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

2

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.
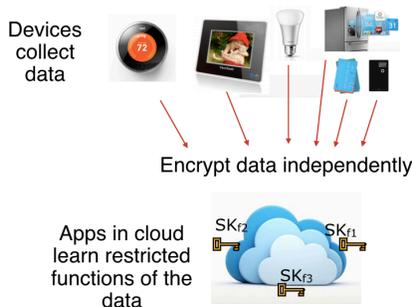

Figure 1: Application of Multi Input Functional Encryption to IoT cloud security.

2 Intel SGX Background

Intel Software Guard Extensions (SGX) [MAB+13] is a set of processor extensions to Intel's x86 design that allow for the creation of isolated execution environments called enclaves. These isolated execution environments are designed to run software and handle secrets in a trustworthy manner, even on a host where the OS and system memory are untrusted. The isolation of enclave resident applications from all other processes is enforced by hardware access controls. The SGX specifications are detailed and complex [SGX16, MAB+13]. We provide only a brief overview of its design and capabilities, with emphasis on the components relevant to our system.

There are three main functionalities that enclaves achieve: *Isolation*–code and data inside the enclave protected memory cannot be read/modified by any process external to the enclave. *Sealing*–data passed to the host environment is encrypted and authenticated with a hardware-resident key. And *Attestation*–a special signing key and instructions are used to provide an unforgeable report attesting to code, static data, and (hardware-specific) metadata of an enclave, as well as outputs of computations performed inside the enclave.

2.1 Isolation

Enclaves reside in a hardware guarded area of memory called the Enclave Page Cache (EPC). The EPC is currently limited to 128 MB, consisting of 4KB page chunks, and applications can use approximately 90 MB. When an enclave program is loaded, its code and static data are copied from untrusted memory to pages inside the EPC. A measurement of the contents of these pages called MRENCLAVE (essentially a SHA256 hash of the page contents) is also stored inside the EPC in a structure that is linked to the enclave. Entry into the enclave is not permitted throughout this process until the measurement has been finalized. The creation process establishes an enclave identity, which is used as a handle to transfer program control to the enclave. The hardware enforces that only the executable code pages associated with a particular enclave identity can access the other pages associated with that identity.

2.2 Sealing

Every SGX processor has a key called the Root Seal Key that is embedded during the manufacturing process. An enclave can use the EGETKEY instruction to derive a key called *Seal Key* from the Root Seal Key that is specific to the enclave identity, which can be used to encrypt/authenticate data and store it in untrusted memory. Sealed data can be recovered by the same enclave even after enclave is destroyed and restarted on the same platform. But the Seal key cannot be derived by a different enclave on the same platform or any enclave on a different platform.

2.3 Attestation

There are two forms of attestation: *local* and *remote*. We sketch here how each of these processes work and provide more details in Appendix B.

Local attestation   Local attestation is between two enclaves on the same platform. Roughly, since enclaves on the same machine share the same Root Seal Key, they can derive a shared key (called *Report Key*) for symmetric authentication. An enclave can call a special instruction EREPORT that fetches the MRENCLAVE and metadata of an enclave and MACs it with the Report Key (along with additional optional data provided as input to the instruction). This is called a *report*.

Remote attestation   Remote attestation generates a report that can be verified by any remote party. Roughly, to generate a quote an enclave first local attests to a special enclave called the Quoting Enclave, sending it a report. The Quoting Enclave will verify this report and convert it into a *quote*. The quote contains the same underlying data but is signed with a private key for an anonymous group signature scheme called Intel Enhanced Privacy ID (EPID) [JSR$^+$16]. Currently, verifying these signatures involves contacting the Intel Attestation Server, though in principle this could be done by any verifier that has the group public key.

2.4 SGX side-channel attacks

The security of SGX is still evolving [SGX] but the current version is susceptible to the following side-channel attacks. Side-channel attacks on SGX can be divided into two classes: *physical attacks*, which are mounted by an attacker with physical access to the CPU, and *software attacks*, which are mounted by software running on the same host as the CPU, such as a compromised OS. SGX does not claim to defend against physical attacks such as power analysis. For instance, this could enable a physical attacker to extract keys from an enclave and to spoof remote attestation. However, successful physical attacks against SGX have not yet been demonstrated.

Several software attacks have been demonstrated so far, including cache-timing attacks [CD16], page-fault attacks [XCP15], branch shadowing [LSG$^+$16] and synchronization bugs [WKPK16]. Cache-timing and page-fault attacks can reveal enclave memory access patterns at cache-line and 4KB page granularity respectively.

Branch shadowing can directly view the control flow branch history in an enclave. Leaking information through these side-channels can be avoided by ensuring that enclave programs are *data-oblivious*, i.e. do not have memory access patterns or control flow branches that depend on the values of sensitive data. *Our implementation of enclave programs that deal with sensitive information are data-oblivious.* Synchronization bugs only apply to multi-threaded code running inside enclaves, and various defense mechanisms are listed in [WKPK16].

We provide a more detailed explanation of side-channel attacks and their defenses in Appendix C.

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.

---

[1]This will change in SGX2, which can dynamically load new code pages into enclaves. This might make the function interpretation idea more feasible; enclave would need to check a signature on the new loaded pages before accepting. We can explore this design in future work when SGX2 becomes available.
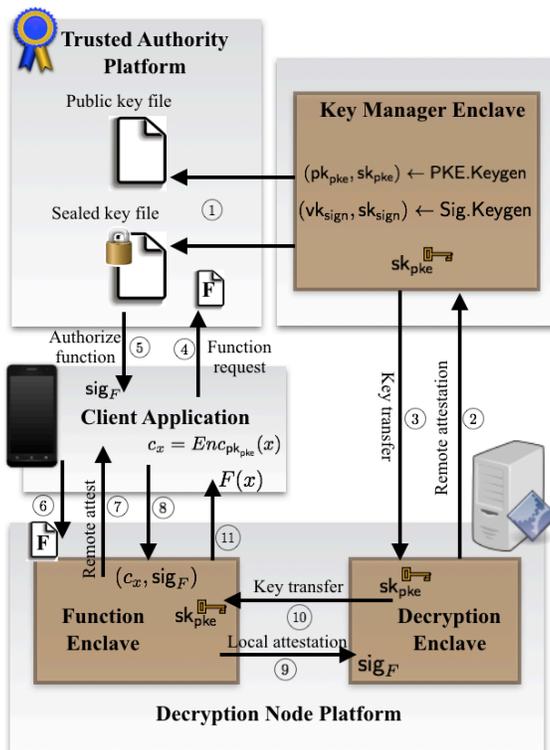
3.2 Architecture

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

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 $\mathsf{sk_{pke}}$ 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 $\mathsf{pk_{pke}}$. 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 $\mathsf{sk_{pke}}$ 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 $\mathsf{vk_{sign}}$. It remote attests to the KME, and includes $\mathsf{vk_{sign}}$ 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 $\mathsf{sk_{pke}}$ 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 $\mathsf{vk_{sign}}$, 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 $\mathsf{sk_{pke}}$ 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.[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

---

[2]Authenticating the DE's message to the function enclave serves the same purpose as authenticating the KME's message to the DE; it is only needed for the formal proof of security.

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 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 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.

4 IRON Implementation

We implemented a prototype of the IRON system with a single decryption node and a client application running locally on the decryption node. The implementation was developed in C++ using the Intel(R) SGX SDK 1.6 for Windows[3]. In principle, any function can be plugged into our system and loaded into the function enclave, provided that the function is implemented in such a way to resist side channel attacks (discussed in Section 2.4). In this paper, we demonstrate three special cases of functional encryption: *identity based encryption* (IBE), *order revealing encryption* (ORE), and *three input DNF* (3DNF). We chose to evaluate these special cases primarily in order demonstrate how our SGX assisted versions of these primitives perform in comparison to purely cryptographic versions that have been implemented, ranging from a widely-used and practical construction (IBE from pairings) to impractical ones (ORE and 3DNF from multilinear maps). We were also able to secure our SGX assisted implementation of these simple primitives against side-channel attacks.

There are two main applications, KeyManagerApp and FEClient. Enclaves are built as Windows DLLs. KeyManagerApp is the trusted authority application, which loads the enclave KeyManager.dll. FEClient combines the client and decryption node applications. It loads two enclaves, FEFunction.dll and FEDecryption.dll. Enclaves contain trusted function calls (ECALLs) that are executed in enclaves and called from the untrusted application. Untrusted function calls (OCALLs) are defined by the application and may be called from within an enclave.

