云数据实验报告

1. 相关信息

1.1 论文题目

EnclavePDP: A General Framework to Verify Data Integrity in Cloud Using Intel SGX

1.2 环境参数

由于本组成员使用的CPU有Intel的,也有AMD的,所以使用SGX的模拟工具来进行本次实验.附录中会介绍如何安装相关的SDK.

- 硬件环境
  - Intel i7-9750H 4 core
  - 内存 4GB
- 软件环境
  - VMware Workstation 16 Pro
  - Ubuntu 20.04 LTS

1.3 实验目的

1. 阅读论文,了解目前PDP方案的发展状况与前景.
2. 了解相关的PDP方案和特点.
3. 尝试更改现有的PDP方案.

1.4 实验背景

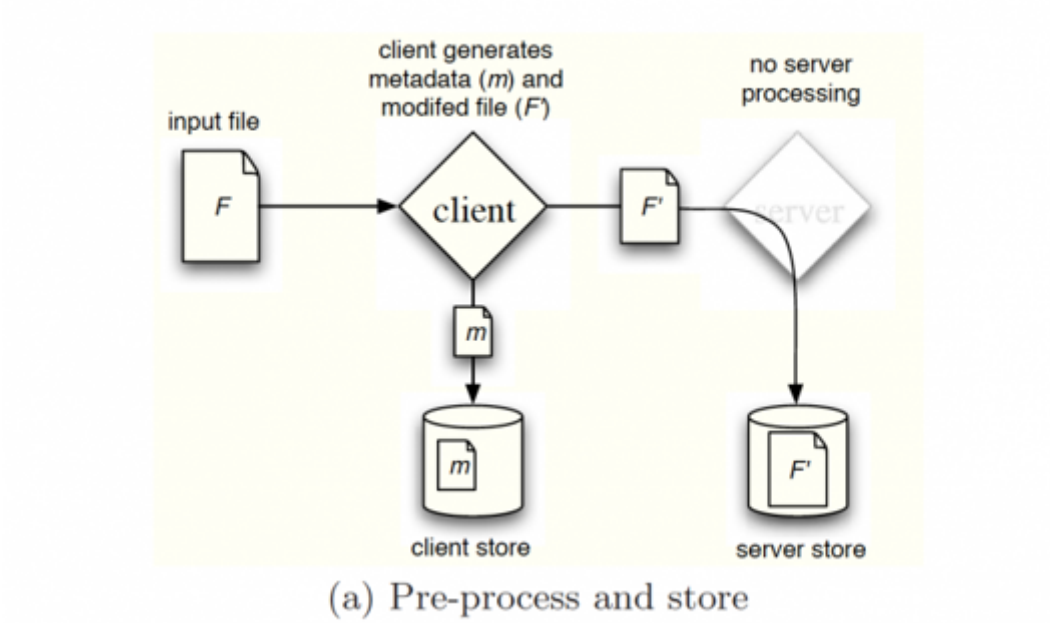
随着云存储服务的普及,远程验证云上外包数据的完整性对用户来说是一项挑战.现有的可证明数据占有(PDP)方案大多是求助于第三方审计员(TPA)来代表用户验证完整性,从而减少他们的通信和计算负担.然而此类方案需要完全可信的TPA,这不是一个合理的假设.所以此篇论文提出EnclavePDP,这是一种安全的通用数据完整性验证框架,依赖Intel SGX为PDP方案建立可信计算基础(TCB),从而消除TPA.

2. PDP协议

2.1 定义

数据所有权证明(PDP)允许客户端在不受信任的服务器上存储数据,它用来验证服务端拥有原始数据而不检索它或自己存储副本.它通过使用生成概率证明协议与远程服务器交互来实现。
由于接入云的设备受计算资源的限制,用户不可能将大量的时间和精力花费在对远程节点的数据完整性检测上.通常,云用户将完整性验证任务移交给经验丰富的第三方来完成.采用第三方验证时,验证方只需要掌握少量的公开信息即可完成完整性验证任任务.

