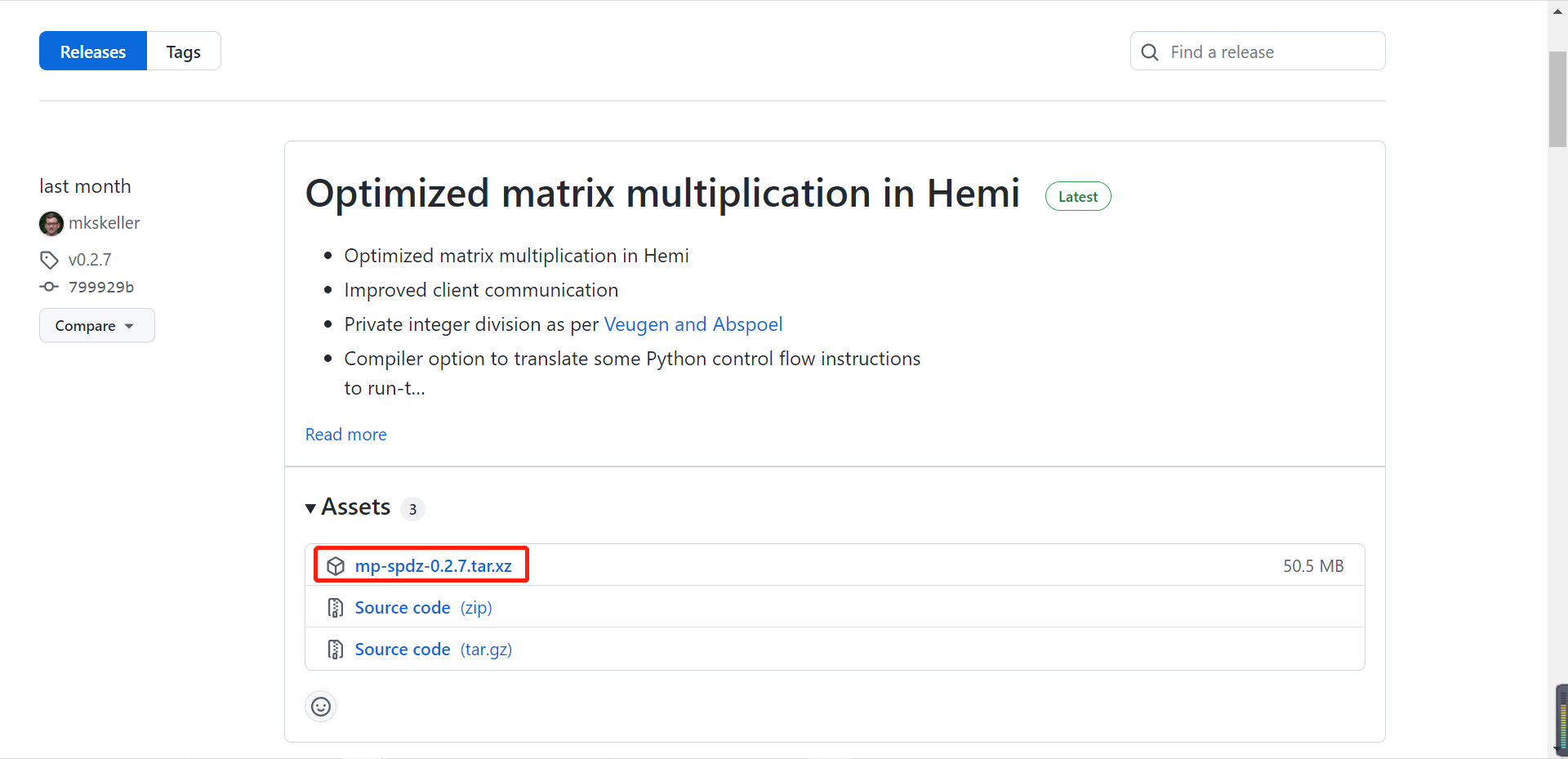
MP-SPDZ实践

下载并解压

<https://github.com/data61/MP-SPDZ/releases>

目前的最新版本如下，上个月（2021.9）才发布的



在ubuntu虚拟机上单独建一个文件夹存放

文本

描述已自动生成

解压

tar xvf mp-spdz-0.2.7.tar.xz

图片包含 日程表

描述已自动生成

虚拟机已经安装了python3



Scripts/tldr.sh

文本

描述已自动生成

执行完后就在/home/linke/MPC/MP-SPDZ/mp-spdz-0.2.7下生成了很多后缀为.x的文件

文本

描述已自动生成

./compile.py tutorial

文本

描述已自动生成

从以上命令的结果可以看出，tutorial的计算逻辑在Programs/Source/tutorial.mpc中定义的，然后被compile.py编译

当前Player-Data目录是空的，接下来执行

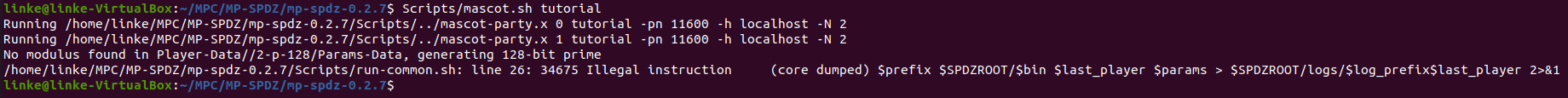
echo 1 2 3 4 > Player-Data/Input-P0-0

echo 1 2 3 4 > Player-Data/Input-P1-0

文本

中度可信度描述已自动生成

Scripts/mascot.sh tutorial



运行命令之后2-p-128文件夹在Player-Data/下生成了，但是里面没有文件

文本

描述已自动生成

从运行的log来看可以看到在执行mascot-party.x的时候参数有ip地址（此处为localhost），可以看出来参与多方安全计算的两方可以分布式执行，各自执行各自的。各自执行mascot-party.x命令。

从log的最后一行可以看出命令执行失败，检查一下log

文本

描述已自动生成

log内容为

No modulus found in Player-Data//2-p-128/Params-Data, generating 128-bit prime

但是经排查，实际上的关键失败log是run-common.sh: line 26: 34675 Illegal instruction

在github上有个人也遇到了同样的错误

