurate) requires spending more bits per item

Example Use: Safe Browsing



www.binfan.com

Example Use: Safe Browsing



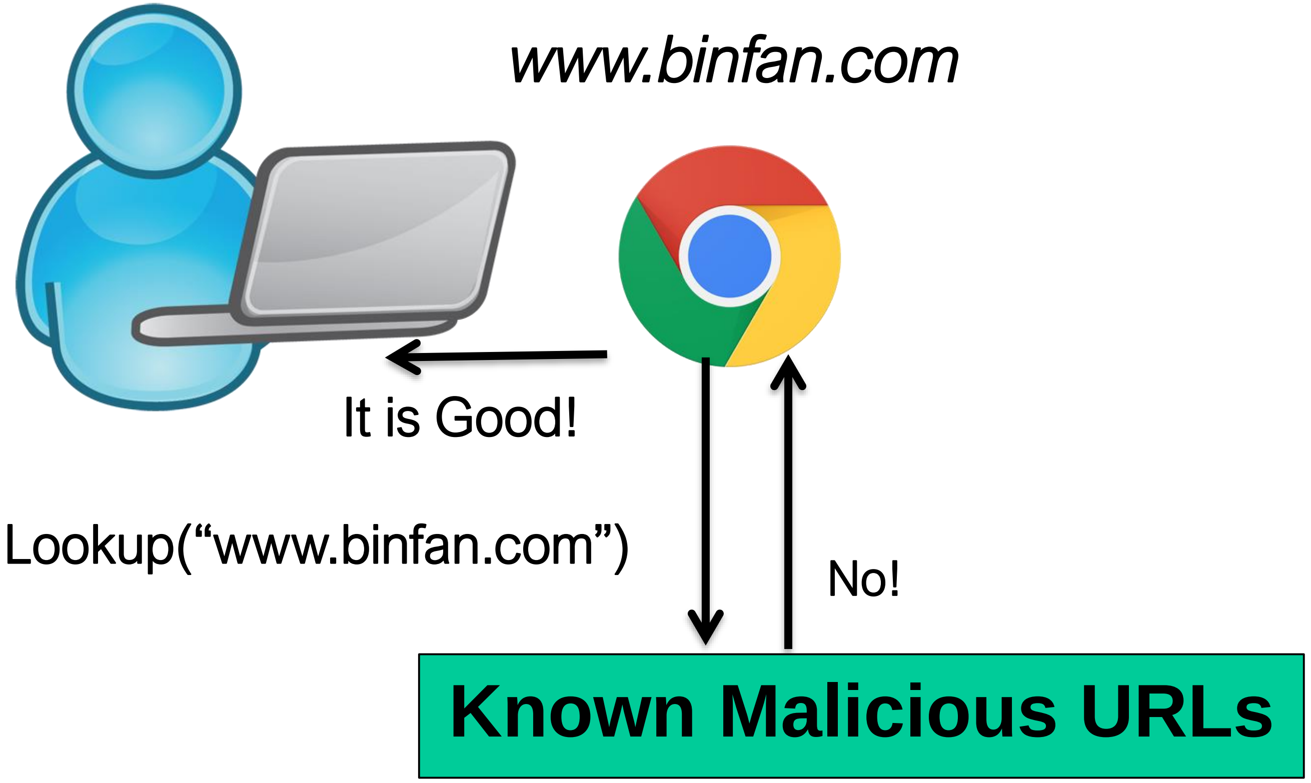
www.binfan.com

Lookup("www.binfan.com")

Known Malicious URLs

Stored in Bloom Filter

Example Use: Safe Browsing



www.binfan.com

It is Good!

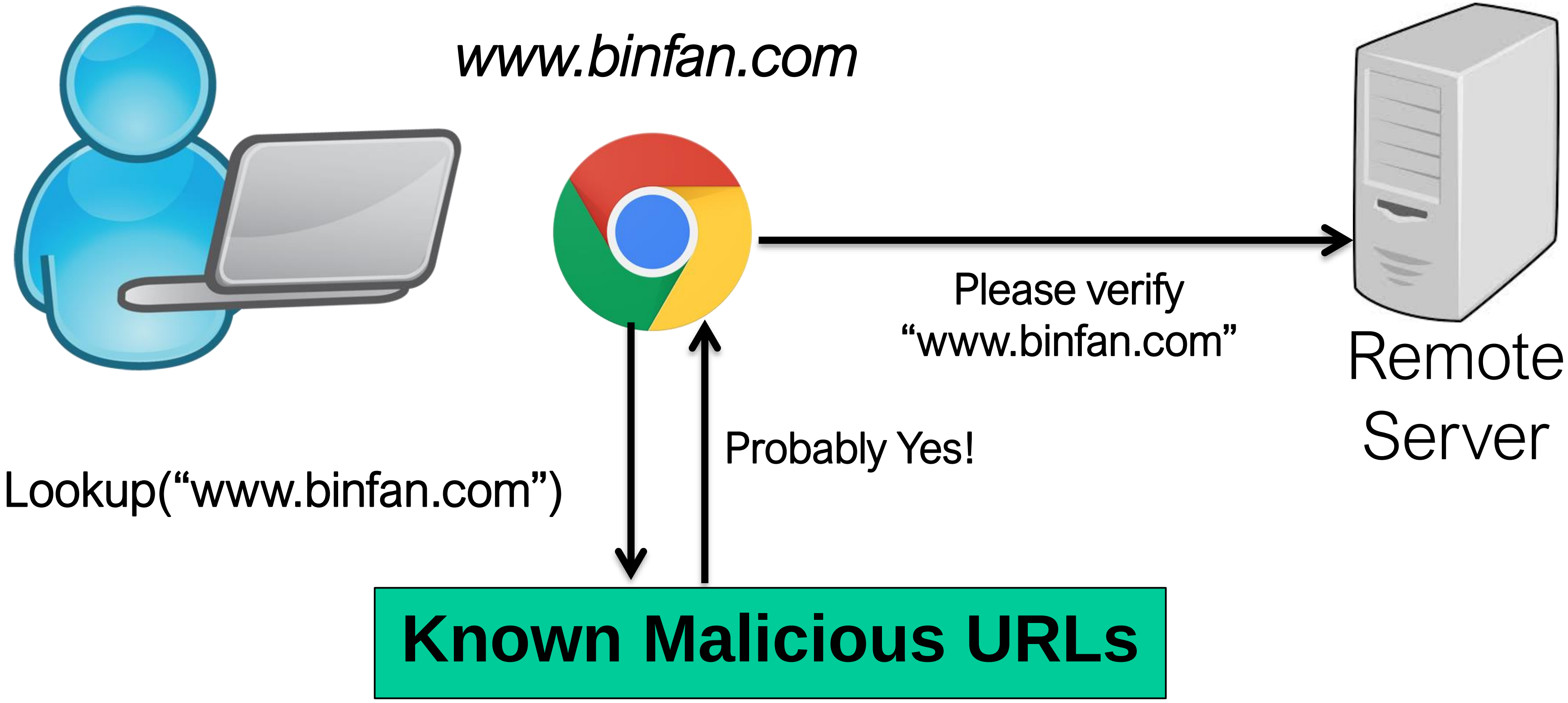
Lookup(“www.binfan.com”)

No!