2.2 协议流程



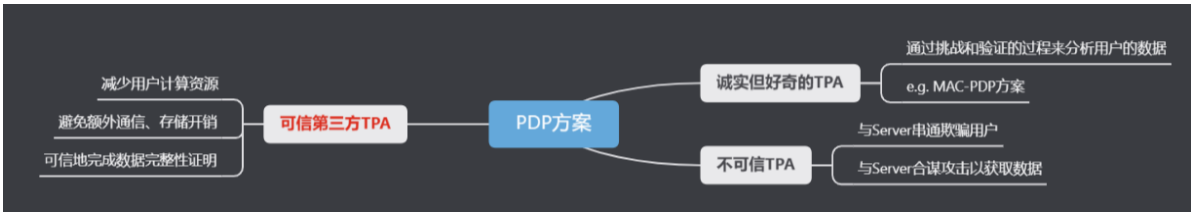
(a) Pre-process and store

数据完整性证明机制由Setup和Challenge两个阶段组成,通过采用抽样的策略对存储在云中的数据文件发起完整性验证.具体实现由4个多项式时间内算法组成,如下所示:

1. **密钥生成算法**:
   $KeyGen(1^n)=(pk,sk)$.由用户在本地执行.n为安全参数,返回一个匹配的公钥,私钥对(pk,sk).
2. **数据块标签生成算法**:
   $TagBlock(sk,F)\rightarrow m$.$TagBlock()$算法由用户执行,为每个文件生成同态签名标签集合$m$,作为认证的元数据.该算法输入参数包括私钥$sk$和数据文件$F$,返回认证的元数据$m$.
3. **证据生成算法**:
   $GenProof(pk,F,m,challenge)\rightarrow P$.该算法由服务器运行,生成完整性证据$P$.输入参数包括公钥$pk$,文件$F$,挑战请求$challenge$和认证元数据集合$m$.返回该次请求的完整性证据$P$.
4. **证据检测算法**:
   $CheckProof(pk,challenge,P)\rightarrow ("true","false")$.由用户或可信第三方$TPA$运行,对服务器返回的证据$P$进行判断.输入参数为公钥$pk$,挑战请求$Challenge$及$P$返回验证成功或失败.

2.3 限制

包含TPA的PDP方案会有如下的限制,也可以通过TPA的类型来进行划分:



可能的威胁:

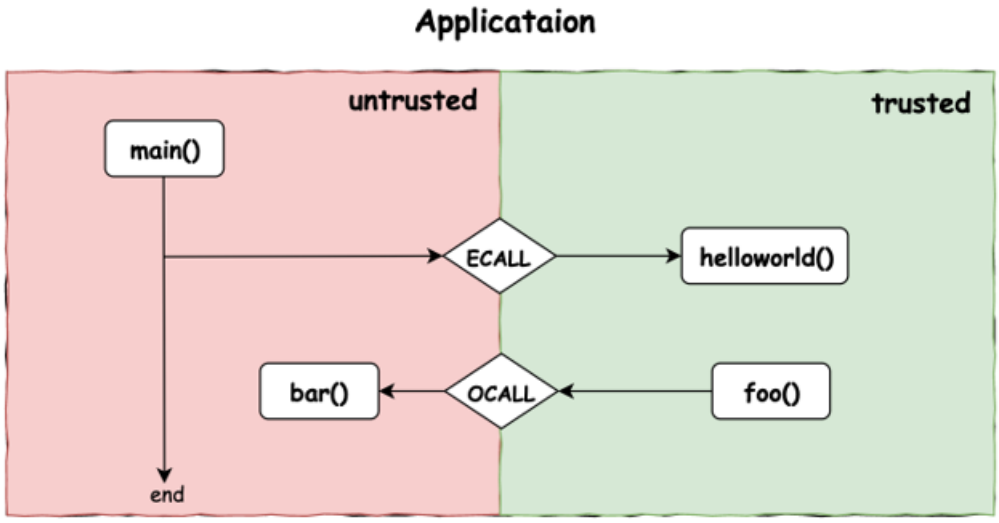
- 来自云存储平台的内部和外部的威胁
- 云存储平台向用户隐藏数据损坏事件
- TPA并不可信

3. SGX简介

3.1 介绍

Intel SGX是一项为满足可信计算行业需求而开发的技术,其保护选定的代码和数据不被泄露和修改.
它允许用户级代码创建称为enclave的私有内存区域,这些区域与以相同或更高权限级别运行的其他进程隔离.在enclave内运行的代码与其他应用程序,操作系统,管理程序等有效隔离.

