浙江大学

本科生毕业论文

文献综述和开题报告

姓名与学号 梁子原 3140103447

指导教师 张帆

年级与专业 2014级信息工程

所在学院 信电学院

一、题目：基于SGX的新型代理重加密技术

二、指导教师对文献综述和开题报告的具体内容要求：

（1）文献阅读：查阅文献（期刊或者会议论文）30篇（含）以上，其中外文文献15篇（含）以上，近三年公开发表的文献10篇（含）以上，书籍不超过3本。（2）外文翻译：1篇学术论文（与导师商定），中文5000字以上。（3）文献综述：中文5000字以上，包括国内外研究现状、研究方向、进展情况、存在问题和参考依据等。（4）开题报告：中文5000字以上，包括选题的意义、可行性分析、研究的内容、研究方法、拟解决的关键问题、预期结果、研究进度计划等。

指导教师（签名）

目　录

文献综述

在这里介绍一下总体介绍一下你毕业设计的需要解决的问题，从而引出相关的技术，进而开始介绍两个比较打的技术背景。

一、背景介绍

1. Intel SGX 技术背景介绍

英特尔软件防护扩展 Intel SGX (Software Guard Extensions)是一种面向应用程序开发人员的英特尔技术。从第六代英特尔酷睿处理器平台开始，英特尔引入了英特尔软件防护扩展新指令集，使用特殊指令和软件可以将应用程序代码放进一个围圈(Enclave)中执行。[1]围圈可提供一个隔离的可信执行环境，可以在主操作系统、BIOS、驱动程序和虚拟机监控器均被恶意代码攻陷的情况下，仍对围圈内的内存数据和代码提供保护。防止恶意软件影响围圈内的代码和数据，从而保障用户的关键代码和数据的机密性和完整性(图 1)。

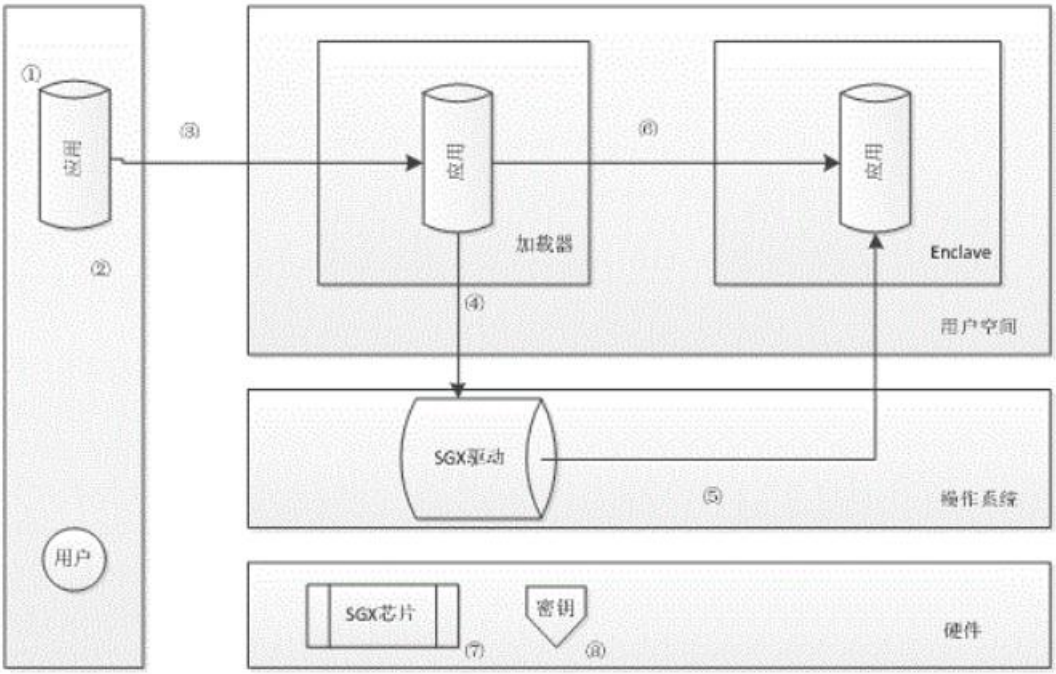

图 1 Intel SGX 原理示意图

Intel 是美国一家主要以研制 CPU 处理器的公司，是全球最大的个人计算机零件和 CPU 制造商，它成立于 1968 年，具有 48 年产品创新和市场领导的历史。Intel 当前的处理器品牌为酷睿系列（Core），先推出的一代 Core 用于移动计算机，上市不久即被 CORE2 取代。酷睿 2，英文 Core 2 Duo，是英特尔推出的

第二代基于 Core 微架构的产品体系统称，于 2006 年 7 月 27 日发布。酷睿 2，是一个跨平台的构架体系，包括服务器版、桌面版、移动版三大领域。其中，服务器版的开发代号为 Woodcrest，桌面版的开发代号为 Conroe，移动版的开发代号为 Merom。

Enclave 安全区域的概念并不是英特尔公司首先提出来的，在 SGX 技术之前也有类似的应用 enclave 的技术。其中比较为人所知的是 TrustZone（TZ）技术。2004 年，ARM 建立安全功能的标准，并正式提出了 TrustZone 技术。TZ 通过以下方式确保系统安全：隔离所有 SoC 硬件和软件资源，使它们分别位于两个区域中。一个是用于安全子系统的安全区域，另一半是用于存储其他所有内容的普通区域。支持 TZ 的 AMBA3 AXI™总线构造中的硬件逻辑可确保普通区域组件无法访问安全区域资源，从而在这两个区域之间构建强大的边界。关于 SGX 技术与 TZ 技术的比较与区别，后面会在开题报告部分予以描述。

SGX 技术于 2013 年在 ISCA 会议的 Workshop HASP 中被提出[2]，但当时只是提出了这一概念和原理，直到 2015 年 10 月第一代支持 SGX 技术的 CPU 才问世。在这两年间，微软基于 SGX 模拟器提出了 SGX 技术两种应用，分别是 Haven(USENIX 2014)[3]和 VC3(SP 2015)。下面解释 SGX 技术中的一些术语：

(1)Enclave Page Cache (EPC)：一个保留加密的内存区域。围圈中的数据代码必需在其中执行。为了在 EPC 中执行一个二进制程序，SGX 指令允许将普通的页复制到 EPC 页中。

(2)Enclave Page Cache Map (EPCM)：为了对每个 EPC 页进行访问控制，会在 CPU 指定的数据结构中维持访问权限和 EPC 页所属的围圈。

(3)State Save Area (SSA): 当围圈内部涉及到一个系统调用时，会发生上下文切换，保存围圈的上下文到 EPC 中保留区域，这部分区域叫做状态保留区域。

SGX 技术的原理主要包含隔离执行和远程认证两个核心机制[4]：

（1）隔离执行

SGX 技术允许一部分应用程序代码在一个安全的运行环境中执行，这个环境称为围圈。SGX 技术可以保障围圈中的数据在运行过程中不会被其他应用程序或高级别的系统软件篡改或监听，例如操作系统，虚拟机管理器，BIOS 等。为了实现保护应用程序代码和数据在运行过程中不受高级别系统软件的攻击，

围圈中的应用代码、数据和围圈相关的关键数据结构在运行过程中被加密存储在内存，即 EPC 之中。SGX 技术不影响传统操作系统对平台资源的管理和分配，实现围圈中的页到 EPC 中帧映射的页表仍由操作系统管理。处理器缓冲线中的数据在写到 RAM 之前，处理器中的内存加密单元（MEE）会对其进行加密，所以操作系统无法获取 EPC 中的数据明文内容。用于对 EPC 页进行访问控制的 EPC 页元数据信息保存在 EPCM 中。通过处理器的访问控制，可以保证每一个 EPC 页只允许来自与其相关联的围圈的访问。

应用在围圈里执行之前，需要将代码，数据和栈等全部加载到围圈中。在所有需要的代码和数据全部加载完毕后，处理器会计算围圈中所有内容的摘要，并与开发者签名中的摘要进行对比，只有经过实际计算的摘要和开发者签名中的摘要相匹配，处理器才会通过应用部署的完整性验证，然后执行应用代码。

（2）远程认证

SGX 技术主要通过 EGETKEY 和 EREPORT 这两条指令完成远程认证的工作。

首先介绍 SGX 技术平台内两个围圈之间的认证流程。同一平台上运行的围圈 A 和围圈 B 之间相互验证身份需要经历以下几个阶段。首先，围圈 A 收到围圈 B 的请求身份认证信息。然后围圈 A 调用 EREPORT 指令生成一个 REPORT 数据结构，以及并生成 REPORT 结构的信息认证码 (MAC)。REPORT 结构信息中主要包含了围圈 A 的身份信息和一些用户数据。MAC 值是由一个报告密钥生成，这个密钥只对目标围圈 B 和相同平台的 EREPORT 指令可见。当围圈 B 接收到 REPORT 信息之后，调用 EGETKEY 指令获取用来计算 REPORT 结构 MAC 值的密钥，进而将重新计算的 MAC 值和收到的 REPORT 的 MAC 值进行对比，确认围圈 A 运行在相同的平台之上。当可信硬件部分得以证实之后，围圈 B 再通过 REPORT 中的信息认证围圈 A 的身份。最后，围圈 A 再用相同的方式验证围圈 B 的身份完成平台内的相互认证。

远程认证机制是在本地平台内认证机制基础上扩展而成。SGX 技术使用一个身份公认的特殊围圈，称为“引用围圈”。只有引用围圈可以访问用于认证的处理器密钥。当目标围圈收到远程认证请求时，目标围圈首先生成回应清单以及包含回应清单摘要的 REPORT 结构信息，并与本地平台内的引用围圈进行

