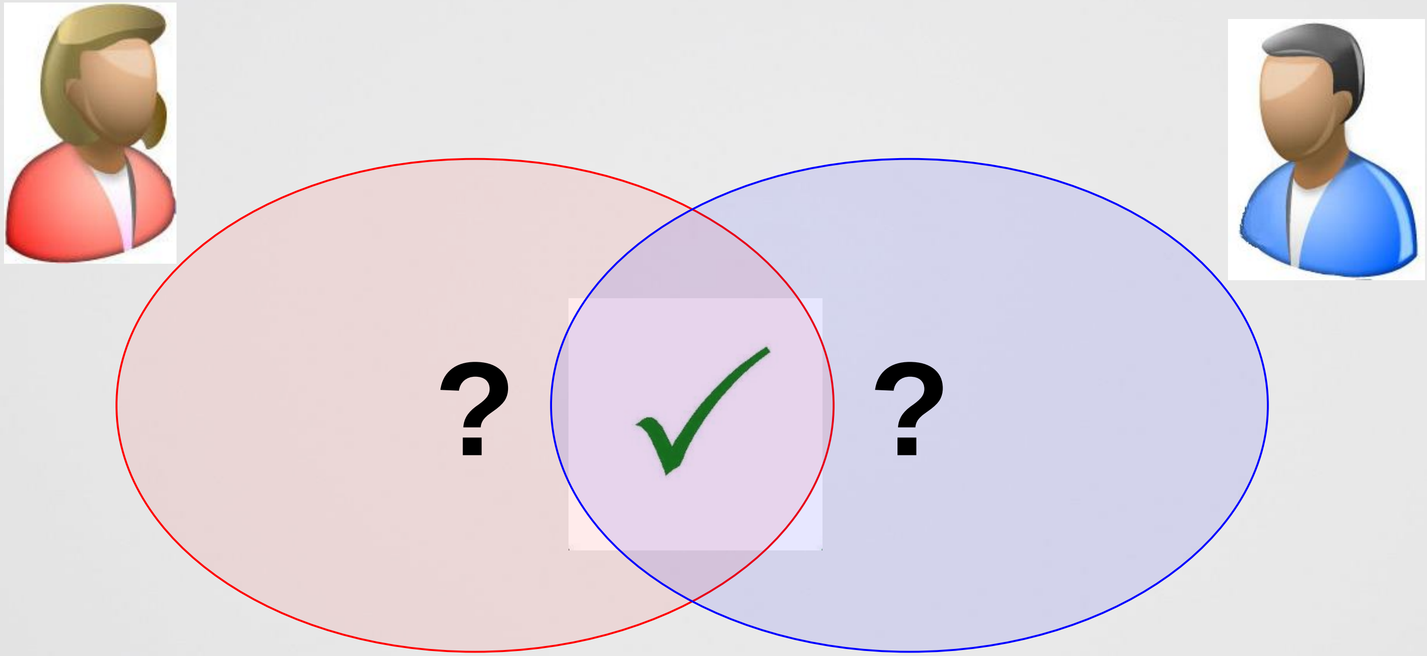
Private Set Intersection (PSI): in the Cloud, or using Circuits

Benny Pinkas

September 10, 2017

Private Set Intersection (PSI)



In this talk

- Computing PSI using linear-size **circuits,** via two-dimensional Cuckoo hashing
  - With Thomas Schneider, Christian Weinert, Udi Wieder.
  - Have efficient implementations for all protocols
  - A very detailed experimental analysis

- PSI of outsourced data in the cloud
  - With Ben Riva
  - Detailed cloud-based experiments

A naïve PSI protocol

- A naïve solution:
  - A has items $x_1,\ldots,x_n$.   B has items $y_1,\ldots,y_n$.
  - A and B agree on a "cryptographic hash function" H()
  - B sends to A: $H(y_1),\ldots, H(y_n)$
  - A compares to $H(x_1),\ldots, H(x_n)$ and finds the intersection

- Does not protect B's privacy if the inputs do not have considerable entropy

Applications of PSI

- **Information sharing**, e.g., intersection of threat information or of suspect lists
- **Matching**, e.g., testing compatibility of different properties (preferences, genomes…)
- Identifying mutual contacts
- **Computing ad conversion rates**

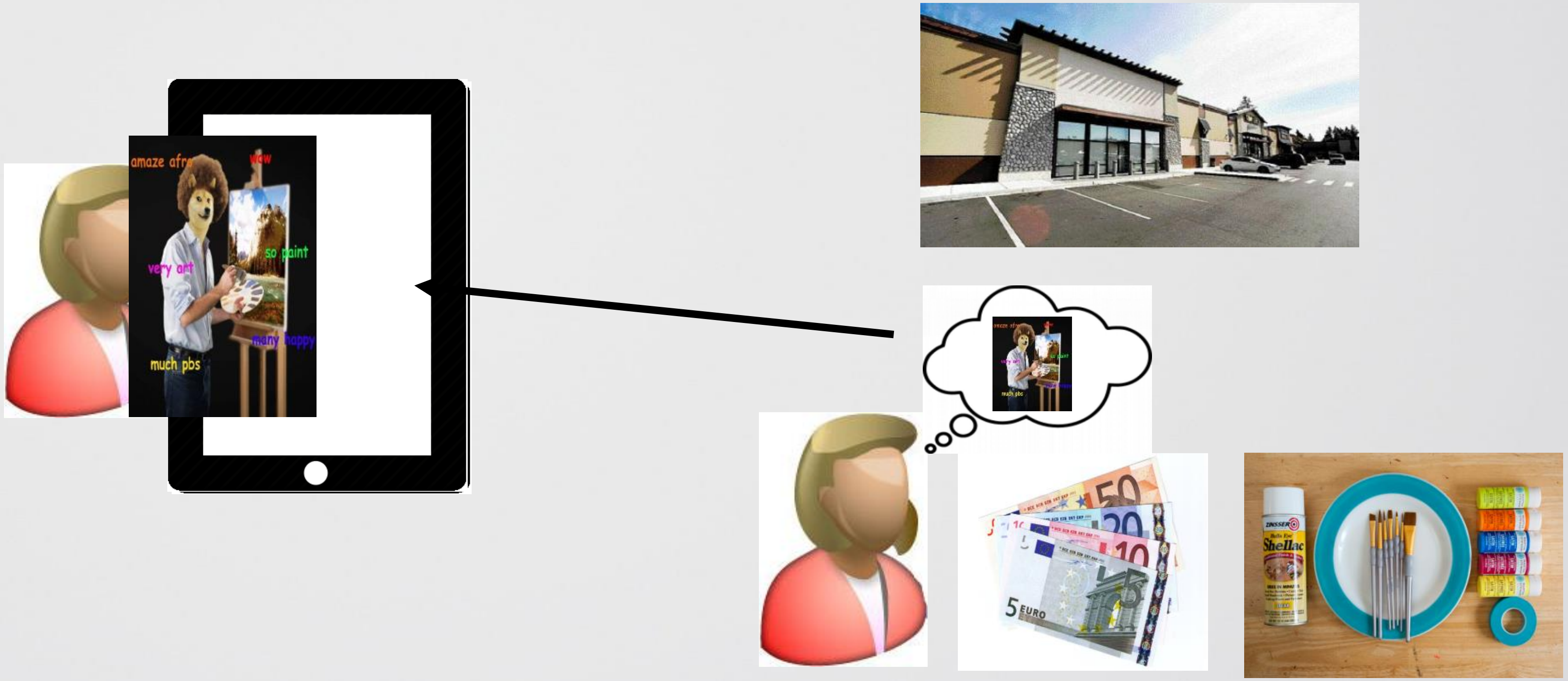
Application: Online Advertising

- Retailers show ads using, e.g., Facebook or Google

- For online web stores, it is easy to measure the effectivity of ads

- For offline shops it is harder

Online **Real-World**



Existing PSI protocols

- Based on the commutativity of Diffie-Hellman [S80, M86, HFH99, AES03]
- Based on blind-RSA [CT10]
- Based on generic MPC and circuits [HEK12,PSSZ15]
- Based on Bloom filters [DCW13]

- Based on Oblivious Transfer and hashing [PSZ14,PSSZ15, KKRT16]

Main challenge

comparing two sets of size n requires n^2 operations

$\Rightarrow$ too many crypto operations

Thunder – when clouds intersect (or, PSI of **outsourced data**)

With Ben Riva

Cloud storage services

Dropbox

Google Drive

OneDrive

iCloud

box

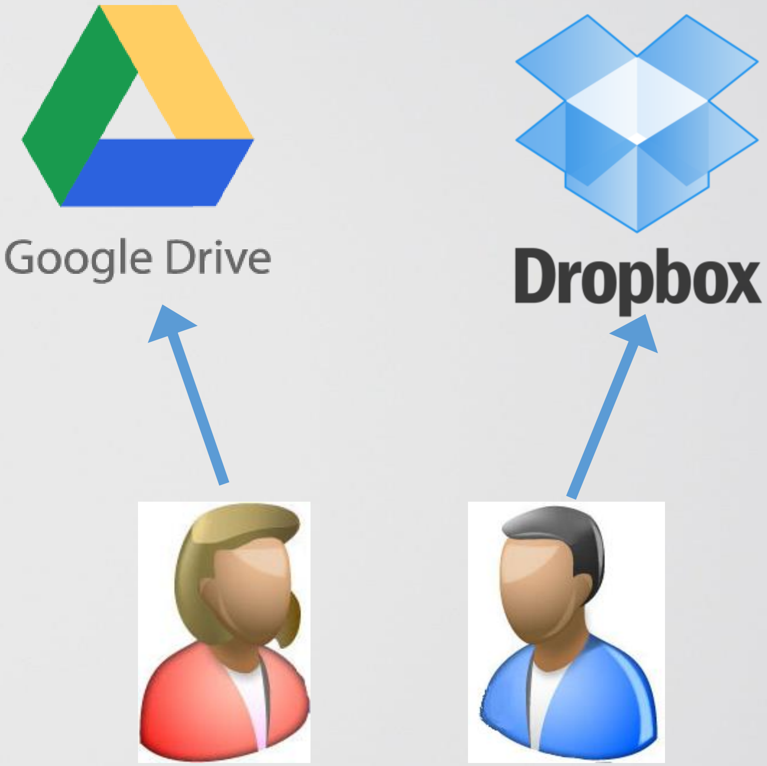
amazon web services™ S3

Google Cloud Storage

Setting

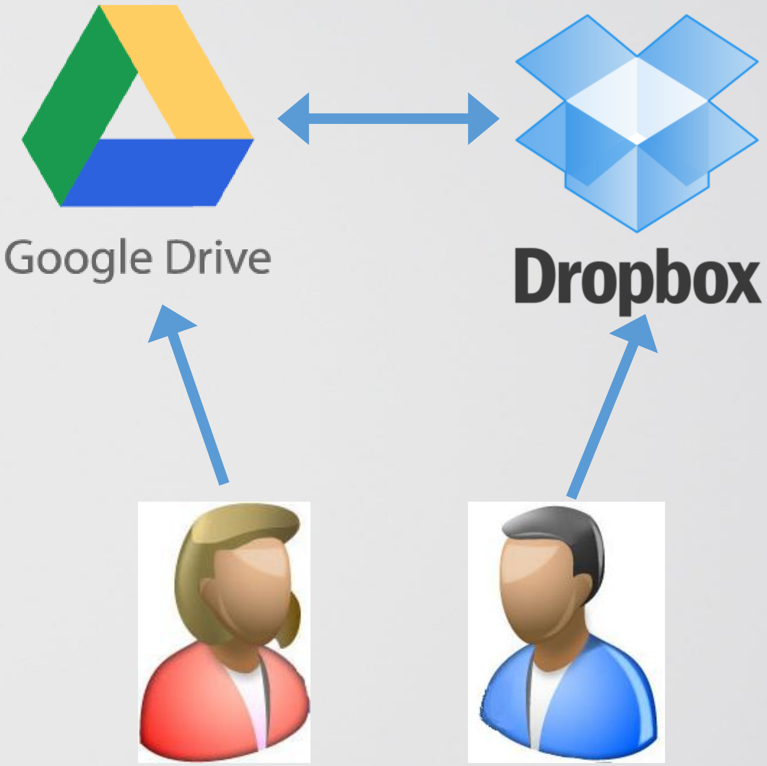
- Users store huge *encrypted* data sets in the cloud
- Want to run an MPC over their data

- MPC protocols are for users that have their input in their possession
- Downloading the data before running the MPC is costly



Motivation for running MPC in the cloud

- Why use a cloud service to run an MPC for you?
  - The data is already stored in the cloud
  - Can achieve very low latency by utilizing the elastic computing resources of the cloud (namely, use hundreds of cores and benefit from parallelism)



Requirements

- Clients encrypt their data before uploading it
- Do not know in advance with whom they will run MPC

- Afterwards, they only need to invest an effort that is **sublinear** in the input size

Single vs. multiple cloud services

- Simple solution given non-colluding clouds:
  - Each client sends encrypted data to one cloud service, key to another.
  - The cloud service run an MPC between themselves.