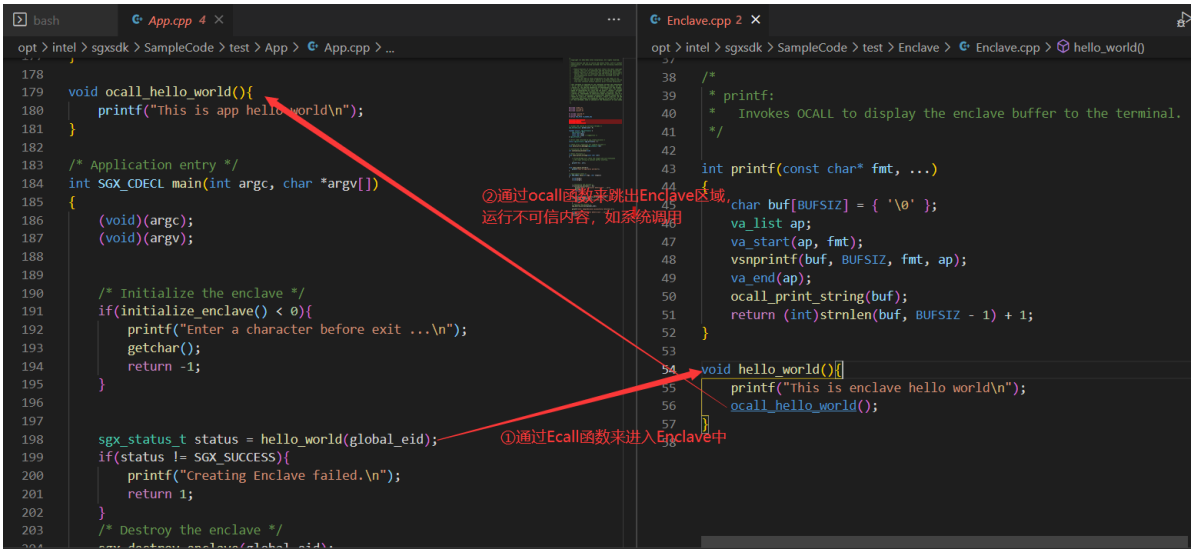
3.2 组成



SGX应用由两部分组成:

1. untrusted不可信区代码和数据运行在普通非加密内存区域,程序main入口必须在非可信区;上图中的main()和bar()函数均在非可信区.
2. truested可信区代码和数据运行在硬件加密内存区域,此区域由CPU创建的且只有CPU有权限访问;
3. 上图的的helloworld()和foo()函数运行在可信区.

下图为示例代码,其中调用流程如图所示:



- 非可信区只能通过ECALL函数调用可信区内的函数.
- 可信区只能通过OCALL函数调用非可信区的函数.
- ECALL函数和OCALL函数通过 EDL 文件声明.

3.3 使用目的

1. **可以防止不可靠的TPA:**
   Intel SGX防止底层的不可信操作系统或管理程序访问飞地内的代码/数据.因此,PDP方案可以在不受信任的平台(即云存储服务器)上忠实地运行,从而消除了对TPA的依赖.
2. **较好的保密性:**
   它还可以保护PDP方案使用的私钥不会泄露给不可信的组件,并且还可以保护验证的完整性不受恶意修改的影响.并且可以提供类似于公共审计的支持.