相互认证，在相互认证通过后，引用围圈生成远程认证结果(QUOTE)，并用处理器私钥进行签名。最后将 QUOTE 及其签名、相关清单发送给认证请求者。远程认证请求者收到相关数据后，通过目标围圈平台的公钥证书来验证签名合法性，通过清单内容和摘要认证清单完整性和目标远程平台围圈的身份。另外，远程围圈之间也可以通过 DH（Diffie—Hellman）密钥交换建立一条安全的通信渠道。

2. 代理重加密技术背景介绍

代理重加密（Proxy Re-encryption)是密文间的一种密钥转换机制，是由 Blaze，Bleumer 和 Strauss 等人在 1998 年的欧洲密码学年会上提出的[5]，并由 Ateniese 等人在 2005 年的网络和分布式系统安全研讨会议和 2007 年的美国计算机学会计算机与通信安全会议上给出了规范的形式化定义[6]。具体历史发展会在后文详述。

在代理重加密中，一个半可信代理人通过代理授权人产生的转换密钥 rk 把用授权人 Alice 的公钥 pka 加密的密文转化为用被授权人 Bob 的公钥 pkb 加密的密文，在这个过程中，代理人得不到数据的明文信息，从而降低了数据泄露风险。而这两个密文所对应的明文是一样的，使 Alice 和 Bob 之间实现了数据共享。所谓的半可信，指的是只需相信这个代理者一定会按方案来进行密文的转换。

基本的代理重加密的流程如下：

（1）A(Alice)用 A 的公钥 pka 加密消息 m,将密文交给代理 D

（2）B(Bob)请求该消息后，A 生成重加密密钥 rk,并将 rk 交给 D

（3）D 用 rk 处理密文，将之转化为用 B 的公钥加密的密文，并交给 B

（4）B 收到密文，用 B 的私钥解密得到 m

重加密密钥 rk 由 A 用 A 的私钥和 B 的公钥生成，并发送给 D。该密钥可以由 A 预先生成，并存储在代理 D 处，也可以选择在 B 发出授权申请时，再将重加密密钥 rk 发送给代理 D。

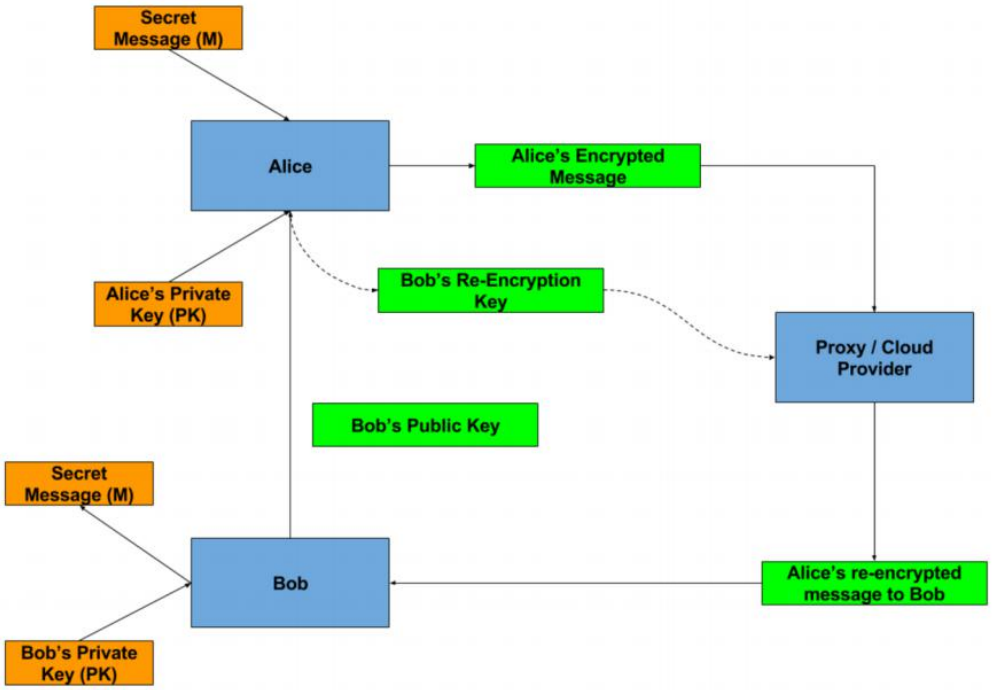

图 2 基本的代理重加密的流程图[7]

这样的代理重加密算法具有以下几个特征：

1）透明性，代理人 D 对于授权人 A 或被授权人 B 来说都是透明的。

2）单向性，一个从用户 A 到用户 B 的授权，不能用于构造一个从用户 B 到用户 A 的授权。

3）非交互性，授权人 A 生成从 A 到 B 的转换函数 Rk 不需要被授权人 B 的参与。

4）安全性，该代理方案应该是 CCA 安全的，并能够抵抗合谋攻击，即用户 A 与代理人的合谋不能得到用户 B 的私钥，用户 B 与代理人合谋也不能得到 A 的私钥。

5）非传递性，通过用户 A 对 B 的授权和 B 对 C 的授权，代理人不能产生 A 对 C 的授权。

根据密文转换次数，代理重加密可以分为单跳代理重加密和多跳代理重加密，单跳代理重加密只允许密文被转换一次，多跳代理重加密则允许密文被转换多次。根据密文转换方向，代理重加密也可以分为双向代理重加密和单向代理重加密。双向代理重加密是指代理者既可以将 Alice 的密文转换成 Bob 的密文，也可以将 Bob 的密文转换成 Alice 的密文。单向代理重加密指代理者只能

将 Alice 的密文转换成 Bob 的密文。当然，任何单向代理重加密方案都可以很容易地变成双向代理重加密方案。而双向代理重加密和单向代理重加密方案只能满足选择明文攻击安全，而实际应用通常要求密码组件能够抵抗选择密文攻击安全。

为此，Calletti 等人在 2007 年的 ACM CCS 会议上提出了首个能在标准模型下证明的 CCA 安全双向代理重加密方案[8]。在 2008 年的公钥密码学会议上，Libert 等人提出了一个无需借助随机预言机的单向代理重加密方案[9]，该方案可以在非自适应攻陷模型下达到选择密文安全。

二、国内外研究现状

1. SGX 技术研究现状

Intel SGX 技术作为一种新兴的前沿技术，目前还处于一个初步的发展阶段，理论方面，国内外的许多学者都已经着眼于 SGX 技术进行相关的研究，理论研究势头迅猛。

目前，围绕 SGX 技术的研究主要可以分成两方面：一部分是基于 SGX 技术自身系统安全的相关问题的研究，一部分是考虑将 SGX 技术应用于安全的相关研究，这个安全既包括应用安全，也包括网络安全。

由参考文献[3]，关于 SGX 技术自身安全方面，Jaebaek Seo 等人实现了防御机制 SGX—ShieldHl 以及代码在围圈中的地址空间随机化，有效地减少了用 SGX 保护的程序受到缓冲溢出攻击的可能性。Ming-Wei Shih 等人实现了在编译器层将原程序自动封装为安全代码的机制 T—SGX 应对恶意操作系统的受控制信道攻击。Sangho Lee 等人提出了分支阴影攻击获知围圈中程序的控制流信息，分支阴影攻击利用了 SGX 技术从围圈模式切换到非围圈模式过程中不清除分支历史的弱点。

关于 SGX 技术在安全方面的应用方面，Andrew Baumann 等人提出的 Haven 系统通过修改 Drawbridge 沙箱，实现了将不经修改的完整 Windows 应用在位于沙箱中的围圈中的安全执行，实现了 SGX 技术对任意应用的支持。Sergei Arnautov 等人实现的安全容器机制 SCONE，实现了用户级别的线程和异步系统调用，在保证较小 TCB 和较低性能开销的情况下，实现了对 Docker 容器进程

完整性和隐私性的保护。Seongmin Kim 等人则较为专注于 SGX 技术在网络系统和网路功能中的应用。利用 SGX 技术实现了保证域间路由认证过程中路由信息隐私性的软件定义域间路由系统，又提出在 Tor 匿名网络中使用支持 SGX 技术的洋葱路由器和目录权威节点。通过 SGX 技术支持，加强目录权威节点中授权密钥和 Tor 节点列表的安全性，并实现洋葱路由器与目录权威节点之间的相互认证，提高 Tor 网络整体的可靠性。更多的研究是关注于用 SGX 技术实现分布式系统中的数据隐私保护。Felix Schuster 等人在保持 TCB 较小的情况下，实现了 MapReducell 框架 VC3。VC3 保证了数据和代码完整性和保密性，并拥有执行可验证性。它的实现依赖于在不同节点中的围圈中执行 Map、Reduce 任务代码，和可验证分布式计算正确执行的高效工作执行协议。Stefan Brenner 等人实现了在较小 TCB 的情况下，保证所管理数据的隐私性和完整性的 ZooKeeper 框架 SecureKeeper。SecureKeeper 通过设计所存储数据的加(解)密方案以及在围圈中对所管理数据进行处理的方式，实现了在不可信环境下保证数据的隐私性和完整性。Tyler Hunt 等人实现了一个在用户使用数据处 理服务时保证用户数据隐私的分布式沙箱 Ryoan。其中沙箱用于避免数据处理代码泄露用户数据，SGX 技术用于保护沙箱的安全。Olga Ohrimenko 等人提出了数据模糊的机器学习算法实现基于 SGX 技术的隐私保护多方机器学习来有效的避免产生由数据依赖访问形式生成的偏信道信息。

