SGX TEE

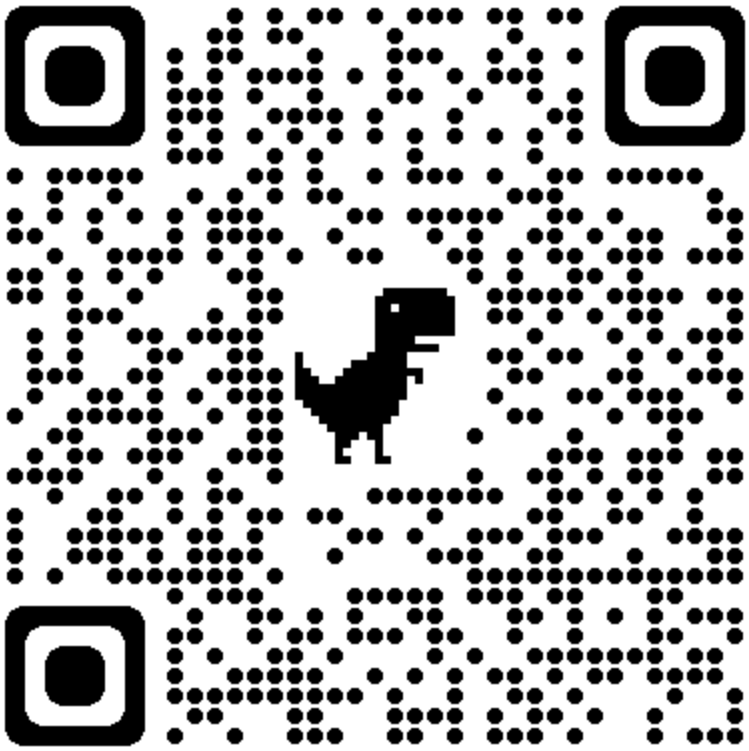
v1.0

简单介绍 SGX TEE 的需求场景和用法。

本期力求简单实用，不设计晦涩的原理细节。

更多资料可以阅读文档：http://xego-dev.basebit.me/doc/xss/2022/05/sgx/

Ver	Date	Description
v1.0	2022-07-07	初版



https://s3.laisky.com/public/slides/SGX.slides.html

需求场景

SGX 一类的 TEE 技术，旨在解决这么一个问题

> 用户要将自己贵重的数字资产，放到不受信任的平台上去运行。需要平台能够确保数字资产不被泄漏。

场景一：数据所有权和使用权分离

最常见的场景就是电影或游戏光盘的产权保护。

厂商作为数据所有权拥有者，希望出售使用权（观看电影，或玩游戏）。

这里的困境是：使用权依托于数据，也就是用户必须要有数据才能使用，但是所有权方却不希望用户能够获取到源数据。

过去人们为了实现产权保护做了很多软件上的尝试，主要手段就是混淆、加壳等，试图增加攻击者做逆向破解的难度。

但是你毕竟是将软件完整地交给不可信的平台去运行，对方有太多的手段可以篡改或破解软件。

你希望找到一个办法，能够在不可信平台上依托于数据输出服务，同时又能保护数据不被泄漏。

场景二：租用第三方算力平台

假如你有大量作为核心资产的数据，但是构建运算模型需要极大的算力，自己没有足够的运算资源。

所以希望能够去 IaaS 云上租用一些机器，但是又害怕这些云厂商会窃取自己的核心数据。

你希望能找到一个办法，能够在不可信的平台上进行运算，并且能保证源数据、中间结果和最终输出的安全性。

安全保证

以 SGX 为代表的 TEE 方案，就是从硬件层面上，提供一个安全可信计算环境的保障。

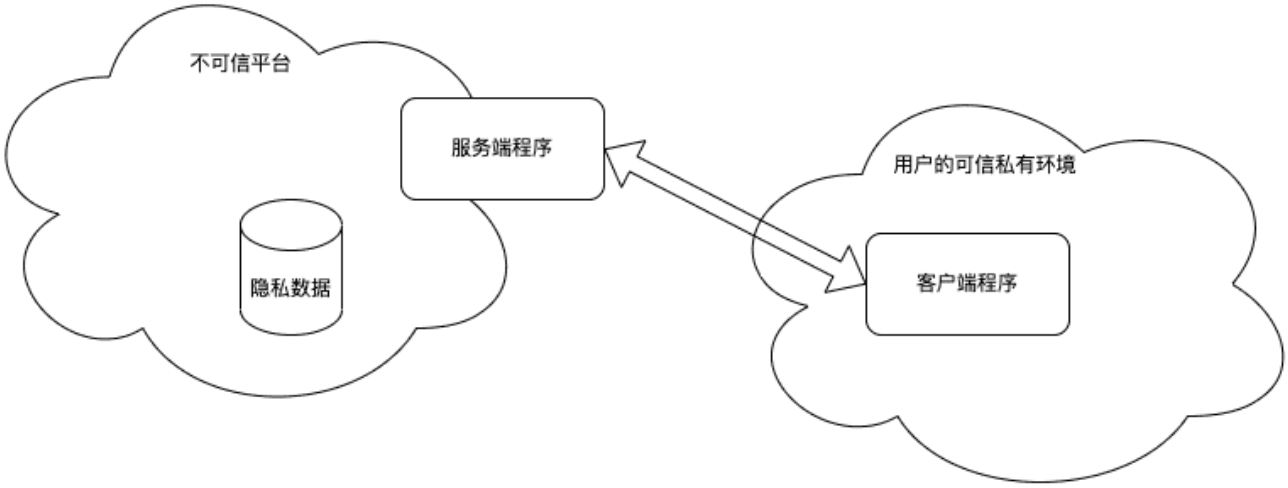
具体内容先不展开，可以简单将 SGX 理解为：为特定的进程，提供一个加密隔离的专属内存区域。

在如下的场景中：用户需要将【源数据】和【服务端程序】放到不可信的第三方【平台】上运行，期间用户的【客户端】程序还会和【服务端程序】进行【通信】，产生一些需要持久化的【中间数据】和最终输出的【结果数据】。

用户的需求可以概括为：

1. 【源数据】加密防泄漏
2. 【服务端程序】完整性（不被篡改）和加密防泄漏
3. 【服务端程序】运行时内存隔离
4. 【通信】加密防泄漏
5. 【中间数据】加密防泄漏