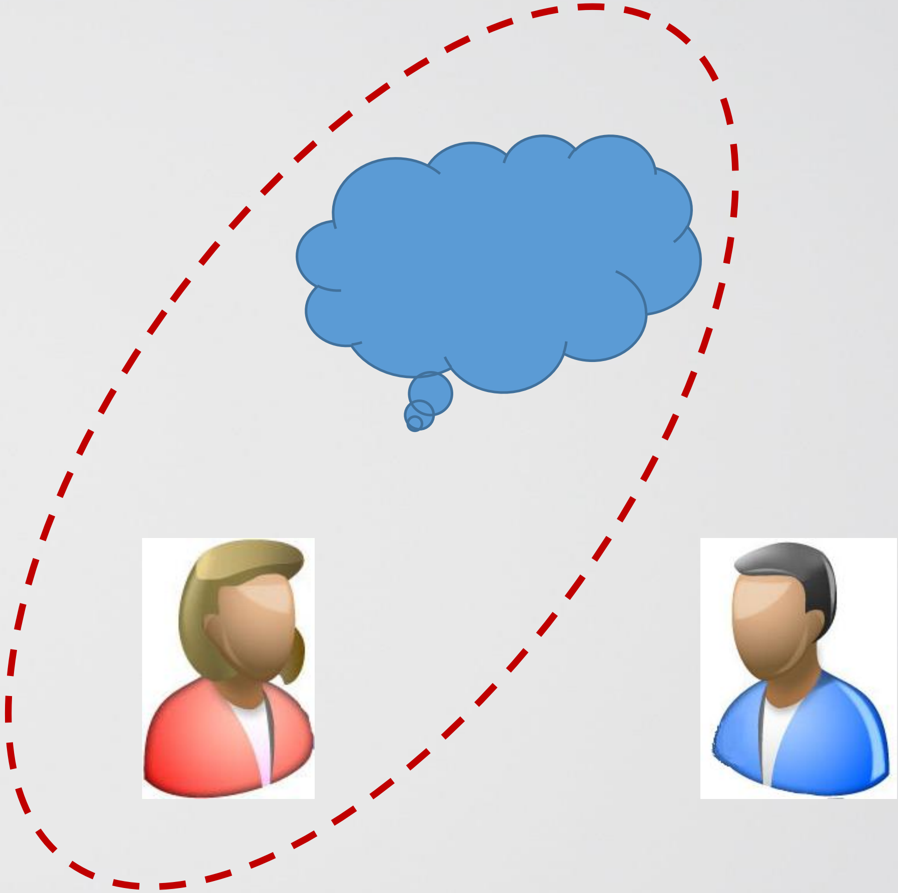
- It is better not to depend on non-collusion between clouds
  - Clients cannot verify that clouds do not collude
  - It is expensive/complicated to setup trust relationships with multiple clouds

- Therefore we assume that cloud services might collude. This is **equivalent** to assuming that a **single** cloud service is used by all clients.

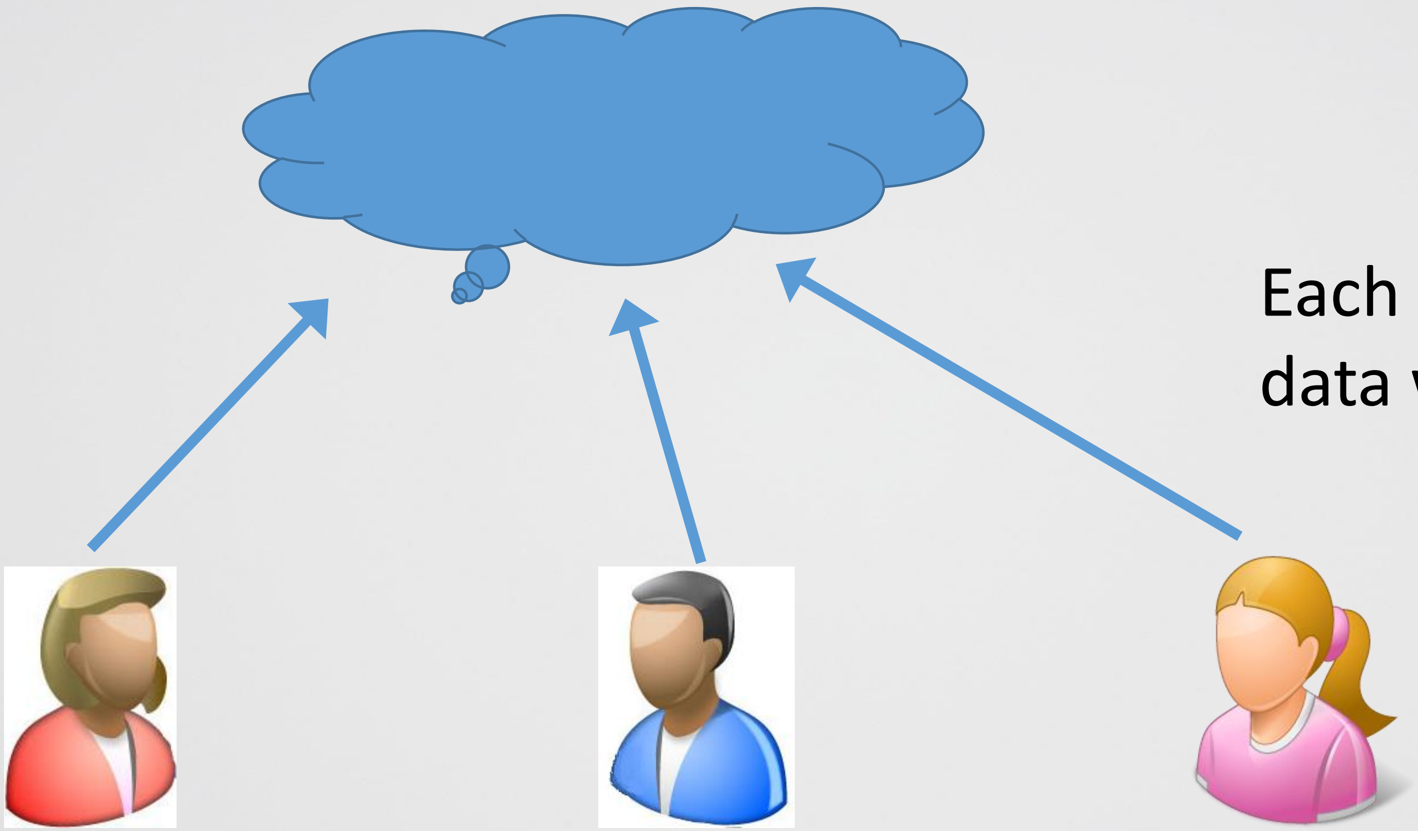
No client-cloud collusion

- We assume that **clients do not collude with the cloud**.

- Otherwise, Alice might collude with the cloud, and this will essentially be a two-party computation between Bob and Alice+cloud.
- The only known 2PC protocols with sublinear communication are based on FHE.

