We are only going to consider semi-honest adversaries, those who are familiar with secure multi-party protocols.

And we have efficient protocol implementations of all the protocols I'll describe here.

So, when we think about this problem, most people when they hear it for the first time come with this naïve solution.

They say, OK, Alice has inputs x\_1, …, x\_n, and Bob has his own inputs.

A possible solution is for both of them to agree on a cryptographic hash function, which is easy to compute and hard to invert.

And then, Bob would send to Alice the hash values of each of his items.

Alice will compute the hash of her items, and compare them and find the intersection.

This seems secure, because this hash function is one way.

So, if you see H(y), you cannot reverse the computation.

So, this seems secure. And it's very efficient.

The problem with this approach is that the items might come from a relatively small domain.

So, consider for instance that the items are IP addresses.

So, there are only 2^32 options for each value.

So, what Alice can do when she receives these values?

She can just compute the hash of each possible IP addresses.

It would take her probably less than a second.

And then, she just compares them to the values that Bob sent. And she'll know Bob’s set.

So, this solution is nice, but it doesn't protect privacy if the items don't come from a universe with high entropy.

And it's interesting because some companies actually use this solution,

because it gives you some kind of a nice fuzzy feeling that this is secure.

So, PSI has many applications.

It can be used for information sharing between different companies that want to see if they were attacked by the same threat,

matching things, matching DNA, or you can think of a dating application when you want to match preferences of two persons.

People talked about it using for all kinds of APPs to identify mutual contacts of phones of different people, and then if identify mutual contacts.

With really new subscriber to the application and old subscriber, I can target that person or give him content that I know that his content like.

And another very appealing application is computing ad conversion rate.

我会详细讲解这一应用场景

And I'll go into it in more detail.

So, think about online advertising like Google or Facebook is doing. And then retailers show ad.

And something which is very useful that these companies are doing is they tell the advertisers how useful the ad was, OK?

So, when you put an ad in the newspaper, you don't know how useful it was.

If you put it on Google, then they can tell you how many people click on the ad.

And they call it the ad conversion rate.

Ideally, it should tell you for each ad, how much money you made out of it, how much ads were converted to actual transactions.

This is very important. This is how these companies are making money. It's worth billions.

So, it's easy for online web shops. But it's very hard with offline purchases.

So, suppose that Alice, she sees an ad for some painting equipment on her machine.

And then the next would she goes to this company and she buys this thing.

So, Google or Facebook would like to somehow tell the advertiser that this purchase was done, because Alice saw this ad, OK?

But they don't know that she went to that store. They don't know how much she bought it, OK?

So, basically, you have to take the database of the real world shop,

and computing the intersection of that database with those who have seen ads for that store.

And based on that, compute how useful the ad was.

So, this seems kind of maybe science fiction.

But this is what these companies are doing all the time.

This is how they make money, because then they can go to the advertisers, and tell them,

look, you made so much money because of this ad, then you have to pay as this and that.

And this is of course much more complicated what I'm saying.

But you have to combine two private databases, OK?

And actually both of these companies, both Facebook and Google are using variants of private set intersection to do exactly this thing,

according to publications in the press and in some presentations that they make.

They are factoring time, they are factoring things like,

the ad is targeted to 30-year-olds who make this amount of money, and live here or there.

So, they showed that to say 95% of that population.

Then they count how many people from the control group, the 5% bought the item, because they're likely to buy the item anyway.

They count how many people from the 95% group bought the item.

And the difference, they can quantify it how much revenue was generated by the ad.

I mean, this is how they make money.

I'm sure that more people are working on that, then working on search quality.

OK, that's it. Yeah.

OK, so PSI is important and there have been lots of work on PSI.

So, there are some solutions, which I will say, OK, they're based on Diffie-Hellman assumption.

Those who are not in crypto probably don't know what I'm talking about.

But you shouldn't worry too much. I'll explain everything that is needed.

So, I knew about this work from 1999 by Huberman, Hogg and Franklin.

But then, I was referred to earlier work.

And apparently Catherine Meadows did something similar in 1986,

and Adi Shamir has something similar in a paper from 1980.

There's another work, which uses public key encryption to do PSI, by Emiliano De Cristofaro and Gene Tsudik.

And they use blinded RSA, or some versions of it.

There are works based on generic secure multi-party computation and circuits.

This is the focus of our talk. And I'll talk more about it.

There are works based on bloom filters.

And there are also works based on oblivious transfer and hashing. I'll describe it in short.

Those who don't know this was Diffie-Hellman, and I would say MPC, bloom filters, oblivious transfer,

don't worry, I'll tell you everything you need to know.

And the main issue is that comparing two sets requires in general n^2 comparisons. And doing n^2 is too complicated.

And these items come from a big universe, so you cannot do any tricks.

And we want to reduce the number of comparisons.

OK, so the recent the most efficient constructions are based…

This is the work of me with Thomas Schneider and his student Michael Zohner and continued afterwards.

