大家好

Hello, everyone.

我很高兴能在这里为大家介绍我们的工作：ObliVM

I'm very happy to be here to present our work about ObliVM.

这是一个安全计算的编程框架

That is a programming framework for secure computation.

这是我和我的队友们一同完成的工作

And this is a joint work with our teammates,

队友有Xiao Wang 坐在那里的Kartik 还有Yan Huang 还有Elaine Shi

Xiao Wang, Kartik sitting there, and Yan Huang, and Elaine Shi.

很高兴自己在前面的一个演讲中为大家充分介绍了什么是安全计算 或安全函数求值

And so, I'm very happy that my previous talk have intensively explained what is secure computation, aka, secure function evaluation.

我这里只需给出一个简短的介绍

I just give you a brief motivation that,

例如 Sheldon和Amy希望互相确认自己是不是对方的另一半

for example, Sheldon and Amy here wants to find a good partner for each other,

他们都相信基因匹配的效果

and they all believe in genomic dating.

因此 他们可以执行一个分析过程 看看他们是否匹配

And so, they can run an analysis to see whether they match or not.

一个很关键的安全问题是 他们不想把敏感的基因数据泄露给对方

So, the key security question is they don't want to leak their sensitive genomic data to each other.

我在前面已经讲过了 安全计算是这一问题的绝佳解决方案

So, and we have been talking that secure computation is a good solution to solve this problem.

我们可以把这个问题抽象成下述形式

For example, we can abstract this problem as following,

给定两个参与方Alice和Bob 以及他们的秘密输入x和y

given two parties Alice and Bob, having their secret input x and y,

他们希望联合计算一个公开函数f(x,y) 得到函数的计算结果

so, they want to jointly compute a public function f(x,y), get some results.

同时 除了计算结果z外 涉及两个秘密输入的计算过程不会向对方泄露任何其它信息

And then, the computation over these two secret inputs should not leak anything other than the result z to each other.

这就是安全计算的概念

So, this is secure computation.

大家已经知道 安全计算的解决方案有姚氏乱码电路 还有GMW协议

And so, you already know there are Yao’s garbled circuits, and there are GMW protocols that can solve the secure competition.

我们关注什么地方呢？

So, what is our focus?

我们的关注点是 如何让安全计算能在实际中得到应用？

Our focus is that how we can make secure computation really practical?

例如 一个开发者希望开发一个安全计算应用

For example, one developer wants to develop a secure computation application.

他们肯定不想把函数写成电路的形式

So, they don't want to write in a circuit format.

他们想用C语言、Java语言、或者Python语言编写代码

They want to write in a C language, write in a Java or Python language.

因此 源程序和安全计算协议之间存在很大的鸿沟

So, there is a gap between the source program and actual secure computation protocol.

这就是我们ObliVM框架要做的事情

So, this is what our ObliVM framework do.

我们的ObliVM是一个工具 可以将源代码转换成实际的安全计算协议

So, our ObliVM is a tool to translate source programs into the actual secure computation protocols.

这是ObliVM的框架概览

So, OK, this is an overview.

这里面最主要的问题是什么？

And then what is the main question?

Kartik已经提到 主要问题是开发人员喜欢用Python等语言模型编写代码

As Kartik just introduced, a question is the gap between the programmers’ favorite model, for example, this is a Python like language.

幻灯片左侧是开发人员撰写的源程序

And this is what the problem that is really want to write.

但实际上 大多数安全计算协议是在电路模型下实现的

But actually, the most secure computation protocols are treating with these circuit models.

因此 在高层语言程序和电路程序之间存在很大的鸿沟

So, there is a big gap between the high level language programs and the circuit.

我们的问题是 如何将左侧的代码转换为右侧的协议？

So, our question is how can we translate the left part into the right part, OK?

非常感谢Kartik的讲解 你已经提出了我们工作的出发点

So, thank you very much Kartik. You have motivated this well.

这里的关键挑战是 如何能让动态内存访问过程不泄露信息

So, the key challenge here is that how we can make the dynamic memory accesses not leak information.

这是一个不太容易解决的挑战

So, this is not a trivial challenge. But this is not a trivial task.

我们的解决方案是 把RAM模型下的问题转换到不经意RAM模型中

So, actually our solution is that we want to translate the problem in a RAM model language into the oblivious counterpart.

不经意的意思是内存访问和指令追踪过程不依赖于秘密输入

So, here oblivious means that memory accesses and instruction traces do not depend on secret input.

这样一来 不经意程序就可以被进一步转换为电路了

So, in this way, the oblivious program can be easily transferred into a circuit.