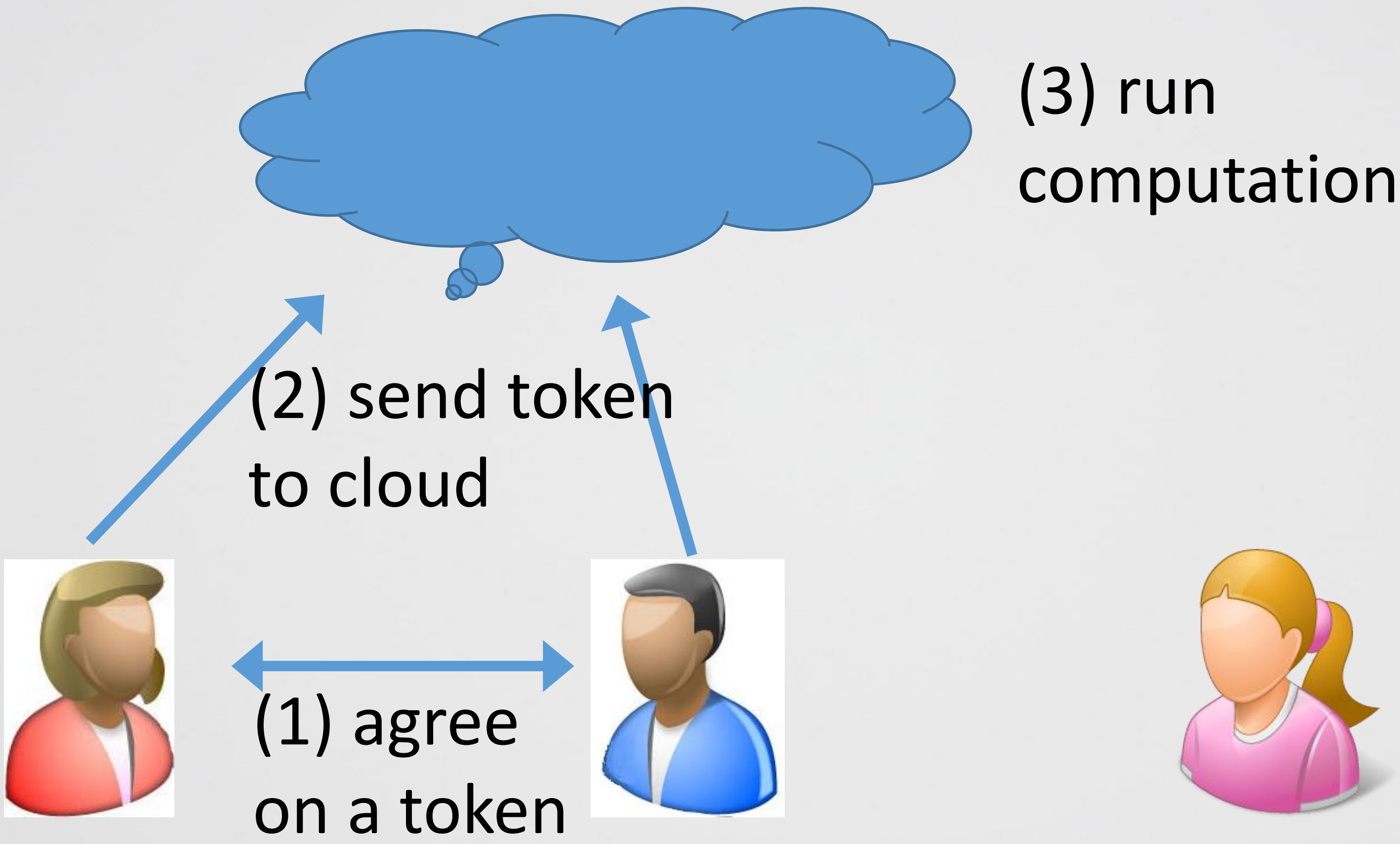


Clients upload data

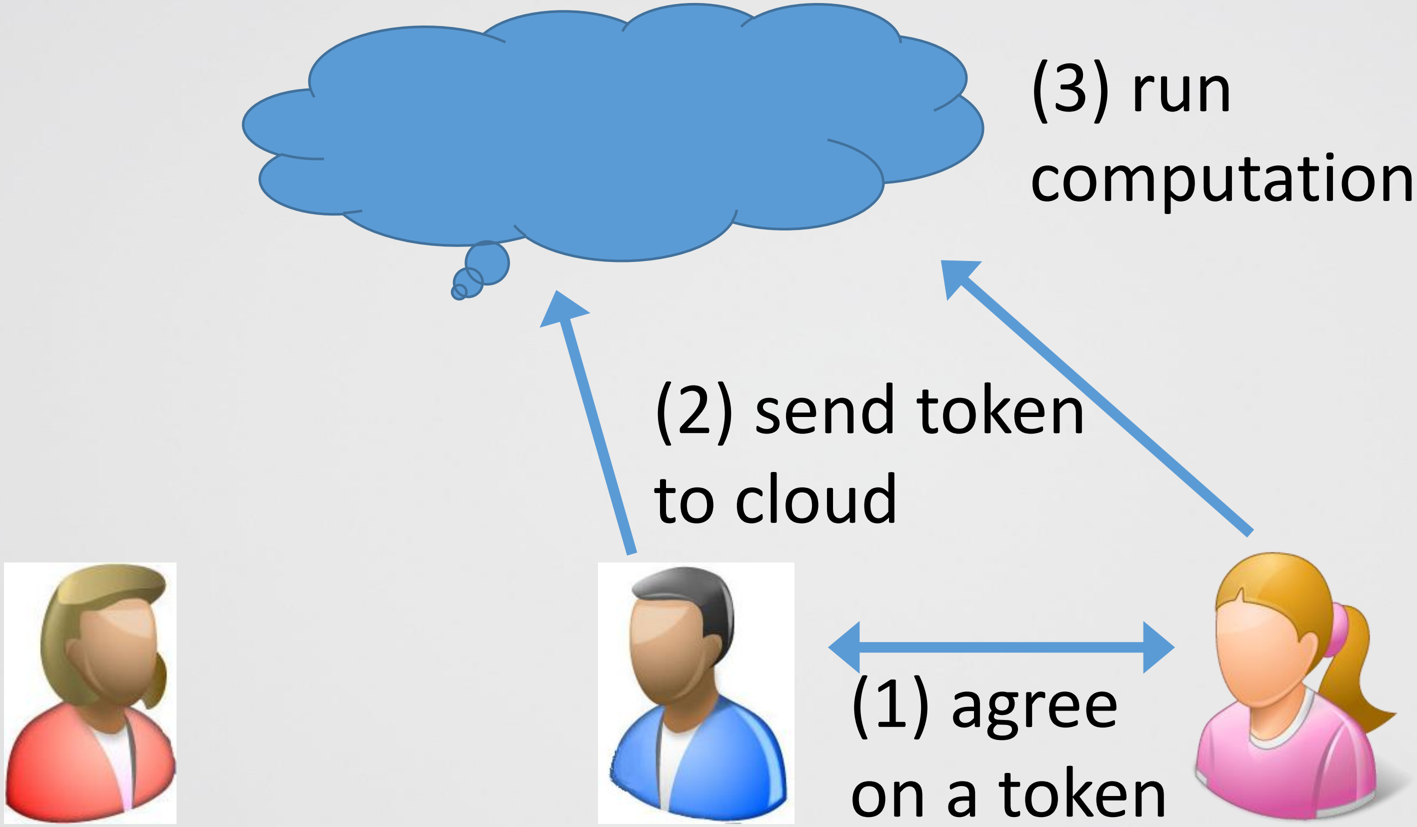


Each client encrypts its data with its own key

Alice and Bob wish to run a computation

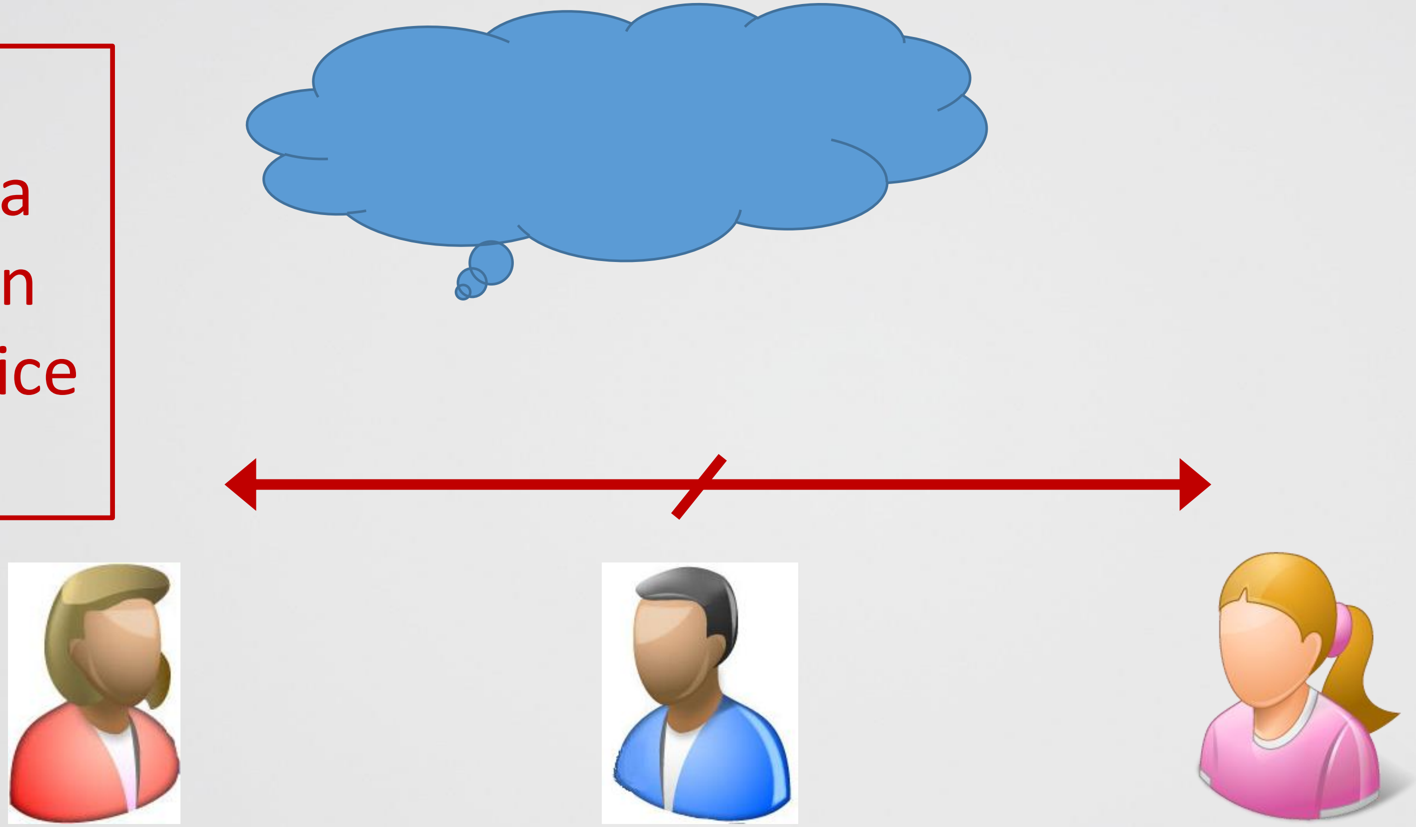


Bob and Carol wish to run a computation



Bob and Carol wish to run a computation

Cloud still cannot run a computation between Alice and Carol



Why is this interesting?

- Need: the outsourced storage market is booming
- Novelty: current MPC techniques (except FHE) are inadequate for the cloud setting
- Performance: we achieve latency similar to that of best PSI protocols, by using mass parallelism. (Most clients can afford <u>renting</u>, but not <u>buying</u> this computing power)
- PSI is the only problem we know how to to solve in this setting

Related work

- “On the fly MPC on the cloud via multi-key FHE” [LTV12]
- Protocols with client work of $\Theta(n)$
  - Server aided MPC [KMR11,KMR12]
  - Server assisted PSI [K12]
  - MPC between three parties [BGW,CCD]
- Proxy re-encryption [AFGH06]
  - Can convert an encryption to an encryption under a different key
  - But cannot compare the two encryptions since they use different randomness

Bilinear maps

- G_1, G_2, G_T are groups of prime order q
- $e: G_1 \times G_2 \rightarrow G_T$ s.t.
  - If g_1, g_2 are generators of G_1, G_2, respectively, then $e(g_1, g_2)$ generates G_T
  - $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$
- We use a Type-III pairing: There is no homomorphism from G_2 to G_T
- The SXDH assumption [BGMM05,GrothSahai08]: Both G_1 and G_2 are DDH hard groups.

The protocol

- Generate **parameters** for G_1, G_2,G_T.
  - g is a generator of G_1
  - A function $H()$: $\{0,1\}^* \to G_2$.
- **Upload** by user Pi
  - Picks a random key $Ki \in [q]$
  - Encrypts each item x by computing $(H(x))^{Ki} \in G_2$

The protocol

- Generate **parameters** for G_1, G_2,G_T.
  - g is a generator of G_1
  - A function $H(): \{0,1\}^* \rightarrow G_2$.
- **Intersection** of the data of Pi and Pj:
  - Pi and Pj agree on a key K. Send $g^{K/Ki}$, $g^{K/Kj}$ to the server, respectively.
- The server
  - For each item $(H(x))^{Ki}$ uploaded by Pi, computes $e(g^{K/Ki}, (H(x))^{Ki}) = (H(x))^K \in G_T$
  - For each item $(H(y))^{Kj}$ uploaded by Pj, computes $e(g^{K/Kj}, (H(y))^{Kj}) = (H(y))^K \in G_T$
  - Check the intersection of the two computed sets

Security

- Security proof in the random oracle model based on SXDH
  - Main property: values computed in the intersection of Pi and Pj ($(H(x))^K \in G_T$), cannot be compared with values computed in the intersection of Pi and another party ($(H(x))^{K'} \in G_T$).

  - It is crucial that there is no homomorphism from G_2 to G_T

  - Important (and hard) property: given tokens for P_i,P_j, and for P_j,P_k, it is impossible to compute intersection of P_i,P_k.

Extensions

- Computing encryptions and pairings is highly parallelizable

- Can also **preprocess** the work of the intersection step, so that in realtime compute exponentiations instead of pairings

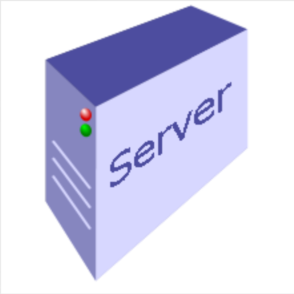
- Computing the intersection of **three (or more) parties**
  - Send tokens $g^{R1/K1}$, $g^{R2/K2}$, $g^{-(R1+R2)/K3}$
  - The server computes $(H(x))^{R1}$, $(H(x))^{R2}$, $(H(x))^{-(R1+R2)}$ and looks for triplets of items that multiply to 1

The Thunder prototype

- Implemented in Microsoft Azure (F16 Linux machines with 16 cores)

- Pairings were implemented using MIRACL 4.0
  - Curve with 80 bit security (CP curve with K=2)
- Batching pairings: many pairings with the same element of G_2
  - Reduced run time by 50% to about 1ms / pairing.