Known Malicious URLs

Stored in Bloom Filter

Example Use: Safe Browsing



www.binfan.com

Lookup("www.binfan.com")

Probably Yes!

Please verify
"www.binfan.com"

Remote
Server

Known Malicious URLs

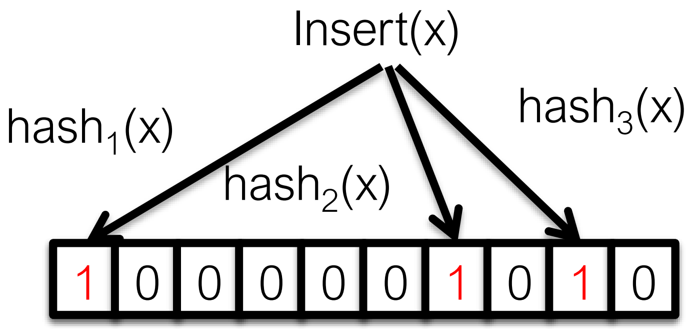
Stored in Bloom Filter

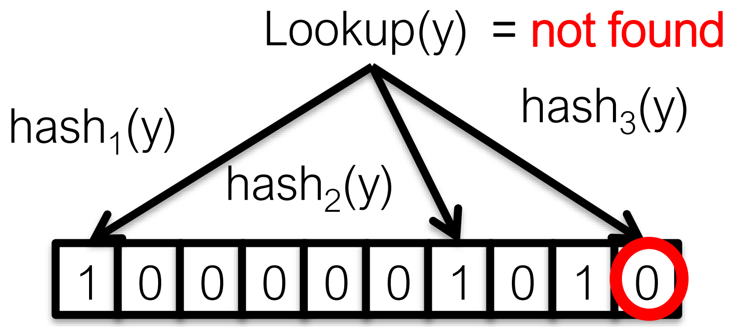
**Scale to
millions URLs**

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	✗	✓	✓

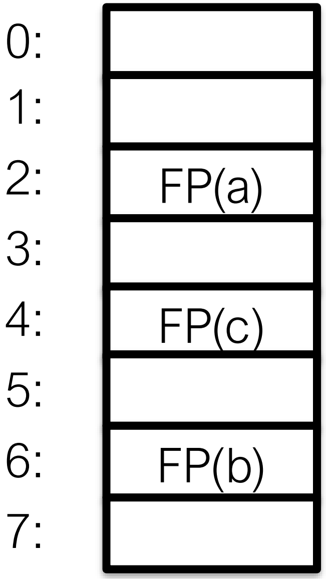
Can we achieve all three in practice?

Outline

- Background
- Cuckoo filter algorithm
- Performance evaluation
- Summary

Basic Idea: Store Fingerprints in Hash Table

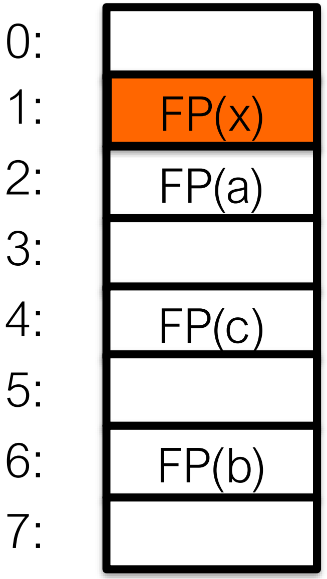
- **Fingerprint**(x): A hash value of x
  - Lower false positive rate ε, longer fingerprint



0:
1:
2: FP(a)
3:
4: FP(c)
5:
6: FP(b)
7:

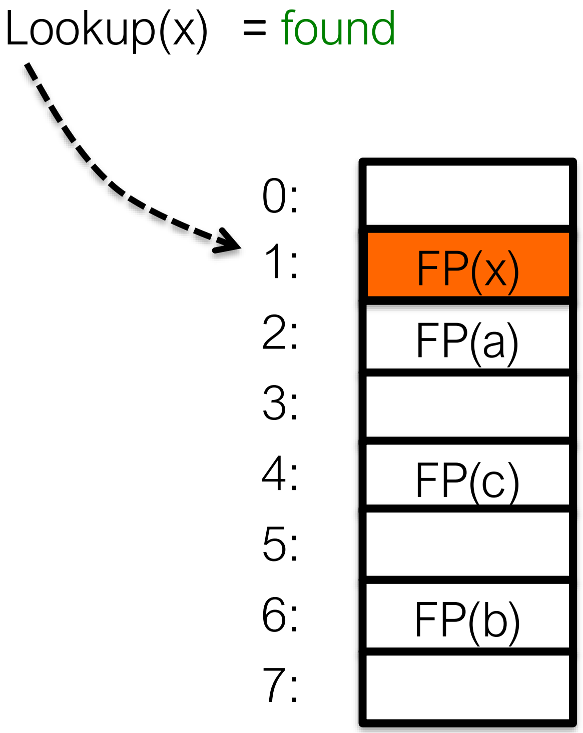
Basic Idea: Store Fingerprints in Hash Table

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



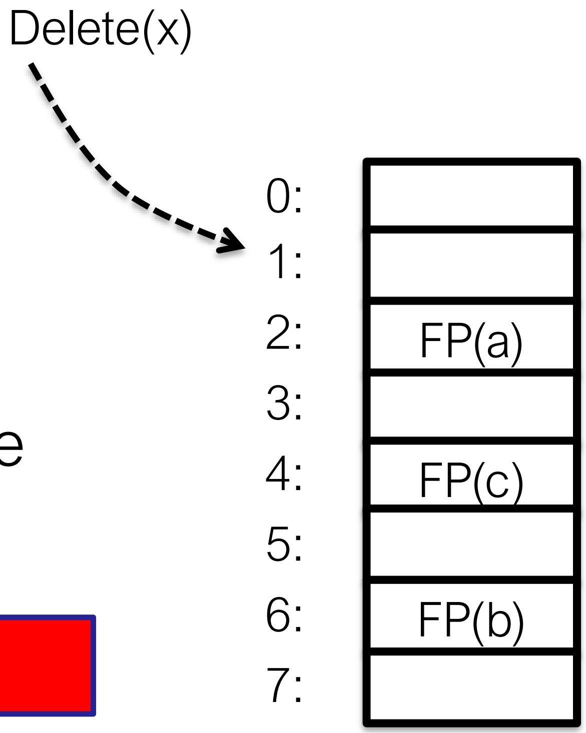
Basic Idea: Store Fingerprints in Hash Table