我们可以看到 在转换链路中 后一部分相对比较简单 前一部分非常有挑战性

So, we can see, in this transformation, the later part is relatively easy, and the first part is really challenging.

本次讲座主要关注前一部分 解释如何做到这一点

So, this talk will focus on the first part, and how to do that.

有一个非常平凡 也不能说非常平凡的解决方案

So, a very trivial, not very trivial, but a solution

这个方案于2013年首先提出 基本思想是使用不经意RAM

that was proposed first in 2013 was to use oblivious RAM.

不经意RAM 又称ORAM 可以把任意程序编译成不经意程序

So, for oblivious RAM, or ORAM, that can compile an arbitrary program into its oblivious counterpart

基于这一思想 我们去年提出了SCVAM框架 此框架可以仿真通用ORAM

And based on this idea, last year, we presented our SCVM framework that does this work to simulate generic ORAM.

可以证明 我们的渐进性能比所有之前的解决方案都要好

And we can show we can achieve asymptotic performance gain over all previous solutions.

这一解决方案是通用的 也很容易实现

However, this solution is generic and easy to implement.

但问题在于 这个方案可能不是最高效的

But the problem is that it might not be the most efficient way.

在过去的几年间 我们观察到研究人员提出了很多定制化安全计算协议

So, over the past few years, we have observed many customized secure computation protocols.

这方面的工作有很多 我实在没办法在幻灯片上把所有相关工作都列出来

There is a long list of all this work and unfortunately I don't have space to list them all.

这些定制化安全计算协议都很高效

So, they are very very efficient,

可以这么说 这些协议比我们去年的工作都高效

basically I can say they are more efficient than our last year's work.

但问题在于 这些协议的设计成本很高

But the problem is that they incur very high desire effort.

例如 我们与Nina Taft聊了聊 她是我们的合作方

For example, we have talked to Nina Taft. She is our collaborator.

我们私下针对CCS 2013上面发表的隐私矩阵分解算法进行了讨论

And we had a private conversation about their paper on private matrix factorization publishing in CCS 2013.

她告诉我们 她们组织了5位研究人员 花费了大约4个月的时间才完成了全部的实现

And she mentioned to us that it spent a team of 5 researchers about 4 months to implement everything.

也就是说 整个过程花费了超过1.5年的研究时间

So, that is basically more than 1.5 research a year.

这里涉及到巨大的工作量

So, that was a big effort.

我们的问题是：我们能不能做得更好？

So, our question is that can we make it better?

我们能否构建一个通用框架 但仍然获得定制化协议的执行效率？

So, can we build a generic framework, but also achieved customized performance?

这就是我们ObliVM的目标

So, this is about our work ObliVM to achieve this goal.

我们希望非领域专家 例如非密码学家

So, we want to allow non-specialties, so for example, non-cryptographer,

可以用它实现一些安全计算协议 同时获得定制化协议的执行效率

to implement some secure computation protocols while achieving the customized performance.

我们如何做到这一点？

So, how do we do that?

关键思想是：在ObliVM内部 我们提供很多编程抽象接口

So, the key idea is that within ObliVM, we provide several programming abstractions.

我在幻灯片上具体列举了一些 如不经意数据结构、MapReduce、循环合并等

For example, I list many here, oblivious data structures, and MapReduce, and loop coalescing.

我后面会介绍其中一个抽象接口

I will introduce one of them later.

如果想了解更多的细节 请阅读我们的论文

But for more details, I will refer you to the paper.

我还想提及的是 Kartik刚刚讲解的GraphSC论文也是一个并行计算编程接口抽象

And I also would like to mention that the GraphSC paper that was just presented by Kartik is also a program abstraction dedicated for parallel computing.

总之 我们提供了一些编程接口抽象

So, we provide program abstraction.

我这里想简单介绍一下编程接口抽象到底是什么

So, I will give you an intuition about what it is.

我们来看看分布式计算社区

So, let's think about the distributed community.

我估计绝大多数人都听说过MapReduce

I think most of you should heard of this, the MapReduce,

这是谷歌于2004年在OSDI会议上发表的论文

that was published by Google in 2004 in OSDI.

在这篇论文发表之前 并行计算 或者说分布式计算 是一个很困难的任务

And before that paper was published, parallel computing or distributed computing was considered a hard task.

但使用MapReduce 开发者只需要将计算过程编码在Mapper和Reducer框架中

But using MapReduce, the developer only needs to encode their computation into a mapper along with a reducer.

开发者不需要关注分布式计算方法 MapReduce框架会实现整个分布式计算过程

So, the developer doesn't need to worry anything about the distribution. But the MapReduce framework can do everything for you.