Uploading data



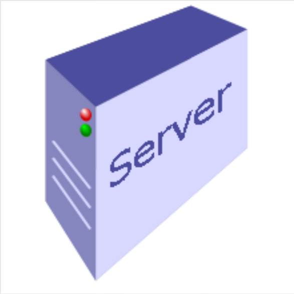
Data stored in MySQL database

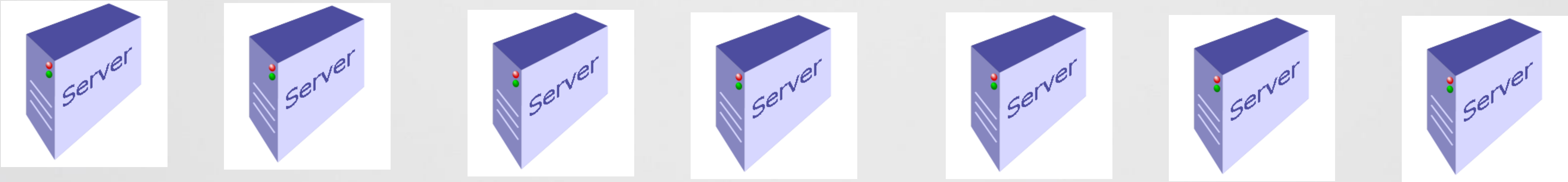
Uploads encrypted data to server

Client encrypts its data

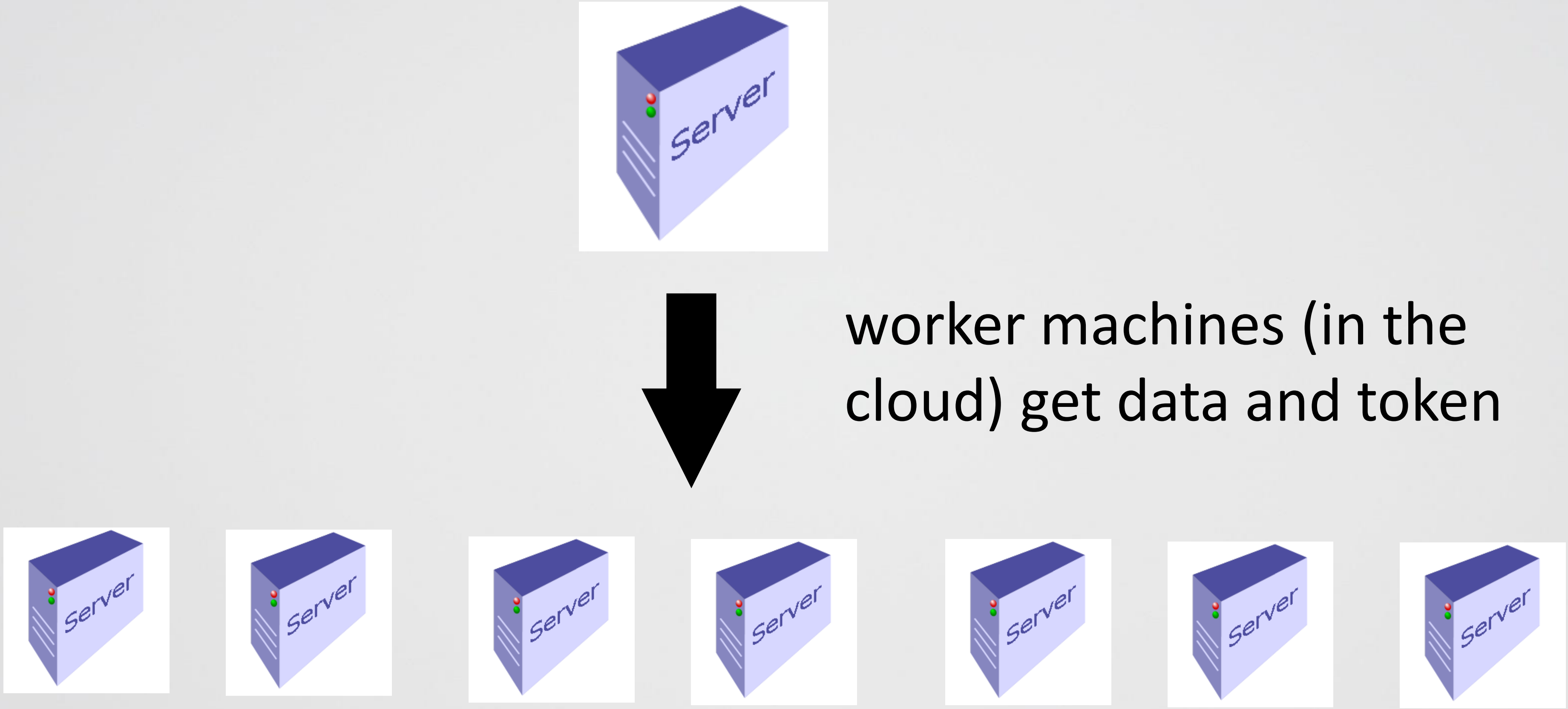
Computing the intersection

Receives intersection token
from a pair of clients



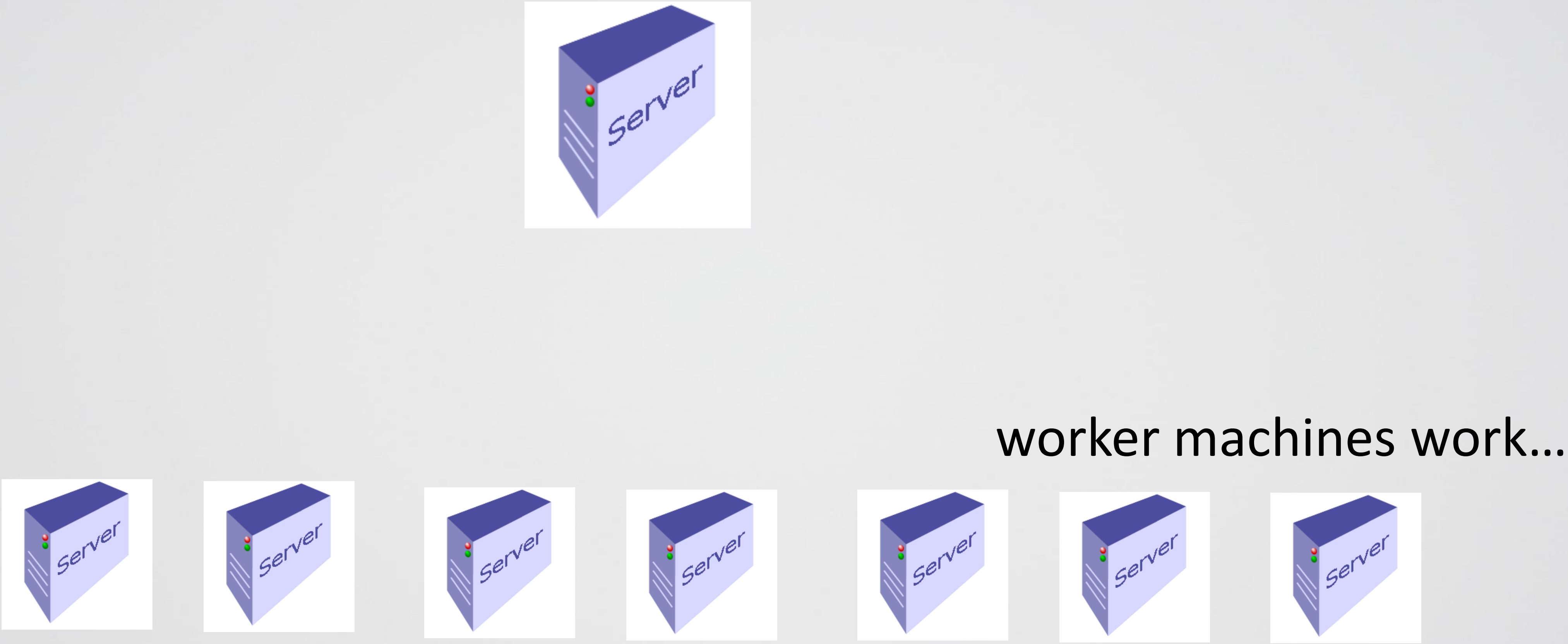


Computing the intersection

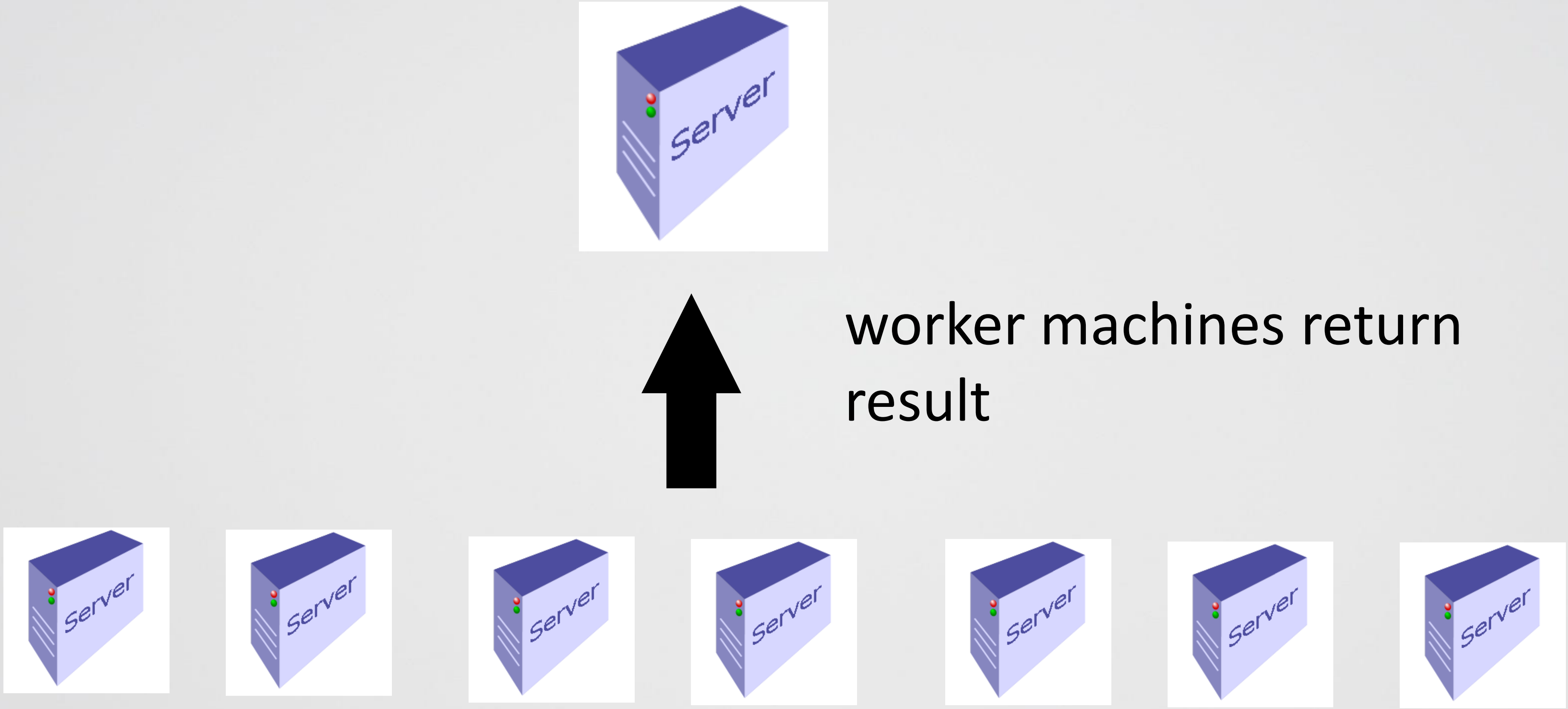


worker machines (in the cloud) get data and token

Computing the intersection

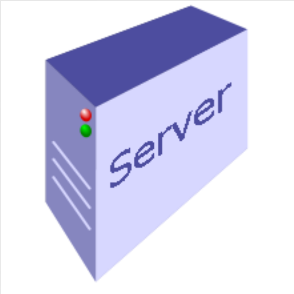


Computing the intersection

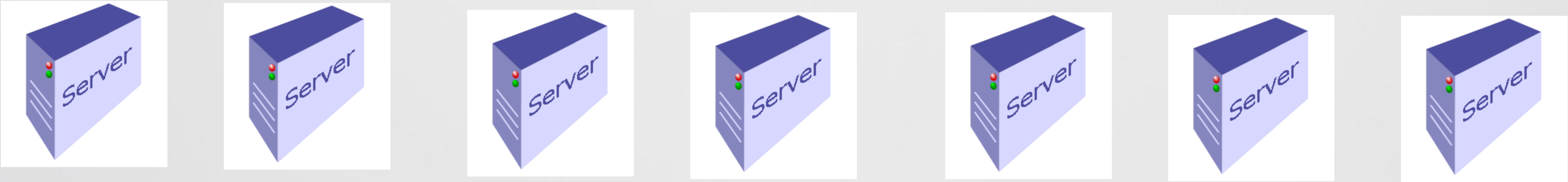


worker machines return result

Computing the intersection



server computes the final intersection results (using C++ `unordered_sets` API)



Results (msec)

Data size	# of Workers	Down-load	Compute	Upload	Total	CPU hours
1M	10	49	11282	1501	12833	0.036
5M		121	61325	2943	64391	0.179
10M		330	125982	4854	131168	0.364
1M	50	40	2367	972	3381	0.047
5M		134	11247	1587	12700	0.176
10M		255	24844	1972	27072	0.376
1M	100	35	1278	800	2115	0.059
5M		75	5721	1225	7022	0.195
10M		109	11352	1474	12936	0.359

Faster than best PSI OT-based protocols [PSSZ15,KKRT16]

Results (msec)

Data size	# of Workers	Down-load	Compute	Upload	Total	CPU hours
1M	10	49	11282	1501	12833	0.036
5M		121	61325	2943	64391	0.179
10M		330	125982	4854	131168	$\boxed{0.364}$
1M	50	40	2367	972	3381	0.047
5M		134	11247	1587	12700	0.176
10M		255	24844	1972	27072	$\boxed{0.376}$
1M	100	35	1278	800	2115	0.059
5M		75	5721	1225	7022	0.195
10M		109	11352	1474	12936	$\boxed{0.359}$

Total CPU time is ~same regardless of # of workers. Latency is improved with more workers.