6. 【结果数据】加密防泄漏

做到上述几点，就可以满足我们之前提到过的那些需求。实现在不可信平台上，安全的放置和运行隐私数据，并对外提供服务。



下面一步步地介绍如何做到这几点

前提

要做到上述几点，所需要的最小信任假设（也称为 TCB）是：【平台】必须运行在可信的 SGX 硬件之上。

这一假设可以通过【SGX Remote Attestation（远程认证）】来证明。此处我们先省略该过程细节，先假设我们已经证明了平台是 SGX 可信的。 等介绍完 SGX 能做什么后，再介绍 SGX 是如何做的。

【服务端程序】完整性（不被篡改）和加密防泄漏

【服务端程序】作为一个 SGX 应用，也被称为 Enclave。

在编译完成后，需要用用户的 RSA 私钥进行签名，并且将公钥和签名注入进二进制程序中。SGX 硬件在加载 Enclave 时会校验其签名，从而保证程序的完整性。

Ps. 校验签名确认完整性的过程也被称为度量（measurement）

【服务端程序】运行时内存隔离

所有的 Enclave 程序会运行在特殊的物理内存区域，其中所有的内容都被硬件加密，而且禁止一切来自外部访问。

从硬件层面保障 Enclave 程序的内存数据安全。

【通信】加密防泄漏

在【服务端程序（Enclave）】启动后，客户端可以通过 TCP 对其建立连接。

客户端会通过【远程认证】校验服务端，确认平台 SGX 可信，以及服务端完整性（防止 MITM 攻击）。

服务端可以生成一个 TLS 证书，发送给客户端，然后客户端使用该证书重新建立一个 TLS 连接，就实现了双方的可信通信。

由于此时服务端运行于 Enclave 环境中，可以认为其内存数据是安全的，客户端可以放心的通过 TLS 发送隐私数据给服务端。

【源数据】加密防泄漏

用户可以将【源数据】加密后发送到第三方【平台】。

【服务端程序】中不包含密钥，所以不用担心被泄漏。

【远程证明】完成后建立可信 TLS 连接，可用来传输密钥。然后 Enclave 在内存中解密数据，内存受到 SGX 硬件保护。

【中间数据】加密防泄漏

如果服务端程序运行过程中需要在第三方存储上持久化一些中间数据。因为第三方存储不受到 SGX 硬件的保护，所以需要对这些数据进行加密。

问题转换为，服务端程序如何安全、可重复地生成一个密钥？

方法有很多：

1. 客户端通过 TLS 把密钥传输给服务端
2. SGX 硬件提供了生成 Sealing Key 的接口
    - 缺点是如果更换了 CPU（如发生漂移），那么同一个程序生成的 Sealing Key 也会变