从理论上来说，SGX 的应用范围比较广泛，Intel SGX 最关键的优势在于在围圈里的代码和数据只信任自己和 Intel 的 CPU，比较符合当前要解决的云计算安全的需求。当然，SGX 的缺点也是比较明显，最大的缺点是需要开发人员对代码进行重构，将程序分成可信部分和非可信部分，目前有 Intel 发布的 SDK 来协助做这方面工作，但工作量仍然很大，而且非常容易造成秘密泄漏的。第二是性能问题，其中围圈的进出是瓶颈。

目前针对 SGX 应用目前主要停留在研究阶段，SGX 要进入工业界应用尚需时间，一个重要的问题是现在在 Intel 发行的服务器芯片上还没有 SGX，而 SGX 的重要应用就是在数据中心和云端的应用。不过目前一些云提供商还是有所跟进，相信不远的将来会在工业界有一席之地。

2. 代理重加密技术研究现状

代理重加密概念最早由 Blaze 等人在 1998 年的欧洲密码学年会上提出。但是，在 Blaze 等人的文章中，他们并没有给出代理重加密规范的形式化定义，特别是对于代理重签名，使得人们没有很好地认识到代理重加密的好处。

为了改变这种情况，Ateniese 等人对代理重加密进行了形式化的定义，并且使用双线性对构建了随机预言模型下安全的单向代理重加密方案，该方案能阻止代理者和受委托者合谋来揭示委托者的密钥。至此代理重密码学引起了广泛的关注，各种各样的代理重密码方案及其应用被提出。

Ateniese 等利用双线性配对构造了单向 PRE 方案，他们的 PRE 方案都只是在标准模型下满足选择明文安全（CPA, chosen-plaintext attack）的。但实际的应用场合通常都是要求方案满足选择密文安全（CCA，chosen-ciphertext attack）的。

Canetti 等首次利用 CHK(canetti-halevi-katz methodology)技术构造了一种在标准模型下满足 CCA 安全的双向多跳 PRE 方案，但是与 Atteniese 等提出的双向 PRE 方案缺点一样，该方案也不能抵抗共谋攻击。

由参考文献[10]，2006 年 Green 和 Ateniese 把 PRE 概念推广到基于身份的密码机制，提出了基于身份的 PRE 方案（Identity-Based PRE, IB-PRE）。Libert 和 Vergnaud 在 2008 年 PKC 会议上，提出了第一个标准模型下选择密文安全的单向 PRE 方案。以上方法最大的缺陷在于双线性配对运算比较消耗计算资源。

传统的代理重加密允许代理者对授权人的所有密文进行转换，因而无法较好地控制代理者的转换能力。为了解决这一问题，翁健等人提出了条件代理重加密(Conditional Proxy Re-Encryption，C.PRE)的概念，在 C．PRE 中，只有当密文符合某种条件时，代理才可以成功地对该密文进行转换。

总体来讲，有关代理重加密的研究范围比较广，关于代理重加密的各种分支以及它们在各个具体领域的应用也比较多，在这里也就不再赘述。

换行

参考文献

搜一下邵俊老师的博士论文或他发表的关于Proxy Re的文章，介绍一下，然后添加引用

请查看一下文献综述的相关要求，参考文献太少了。建议要达到15篇以上。最好20篇。

[1] Hoekstra.M　Intel SGX for Dummies https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx 2013.9.

[2] McKeen F, Alexandrovich I, BerenzonInnovative A. Instructions and Software Model for Isolated Execution.　HASP workshop 2013

[3] Baumann A，Peinado M，Hunt G．Shielding applications from an untrusted cloud with haven．ACM　Transactions on Computer Systems(TOCS)，2015

[4] 王进文, 江勇，李琦等. SGX 技术应用研究综述. 网络新媒体技术. 2017.9

[5] M.Blaze, G.Bleumer,　M.Strauss．Divertible Protocols and Atomic Proxy Cryptography. In advances in Cryptology—Eurocrypt'98，1998

[6] G．Ateniese，K．Fu，M．Green．Improved Proxy Re-encryption Schemes with Applications to Secure Distributed Storage．NDSS 2005.　2005．

[7] Vassallo D. Proxy re-encryption-David Vassallo's Blog. http://blog.davidvassallo.me/2016/04/11/proxy-re-encryption/　2016

[8] R.Canetti and S.Hohenberger. Chosen-Ciphertext Secure Proxy Re-Encryption. In Proceeding of ACM CCS 2007,　2007．

[9] B.Libert, D.Vergnaud. Unidirectional Chosen-Ciphertext Secure Proxy Reencryption. PKC'08.　2008

[10]周德华．代理重加密体制的研究[学位论文] 上海交通大学计算机体系结构专业 2013.5

[11]其他参考文献：http://www.isee.zju.edu.cn/fanzhang/site_en/research/area03.html

开题报告

一、研究开发的背景、意义与目的

1. 背景介绍

本次研究的主题为“基于 Intel SGX 的新型代理重加密技术的实现”，其中，最主要也是最关键的两个概念就是 Intel SGX 技术和代理重加密技术。下面分别对 Intel SGX 技术和代理重加密技术两者的背景进行简要的介绍。

Intel SGX 是近两年发展势头迅猛的新兴技术。SGX 技术全名英特尔软件防护扩展 Intel SGX (Software Guard Extensions)，顾名思义，其是对英特尔体系（IA）的一个扩展，用于增强软件的安全性。

2013-2014 年，Intel 发布了三个博客给出了 SGX 八个的设计目标[1]：

（1）允许应用开发者保护敏感信息不被运行在更高特权等级下的欺诈软件非法访问和修改。

（2）能够使应用可以保护敏感代码和数据的机密性和完整性并不会被正常的系统软件对平台资源进行管理和控制的功能所扰乱。

（3）使消费者的计算设备保持对其平台的控制并自由选择下载或不下载他们选择的应用程序和服务。

（4）使平台能够验证一个应用程序的可信代码并且提供一个源自处理器内的包含此验证方式和其他证明代码已经正确的在可信环境下得到初始化的凭证的符号化凭证。

（5）能够使用成熟的工具和处理器开发可信的应用软件.

（6）允许受信任的应用程序，可随应用程序所采用的处理器的功能的性能。

（7）使软件开发商通过他们选择的分销渠道可以自行决定可信软件的发布和更新的频率。

（8）能够使应用程序定义代码和数据安全区即使在攻击者已经获得平台的实际控制并直接攻击内存的境况下也能保证安全和隐秘。

Intel 的 SGX 技术增强安全的方式并不是识别和隔离平台上的所有恶意软件，而是将合法软件的安全操作封装在一个围圈（enclave）中，保护其不受恶意软件的攻击，特权或者非特权的软件都无法访问该围圈，也就是说，一旦软件和数据位于围圈中，即便操作系统也无法影响该围圈里面的代码和数据。围圈的安全边界只包含 CPU 和它自身。这与早前 ARM 提出的 TrustZone（TZ）的概念比较相似。二者都可以理解为一个可信执行环境（TEE，Trusted Execution Environment），不过其与 TZ 还是有区别的，TZ 中通过 CPU 划分为两个隔离环境（安全世界和正常世界），两者之间通过 SMC 指令通信；而 SGX 中一个 CPU 可以运行多个安全围圈，并发执行亦可。简单比喻来讲，TZ 是个公用大保险柜，什么东西都装进去，有漏洞的代码或程序可能也进去了，而且保险柜钥匙在管理员手上，必须相信管理员。SGX 每个围圈有自己的保险柜，钥匙在自己手上，提高了程序或代码自身的安全性。

近几年 SGX 技术受到了较多学者的关注，有关 SGX 技术的研究也非常多[2]。目前，对 SGX 技术的研究还依然停留在理论研究方面，想要进入工业应用至少也需要十年左右的时间。

相比于 SGX 技术，代理重加密概念的提出相对较早。代理重加密(Proxy Re-Encryption)是由密码学家 Blaze，Bleumer 和 Strauss 在 1998 年的欧洲密码学年会上提出的一个密码学概念[3]。在代理重加密系统中，代理者在获得由授权人产生的针对被授权人的转换钥（即代理重加密密钥）后，能够将原本加密给授权人的密文转换为针对被授权人的密文，然后被授权人只需利用自己的私钥就可以解密该转换后的密文。代理重加密能够进一步保证:虽然代理者拥有转换钥，他依然无法获取关于密文中对应明文的任何信息。代理重加密中最重要的问题就是代理重加密的重加密密钥的生成，如果 A 与 B 属于同种加密系统，重加密密钥的生成还没有非常复杂，但如果 A 想要与处于另一种加密系统中的 C 完成代理重加密的过程，那么这个重加密密钥相对较为复杂，也非常影响代理重加密系统整体的效率。代理重加密在很多场合有着广泛的应用，如数字版权保护、分布式文件系统、加密垃圾邮件过滤、云计算等等[4]。

2. 本研究的意义和目的

目前，SGX 无疑是一颗潜力巨大的新星，发展势头迅猛，近年来安全方面的国际会议也都无一例外或多或少地涉及了 Intel SGX 技术的应用。SGX 技术在密码学方面也取得了突破。函数加密（FE, Functional Encryption）一直是密码学中较受关注的课题之一，然而之前的函数加密实现方案大多复杂度高且效率较低。Ben Fisch 等人构建了基于 SGX 技术的函数加密系统[5]，大大提高了函数加密实现的效率，增加了函数加密从理论研究走向实际应用的可能。

代理重加密技术比函数加密在应用方面走的更久更远，但是代理重加密本身依然存在一定的缺陷。比如传统的代理重加密无法抵抗合谋攻击，只能抵抗选择明文攻击等。有关代理重加密的相关研究很多，也衍生出了一些代理重加密的分支去弥补代理重加密本身的缺陷，但如果将新兴的英特尔 SGX 技术与代理重加密技术结合起来，能否也利用 SGX 技术解决代理重加密机制中的这些隐患，增加其安全性？