- **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

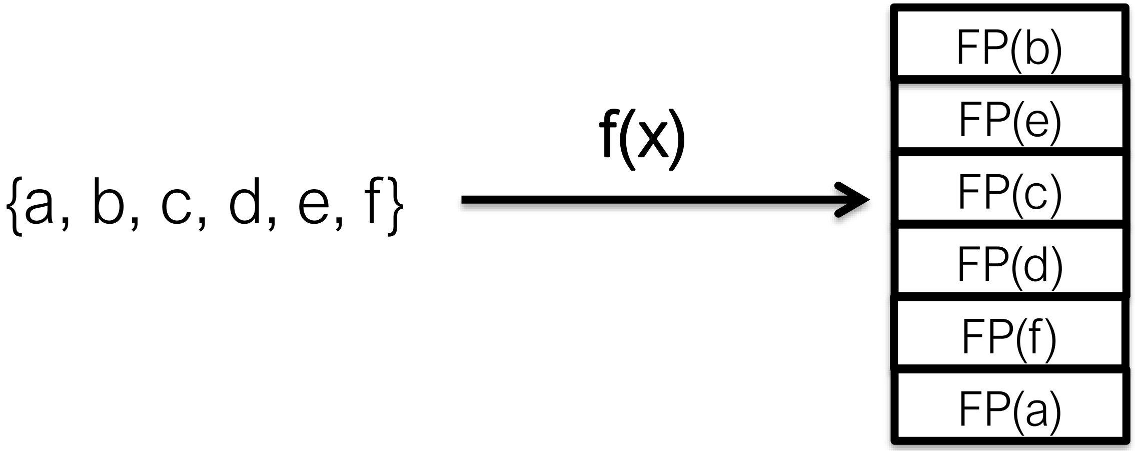


How to Construct Hashtable?

Strawman

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

- Perfect hashing: maps all items with no collisions



{a, b, c, d, e, f}

f(x)

FP(b)

FP(e)

FP(c)

FP(d)

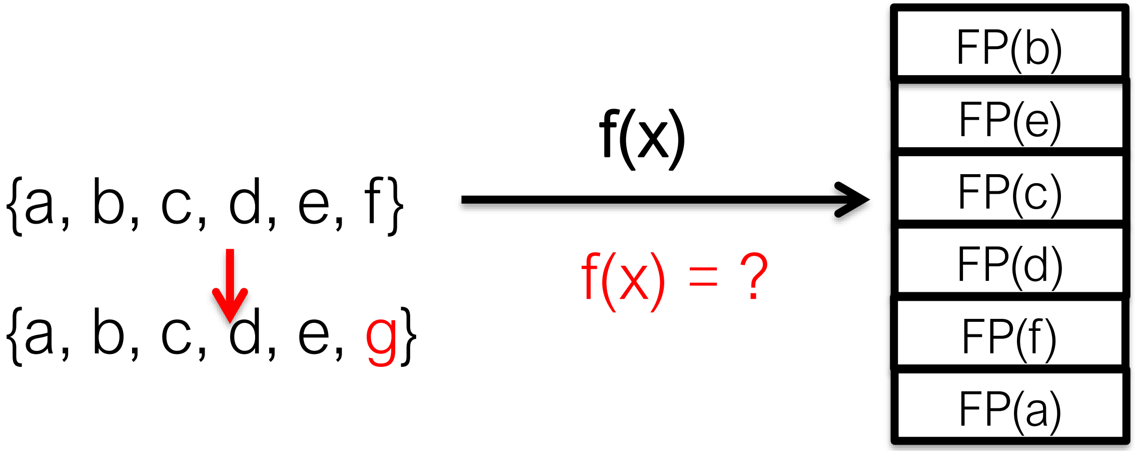
FP(f)

FP(a)

Strawman

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

- Perfect hashing: maps all items with no collisions

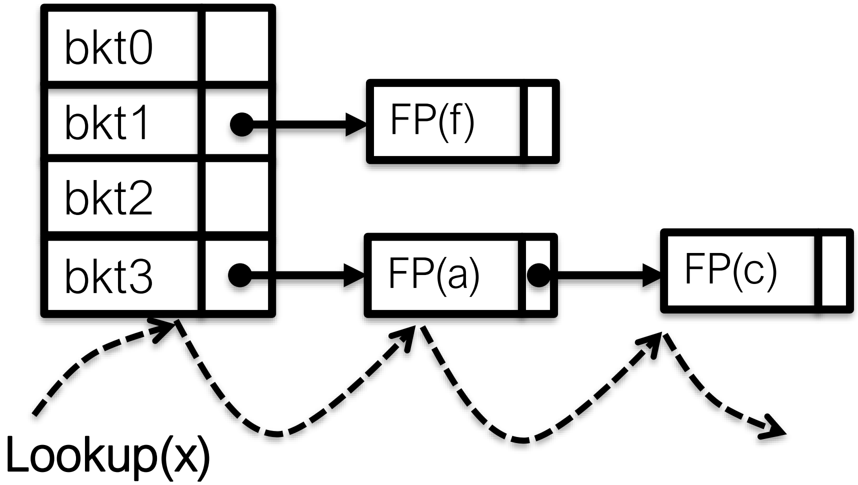


- Changing set must recalculate f ➔ high cost/bad performance of update

Strawman

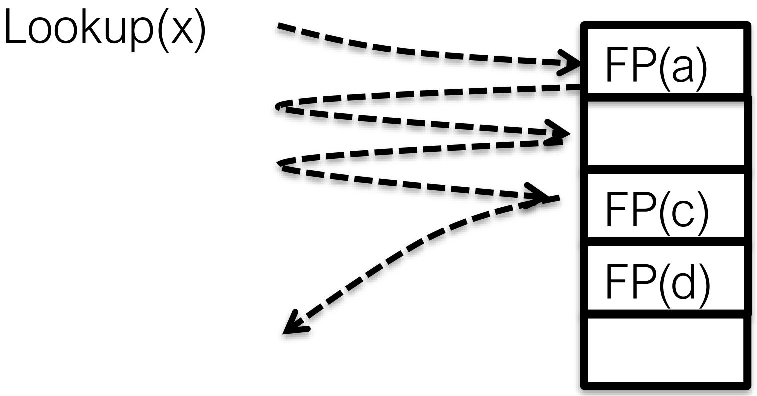
Convention Hash Table: High Space Cost

- Chaining :