这些密钥都不会离开内存，所以可以放心的对数据加解密，然后放置到第三方存储。

【结果数据】加密防泄漏

如果需要放置到磁盘，那么和上一步【中间数据】的处理方式类似。

如果要传回客户端，那么可以直接通过 TLS 传输。

回顾一下 TEE 四大要素，是不是都涵盖到了：

1. Endorsement Key：程序完整性度量
2. Memory Curtaining：内存隔离
3. Sealed Storage：存储加密
4. Remote Attestation：远程认证

远程认证

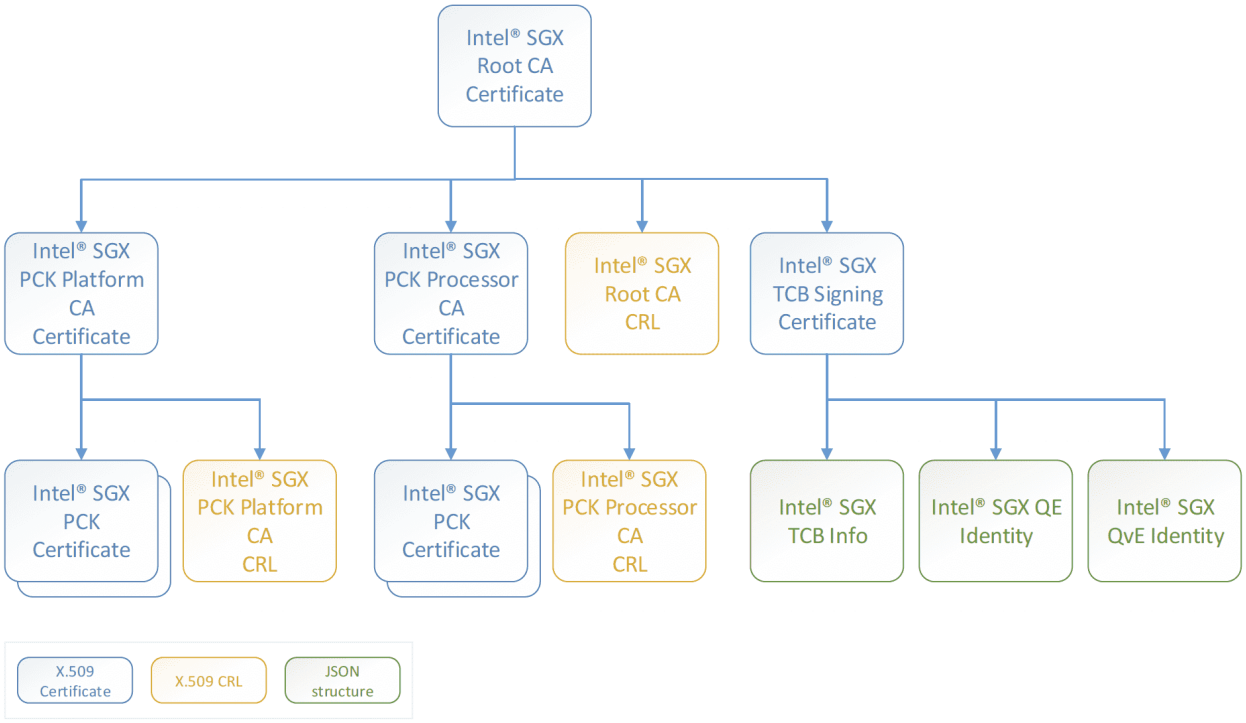