同时代理重加密的一个重要问题是，在同种加密系统下，重加密密钥的生成效率比较高，但跨系统的代理重加密中重加密密钥生成的效率较低。能否利用 SGX 技术达成不同加密系统之间的代理重加密，提高其效率？

本研究的主要目的即是将 Intel SGX 技术与代理重加密结合起来，构造一个基于 SGX 技术的代理重加密系统，并根据上述问题对该系统进行分析和优化。同时，于我个人而言，本次研究也是一个比较合适的入门接触 Intel SGX 技术的机会。

二、主要研究开发内容

1. 主要研究内容

目前，有关 SGX 技术的研究大多可分为两方面：一个是 SGX 用于解决安全问题，一个是研究 SGX 本身的安全问题（详见文献综述对应部分），本次研究内容属于第一类，利用 SGX 技术解决安全问题。

本研究的主要研究内容是利用 Intel SGX 技术实现一个代理重加密系统仿真

以及硬件实现，并对其进行安全性与加密效率进行测试。并与非 Intel SGX 技术实现的代理重加密技术的性能进行对比。

第一，对于同种加密系统下的代理重加密，此时重加密密钥的生成以及转化效率已经足够高，而利用 Intel SGX 技术进出围圈的过程中，也会产生一定的消耗，所以理论上讲，对于同种加密系统下的代理重加密系统，添加 SGX 技术后，系统的效率会降低，安全性会有所提高，所以会主要考虑测试并比较其安全性。

第二，对于不同加密系统之间的代理重加密，存在的主要问题是不同加密系统之间的用户之间的重加密密钥的生成会较为复杂。为了解决这一问题，初步想法是放弃代理重加密密钥的生成，而是代替为在围圈中执行解密再加密的过程，即在围圈中用 A 的私钥对密文解密，在用 B 的公钥对解密得到的明文进行加密。这样操作得到的结果与代理重加密得到的结果一致，但是却不需要生成重加密密钥，起到了两个加密系统之间的接口与桥梁的功能，但是由于省去了生成重加密密钥的过程，规避了由其带来的效率问题，同时，由于围圈内本身的安全性，有保证了中间过程产生的明文的安全性。因此，对于不同加密系统间的代理重加密，会主要测试与比较其效率。

同种加密系统之间的代理重加密已经有比较多的非 SGX 方法来增加其面对各种攻击的抗性，因而相比第一部分，我认为第二部分的实现更具有实际意义。因此，本研究会以第二部分为主。在基本实现不同加密体系之间的代理重加密后，如有机会，再去探究第一部分的问题。

2. 技术路线

后面的研究过程总体可以分为四阶段：

首先是仿真阶段。

仿真将利用 Intel 官方发布的 SGX_SDK。2016 年，英特尔在其官方网站上发布了 Intel SGX 的 SDK 以及 PSW。英特尔软件防护扩展 SGX_SDK 是 API、函数库、文档、样本源代码和工具的集合,允许软件开发人员用 C/C++创建和调试启用英特尔软件防护扩展的应用程序。SGX_SDK 同时提供 Microsoft Visual Studio 插件,可用标准开发工具开发围圈。通过和 Microsoft Visual Studio 的集成,

调试围圈的方法和调试普通 Windows 程序的方法完全一样。要注意的一点是 Microsoft Visual Studio 默认的 debugger 为 Local Windows Debugger，必须修改为 Intel(R) SGX Debugger ,否则围圈内的代码部分打的断点不起作用。

本研究的主要仿真过程都将利用 SGX_SDK 完成。当仿真过程构建系统基本完成后，会对仿真系统的功能，性能与安全性进行测试，并进行针对性的调整与改良。

其次是硬件实现阶段。

硬件实现阶段主要是在支持 SGX 技术的计算机上实际运行，并进行调试与测试性能。

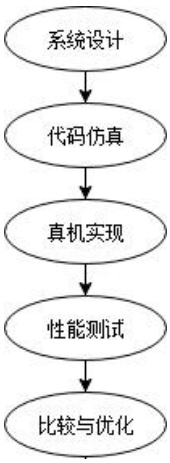

图 1 技术路线

这个图是故意这么放的吗？ 感觉格式问题

最后是性能测试与对比优化阶段。

本阶段会将基于 Intel SGX 技术的代理重加密与非 SGX 的代理重加密进行对比，分析数据，优化系统，并与预期结果相对照。

其中，至少要完成仿真，实现，与测试三个部分。至于优化与对比，会根据剩余时间来调整任务量和规模。

3. 可行性分析

我认为利用 Intel SGX 技术构建一个代理重加密系统是可行的。之前已经有将 SGX 技术运用到加密系统研究中，并获得国际学界的认可。Ben A Fisch 与 Dhinakaran Vinayagamurthy 等人在 2016 年提出利用 Intel SGX 技术构建一个函数加密系统，他们称其为 Iron。他们所实现的 Iron 系统在保证了安全性的同时，大大提高了函数加密实现的效率，是使函数加密从理论走向应用的重要一步，他们的这篇论文也因此摘得 CCA2017 最佳论文之一。

函数加密的概念由 Boneh, Sahai and Waters 在 2011 年提出[6]。函数加密其实和代理重加密的概念有一部分相似。代理重加密是希望跳过解密的步骤，直接完成从密文 A 到密文 B 之间的转化，将先解密再加密的过程合二为一，以防止本应作为中间过程的明文发生泄露。函数加密则是希望跳过解密的步骤，直接完成从密文到函数结果的转化，将先解密在用明文输入的过程合二为一，同样为了防止中间过程的明文发生泄露，这个过程被称作函数密钥。不同的加密系

统与不同的实际应用函数的组合不计其数。因而与代理重加密类似，函数加密中最大的难题就是这个函数密钥的生成，同时这一步也往往复杂度较高，影响系统效率。

前文已经说过，SGX 技术中的围圈的安全边界只包含 CPU 和它自身。也就是说，在我们信任 Intel 公司的前提下，SGX 技术生成的围圈可以视为一个接近绝对安全的环境。Ben A Fisch 与 Dhinakaran Vinayagamurthy 等人即是利用这一点，将解密和运行函数的过程通通放到围圈里面，分别将之命名为解密围圈（Decryption Enclaves）和函数围圈（Function Enclaves）。这样先解密在运行函数的运行效率要比生成函数密钥高很多，而安全性就由 SGX 技术来保障。从而实现了一个基于 SGX 技术的函数加密系统。

本研究打算里用同样的方法来解决不同加密系统之间的代理重加密密钥生成的问题，因为学术界已经有了这样的前车之鉴，所以我认为用 SGX 技术实现新型代理重加密技术是完全可行的。

本研究面对的另一个问题是如何测试系统的安全性，目前绝大多数有关加密体系模型构建的论文都只是停留在理论与仿真，对其安全性的考虑也只停留在安全分析上。如何测试已实现的加密体系是目前我还没有想好解决策略的一个比较重要的问题。

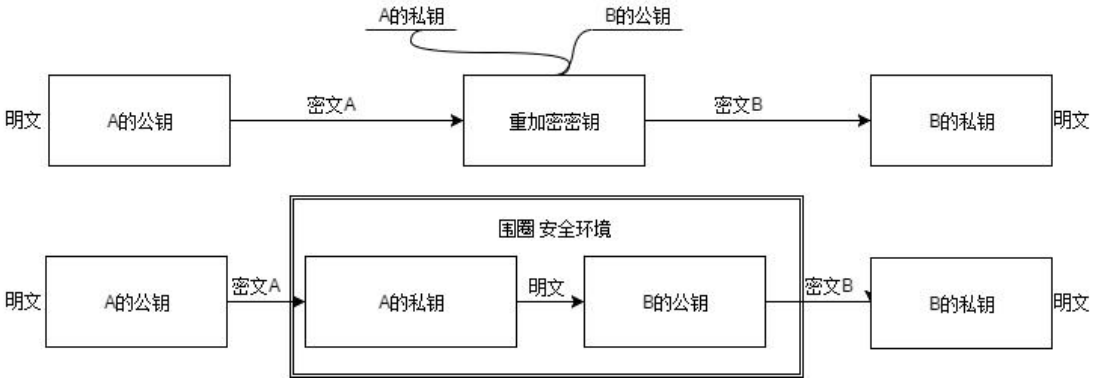

图 2 传统代理重加密与基于 SGX 的代理重加密

三、进度安排及预期目标

1. 进度安排

预计进度安排如下：

在三月初完成基于 SGX 技术的代理重加密系统的仿真；

在三月完成该系统在支持 SGX 的硬件系统上的具体实现，并进行调试与优化；

四月进行与其他系统之间的横向比较，并撰写小论文；

五月完成毕业论文的撰写和结题答辩。

2. 预期目标

预期目标为，实现一个利用 SGX 技术实现代理重加密功能的系统，该系统在实现不同加密系统间的代理重加密时，可以在满足安全要求的同时，在较高的性能下完成不同密文之间的转换。

参考文献

[1] Hoekstra.M. Intel SGX for Dummies. https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx 2013.9.

[2] 王进文, 江勇，李琦等. SGX 技术应用研究综述. 网络新媒体技术. 2017.9

[3]M.Blaze, G.Bleumer, M.Strauss． Divertible Protocols and Atomic Proxy Cryptography. In advances in Cryptology—Eurocrypt'98，1998

[4]周德华. 代理重加密体制的研究[学位论文] 上海交通大学计算机体系结构专业 2013.5

[5] Fisch.B, Vinayagamurthy.D, Boneh.D . Iron: Functional Encryption using Intel SGX. ACM CCS 2017. 2017

[6]Boneh D, Sahai A and Waters. B., Functional Encryption: Definitions and Challenges. Theory of Cryptography Conference[C]. 2011.