Basically we realize that computing PSI in some sense can be very efficient when using oblivious transfer, which I won't describe.

And oblivious transfer can be computed with oblivious transfer extension, which is very fast.

And therefore, we just have to push everything to do oblivious transfer,

and use hashing to reduce how many we have to compare with how many,

and then we do it very efficiently.

So, how efficiently?

This is a slide that shows how the different approaches work.

横坐标是运行时间 纵坐标是通信量

So, this is run-time. This is communication.

这里使用了对数坐标轴 单位长度表示时间或者通信量扩大或者缩小10倍

These are logarithmic scales. So, from here to here, you go by a factor of 10.

我们希望方案能达到左下角的位置 运行时间最小 通信量最小

And you want to be here, minimal run-time, minimal communication.

These are the public-key based approaches, the RSA and Diffie-Hellman, OK?

They have relatively very good communication. The runtime is pretty high.

And this is Diffie-Hellman with elliptic curves, which uses a smaller model size.

So, the communication is better.

These are the circuit-based approaches, which are our focus. I will talk more about them.

They have high communication and high runtime.

This is the most recent oblivious transfer based approaches.

They have very good runtime and good communication.

基于OT的方案之比朴素哈希方案慢6倍 而朴素哈希方案是不安全的

And actually, they are only 6 times slower than using this hash function, the naive solution, which is insecure.

基于OT的方案效率极高

So, it's pretty good.

Our goal is to take this generic circuit-based constructions, and try to move them here.

We'll just move them by a little bit. But this is the goal.

And I'll tell you why that's important.

OK, so PSI is a specific example of secure two-party computation.

Secure two-party computation is the problem where you have two parties with private inputs,

and they want to compute some function of the inputs without revealing anything else about the inputs.

So, in our case, they want to compute the intersection.

They could do other things, say compute some statistical test of the inputs, or whatever, OK?

And MPC is the generic name for doing these secure multi-party computations.

So, luckily, there are generic protocols for securely computing any function, OK?

In particular, they can be used to compute set intersection, OK?

So, the question is why am I focusing on set intersection where there are very good generic solutions that can be applied anywhere.

And the issue is that these generic solutions, they require you to have the function represented as a binary circuit

using end AND and NOT gates that you probably learned in your first year at university, OK?

So, if you want to compute a function, you have to describe it as a binary circuit.

And then, there are generic protocols and software packages that you can use, OK?

So, basically, you have to take this set intersection problem and represent it as a circuit.

Now, there are good reasons to use a generic solution.

One of them is adaptability.

So, consider that you designed a very good solution for computing the set intersection.

And then, your boss or customer or colleague comes and say,

我对计算两个集合的交集不感兴趣

OK, I'm not interested in computing the intersection,

我想知道交集中包含多少个元素 但我不想知道交集本身

I want to know how many items are in the intersection, but not the intersection itself.

So then, you have to work hard or hire a crypto expert to do it for you.

And then, they come and say, we don't want to compute this size in the intersection,

but rather a bit, which is 1 if the intersection is greater than 100 and 0 otherwise.

So again, you have to hire a crypto expert to do it for you.

Now, suppose you had a circuit which computes the intersection, OK?

Then, if someone comes to you and says, OK, I want to compute the size of the intersection,

so, basically you should get a second-year undergrad who learned about Boolean circuits,

and asked him to design a circuit, which counts how many items are in the intersection.

And then, if you want to output a bit depending on whether the number is greater than 100 or not,

then he should compare the result to 100 and then output the bit.

So, basically instead of having to design a new protocol for each variant of the problem,

you basically have a programming language, or the circuits, with which you can compute any version of the problem.

Another reason for using this generic approaches with circuits is that

we have very good implementations that do this secure two-party computation. And they run extremely fast.

So, once you have a circuit, we can use them.

And also the existing examples, for instance, computing the ad revenues,

they don't compute the intersection itself, but rather it has revenue for each transaction.

They find which transactions are at the intersection, then they sum them up, OK?

So, this is something that can be naturally done with the circuit.

And it's harder to design a specific protocol for that.

By the way, you can ask questions of course throughout the talk.

So, we'd like to design a circuit based protocol.

And there are generic protocols for computing any function with a circuit.

There's the GMW protocol, the Yao’s protocol. I won't go into details, OK?

But a lot of work has been done on implementing them very efficiently.

And we are sure that the parties don't learn anything except with the desired output.

And the overhead only depends on the size of the circuit.

The problem is that the naive circuit for doing PSI has to compare each item of the first set with each item of the second set.

因此 总共要执行n^2次比较操作 涉及的比较次数实在是太多了

So, it's going to be doing n^2 comparisons. And this might be too much, OK?

如果n为100万 总比较次数是不可接受的 而实际中n的确可以达到100万

If n is a million, this is too much. And n is a million, OK?

能不能进行优化呢？

So, can we do better?