<https://github.com/data61/MP-SPDZ/issues/136>

虚拟机缺少ADX，经查，ADX是

Intel ADX (Multi-Precision Add-Carry Instruction Extensions) is [Intel](https://en.wikipedia.org/wiki/Intel)'s [arbitrary-precision arithmetic](https://en.wikipedia.org/wiki/Arbitrary-precision_arithmetic) extension to the [x86](https://en.wikipedia.org/wiki/X86) [instruction set architecture](https://en.wikipedia.org/wiki/Instruction_set_architecture) (ISA). Intel ADX was first supported in the [Broadwell microarchitecture](https://en.wikipedia.org/wiki/Broadwell_(microarchitecture)).

<https://en.wikipedia.org/wiki/Intel_ADX>

在mp-spdz-0.2.7目录下有一个CONFIG文件，即配置文件，里面有两行定义ARCH的



根据github issue里的建议，把第一行注释掉，只保留ARCH = -march=native，还是失败，再把第一行最后的-madx删除，还是失败。暂时无法解决这个问题，目前看起来可能和虚拟机的硬件和操作系统相关，暂且搁置不管。

替代方案：使用docker容器，有人把MP-SPDZ封装到了docker镜像中，这样执行环境就完全一样了。

<https://github.com/MPC-SoK/frameworks>

这个仓库目前为止有349 star，说明被大家认可。

首先创建一个/home/linke/MPC/MPC-SoK文件夹，然后git clone <https://github.com/MPC-SoK/frameworks.git>

进入mp-spdz目录，然后执行docker build -t mp-spdz:0.1.5 .