Results (msec)

Data size	# of Workers	Down -load	Compute	Upload	Total	CPU hours
1M	10	49	11282	1501	12833	0.036
5M		121	61325	2943	64391	0.179
10M		330	125982	4854	131168	0.364
1M	50	40	2367	972	3381	0.047
5M		134	11247	1587	12700	0.176
10M		255	24844	1972	27072	0.376
1M	100	35	1278	800	2115	0.059
5M		75	5721	1225	7022	0.195
10M		109	11352	1474	12936	0.359

Most of the latency

- 10 workers: 88%-96%
- 50 workers: 70%-92%
- 100 workers: 60%-88%

Results

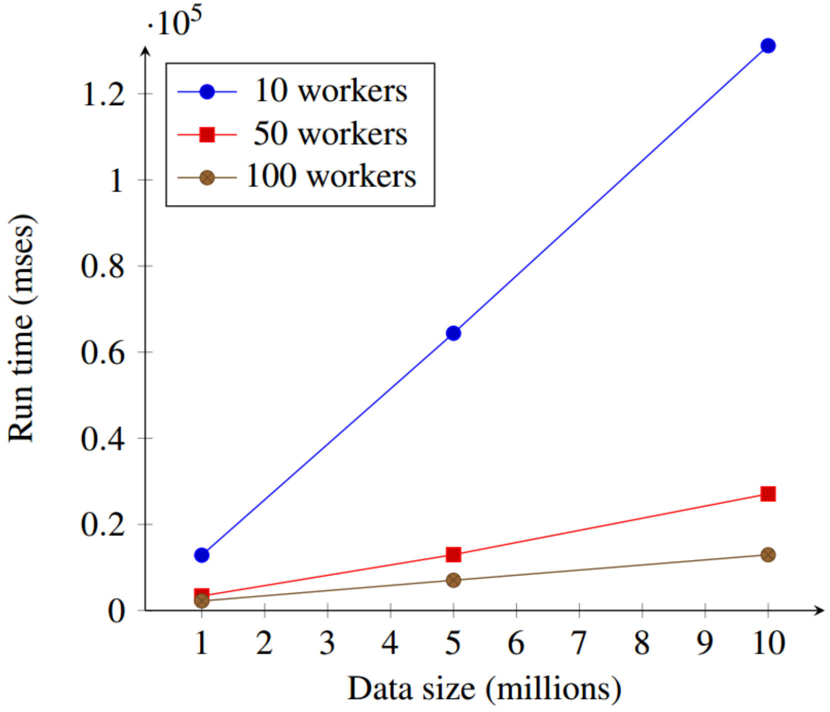


Figure 3: Run times for different data sizes.

Cost of F16 machine is \$0.80 / hour

Therefore, computing PSI on sets of 10^6 items costs

- \$0.0286 with 10 workers
- \$0.0469 with 100 workers

Computing PSI on sets of 10^7 items costs between \$0.286 to \$0.299

Running experiments in the cloud

- Distributing data to workers and gathering the results is not simple
  - Different ideas we had were not compatible with the existing API
- AWS does not guarantee which machine will run your program
  - Therefore used Azure
- Network congestion depends on other users and on time of day
- It’s expensive

Linear size **circuit-based** PSI via two-dimensional Cuckoo hashing

With Thomas Schneider, Christian Weinert, Udi Wieder

Existing PSI protocols

- Based on the commutativity of Diffie-Hellman [S80, M86, HFH99, AES03]
- Based on blind-RSA [CT10]
- Based on generic MPC and circuits [HEK12,PSSZ15]
- Based on Bloom filters [DCW13]

- Based on Oblivious Transfer and hashing [PSZ14,PSSZ15, KKRT16]

<u>Main challenge</u>

comparing two sets of size n requires n^2 operations

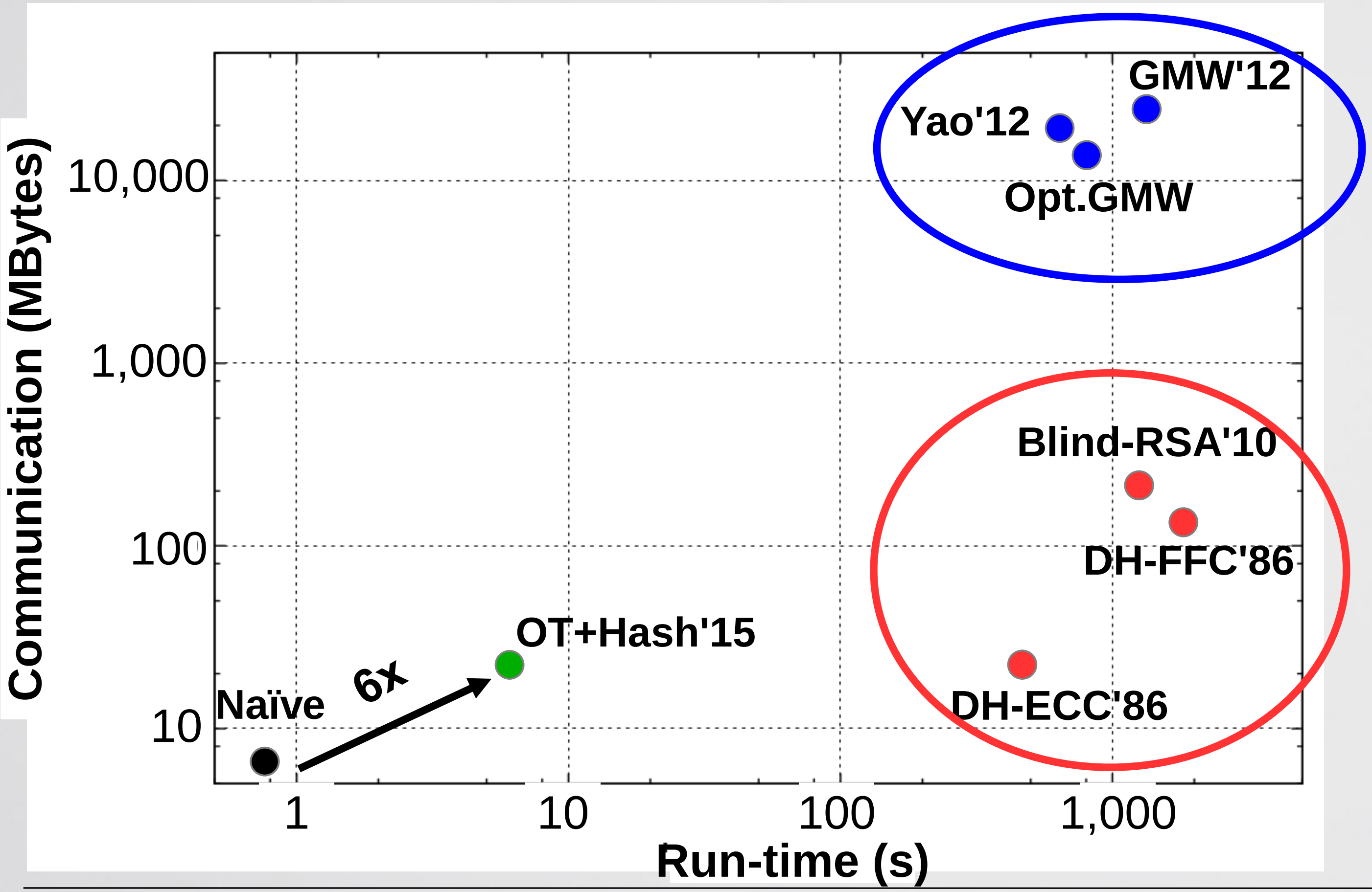
$\Rightarrow$ too many crypto operations

Recent constructions [PSZ1,PSSZ15,KKRT16]

- PSI is “equivalent” to oblivious transfer
- Realized that oblivious transfer extension (which is very fast) can enable very efficient PSI
- Used different hashing ideas to dramatically reduce the overhead of PSI

Performance Classification [PSZ]

- PSI on $n = 2^{18}$ elements of $s=32$-bit length for 128-bit security on Gbit LAN



Circuit-Based:
- high run-time & communication,
but easily extensible to arbitrary functions

PK-Based:
- high run-time
+ best communication

OT-Based: good communication and run-time

Motivation for using circuits

- PSI is a specific case of secure two-party computation:

  Two parties with private inputs want to compute a function of their inputs while leaking no other information

- There are generic protocols ("MPC") for securely computing **any function**, as long as it is expressed as a **binary circuit**

Motivation for using circuits

Why use a circuit-based generic protocol for computing PSI?

- Adaptability
  - Instead of hiring a crypto expert, hire an undergrad
- Existing code base
- Existing applications compute functions over the results of PSI
  - E.g., computing the sum of revenues from ad views

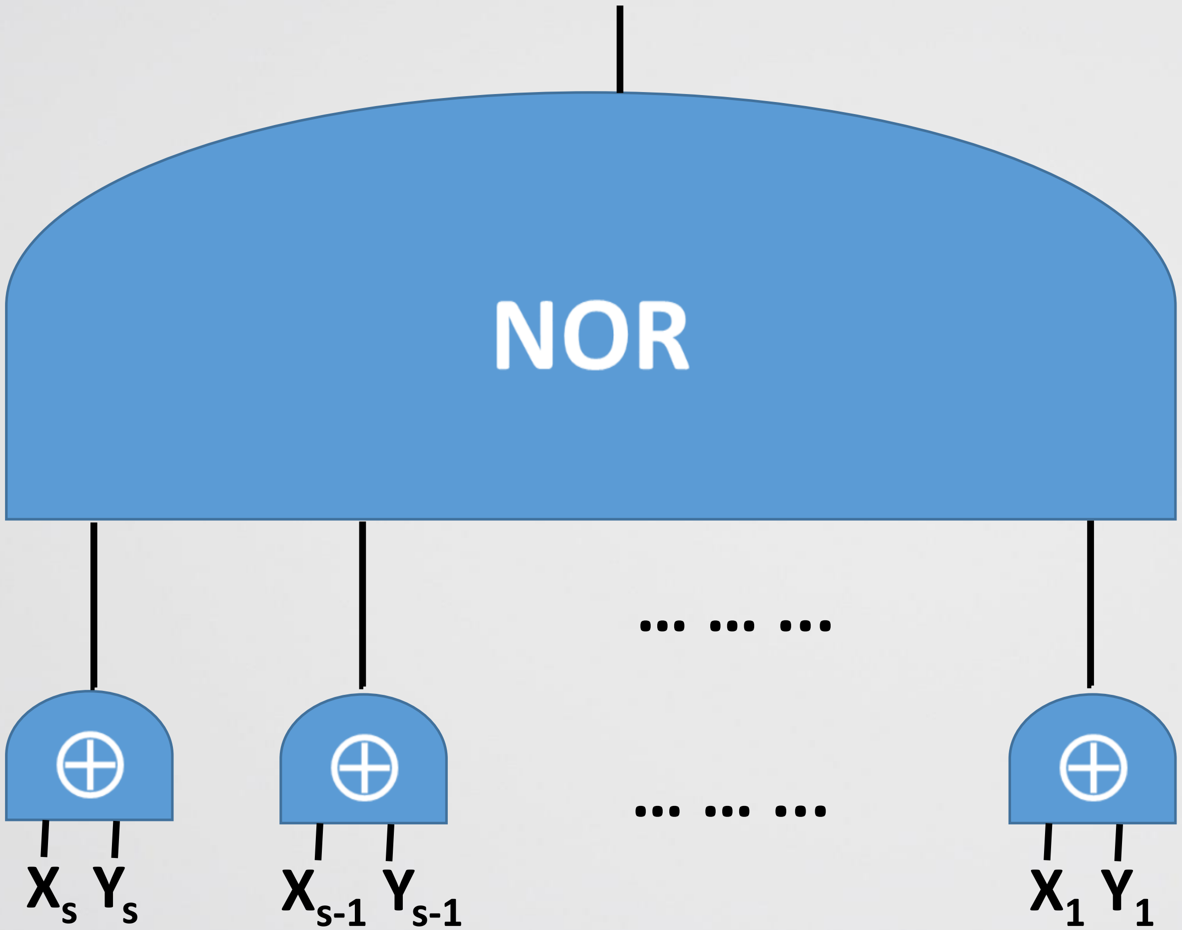
A circuit based protocol

- There are generic protocols for securely computing **any function** expressed as a Binary circuit
  - GMW, Yao,…
  - Parties do not learn anything but the required output
  - The overhead depends on the size of the circuit

- A naïve circuit for PSI uses n^2 comparisons of words

- Can we do better?