[7]其他参考文献：http://www.isee.zju.edu.cn/fanzhang/site_en/research/area03.html

文献翻译

Iron：基于 Intel SGX 的函数加密系统

Ben A Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, Sergey Gorbunov

斯坦福大学，滑铁卢大学

benafisch@gmail.com, dvinayag@uwaterloo.ca, dabo@cs.stanford.edu,

sgorbunov@uwaterloo.ca

摘要：函数加密(FE)是一种非常强大的加密机制，可以让已被授权的用户对加密数据进行计算，并且清楚地得到结果。然而，当前所有的常规 FE 的加密实例可行性都比较差。我们使用 Intel 最近的软件防护扩展(SGX)构建了一个实际可用的 FE 系统--Iron。我们证明，Iron 可以应用于复杂的功能，甚至是实现简单功能时，表现也优于最知名的的加密方案。我们通过在硬件元件的环境中对 FE 建模来论证其安全性，并证明 Iron 满足安全模型。

一、背景介绍

函数加密(FE)是一种强大的加密工具，它可以为加密的数据提供非交互式的细粒度访问控制[BSW12]。一个拥有主密钥 msk 的可信授权机构可以生成多个特殊的函数密钥，其中每个函数密钥 sk_f 与一个对明文数据操作的函数 f(或程序)相关联。当密钥 sk_f 用于解密消息 m 的加密密文 ct 时，其结果写作 f(m)。过程中没有泄露关于明文 m 的其他信息。多输入函数加密(MIFE)[GGG + 14] 是 FE 的一个扩展，其函数秘钥 sk_g 与一个以 l>1 个明文为输入的函数 g 相关联。在输入 $D(sk_g, c_1, \ldots, c_l)$上调用解密算法 D，此时密文序号 i 是消息 m_i 的加密，算法输出是 $g(m_1, \ldots, m_l)$。 同样，过程中没有泄露关于明文 $m_1, \ldots, m_l$ 的其他信息。在单个和多输入设置中，可以根据输入选择确定性的或随机性的函数[GJKS15,GGG + 14]。

如果 FE 和 MIFE 可实现，它们将有许多实际应用。例如，考虑一个从个体收集用公钥加密的基因组的遗传学研究人员，他可以向一个授权机构提出申请，例如国家卫生研究院(NIH)，要求对这些基因组进行特定的分析。如果获得批准，

研究人员将得到一个函数密钥 sk_f 。函数 f 可以实现所需的分析算法，使用 sk_f ，研究人员可以对加密的基因组进行分析，并清楚地得到结果，但得不到其他有关基础数据的信息。

类似地，一个存储重要加密数据的云可以被赋予一个函数密钥 sk_f ，其中函数 f 的输出是对数据应用数据挖掘算法的结果。云可以使用 sk_f，对加密的数据运行算法，清楚地得到结果，但得不到其他信息。数据的主人持有主密钥，并决定向云提供哪些函数密钥。

银行也可以使用 FE/MIFE 来提高客户的隐私和安全性，客户交易采用端到端加密，并通过函数解密运行所有事务审计。银行只会收到必要审计所需的密钥。

问题是当前为复杂功能设计的 FE 结构可行性差[GGH + 13]。它们基于目前还无法实现的程序混淆。

我们的贡献：我们提出使用英特尔的软件防护扩展(SGX)实现 FE/MIFE 的一个可行方案。Intel SGX 为孤立程序执行环境提供的硬件支持称为围圈。围圈也可以认证一个远程的组织，它在一个孤立的环境中运行一个特定的程序，甚至将由该程序执行的计算的输入和输出包含在它的远程认证书中。我们设计 的由 SGX 作为辅助的 FE/MIFE 系统称为“Iron”，它可以在处理器最快运行速度下运行加密数据的函数。Iron 的安全性依赖于对英特尔制造过程的信任和 SGX 系统的鲁棒性。此外，这项工作的主要成果不仅是提出用 SGX 构建 FE/MIFE，还可以使我们的信任假设形式化，并在此正式模型中提供严密的安全证明。

第 3 节详细描述了 Iron 的设计。在高层次上，系统使用一个密钥管理器围圈(KME)，它受信任且持有主密钥。它建立了一个标准的公钥加密系统和签名方案。任何人都可以使用 KME 发布的公钥加密数据。当一个客户(例如，研究人员)希望用数据运行一个特定的函数 f 时，他请求 KME 的授权。如果获得批准，KME 将生成一个函数密钥 sk_f，它在 f 用 ECDSA 签名的形式获得的。然后，为了解密，客户在 Intel SGX 平台上运行一个解密围圈(DE)。利用远程认证，DE 可以通过一个安全通道从 KME 处获得解密密钥去解密密文。然后，研究人员将 sk_f 以及被操作的密文加载到 DE。DE 在接收 sk_f 和密文时，检查 f 上的签名，解密给定的密文，并将函数 f 应用于明文的结果作为输出。然后这块围圈

将它的所有记忆清空。

在实现此方法时，会出现一些微妙之处。首先，SGX 的具体细节使得我们很难像上面描述的那样用一个可以解释任何函数 f 的 DE 来构建系统。我们在第 3 节通过引入第三个围圈来解决这个问题。其次，KME 需要机制来确保 DE 拥有正确的签名认证密钥，然后才发送解密密钥。围圈不能简单地访问公共信息，因为所有 I/O 通道都由一个可能不受信任的主机控制。最简单的想法似乎是令 KME 把认证密钥和自己安全信息中的解密密钥发送给 DE。然而事实证明，为了正式地证明信息从 KME 传递到 DE 的安全性，需要用认证密钥去签名和认证。我们可以将 KME 生成的认证密钥静态编码到 DE 中，但这将使 KME 对 DE 的远程认证的验证变得更加复杂。将一种固定认证的、属于 KME（授权代理）的公共密钥硬件编码到 DE 上似乎是一个简单的解决方案，但它需要一个辅助的公钥基础设施(PKI)和密钥管理机制，这就排除了 KME 的大部分作用。最好的选择是定义这样的 DE：认证密钥在本地加载，并将其合并到远程认证中作为程序的输入。最后，对于几种已知的针对 SGX 的侧向攻击，我们讨论了 Iron 如何防御它们。

我们实现了 Iron，并总结了它在许多功能方面的表现。对于复杂的功能，这一实现方案远远优于任何对 FE(不依赖硬件假设)的加密实现。我们将在第 5 节中证明，即使是简单的函数，例如比较和小型逻辑电路，我们的实现方案比最好的加密方案要高出 1 万倍以上。

在第 7 节，我们讨论了此结构的安全性。为此，我们在一个类 SGX 系统中给出了一个具体的 FE/MIFE 安全模型，并证明 Iron 满足此模型。

Figure 1: Application of Multi Input Functional Encryption to IoT cloud security.

三、系统设计

3.1 概述

平台　系统由一个可信的授权平台和任意多的可以动态添加的解密节点平台组成。可信授权平台和解密节点平台都应用 Intel SGX 技术。就像在标准的 FE 系统中一样，这个受信任的授权平台有设置公共参数，分发函数密钥和分发解密密文函数所需的认证的作用。一个不需要 Intel SGX 启用平台上运行的客户端应用程序将与授权平台进行交互，以获得对某个函数的授权，然后与解密节点平台进行交互，以执行对密文的函数解密。

协议流 授权平台生成的公共参数包括，一个公钥加密系统的加密公钥，以及用于加密签名方案的认证公钥。密文使用加密公钥加密。授权平台交给客户端应用程序的函数密钥是在函数描述上的签名。利用远程认证，授权平台将解密密钥交给在解密节点上的一个特殊的围圈。当客户端应用程序向解密节点发送密文、函数描述和有效签名时，带有认证密钥的围圈将检查签名，解密密文，在明文上运行函数，并输出结果。围圈将中止无效的签名的操作。

函数解释　最简单的设计是在解密节点上有一个解密围圈，它获得解密密钥，检查函数签名，并执行函数解密。然而，这需要解释围圈内部的函数逻辑描述。由于设计原因，本地代码在初始化后不能转移到 SGX 围圈。这对于简单的函数来说是合理的，但是对于更复杂的函数来说可能会有很大的影响。此外，实现合适的通用解释器也是个的挑战，它需要对旁路攻击由较强的鲁棒性，并且不会由它的访问模式向外部内存泄漏敏感信息。

函数围圈　我们在工作中采用的另一种设计利用了本地认证，避免了需要在一个围圈中运行一个解释器的问题，本地认证已经为围圈提供了验证代码在另一个围圈运行的方法。函数代码被加载到同一个平台上另一个已本地认证到解密围圈的函数围圈。授权平台不再生成函数描述，而是生成“此函数围圈将在本地认证中产生”的报告。这个设计的权衡之处在于，每个授权函数都在一个单独的围圈中运行。这对在大量密文中运行少量函数的应用程序几乎没有影响，但是对同一密文运行大量解密函数的应用程序必须为每个计算创建一个新的围圈，操作相对昂贵。事实上，我们在我们的评估(第 5 节)中演示对于一个简单的函数，比如基于身份的加密，在围圈上解释函数(如身份匹配)快了一个数量级。

