Cuckoo Hashing in Cryptography: Optimal Parameters, Robustness and Applications

Kevin Yeo

Google | Columbia University

Outline

What is Cuckoo Hashing?

New Cuckoo Hashing Constructions

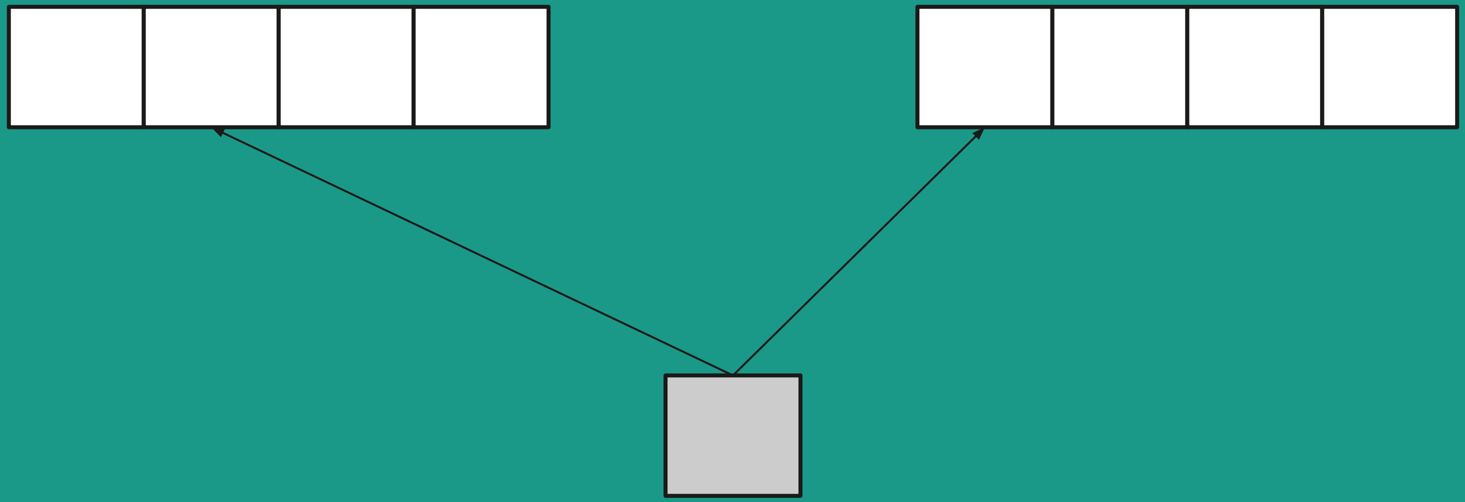
- Quadratic Improvement
- Lower Bound

Robust Cuckoo Hashing

- Optimal Construction
- Lower Bound

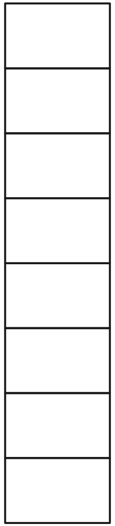
Applications

What is Cuckoo Hashing?



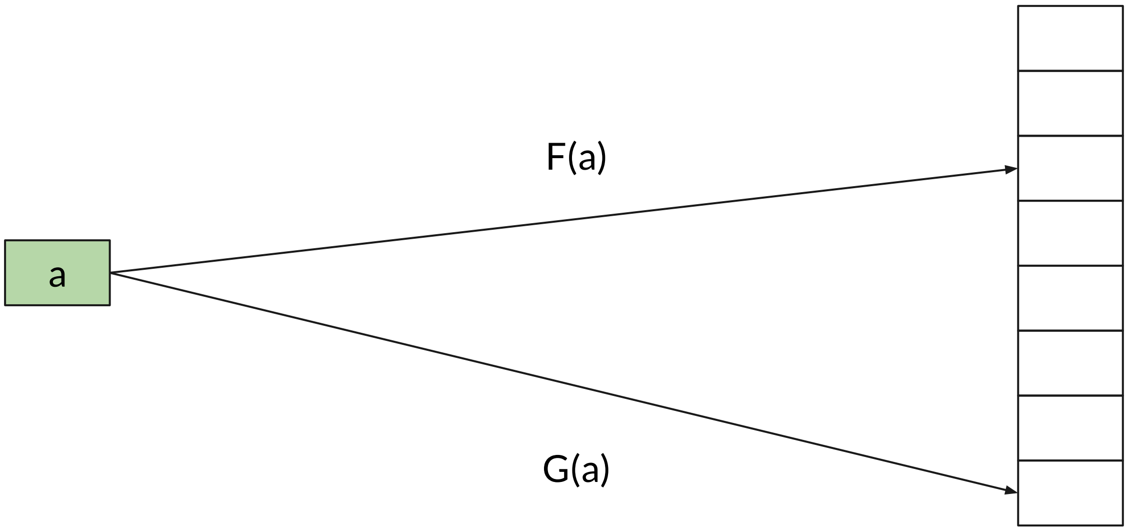
Cuckoo Hashing [PR04]

Hash Functions: F, G



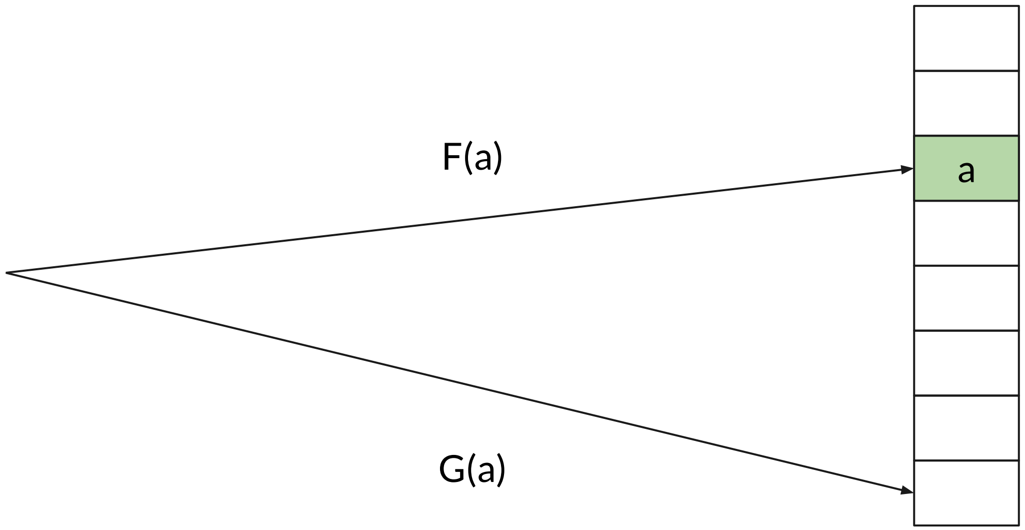
Cuckoo Hashing [PR04]

Hash Functions: F, G



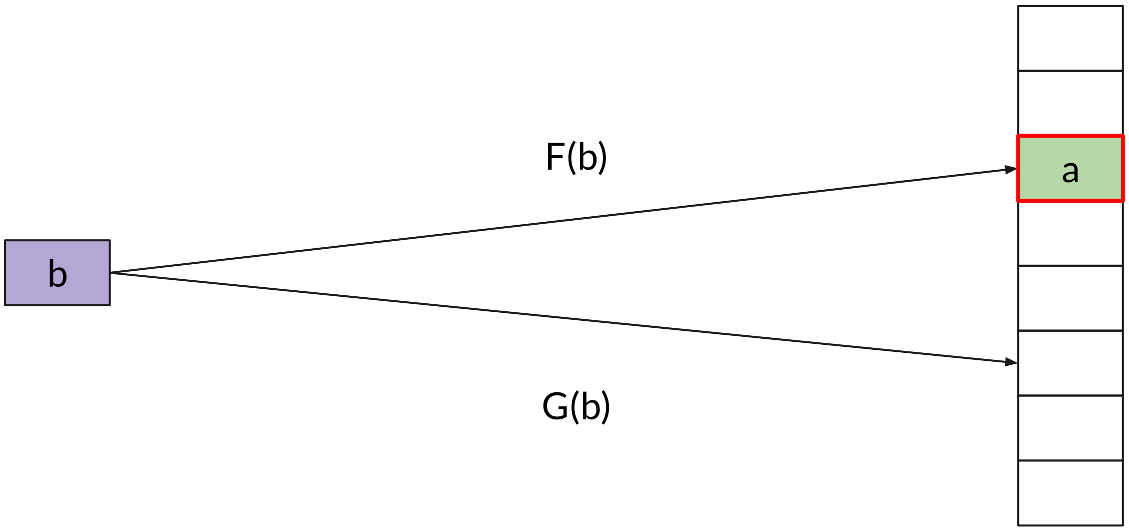
Cuckoo Hashing [PR04]