如果只计算两个s比特长的特定值是否相等 这一过程是非常高效的

So, if we look just comparing two specific values, the value is set up s bits, then comparing two values is very efficient.

Basically, you have to XOR each bit, the i-th bit of x with the i-th bit of y.

The result is 0 if they are the same.

And then you should compute the NOR of this.

And with MPC, actually XOR you're doing for free.

So, I have to do s-1 NOR gates for doing that.

So, comparing two 32-bit numbers can be done using essentially 31 gates.

And we can process millions of gates per second, even more than that. So, that's great.

So, comparing two items is efficient.

The goal is to arrange the two sets of items so that we have to do as few comparisons as possible.

这并不是一个新问题 第一个解决方案是使用排序网络

So, this is actually not new. And the first solutions were based on sorting networks.

So, sorting network is something old from the 60s, I guess, if not earlier.

And they were used to actually sort phone conversations, where you had copper wires and actual sorting machinery, OK?

So, a sorting network is a network of wires and small comparator modules,

which get two inputs, and compares them, and then outputs them in the sorted order, OK?

And there was a lot of work I guess in the 60s on making them as efficient as possible,

because this is how phone networks were built at the time.

So, we can do PSI based on these protocols.

So, sorting networks were designed for actual networks.

But we can design a binary circuit based on them, and using this circuit compute PSI.

Everything will be done virtually in the computer, but this will be based on that work.

So, suppose you have two lists of Alice and Bob, each list contains values.

And we know that all the values in a single list are different.

So, there are not two identical values in the same list.

The first thing they would like to do is to merge the two lists to have one long sorted list of the union, OK?

So, this was done in 1968 by Batcher.

And this is a network which gets two lists of size n.

Each thing here is a comparison which take this value and the other value, compares them and outputs them in sorted order.

And using exactly nlog(2n) comparisons, you can get the sorted list, basically merge two sorted lists.

So, the whole circuit works like that.

Each party sorts it lists.

They insert the list to a bitonic merging network, the one we have seen before, which output a sorted union of the two lists.

Now, if Alice and Bob had the same item, then in the sorted union, the two copies of that item will be adjacent to each other.

So now, all we have to do is go through the list, compare each two adjacent items, and see if they're equal or not.

This can be done using just 2n equality checks.

Then at the end, they want to shuffle the results,

because we don't want to show that there was a match saying the first two items that compared,

because where the matches are occur reveal something to Alice about Bob's list and vice versa.

So, we have to shuffle the result. This is also with nlog(n).

So, everything can be done with 3nlog(n)+4n comparisons.

And each comparison takes s gates.

So, this is pretty good. This was done by Huang, Evans and Katz.

Another version of a circuit was done by me, with Thomas Schneider, Gil Segev, and Michael Zohner.

And it works in a different way.

So, OK, one party maps its items to about 2n bins using Cuckoo hashing.

And I'll describe Cuckoo hashing a bit later.

So, and it ensures that each bin… So, we have about 2n bins, and each bin will contain a single item.

If you don't know cuckoo hashing, you'll understand shortly why it works.

The other party maps its items to these bins using simple hashing.

You'll have this number (O(log(n)/loglog(n)) of items in each bin.

Then basically, you have to compare each bin of this party to each bin of that party.

So, you have to compare n times. We have n bins, one item to O(log(n)/loglog(n) items.

So, the total number of comparisons is O(n·log(n)/loglog(n)).

It's better than the nlog(n) that was before. But we want to reduce it to just O(n).

So, the challenge here is to find the smallest circuit for computing PSI.

Both parties can prepare their inputs. But the circuit must not depend on the data.

It should be the same circuit no matter what the data is.

And then, once we have the circuit, we can compute any function we want of the intersection afterwards.

So, for instance, count how many items are in the intersection or whether the number of items is greater than some threshold.

Or if we care about privacy, we can add some noise to the count to ensure differential privacy.

Or we can have some values associated with transactions or whatever.

And the goal is to minimize the number of comparisons in the circuits, and also the length of the items.

So, what I'm going to show here is a circuit with a linear number of comparisons.

There's one construction which has a provable linear overhead,

and another construction which has a linear overhead which was experimentally verified. And we don't know yet how to prove it.

But the constants here are much smaller.

We run implementation and experiments, the run-time is better than that of the O(nlog(n)/loglog(n)) construction.

I write here surprisingly, because Udi Wieder who's a co-author, he works a lot on algorithms and said,

it's usually when you have an algorithm like that,

and there are improvements which improve the algorithm to run in linear time, it's better sympathetically.

But practically this algorithm runs better.

Even if n is a million, log(n) is 20, loglog(n) is 5, this is 4, OK?

So, but surprisingly, we get an algorithm which runs faster, OK?

And on the world, we also have a new analysis for Cuckoo hashing,

both for the old version of Cuckoo hash and for the new variant of it, OK.

So, I'm talking a lot about hashing. Let's talk about it in more detail.