3.2 结构

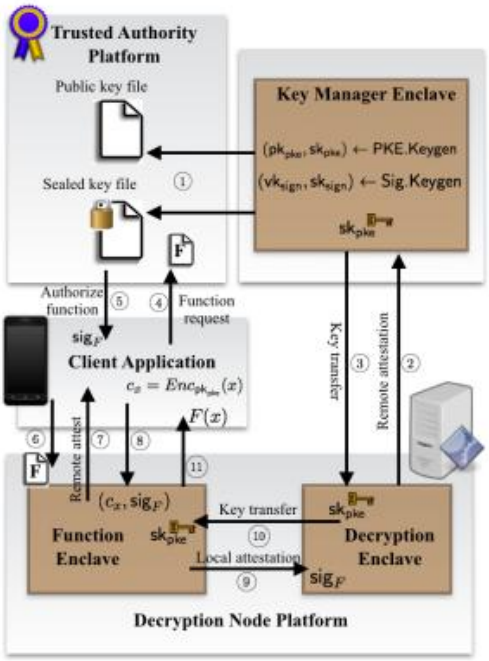

Figure 2: IRON Architecture and Protocol Flow

3.2.1 受信授权机构

授权平台运行一个名为密钥管理围圈(KME)的安全围圈，并有三个主要的协议:设置、函数授权和提供解密密钥。

设置 KME 生成一对用于 CCA2 安全的公钥密码系统公私密钥对(pk_{pke},sk_{pke})，和一对密码签名方案的认证/签名密钥对(vk_{sign},sk_{sign})。公钥 pk_{pke} 和 vk_{sign} 对外公布，私钥 sk_{pke} 和 sk_{sign} 用 KME 的密封密钥密封，并保存在非易失性存储中。

函数授权 为了授权客户端应用程序对一个特定的函数 f 执行函数解密，即发出 f 的“密钥”sk_f，授权机构给客户端应用程序提供一个 f 的签名，并使用签名密钥 sk_{sign}。由于只有 KME 知道 sk_{sign}，所以授权机构用 KME 来生成这个签名。函数 f 将被表示为一个被称为“函数围圈”的围圈程序，之后我们将更详细地描述这个程序。授权机构在这个围圈的报告中标识了 MRENCLAVE 的值，这个围圈由 EREPORT 指令创建，MRENCLAVE 的值标识了在初始化时加载到围圈的代码和静态数据。重要的是，这个值将与在任何其他支持 sgx 的平台上运行的同一个函数围圈程序实例生成的值相同。

提供解密密钥 当一个新的解密节点被初始化时，KME 将建立一个安全通道，该通道另一端连接在一个解密围圈(DE)上，该解密围圈是在解密节点 sgx 平台上运行的。KME 从解密节点接收到一个远程认证，这表明解密节点运行的是预

期的 DE 软件程序，而 DE 拥有正确的签名验证密钥 vk_{sign}。 远程认证也建立了一个安全通道，即包含在 DE 内部生成的公钥，验证远程认证后，KME 通过已建立的安全通道将 sk_{pke} 发送到 DE，并通过 sk_{sign} 对该消息进行验证。

验证 KME 的信息 为什么 KME 需要在交给 DE 的消息上签名这一点并不明显，因为 sk_{pke} 是加密的，似乎并没有中间人攻击可以影响其安全性。如果从 KME 到 DE 的消息被替换，那么显然解密节点平台将无法解密在 pk_{pke} 下加密的密文。然而事实证明，验证 KME 的消息对于我们正式的安全工作证明来讲是必要的(参见第 7 节)。

为什么要在围圈运行密钥管理机构? 由于授权机构已受信任并可以授权函数，有人可能会想，为什么我们选择用一个独立的围圈生成和管理密钥，而不是授权机构本身？对授权机构隐瞒这些密钥并不会减少任何信任假设，因为授权机构可以运行 KME 来标记其选择的任何函数，因此可以授权自己对任何密文进行解密。在另一个围圈上运行 KME 的原因是要分离那些本质上与授权机构相关联的协议和其他协议。特别是由于 KME 可以在一个完全独立的不可信平台上运行，所以为解密节点提供解密密钥 sk_{pke} 的协议根本不需要涉及授权机构。这个分离很重要。在函数加密方案的标准概念中，解密不需要与授权机构的交互。虽然授权机构受信任，生成公共参数并分发函数密钥，但它不能实现例如突然决定阻止某个特定客户使用已经收到的解密密钥的操作。此外，在围圈内管理密钥可以更好地存储密钥(例如 HSM 的功能)。

3.2 解密节点

解密节点运行解密围圈(DE)的单独实例。它还接收客户端运行函数围圈的请求。如果得到授权，函数围圈将解密密文，对解密的明文运行特定的函数，并用客户端的密钥加密输出结果。对于远程客户端，这个函数围圈也可以生成一个远程认证来证明输出的完整性。

解密围圈(DE) 当 DE 被初始化时，它接收认证公钥 vk_{sign}，与 KME 进行远程认证，认证书中包含 vk_{sign}。认证书也建立了一个安全通道(认证书包含 DE 生成的公钥)，而 DE 将通过这个安全通道接收解密密钥 sk_{pke}。DE 的作用是将解密密钥转移到在同一个平台的围圈运行的授权程序，我们称之为函数围圈。DE

将通过本地认证验证在函数围圈内运行的代码。具体来说，DE 接收到有关此函数围圈的 MRENCLAVE 值的 KME 签名，用 vk_{sign} 进行验证，并对函数围圈本地认证书中的 MRENCLAVE 值进行检查，以确保受信授权机构已经授权给了此函数围圈。本地认证书还建立了从 DE 到函数围圈的安全通道(包含函数围圈内部生成的公钥)。如果本地认证和签名通过，则 DE 将 sk_{pke} 通过该安全通道转移到函数围圈。DE 还通过将信息包装在本地认证书中的方式来验证其交给函数围圈的信息。

函数围圈 最终，对于特定函数 f 的函数解密在函数围圈内执行。函数围圈在初始化时加载函数。已被授权解密 f 的客户端应用程序可以在本地或远程操作该函数围圈。如果应用程序正在解密节点上本地运行，则其可以直接调用该函数围圈，输入密文的向量和签名。远程客户端应用程序需要通过远程认证建立安全通道。有效的密文输入必须是由密钥 pk_{pke} 加密的密文，而有效的签名输入必须是这个函数围圈的 MRENCLAVE 值用 KME 的签名密钥 sk_{sign} 生成的。接收到输入后，函数围圈与 DE 本地认证，认证包含签名输入。如果签名输入是有效的，则该函数围圈将通过本地认证建立的安全通道接收 sk_{pke}。 它接收到的消息必须通过来自 DE 的本地认证书进行验证。然后函数围圈使用 sk_{pke} 对密文进行解密，将解密的明文向量作为输入运行客户定义的函数，并记录输出。对于本地客户机应用，输出直接返回到应用。对于远程客户端应用，围圈通过与客户端建立的会话密钥对输出进行加密。

3.3 协议

我们在这里提供一个非正式的总结，介绍了在 3.2 节中描述的 Iron 系统如何实现四个函数加密协议 FE.Setup, FE.Encrypt, FE.Keygen 和 FE.Decrypt。我们将在第 7 节和第 4 节的实现细节级别重新正式描述这些协议(为了安全证明)。

FE.Setup 可信平台如 3.2 所述运行 KME 设置，并发布公钥 pk_{pke} 和认证密钥 vk_{sign}。调用 KME 的签名函数的操作使用 sk_{sign} 生成签名，充当受信任授权机构的主密钥。

FE.Keygen 授权机构从客户端接收到授权函数 f 的请求，请求的函数被包装在一个函数围圈源文件 $enclave_f$ 中。可信平台编译围圈源文件，并为围圈生成认

证书，包括 MRENCLAVE 的值 MRENCLAVE_f。然后使用 KME 签名操作用 sk_{sign} 给 mrenclave_f 签名。签名 sig_f 返回给客户端。

FE.Encrypt 加密过程输入采用 CCA2 公钥加密方案，密钥为 pk_{pke}。

解密过程开始时，客户端应用程序连接到一个解密节点，如 3.2 所述，该节点已经被分配使用解密密钥 sk_{pke}。客户端应用程序也可以在解密节点上本地运行。之后的步骤如下:

如果这是客户端第一个对函数 f 进行解密的请求，则客户端将函数围圈源文件 enclave_f 发送到解密节点，解密节点随后编译并运行。(本地客户机应用程序只运行 enclave_f)。

客户端启动与这个函数围圈的密钥交换，并接收一个远程认证，该认证说明已成功建立与安全通道运行 enclave_f 的 Intel SGX 围圈的通道。(本地客户机应用程序跳过这一步)。

客户端通过已建立的安全通道发送密文矢量和它从 FE.Keygen 获得的 KME 签名 sig_f。

该函数围圈在本地对 DE 进行测试，并传递 sig_f。DE 检验了该签名与 vk_{sign} 和 MRENCLAVE 值 mrenclave_f 是否对应，mrenclave_f 本地认证中得到。如果验证通过，DE 将密钥 sk_{pke} 交付给该函数围圈，它使用 sk_{pke} 来解密密文并对明文计算 f。输出将通过函数围圈和客户端之间的安全通道返回给客户端应用程序。

附：原稿

IRON: Functional Encryption using Intel SGX

Ben A Fisch*[1], Dhinakaran Vinayagamurthy †[2], Dan Boneh ‡[1], and Sergey Gorbunov §[2]

[1]Stanford University
[2]University of Waterloo

Abstract

Functional encryption (FE) is an extremely powerful cryptographic mechanism that lets an authorized entity compute on encrypted data, and learn the results in the clear. However, all current cryptographic instantiations for general FE are too impractical to be implemented. We build IRON, a practical and usable FE system using Intel's recent Software Guard Extensions (SGX). We show that IRON can be applied to complex functionalities, and even for simple functions, outperforms the best known cryptographic schemes. We argue security by modeling FE in the context of hardware elements, and prove that IRON satisfies the security model.