A circuit comparing two s-bit values



$\Leftarrow$ **s-1 gates**

$\Leftarrow$ **Free XORs**

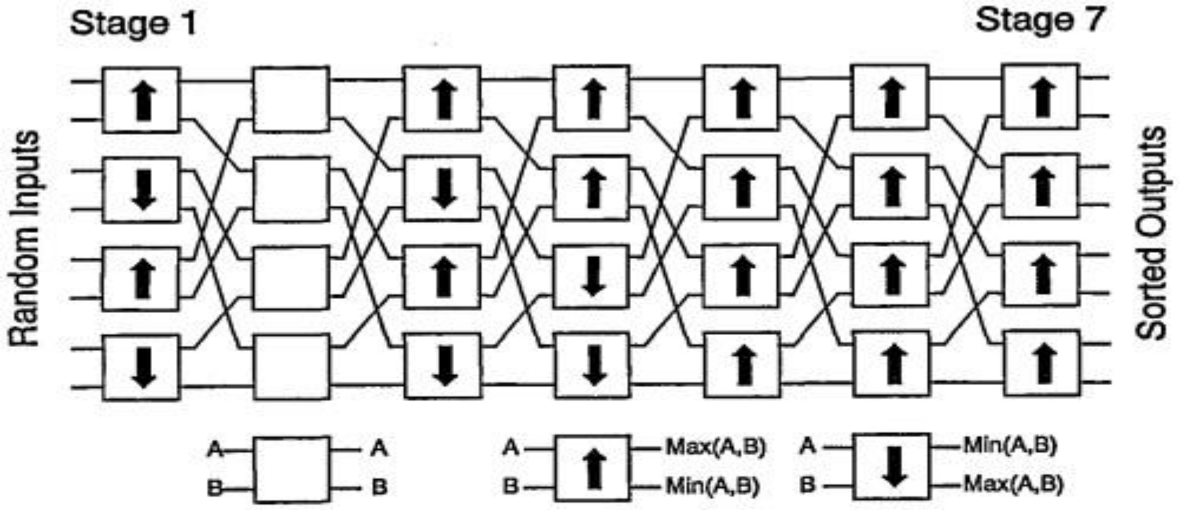
Comparing two items is efficient

Our goal is to arrange two sets of n items so that the intersection can be computed with as few comparisons as posible

Sorting networks

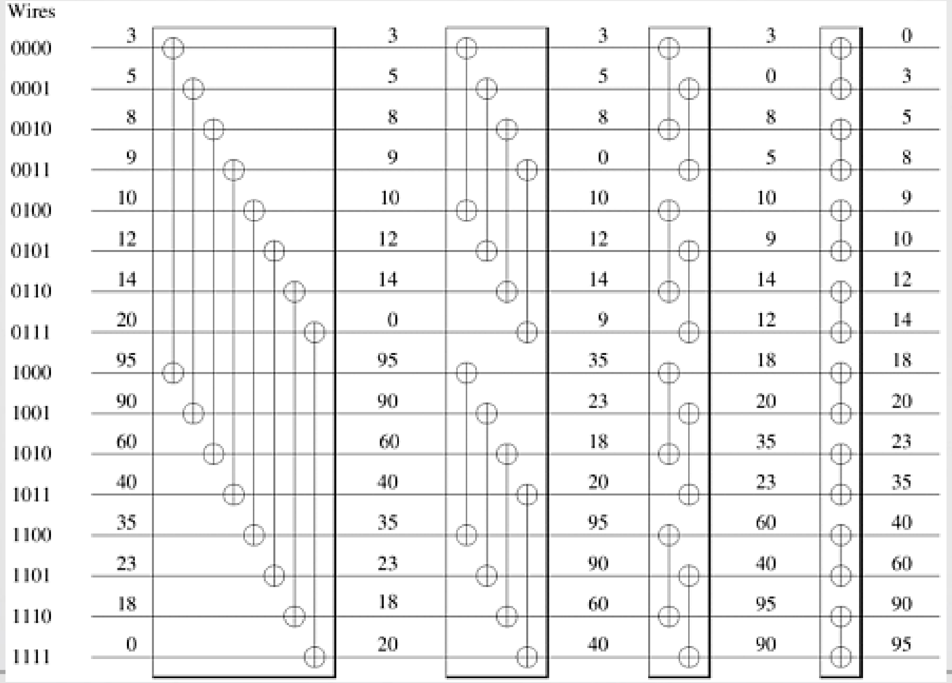
- An algorithm that sorts values using a **fixed sequence of comparisons**

- Can be thought of as a network of wires and comparator modules



A circuit based PSI protocol [HEK12]

- A PSI circuit that has three steps
  - Sort: merge two sorted lists using a bitonic merging network [Bat68]. Uses $n\log(2n)$ comparisons.



A circuit based PSI protocol [HEK12]

- A circuit that has three steps
  - Sort: Merge two sorted lists using a bitonic merging network [Bat68]. Computes the sorted union using $n\log(2n)$ comparisons.
  - Compare: Compare adjacent items. Uses $2n$ equality checks.
  - Shuffle: Randomly shuffle results using a Waxman permutation network [W68], using $\sim n\log(n)$ swappings.
  - Overall Computes $O(n\log n)$ comparisons.
    Uses $s\cdot(3n\log n + 4n)$ AND gates. (s is input length)

The Algorithmic Challenge

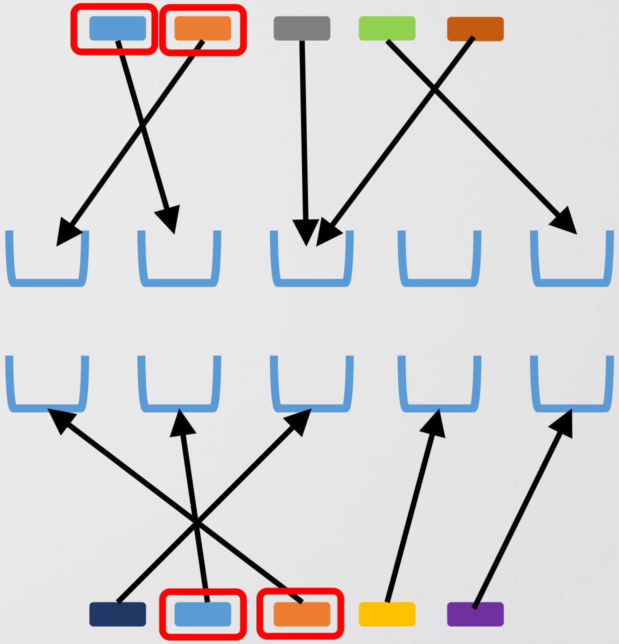
- Goal: Find the smallest circuit for computing PSI
  - Alice and Bob can prepare their inputs
  - Circuit must not depend on data!
- Any symmetric function of the intersection could be added on top
  - The **size** of the intersection, or whether size is greater than some **threshold**, potentially after adding noise to ensure **differential privacy**
  - **Sum** of values associated with the items in the intersection
- Minimize # of comparisons (and length of items)

Contributions

- O(n) circuit-based PSI
  1. A construction with O(n) (*) **provable** asymptotic overhead
     (*) ω(n) if failure probability should be negligible
  2. A construction with O(n) **experimentally verified** overhead, with very small constants
- Implementation and experiments
  - Run time is (surprisingly) better than that of a former O(n logn / loglogn) construction
- New analysis of Cuckoo hashing

Hashing

- Suppose each party uses a hash function H(), (known to both parties) to hash his/her n items to n bins.
  - Then obviously if Alice and Bob have the same item, both of them map it to the same bin
  - Need only compare matching bins



- The problem
  - Some bins have more items than others
  - Must hide how many items were mapped to each bin

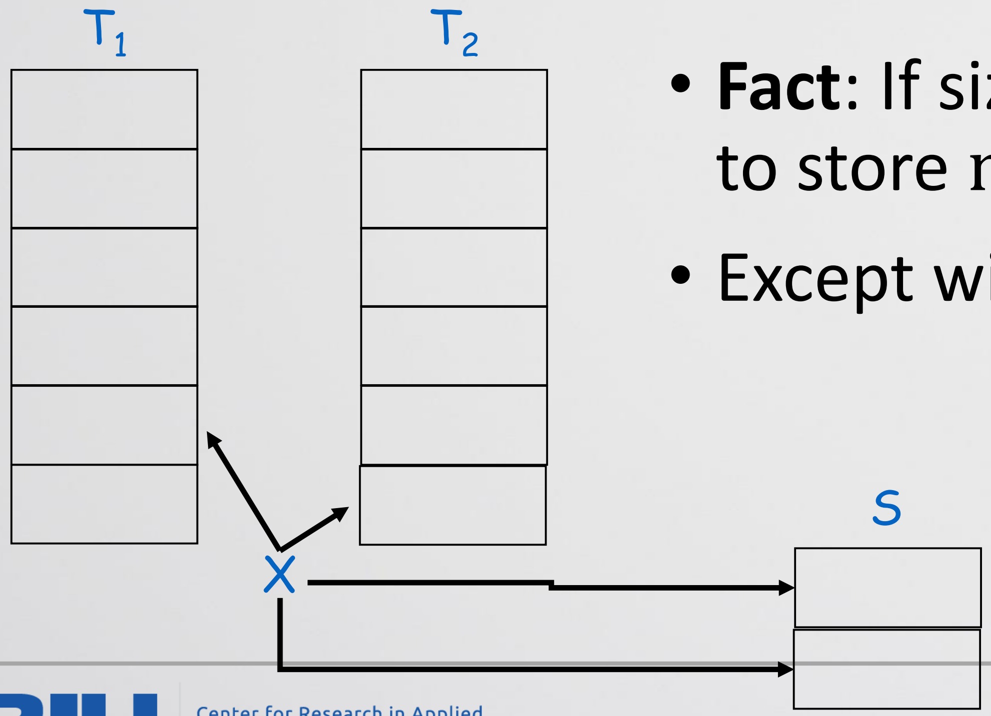
Hashing

- Solution
  - Pad each bin with dummy items
  - so that all bins are of the same size as the most populated bin

- Mapping n items to n bins
  - The expected size of a bin is $O(1)$
  - The maximum size of a bin is whp $O(\log n/\log\log n)$
  - The resulting size of a circuit is …

Cuckoo Hashing with a Stash [PR01], [KMW08]

- Tables T_1, T_2 and stash S
- Hash functions h_1, h_2
- Invariant: Store x in $\mathrm{T}_1[\mathrm{h}_1(\mathrm{x})]$ or in $\mathrm{T}_2[\mathrm{h}_2(\mathrm{x})]$ or in S



- **Fact**: If size of table $> (1+\epsilon)n$ then it is possible to store n items and keep the invariant
- Except with probability $n^{-(s+1)}$

- Slightly more than 2n table entries
- Each of size 1

Handling the Error Probability

- A stash of size s fails with probability $O(n^{-(s+1)})$
- In PSI this results in a (minor?) privacy/accuracy breach

- What should be the failure probability?