Hash Functions: F, G



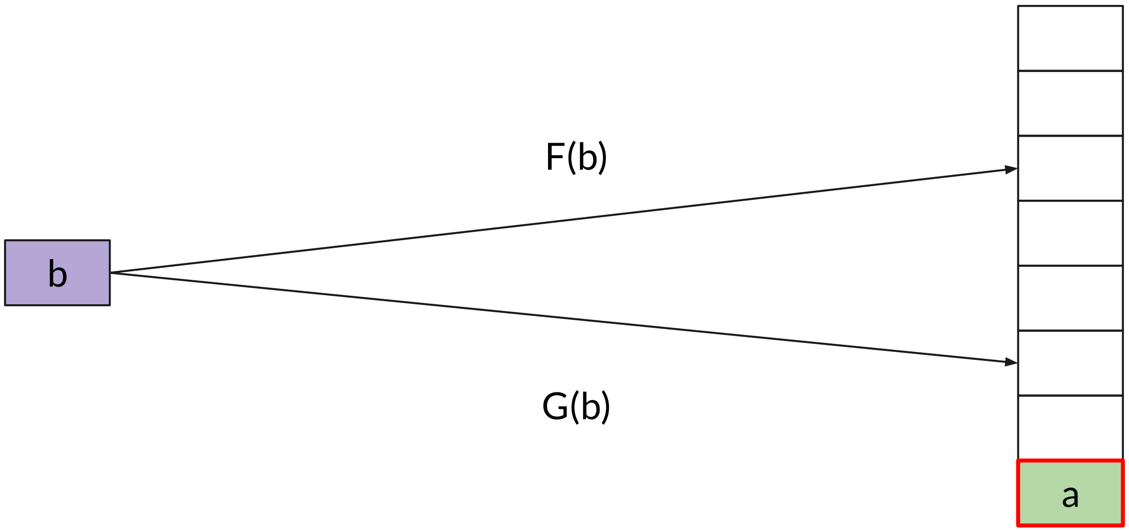
Cuckoo Hashing [PR04]

Hash Functions: F, G



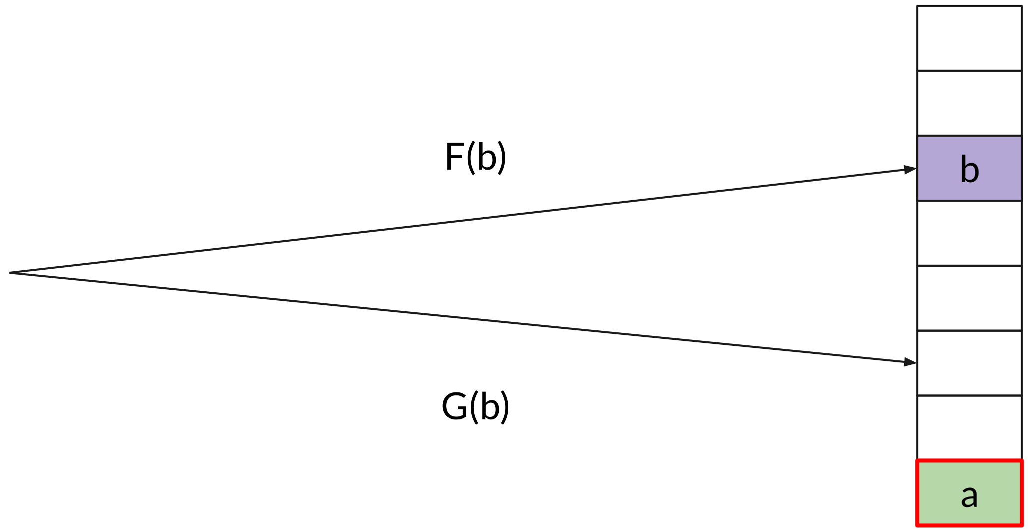
Cuckoo Hashing [PR04]

Hash Functions: F, G



Cuckoo Hashing [PR04]

Hash Functions: F, G



Cuckoo Hashing [PR04]

Theorem. For a cuckoo hashing table with O(N) entries and for any set of N items, the insertion process fails at allocating the N items with probability 1/poly(N) over the random choice of the hash functions.

Query Time: O(1)

Failure Probability ε: 1/poly(N)

Cuckoo Hashing [PR04]

Theorem. For a cuckoo hashing table with O(N) entries and for any set of N items, the insertion process fails at allocating the N items with probability 1/poly(N) over the random choice of the hash functions.

Query Time: O(1)

Failure Probability ε: 1/poly(N)

Failure only considers the inability to construct a cuckoo hashing table.

Perfect Construction Algorithms

Definition. A construction algorithm is *perfect* if it the algorithm always outputs an allocation assuming there exists at least one successful allocation.

There exists several perfect construction algorithms running in time O(N * polylog(N)). See paper for details.

Goal. Construct cuckoo hashing schemes that emit at least one successful allocation for every set of N items.

Example Cryptographic Usage

Private Key: K

(L_1, V_1)
(L_2, V_2)
...
(L_n, V_n)

Example Cryptographic Usage

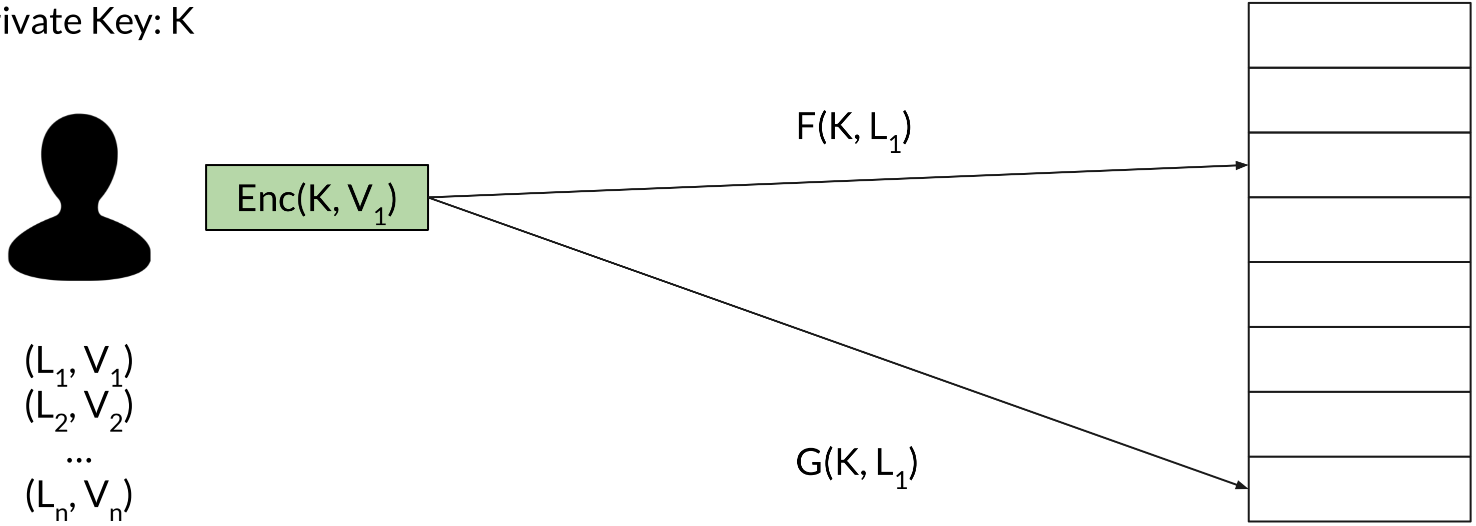
Private Key: K

$Enc(K, V_1)$

(L_1, V_1)
(L_2, V_2)
...
(L_n, V_n)

Example Cryptographic Usage

Private Key: K



Example Cryptographic Usage

Private Key: K

(L_1, V_1)
(L_2, V_2)
...
(L_n, V_n)

Enc(K, V_6)
Enc(K, V_4)
Enc(K, V_2)
Enc(K, V_3)
Enc(K, V_1)

Example Cryptographic Usage

Private Key: K

(L_1, V_1)
(L_2, V_2)
…
(L_n, V_n)

ORAM: [PR10, GM11, HFNO21]

Encrypted Search: [PPYY19, BBF+21]

PIR: [ACLS18, DRRT18, ALP+21]

$\mathrm{Enc}(K, V_6)$
$\mathrm{Enc}(K, V_4)$
$\mathrm{Enc}(K, V_2)$
$\mathrm{Enc}(K, V_3)$
$\mathrm{Enc}(K, V_1)$

Cuckoo Hashing [PR04]

Theorem. For a cuckoo hashing table with O(N) entries and for any set of N items, the insertion process fails at allocating the N items with probability **1/poly(N)** over the random choice of the hash functions.

Query Time: O(1)

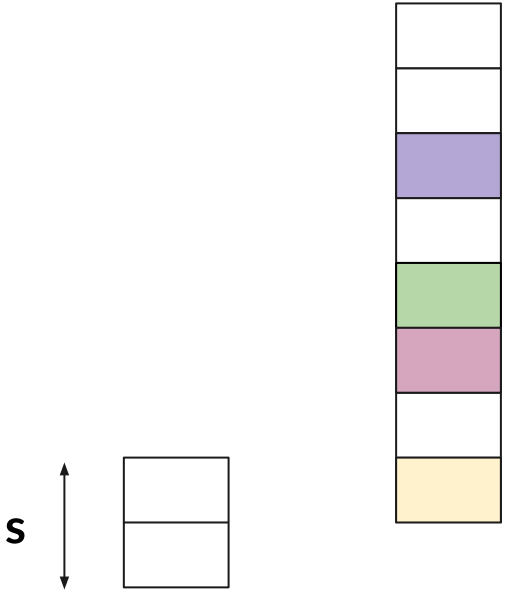
Failure Probability ε: **1/poly(N)**

Prior works [GM11, KLO12] showed that 1/poly(N) failure incurs privacy leaks.