- Pointers → low space utilization

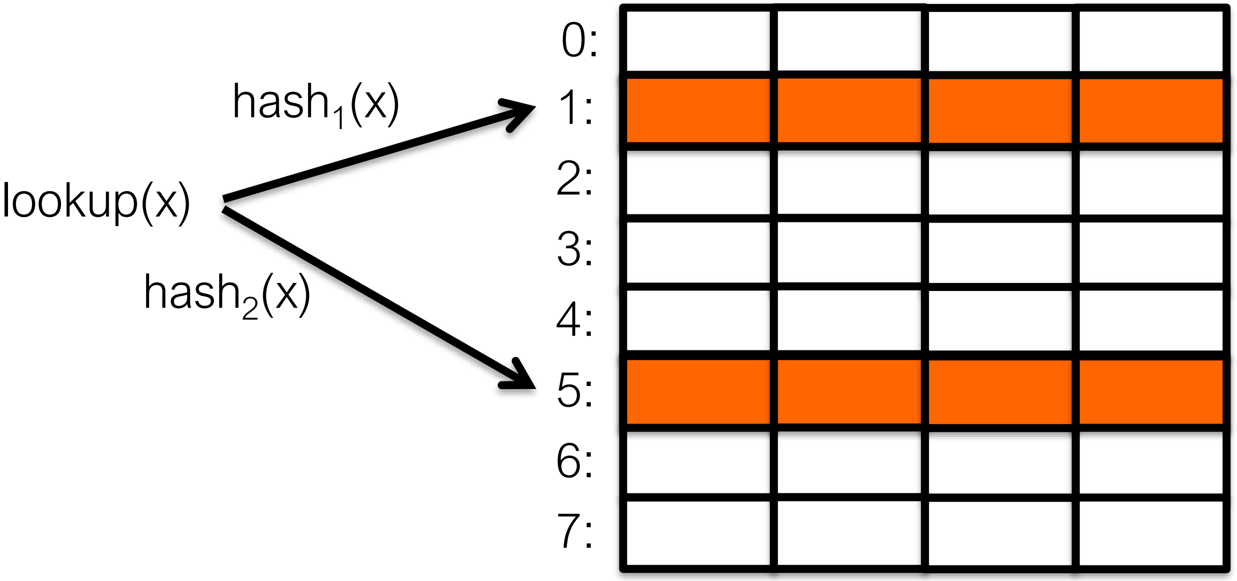
- Linear Probing



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

Cuckoo Hashing[Pagh2004] Good But ..

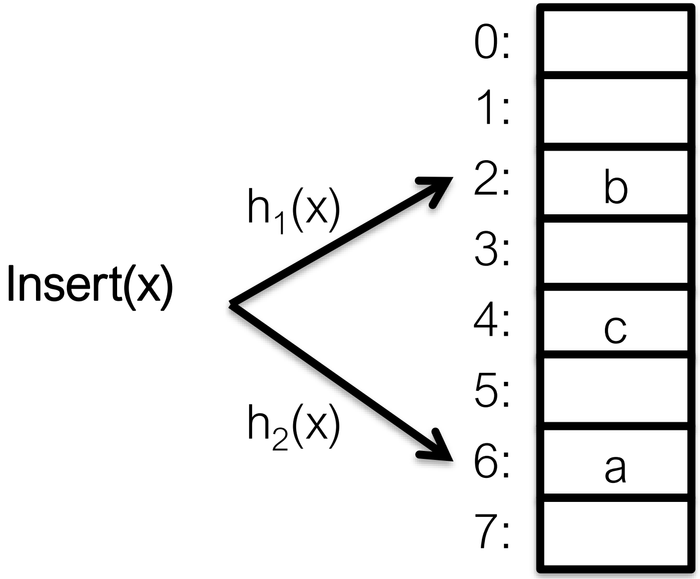
- High Space Utilization
  - 4-way set-associative table: >95% entries occupied
- Fast Lookup: O(1)



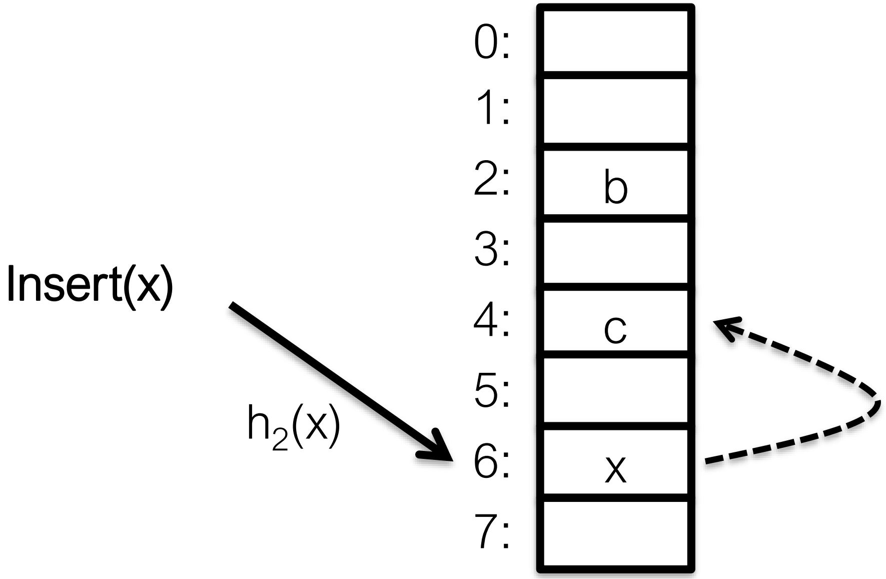
Standard cuckoo hashing doesn’t work with fingerprints

[Pagh2004] Cuckoo hashing.