All three enclaves link the MSR Elliptic Curve Cryptography Library 2.0 MSR_ECClib.lib[4] as a trusted static library, which is used to implement the elliptic curve ElGamal cryptosystem (in a Weierstrass curve over a 256-bit prime field), as well as the SDK's cryptographic library sgx_tcrypto.lib. The sgx_tcrypto.lib implements 256-bit elliptic curve Diffie-Hellman key exchange (EC256-DHKE) and ECDSA signatures over the NIST P-256 elliptic curve. It also implements Rijndael AES-GCM encryption on 128-bit key sizes and SHA256. We implement a CCA2-secure hybrid encryption scheme with ECC ElGamal and AES-GCM in the standard way.

We broadly describe the fucntion of each application below. The implementation details of all ECALLS are included in Appendix A.

4.1 KeyManagerApp

KeyManagerApp loads the enclave KeyManager.dll, i.e. the KME. The KME has a statically defined variable containing the expected measurement (MRENCLAVE) of FEDecryption.dll, i.e. the DE. If ECDSA and ElGamal public/secret keys have already been stored in files they are loaded into the KME, otherwise the application runs ECALLs ecdsa_setup and elgamal_setup and writes the output to files. Next, the application starts two threads: one that listens for command line inputs, and a second that listens for network connections from a decryption node. The ECALL sign_function is used to sign 256-bit length command line input messages. When a connection from a decryption node is received, the application receives from the decryption node a DHKE key share ga and an enclave quote structure de_quote generated by the DE. This is passed to km_ra_proc, which verifies the quote contents and signature against the output of ecdsa_setup and the statically defined expected MRENCLAVE value. If the verification passes it returns the KME's DHKE key share gb and an authenticated encryption of the ElGamal private key paired with the ECDSA verification key. These are sent back to the decryption node.[5]

4.2 FEClient

Decryption Enclave Setup The application FEClient first loads the enclave FEDecryption.dll, i.e. the DE. The application loads the Authority's public ECDSA verification key (generated by the KME) into the

---

DE. If the ElGamal secret key and ECDSA verification key have already been stored in a file it loads this into the DE, otherwise the application connects to the server running KeyManagerApp, sending a DHKE key share and an enclave quote. The server's response containing a EC256-DHKE key share and an encryption of an ElGamal private key are passed to the DE, which decrypts and seals the ElGamal private key. This sealed key is returned to the application and written to a file.

Function Enclave Local Attestation   The application next loads the enclave FEFunction.dll. The KME signature on the MRENCLAVE value of FEFunction.dll, `fe_report_signature`, is loaded into the enclave. Local attestation between the function and decryption enclaves is implemented with a sequence of ECALLs and OCALLs. The application calls `local_attest_to_decryption_enclave` in FEFunction.dll, which generates a DHKE key share wrapped in an enclave report. The OCALL `request_local_dh_session_ocall` passes these along with `fe_report_signature` to the ECALL `proc_local_attest` in FEDecryption.dll, which verifies the report and signature. If successful, it returns a DHKE key share wrapped in an enclave report and the ElGamal private key encrypted under the DHKE key derived from the two shares. The OCALL returns this output to its caller `local_attest_to_decryption_enclave`, which derives the DHKE key, decrypts the ElGamal private key, and stores it.

Functional decryption   Lastly, FEClient runs the functional decryption ECALL. This functional decryption ECALL is specific to the particular function implemented in FEFunction.dll. In our prototype, we demonstrate implementations of three functions: `decrypt_order`, `decrypt_ibe`, and `decrypt_3dnf`.

- **Order:** The ECALL `decrypt_order` takes as input a pair of ciphertexts (encrypted integers) and returns 1 if the first integer is less than the second, otherwise 0.

- **IBE:** In IBE, plaintexts consist of tagged payloads, i.e. have the form (tag, m) , and decrypting a ciphertext requires a key specific to the value of tag. To avoid creating a separate enclave for each key issued by the Authority, we have a single ECALL `decrypt_ibe` that multiplexes over all possible tags. This takes as input a ciphertext and a signature sig_{tag}. The Authority will issue sig_{tag} as part of the key for F_{tag} (this is in addition to the signature on the MRENCLAVE value of FEFunction.dll). The ECALL `decrypt_ibe` decrypts the ciphertext to obtain (tag, m), uses the public verification key to check that sig_{tag} is a valid signature on tag, and if so outputs m (otherwise it returns an error).

- **3DNF:** The ECALL `decrypt_3dnf` takes three ciphertexts as input, which are encryptions of n-bit inputs $x = x_1 \cdots x_n$, $y = y_1 \cdots y_n$, and $z = z_1 \cdots z_n$. It outputs $(x_1 \wedge y_1 \wedge z_1) \vee \cdots \vee (x_n \wedge y_n \wedge z_n)$.

ElGamal-AES-GCM hybrid encryption   The function `elgamal_aes_hybrid_encrypt` takes a plaintext and ECC ElGamal public key. It samples a random curve point p, derives from it a 128-bit AES key by applying SHA256 (and truncating). It ElGamal encrypts p, then AES-GCM encrypts the plaintext with derive key.

4.3   Side-channel resilience

The function and decryption enclave programs must be implemented to resist the software based side-channel attacks on SGX described in Section 2.4. The only enclave operations that touch secret data are decryption operations (AES-GCM and ElGamal) and the specific client functions that are loaded into the function enclave. Our implementation of AES-GCM uses the SGX SDK cryptographic library, which calls the AES-NI instruction for AES-GCM, and hence is resilient to software-based side-channels. Our implementation of ElGamal decryption uses the MSR Elliptic Curve Cryptography Library 2.0, which also claims resistance to timing attacks and cache-timing attacks. We implemented oblivious versions of all three client-loaded functions that we include in our evaluation (`decrypt_ibe`, `decrypt_order`, and `decrypt_3dnf`). This was easy to achieve by implementing data comparisons in x86 assembly with the `setg` and `sete` conditional instructions (similar to [OSF+16]), see Figure 3.

```
o_greater(x, y):           o_bytecmp(a, b):
   cmp    ecx, edx            cmp    ecx, edx
   setg   al                  sete   al
   ret                        ret
```

```
bool o_memcmp(char* a, char* b, int len){
        bool ret = 1;
        for (int i = 0; i < len; i++){
                ret = ret & o_bytecmp(a[i], b[i]);
        }
        return ret;
}
bool o_order(int x, int y){
        return o_greater(x, y);
}
```

Figure 3: Data oblivious comparison functions.

In general, for more complex functionalities, the implementation may require ORAM or other mitigation techniques. The Authority would need to ensure that it only signs function enclave programs that have data-oblivious implementations or are otherwise resilient to known side channel attacks.

5   Evaluation

We tested the prototype implementation on a platform running an Intel Skyake i7-6700 processor at 3.40 GHz with 8 GiB of RAM and Windows Server 2012 R2 Standard operating system. The code was developed and compiled in Visual Studio 2012 (platform toolset v110) with the Intel(R) SGX SDK 1.6 and Intel(R) SGX PSW 1.6 add-ons. We compiled with a 64-bit and Debug mode build configurations (currently, an Intel license is required to build enclaves in Release mode).

We only report on the performance of FE.Decrypt, FE.Setup, and FE.Keygen (Figures 4 and 5). FE.Encrypt in our system is standard public key encryption (our implementation uses ElGamal), and this is done outside of SGX enclaves. Note that all the procedures we evaluate are entirely local, which is why we do not include any network performance metrics. We omit performance measures on decryption node setup since the setup procedure requires contacting the Intel Attestation Server to process a remote attestation, which we were unable to test without a license from Intel. Nonetheless, the setup is a one-time operation that is completed when a decryption node platform is first established, and thus has little overall impact on decryption performance.

Our evaluation demonstrates that the SGX-based functional encryption examples we implemented (IBE, ORE, and 3DNF) are not only practical but also orders of magnitude faster than cryptographic solutions without secure hardware, particularly for the multi-input functions ORE and 3DNF. In general, multi-input functional encryption (without SGX) is totally impractical given current cryptographic techniques. But we have shown that SGX is considerably faster even for a type of functional encryption widely used in practice, i.e. IBE. We recognize that more complex functionalities than the ones we have implemented, particularly functions that operate on data outside the EPC, may require additional side-channel mitigation techniques such as ORAM, which will impact performance. However, we would still expect these to outperform traditional functional encryption by orders of magnitude.