So, the basic observation which I hinted already is that,

if both parties agree on a hash function, which is independent of the inputs, OK,

then the parties can each map their inputs to bins, say if they have n inputs, they can map them to n bins, OK?

And then the nice property is that if both Alice and Bob have the same input x, they are both going to map it to the same bin.

So, they map x to the bin h(x), the the range of this hash function is 1 to n the bins, OK?

So, if they have the same input, it is going to be mapped to the same bin by both of them.

So now, instead of comparing all of Alice items to all of Bob items,

we just have to compare the items that Alice put in that bin to the items that Bob put in that bin, OK, much smaller.

So, if you have n items which are mapped to n bins, how many items do we expect to find each bin?

A constant number of items.

从期望上看 如果这个桶里面有一些元素 另一个桶里面有一些元素

So, in expectation, we have one item here, one item here,

对比两个桶是否包含相同的元素 仍然需要平方级比较次数

comparing them is square the number of items.

但此比较次数可以归约为常数

But it's also going to be constant.

So, overall we're going to have a linear overhead, OK, n times constant, this looks great.

The only problem is or the major problem is that if we throw n items to n beans,

some beans are going to have more items than others.

And the parties need to hide from each other how many items they place in each bin, because the hash function is public,

because if, for instance, Bob knows that Alice didn't place any item in the first bin,

and he knows that a certain value, the hash of that value would have been placed in the first bin,

then he knows that Alice doesn't have that value.

And this might leak information.

So, we need to hide how many items that will map to each bin, OK?

And the only way I know to solve this is to pad each bin with dummy items

so that all bins will have the same number of items of them.

So, some bins are quite empty. Others are full.

But we pad each one of them to be as full as the largest bin.

And when we map n items to n bins, the expected size of a bin is constant.

But with high probability, the most occupied bin will have O(log(n)/loglog(n)) items.

So, we have to pad each bin to be of that size.

So, then the circuit will be O(n·log(n)/loglog(n)).

So, this could be log^2(n) over something, or we can do sorting network for that.

But that's not linear overhead, OK? And we want to do better.

So, here is where Cuckoo hashing comes to help us.

Cuckoo hashing is a very useful tool which was designed by Rasmus Pagh and Rodler in 2001.

And the version I talk about here is by Kirsch, Mitzenmacher, and Wieder.

The idea is the following.

Instead of just throwing n items to n bins to a table of size n, in which case a bin might have many items,

I have two tables, each one of them is of size n, and also a stash, which would be very small, like 2 or 3 items.

And I have the hash function associated with each table, h\_1 is for T\_1, and h\_2 is for T\_2.

And a value x could be either in location h\_1(x) in T\_1, or h\_2(x) in T\_2.

Or it could be placed in this small bin of size 2 or 3, OK?

And then, when you look in for an item, you know that is either here or here or here.

So, you can look for an item with a constant number of lookups.

So, that's very useful. This is essentially a dictionary. Each item can be found with the constant number of lookups.

And there's a very efficient algorithm for inserting items to Cuckoo hashing. I won't describe it here.

And the nice property is that the size of the table is slightly more than linear.

Then it's possible to store n items in this table except with probability which is n^-(s+1).

So, if the stash is of size 2, except with cubic small probability, you'll be able to store n items here.

So, this is extremely useful.

You see that we lose a little bit.

We have to store n items, and this is of size n, and this is of size n, so the size is 2n. So, we only use that 50% of the memory.

But it's an extremely useful tool.

By the way, who has heard of Cuckoo hashing? Please raise your hand.

OK, it's extremely useful. I think they should be teaching it in undergrad algorithms course.

It's really good just as a tool.

OK, now, there is some probability with which we have a set of n items.

And we will be able to place them here, here, or here, OK?

And the probability is n^-(s+1).

So, what happens if we use Cuckoo hashing for any security related application?

Then it might happen that we won't be able to map the items to the tables.

The hash functions just happen to not work well with the inputs.

The probability is this small, but this might cause either an accuracy problem or privacy breach.

If we cannot put an item in the tables, then the result wouldn't be accurate.

And if I am not able to map my items to the tables,

and I'm telling the other party, look we should choose other functions,

this leaks some information about my inputs to the other party, OK?

So, when this happens, that's bad, that's some failure of the protocol.

And you would like to limit this probability.

So, in cryptographic applications, usually we set some statistical security parameter.

And we want the failure probability to be smaller than that.

Usually we want to make it smaller than 2^-40, OK?

This is the probability of just things not working well.

Since we know that this is n^-(s+1), we know that

for any stash size s from some value of n, OK, the probability will be smaller than 2^-40.

So, basically if n is greater than some threshold N, then if the stash is bigger than that, then we should be fine.

So, this is good. But there is one issue.

The guys who did this algorithmic analysis, they only cared about asymptotic.

So the theorem we have just claims that the failure probability is O(n^-(s+1)),

whereas in cryptography, we really want to get concrete numbers.