此处用0.1.5的原因是build过程中下载了MP-SPDZ 0.1.5的压缩包，这样镜像的版本和MP-SPDZ的版本保持对齐。build镜像的时间很长，等待。镜像build完之后大小为747MB。

执行docker run -it --rm mp-spdz:0.1.5，创建一个新容器并进入容器。

编译示例source，总共提供三种：mult3, innerprod, xtabs。我们就以mult3来实操。

$ cd MP-SPDZ

$ ./compile.py -C -p 128 mult3

图形用户界面, 文本

中度可信度描述已自动生成

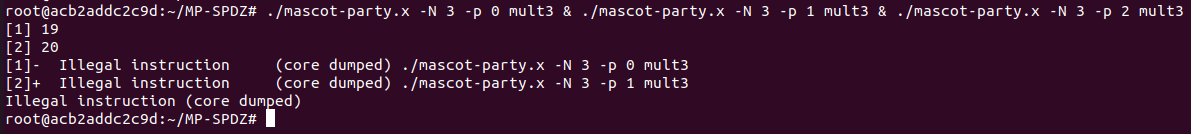
这个示例一共有三方参与计算，接下来为每一方都生成input。

文本

描述已自动生成

接下来就到了三方共同执行计算的时候了

./mascot-party.x -N 3 -p 0 mult3 & ./mascot-party.x -N 3 -p 1 mult3 & ./mascot-party.x -N 3 -p 2 mult3



又遇到了和在非docker环境执行时类似的错误，检查CONFIG文件，发现里面的ARCH配置为

文本

描述已自动生成

容器内执行apt install vim，安装编辑器，修改CONFIG文件的ARCH，修改为

ARCH = -march=native

再次执行./mascot-party.x -N 3 -p 0 mult3 & ./mascot-party.x -N 3 -p 1 mult3 & ./mascot-party.x -N 3 -p 2 mult3

还是报和上面一模一样的错误。

至此，无论通过非docker还是docker环境，看起来都需要CPU INTEL的某些硬件支持，可能是虚拟机没有这些支持，得出初步结论，在虚拟机上跑不了MP-SPDZ。

文本

描述已自动生成

既然本机虚拟机跑不了，去正儿八经的服务器上试一试。

选定服务器oxh-test-hos03，由于centos 7.6默认没有安装python3，安装起来也没有ubuntu那么方便，所以就使用docker的方式运行MP-SPDZ。

首先在本机虚拟机把build好的镜像push到公司的harbor

docker tag 666a226f5b57 10.100.102.82/hos-dev/mp-spdz:0.1.5

docker push 10.100.102.82/hos-dev/mp-spdz:0.1.5

图形用户界面, 应用程序

描述已自动生成

然后在服务器上pull镜像

sudo docker pull 10.100.102.82/hos-dev/mp-spdz:0.1.5

sudo docker tag 666a226f5b57 mp-spdz:0.1.5

sudo docker run -it --rm mp-spdz:0.1.5

echo 14 > Player-Data/Input-P0-0

echo 12 > Player-Data/Input-P1-0

echo 8 > Player-Data/Input-P2-0

然后按照上面的操作再重复操作一遍，终于成功了

文本

描述已自动生成

得出了14\*12\*8=1344的结果，花费7.8秒，又测了一次，花了7.5秒。做一个三个数的乘法要花费7秒以上，三个数来自于三个不同的参与方，不泄露自己的原始参数。