FE.Setup and FE.Keygen evaluation   Figure 4 contains a break down of the run time for FE.Setup and FE.Keygen.

create enclave	57 ms
ECDSA setup	74 ms
ElGamal setup	8 ms
server setup	2 ms
sign message	11 ms
Total	141 ms

Figure 4: FE.Setup and FE.Keygen run time, including enclave creation and generation of public/secret keys for ECDSA and ElGamal on 256 bit EC curves. FE.Keygen corresponds to sign message, which generates an ECDSA signature on a 256-bit input.

FE.Decrypt evaluation We evaluated the performance of FE.Decrypt for three special cases of function encryption: *identity based encryption* (IBE), *order revealing encryption* (ORE), and *three input DNF* (3DNF). We chose these functionalities primarily to demonstrate how our SGX assisted versions of these primitives perform in comparison to their purely cryptographic versions (IBE from pairings, DNF and 3DNF from multilinear maps). The table in Figure 5 summarizes the decryption times for the three functionalities, including a breakdown of the time spent on the three main ECALLS of the decryption process: enclave creation, local attesting to the DE, and finally decrypting the ciphertext and evaluating the function.

Functionality:	**IBE**	**ORE**	**3DNF**
create enclave	14.5 ms	20.7 ms	19.7 ms
local attest	1.6 ms	2.1 ms	2.1 ms
decrypt & eval	0.98 ms	0.84 ms	0.96 ms
Total	17.8 ms	23.78 ms	22.76 ms

Figure 5: Breakdown of FE.Decrypt run times for each of our IRON implementations of IBE, ORE, and 3DNF. The input in IBE consisted of a 3-byte tag and a 32-bit integer payload. The input pairs in ORE were 32-bit integers, and the input triplets in 3DNF were 16-bit binary strings. (The input types were chosen for consistency with the 5Gen experiments). The column decrypt gives the cost of running a single decryption.

Amortized decryption costs As shown in Figure 5, for each of the functionalities the time spent creating the enclave dominates the time spent on decryption and evaluation by 2 orders of magnitude. Once the function enclave has been created and local attestation to the DE is complete, the same enclave can be used to decrypt an arbitrary number of input ciphertext tuples. Thus, the amortized cost of running decryption on many ciphertexts (or tuples of ciphertexts) is much lower than the cost of running decryption on a single input. (This is not the case with cryptographic implementations of these functionalities). The amortized cost of running decryption on 1000 inputs (ciphertext tuples) is included in the next table, Figure 6.
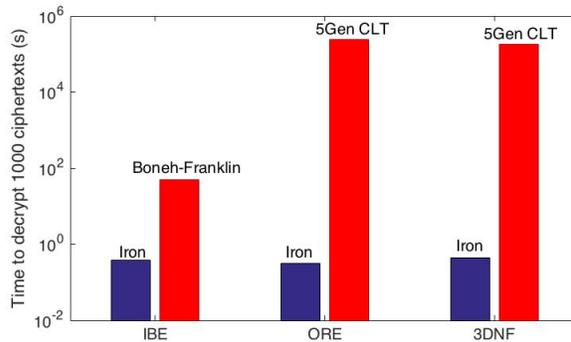

Figure 7: Comparison of time for decrypting 10^3 ciphertext tuples using the IRON implementation of IBE, ORE, 3DNF vs cryptographic implementations from pairings and mmaps respectively.

	IBE$^{\text{SGX}}$	**IBE**$^{[\text{BF01}]}$	$\times$ increase
$\|\mathsf{msg}\|$:	35 bits	35 bits	NA
$\|\mathsf{c}\|$:	175 bytes	471 bytes	2.69
decrypt:	17.8 ms	49 ms	2.75
decrypt*:	0.39 ms	49 ms	125.64

	ORE$^{\text{SGX}}$	**ORE**$^{\text{5Gen}}$	$\times$ increase
$\|\mathsf{msg}\|$:	32 bits	32 bits	NA
$\|\mathsf{c}\|$:	172 bytes	4.7 GB	$27.3 \cdot 10^6$
decrypt :	23.78 ms	4 m	$10.1 \cdot 10^3$
decrypt* :	0.32 ms	4 m	$750 \cdot 10^3$

	3DNF$^{\text{SGX}}$	**3DNF**$^{\text{5Gen}}$	$\times$ increase
$\|\mathsf{msg}\|$:	16 bits	16 bits	NA
$\|\mathsf{c}\|$:	170 bytes	2.5 GB	$14.7 \cdot 10^6$
decrypt:	22.76 ms	3 m	$7.9 \cdot 10^3$
decrypt*:	0.45 ms	3 m	$400 \cdot 10^3$

Figure 6: Comparison of decryption times and ciphertext sizes for the Iron implementation of IBE, ORE, 3DNF to cryptographic implementations. The 5Gen ORE and 3DNF implementation referenced here uses the CLT mmap with an 80-bit security parameter. The column decrypt gives the cost of running a single decryption, and decrypt* gives the amortized cost (per ciphertext tuple) of 10^3 decryptions.

Comparison to cryptographic implementations We measured decryption time for an implementation[6] of Boneh-Franklin IBE [BF01] on our platform. We also include decryption time performance numbers for the 5Gen implementation[7] of mmap-based ORE and 3DNF as reported in [LMA$^+$16]. We did not deem it necessary to measure 5Gen implementations of ORE and 3DNF on our platform since their performance is 4 orders of magnitude slower than that of our SGX-based implementation. The comparison for these multi-input functionalities simply illustrates how our Iron system makes possible primitives that are currently otherwise infeasible to build for practical use without secure hardware.

6 Formal Models and Definitions

6.1 Formal HW model

In our FE model, parties will have access to the secure hardware defined below. Our definition is expressed as a black-box program HW that captures the secure hardware's functionality and its interface exposed to the user.

Definition 6.1. *A secure hardware functionality* HW *for a class of (probabilistic polynomial time) programs* $\mathcal{Q}$ *consists of the following interface:* HW.Setup, HW.Load, HW.Run, HW.Run&Report, HW.Run&Quote, HW.ReportVerify, HW.QuoteVerify. HW *also has an internal state* state *that consists of two variables* HW.sk$_{\mathsf{quote}}$ *and* HW.sk$_{\mathsf{report}}$ *and a table* T *consisting of enclave state tuples indexed by enclave handles. The two variables* HW.sk$_{\mathsf{quote}}$ *and* HW.sk$_{\mathsf{report}}$ *will be used for storing signing keys and the table* T *will be used for managing the internal states of loaded enclave programs.*

- HW.Setup(1^λ): HW.Setup takes in a security parameter λ. It generates the secret keys sk$_{\mathsf{quote}}$, sk$_{\mathsf{report}}$,

---
[6]The Stanford IBE command-line utility ibe-0.7.2-win, available at https://crypto.stanford.edu/ibe/download.html
[7]5Gen, available https://github.com/5GenCrypto

and stores these in $\mathsf{HW}.\mathsf{sk_{quote}}, \mathsf{HW}.\mathsf{sk_{report}}$ respectively. Finally, it generates and outputs public parameters params.

- $\mathsf{HW}.\mathsf{Load}(\mathsf{params}, Q)$: This loads a stateful program into an enclave. $\mathsf{HW}.\mathsf{Load}$ takes as input a program $Q \in \mathcal{Q}$ and some global parameters params. It first creates an enclave and loads Q and generates a handle hdl that will be used to identify the enclave running Q. It initializes the entry $T[\mathsf{hdl}] = \emptyset$.

- $\mathsf{HW}.\mathsf{Run}(\mathsf{hdl}, \mathsf{in})$: This runs an enclave program. It takes in a handle hdl corresponding to an enclave running the stateful program Q and an input in. It runs Q at state $T[\mathsf{hdl}]$ with input in and records the output out. It sets $T[\mathsf{hdl}]$ to be the updated state of Q and outputs out.