So, that's not good enough.

It's good asymptotically, but we want to know about the actual numbers that are using.

And this is actually quite a challenge here.

And if you look at the proof, it's quite impossible to get from the proof what are the actual numbers,

because they depend on other proofs, which were asymptotic. And it's impossible.

It's possible, but it's very complicated.

OK, so can Cuckoo hashing help us?

So, OK, the problem was that we had to pad the bins to have about log(n) items, OK?

So, if Alice and Bob use Cuckoo hashing, that's great,

because Alice maps her items to these two tables, so it only will have one item in each bin,

and Bob will have one item in each bin,

so we can compare two bins with just one comparison.

The problem is that Alice might decide to put x in table T\_2, and Bob might decide to put x in table T\_1.

So, therefore, they won't be able to compare it, OK?

So, Cuckoo hashing is so efficient because each item can be placed in two locations.

But this is also why we cannot use it as it is.

And previously it was solved in the following way.

Bob uses Cuckoo hashing.

So, its items are placed either here or here, or in the stash.

So, Bob puts only a single item in each bin.

Alice uses simple hashing.

So, basically she maps all her items to T\_1 using the first hash function,

and all items to T\_2 using the second hash function, OK?

She's going to get a O(log(n)/loglog(n)) items in each bin.

Then we know that if Bob happens to put x here, Alice puts x both here and here.

So, regardless of where Bob decides to put it, he'll be able to compare it with Alice's data structure, OK?

So, how many comparisons we have?

We have O(n) bins. For each one, we have to compare one item with O(log(n)/loglog(n)).

So, the total number of comparisons is this.

Also, for each item that Bob placed in the stash, which is of constant size, we have to compare it to each item of Alice.

So, this is the total number of comparisons.

And we can use something called permutation based hashing to store only short values.

So, from this, you can get a circuit. How would the circuit work, OK?

Bob uses cuckoo hashing to map his items to bins, OK? Alice uses simple hashing.

For each bin, we have O(log(n)/loglog(n)) items of Alice, one item of Bob.

The circuit should take each of these O(log(n)/loglog(n)) items, and compare it with the item of Bob.

The total number of comparisons is O(nlog(n)/loglog(n)) plus O(n) comparisons for each of the item in this stash.

So, this is a circuit construction, OK?

The size is O(nlog(n)/loglog(n)). That's what's the best that was known before, OK.

So, this is what we've done.

So, I'll start with the asymptotic construction, and we use something which we called mirror hashing.

So, we have 4 tables, each one of size a bit more than n.

And we actually have two sets of 4 tables, 4 here and 4 here.

And we have only 4 hash function.

A color here is the hash function. So, we have the blue, red, purple, and yellow hash functions.

And you see that there we use between the tables, OK?

So, we use the same set of hash functions, both in the left set and the right set, but in different order.

OK, so what Bob is doing?

He uses Cuckoo hashing to put each item of his in one of the two tables in each column.

So, if you look at one color, you have here two tables with two hash functions.

You can use Cuckoo hashing to place each item either here or here.

And let's forget about this stash, OK?

Similar here, he puts all of his set either in this table or in that table.

Similar here, he puts each item either here or here, and so on.

So, he ends up putting each of his item in one location in each column.

This is what Bob is doing using simple Cuckoo hashing.

Now, Alice starts with a set of the left 4 hash function.

And she uses Cuckoo hashing to put each of her items either here or here, and then either here or here.

Just like Bob, but just on one set.

They look at the specific item of Alice.

It might happen that the Cuckoo hashing algorithm chose to insert this item to the left table on the first pair of tables, and to the left one on the second pair of tables.

So, basically it chose to put both of them on the same column.

Because the algorithm puts either here or here, so it might happen for some items that they were both put in the same column.

That's a good property.

And these items will keep here.

Other items, the algorithm might have chosen to put in the left column here and in the right column here, OK?

But then, Alice will store these items that should not put in the same column here.

She stored them in the right set of tables using the same hash functions.

But in the right set of tables, we use the same hash functions,

but we kind of reverse or mirror the order of the lower two hash functions.

So, if two items were mapped in the left set to two tables which are not in the same column,

they'll be mapping the right set to two items which are in the same column.

So, Alice, basically, the items which were lucky enough to be put in the same color at the first set will be kept there.

Those that were mapped to different columns, she put them in the right set,

but then the hash functions would put them in the same column.

So, at the end, Alice gets that all of her items are mapped to the same column, OK, either in the left set or in the right set.

OK, so what we got that both Bob, if you remember, put each of his item to one table in each column,

and Alice put each of her items in the two tables in exactly one column.

Overall, we see that there's intersection of exactly one table,

for each item in the intersection, there's exactly one table where both of them put it.

In this case, it's this yellow one.

So, therefore, what we do here is each one of them is doing this hashing at home without talking with the other party.

Then, they have a simple circuit which just takes the first item here, compare it with the first item here, the second item with the second item here.