以上是在一台服务器上模拟三方运行的，接下来需要实际探究如何在三台服务器上，一台服务器代表一方，来执行安全多方计算，且不泄露自己的原始input。

看看命令的帮助信息

./mascot-party.x

文本

描述已自动生成

文本

描述已自动生成

重点是一份--ip-file-name文件，里面指定了各方的ip和port，每一方的这个文件都是相同的。

在我们的环境中，文件内容是

10.100.102.118:50000

10.100.102.119:50000

在10.100.102.118和10.100.102.119上都先把mp-spdz:0.1.5镜像拉下来。

然后在两台机器上都创建一个文件parties-ip-port，包含ip和port。启动容器时需要挂载这个文件到容器的/root/MP-SPDZ目录下，和mascot-party.x一个目录。

sudo docker run -it --rm -p 50000:50000 -v /home/admin/MP-SPDZ/parties-ip-port:/root/MP-SPDZ/parties-ip-port mp-spdz:0.1.5

进入容器的MP-SPDZ容器中

10.100.102.116：在Player-Data/Input-P0-0中写入14

10.100.102.118：在Player-Data/Input-P1-0中写入12

10.100.102.119：在Player-Data/Input-P1-0中写入8

echo 14 > Player-Data/Input-P0-0

echo 12 > Player-Data/Input-P1-0

echo 8 > Player-Data/Input-P2-0

然后在三个容器中都运行

./compile.py -C -p 128 mult3

在三台机器依次执行

./mascot-party.x -N 3 -p 0 --ip-file-name parties-ip-port mult3

./mascot-party.x -N 3 -p 1 --ip-file-name parties-ip-port mult3

./mascot-party.x -N 3 -p 2 --ip-file-name parties-ip-port mult3

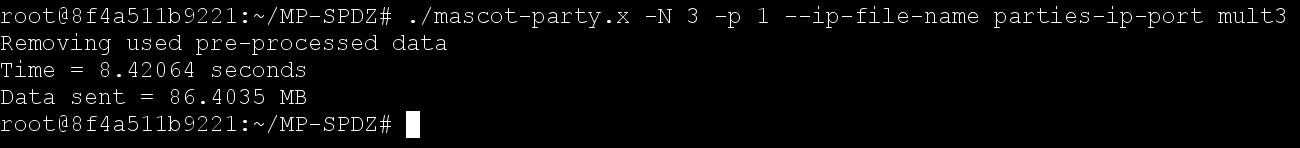
执行后的结果

10.100.102.116(-p 0)

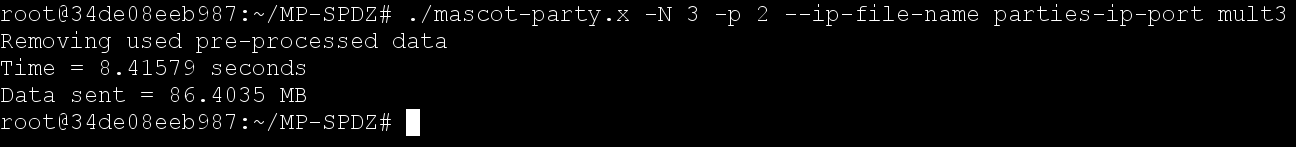
图形用户界面, 文本

描述已自动生成

10.100.102.118(-p 1)



10.100.102.119(-p 2)



从以上三方的结果可以看出，只有party 0即10.100.102.116才得到了最终的结果，这是因为party number为0代表这一方是多方安全计算的发起方，计算的发起方是想通过其他方的数据作为输入，然后得出计算结果的一方，所以只有计算的发起方能够获得最终的结果，其他方都只是提供计算的输入，但是不会泄露原始输入。

另外，值得注意的是，三方提供三个数，计算三个数的乘积，花费了8秒多。

还有一点很值得注意，计算三个数相乘，三方之间的通信量是86MB左右，目的就是为了使原始数据不泄露。

可以看出来通信量大，延时还过得去（适用于一些对延时要求不高的离线分析的场景）。

接下来研究一下以上测试用到的算法mult3.mpc

再次进入其中一个容器

sudo docker exec -it 8f4a511b9221 /bin/bash

文本

描述已自动生成

其实逻辑很简单，就是把三方，每一方都提供一个input，然后把三方的input相乘，得到最后的结果（只有计算发起方才能得到结果）。

接下来以此类推，测试一下三方提供input相加的场景，看需要花费多少时间。重新写一个add3.mpc存放在容器中Programs/Source/目录下，内容为

a = sint.get\_input\_from(0)

b = sint.get\_input\_from(1)

c = sint.get\_input\_from(2)

sum = a + b + c

print\_ln('add3 sum = %s', sum.reveal());