- $\mathsf{HW}.\mathsf{Run\&Report}_{\mathsf{sk_{report}}}(\mathsf{hdl}, \mathsf{in})$: This executes a program in an enclave and also generates an attestation of its output that can be verified by an enclave program on the same HW platform. It takes as inputs a handle hdl for an enclave running a program Q and an input in for Q. The algorithm first executes Q on in to get out, and updates $T[\mathsf{hdl}]$ accordingly. $\mathsf{HW}.\mathsf{Run\&Report}$ outputs the tuple $\mathsf{report} := (\mathsf{md_{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \mathsf{mac})$, where $\mathsf{md_{hdl}}$ is the metadata associated with the enclave, tag_Q is a program tag that can be used to identify the program running inside the enclave (it can be a cryptographic hash of the program code Q) and mac is a cryptographic MAC produced using $\mathsf{sk_{report}}$ on $(\mathsf{md_{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out})$.

- $\mathsf{HW}.\mathsf{Run\&Quote}_{\mathsf{sk_{HW}}}(\mathsf{hdl}, \mathsf{in})$: This executes a program in an enclave and also generates an attestation of its output that can be publicly verified, e.g. by a remote party. This takes as inputs a handle hdl corresponding to an enclave running a program Q and an input in for Q. This algorithm has a restricted access to the key $\mathsf{sk_{HW}}$ for using it to sign messages. The algorithm first executes Q on in to get out, and updates $T[\mathsf{hdl}]$ accordingly. $\mathsf{HW}.\mathsf{Run\&Quote}$ then outputs the tuple $\mathsf{quote} := (\mathsf{md_{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \sigma)$, where $\mathsf{md_{hdl}}$ is the metadata associated with the enclave, tag_Q is a program tag for Q and σ is a signature on $(\mathsf{md_{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out})$.

- $\mathsf{HW}.\mathsf{ReportVerify}_{\mathsf{sk_{report}}}(\mathsf{hdl}', \mathsf{report})$: This is the report verification algorithm. It takes as inputs, a handle hdl' for an enclave and a $\mathsf{report} = (\mathsf{md_{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \mathsf{mac})$. It uses $\mathsf{sk_{report}}$ to verify the MAC. If mac is valid, it outputs 1 and adds a tuple $(\mathsf{report}, 1)$ to $T[\mathsf{hdl}']$. Otherwise it outputs 0 and adds $(\mathsf{report}, 0)$ to $T[\mathsf{hdl}']$.

- $\mathsf{HW}.\mathsf{QuoteVerify}(\mathsf{params}, \mathsf{quote})$: This is the quote verification algorithm. This takes params and $\mathsf{quote} = (\mathsf{md_{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \pi)$ as input. It outputs 1 if the signature verification of σ succeeds. It outputs 0 otherwise.

Correctness A HW scheme is correct if the following things hold (using the syntax from Definition 6.1): For all $\mathsf{aux}, Q \in \mathcal{Q}$, all in in the input domain of Q and all handles $\mathsf{hdl}' \in \mathcal{H}$,

- Correctness of Run: $\mathsf{out} = Q(\mathsf{in})$ if Q is deterministic. More generally, $\exists$ random coins r (sampled in run time and used by Q) such that $\mathsf{out} = Q(\mathsf{in})$.

- Correctness of Report and ReportVerify:

$$\Pr\left[\mathsf{HW}.\mathsf{ReportVerify}_{\mathsf{sk_{report}}}(\mathsf{hdl}', \mathsf{report}) = 0\right] = \mathrm{negl}(\lambda)$$

- Correctness of Quote and QuoteVerify:

$$\Pr\left[\mathsf{HW}.\mathsf{QuoteVerify}(\mathsf{params}, \mathsf{quote}) = 0\right] = \mathrm{negl}(\lambda)$$

6.1.1 Local attestation unforgeability

The local attestation unforgeability (LocAttUnf) security is defined similarly to the unforgeability security of a MAC scheme. Informally, it says that no adversary can produce a $\mathsf{report} = \big(\mathsf{md}'_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \mathsf{mac}\big)$ that verifies correctly for any $\mathsf{hdl}' \in \mathcal{H}$ and $\mathsf{out} = Q(\mathsf{in})$, without querying the inputs $(\mathsf{hdl}, \mathsf{in})$.

This is formally defined by the following security game.

Definition 6.2. (LocAttUnf-HW). *Consider the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$.*

1. *$\mathcal{A}$ provides an aux.*

2. *$\mathcal{C}$ runs the $\mathsf{HW.Setup}(1^\lambda, \mathsf{aux})$ algorithm to obtain the public parameters params, secret keys $(\mathsf{sk}_{\mathsf{HW}}, \mathsf{sk}_{\mathsf{report}})$ and an initialization string state. It gives params to $\mathcal{A}$, and keeps $(\mathsf{sk}_{\mathsf{HW}}, \mathsf{sk}_{\mathsf{report}})$ and state secret in the secure hardware.*

3. *$\mathcal{C}$ initializes a list $\mathrm{query} = \{\}$.*

4. *$\mathcal{A}$ can run $\mathsf{HW.Load}$ on any input (params, Q) of its choice and get back hdl.*

5. *$\mathcal{A}$ can run $\mathsf{HW.Run\&Report}$ on input $(\mathsf{hdl}, \mathsf{in})$ of its choice and get $\mathsf{report} := \big(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \mathsf{mac}\big)$. For every run, $\mathcal{C}$ adds the tuple $(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out})$ to the list query.*

6. *$\mathcal{A}$ can also run $\mathsf{HW.ReportVerify}$ on input $(\mathsf{hdl}', \mathsf{report})$ of its choice and gets back the result.*

We say the adversary wins the above experiment if:

1. *$\mathsf{HW.ReportVerify}(\mathsf{hdl}'^*, \mathsf{report}^*) = 1$, where $\mathsf{report}^* = (\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_Q, \mathsf{in}^*, \mathsf{out}^*)$ and*

2. *$(\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_Q, \mathsf{in}^*, \mathsf{out}^*)$ was not added to query before $\mathcal{A}$ queried $\mathsf{HW.ReportVerify}$ on $(\mathsf{hdl}'^*, \mathsf{report}^*)$.*

The HW scheme is LocAttUnf-HW secure if no adversary can win the above game with non-negligible probability.

6.1.2 Remote attestation unforgeability

The remote attestation unforgeability (RemAttUnf) security is defined similarly to the unforgeability security of a signature scheme. Informally, it says that no adversary can produce a $\mathsf{quote} = \big(\mathsf{hdl}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \pi\big)$ that verifies correctly and $\mathsf{out} = Q(\mathsf{in})$, without querying the inputs $(\mathsf{hdl}, \mathsf{in})$.

This is formally defined by the following security game.

Definition 6.3. (RemAttUnf-HW). *Consider the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$.*

1. *$\mathcal{A}$ provides an aux.*

2. *$\mathcal{C}$ runs the $\mathsf{HW.Setup}(1^\lambda, \mathsf{aux})$ algorithm to obtain the public parameters params, secret keys $(\mathsf{sk}_{\mathsf{HW}}, \mathsf{sk}_{\mathsf{report}})$ and an initialization string state. It gives params to $\mathcal{A}$, and keeps $(\mathsf{sk}_{\mathsf{HW}}, \mathsf{sk}_{\mathsf{report}})$ and state secret in the secure hardware.*

3. *$\mathcal{C}$ initializes a list $\mathrm{query} = \{\}$.*

4. *$\mathcal{A}$ can run $\mathsf{HW.Load}$ on any input (params, Q) of its choice and get back hdl.*

5. *Also, $\mathcal{A}$ can run $\mathsf{HW.Run\&Quote}$ on input $(\mathsf{hdl}, \mathsf{in})$ of its choice and get $\mathsf{quote} := \big(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \pi\big)$. For every run, $\mathcal{C}$ adds the tuple $(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out})$ to the list query.*

6. *Finally, the adversary outputs $\mathsf{quote}^* = (\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_Q, \mathsf{in}^*, \mathsf{out}^*, \pi^*)$.*

We say the adversary wins the above experiment if:

1. $\mathsf{HW.QuoteVerify}(\mathsf{params}, \mathsf{quote}^*) = 1$,

2. $(\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_Q, \mathsf{in}^*, \mathsf{out}^*) \notin \mathsf{query}$

The HW *scheme is RemAttUnf-HW secure if no adversary can win the above game with non-negligible probability.*

Note that the scheme is secure even if $\mathcal{A}$ can produce a quote^* different from the query outputs for some $(\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_Q, \mathsf{in}^*, \mathsf{out}^*) \in \mathsf{query}$. But quote^* cannot be a proof for a different program or input or output. This definition resembles the existential unforgeability like notions.