And since in this table, both parties placed x in this yellow table, and they placed it to H(x) in that table which is the same here and here, OK,

there'll be one location in the circuit where both of them place x.

So, therefore the comparison will see the two occurrences of x, OK?

So, great, how big is this circuit going to be?

So, it's 8 times n times… Cuckoo hashing always requires the circuit to be a bit larger than n. So, it's usually 1.2 times n.

So, it's a constant size circuit for doing that.

And actually, we're doing Cuckoo hashing here.

So, some items might not be placed. It will be placed in the stash.

So, each item which is placed in the stash will have to compare to all items of the other party.

But since we have a constant number of stashes, and each stash has a constant number of items,

we have to do a constant number of comparisons with n items.

So, it seems OK.

So, this is where constants start to play.

So, we have a constant size of stash per table.

If we look at it, all items in the main tables, OK, we have 8 tables, each one of them could requires n comparison.

So, we take care of all the tables with just 8n comparisons.

That's great.

Now, each item in the stash must be compared to n items.

So, each party uses 4 Cuckoo hashing. So, it has 4 stashes.

Each stash might be of size 2 or 3 or whatever, so or say of size s.

So, we have 4sn comparisons for the items in the stash for each party.

So, for these items in the stash, we pay more than for the main tables, OK?

We can improve that a little bit. But the stash here are actually quite annoying, OK

They are very annoying, because for these a few items that are mapped to the stash, you pay more than for the rest of the items.

But this is a asymptotically linear construction, OK?

So, that's good. It wasn't known before.

OK, we have a better construction, which we have an experimental result.

And it goes like this.

The parties, they consider 4 tables. Each table has a hash function associated with it.

And they agree on the tables and the hash functions.

OK, what we would like is to Alice to map each of her items, add it to the…

So, we only have 4 tables, not 8, OK?

Each of Alice's items should be mapped either to the 2 top tables or to the 2 lower tables.

Each of Bob items should be placed either in the two left tables or in the two right tables, OK?

So, basically it's like Cuckoo hashing, but it's different.

Instead of putting it in one location or the other,

you have to put it in these two locations or in these two locations, in these two tables or in these two tables, OK?

It wasn't clear before we started if it works or not, OK?

The actual protocol is a bit different, but let's talk about this one, OK?

此协议像是一个分布式系统

So, it's like a quorum system.

And if they manage to do it, then if Alice owns x, she either puts x in both these tables or in both these tables.

And Bob will put x either here or here.

So, there's exactly one table in which both of them intersect.

So, therefore if we build the circuit, which basically takes…

This is where Alice put items, this is where Bob puts his item.

The circuit takes each item in this table, compares it which item in this table.

If both of them has x, there's going to be exactly one location where both of them have put x.

And therefore we see the intersection, OK?

So, the circuit is very very simple.

And the question is can this be done, OK?

And if we make the tables larger and larger, before each table was of size n,

if we start making them larger and larger, this would be possible, but how big should we make these tables?

So, the new result is that if we make each table of size 2n, OK, then this is possible with high probability, OK?

So, going back, we have 4 tables, before we had 8 tables of size n.

Now we have 4 tables of size 2n, each of them, OK?

So, the overall size is about the same.

And the new proof shows that this works with high probability。

And it will have the total of 8n bins or buckets, because each table has 2n bins.

And we can do it with a 8n comparisons.

And there are a few variants.

So, this theorem was proved using a new technique.

Probably we won't have time to describe it.

But it is a new proof technique that can also be applied to prove the original versions of Cuckoo hashing,

as well as more general constructions which are generalizations of Cuckoo hashing.

我能问个问题吗？我有一个问题想问

Can I ask a question? I just have a question.

请问

Yeah.

我们有一个哈希函数

So, you have a hash function.

是的

Yeah.

That maps every entity, every x into a location in this table.

So, for each table, we have a different hash function, which takes a value and maps it to a location.

So, if I'm looking for a specific value x, I know there's only one location the table in which x can appear.

No, so if I have x, I'm going to put it in this entry.

The actual x is 32 bits long or whatever.

And then, the circuit will take the x that I put here, and the x that you put here.

You put here in your copy of the table, and compare both of them.

OK, so actually we can do a little better.

So, OK, here we have a table of size that the theorem has proved each table of size 2n, OK?

And each entry can hold 1 item.

We can actually do better.

Instead of looking up table of size 2n, we look a table of size n.

But each bin in the table will be able to store 2 items.

So, the total size will be the same.

Instead of a table with 2n entries of size 1, we have n entries of size 2, OK?

So, the total size will still be 8n.

It's known in regular Cuckoo hashing that

if you do this instead of having a certain number of entries of size 1, have half the number of entries of size 2,

you'll get better better memory utilization in it.

And in Cuckoo hashing, it is known both theoretically and experimentally,

here we suspected that the same thing happens.