4. 开销分析与限制

4.1 敏感分析

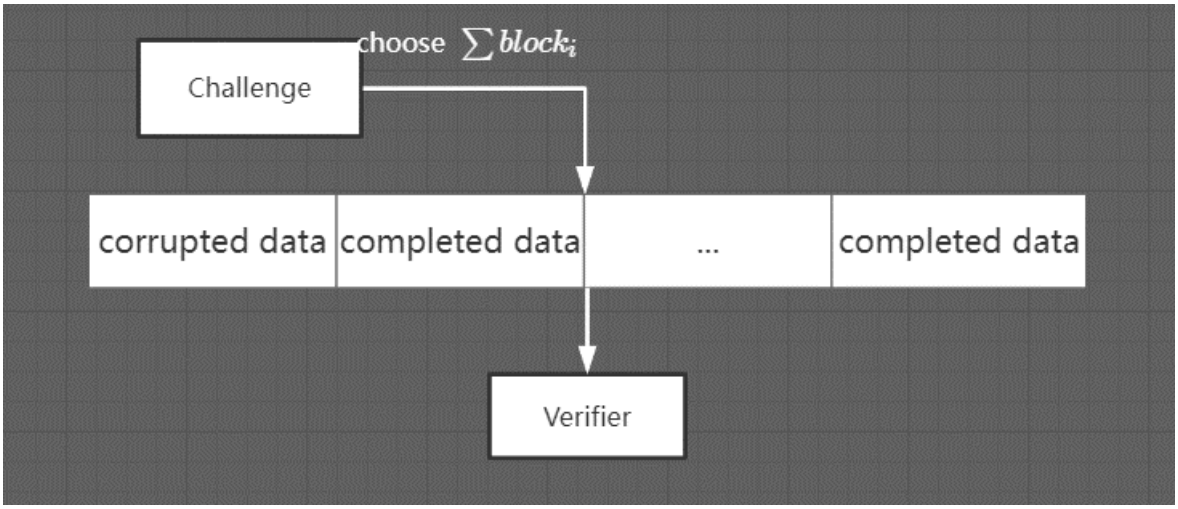
PDP方案的Challenge和Verify操作,是两个安全敏感的功能.并且将密钥加载到Enclave里也是一个敏感的过程,不同的PDP方案在执行时可能涉及额外的验证也会增减敏感的SLOC(即需要保护的代码长度).所以PDP方案移植到Enclave内运行时会一定程度增加敏感的SLOC.下图为论文中给出的各种更改为EnclavePDP方案的开销:

Schemes	SLOC	Security -sensitive SLOC	Security -sensitive functions	SGX -enabled SLOC
MACPDP	1483	115 (7%)	3	121 (8%)
APDP [9]	1348	300 (22%)	3	350 (25%)
MRPDP [11]	1440	476 (33%)	3	624 (43%)
SEPDP [10]	1259	106 (8%)	3	153 (12%)
CPOR [12]	1057	167 (15%)	3	210 (19%)
DPDP [7]	950	117 (12%)	4	145 (15%)
FlexDPDP [8]	945	139 (14%)	4	158 (16%)
PPPAS [19]	1012	199 (20%)	3	249 (24%)
SEPAP [17]	620	162 (26%)	3	225 (36%)
DHT-PA [18]	720	187 (26%)	3	255 (35%)

4.2 Challenge和Verify

PDP方案使用抽样验证,也就是说,通过一次只检查部分数据来实现高精度的验证.如果数据的t部分被破坏,随机抽样c块,将以$p=1-{(1−t)}^c$的概率检测这种破坏.当$t=1\%$时,验证者只需要验证460个随机选择的块来检测这种损坏,概率大于99%.如果文件本身的大小小于460个块,则检验整个文件.

下图为Challenge和Verify的流程:



4.3 可能带来的问题

1. **空间大小局限**:
   在Intel SGX当前的实现中,Enclave可以使用的EPC被限制为128MB,只有93MB可用于应用程序.较小的可用页大小意味着频繁的页面交换操作,因此需要减少代码和数据的内存开销,修建不必要的代码模块.
2. **生态不完善**:
   Enclave中运行的C语言代码中部分库函数并未移植到英特尔SGX,PDP方案中的部分函数内容以及变量需要进行改写以完成对应接口调用,相对改写和植入工作是一个较为麻烦的过程.

3. **切换开销大**:
   在Enclave代码执行可信函数时,SGX调用切换为原语模式,将带来高昂的运行时开销,从而对实用性造成影响.

5. 代码分析与改写

5.1 PDP方案分析

本次实验参考PDP方案为Ahmad1234567/provable-data-possession,我们要想更改他的代码,就需要将改代码读懂.

clone下来代码后,尝试直接编译运行,会发现报错.经过查询可以得知是OpenSSL该库在1.0.x版本中对RSA的相关函数进行了更新,这份代码是10年前的,所以编译不通过,我们只需要使用现在的OpenSSL的接口函数替换即可,最终可以编译运行.

```
In file included from pdp-misc.c:30:
pdp.h:89:19: error: expected ‘=’, ‘,’, ‘;’, ‘asm’ or ‘__attribute__’
before ‘params’
   89 | extern PDP_params params;
      |                   ^~~~~~
pdp-misc.c: In function ‘generate_fdh_h’:
pdp-misc.c:250:14: error: dereferencing pointer to incomplete type
‘RSA’ {aka ‘struct rsa_st’}
  250 |  if(!key->rsa->n) return NULL;
      |              ^~
make: *** [Makefile:23: pdp-misc.o] Error 1
```

阅读代码可以发现,该PDP程序共有三个功能,分别为:

- 生成公私钥对并存入磁盘中.
- 为目标文件使用私钥生成对应Tag.
- 使用公私钥生成挑战来验证文件完整性.