前面介绍了，SGX 硬件是如何满足用户的 TEE 需求的。

接下来的问题就是，用户如何验证平台上的 SGX 环境是可信的呢？

这也是 Remote Attestation 要做的事情。

信任链

SGX 生态也是靠一系列 X.509 证书信任链支撑起来的。

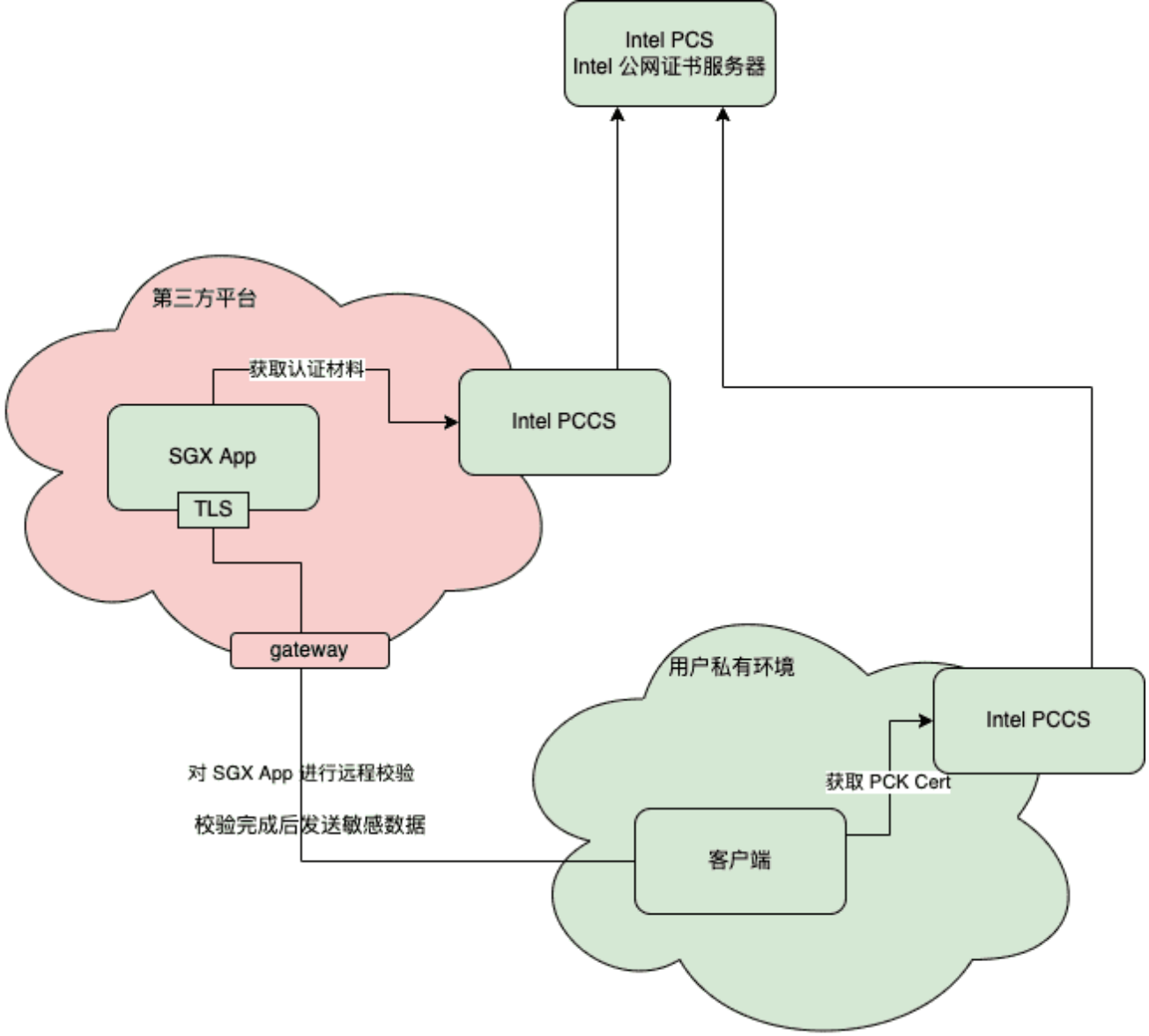


每一个 CPU 硬件，都会内置一个非对称密钥 `Root Provisioning Key`， CPU 会用这个私钥派生出一系列其他公私钥，如 PCK、AK 等，用来签发各种数据， 试图证明这些数据都是由可信的 Intel CPU 所签发。