1 Introduction

Functional Encryption (FE) is a powerful cryptographic tool that facilitates non-interactive fine-grained access control to encrypted data [BSW12]. A trusted authority holding a master secret key msk can generate special functional secret keys, where each functional key sk_f is associated with a function f (or program) on plaintext data. When the key sk_f is used to decrypt a ciphertext ct, which is the encryption of some message m, the result is the quantity $f(m)$. Nothing else about m is revealed. Multi-Input Functional Encryption (MIFE) [GGG+14] is an extension of FE, where the functional secret key sk_g is associated with a function g that takes $\ell \geq 1$ plaintext inputs. When invoking the decryption algorithm D on inputs $D(\mathsf{sk}_g, c_1, \ldots, c_\ell)$, where ciphertext number i is an encryption of message m_i, the algorithm outputs $g(m_1, \ldots, m_\ell)$. Again, nothing else is revealed about the plaintext data $m_1, \ldots, m_\ell$. Functions can be deterministic or randomized with respect to the input in both single and multi-input settings [GJKS15, GGG+14].

If FE and MIFE could be made practical, they would have numerous real-world applications. For example, consider a genetics researcher who collects public-key encrypted genomes from individuals. The researcher could then apply to an authority, such as the National Institutes of Health (NIH), and request to run a particular analysis on these genomes. If approved, the researcher is given a functional key sk_f, where the function f implements the desired analysis algorithm. Using sk_f the researcher can then run the analysis on the encrypted genomes, and learn the results in the clear, but without learning anything else about the underlying data.

Similarly, a cloud storing encrypted sensitive data can be given a functional key sk_f, where the output of the function f is the result of a data-mining algorithm applied to the data. Using sk_f the cloud can run the algorithm on the encrypted data, to learn the results in the clear, but without learning anything else.

*Email: benafisch@gmail.com
†Email: dvinayag@uwaterloo.ca
‡Email: dabo@cs.stanford.edu
§Email: sgorbunov@uwaterloo.ca

The data owner holds the master key, and decides what functional keys to give to the cloud.

Banks could also use FE/MIFE to improve privacy and security for their clients by allowing client transactions to be end-to-end encrypted, and running all transaction auditing via functional decryption. The bank would only receive the keys for the necessary audits.

The problem is that current FE constructions for complex functionalities cannot be used in practice [GGH+13]. They rely on *program obfuscation* which currently cannot be implemented.

Our contribution. We propose a practical implementation of FE/MIFE using Intel's Software Guard Extensions (SGX). Intel SGX provides hardware support for isolated program execution environments called *enclaves*. Enclaves can also *attest* to a remote party that it is running a particular program in an isolated environment, and even include in its *remote attestation* the inputs and outputs of computations performed by that program. Our SGX-assisted FE/MIFE system, called Iron, can run functionalities on encrypted data at full processor speeds. The security of Iron relies on trust in Intel's manufacturing process and the robustness of the SGX system. Additionally, a major achievement of this work was not only to propose a construction of FE/MIFE using SGX, but also to formalize our trust assumptions and supply rigorous proofs of security inside this formal model.

The design of Iron is described in detail in Section 3. At a high level, the system uses a *Key Manager Enclave* (KME) that plays the role of the trusted authority who holds the master key. This authority sets up a standard public key encryption system and signature scheme. Anyone can encrypt data using the KME's published public key. When a client (e.g., researcher) wishes to run a particular function f on the data, he requests authorization from the KME. If approved, the KME releases a functional secret key sk_f that takes the form of an ECDSA signature on the code of f. Then, to perform the decryption, the client runs a *Decryption Enclave* (DE) running on an Intel SGX platform. Leveraging remote attestation, the DE can obtain over a secure channel the secret decryption key from the KME to decrypt ciphertexts. The researcher then loads sk_f into the DE, as well as the ciphertext to be operated on. The DE, upon receiving sk_f and a ciphertext, checks the signature on f, decrypts the given ciphertext, and outputs the function f applied to the plaintext. The enclave then erases all of its state from memory.

Several subtleties arise when implementing this approach. First, the specifics of SGX make it difficult to build the system as described above with a single DE that can interpret any function f. We overcome this complication by involving a third enclave as explained in Section 3. Second, we need a mechanism for the KME to ensure that the DE has the correct signature verification key before sending it the secret decryption key. Enclaves cannot simply access public information because all I/O channels are controlled by a potentially untrusted host. The simplest idea, it seems, is to have the KME send the verification key along with the secret decryption key in its secure message to the DE. However, it turns out that in order to formally prove security the message from KME to the DE also needs to be signed and verified with this verification key. We could statically code the verification key generated by the KME into the DE, but this would complicate the KME's verification of the DE's remote attestation. Hardcoding a fixed certified public key belonging to the KME/authority into the DE may appear to be a simple solution, but it requires an auxiliary Public Key Infrastructure (PKI) and key management mechanism, which obviates much of the KME's role. The best option is to define the DE such that the verification key is locally loaded and incorporated into the remote attestation as a program input. Finally, there are several known side-channel attacks on SGX, and we discuss how Iron can defend against them.

We implemented Iron and report on its performance for a number of functionalities. For complex functionalities, this implementation is far superior to any cryptographic implementation of FE (which does not rely on hardware assumptions). We show in Section 5 that even for simple functionalities, such as comparison and small logical circuits, our implementation outperforms the best cryptographic schemes by over a 10,000

fold improvement.

In Section 7 we argue security of this construction. To do so we give a detailed model of FE/MIFE security in the context of an SGX-like system, and prove that IRON satisfies the model.

SGX limitations. It is important to be wary of the limitations of basing security on trust in SGX, and shy away from viewing it as a "magic box". Unsurprisingly, several side-channel attacks have come to light since SGX's initial release. While we discuss known side-channel attacks and how to defend IRON against them, we acknowledge that new side-channel attacks may yet be discovered in the future. Moreover, components of the Intel SGX implementation are proprietary, making it difficult for security researchers to fully assess its security. There has been academic effort to develop open-source SGX-like systems that achieve the same strong software isolation (see Sanctum, [CLD16]) and that can be fully examined by the entire community of security researchers.

On the other hand, Intel SGX is widely available for use in commodity chips, whereas academic systems like Sanctum are not. Ideally, we would have the best of both worlds. Hardware software isolation is an evolving technology, and the research contributions of this work should be viewed as a paradigm for building functional encryption with both present and future generations of SGX-like systems.

Figure 1: Application of Multi Input Functional Encryption to IoT cloud security.

3 System Design

3.1 Overview

Platforms The IRON system consists of a single *trusted authority* (Authority) platform and arbitrarily many *decryption node* platforms, which may be added dynamically. Both the trusted authority and decryption node platforms are Intel SGX enabled. Just as in a standard FE system, the trusted authority has the role of setting up public parameters as well as distributing *functional secret keys*, or the credentials required to decrypt functions of ciphertexts. A *client application*, which does not need to run on an Intel SGX enabled platform, will interact once with the trusted authority in order to obtain authorization for a function and will then interact with a decryption node in order to perform functional decryptions of ciphertexts.

Protocol flow The public parameters that the Authority platform generates will consists of a public encryption key for a public key cryptosystem and a public verification key for a cryptographic signature scheme. Ciphertexts are encrypted using the public encryption key. The functional secret keys that the Authority platform issues to client applications are signatures on function descriptions. Leveraging remote attestation, the Authority platform provisions the secret decryption key to a special enclave on the decryption node. When a client application sends a ciphertext, function description, and valid signature to the decryption node, an enclave with access to the secret key will check the signature, decrypt the ciphertext, run the function on the plaintext, and output the result. The enclave will abort on invalid signatures.

Function interpretation The simplest design is to have a single *decryption enclave* on the decryption node obtain the secret decryption key, check function signatures, and perform functional decryption. However, this would require interpreting a logical description of the function inside the enclave. By design, native code cannot be moved into an SGX enclave after initialization.[1] This is reasonable for simple functions, but could greatly impact performance for more complex functions. Moreover, it is an additional challenge to implement a general purpose interpreter that will be robust to side-channel attacks and will not leak sensitive information through its access pattern to external memory.

Function enclaves An alternative design, which we implement in this work, circumvents the need for running an interpreter inside an enclave by taking advantage of *local attestation*, which already provides a way for one enclave to verify the code running in another. The function code is loaded into a separate *function enclave* on the same platform that locally attests to the decryption enclave. Instead of signing the description of a function, the Authority platform signs the report that the function enclave will generate in local attestation. A tradeoff of this design is that every authorized function runs in a separate enclave. This has little impact on applications that run a few functions on many ciphertexts. However, a client application that decrypts many functions of a ciphertext will have to create a new enclave for each computation, which is a relatively expensive operation. In fact, we demonstrate in our evaluation (Section 5) that for a simple functionality like Identity Based Encryption (IBE) interpreting the function (i.e. identity match) in an enclave is an order of magnitude faster.

3.2 Architecture

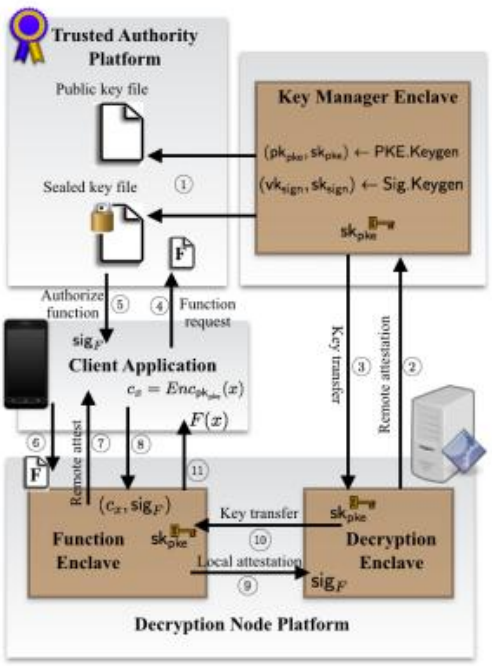

Figure 2: IRON Architecture and Protocol Flow

3.2.1 Trusted authority

The Authority platform runs a secure enclave called the *key manager enclave* (KME) and has three main protocols: *setup*, *function authorization*, and *decryption key provisioning*.