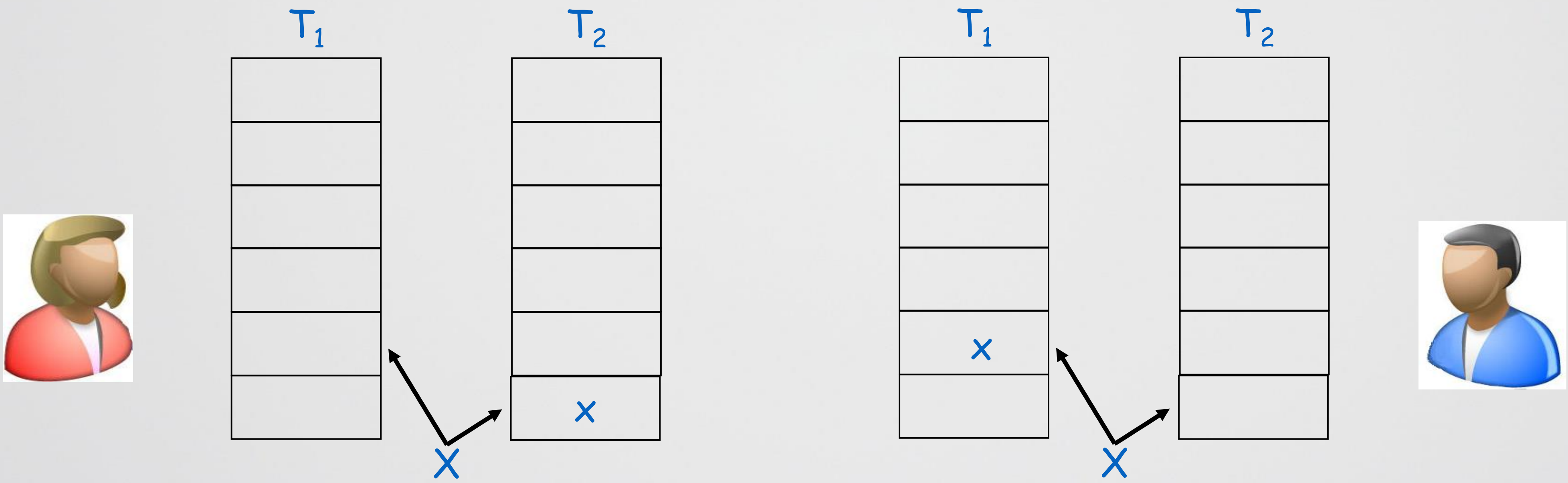
Handling the Error Probability

- A stash of size s fails with probability $O(n^{-(s+1)})$
- In PSI this results in a (minor?) privacy/accuracy breach

- What should be the failure probability?
- Smaller than 2^{-Stat}, e.g. 2^{-40} ?
  - $s = O(1)$ (but what is the exact size?)
- Negligible in n ?
  - $s = \omega(1)$

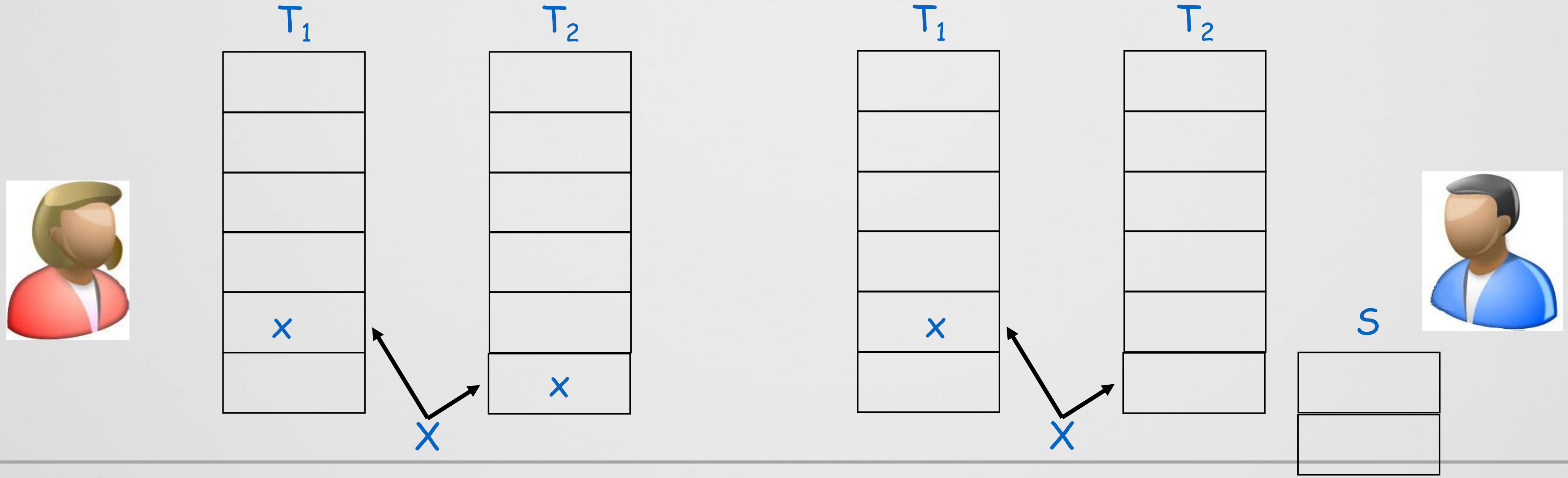
Cuckoo Hashing – can it help?

- What if each party stores its items using CH
  - Can we get O(n) comparisons?
  - No. Alice may store x in T_2 while Bob in T_1



[FNP04], [PSSZ15]

- Alice places its items in **both** tables. Bob uses Cuckoo hashing.
  - In Alice's tables the buckets are of size O(log n/ loglog n)
  - Total of O(n log n / loglogn) comparisons + O(n) for Bob's stash
  - "Permutation based hashing" can be used to store only short values

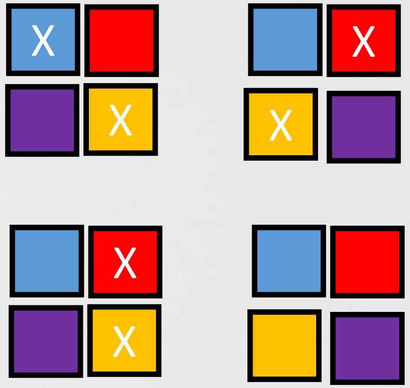


The New Constructions

An Asymptotic Solution

Mirror based PSI:

- 8 tables, of total size $8(1+\varepsilon)n$
- Organized as 4 columns of 2 tables
- Bob maps each of his items to one table in each column (using simple CH)
- Alice maps each of her items to both tables in exactly one column

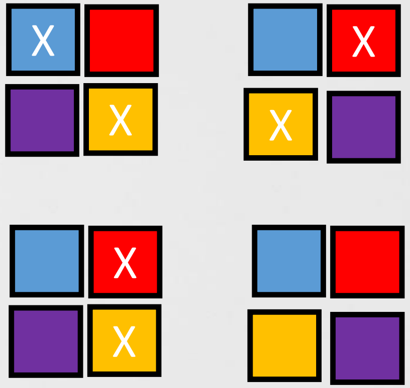


- Now build a circuit which compares each entry in Bob's tables to the corresponding entry in Alice's tables

An Asymptotic Solution

Mirror based PSI:

- 8 tables, of total size $8(1+\varepsilon)n$
- Organized as 4 columns of 2 tables
- Bob maps each of his items to one table in each column (using simple CH)
- Alice maps each of her items to both tables in exactly one column



Circuit size:

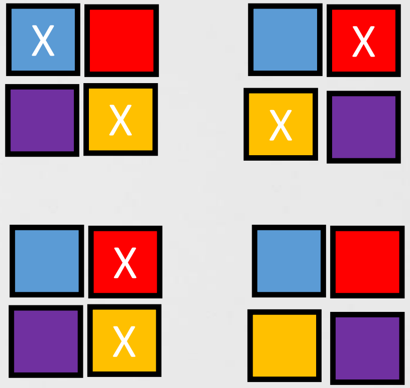
- $8(1+\varepsilon)n$
- Plus a constant (or $\omega(1)$) size stash per each table…

- Now build a circuit which compares each entry in Bob's tables to the corresponding entry in Alice's tables

An Asymptotic Solution

Mirror based PSI:

- 8 tables, of total size $8(1+\varepsilon)n$
- Organized as 4 columns of 2 tables
- Bob maps each of his items to one table in each column (using simple CH)
- Alice maps each of her items to both tables in exactly one column



Circuit size:

- $8(1+\varepsilon)n$
- Plus a constant (or $\omega(1)$) size stash per each table…

- Now build a circuit which compares each entry in Bob's tables to the corresponding entry in Alice's tables

- Analysis is based on known properties of Cuckoo hashing ☺
- But the constants are not small ☹

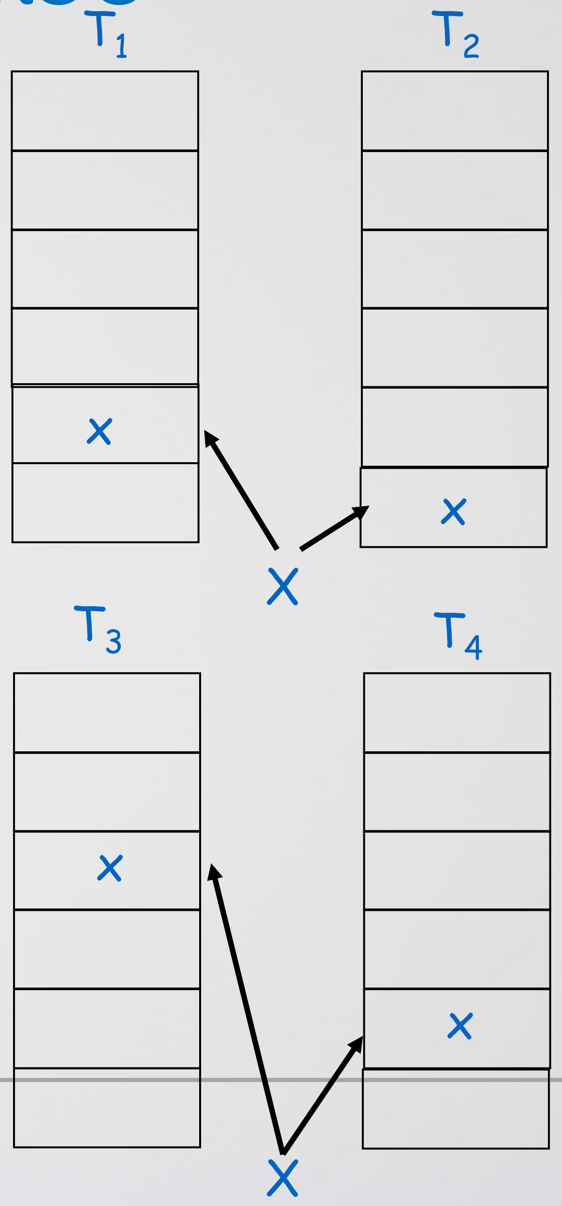
Why does the stash size matter?

- **All** items in the main tables are compared using **O(n)** comparisons (namely, **8n** comparisons)
  - Permutation based hashing [PSSZ16] => compared values are short

- **Each item** in the stash must be compared with **n** items
  - With s items in each stash, and 4 CHs, and two parties, we end up adding **8sn** comparisons.