每个 Intel CPU 的 RPK 都由 Intel 的 RootKey 签发。 Intel 在公网上通过 PCS 服务公开根证书 `PCK Cert` 和吊销列表 `CRL`，可以用来校验 Intel CPU 签发的签名。

RA 流程

1. 客户端要求服务端进行远程认证
2. 服务端请求 SGX 硬件生成 REPORT 证明
3. 服务端将 REPORT 证明发送给客户端
4. 客户端校验 REPORT，证明 SGX 硬件可信，以及服务端程序完整性



REPORT

远程认证的核心数据就是 REPORT。

这是 Intel SGX 软硬件为 CPU 和用户的服务端程序（enclave）签发的证明，其中包括：

- Data: 用户自定义 64 bytes 的数据
- Security Version(`ISVSVN`): 用户自定义应用版本
- Product ID（ `ISVPRODID` ）：用户自定义的产品版本
- Debug: 是否是 DEBUG 模式
- `MRENCLAVE` : 服务端程序的 SHA256
- `MRSIGNER` : 服务端程序的公钥签名
- TCBStatus：硬件状态

证书本身会用 SGX PCK 和 AK 签名。客户端可以用从公网上 Intel PCS 获取到的 `PCK Cert` 确认证书的真实性。（Intel PCS 也提供 CRL 吊销列表）。

一旦确认了证书的真实性，也就证明了平台 SGX 可信，客户端就可以信赖 REPORT 的内容。

通过 `ISVSVN` 、 `ISVPRODID` 、 `MRENCLAVE` 、 `MRSIGNER` 确认服务端程序的完整性。

需要注意的是，假设有一个中间人也运行在 SGX 硬件上，那么他是可以伪造 `ISVSVN`、`ISVPRODID` 的。

只有 `MRENCLAVE`、`MRSIGNER` 是无法伪造的，所以客户端需要至少校验其中一项，来防止 MITM 攻击。

TLS 安全通信

`REPORT.DATA` 是 SGX 生成证书时允许服务端程序自定义填入的 64 字节内容。

服务端可以填入 TLS 证书的哈希。客户端就可以拿来验证服务端的 TLS 证书。从而和服务端建立可信 TLS 连接，防止 MITM 攻击。

这里提到了两次 MITM 攻击：

1. 中间人在 SGX 硬件上运行假的服务端程序，来骗取客户端敏感数据
   - 防御：校验 `MRENCLAVE`、`MRSIGNER`
2. 中间人拦截 TLS 握手，骗取客户端建立 TLS 信道
   - 防御：校验 `REPORT.DATA = SHA256(TLS Cert + nonce)`

`nonce` 是客户端生成的随机串，发送给服务端用于生成 REPORT。这是为了防止重放攻击。 即中间人可能会保存上一次使用过的真 REPORT，来欺骗客户端。

QuickStart

听上去很美好，接下来动手试试。

以 https://git.basebit.me/xss/egox 项目为例

为 Golang 编写 SGX 程序

安装

- http://xego-dev.basebit.me/doc/xss/2022/05/sgx/sdk/
- http://xego-dev.basebit.me/doc/xss/2022/06/sgx/pccs/

参照如上两个文档，安装好 Driver 和 PCCS

EGoX

```
git clone git@git.basebit.me:xss/egox.git
cd egox/sample/raw
yes | cp -rf ../../_deploy/sample .

go mod vendor

# run server
docker build . -f sample/server.Dockerfile -t srv
docker run --rm -it --device /dev/sgx --network host srv

# run client
docker build . -f sample/client.Dockerfile -t cli
docker run --rm -it --device /dev/sgx --network host cli
```

在 build 服务端的时候，日志中会输出 SignerID（`MRSIGNER`） 和 UniqueID（`MRENCLAVE`）。 客户端注意保存这个值，稍后用于远程认证。

egox 主要实现了：

- 把 SGX Enclave 的编译、签发过程完全 Docker 化，极大简化了开发和部署。
  - 客户端机器只需要有 Docker
  - 服务端机器仅需要 Docker & SGX Driver
- 把远程认证流程完全封装，应用内一行启用，基本无感