因此 与之前的工作相比 使用MapReduce涉及的开发工作量会非常小

So, the development effort using MapReduce is very low, comparing to previous work.

我们想用与之非常类似的方法解决这一问题

So, we want to take a very similar approach here.

我们希望提供一些抽象接口 允许开发者将计算任务编码在抽象接口中

So, we want to provide some abstractions to allow programmers to encode their tasks into these abstractions,

这样他们就不需要关注底层的密码学原语 但仍然获得相同的计算性能

so that they don't need to worry about the underlying cryptography primitives, while achieving the same performance.

这就是我们的目标 这就是我们的解决方案

So, here is our goal. So, here is our solution.

我们如何对外提供这些编程接口抽象？

So, how can we provide these program abstractions?

我们希望实现一个新的语言支持体系

So, we want to implement a language support.

例如 我们想为我们的开发者实现一个新的编程语言

So, for example, we want to implement a new language for our developers.

在开发这个语言时 我们脑海中有两个目标

So, when developing this language, we have two goals in our mind.

第一个目标是：领域专家或密码学家可以很容易地应用此语言实现不同的编程接口

The first one is that, using our language, the experts or cryptographers can implement different programming abstractions very easily.

另一方面 对于非专家来说 他们可能不是特别了解密码学技术

And on the other hand, for those non-specialists, so for example, they do not know very much about the cryptography,

他们可以很容易地使用这些编程语言抽象构建他们自己的应用程序

then they can use these abstractions very easily to build their own applications.

结合脑海中的这两个目标 我们的解决方案是构建新的语言特性

So, with these two goals in our mind, our solution is to support new language features

支持之前系统无法支持的功能

that was never supported before by any previous system.

我这里列举了一些特性 请大家阅读论文 了解相应的技术细节

And so, I list a couple of them here. And I will refer you to the reader for more details.

后续我们会在幻灯片给出的地址上开源我们的编译器

And further, we will open source our compiler here

这样大家可以更好地了解编译器的实现细节

so that you can know even more details about the compiler.

我认为这些特性都很不错

So, I think these features are really cool.

例如 我们可以使用随机类型、虚函数等

For example, we can use random type, phantom functions

从源代码阶段 而不是从后端原语阶段 实现不经意RAM协议

to actually implement the oblivious RAM protocols in the source level rather than as a back-end primitive.

有了这些编程接口抽象 开发人员该怎么做呢？

And OK, so having all of these, then what programmer can do?

例如 我们希望实现一个稀疏图算法

For example, we want to implement a sparse graph algorithm.

我们已经有了编程接口抽象这样一个武器库了

Then we have this arsenal about our programming abstractions.

假设我们想要实现稀疏Dijkstra最短距离算法

Then, for example, we want to implement a sparse Dijkstra’s shortest distance algorithm.

我们可以选取适当的不经意数据结构 这里要选择不经意堆

Then, we can pick oblivious data structure, that is oblivious heap.

随后 我们使用循环合并抽象实现相应的算法

And then, you pick a loop coalescing abstraction to implement them.

通过使用这些工具 我们实现了3个不同的稀疏图算法

OK, then using these tools, we actually implement 3 different sparse graph algorithms.

整个流程好像都走得通

And everything looks fine, right?

但我们得到了一个超出期望的结果 我们实现的算法从理论角度也得到了突破

But we have a very unexpected result, that is, our implemented algorithm made a theoretical breakthrough.

所有3个算法的渐进复杂度都比当前最优算法的渐进复杂度低

So, for example, all these 3 algorithms have better asymptotic complexity than the state-of-the-art.

这个结论令我们感到十分惊讶

So, this is very surprising to us.

如果想了解算法的更多细节 请阅读我们的论文

And for more detail about those algorithms, I would also like to refer you to the reader.

开发人员具体该做些什么呢？

OK, so how do we actually do that?

我在几分钟之前已经向大家许诺过了 我会为大家介绍一个编程接口抽象：循环合并

So, as I just promised several minutes ago, I will introduce you one program abstraction, that is loop coalescing.

这是非常细节的内容了

So, this is a deep detail.

在安全计算中 实现秘密循环是非常有挑战性的工作

So, in secure computation, it is very challenging to implement a secret loop in secure computation,

因为循环次数本身就会泄露信息

because the number of iterations of the loop will leak information.

这里我们允许编程人员编写有上界循环次数限制的循环语句

So, here, we allow the programmers to write a bounded loop.

例如 我们允许协议保护循环次数 但我们要求开发人员公开告知循环次数的上限

For example, we allow the loop to have a secret guard, but we ask the programmer to provide a public bound after the loop.

这是一个嵌套循环代码

So, this is a nesty loop.