We also point out some other important properties of the secure hardware that we impose in our model.

- Any user only has black box access to these algorithms and hence hidden from the internal secret key $\mathsf{sk_{HW}}$, initial state state or intermediary states of the programs running inside secure containers.

- The output of the $\mathsf{HW.Run\&Quote}$ algorithm is succinct: it does not include the full program description, for instance.

- We also require the params and the handles hdl to be *independent* of aux. In particular, for all $\mathsf{aux}, \mathsf{aux}'$,

$$(\mathsf{params}, \mathsf{sk_{HW}}, \mathsf{sk_{report}}, \mathsf{state}) \leftarrow \mathsf{HW.Setup}(1^\lambda, \mathsf{aux})$$
$$(\mathsf{params}', \mathsf{sk}'_{\mathsf{HW}}, \mathsf{sk}'_{\mathsf{report}}, \mathsf{state}') \leftarrow \mathsf{HW.Setup}(1^\lambda, \mathsf{aux}')$$

and for $\mathsf{hdl} \leftarrow \mathsf{HW.Load_{state}}(\mathsf{params}, Q)$ and $\mathsf{hdl}' \leftarrow \mathsf{HW.Load_{state'}}(\mathsf{params}', Q)$, the tuples $(\mathsf{params}, \mathsf{hdl})$ and $(\mathsf{params}', \mathsf{hdl}')$ are identically distributed.

Differences from real SGX The purpose of this abstraction is to provide us with a way of formally modeling our construction of FE in the simplest way possible in order to analyze its security properties. Here, we explain and justify key differences between HW and the actual Intel SGX implementation. We stress that these justifications are not formal security arguments. Formally proving that the Intel SGX implementation securely realizes a given specification is a separately challenging task.

- In our model, HW is a black-box program that loads and manages enclaves, including updating their state in $\mathsf{HW.ReportVerify}$. This internal management is entirely hidden from the user, which only sees the interface, inputs, and outputs. In real Intel SGX, only operations internal to a program running in an enclave (i.e. instructions that operate on registers/memory in the EPC) are entirely hidden from the user, and the enclave program's state cannot be modified by an external entity. Programs running in enclaves can directly run instructions to generate and verify reports.

- The key $\mathsf{sk_{HW}}$ used to sign remote attestations in our model is generated during $\mathsf{HW.Setup}$ (i.e. in the trusted manufacturing facility). In Intel SGX, this key is not actually fused into the device. It is delivered to an special enclave QE (the Quoting Enclave) running on the device that symmetrically authenticates to the Intel Provisioning Server by accessing a key (the Root Provisioning Key) that is fused into the device and also given to Intel. The QE then receives the private key for a group signature scheme through a blind join protocol (see [JSR+16]), and uses this key to generate quotes on behalf of other enclaves. Our model compresses the manufacturing and provisioning processes into $\mathsf{HW.Setup}$.

- Our $\mathsf{HW.Run\&Report}$ algorithm generates a report that can be verified by any enclave on the same HW platform. In SGX, a report is generated for a specific destination enclave and only that enclave can verify its validity. However, this particular feature of SGX is not relevant for our application.

- Intel SGX also has the capability of sealing data with a hardware fused Seal Key. In particular, this allows the device to use persistent storage for keys. For simplicity, we do not include this in our formal model, and assume the trusted HW functionality is persistent.

- HW.Run&Quote and HW.QuoteVerify use a standard cryptographic signature scheme to sign and verify quotes. In Intel SGX the signatures used for quotes are actually anonymous group signatures, but this additional property is not relevant to our application, so we omit it for simplicity. Moreover, currently Intel SGX requires the user to contact the Intel Attestation Server (IAS) to verify group signatures. Theoretically, verifying an anonymous group signature only requires the public group key, and needn't involve the IAS.

HW security models   One way of viewing this definition of HW is that is describes the ideal functionality or oracle that models the real (physical) world assumptions about the hardware security properties of Intel SGX (see Katz [Kat07] on this approach to modeling secure hardware in cryptographic protocols). In this view, an adversary shouldn't be able to distinguish between interacting with the real world hardware and the ideal functionality. Moreover, in a proof of security, the adversary's interaction with the ideal functionality can be simulated. This is a very strong assumption on the secure hardware being used, particularly since the adversary has access to the physical hardware and can closely monitor its behavior. A weaker assumption is that HW is merely an abstraction of the actual hardware instantiation, and an adversary's physical interactions with HW cannot be simulated. Our security proof of the main system/construction we have presented assumes the first model. In Section 8.1 we explore the second model, though it turns out that we cannot achieve the standard non-interactive notion of functional encryption in this stronger security model.

Related models   Barbosa et. al. [BPSW16] define a similar interface/ideal functionality to represent systems like SGX that perform attested computation. Compared to their model, our model sacrifices some generality for a simpler syntax that more closely models SGX. Their security model uses a game-based definition of attested computation, similar to the second security model we discuss in Section 8.1.

Pass, Shi, and Tramer [PST17] also define an ideal functionality for attested computation in the Universal Composability framework [Can01]. The goal of their model is to explore *composable* security for protocols using secure processors performing attested computation. Similar to [BPSW16] their syntax is more abstract that ours, e.g. does not distinguish between local and remote attestation. However, their hardware security model is more similar in that it allows the hardware functionality to be simulated. A key difference is that their simulator does not possess the hardware's secret signing key(s) used to generate attestations. Our simulator will be given the hardware's secret keys, similar to trapdoor information in CRS-model proofs.

Bahmani et al [BBB+16] adapts the SGX model of [BPSW16] to deal with sequences of SGX computations that may be stateful, asynchronous, and interleaved with other computations. Their model is called *labelled attested computation*, which refers to labels being appended to every enclave input/output in order to track state. This capability is implicitly captured in our model as well.

6.2   Functional Encryption

We adapt the definition of functional encryption to fit the computational model of our system. Recall that the decryption process in our system utilizes local SGX enclaves and also may communicate with a remote SGX enclave, the KME. We model interaction with any local enclaves as calls to the HW functionality defined in Definition 6.1. However, communication with the KME is over an untrusted channel with the remote platform on which the KME is running. Therefore, we separately model this as communication with an oracle $\mathsf{KM}(\cdot)$.

Non-interaction   Non-interaction is central to the standard notion of functional encryption. Our construction of hardware assisted FE requires a one-time setup operation where the decryptor's hardware contacts the KME to receive a secret key. However, this interaction only occurs once in the setup of a decryption node, and thereafter decryption is non-interactive. To capture this restriction on interaction we add to the standard FE algorithms an additional algorithm FE.DecSetup, which is given oracle access to a Key Manager $\mathsf{KM}(\cdot)$. The decryption algorithm FE.Dec is only given access to HW.

Pre-processing In our model, we allow all the parties performing decryption to complete a pre-processing phase. The pre-processing is executed by the trusted environment. In our construction, this step is executed by the manufacturer to setup and initialize the secure hardware. Pre-processing is executed before any FE algorithm, and hence does not depend on any of its parameters. An output of the pre-processing phase includes public parameters which are implicitly given to all subsequent algorithms.

A functional encryption scheme $\mathcal{FE}$ for a family of programs $\mathcal{P}$ and message space $\mathcal{M}$ consists of five p.p.t. algorithms $\mathcal{FE} = (\mathsf{FE.Setup}, \mathsf{FE.Keygen}, \mathsf{FE.Enc}, \mathsf{FE.DecSetup}, \mathsf{FE.Dec})$ defined as follows.

- $\mathsf{FE.Setup}(1^\lambda)$: The setup algorithm takes as input the unary representation of the security parameter λ and outputs the master public key mpk and the master secret key msk.

- $\mathsf{FE.Keygen}(\mathsf{msk}, P)$: The key generation algorithm takes as input the master secret key msk and a program $P \in \mathcal{P}$ and outputs the secret key sk_P for P.

- $\mathsf{FE.Enc}(\mathsf{mpk}, \mathsf{msg})$: The encryption algorithm takes as input the master public key mpk and an input message $\mathsf{msg} \in \mathcal{M}$ and outputs a ciphertext ct.