然后进行编译

./compile.py -C -p 128 add3

然后执行

./mascot-party.x -N 3 -p 0 --ip-file-name parties-ip-port add3

./mascot-party.x -N 3 -p 1 --ip-file-name parties-ip-port add3

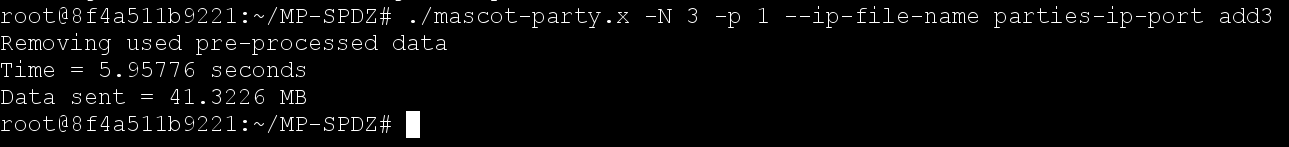
./mascot-party.x -N 3 -p 2 --ip-file-name parties-ip-port add3

party 0（计算发起方）

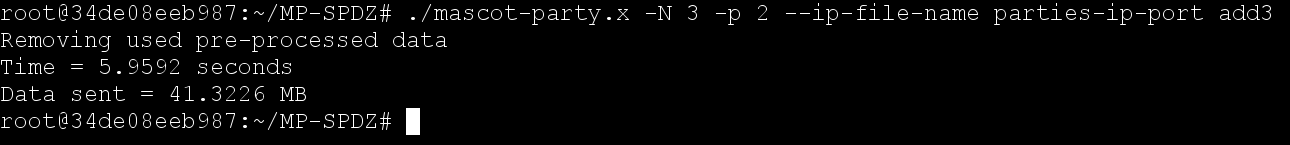
文本

描述已自动生成

party 1



party 2



三个数相加的时延5秒多，通信量41MB。

以上的所有计算都是使用的MASCOT协议，使用不经意传输，恶意模型

MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer

其论文地址为

<https://eprint.iacr.org/2016/505.pdf>

MP-SPDZ 0.1.5提供的所有协议如下，.x后缀

文本

描述已自动生成

接下来试一下spdz2k-party.x计算同样的加法计算。在三个容器依次执行以下三个命令：

./spdz2k-party.x -N 3 -p 0 --ip-file-name parties-ip-port add3

./spdz2k-party.x -N 3 -p 1 --ip-file-name parties-ip-port add3

./spdz2k-party.x -N 3 -p 2 --ip-file-name parties-ip-port add3

party 0

文本

描述已自动生成

party 1

文本

描述已自动生成

party 2

文本

描述已自动生成

可以看到，总共花费1.5秒，比mascot协议花费的时间少；且通信量20MB左右，也比mascot少。

再计算一下三个数相乘

./spdz2k-party.x -N 3 -p 0 --ip-file-name parties-ip-port mult3

./spdz2k-party.x -N 3 -p 1 --ip-file-name parties-ip-port mult3

./spdz2k-party.x -N 3 -p 2 --ip-file-name parties-ip-port mult3

party 0

文本

描述已自动生成

时间和通信量也比mascot少。party 1和party 2的结果就不展示了。

接下来再试试shamir-party.x，基于秘密共享的方案。

./shamir-party.x -N 3 -p 0 --ip-file-name parties-ip-port add3

./shamir-party.x -N 3 -p 1 --ip-file-name parties-ip-port add3

./shamir-party.x -N 3 -p 2 --ip-file-name parties-ip-port add3

文本

描述已自动生成

从错误来看，需要先设置SSL，跳过。

./malicious-shamir-party.x -N 3 -p 0 --ip-file-name parties-ip-port add3

./malicious-shamir-party.x -N 3 -p 1 --ip-file-name parties-ip-port add3

./malicious-shamir-party.x -N 3 -p 2 --ip-file-name parties-ip-port add3

malicious-shamir-party.x也会遇到一样的错误

接下来试试semi2k-party.x，半诚实模型。加法

./semi2k-party.x -N 3 -p 0 --ip-file-name parties-ip-port add3

./semi2k-party.x -N 3 -p 1 --ip-file-name parties-ip-port add3