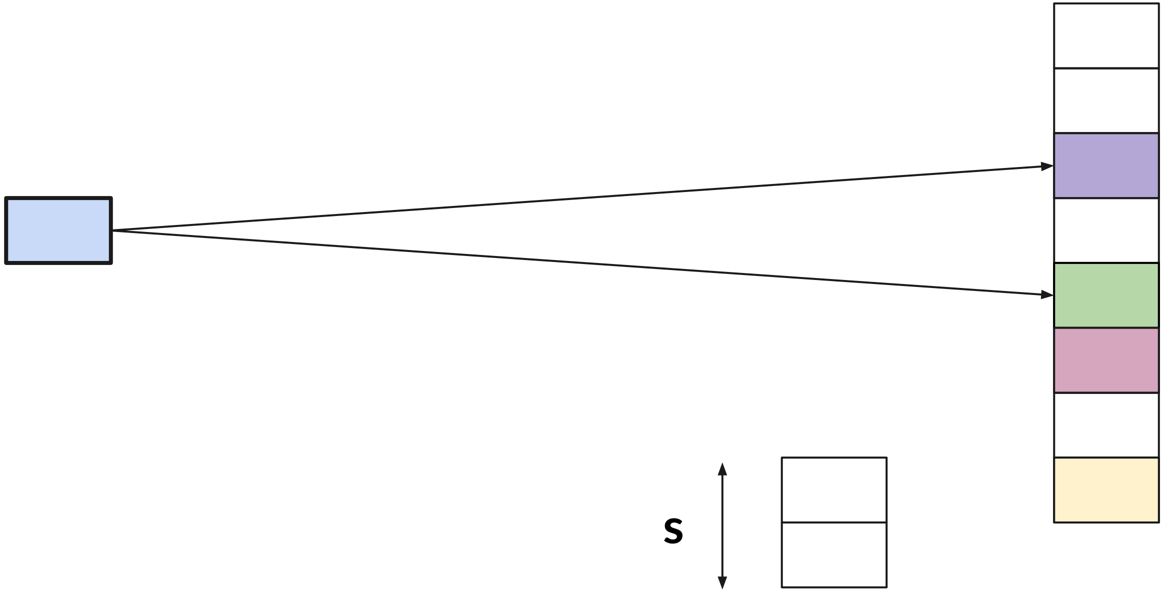
Prior Extensions for Negligible Failure

- Cuckoo Hashing with an Overflow Stash [KMW08, ADW14, MP23]
- Larger Entries [DW07,MP23]
- More Hash Functions [FPSS05]

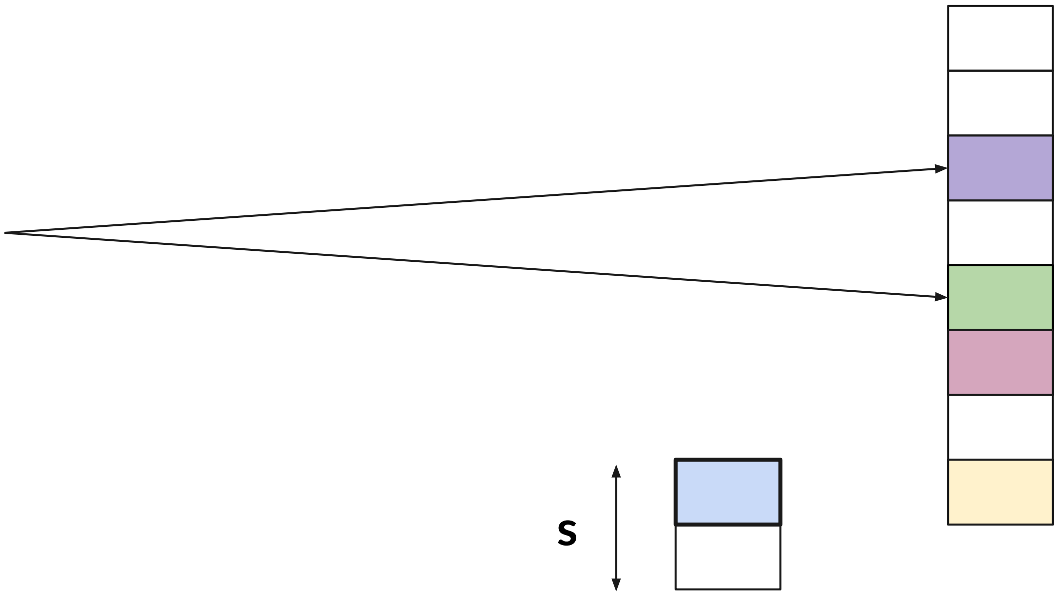
Overflow Stash (s)



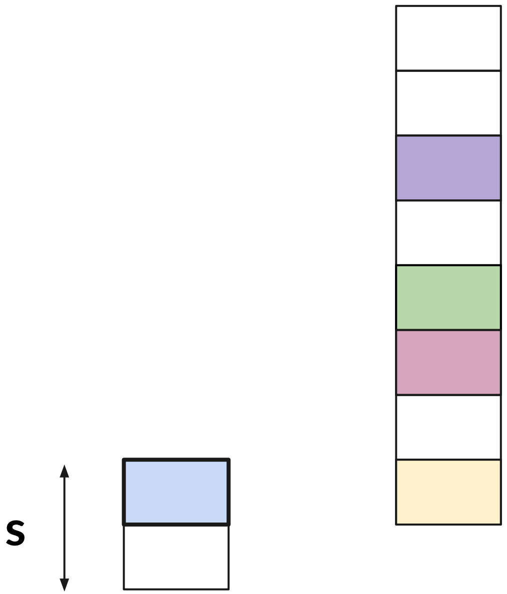
Overflow Stash (s)



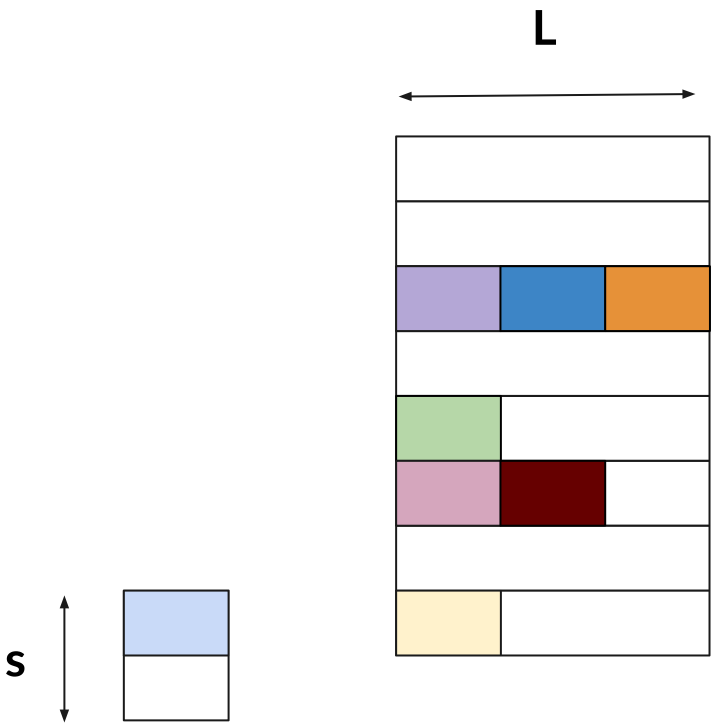
Overflow Stash (s)



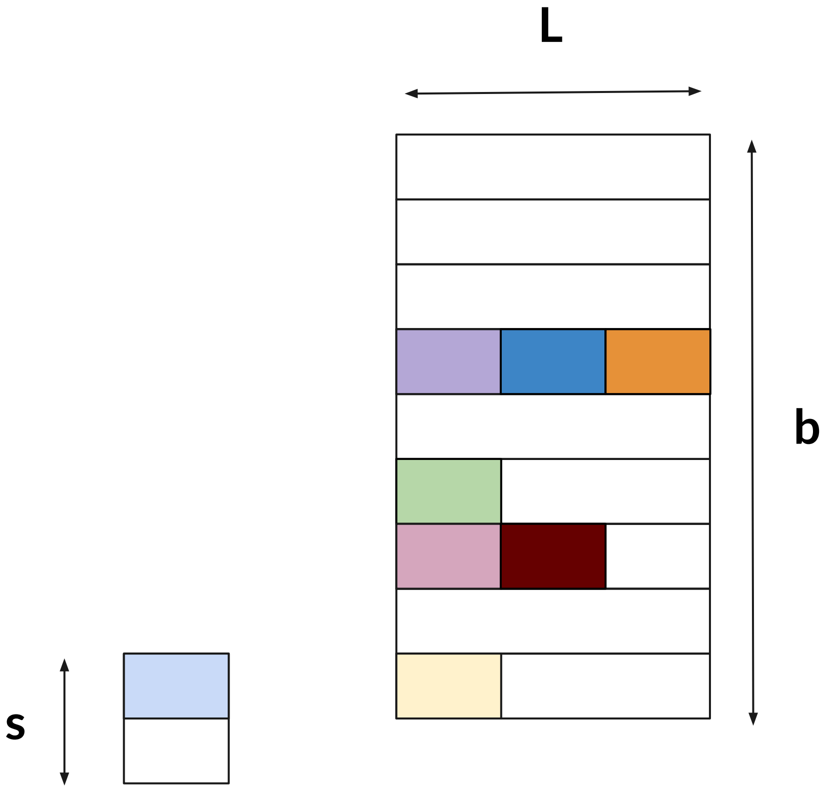
Entry Size (L)