Standard Cuckoo Requires Storing Each Item

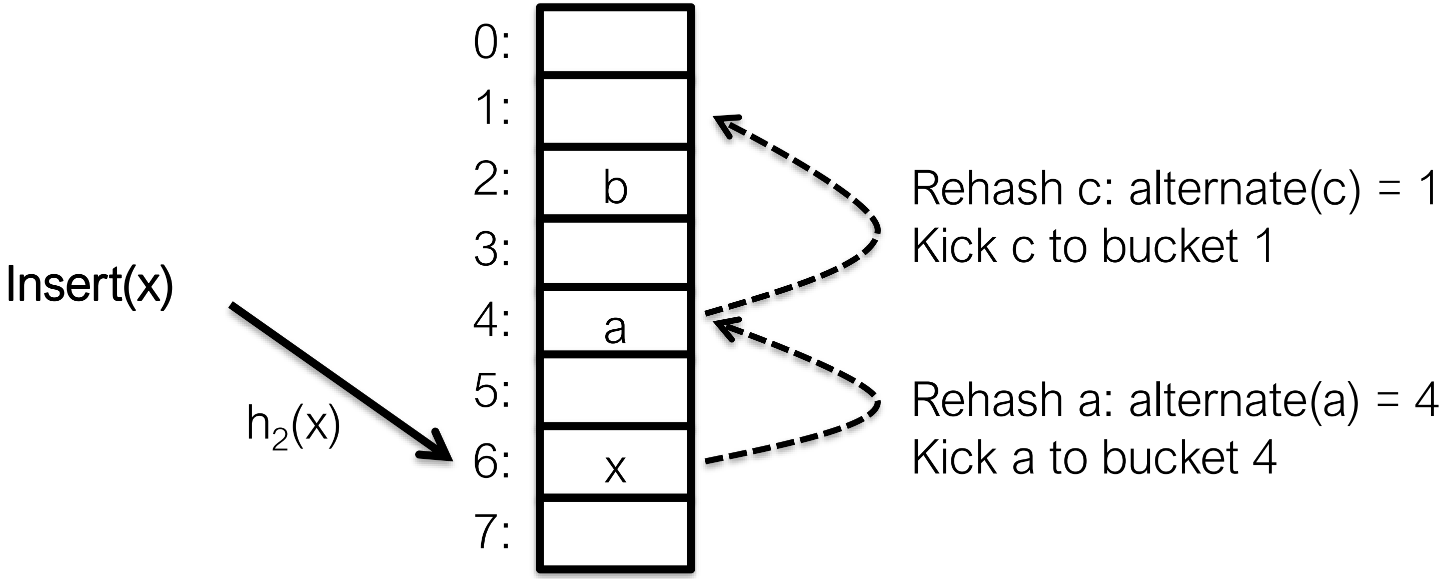


Standard Cuckoo Requires Storing Each Item



Rehash a: alternate(a) = 4
Kick a to bucket 4

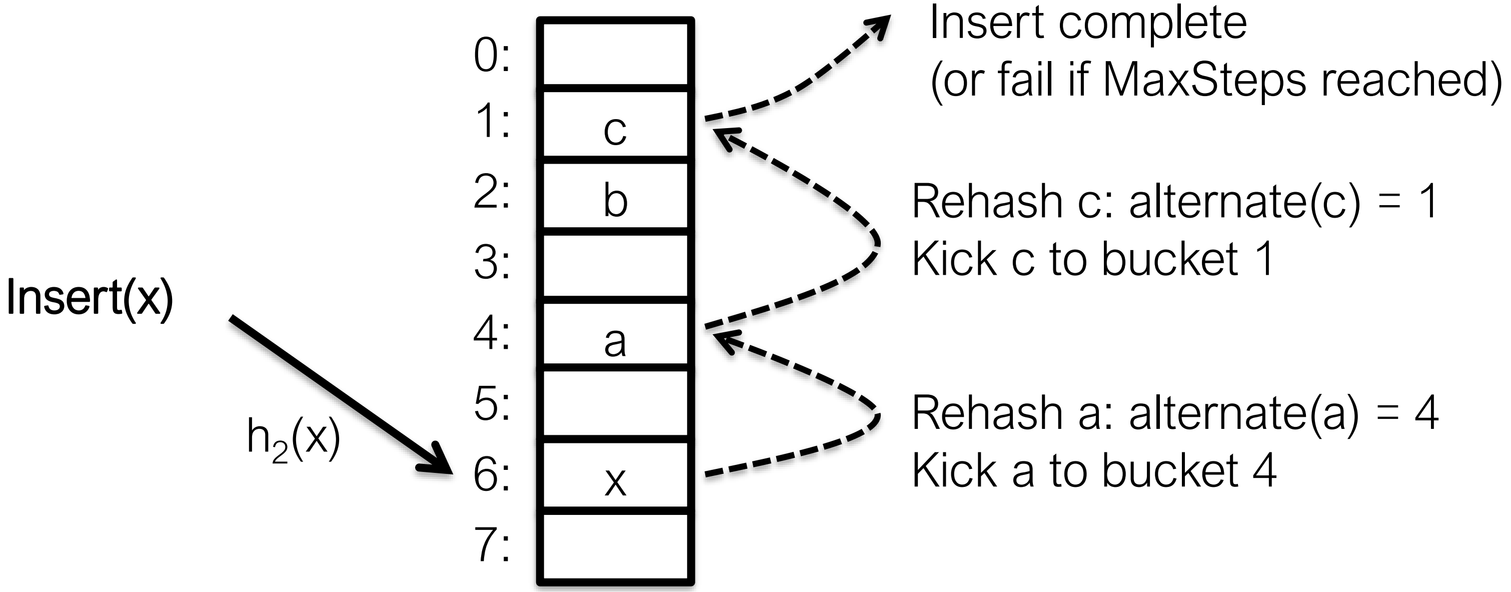
Standard Cuckoo Requires Storing Each Item



Rehash c: alternate(c) = 1
Kick c to bucket 1

Rehash a: alternate(a) = 4
Kick a to bucket 4

Standard Cuckoo Requires Storing Each Item



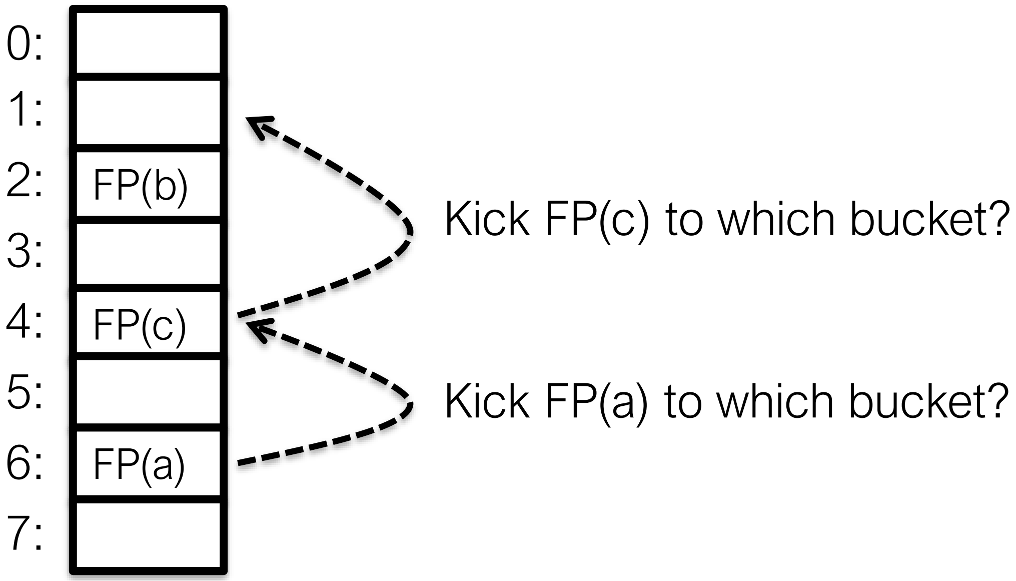
Insert complete
(or fail if MaxSteps reached)

Rehash c: alternate(c) = 1
Kick c to bucket 1

Rehash a: alternate(a) = 4
Kick a to bucket 4

Challenge: How to Perform Cuckoo?

- Cuckoo hashing requires rehashing and displacing existing items



With only fingerprint,
how to calculate item's alternate bucket?