- $\mathsf{FE.DecSetup}^{\mathsf{KM}(\cdot),\mathsf{HW}(\cdot)}(\mathsf{mpk})$: The decryptor node setup algorithm takes as input the master public key mpk and gets access to the KM oracle and the HW oracles. And it outputs a handle hdl to be used by the actual decryption algorithm.

- $\mathsf{FE.Dec}^{\mathsf{HW}(\cdot)}(\mathsf{hdl}, \mathsf{sk}_P, \mathsf{ct})$: The decryption algorithm takes as input a handle hdl for an enclave, a secret key sk_P and a ciphertext ct and outputs $P(\mathsf{msg})$ or $\perp$. This algorithm has access to the interface for all the algorithms of the secure hardware HW.

Correctness A functional encryption scheme $\mathcal{FE}$ is correct if for all $P \in \mathcal{P}$ and all $\mathsf{msg} \in \mathcal{M}$, the probability for $\mathsf{FE.Dec}^{\mathsf{HW}(\cdot)}\Big(\mathsf{hdl}, \mathsf{sk}_P, \mathsf{ct}\Big)$ to be not equal to $P(\mathsf{msg})$ is $\mathsf{negl}(\lambda)$, where $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$, $\mathsf{sk}_P \leftarrow \mathsf{FE.Keygen}(\mathsf{msk}, P)$, $\mathsf{ct} \leftarrow \mathsf{FE.Enc}(\mathsf{mpk}, \mathsf{msg}$ and $\mathsf{hdl} \leftarrow \mathsf{FE.DecSetup}^{\mathsf{KM}(\cdot),\mathsf{HW}(\cdot)}(\mathsf{mpk})$ and the probability is taken over the random coins of the probabilistic algorithms $\mathsf{FE.Setup}, \mathsf{FE.Keygen}, \mathsf{FE.Enc}, \mathsf{FE.DecSetup}$.

Security definition Here, we define a strong simulation-based security of FE similar to [BSW12,GVW12, AGVW13]. In this security model, a polynomial time adversary will try to distinguish between the real world and a "simulated" world. In the real world, algorithms work as defined in the construction. In the simulated world, we will have to construct a polynomial time simulator which has to do the experiment given only the program queries P made by the adversary and the corresponding results $P(\mathsf{msg})$.

Definition 6.4 (SimSecurity-FE). *Consider a stateful simulator $\mathcal{S}$ and a stateful adversary $\mathcal{A}$. Let $U_{\mathsf{msg}}(\cdot)$ denote a universal oracle, such that $U_{\mathsf{msg}}(P) = P(\mathsf{msg})$.*

Both games begin with a pre-processing phase executed by the environment. In the ideal game, pre-processing is simulated by $\mathcal{S}$. Now, consider the following experiments.

$$
\begin{array}{ll}
\mathbf{Exp}_{\mathcal{FE}}^{\mathsf{real}}(1^\lambda): & \mathbf{Exp}_{\mathcal{FE}}^{\mathsf{ideal}}(1^\lambda): \\[4pt]
(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda) & \mathsf{mpk} \leftarrow \mathcal{S}(1^\lambda) \\[4pt]
(\mathsf{msg}) \leftarrow \mathcal{A}^{\mathsf{FE.Keygen}(\mathsf{msk},\cdot)}(\mathsf{mpk}) & \mathsf{msg} \leftarrow \mathcal{A}^{\mathcal{S}(\cdot)}(\mathsf{mpk}) \\[4pt]
\mathsf{ct} \leftarrow \mathsf{FE.Enc}(\mathsf{mpk}, \mathsf{msg}) & \mathsf{ct} \leftarrow \mathcal{S}^{U_{\mathsf{msg}}(\cdot)}(1^\lambda, 1^{|\mathsf{msg}|}) \\[4pt]
\alpha \leftarrow \mathcal{A}^{\mathsf{FE.Keygen}(\mathsf{msk},\cdot),\mathsf{HW},\mathsf{KM}(\cdot)}(\mathsf{mpk}, \mathsf{ct}) & \alpha \leftarrow \mathcal{A}^{\mathcal{S}^{U_{\mathsf{msg}}(\cdot)}(\cdot)}(\mathsf{mpk}, \mathsf{ct}) \\[4pt]
Output\ (\mathsf{msg}, \alpha) & Output\ (\mathsf{msg}, \alpha)
\end{array}
$$

In the above experiment, oracle calls by $\mathcal{A}$ to the key-generation, HW and KM oracles are all simulated by the simulator $\mathcal{S}^{U_{\mathsf{msg}}(\cdot)}(\cdot)$. An FE scheme is simulation-secure against adaptive adversaries if there is a stateful probabilistic polynomial time simulator $\mathcal{S}$ that on each FE.Keygen query P queries its oracle $U_{\mathsf{msg}}(\cdot)$ only on the same P (and hence learn just $P(\mathsf{msg})$), such that for every probabilistic polynomial time adversary $\mathcal{A}$ the following distributions are computationally indistinguishable.

$$\mathsf{Exp}^{\mathsf{real}}_{\mathcal{FE}}(1^\lambda) \overset{c}{\approx} \mathsf{Exp}^{\mathsf{ideal}}_{\mathcal{FE}}(1^\lambda)$$

Note that the above definition handles one message only. This can be extended to a definition of security for many messages by allowing the adversary to adaptively output many messages while providing him the ciphertext for a message whenever he outputs one. Here, the simulator will have an oracle $U_{\mathsf{msg}_i}(\cdot)$ for every msg_i output by the adversary.

Simulating HW   As previously discussed, we let the simulator intercept all the adversary's queries to HW and return simulated responses, just as in [CKZ13]. If we do not allow simulation of HW, it is impossible to achieve Definition 6.4. In Section 8.1 we provide a modified FE definition to allow minimal interaction[8] with an efficient KM oracle during every run of FE.Dec, and give a construction that realizes this modified FE in the stronger security model.

6.3   Crypto primitive definitions

6.3.1   Secret key encryption

A secret key encryption scheme E supporting a message domain $\mathcal{M}$ consists of the following polynomial time algorithms:

E.KeyGen(1^λ) The key generation algorithm takes in a security parameter and outputs a key sk from the key space $\mathcal{K}$.

E.Enc(sk, msg) The encryption algorithm takes in a key sk and a message $\mathsf{msg} \in \mathcal{M}$ and outputs the ciphertext ct.

E.Dec(sk, ct) The decryption algorithm takes in a key sk and a ciphertext ct and outputs the decryption msg.

The first two algorithms are probabilistic whereas the decryption algorithm is deterministic.

Correctness   A secret key encryption scheme E is correct if for all λ and all $\mathsf{msg} \in \mathcal{M}$,

$$\Pr\left[\mathsf{E.Dec}\Big(\mathsf{sk}, \mathsf{E.Enc}(\mathsf{sk}, \mathsf{msg})\Big) \neq \mathsf{msg}\Big|\mathsf{sk} \leftarrow \mathsf{E.KeyGen}(1^\lambda)\right] = \mathrm{negl}(\lambda)$$

where the probability is taken over the random coins of the probabilistic algorithms E.KeyGen, E.Enc.

An encryption scheme provides data confidentiality. So, it should prevent an adversary from learning which message is encrypted in a ciphertext. The security of E is formally defined by the following security game.

Definition 6.5. (IND-CPA security of E). *Security is depicted by the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$.*

1. *The challenger run the E.KeyGen algorithm to obtain a key sk from the key space $\mathcal{K}$.*

2. *The challenger also chooses a random bit $b \in \{0, 1\}$.*

---

[8]Allowing unbounded interaction would lead to trivial constructions where KM simply decrypts the ciphertext and returns the function of the message.

3. *Whenever the adversary provides a pair of messages* $(\mathsf{msg}_0, \mathsf{msg}_1)$ *of its choice, the challenger replies with* $\mathsf{E.Enc}(\mathsf{sk}, \mathsf{msg}_b)$.

4. *The adversary finally outputs its guess* b'.

The advantage of adversary in the above game is

$$\mathsf{Adv}_{\mathsf{enc}}(\mathcal{A}) := \Pr[b' = b] - \frac{1}{2}$$

A secret key encryption scheme E *is said to have indistinguishability security under chosen plaintext attack if there is no polynomial time adversary* $\mathcal{A}$ *which can win the above game with probability non-negligible in* λ.