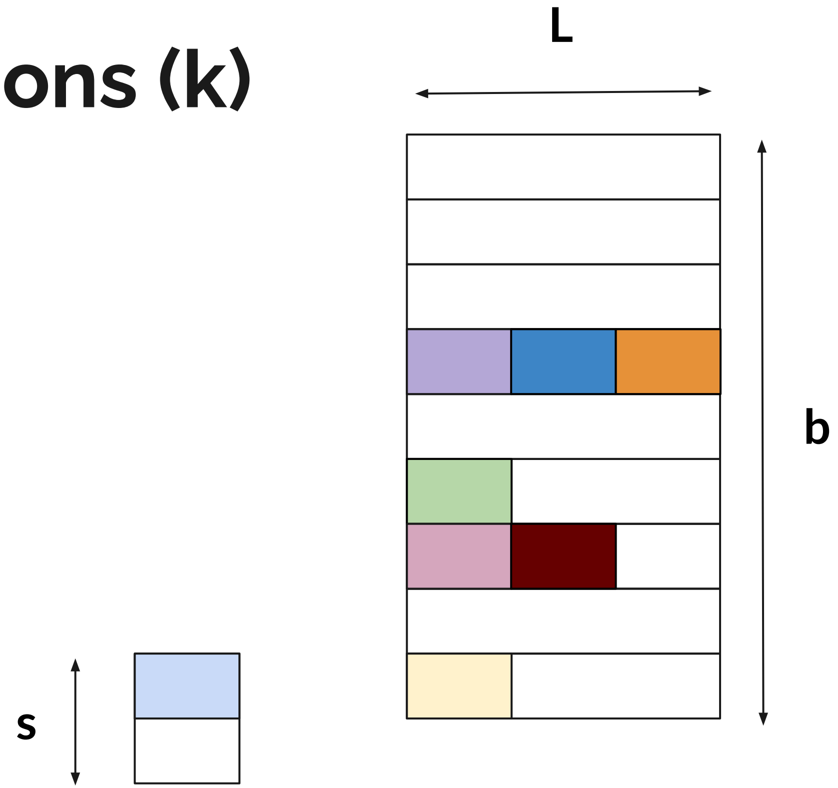
Entry Size (L)



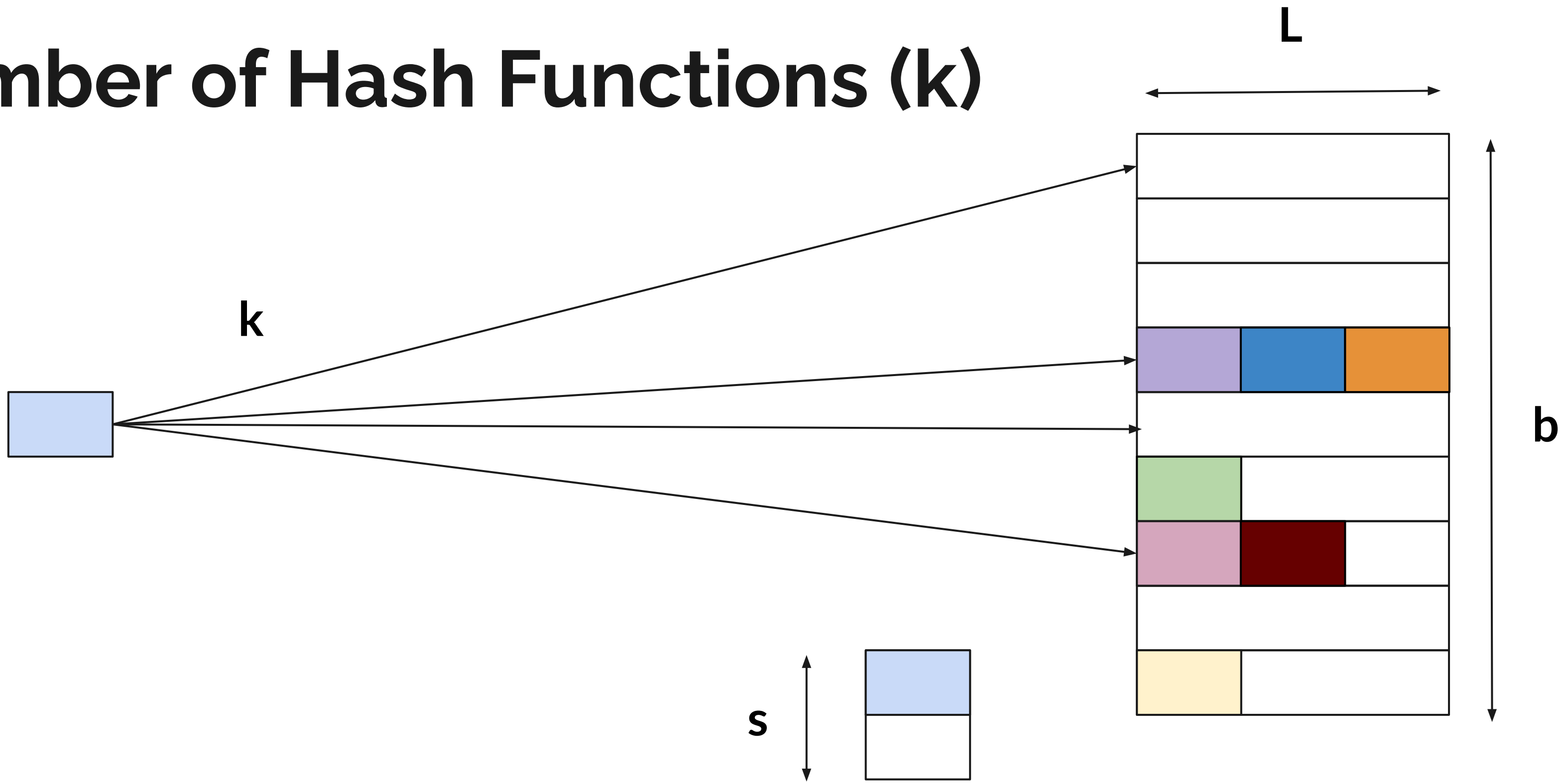
Number of Entries (b)



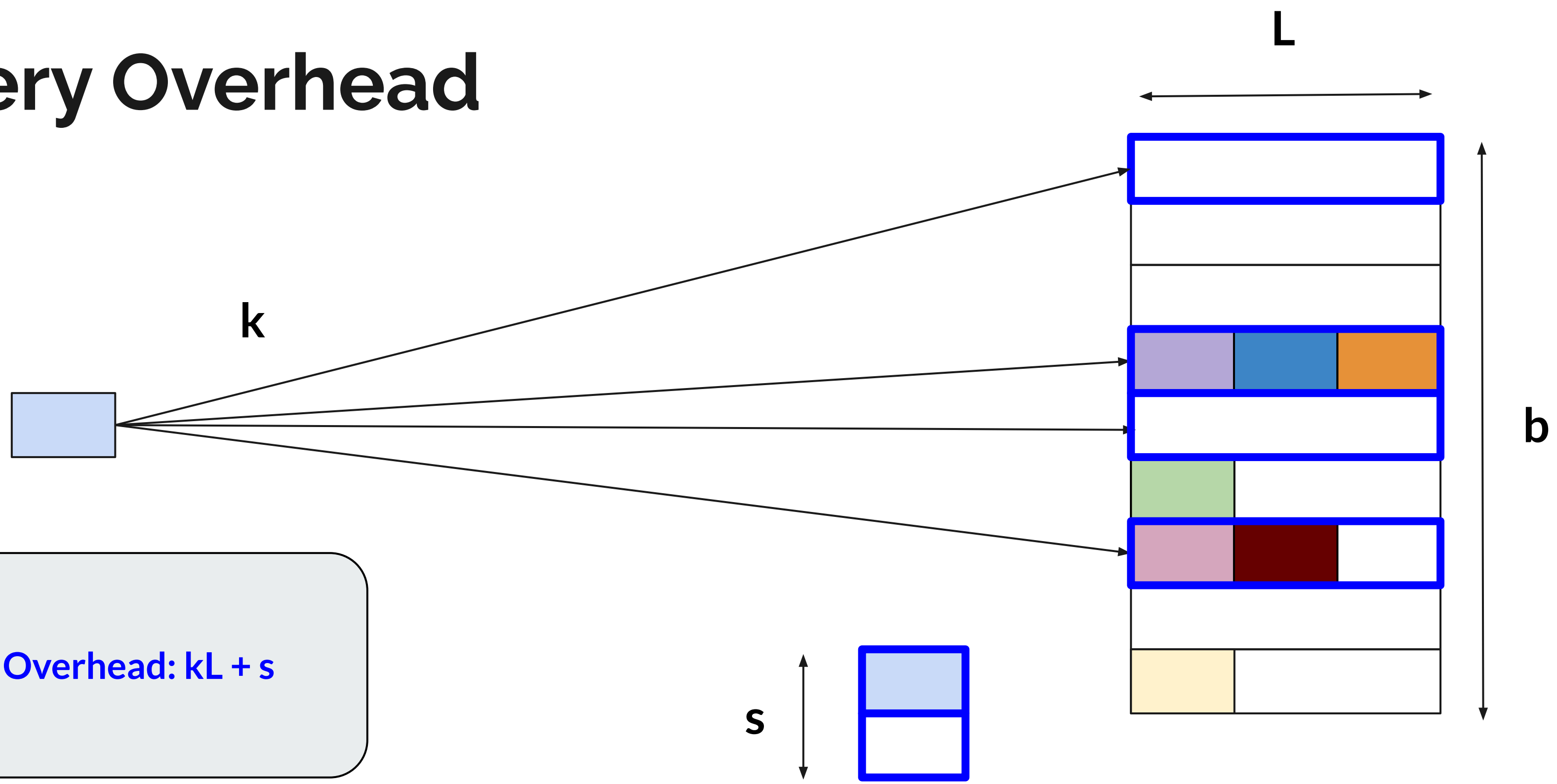
Number of Hash Functions (k)