Solution

We Apply Partial-Key Cuckoo

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

$$\texttt{bucket1} = \texttt{hash}_1\texttt{(x)}$$

$$\texttt{bucket2} = \texttt{hash}_2\texttt{(x)}$$

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

$$\texttt{bucket1} = \texttt{hash(x)}$$

$$\texttt{bucket2} = \texttt{bucket1} \oplus \texttt{hash(FP(x))}$$

To displace existing fingerprint:

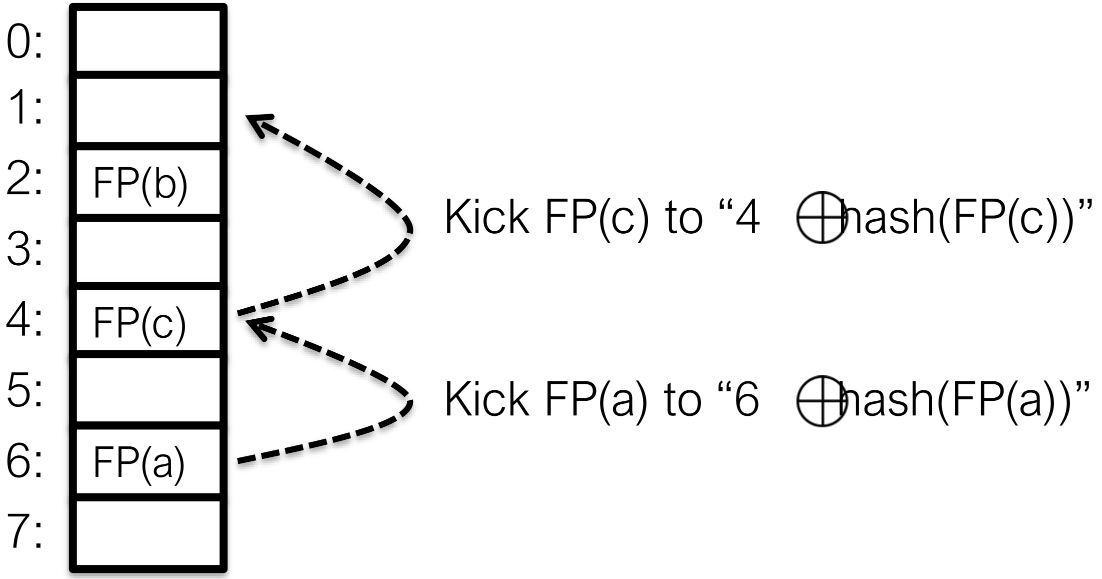
$$\texttt{alternate(x)} = \texttt{current(x)} \oplus \texttt{hash(FP(x))}$$

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

Solution

Partial Key Cuckoo Hashing

- Perform cuckoo hashing on fingerprints



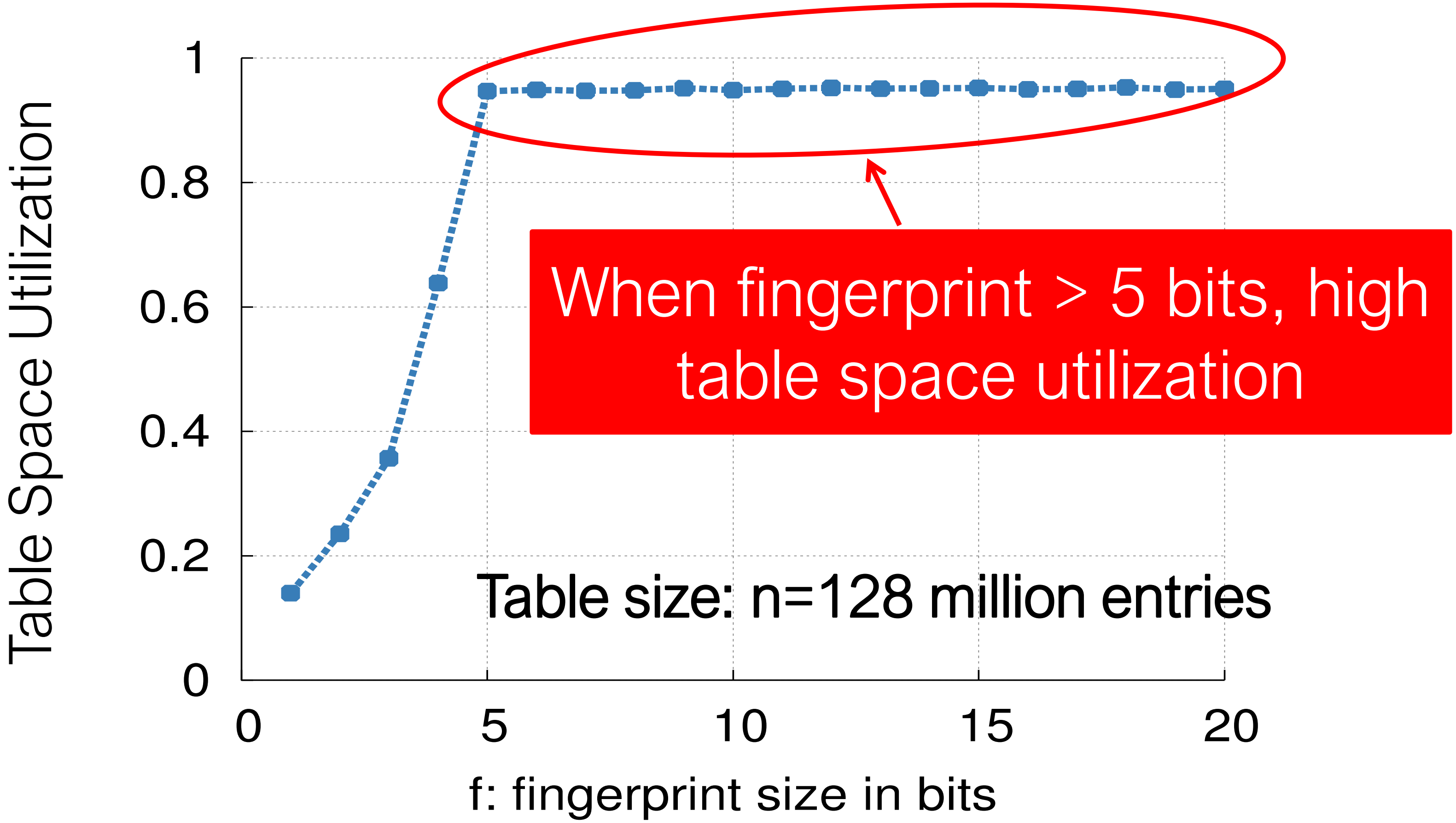
0:
1:
2: FP(b)
3:
4: FP(c)
5:
6: FP(a)
7:

Kick FP(c) to “4 $\oplus$ hash(FP(c))”

Kick FP(a) to “6 $\oplus$ hash(FP(a))”