6.3.2 A signature scheme

A digital signature scheme S supporting a message domain $\mathcal{M}$ consists of the following polynomial time algorithms:

$\mathsf{S.KeyGen}(1^\lambda$ The key generation algorithm takes in a security parameter and outputs the signing key sk and a verification key vk.

$\mathsf{S.Sign}(\mathsf{sk}, \mathsf{msg})$ The signing algorithm takes in a signing key sk and a message $\mathsf{msg} \in \mathcal{M}$ and outputs the signature σ.

$\mathsf{S.Verify}(\mathsf{vk}, \sigma, \mathsf{msg}))$ The verification algorithm takes in a verification key vk, a signature σ and a message msg and outputs 0 or 1.

The first two algorithms are probabilistic whereas the verification algorithm is deterministic.

Correctness A signature scheme S is correct if for all $\mathsf{msg} \in \mathcal{M}$,

$$\Pr\left[\mathsf{S.Verify}\left(\mathsf{vk}, \mathsf{S.Sign}(\mathsf{sk}, \mathsf{msg}), \mathsf{msg}\right) = 0 \middle| (\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{S.KeyGen}(1^\lambda)\right] = \mathsf{negl}(\lambda)$$

where the probability is taken over the random coins of the probabilistic algorithms $\mathsf{S.KeyGen}, \mathsf{S.Sign}$.

Signatures provide authenticity. So, an adversary without the signing key should not be able to generate a valid signature. The security of S is formally defined by the following security game.

Definition 6.6. (EUF-CMA security of $\mathcal{S}$). *Consider the following game between a challenger* $\mathcal{C}$ *and an adversary* $\mathcal{A}$.

1. *The challenger runs the* $\mathsf{S.KeyGen}$ *algorithm to obtain the key pair* $(\mathsf{sk}, \mathsf{vk})$, *and provides the verification key* vk *to the adversary.*

2. *Initialize* query = {}.

3. *Now, whenever the adversary provides a query with a message* msg, *the challenger replies with* $\mathsf{S.Sign}(\mathsf{sk}, \mathsf{msg})$. *Also,* query = query $\cup$ msg.

4. *Finally, the adversary outputs a forged signature* σ^* *corresponding to a message* msg^*.

The advantage of $\mathcal{A}$ *in the above security game is*

$$\mathsf{Adv}_{\mathsf{sign}}(\mathcal{A}) := \Pr\left[\mathsf{S.Verify}(\mathsf{vk}, \sigma^*, \mathsf{msg}^*) = 1 \middle| \mathsf{msg}^* \notin \text{query}\right]$$

A signature scheme S *is said to be existentially unforgeable under chosen message attack if there is no polynomial time adversary which can win the above game with probability non-negligible in* λ.

6.3.3 Public key encryption

A public key encryption (PKE) is a generalization of secret key encryption where anyone with the public key of the receiver can encrypt messages to the receiver. A PKE scheme supporting a message domain $\mathcal{M}$ consists of the following algorithms:

PKE.KeyGen(1^λ) The key generation algorithm takes in a security parameter and outputs a key pair $(\mathsf{pk}, \mathsf{sk})$.

PKE.Enc($\mathsf{pk}, \mathsf{msg}$) The encryption algorithm takes in a public key pk and a message $\mathsf{msg} \in \mathcal{M}$, outputs a ciphertext ct which is an encryption of msg under pk.

PKE.Dec(sk, ct) The decryption algorithm takes in a secret key sk and a ciphertext ct and outputs the decryption msg or $\bot$.

The first two algorithms are probabilistic whereas the decryption algorithm is deterministic.

Correctness A PKE scheme PKE is correct if for all λ and $\mathsf{msg} \in \mathcal{M}$,

$$\Pr\left[\mathsf{PKE.Dec}\Big(\mathsf{sk}, \mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{msg})\Big) \neq \mathsf{msg}\Big|(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)\right] = \mathrm{negl}(\lambda)$$

where the probability is taken over the random coins of the probabilistic algorithms KeyGen, Enc.

A PKE scheme provides confidentiality to the encrypted message. The security of PKE is formally defined by the following security game.

Definition 6.7. (IND-CCA2 security of PKE). *Consider the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$.*

1. *$\mathcal{C}$ runs the PKE.KeyGen algorithm to obtain a key pair $(\mathsf{pk}, \mathsf{sk})$ and gives pk to the adversary.*

2. *$\mathcal{A}$ provides adaptively chosen ct and get back PKE.Dec(sk, ct).*

3. *$\mathcal{A}$ provides $\mathsf{msg}_0, \mathsf{msg}_1$ to $\mathcal{C}$.*

4. *$\mathcal{C}$ then runs PKE.Enc(pk) to obtain $\mathsf{ct}^* = \mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{msg}_b)$ for $b \xleftarrow{\$} \{0,1\}$. $\mathcal{C}$ provides ct^* to $\mathcal{A}$.*

5. *$\mathcal{A}$ continues to provide adaptively chosen ct and get back PKE.Dec(sk, ct), with a restriction that $\mathsf{ct} \neq \mathsf{ct}^*$.*

6. *$\mathcal{A}$ outputs its guess b'.*

The advantage of the adversary $\mathcal{A}$ in the above game is

$$\mathsf{Adv}_{\mathsf{pke}}(\mathcal{A}) := \Pr[b' = b] - \frac{1}{2}$$

A PKE scheme PKE is said to have indistinguishability security under adaptively chosen ciphertext attack if there is no polynomial time adversary $\mathcal{A}$ which can win the above game with probability non-negligible in λ.

We also require the PKE scheme to be "weakly robust" [ABN10]. Informally, a ciphertext when decrypted with an "incorrect" secret key should output $\bot$ when all the algorithms are honestly run.

Definition 6.8. ((Weak) robustness property of PKE). *A PKE scheme PKE has the (weak) robustness property if for all λ and $\mathsf{msg} \in \mathcal{M}$,*

$$\Pr\left[\mathsf{PKE.Dec}\Big(\mathsf{sk}', \mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{msg})\Big) \neq \bot\right] = \mathrm{negl}(\lambda)$$

where $(\mathsf{pk}, \mathsf{sk})$ and $(\mathsf{pk}', \mathsf{sk}')$ are generated by running PKE.KeyGen(1^λ) twice, and the probability is taken over the random coins of the probabilistic algorithms PKE.KeyGen, PKE.Enc.

One heuristic way of providing this property to a PKE scheme is by padding the message with 0^λ before encrypting it, and checking the suffix for 0^λ during decryption. We refer the readers to [ABN10] for a formal treatment of this property.

6.3.4 Collision resistant hash functions

A set of functions $\mathsf{H} = \{H_i\}$ is a collision resistant hash function family with each $H_i : \{0,1\}^{\mathrm{poly}(\lambda)} \to \{0,1\}^\lambda$ (for all $\mathrm{poly}(\lambda) > \lambda$), if for all λ, for every x in the domain of H, the value of

$$\Pr\left[H(x) = H(y) | H \leftarrow \mathsf{H.Gen}(1^\lambda), (x, y) \leftarrow \mathcal{A}(H)\right]$$

is $\mathrm{negl}(\lambda)$ for any polynomial time adversary $\mathcal{A}$, where the probability is taken over the random coins of Gen. In particular, we will use a function family which consists of functions with domain $\{0,1\}^{|\mathsf{ct_{enc}}|}$, where $\mathsf{ct_{enc}}$ is a ciphertext of a secret key encryption scheme which also depends on the length of the message encrypted.

7 Security analysis

7.1 Formal construction

We present here the formal description of our FE system using the syntax of the HW model from Definition 6.1. The trusted authority platform TA and decryption node platform DN each have access to instances of HW. Let PKE denote an IND-CCA2 secure public key encryption scheme (Definition 6.7) and let S denote an existentially unforgeable signature scheme (Definition 6.6).

Pre-processing phase TA and DN run $\mathsf{HW.Setup}(1^\lambda)$ for their HW instances and record the output params.

FE.Setup$^{\mathsf{HW}}(1^\lambda)$ The key manager enclave program Q_{KME} is defined as follows. The value tag_{DE}, the measurement of the program Q_{DE}, is hardcoded in the static data of Q_{KME}. Let state denote an internal state variable.