5.2 Enclave代码分析

以安装好SGXSSL的示例代码为例,

```
.
├── app                          不可信区域
│   ├── TestApp.cpp
│   └── TestApp.h
├── enclave                      可信区域
│   ├── TestEnclave.config.xml
│   ├── TestEnclave.cpp          可信函数定义
```

```
│   ├── TestEnclave.edl          可信和不可信函数说明
│   ├── TestEnclave.h            可信函数头文件
│   ├── TestEnclave.lds
│   ├── TestEnclave_private.pem 签名私钥
│   └── tests                    自定义文件
│       ├── bn_conf.h
│       ├── bn_int.h
│       ├── bn_lcl.h
│       ├── bntest.c
│       ├── dhtest.c
│       ├── ecdhtest.c
│       ├── ecdhtest_cavs.h
│       ├── ecdsatest.c
│       ├── ectest.c
│       ├── e_os.h
│       ├── evp_smx.c
│       ├── missing_funcs.c
│       ├── rsa_test.c
│       ├── sha1test.c
│       ├── sha256t.c
│       ├── stdio_func.c
│       ├── threads.h
│       └── threadstest.c
├── Makefile                     整体makefile文件
├── sgx_t.mk                     可信区域makefile文件
└── sgx_u.mk                     不可信区域makefile文件
```

首先我们要更改app中的代码,因为整个程序的入口就在app的TestAPP.cpp中,所以我们可以将pdp-app.c的代码进行选择性的复制.其中生成密钥和生成tag可以放在app环境下运行,因为正常情况下,这两个过程是在客户端完成的,我们这里集合成了一个程序,所以不再进行区分(其实真想区分也可以)

其次,在该运行环境下,我们不能链接动态库,所以我们可以将我们的函数部分的代码编译成一个静态可信库,这样链接就不会有问题了.

- app/TestApp.cpp TestApp.h
  程序的入口代码,以及相关函数定义,或者ocall函数定义
- enclave/TestEnclave.cpp TestEnclave.h
  ecall函数的定义,以及相关声明
- enclave/TestEnclave.edl
  声明ocall函数定义以及ecall函数的定义,包括变量的类型,是否传入以及是否有影响,指针处理问题
- Makefile
  编译该项目的文件,包含相关设置以及可信SSL库的位置,由于有些设置是包含在上级目录中的buildenv.mk中,所以我们需要将其复制到该文件中.
- sgx_t.mk sgx_u.mk
  包含可信(t)和不可信(u)区域代码的编译情况,以及包含某些文件,我们也需要对其

进行简单更改.

5.3 Enclave代码改写

首先我们要明白,Enclave中是不可以链接系统的动态库的,而且静态库要使用编译好的,例如SGXSSL等.而我们的参考代码中还有包含其他头文件,因此直接放入Enclave中是不可取的,而且也尝试过了,编译不过,会出很多问题.因此我们需要将原本的PDP方案代码进行简单更改.

- 生成可信静态库(已经放入lib.mk中)
  首先将pdp-measurements.c和pdp-s3.c去掉后缀,我们用不到这个文件.然后我们将剩下的除了pdp-app.c文件编译程.o文件,最后使用ar进行生成静态库

  ```
  gcc -c *.c
  rm pdp-app.o
  ar crv libpdp.a *.o
  ```

  即可得到一个叫libpdp.a的静态库文件,我们之后就可以将该文件放在app和enclave部分进行链接,当作可信静态库.

- 改写app/TestApp.cpp
  我们需要设置程序的入口,对相关参数进行简单的判断.而且在其中我们需要创建enclave返回量,所以需要将原来的switch结构更改为goto结构.

- 改写enclave/TestEnclave.cpp
  这里我们放ecall函数,所有需要可信环境运行的代码都要放在这里.在edl文件中设置public属性可以在TestApp.cpp中直接调用对应函数.