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

Fingerprints Must Be “Long” for Space Efficiency



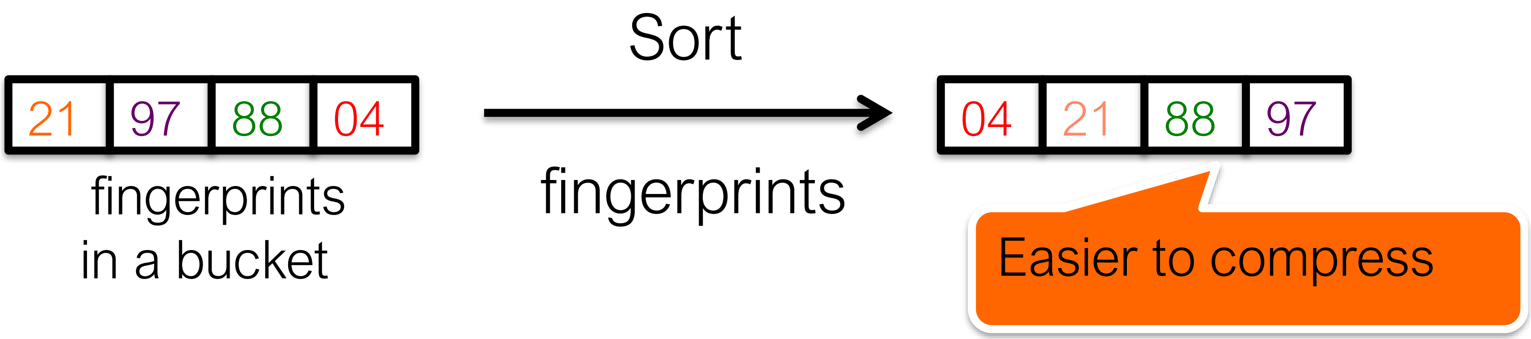
When fingerprint > 5 bits, high table space utilization

Table size: n=128 million entries

- Fingerprint must be $\Omega(\log n / b)$ bits in theory
  - n: hash table size, b: bucket size
  - see more analysis in paper

Semi-Sorting: Further Save 1 bit/item

- Based on observation:
  - A monotonic sequence of integers is easier to compress[Bonomi2006]
- Semi-Sorting:
  - Sort fingerprints sorted in each bucket
  - Compress sorted fingerprints