Number of Hash Functions (k)



Query Overhead



Query Overhead: kL + s

Generalized Cuckoo Hashing

- **k**: number of hash functions
- **L**: size of each entry
- **s:** size of stash
- **b**: number of entries
- **Query Overhead: kL + s**

Prior Works

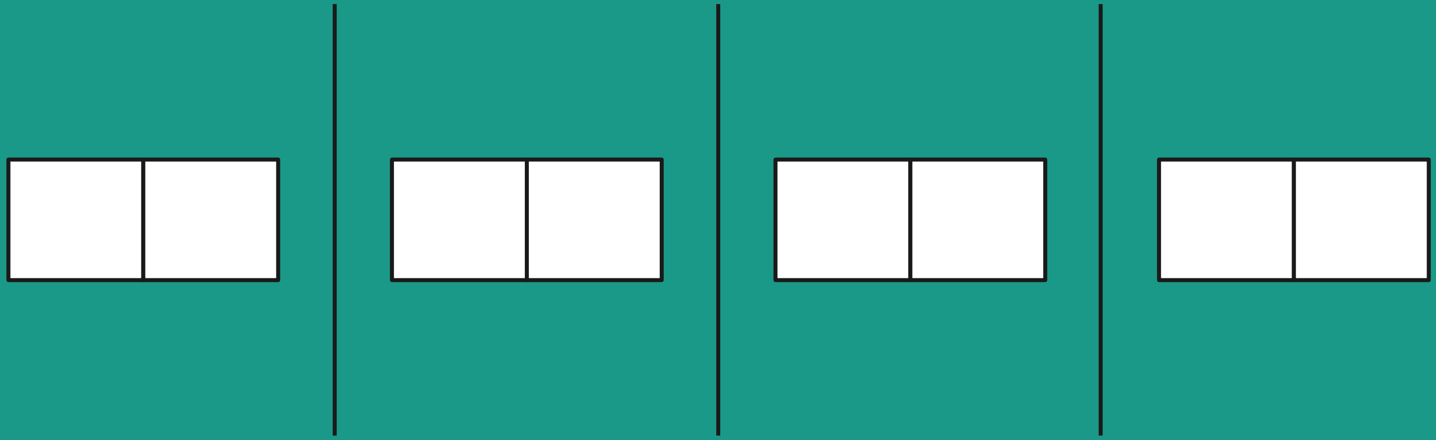
	Hash Functions k	Entry Size ℓ	Entries b	Stash Size s	Failure ϵ	Query Overhead
Cuckoo Hashing [PR04]	2	1	$O(n)$	0	$1/n^{O(1)}$	$O(1)$
Large-Sized Entries [DW07]	2	$O(1)$	$(1+\alpha)n/\ell$	0	$1/n^{O(1)}$	$O(1)$
Large-Sized Entries [MP20]	2	$O(1+\log(1/\epsilon)/\log n)$	$O(n/\ell)$	0	ϵ	$O(1+\log(1/\epsilon)/\log n)$
Constant-Sized Stash [KMW10]	2	1	$O(n)$	$O(1)$	$1/n^{O(s)}$	$O(1)$
Large-Sized Stash [ADW14]	2	1	$O(n)$	$O(1+\log(1/\epsilon)/\log n)$	ϵ	$O(1+\log(1/\epsilon)/\log n)$
More Hash Functions [FPSS05]	$O(1+\log(1/\epsilon)/\log n)$	1	$O(n)$	0	ϵ	$O(1+\log(1/\epsilon)/\log n)$

Figure 1: Comparison table of known cuckoo hashing instantiations. The query overhead $k\ell + s$ is the number of locations to search when retrieving an item.

Failure Probability: ε

Query Overhead: O(log(1/ε)/log(N))

New Cuckoo Hashing Constructions



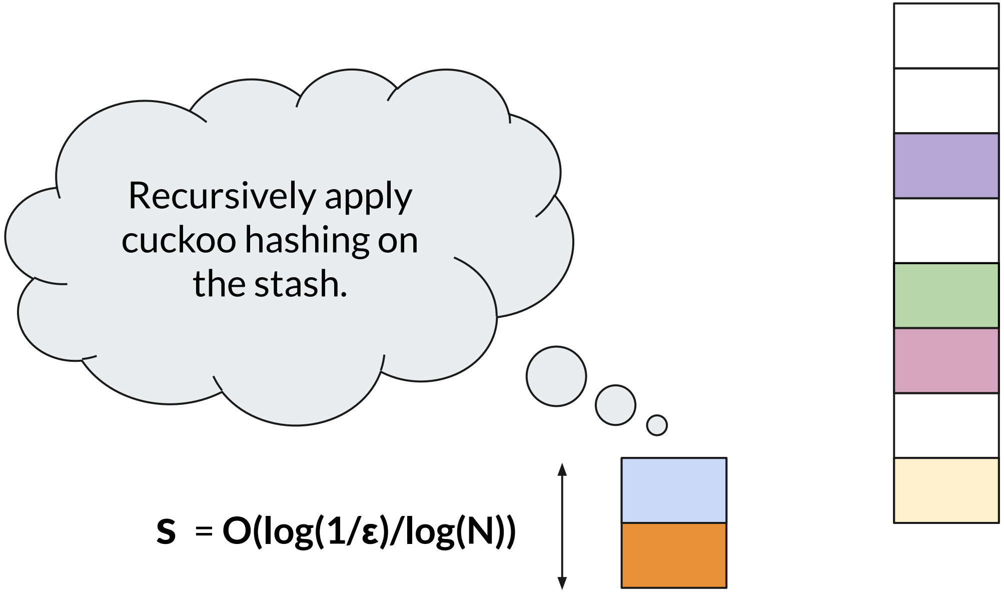
Failed Recursive Construction

Theorem [GM11, ADW14]: Cuckoo hashing with the following parameters has failure probability ε:

- **k = 2** hash functions
- **b = O(n)** entries
- **L = 1** entry size
- **s = O(log(1/ε)/log(N))** overflow stash size

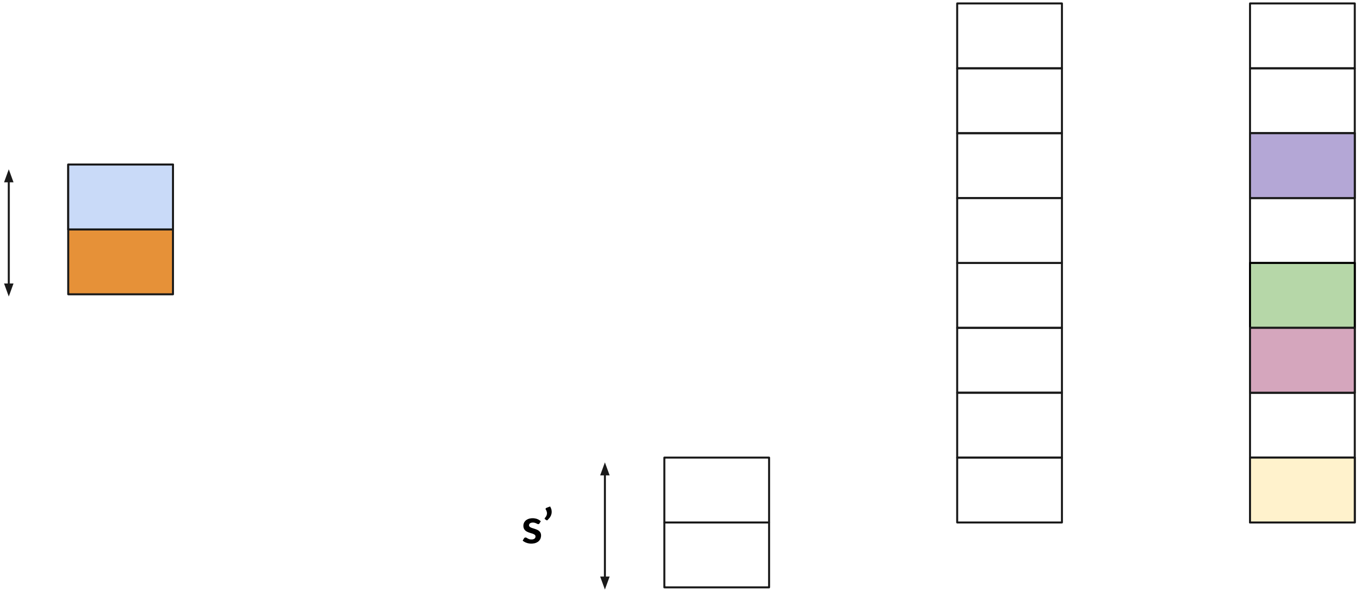
Query Overhead: O(log(1/ε)/log(N))