An Experimental Solution – 2D Cuckoo

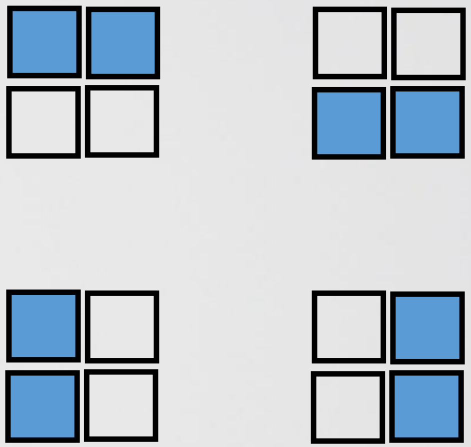
- Alice and Bob each hold 4 tables, and the same 4 hash functions



An Experimental Solution – 2D Cuckoo

- Alice and Bob each hold 4 tables, and the same 4 hash functions
- **Alice:** Places item in (T_1 and T_2) or (T_3 and T_4)
- **Bob:** Places item in (T_1 and T_3) or (T_2 and T_4)

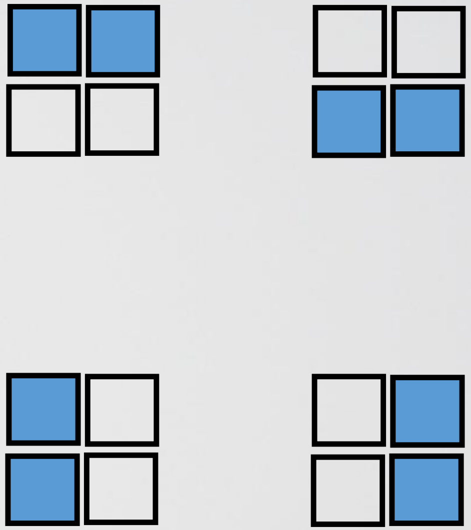
Possible cases



(the actual protocol is a bit different)

An Experimental Solution – 2D Cuckoo

- Like a quorum system
- If both parties have the same item then there is exactly one location in which both store it
- The circuit simply compares the item that Alice places in a bin to the item that Bob places in the same bin

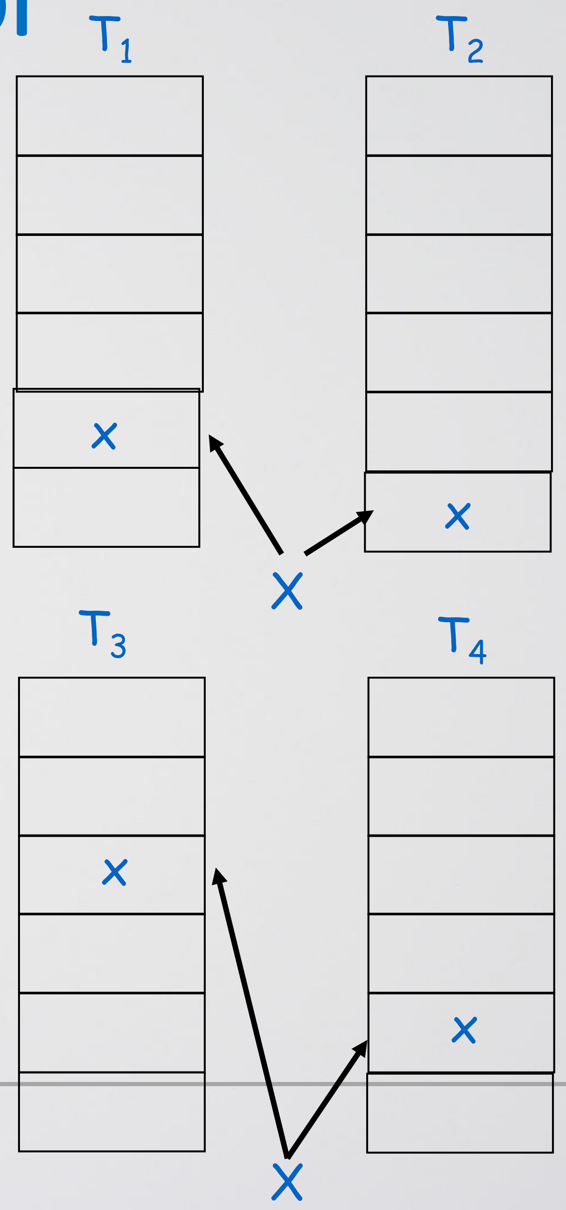


- **Question:** Can this be done? how big should the tables be so that n items could be placed w.h.p ?

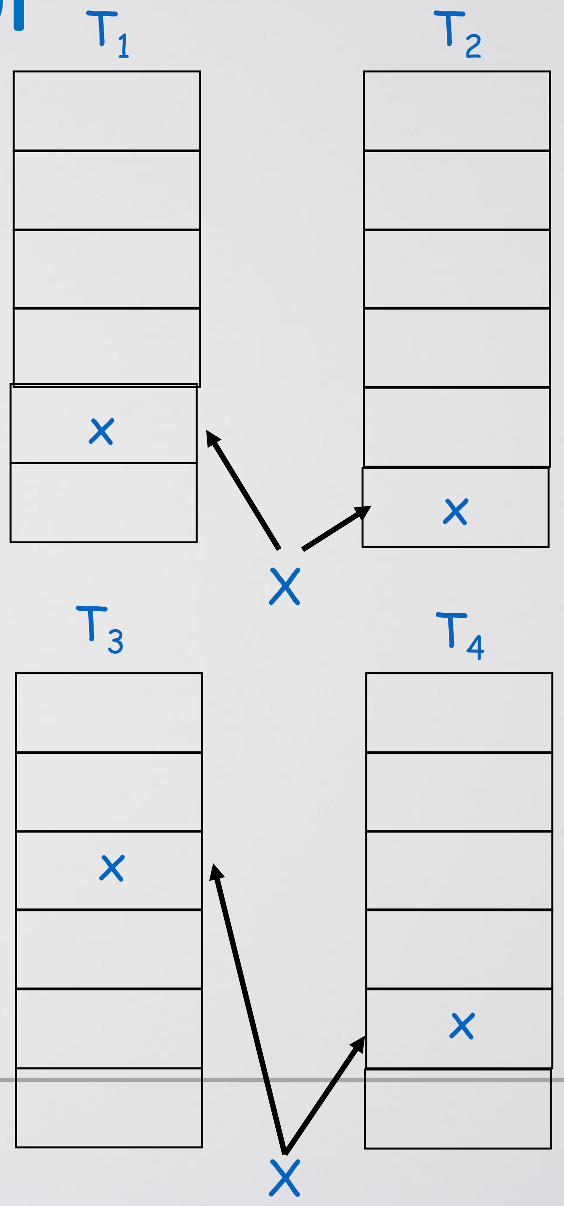
2D cuckoo hashing $\Rightarrow$ O(n) protocol

- **Invariant:** Item in (T_1 and T_2) or (T_3 and T_4)
- **Theorem:** n items could be placed maintaining the invariant w.h.p.
  if each table has > 2n buckets of size 1.

- Total of 8n buckets and 8n comparisons
- The stash adds 2sn comparisons (there are many protocol variants; stash size is the main differentiator)



2D cuckoo hashing ⇒ O(n) protocol



- **Invariant:** Item in (T_1 and T_2) or (T_3 and T_4)
- **Theorem:** n items could be placed maintaining the invariant w.h.p.
  if each table has > 2n buckets of size 1.

- **THM was proved using a new proof technique!**
- The new proof can also prove known theorems about CH, as well as more general constructions
- BUT, we don't have (yet) an analysis for the size of the stash

An even better 2D Cuckoo variant

- Instead of 4 tables of size **(2+ε)n**, where each entry holds **one** item…
- Use 4 tables of size **(1+ε)n**, where each entry can store **two** items

- In simple CH it was shown (first experimentally and then theoretically) that storing two items in a bin reduces the overall size of the tables

- We don't know how to prove this for 2D CH
  - But we can check experimentally

Using Probabilistic Data Structures in Crypto

- E.g., hash tables, dictionaries, etc.
- We want the failure probability to be small (2^{-40}?, negligible in n?)
- Different levels of assurance
  1. There is an exact analysis of the failure probability (e.g., for collisions in a hash table or Bloom filter)
  2. There is an asymptotic analysis of the failure probability (e.g., for simple Cuckoo hashing)
  3. No analysis of the failure probability (e.g., 2D Cuckoo hashing with 2 items in each bin)

Using Probabilistic Data Structures in Crypto

- E.g., hash tables, dictionaries, etc.
- We want the failure probability to be small (2^{-40}?, negligible in n?)
- Different levels of assurance
    1. There is an exact analysis of the failure probability (e.g., for collisions in a hash table or Bloom filter)
    2. There is an asymptotic analysis of the failure probability (e.g., for simple Cuckoo hashing)
    3. No analysis of the failure probability (e.g., 2D Cuckoo hashing with 2 items in each bin)

Best

Using Probabilistic Data Structures in Crypto

- E.g., hash tables, dictionaries, etc.
- We want the failure probability to be small (2^{-40}?, negligible in n?)
- Different levels of assurance
  1. There is an exact analysis of the failure probability (e.g., for collisions in a hash table or Bloom filter)
  2. There is an asymptotic analysis of the failure probability (e.g., for simple Cuckoo hashing)
  3. No analysis of the failure probability (e.g., 2D Cuckoo hashing with 2 items in each bin)

Must use experiments to find exact parameters

Experiments

- How to verify a failure probability of 2^{-40}?
- We ran **2^{40}** experiments of hashing n items to 4 tables, where each table has $1.2 \cdot n$ entries of size 2
  - We used $n = 2^6, 2^8, 2^{10}, 2^{12}$
  - The # of times that a stash was needed (i.e., the failure probability) behaved as $\mathbf{n^{-3}}$. (Agreeing with a sketch of a theoretical analysis)
- Used about 2,230,000 core hours!
  - Possibly the largest hashing experiment per date?
- For $n=2^{12}$ the stash was needed only once (in experiment # $2^{39.15}$)
  - Giving a 99.9% confidence level that $p \le 2^{-37}$ for $n=2^{12}$.
  - Therefore for $2^{13} \le n$ we have 99.9% confidence that $p \le 2^{-40}$

Circuit size

Circuit size (# of AND gates) for sets of $n=2^{20}$ elements of length 32 bit each

Construction	Circuit size (AND gates)	Normalized size
Sorting network [HEKM12]	1,408,238,538 O(nlogn)	2.04
Cuckoo + simple hashing [PSSZ15]	688,258,388 O(nlog/loglogn)	1
2D Cuckoo with separate stashes	313,183,300 O(n)	0.45
2D Cuckoo with a **combined stash**	215,665,732 O(n)	0.31

Evaluation – run time

	LAN $n=2^{16}$	LAN $n=2^{20}$	WAN $n=2^{16}$	WAN $n=2^{20}$
DH/ECC PSI-CA [DGT12]	51,469	819,820	52,178	831,108
[PSSZ15]	15,322		177,245	
2D Cuckoo separate stashes	7,655	90,078	81,995	1,113,169
2D Cuckoo combined stash	6,046	64,258	63,369	761,318

- Run times (in msec) for computing the **size** of the intersection
- ECC PSI-CA is a Diffie-Hellman based protocol for computing size of the intersection

Evaluation

	LAN n=2^{16}		LAN n=2^{20}		WAN n=2^{16}	WAN n=2^{20}
DH/ECC PSI-CA [DGT12]	8.5	51,469	12.8	819,820	52,178	831,108
[PSSZ15]	2.53	15,322			177,245	
2D Cuckoo separate stashes	1.26	7,655	1.4	90,078	81,995	1,113,169
2D Cuckoo combined stash	1	6,046	1	64,258	63,369	761,318

Over a LAN, the new two-dimensional hashing protocols perform best

Contributions of the new protocol

- Asymptotically better: O(n) vs. O(nlogn/loglogn)
- Runs faster
- New analysis techniques for Cuckoo hashing

- Simplifies the usage of PSI

Conclusions

- PSI in an important and interesting primitive
- Research benefits from ideas from other subfields
- Most previous work was on simple two-party PSI
- New results:
  - Generic computation over PSI
  - PSI over outsourced data
  - Multi-party PSI