21	97	88	04

fingerprints in a bucket

Sort fingerprints

04	21	88	97

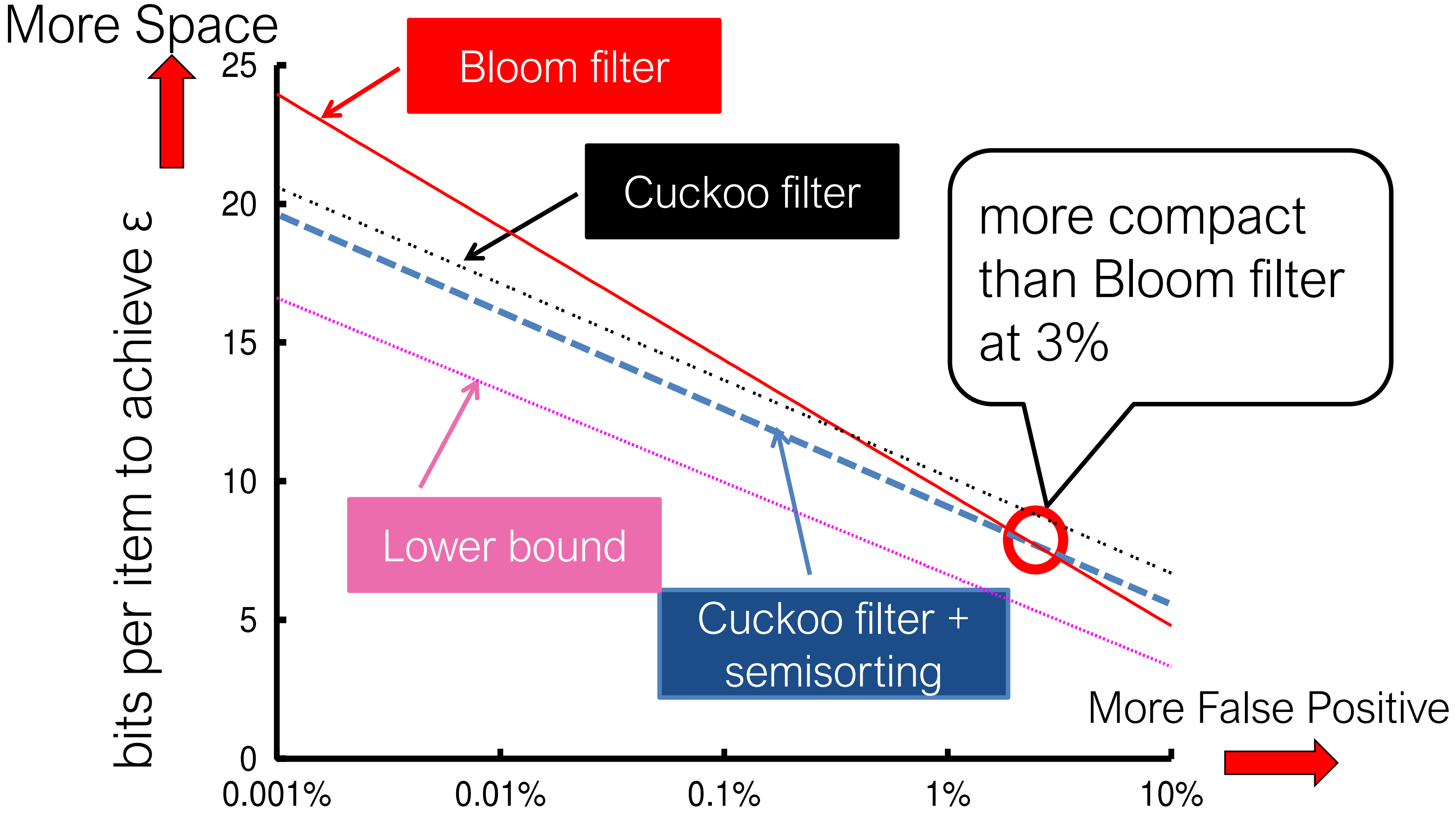
Easier to compress

+ For 4-way bucket, save one bit per item

-- Slower lookup / insert

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

Space Efficiency



ε: target false positive rate

Outline

- Background
- Cuckoo filter algorithm
- Performance evaluation
- Summary

Evaluation

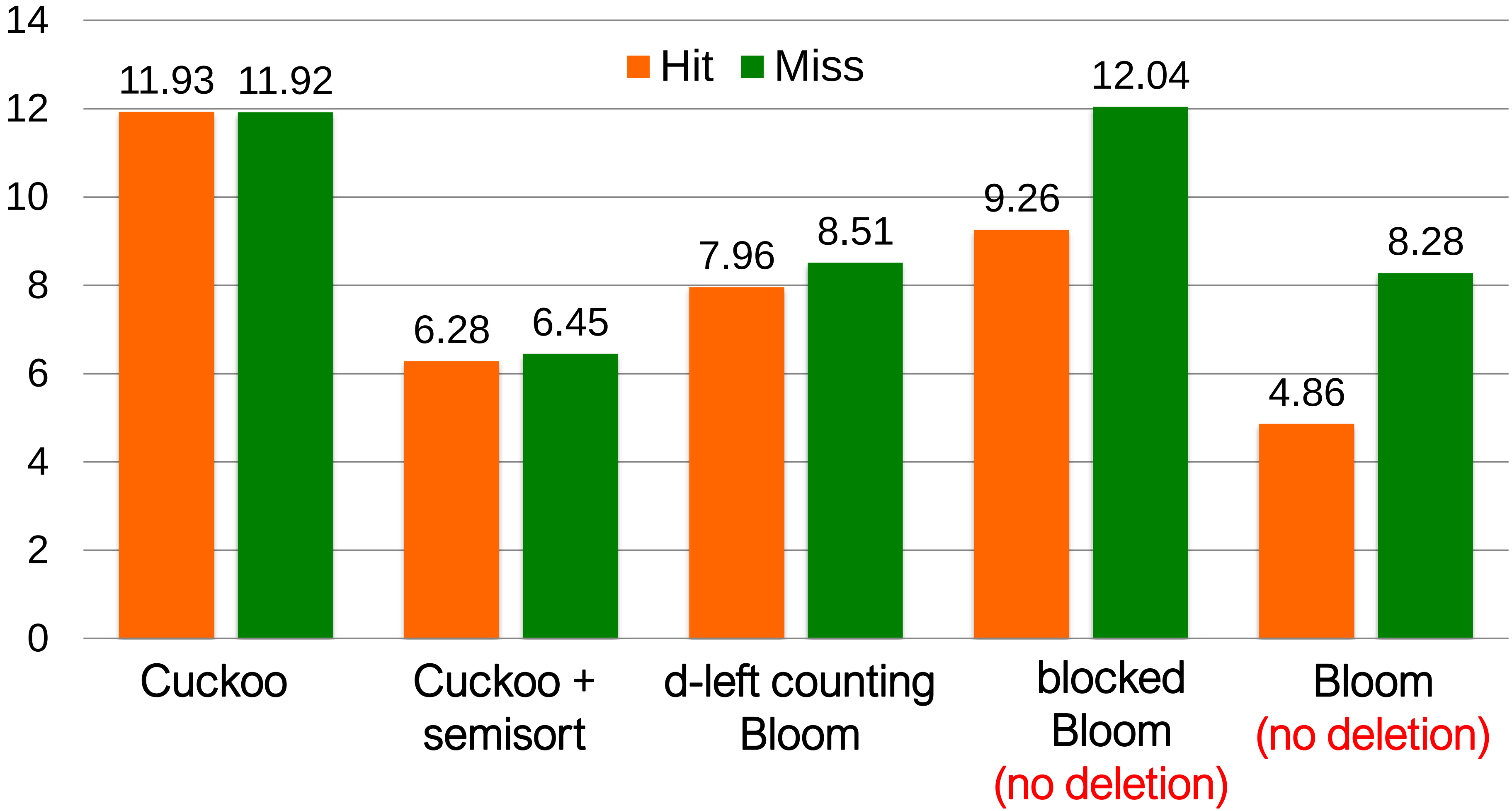
- Compare cuckoo filter with
  - Bloom filter (cannot delete)
  - Blocked Bloom filter [Putze2007] (cannot delete)
  - d-left counting Bloom filter [Bonomi2006]
  - Cuckoo filter + semisorting
  - More in the paper

- C++ implementation, single threaded

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

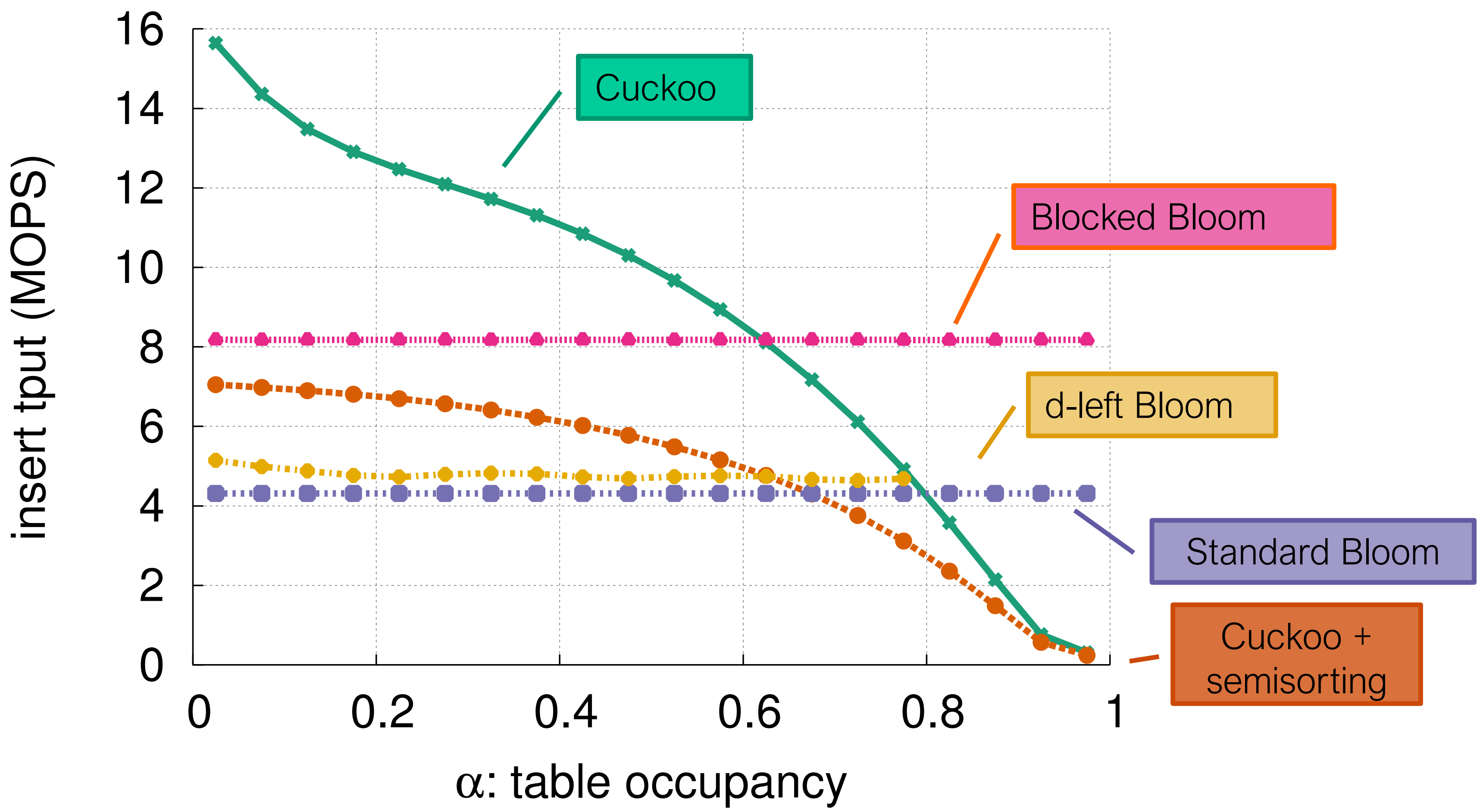
[Bonomi2006] Beyond Bloom filters: From approximate membership checks to approximate state machines.

Lookup Performance (MOPS)



Cuckoo filter is among the fastest regardless workloads.

Insert Performance (MOPS)



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

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