- 改写enclave/TestEnclave.edl
  设置可信函数和不可信函数,需要对我们使用的函数进行可信说明和不可信说明,例如传入指针则需要说明指针类型,in表示复制到enclave中,out表示在enclave中的变化会影响app,此外还有一个user_check属性,这表示指针内容由用户自行检查,在此我们使用该属性即可.

  ```
  enclave {

    from "sgx_tsgxssl.edl" import *;
    from "sgx_pthread.edl" import *;

    include "../app/pdp.h"
    untrusted {
        void uprint([in, string] const char *str);
        int pdp_tag_file([user_check] char *filepath, size_t
  filepath_len,[user_check] char *tagfilepath, size_t
  tagfilepath_len);
  ```

```
        PDP_challenge *pdp_challenge_file(unsigned int
numfileblocks);

        PDP_proof *pdp_prove_file([user_check] char *filepath,
size_t filepath_len,[user_check] char *tagfilepath, size_t
tagfilepath_len,[user_check] PDP_challenge *challenge,[user_check]
PDP_key *key);
      ...
    };

    trusted {
        public void test();
        public void ecall_verify([in,out] char **optarg,long
st_size);
    };
};
```

- 改写Makefile文件
  该文件原本最开始是`include ../buildenv.mk`的,但是我们并不想包含外部文件,所以可以将该行替换为如下内容:

```
SGX_SSL := /opt/intel/sgxssl
export PACKAGE_LIB := $(SGX_SSL)/lib64/
export PACKAGE_INC := $(SGX_SSL)/include/
export SGX_SDK ?= /opt/intel/sgxsdk/
export VCC := @$(CC)
export VCXX := @$(CXX)
export OS_ID=0
export LINUX_SGX_BUILD ?= 0

UBUNTU_CONFNAME:=/usr/include/x86_64-linux-gnu/bits/confname.h
ifneq ("$(wildcard $(UBUNTU_CONFNAME))","")
  OS_ID=1
else ifeq ($(origin NIX_STORE),environment)
  OS_ID=3
else
  OS_ID=2
endif
```

- 改写sgx_t.mk
  这里设置的是enclave中代码的编译链情况,例如先通过某些文件生成.c文件,然后编译某些文件成为.o文件,最后使用私钥进行签名防止更改等等操作.
- 改写sgx_u.mk
  这里是设置app中代码的编译链情况,同sgx_t.mk,只不过这里多了一步链接的过程,所以这里需要更改的东西比较多.以文件中的代码为例,需要添加包含的库路径,而且链接时也要包含库的内容,还需要更改部分设置.

代码已经上传到了GitHub上面,使用下面的命令即可下载代码.

```
git clone https://github.com/blackh1/pdp_demo.git
```

4. 代码结构介绍

附录

A. SGX相关安装

环境设置

首先对于Ubuntu20.04进行换源,以便于下载相关程序和依赖项,在此不再介绍,网上有详细讲解.

- 更新

将程序进行更新

```
sudo apt update
sudo apt upgrade
```

- 配置相关环境

安装相关依赖项

```
sudo apt-get update
sudo apt-get install libssl-dev libcurl4-openssl-dev libprotobuf-dev
sudo apt-get install build-enssential python
```

安装SGX Driver

- 检查相关需求(以下是Ubuntu)

```
dpkg-query -s linux-headers-$(uname -r)
sudo apt-get install linux-headers-$(uname -r)
```

- 下载Intel SGX Driver

以下是CSDN给出的安装方法

```
wget https://download.01.org/intel-sgx/sgx-
linux/2.16/distro/ubuntu20.04-
server/sgx_linux_x64_driver_2.11.054c9c4c.bin
chmod +x sgx_linux_x64_driver_2.11.054c9c4c.bin
./sgx_linux_x64_driver_2.11.054c9c4c.bin
```

以下是Intel官方给出的安装方法

首先使用git命令下载官方代码,然后使用make命令自行build.

```
git clone https://github.com/intel/linux-sgx-driver.git
cd linux-sgx-driver
make
sudo mkdir -p "/lib/modules/"`uname -r`"/kernel/drivers/intel/sgx"
sudo cp isgx.ko "/lib/modules/"`uname -r`"/kernel/drivers/intel/sgx"
sudo sh -c "cat /etc/modules | grep -Fxq isgx || echo isgx >>
/etc/modules"
sudo /sbin/depmod
sudo /sbin/modprobe isgx
```

官方的卸载方法也写在这里

```
sudo /sbin/modprobe -r isgx
sudo rm -rf "/lib/modules/"`uname -r`"/kernel/drivers/intel/sgx"
sudo /sbin/depmod
sudo /bin/sed -i '/^isgx$/d' /etc/modules
```

最后可以使用 `lsmod|grep isgx` 来查看是否安装成功.