服务端一行启用远程认证接口：

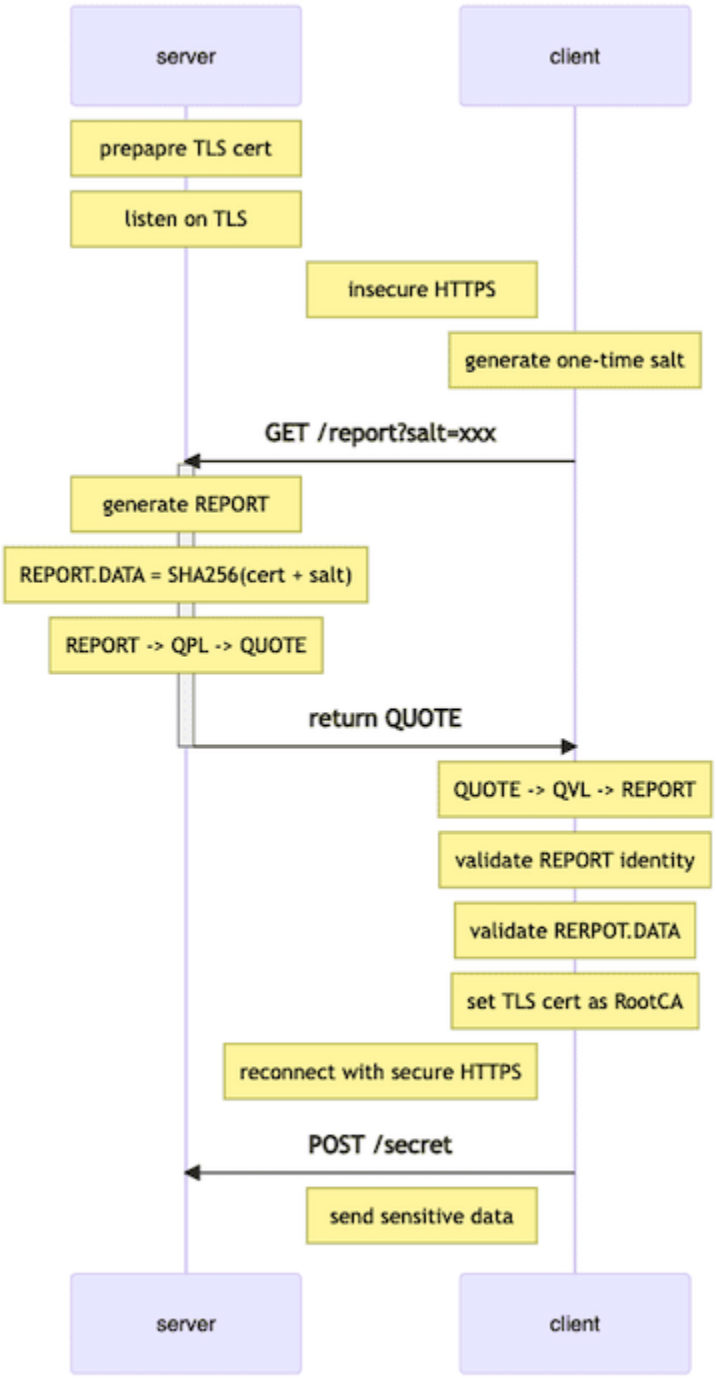
```
// register xego RA handlers
if err := xsrv.RegisterHTTPHandler("/ra", mux, cert); err != nil {
    log.Panic("register ra handler, ", err.Error())
}
```

客户端一行执行远程调用。 成功后返回 tlsConfig，可用于和服务端建立可信 TLS 连接：

```
tlsConfig, err := xcli.VerifyHTTPRemote(endpoint+"/ra",
    xcli.WithVerifySignerID("a4e41e8a269"),
    xcli.WithVerifyUniqueID("e552e96854f"),
    xcli.WithVerifyProductID(1234),
    xcli.WithVerifySecurityVersion(2),
)
if err != nil {
    log.Panic("verify remote", err.Error())
}
```

更多例子可以看 `egox/sample` 文件夹，目前实现了：

- sample/raw: pure Go HTTPS application
- sample/grpc: GRPC server with TLS
- sample/gin: Gin server with TLS