这里有趣的地方在于 第4行到第7行的内部循环中

And the interesting thing is that, in the inner loop from line 4 to line 7,

循环次数上界m并不是外部循环中每次迭代的最大迭代次数上界

so the the bound m is not the bound for each iteration of the outer loop.

而是两层嵌套代码中 内部循环的总执行次数上界

But that is a total bound for the entire execution after the 2 nasty loop.

这样我们可以避免重复的执行过程

So that we don't do repeat anything.

例如 如果我们按照传统方法给出上界 则总迭代次数是n·m

So, for example, if we provide a bound, a natural way is to make n·m iterations.

但这样设置的总迭代次数是n+m

But here we only make n+m iterations.

我们的编译器会分析这段代码 自动将此类形式的代码转换成右下的代码形式

And then, our compiler will analyze this code, and automatically transform that into the following one.

这段代码看起来像是一个状态机 这样就不会为算法引入额外的复杂度了

That looks like a state machine, so that we do not incur any extra complexity of the algorithm.

应用所有这些技术以后 我们能得到什么呢？

And so, with all these techniques, so what can we achieve?

我之前已经提到 实现矩阵分解算法涉及的人力开销大约为1年1.5个研究者

So, I I have mentioned that for the matrix factorization, it spends about more than 1.5 researcher a year.

如果使用ObliVM呢？

So, what can we do now using ObliVM?

结果非常令人惊讶 只需要一天就够了

So, that is very surprising. We only do that in one day.

一个博士研究生只需要一天就可以实现全部功能

Only one PhD student can implement everything in one day.

你可能会想 实现结果是否高效？可能自动化的实现结果并不高效

So, you may wonder whether it is efficient or not? For example, that might not be efficient.

事实上 我们的实现的算法效率要高10到20倍

So, but the thing is that we can achieve 10 to 20 times faster.

因此实现结果甚至更高效了

So, that is even better.

这就是我们现在得到的优化结果

So, this is what we can do today.

我相信ObliVM对于所有安全计算开发者来说都是一个福音

So, I believe ObliVM is a valuable thing for all secure computation developers.

我们深入解析一下各个优化点所带来的性能提升情况

OK, so let's look at a detailed breakdown of the performance gain.

我们这里给出的是Dijkstra算法的实现结果 论文中给出了更多的实现结果

So, we give the result of the Dijkstra's algorithm and more result will be shown in the paper.

虽然我们这里只关注ObliVM的编程语言部分

So, although I focus on the programming language part of ObliVM,

但实际上ObliVM拥有一个经过深度优化的后端ObliVM-SC 此后端代码也开源了

actually it has a highly optimized back-end called ObliVM-SC that is already open source.

大家可以访问我们的网站 获取源代码链接

And you can visit our website for the link.

我们实现了一个当前最优的电路ORAM 此ORAM专门为安全计算进行优化

And we implement a state-of-the-art circuit ORAM, that is optimized for secure computation.

我们将我们的系统和之前在CCS 2012上公开的最佳结果进行了对比

And we compare our system with the best previous solution publishing in CCS 2012.

电路ORAM本身可以为我们带来50倍的性能提升

So, the circuit ORAM itself can gain us about 50 times performance speedup.

编程语言和编译器可以为我们带来2500倍的性能提升

And the language and compiler can give us about 2500 speedup.

我们进一步对后后端的其它部分进行了深度优化

And further, we also did some other backend optimizations.

可以在我们的论文中找到相应的技术细节描述

And those details can be found in our paper.

这些深度优化可以为我们带来7倍的性能提升

So, they also gain us 7 times speedup.

总体来说 我们获得了大约100,000倍的性能提升 这是很大的性能提升倍数

So, in total, we achieve about 100,000 times speedup. So, that's a huge thing.

这里给大家一个更直观的方案效率描述

So, give you an even more intuitive solution.

2012年在同一篇论文中 他们在1GB数据集上执行了二分搜索算法

So, in 2012, in that same paper, they did binary search over 1 GB of database.

一次单独问询的计算时间大约为12小时

And only one single query spends about more than 12 hours to compute.

现在情况又怎么样呢？

And what about today?

使用我们的ObliVM框架 每次问询的执行时间仅为7.3秒

So, using our ObliVM framework, it only requires 7.3 seconds for each query.

我们还将我们的SCVM框架与不安全的解决方案进行了效率对比

OK, we also compare our SCVM framework with an insecure solution.

也就是说 我们直接在计算机上执行明文程序 从而对比效率

For example, the plain program running on top of machine or something like that.

我们计算了效率损失量

And we compute the slowdown.