安装SGX SDK

首先安装相关的依赖工具

```
sudo apt-get install build-essential ocaml ocamlbuild automake
autoconf libtool wget python-is-python3 libssl-dev git cmake perl
```

如果这步出现依赖问题,可以使用aptitude来解决.

```
sudo apt-get install aptitude
sudo aptitude install build-essential ocaml ocamlbuild automake
autoconf libtool wget python-is-python3 libssl-dev git cmake perl
```

在解决方案中先选择n,然后选择Y即可.可见这篇文章

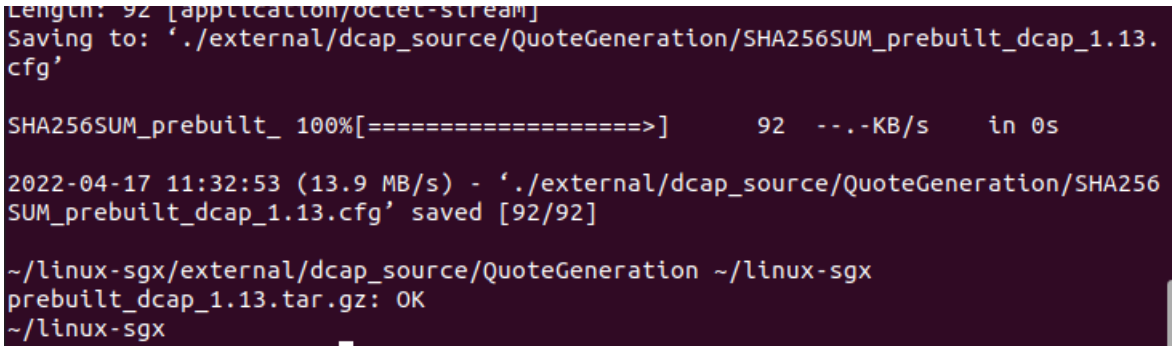
安装SGX PSW(硬件不支持可以不装)

```
sudo apt-get install libssl-dev libcurl4-openssl-dev protobuf-compiler
libprotobuf-dev debhelper cmake reprepro unzip
```

下载相关资源

```
git clone https://github.com/intel/linux-sgx.git
cd linux-sgx && make preparation
```

运行该命令后会进行一系列的下载操作，只需要耐心等候,最终会出现以下字样,即可代表完成



然后我们将对应的工具复制到对应位置以便于之后的使用.

```
sudo cp external/toolset/ubuntu20.04/* /usr/local/bin
which ar as ld objcopy objdump ranlib
```

Build SGX安装程序

```
make sdk
make sdk_install_pkg
```

运行完以上命令后,就可以在 `linux/installer/bin/` 下面了,然后运行 `./sgx_linux_x64_sdk_2.16.100.4.bin` 就可以安装了

第二种方式:

```
wget https://download.01.org/intel-sgx/sgx-
linux/2.16/distro/ubuntu20.04-server/sgx_linux_x64_sdk_2.16.100.4.bin
chmod +x sgx_linux_x64_sdk_2.16.100.4.bin
./sgx_linux_x64_sdk_2.16.100.4.bin
```

如果出现权限不足就使用sudo安装,或者更改/opt文件夹的权限.一定要安装在/opt/intel下,因为后面的SGXSSL默认配置makefile里面都是这个目录.如果更换安装位置,需要更改makefile文件内容,比较麻烦.

安装SGX SSL

首先下载相关源码.

```
wget https://github.com/intel/intel-sgx-
ssl/archive/refs/tags/lin_2.16_1.1.1m_update.zip
unzip lin_2.16_1.1.1m_update.zip
cd intel-sgx-ssl-lin_2.16_1.1.1m_update/openssl_source
wget
https://github.com/openssl/openssl/archive/refs/tags/OpenSSL_1_1_1m.ta
r.gz
tar -zxvf OpenSSL_1_1_1m.tar.gz
mv openssl-OpenSSL_1_1_1m openssl-1.1.1m
tar -zcf openssl-1.1.1m.tar.gz openssl-1.1.1m
cd ../Linux
make all test SGX_MODE=SIM
sudo make install
```

在test运行之后会有结果,显示一堆OK而且有各种测试说明成功.SGXSSL也会安装在 `/opt/intel` 下面