Sealing

- http://xego-dev.basebit.me/doc/xss/2022/05/ego/#sealing>

```
// 可以自己手动加解密
func Decrypt(ciphertext []byte, key []byte, additionalData []byte) ([]byte, error)
func Encrypt(plaintext []byte, key []byte, additionalData []byte) ([]byte, error)

// 也可以使用 SGX 提供的密钥

// 基于 SignerID 的加解密
func SealWithProductKey(plaintext []byte, additionalData []byte) ([]byte, error)
// 基于 UniqueID 的加解密
func SealWithUniqueKey(plaintext []byte, additionalData []byte) ([]byte, error)
```

```
// 解密（不需要指定密钥）
func Unseal(ciphertext []byte, additionalData []byte) ([]byte, error)
```

如果是临时文件，可以考虑采用 SGX Sealing Key。如果是长期存储的文件，还是靠客户端发送对称密钥比较好。 （因为不同 CPU 生成的 SGX Sealing Key 也不一样）

威胁模型

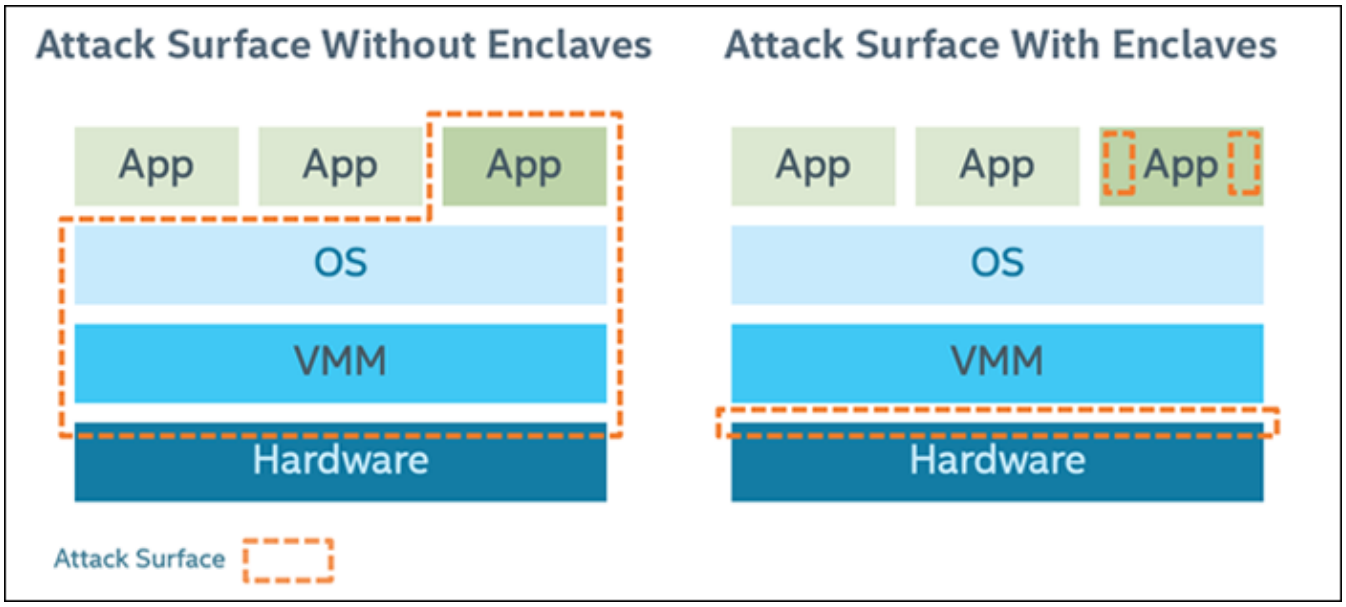
在分析一个安全解决方案的时候，首先要看它的威胁模型是什么，它的 TCB（可信计算基）是什么。