./semi2k-party.x -N 3 -p 2 --ip-file-name parties-ip-port add3

文本

描述已自动生成

半诚实模型效率非常高，可以用于大量生产环境。

再测试乘法

./semi2k-party.x -N 3 -p 0 --ip-file-name parties-ip-port mult3

./semi2k-party.x -N 3 -p 1 --ip-file-name parties-ip-port mult3

./semi2k-party.x -N 3 -p 2 --ip-file-name parties-ip-port mult3

文本

描述已自动生成

接下来试试semi-party.x，加法

./semi-party.x -N 3 -p 0 --ip-file-name parties-ip-port add3

./semi-party.x -N 3 -p 1 --ip-file-name parties-ip-port add3

./semi-party.x -N 3 -p 2 --ip-file-name parties-ip-port add3

图形用户界面, 文本

描述已自动生成

乘法

./semi-party.x -N 3 -p 0 --ip-file-name parties-ip-port mult3

./semi-party.x -N 3 -p 1 --ip-file-name parties-ip-port mult3

./semi-party.x -N 3 -p 2 --ip-file-name parties-ip-port mult3

图形用户界面, 文本

描述已自动生成

可以看出来，semi2k-party.x比起semi-party.x有很大的性能提升，做了优化，尤其是乘法，体现得很明显。

再试一下semi-bin-party.x

文本

描述已自动生成

有错误，姑且不理。

再测一下hemi-party.x



半同态加密，半诚实模型，不诚实大多数

加法

./hemi-party.x -N 3 -p 0 --ip-file-name parties-ip-port add3

./hemi-party.x -N 3 -p 1 --ip-file-name parties-ip-port add3

./hemi-party.x -N 3 -p 2 --ip-file-name parties-ip-port add3

文本

描述已自动生成

乘法

图形用户界面, 文本

描述已自动生成

可以看出来，hemi和semi2k的加法速度差不多，但是乘法速度就差远了。semi2k的乘法速度是毫秒级。

至此，已经验证了大部分MP-SPDZ支持的协议。最快的是semi2k-party.x，半诚实模型，基于不经意传输。

Semi/Semi2k for semi-honest OT-based computation

Semi and Semi2k denote the result of stripping MASCOT/SPDZ2k of all steps required for malicious security, namely amplifying, sacrificing, MAC generation, and OT correlation checks. What remains is the generation of additively shared Beaver triples using OT.

semi和semi2k是MASCOT/SPDZ2K的阉割版，去掉了和恶意安全性相关的所有步骤。

图形用户界面, 表格

描述已自动生成

表格

描述已自动生成

总结：semi和semi2k用到的技术有不经意传输，秘密共享，半诚实模型（不诚实大多数）。

异常测试

测试一下只有一方输入命令，另外两方一直不输入命令，看一方会等多久

经测试，大概等了1分多钟，重试了25次连接

文本

描述已自动生成

MP-SPDZ的基本使用已经搞清楚了，接下来就要分析如何写更为复杂逻辑的.mpc了，因为很多业务需求是比较复杂的。

<https://mp-spdz.readthedocs.io/en/latest/Compiler.html>