We don't know to prove that this is indeed the case,

that if we make the tables half as big, but make each bin being able to occupy 2 items, then we have better hashing.

Better hashing means that we'll have to use stash less.

And we really don't want to use this, because each item in the stash we have to pay for it, comparing it with other items.

This was the conjecture. We don't know how to prove it. But then, we run experiments to verify it.

So, the experiments we verified were quite extensive.

So, we run 2^40 experiments of hashing n items to 4 tables, where each table had about n entries of size 2.

Now, these 2^40 is a lot, OK? It's like a million times million times hashing n items to a table.

We end up spending more than 2 million core hours, OK?

That's even a lot of money.

If you might have to buy that Amazon, that's a lot, if you don't have a local cluster to work on, OK?

What we did? We checked it for 2^8, 2^10, 2^12, although we are interested in larger n.

And we saw that the probability of having to use the stash decreases cubically with n.

So, it's like n^(-3), quite accurately.

And for n=2^12, we only had to use this stash actually in experiment 2^39.15.

We were almost true that we won't have to use the stash. And then towards the end, we had to use it.

But what we can claim is that, OK, the probability of having to use the stash decreases as n^(-3).

So, therefore, for n which is greater than 2^12, the failure probability is smaller than 2^(-40).

We don't know how to prove this theoretically, even if there will be a proof, OK?

The proof won't have the constants in it.

We'll be able to show that this holds for large enough n. But we won't be able to show the proof.

But I think these experiments are quite convincing.

Now, there's another related work, which we heard about recently.

This is also a work of Rasmus Pagh and his student.

They were trying to do intersection with GPUs.

And they used a similar model, OK?

They had 3 tables, and they used a variant of Cuckoo hashing, where each item had to be put in 2 out of the 3 tables, OK?

So, we had 4 tables. You have to put it either in this 2 or that 2. They had to put it in 2 out of the 3.

And then, if Alice puts in 2 out of 3, Bob puts in 2 out of 3, then they are guaranteed to have an intersection.

They didn't care about privacy. They cared about doing it on the GPU with many processors.

The problem is they might be intersected in 2 tables, not just in 1.

But with another bit for signaling, you can make sure that you only count this once, not more than that.

The total number of buckets is only 6n, not 8n like us.

They didn't have an analysis.

There's a new work of Eppstein, Goodrich, Mitzemacher and Torres, which analyzes the stashes of this size.

But experimentally, it seemed to behave worse than ours for the size of the stash, which really matters.

And they didn't care about constants here, OK?

So, asymptotically 6n, not 8n, experimentally the stash seems to cause problems here.

OK, so we needed to set the parameters.

We run experiments for what the size of the stash should be.

We use something called permutation based hashing. So, we can store less items in each bin.

And we used 4 tables of size n, where each entry can store 2 items.

And somehow in the version that I describe here, Bob needs to use a combined stash.

Alice doesn't, Bob needs to use one, I won’t say why.

OK, so here are the results.

So, this is a circuit for comparing a million items of size 32 bits long.

This is the Sorting network approach I describe.

This is the one where one party is doing Cuckoo, the other one is doing simple hashing.

And this is our best version.

It's better to look at the last column.

The last column has normalized numbers.

第二行PSI协议是在我们方案提出之前的最优方案 我这里设置成了1

Basically, this was the best that was known before our construction. So, I put it the value 1.

It was twice as small as the Sorting network. The circuit size was twice as small.

And we are able to improve this by a factor of 3 or more.

So, our circuit is much smaller.

If we look at the evaluation time, I'll highlight different properties of this table, but let's look what's there.

第一行是计算交集大小的专用协议

This is a specific protocol for computing the size of the intersection.

下面三行是应用电路计算交集大小的通用协议

These are circuits that compute the size of the intersection.

第二行是之前已知的最优协议

This was the best one that was known before our work.

第三和第四行是我们的方案

And these are our constructions.

左侧两列是LAN下64,000个元素和1,000,000个元素求交集

This is on the LAN for 64,000, 1,000,000 items,

右侧两列是WAN下64,000个元素和1,000,000个元素求交集

on WAN network 64,000 items and 1,000,000 items.

来看看实验结果

So, let's see.

第一行协议基于椭圆曲线密码学

So, this was based on Elliptic Curve Crypto.

下面三行协议基于安全多方计算

These were based on secure computation.

LAN下通信效率为1Gbit/秒

This is 1Gbit/sec.

WAN下通信效率为100Mbit/秒 单轮时延为100毫秒

The other is 100Mbit/sec, and the round trip cost is 100ms.

So, what do we get here?

One thing we see for the approach which doesn't use a circuit,

this is just using a specific protocol. It can only compute the size of the intersection, not other variants.

These are more versatile.

This one uses elliptic curve crypto. It is very good in terms of communication, but it is doing a lot of computation.

So, you see that the run time is independent of the network.

If you do it for 2^16 items on a LAN or WAN, is about the same time, same for larger sets.