TCB 指的就是指在建立一个微型模型的时候，我们所必须要信任的部分。

就像是在推导一个理论前，所必须依赖的基础定理。

越小的 TCB，往往也意味着更小的威胁面。相对而言，安全程度一般也会更高。

SGX 的 TCB 非常小，仅要求 SGX 硬件可信。



因为所有的程序在执行前都会被 SGX 验签，运行时内存都会被隔离。所以只需要 CPU 可信，就足以构建起整个安全体系了。

而 CPU 可信，由远程认证来证明。

侧信道攻击

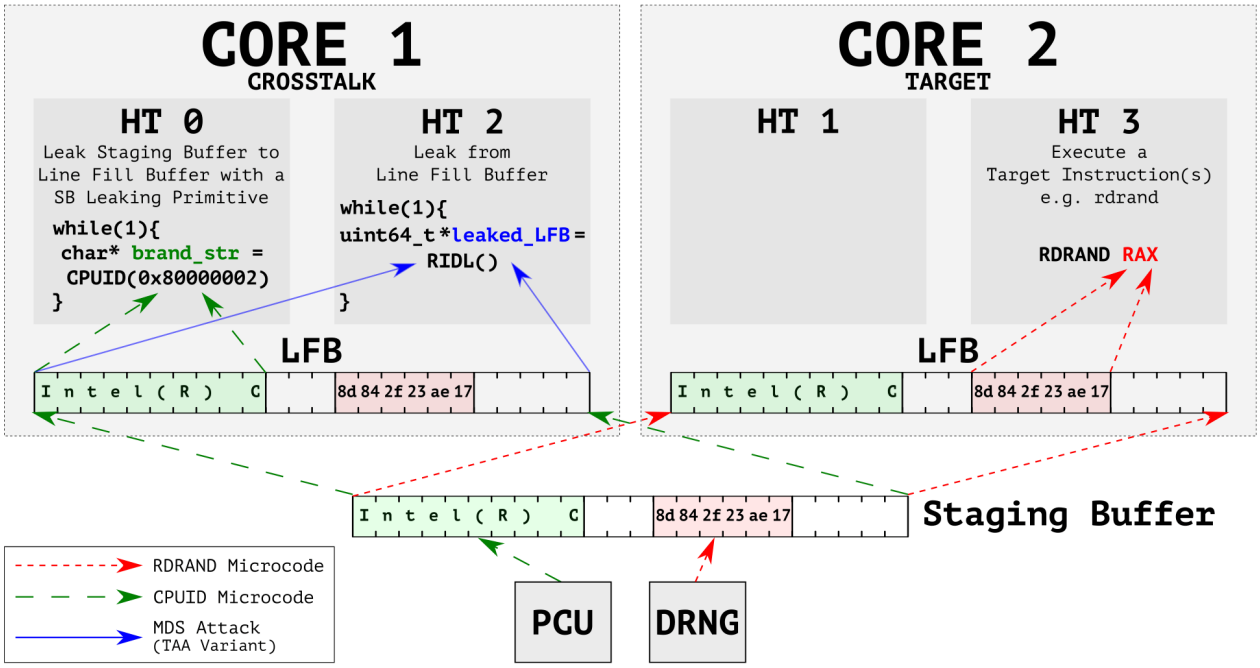
不过经过近十年的应用，SGX 几乎也被打成了筛子。事实证明仅仅依靠 CPU 和内存隔离来保障安全是远远不够的。

以为 TCB 仅包含 CPU，也就意味着 OS 等特权软件都可能是恶意的。

而现代 CPU 中因为各种复杂的优化，实际上在特权软件面前存在大量的暴露面。 而恶意的 OS 通过各种侧信道攻击，就可以突破 SGX 的内存隔离。

一个经典的例子就是 Cache Evict，如 TAA Attack。

TAA 漏洞的本质就是通过占满缓存，让目标数据被 cache evict，然后通过 TAA 漏洞嗅探 LFB，从而获取隐私数据。



TPM SGX

所以目前人们普遍认为 SGX 的威胁模型是不可信的。

TCB 应该包含 SGX 硬件和 OS 等特权软件。而 OS 的完整性就需要 SecureBoot/TPM 等手段来保障。

结语

现在已经可以基于 `xss/egox` 非常便捷的开发、部署 SGX Golang 应用。

各个业务线有需求，或者有兴趣的话，都可以基于此进行开发。 遇到问题可以查阅 XSS 文档，或者直接找我。

如果后续真的产生生产需求，那么还需要建立一套证书管理的基础设施和流程方法，用于维护 `SignerID` 以及对应的公钥轮替方案。

Thanks