Failed Recursive Construction

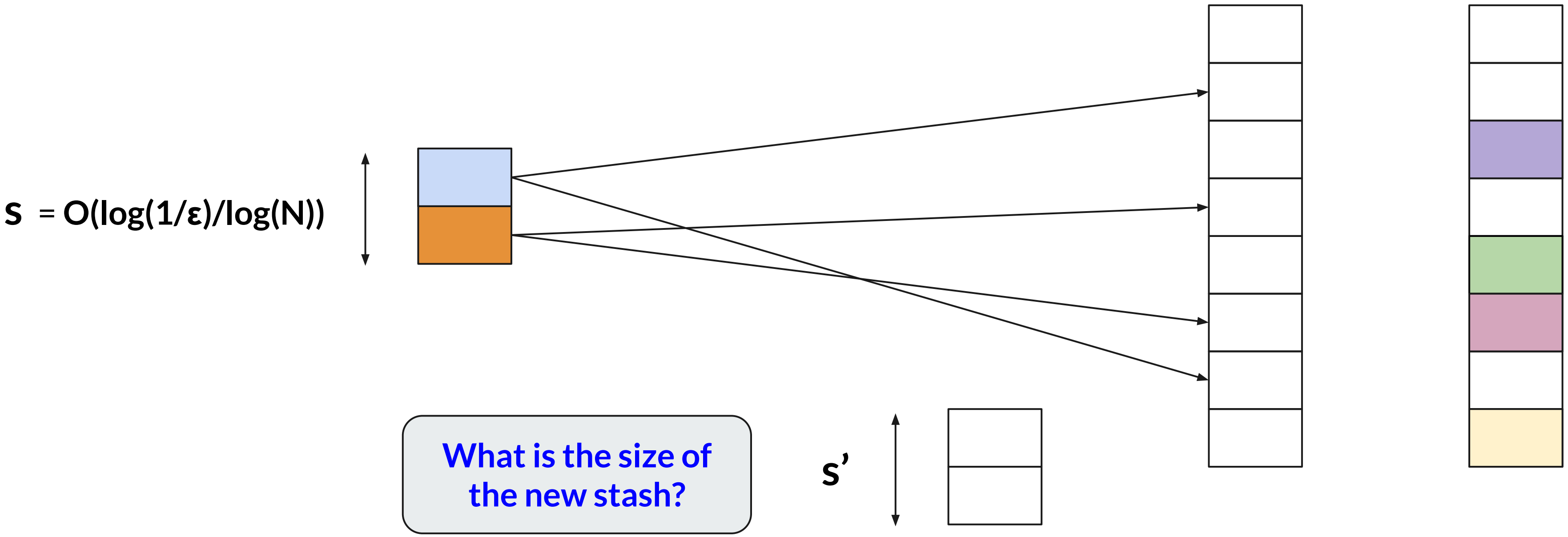


Failed Recursive Construction

$s = O(\log(1/\varepsilon)/\log(N))$

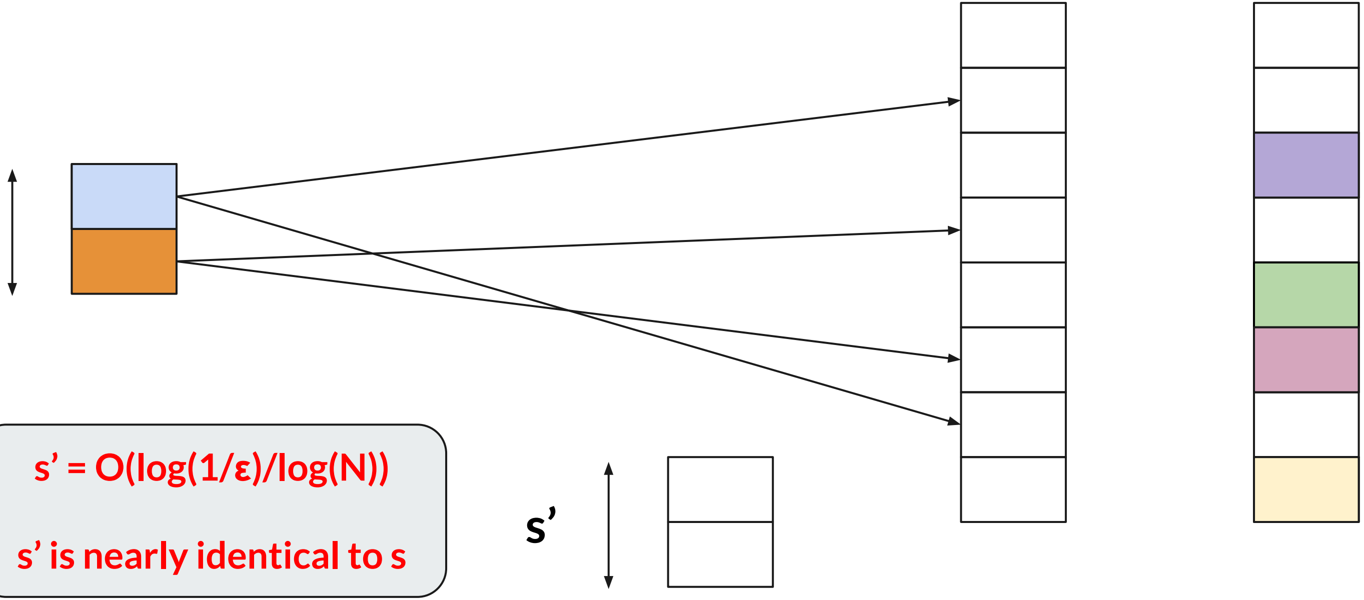


Failed Recursive Construction



Failed Recursive Construction

s = O(log(1/ε)/log(N))



s’ = O(log(1/ε)/log(N))

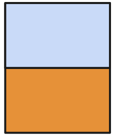
s’ is nearly identical to s

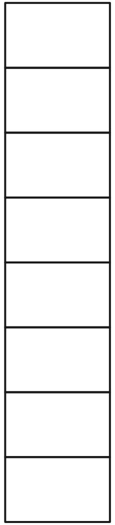
Insights from Failed Construction

Insight 1: Failures in cuckoo hashing are localized to small sets of items. Allocating small sets of items is as challenging as allocating large sets of items.

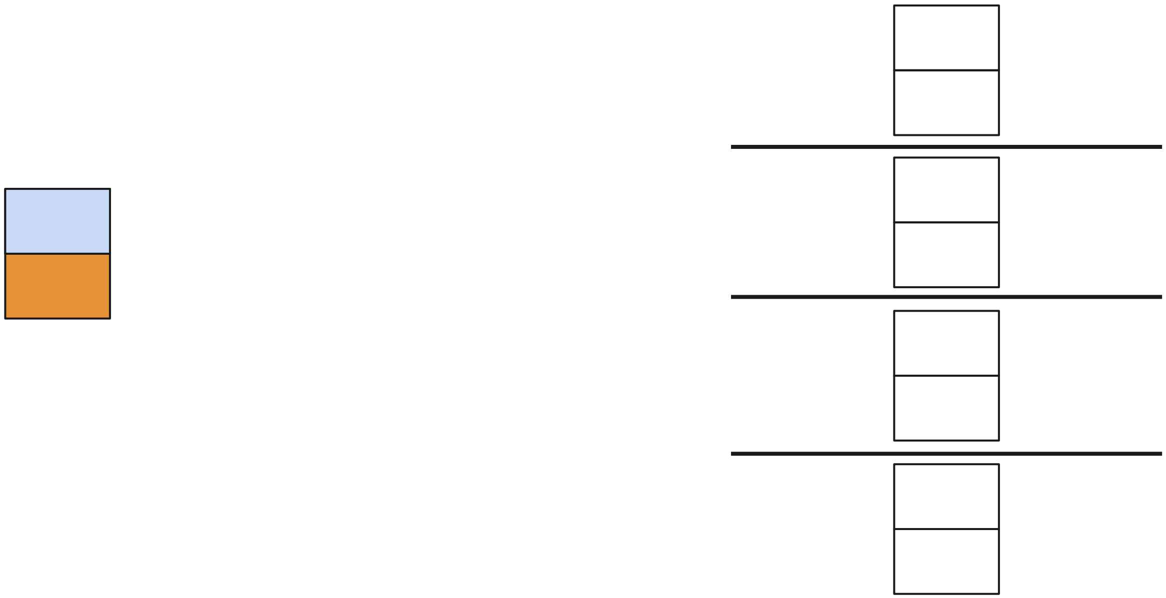
Insight 2: If a cuckoo hashing scheme can handle allocating small sets of items, it seems that they can immediately scale towards handling much larger sets of items.