Setup The KME generates a public/private key pair $(\mathsf{pk}_{\mathsf{pke}}, \mathsf{sk}_{\mathsf{pke}})$ for a CCA2 secure public key cryptosystem and a verification/signing key pair $(\mathsf{vk}_{\mathsf{sign}}, \mathsf{sk}_{\mathsf{sign}})$ for a cryptographic signature scheme. The keys $\mathsf{pk}_{\mathsf{pke}}$ and $\mathsf{vk}_{\mathsf{sign}}$ are published while the keys $\mathsf{sk}_{\mathsf{pke}}$ and $\mathsf{sk}_{\mathsf{sign}}$ are sealed with the KME's sealing key and kept in non-volatile storage.

Function authorization In order to authorize a client application to perform functional decryption for a particular function f, i.e. issue the "secret key" sk_f for f, the Authority provides the client application with a signature on f using the signing key $\mathsf{sk}_{\mathsf{sign}}$. Since $\mathsf{sk}_{\mathsf{sign}}$ is only known to the KME, the Authority uses the KME to produce this signature. The function f will be represented as an enclave program called a *function enclave*, which we will describe in more detail. The Authority signs the MRENCLAVE value in the report of this enclave (created by the EREPORT instruction), which identifies the code and static data that was loaded into the enclave upon initialization. Crucially, this value will be the same when generated by an instance of the same function enclave program running on any other SGX-enabled platform.

Decryption key provisioning When a new decryption node is initialized, the KME will establish a secure channel with a *decryption enclave* (DE) running on the decryption node SGX-enabled platform. The KME receives from the decryption node a *remote attestation*, which demonstrates that the decryption node

is running the expected DE software and that the DE has the correct signature verification key $\mathsf{vk}_{\mathsf{sign}}$. The remote attestation also establishes a secure channel, i.e. contains a public key generated inside the DE. After verifying the remote attestation, the KME sends $\mathsf{sk}_{\mathsf{pke}}$ to the DE over the established secure channel, and authenticates this message by signing it with $\mathsf{sk}_{\mathsf{sign}}$.

Authenticating KME's message At this point, it is not at all obvious why the KME needs to sign its message to the DE. Indeed, since $\mathsf{sk}_{\mathsf{pke}}$ is encrypted, it seems that there isn't anything a man-in-the-middle attacker could do to harm security. If the message from the KME to the DE is replaced, the decryption node platform will simply fail to decrypt ciphertexts encrypted under $\mathsf{pk}_{\mathsf{pke}}$. However, it turns out that authenticating the KME's messages is necessary for our formal proof of security to work (see Section 7).

Why run the key manager in an enclave? Since the Authority is already trusted to authorize functions, one might wonder why we chose to have an enclave generate and manage keys rather than the Authority itself. Hiding these secret keys from the Authority does not reduce any trust assumptions since the Authority can use the KME to sign any function of its choice and therefore authorize itself to decrypt any ciphertext. The reason for running the KME in an enclave is to create a separation between the protocols that inherently involve the Authority and those that do not. In particular, since the KME could be run on an entirely separate untrusted platform, the protocol that provisions the decryption key $\mathsf{sk}_{\mathsf{pke}}$ to decryption nodes does not need to involve the Authority at all. This is an important separation. In the standard notion of a functional encryption scheme, decryption does not require interaction with the Authority. While the Authority is trusted to generate public parameters and to distribute functional secret keys, it could not, for example, suddenly decide to prevent a particular client from using a decryption key that is has already received. Additionally, managing keys inside an enclave offers better storage protection of keys (i.e. functions as an HSM).

3.2.2 Decryption node

A decryption node runs a single instance of the *decryption enclave* (DE). It will also receive requests from clients to run *function enclaves*. If properly authorized, a function enclave will be able to decrypt ciphertexts, run a particular function on the decrypted plaintext, and output the result encrypted under the client's key. For remote clients, the function enclave can also produce a remote attestation to demonstrate the integrity of the output.

Decryption Enclave When the DE is initialized it is given the public verification key $\mathsf{vk}_{\mathsf{sign}}$. It remote attests to the KME, and includes $\mathsf{vk}_{\mathsf{sign}}$ in the attestation. A secure channel is also established within the attestation (i.e. the attestation contains a public key generated inside the DE), and the DE receives back the decryption key $\mathsf{sk}_{\mathsf{pke}}$ over this secure channel. The DE has the role of transferring the secret decryption key to authorized programs running within enclaves on the same platform, which we refer to as *function enclaves*. The DE will verify the code running inside a function enclave via local attestation. Specifically, it receives a KME signature on the function enclave's MRENCLAVE value, which it will verify using $\mathsf{vk}_{\mathsf{sign}}$, and check this against the MRENCLAVE value in the function enclave's local attestation report. This ensures that the trusted authority has authorized the function enclave. The local attestation also establishes a secure channel from the DE to the function enclave (i.e. contains a public key generated inside the function enclave). If the verifications of the local attestation and the signature pass, then the DE transfers $\mathsf{sk}_{\mathsf{pke}}$ to the function enclave over the secure channel. The DE also authenticates its message to the function enclave by wrapping it inside its own local attestation report.[2]

Function Enclaves Ultimately, functional decryption for a particular function f is performed inside a function enclave that loads the function upon initialization. A client application authorized to decrypt f can operate the function enclave either locally or remotely. If the application is running locally on the decryption

node, then it can directly call into the function enclave, providing as input a vector of ciphertexts and a signature. A remote client application will need to establish a secure channel with the enclave via remote attestation. A valid ciphertext input must be an encryption under the key $\mathsf{pk}_{\mathsf{pke}}$ and a valid signature input must be a signature on the function enclave's MRENCLAVE value produced with $\mathsf{sk}_{\mathsf{sign}}$, the KME's signing key. After receiving the inputs, the function enclave local attests to the DE and includes the signature input. If the signature input was valid, the function enclave will receive back $\mathsf{sk}_{\mathsf{pke}}$ over a secure channel established in the local attestation. The message it receives back will be authenticated with a local attestation report from the DE, which it must verify. It then uses $\mathsf{sk}_{\mathsf{pke}}$ to decrypt the ciphertexts, passes the vector of decrypted plaintexts as input to the client-defined function and records the output. In the case of a local client application, the output is returned directly to the application. In the case of a remote client application, the enclave encrypts the output with the session key it established with the client.

3.3 Protocols

Here we provide an informal summary of how the IRON system, as described above in 3.2, realizes each of the four functional encryption protocols FE.Setup, FE.Encrypt, FE.Keygen and FE.Decrypt. These protocols will be redescribed formally (for the purpose of security proofs) in 7 and at the implementation detail level in 4.

FE.Setup The trusted platform runs the KME setup as described in 3.2 and publishes the public key $\mathsf{pk}_{\mathsf{pke}}$ and the verification key $\mathsf{vk}_{\mathsf{sign}}$. A handle to the KME's signing function call, which produces signatures using $\mathsf{sk}_{\mathsf{sign}}$, serves as the trusted authority's master secret key.

FE.Keygen The Authority receives a request from a client to authorize a function f. The requested function is wrapped in a function enclave source file $\mathsf{enclave}_f$. The trusted platform compiles the enclave source and generates an attestation report for the enclave including the MRENCLAVE value $\mathsf{mrenclave}_f$. It then uses the KME signing handle to sign $\mathsf{mrenclave}_f$ using $\mathsf{sk}_{\mathsf{sign}}$. The signature sig_f is returned to the client.

FE.Encrypt Inputs are encrypted with $\mathsf{pk}_{\mathsf{pke}}$ using a CCA2 secure public key encryption scheme.

FE.Decrypt Decryption begins with a client application connecting to a decryption node that has already been provisioned with the decryption key $\mathsf{sk}_{\mathsf{pke}}$ as described in 3.2. The client application may also run locally on the decryption node. The following steps ensue:

1. If this is the client's first request to decrypt the function f, the client sends the function enclave source file $\mathsf{enclave}_f$ to the decryption node, which the decryption node then compiles and runs. (A local client application would just run $\mathsf{enclave}_f$).
2. The client initiates a key exchange with the function enclave, and receives a remote attestation that it has successfully established a secure channel with an Intel SGX enclave running $\mathsf{enclave}_f$. (Local client applications skip this step).
3. The client sends over the established secure channel a vector of ciphertexts and the KME signature sig_f that it obtained from the Authority in FE.Keygen.
4. The function enclave locally attests to the DE and passes sig_f. The DE validates this signature against $\mathsf{vk}_{\mathsf{sign}}$ and the MRENCLAVE value $\mathsf{mrenclave}_f$, which it obtains during local attestation. If this validation passes, the DE delivers the secret key $\mathsf{sk}_{\mathsf{pke}}$ to the function enclave, which uses it to decrypt the ciphertexts and compute f on the plaintext values. The output is returned to the client application over the function enclave's secure channel with the client application.

毕业论文文献综述和开题报告考核

一、对文献综述、外文翻译和开题报告评语及成绩评定：

评语：

成绩：

开题报告答辩小组负责人（签名）