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。

1　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，. . . ，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，. . . ，m_l)。同样，过程中没有泄露关于明文m_1，. . . ，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飞地。飞地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飞地(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飞地(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。然后这块飞地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节通过引入第三个飞地来解决这个问题。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。飞地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。

3 系统设计

3.1 概述

平台　Iron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飞地。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，带有认证密钥的飞地将检查签名，解密密文，在明文上运行函数，并输出结果。飞地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飞地，它获得解密密钥，检查函数签名，并执行函数解密。然而，这需要解释飞地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飞地。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。

函数飞地　我们在工作中采用的另一种设计利用了本地认证，避免了需要在一个飞地中运行一个解释器的问题，本地认证已经为飞地提供了验证代码在另一个飞地运行的方法。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飞地的函数飞地。授权平台不再生成函数描述，而是生成“此函数飞地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飞地中运行。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飞地，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，在飞地上解释函数(如身份匹配)快了一个数量级。

3.2 结构

3.2.1 受信授权机构

授权平台运行一个名为密钥管理飞地(KME)的安全飞地，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“函数飞地”的飞地程序，之后我们将更详细地描述这个程序。授权机构在这个飞地的报告中标识了MRENCLAVE的值，这个飞地由EREPORT指令创建，MRENCLAVE的值标识了在初始化时加载到飞地的代码和静态数据。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飞地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飞地(DE)上，该解密飞地是在解密节点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)。

为什么要在飞地运行密钥管理机构?　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飞地生成和管理密钥，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。在另一个飞地上运行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，在飞地内管理密钥可以更好地存储密钥(例如HSM的功能)。

3.2.2　解密节点

解密节点运行解密飞地(DE)的单独实例。它还接收客户端运行函数飞地的请求。如果得到授权，函数飞地将解密密文，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，这个函数飞地也可以生成一个远程认证来证明输出的完整性。

解密飞地(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飞地运行的授权程序，我们称之为函数飞地。DE将通过本地认证验证在函数飞地内运行的代码。具体来说，DE接收到有关此函数飞地的MRENCLAVE值的KME签名，用vk_{sign} 进行验证，并对函数飞地本地认证书中的MRENCLAVE值进行检查，以确保受信授权机构已经授权给了此函数飞地。本地认证书还建立了从DE到函数飞地的安全通道(包含函数飞地内部生成的公钥)。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飞地。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飞地的信息。

函数飞地　最终，对于特定函数f的函数解密在函数飞地内执行。函数飞地在初始

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飞地。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飞地，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飞地的MRENCLAVE值用KME的签名密钥sk_{sign}生成的。接收到输入后，函数飞地与DE本地认证，认证包含签名输入。如果签名输入是有效的，则该函数飞地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。然后函数飞地使用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，飞地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飞地源文件$\mathrm{enclave}_f$中。可信平台编译飞地源文件，并为飞地生成认证书，包括MRENCLAVE的值$\mathrm{MRENCLAVE}_f$。然后使用KME签名操作用sk_{sign}给$\mathrm{mrenclave}_f$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，则客户端将函数飞地源文件$\mathrm{enclave}_f$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$\mathrm{enclave}_f$)。

客户端启动与这个函数飞地的密钥交换，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$\mathrm{enclave}_f$的Intel SGX飞地的通道。(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。

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

Iron: Functional Encryption using Intel SGX

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

Stanford University University of Waterloo
benafisch@gmail.com, dvinayag@uwaterloo.ca, dabo@cs.stanford.edu, sgorbunov@uwaterloo.ca

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 $l \geqslant 1$ plaintext inputs. When invoking the decryption algorithm D on inputs $\mathrm{D}(\mathrm{sk}_g, \mathrm{c}_1, \ldots, \mathrm{c}_l)$, where ciphertext number i is an encryption of message m_i, the algorithm outputs $\mathrm{g}(\mathrm{m}_1, \ldots, \mathrm{m}_l)$. Again, nothing else is revealed about the plaintext data $\mathrm{m}_1, \ldots, \mathrm{m}_l$. 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.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.

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

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 (pk_{pke},sk_{pke}) for a CCA2 secure public key cryptosystem and a verification/signing key pair (vk_{sign},sk_{sign}) for a cryptographic signature scheme. The keys pk_{pke} and vk_{sign} are published while the keys sk_{pke} and sk_{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 sk_{sign}. Since sk_{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 vk_{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 sk_{pke} to the DE over the established secure channel, and authenticates this message by signing it with sk_{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 sk_{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 pk_{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 sk_{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 vk_{sign}. It remote attests to the KME, and includes vk_{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 sk_{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 vk_{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 sk_{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.

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 pk_{pke} and a valid signature input must be a signature on the function enclave’ s MRENCLAVE value produced with sk_{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 sk_{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 sk_{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 协议

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 pk_{pke} and the verification key vk_{sign}. A handle to the KME’ s signing function call, which produces signatures using sk_{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 enclave_f. The trusted platform compiles the enclave source and generates an attestation report for the enclave including the MRENCLAVE value mrenclave_f. It then uses the KME signing handle to sign mrenclave_f using sk_{sign}. The signature sig_f is returned to the client.

FE.Encrypt Inputs are encrypted with pk_{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 sk_{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 enclave_f to the decryption node, which the decryption node then compiles and runs. (A local client application would just run 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 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 vk_{sign} and the MRENCLAVE value mrenclave_f, which it obtains during local attestation. If this validation passes, the DE delivers the secret key sk_{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.