Handling Small Sets

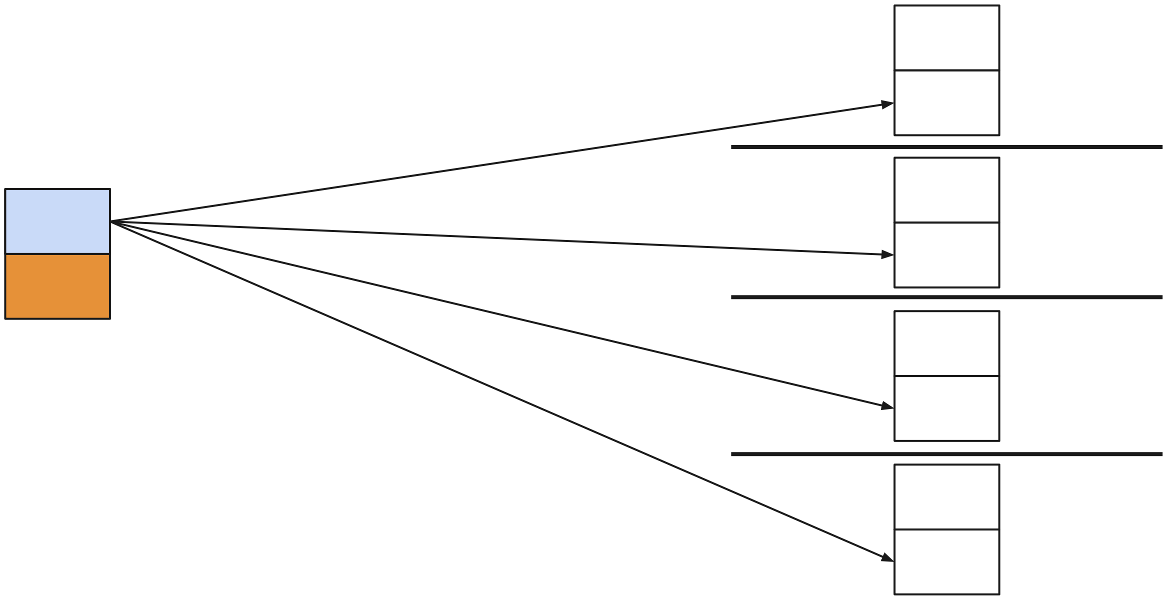




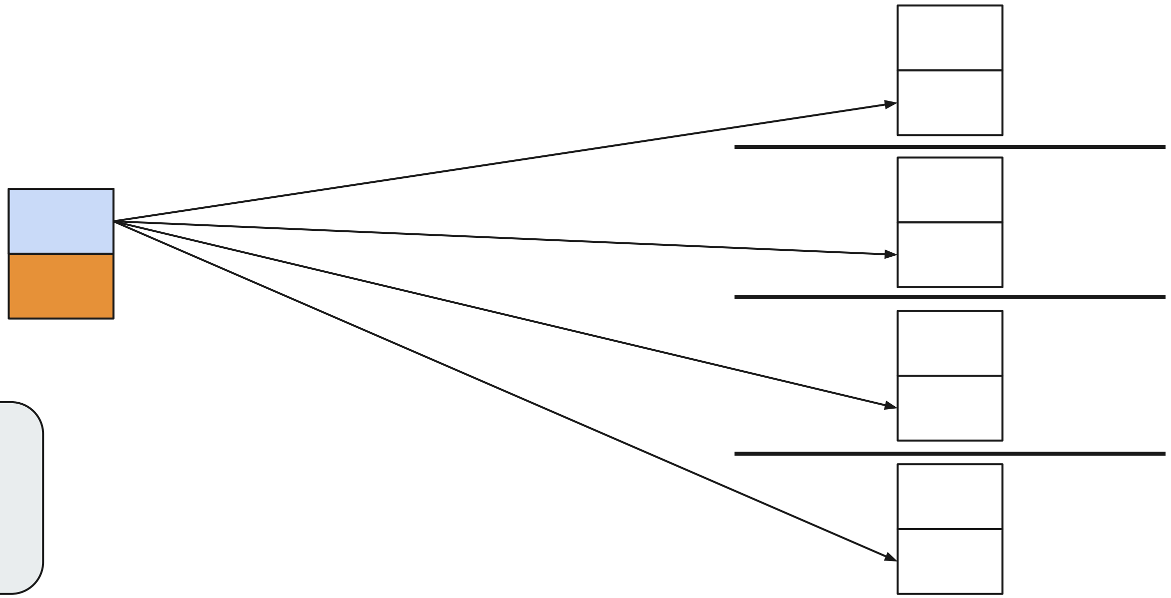
Disjoint Tables and More Hash Functions



Disjoint Tables and More Hash Functions



Disjoint Tables and More Hash Functions



Small sets of items are more well distributed.

Our New Construction

Theorem (Ours). Cuckoo hashing with the following parameters has failure probability ε:

- **$k = O((\log(1/\varepsilon)/\log(N))^{1/2})$ hash functions**
- $b = O(n)$ entries
- **k disjoint tables of size b/k**
- $L = 1$ entry sizes
- $s = 0$ (no overflow stash)

Query Overhead: $O((\log(1/\varepsilon)/\log(N))^{1/2})$

Our New Construction

	Hash Functions k	Entry Size ℓ	Entries b	Stash Size s	Failure ϵ	Query Overhead
Cuckoo Hashing [PR04]	2	1	$O(n)$	0	$1/n^{O(1)}$	$O(1)$
Large-Sized Entries [DW07]	2	$O(1)$	$(1+\alpha)n/\ell$	0	$1/n^{O(1)}$	$O(1)$
Large-Sized Entries [MP20]	2	$O(1+\log(1/\epsilon)/\log n)$	$O(n/\ell)$	0	ϵ	$O(1+\log(1/\epsilon)/\log n)$
Constant-Sized Stash [KMW10]	2	1	$O(n)$	$O(1)$	$1/n^{O(s)}$	$O(1)$
Large-Sized Stash [ADW14]	2	1	$O(n)$	$O(1+\log(1/\epsilon)/\log n)$	ϵ	$O(1+\log(1/\epsilon)/\log n)$
More Hash Functions [FPSS05]	$O(1+\log(1/\epsilon)/\log n)$	1	$O(n)$	0	ϵ	$O(1+\log(1/\epsilon)/\log n)$
Our Work	$\boldsymbol{O(1+\sqrt{\log(1/\epsilon)/\log n})}$	1	$O(n)$	0	ϵ	$\boldsymbol{O(1+\sqrt{\log(1/\epsilon)/\log n})}$

Figure 1: Comparison table of known cuckoo hashing instantiations. The query overhead $k\ell + s$ is the number of locations to search when retrieving an item.

Necessity of Disjoint Tables

	Hash Functions k	Entry Size ℓ	Entries b	Stash Size s	Failure ϵ	Query Overhead
Cuckoo Hashing [PR04]	2	1	$O(n)$	0	$1/n^{O(1)}$	$O(1)$
Large-Sized Entries [DW07]	2	$O(1)$	$(1+\alpha)n/\ell$	0	$1/n^{O(1)}$	$O(1)$
Large-Sized Entries [MP20]	2	$O(1+\log(1/\epsilon)/\log n)$	$O(n/\ell)$	0	ϵ	$O(1+\log(1/\epsilon)/\log n)$
Constant-Sized Stash [KMW10]	2	1	$O(n)$	$O(1)$	$1/n^{O(s)}$	$O(1)$
Large-Sized Stash [ADW14]	2	1	$O(n)$	$O(1+\log(1/\epsilon)/\log n)$	ϵ	$O(1+\log(1/\epsilon)/\log n)$
More Hash Functions [FPSS05]	$O(1+\log(1/\epsilon)/\log n)$	1	$O(n)$	0	ϵ	$O(1+\log(1/\epsilon)/\log n)$
Our Work	$\boldsymbol{O(1+\sqrt{\log(1/\epsilon)/\log n})}$	1	$O(n)$	0	ϵ	$\boldsymbol{O(1+\sqrt{\log(1/\epsilon)/\log n})}$

Figure 1: Comparison table of known cuckoo hashing instantiations. The query overhead $k\ell + s$ is the number of locations to search when retrieving an item.

Necessity of Joint Tables

Theorem (Ours). For cuckoo hashing with a single shared table, it must be that **$k = \Omega(\log(1/\varepsilon)/\log(N))$** when there are $b = O(n)$ entries of size $L = 1$ and no overflow stash ($s = 0$).

Corollary. The construction in [FPSS05] with a single shared table is optimal.

Lower Bound from Insights

Insight 1: Failures in cuckoo hashing are localized to small sets of items. Allocating small sets of items is as challenging as allocating large sets of items.

Insight 2: If a cuckoo hashing scheme can handle allocating small sets of items, it seems that they can immediately scale towards handling much larger sets of items.