效率损失量相对还是可以接受的

So, there is a very moderate slowdown.

对于分布式GWAS 效率损失仅为130倍

So, for example, for the distributed GWAS, so the slowdown is only 130.

我们可以期待 未来这一数字可能会进一步降低

So, we can expect that in the future, this number will be even lower.

有很多合作方都与我们合作 ObliVM已经在多个场景下得到了应用

OK, so there are several collaborations with us that our ObliVM has been adopted in all these places.

我们刚刚赢得了3月份举办的基因分析竞赛 这是大约2个月前举办的竞赛

And we just won one task of the genomic analysis competition in March, that was 2 months ago.

未来 我们希望在ObliVM框架的基础上实现更多的密码学计算任务 如同态加密等

And for future directions, we are adopting our ObliVM framework to more cryptography tasks, like following homomorphic encryption and something more.

非常感谢大家 这就是我讲座的全部内容了 接下来我可以回答一些问题

So, thank you very much. Here is my talk. I like to take questions.

在演讲开始阶段你给出了一个二分搜索的例子

Hi, so the example you gave right at the beginning of binary search,

开发人员应该怎么实现二分搜索？开发人员需要写什么代码？

what would that look like? What would the programmer have to write?

你指的是哪个例子？

So, which example are you…?

在最开始的地方 你给出了一个二分搜索的例子 那时候你提到…

Right in the beginning, you gave an example of binary search. And you said…

这张图中左侧是开发人员实现循环合并时要编写的代码

Oh, so the left part of this figure is what programmer need to write for the loop coalescing.

你提到了二分搜索的例子 对吧？

So, you mentioned the binary search example, right?

是的

Right.

我觉得用二分搜索举例子会比较好

Yeah, I think that's a good point.

开发人员要做的事情是…

So, the thing is that…

我们回到那页幻灯片上

Let me go back to there.

在这里

Oh here.

就是这里

Here we go.

这就是开发人员需要编写的代码… 我看看这个代码能不能编译通过…

So, this is actually what programmer need to… Let me see if it can compile…

是的 可以用我们的ObliVM编译器编译这个程序

Yes, this program can be compiled in our ObliVM compiler.

编译器会自主处理内存访问过程

So, the compiler will handle the memory access by itself.

是的 是的 编译器会识别相应的模块

Yeah, yeah, yeah. So, yes, the compiler can figure that out,

判断哪些部分要替换为ORAM 哪些部分不需要替换

which part needs to be replaced with ORAM, and which part do not need to do that.

ObliVM是不是有一种潜在使用方法 就是让ObliVM输出C代码 而不是电路

OK, so, is it a potential use for ObliVM to have it just output like C code instead of a circuit,

这样我们就可以把任意一个程序输入给ObliVM 使得程序无法被旁路攻击？

so that you can take basically an arbitrary program, and turn it into one that doesn't have side channels?

这是一个非常好的问题

Oh yeah, that's a very good question.

一个非常有趣的事实是 我们的ObliVM编译器输出的是Java代码

So, it's a very interesting thing that our ObliVM compiler actually emits Java code.

明白

Yeah.

随后 执行Java代码 会生成一个电路

And then, running the Java code will actually produce a circuit.

所以我觉得你提的问题非常好

So, I think that's a very good question that…

我认为未来一个很有趣的研究方向是 如何阻止ObliVM编译器遭到旁路攻击

And I think that will be an interesting future direction to see how our ObliVM compiler can be customized to avoid side channels.

好的 谢谢

OK, thanks.

谢谢你

Yeah, thank you.

什么？

Oh?

你可以使用麦克风的

Actually you can come to the microphone.

我们的编译器没办法被旁路攻击

We don't have side channels.

如果你直接编写类似这样的代码 编译的输出结果是通用ORAM程序

And this one, if you just write it like this, you will be compiled to generic ORAM.

但你也可以调用我们提供的不经意数据结构

But you can alternatively just call our obliviously data structure.

我们在不经意数据结构中实现了二分搜索树

We have binary search trees implemented in our oblivious data structure.

如果直接调用不经意数据结构抽象接口 编译时间会减少log(n)倍

And those, if you use our oblivious data structure abstraction, it will cut a log(n) factor during the compilation.

明白

OK.

很容易让编译器输出C代码 修改后端编译器即可

And it is very easy to emit a C code, just change the backend compiler.

比较困难的点在于前端 如何编译得到一个高效的电路

The more difficult part is in the front-end, how do you compile efficient circuits.

感谢Elaine的解释

Thanks Elaine.

是的 感谢Elaine

Yeah, thanks Elaine.

再次感谢我们的演讲者

And thank our speaker again