So, communication here is very small. It doesn't matter. Just computation here invests.

Here you see that when the network slows down by a factor of 10, as you can see from here to here,

the run time from here to here decreases by a factor of slightly more than 10.

So, here communication is the bottleneck.

This is the best protocol we have.

Now, if we look at the LAN, our protocol performs much better.

It's like 2.5 faster than the previous circuit-based protocol, and here 8.5 faster than the specific protocol.

The previous circuit-based protocol couldn't run on a LAN on this size of inputs.

The implementation wasn't good enough for that.

If you look on a WAN, then actually the size of the circuit matters less.

应用椭圆曲线密码学计算交集的协议和我们的协议性能几乎是一样的

I mean, the specific protocols that uses elliptic curve crypto of computing the intersection, and our protocol, perform about the same.

左侧他们的协议更优 右侧我们的协议更优

Here they're better. Here we're better.

但他们的协议只能用于计算交集大小

But they can only be used to compute the size.

我们可以用我们的电路计算任意所需的函数

We can use our circuit to compute any function that you want.

So, the contribution of the new protocol is asymptotically better.

We can do a circuit with only a linear size and linear overhead, compared to O(nlog(n)/loglog(n)) before. It runs faster.

And we run new analysis techniques for Cuckoo hashing that might be used elsewhere.

And it simplifies the use of PSI.

And I'll conclude with further research directions, OK?

Annoying thing is this stash.

(52:58)

You have a few items nap you cannot ignore them because if you ignore them then you leak some privacy I mean you leak some information and still have to pay a lot for them so you would like to mean ideally not use it at all and at least you know minimize the effect of the stache another interesting is to do protocols for the party case where you have more than two parties okay and you want to construct a circuit where you know if you have three or four or five parties and somehow they all map the same item to the same bin and we started walking on there on that and also local alternative approaches without multi-table cooking for doing a circuit based psi and with this I conclude I can also talk about the proof for the cuckoo hashing new cuckoo hashing but it's it's it's quite technical oh thanks okay okay so the question which has actually a slide which I removed from the photo talk was that a if we reveal in which being a match was done ok this leaks some information about the other parties input so one way to solve that like what black was done for the assaulting Network is to get the results and then shuffle them now this final shuffling costs n log n so with with sorting netbooks they paid n log n for merging Elina oh and for comparing and n log n for shuffling we do the first two step with a linear overhead and if we have to do shuffling at the end we have to pay a log n now but shuffling is important if you want to reveal in this section itself and to compute in this section itself you you have better protocols in using a circuit a circuit a circuit is useful for computing a function of the intersection and like the size of the intersection or the size with some noise or you are revenues or whatever and basically if what you compute is a symmetric function of the intersection you find so what's a symmetric function a symmetric function is one that has the same output regardless of the order of the inputs so if you compute the number of items in the intersection it's basically a symmetric function you don't care where the intersection in which means the intersections happens you just add if you add revenues or whatever so basically if you are computing a symmetric function of the intersection in all applications I can think of symmetric functions of the intersection then the circuit that computes the final output after you had the intersection doesn't have to shaffer the results because the final output the symmetric function output doesn't reveal information about the in the order of the inputs okay so in these cases you don't need to do this final shuffling so we don't have to add the shopping which we have so I'll say the question cuz I'm not sure what you're saying is recorded or not so the question was as far as I could tell is suppose we're doing cuckoo hashing and we don't have to use this - if I learned that Alice put she could put X in one of two locations if I learned that she chose to put X in this location would that leak information and the answer is that it does make information I mean the the worst case is where I know a priori that Alice has one of two sets of inputs that she has this set or that set and both sides have the same intersection with my set but I want to know which one of them she has okay like you know I only care about you know specific person if he's in her set and it's he's not even in this section okay so in that case it might happen that having this person in the set or not will affect how other items are placed in bins it's very like it's actually it's know if there is a collision between that that item and other items will we see that okay and actually it's not known as far as far as I know but I'm pretty sure how to do cuckoo hashing where the location the choice of the between the two location where each item can be placed is random and doesn't depend on the other items so this would be actually very nice but we don't know how to do it so we have to hide the order where well where we wear them in which means matching yeah I think we have it in the in the paper we have it's it's it's the size of the day of the size of the circuit times you know the cost per gate and we use the gym W protocol but we we we have it for sure in the paper so it would be easy to find the communication overhead okay thank you promote you can be NPC yeah okay so the question was whether I said that psi is equivalent to ot its equivalent in a very you know fundamental but weak say meaning meaning that if you can compute set intersection we can do oblivious transfer okay how can we do that so okay you have two values I need to learn one of them you have to value okay I think you can get it so if your two values are bits we can do bit oblivious transfer using psi and then from bit oblivious to answer we can do string oblivious transfer from that we can do all crypto so we cannot hope to do psi more efficiently than OT but I don't think that psi is the most efficient way of implementing okay okay okay thanks [Applause]