Our Lower Bound

Theorem (Ours). For any cuckoo hashing scheme with failure probability ε and b = O(N) entries,

$$(k^2 * L) + (k * s) = \Omega(\log(1/\varepsilon)/\log(N))$$

Our Lower Bound

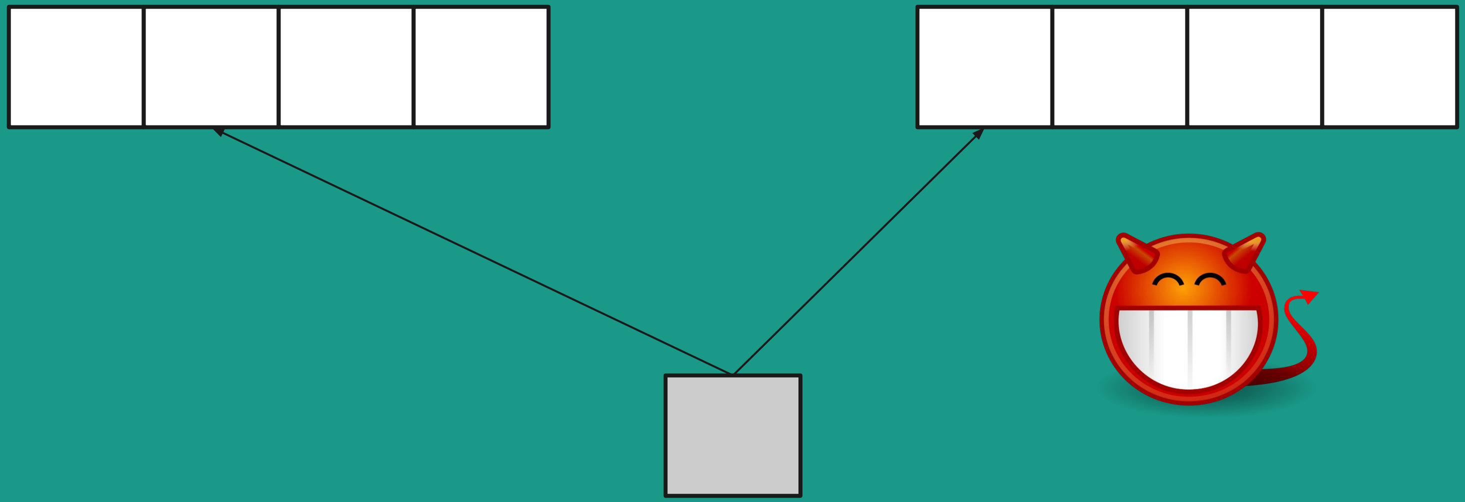
Theorem (Ours). For any cuckoo hashing scheme with failure probability ε and $b = O(N)$ entries,

$$(k^2 * L) + (k * s) = \Omega(\log(1/\varepsilon)/\log(N))$$

Corollary 1. The most efficient possible construction is ours with query overhead $O((\log(1/\varepsilon)/\log(N))^{1/2})$.

Corollary 2. The most efficient approach is using many hash functions (large k).

Robust Cuckoo Hashing



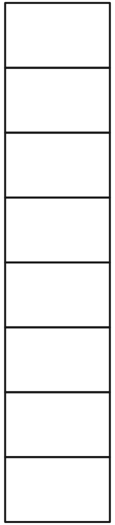
Robust Cuckoo Hashing

Theorem. For a cuckoo hashing table with O(N) entries and for any set of N items, the insertion process fails at allocating the N items with probability 1/poly(N) **over the random choice of the hash functions**.

Query Time: O(1)

Failure Probability ε: 1/poly(N)

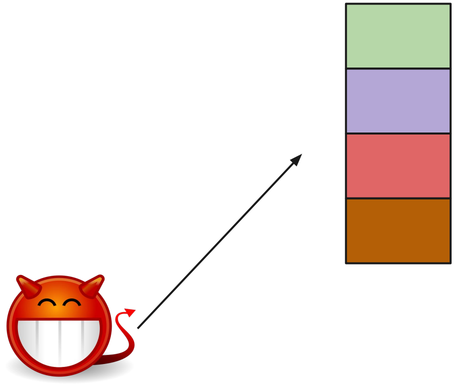
Standard Cuckoo Hashing

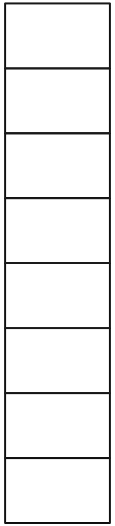


Standard Cuckoo Hashing

Standard Cuckoo Hashing

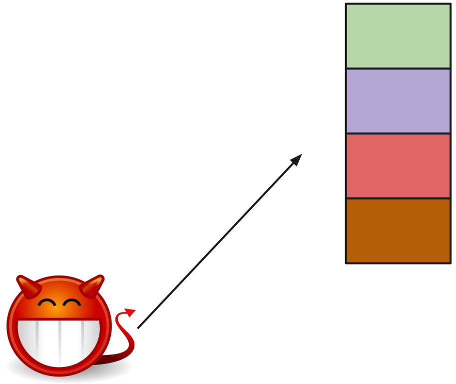
Hash Functions: F, G



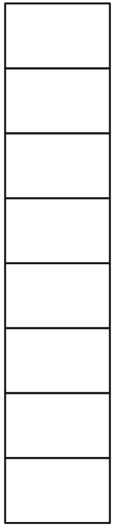


Standard Cuckoo Hashing

Hash Functions: F, G

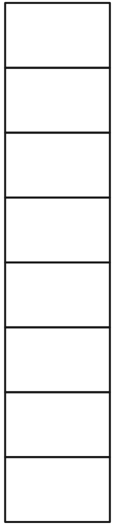


Adversary wins if chosen set of items causes construction failure.

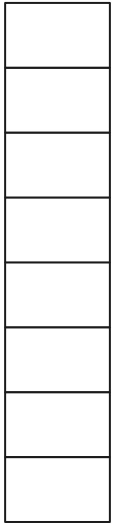


Robust Cuckoo Hashing

Hash Functions: F, G



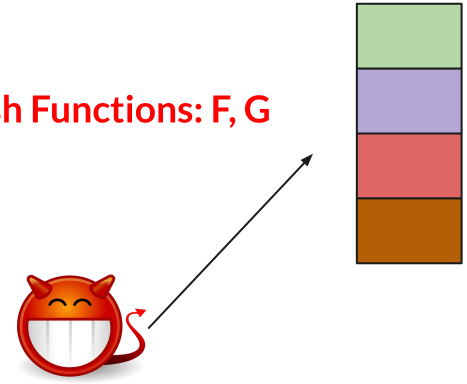
Robust Cuckoo Hashing



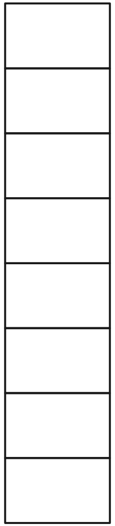
Hash Functions: F, G

Robust Cuckoo Hashing

Hash Functions: F, G



Adversary wins if chosen set of items causes construction failure.



Our Robust Construction

Theorem (Ours). Cuckoo hashing with the following parameters is robust against poly(N) adversaries:

- **$k = f(N) * \log N$ hash functions where $f(N) = \omega(1)$.**
- $b = O(n)$ entries
- **k disjoint tables of size b/k**
- $L = 1$ entry sizes
- $s = 0$ (no overflow stash)

Query Overhead: $O(f(N) * \log N)$

Our Robust Lower Bound

Theorem (Ours). For any cuckoo hashing scheme that is robust against poly(N) adversaries with $b = O(N)$ entries, one of the following must hold:

1. $k = \omega(\log N)$ hash functions
2. Query overhead must be $\Omega(N)$

Applications: Batch Codes and Batch PIR

Explicit Batch PIR	Computational Time	Queries	Error
Subset [IKOS04]	$O(n)$	$q^{O(1)}$	0
Balbuena Graphs [RSDG16]	$O(n)$	$O(q^3)$	0
Pung [AS16]	$4.5n$	$9q$	2^{-20*}
3-way Cuckoo Hashing [ACLS18]	$3n$	$1.5q$	2^{-40*}
Our Work	$O(n \cdot \sqrt{\lambda / \log \log n})$	$O(q)$	$2^{-\lambda}$

Figure 5: A comparison table of explicit blackbox single to batch PIR transformations. The error probability considers queries chosen independently of the hash functions. Asterisks (*) denote experimental error probabilities.

Applications: Re-usable Batch PIR

Explicit Batch PIR	Computational Time	Queries	Adversarial Error
Subset [IKOS04]	$O(n)$	$q^{O(1)}$	0
Balbuena Graphs [RSDG16]	$O(n)$	$O(q^3)$	0
Pung [AS16]	$4.5n$	$9q$	$\geq 1/2$
3-way Cuckoo Hashing [ACLS18]	$3n$	$1.5q$	$\geq 1/2$
Our Work	$O(n \cdot (f(n) + \lambda)), f(n) = \omega(\log n)$	$O(q)$	$2^{-\lambda}$

Figure 6: A comparison table of explicit re-usable batch PIR schemes. Adversarial error ϵ means an adversary running in $\mathsf{poly}(n)$ time cannot find an erring input except with probability ϵ.

Applications: And More

- Private Set Intersection (PSI)
- Volume-Hiding Encrypted Search
- Vector Oblivious Linear Evaluation (VOLE)
- Batch PIR with Private Preprocessing (Batch Offline/Online PIR)

Thank You!

Email: kwlyeo@google.com

ePrint: ia.cr/2022/1455