Q_{KME}:
- On input ("init", 1^λ):
    1. Run $(\mathsf{pk_{pke}}, \mathsf{sk_{pke}}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$ and $(\mathsf{vk_{sign}}, \mathsf{sk_{sign}}) \leftarrow \mathsf{S.KeyGen}(1^\lambda)$
    2. Update state to $(\mathsf{sk_{pke}}, \mathsf{sk_{sign}}, \mathsf{vk_{sign}})$ and output $(\mathsf{pk_{pke}}, \mathsf{vk_{sign}})$
- On input ("provision", quote, params):
    1. Parse $\mathsf{quote} = (\mathsf{md_{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \sigma)$, check that $\mathsf{tag}_Q = \mathsf{tag}_{DE}$. If not, output $\perp$.
    2. Parse $\mathsf{in} = ($"init setup"$, \mathsf{vk_{sign}})$ and check if $\mathsf{vk_{sign}}$ matches with the one in state. If not, output $\perp$.
    3. Parse $\mathsf{out} = (\mathsf{sid}, \mathsf{pk})$ and run $b \leftarrow \mathsf{HW.QuoteVerify}(\mathsf{params}, \mathsf{quote})$ on quote. If $b = 0$ output $\perp$.
    4. Retrieve $\mathsf{sk_{pke}}$ from state and compute $\mathsf{ct}_{sk} = \mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{sk_{pke}})$ and $\sigma_{sk} = \mathsf{S.Sign}(\mathsf{sk_{sign}}, (\mathsf{sid}, \mathsf{ct}_{sk}))$ and output $(\mathsf{sid}, \mathsf{ct}_{sk}, \sigma_{sk})$.
- On input ("sign", msg):
    Compute $\mathsf{sig} \leftarrow \mathsf{S.Sign}(\mathsf{sk_{sign}}, \mathsf{msg})$ and output sig.

Run $\mathsf{hdl}_{KME} \leftarrow \mathsf{HW.Load}(\mathsf{params}, Q_{KME})$ and $(\mathsf{pk_{pke}}, \mathsf{vk_{sign}}) \leftarrow \mathsf{HW.Run}(\mathsf{hdl}_{KME}, ($"init"$, 1^\lambda))$. Output the master public key $\mathsf{mpk} := (\mathsf{pk_{pke}}, \mathsf{vk_{sign}})$ and the master secret key $\mathsf{msk} := \mathsf{hdl}_{KME}$.

FE.Keygen$^{\mathsf{HW}}(\mathsf{msk}, P)$ Parse $\mathsf{msk} = \mathsf{hdl}_{KME}$ as a handle to HW.Run. Derive tag_P and call $\mathsf{sig} \leftarrow \mathsf{HW.Run}(\mathsf{hdl}_{KME}, ($"sign"$, \mathsf{tag}_P))$. Output $\mathsf{sk}_p := \mathsf{sig}$.

FE.Enc$(\mathsf{mpk}, \mathsf{msg})$ Parse $\mathsf{mpk} = (\mathsf{pk}, \mathsf{vk})$. Compute $\mathsf{ct} \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{msg})$ and output ct.

FE.DecSetup$^{\mathsf{HW},\mathsf{KM}(\cdot)}(\mathsf{sk}_P,\mathsf{ct})$ The decryption enclave program Q_{DE} is defined as follows. The security parameter λ is hardcoded into the program.

Q_{DE}:
- On input ("init setup", $\mathsf{vk}_{\mathsf{sign}}$):
    1. Run $(\mathsf{pk}_{\mathsf{ra}},\mathsf{sk}_{\mathsf{ra}}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$.
    2. Generate a session ID, $\mathsf{sid} \leftarrow \{0,1\}^\lambda$.
    3. Update state to $(\mathsf{sid},\mathsf{sk}_{\mathsf{ra}},\mathsf{vk}_{\mathsf{sign}})$, and output $(\mathsf{sid},\mathsf{pk}_{\mathsf{ra}})$.

- On input ("complete setup", $\mathsf{sid},\mathsf{ct}_{sk},\sigma_{sk}$):
    1. Look up the state to obtain the entry $(\mathsf{sid},\mathsf{sk}_{\mathsf{ra}},\mathsf{vk}_{\mathsf{sign}})$. If no entry exists for sid, output $\perp$.
    2. Verify the signature $b \leftarrow \mathsf{S.Verify}(\mathsf{vk}_{\mathsf{sign}},\sigma_{sk},(\mathsf{sid},\mathsf{ct}_{sk}))$. If $b=0$, output $\perp$.
    3. Run $m \leftarrow \mathsf{PKE.dec}(\mathsf{sk}_{\mathsf{ra}},\mathsf{ct}_{sk})$ and parse $m = (\mathsf{sk}_{\mathsf{pke}})$.
    4. Add the tuple $(\mathsf{sk}_{\mathsf{pke}},\mathsf{vk}_{\mathsf{sign}})$ to state[9].

- On input ("provision", $\mathsf{report},\mathsf{sig}$):
    1. Check to see that the setup has been completed, i.e. that state contains the tuple $(\mathsf{sk}_{\mathsf{pke}},\mathsf{vk}_{\mathsf{sign}})$. If not, output $\perp$.
    2. Check to see that the report has been verified, i.e. that state contains the tuple $(1,\mathsf{report})$. If not, output $\perp$.
    3. Parse $\mathsf{report} = (\mathsf{md}_{\mathsf{hdl}},\mathsf{tag}_Q,\mathsf{in},\mathsf{out},\mathsf{mac})$ and compute $b \leftarrow \mathsf{S.Verify}(\mathsf{vk}_{\mathsf{sign}},\mathsf{sig},\mathsf{tag}_Q)$. If $b=0$, output $\perp$.
    4. Parse out as $(\mathsf{sid},\mathsf{pk})$. If $b=1$ output $(\mathsf{sid},\mathsf{PKE.Enc}(\mathsf{pk},\mathsf{sk}_{\mathsf{pke}}))$. Else, output $\perp$.

Run $\mathsf{hdl}_{DE} \leftarrow \mathsf{HW.Load}(\mathsf{params},Q_{DE})$. Parse $\mathsf{mpk} = (\mathsf{sk}_{\mathsf{pke}},\mathsf{vk}_{\mathsf{sign}})$ and call $\mathsf{quote} \leftarrow \mathsf{HW.Run\&Quote}_{\mathsf{sk}_{\mathsf{HW}}}(\mathsf{hdl}_{DE},\text{"init setup"},\mathsf{vk}_{\mathsf{sign}})$. Query $\mathsf{KM}(\mathsf{quote})$, which internally runs $(\mathsf{sid},\mathsf{ct}_{sk},\sigma_{sk}) \leftarrow \mathsf{HW.Run}(\mathsf{hdl}_{KME},(\text{"provision"},\mathsf{quote},\mathsf{params}))$[10]. Call $\mathsf{HW.Run}(\mathsf{hdl}_{DE},(\text{"complete setup"},\mathsf{sid},\mathsf{ct}_{sk},\sigma_{sk}))$. Output hdl_{DE}.

FE.Dec$^{\mathsf{HW}(\cdot)}(\mathsf{hdl},\mathsf{sk}_P,\mathsf{ct})$ Define a function enclave program parameterized by P.

$Q_{FE}(P)$:
- On input ("init"):
    1. Run $(\mathsf{pk}_{\mathsf{la}},\mathsf{sk}_{\mathsf{la}}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$.
    2. Generate a session ID, $\mathsf{sid} \leftarrow \{0,1\}^\lambda$.
    3. Update state to $(\mathsf{sid},\mathsf{sk}_{\mathsf{la}})$, and output $(\mathsf{sid},\mathsf{pk}_{\mathsf{la}})$.

- On input ("run", $\mathsf{report}_{sk},\mathsf{ct}_{msg}$):
    1. Check to see that the report has been verified, i.e. that state contains the tuple $(1,\mathsf{report}_{sk})$. If not, output $\perp$.
    2. Parse $\mathsf{report}_{sk} = (\mathsf{md}_{\mathsf{hdl}},\mathsf{tag}_Q,\mathsf{in},\mathsf{out},\mathsf{mac})$. Parse out as $(\mathsf{sid},\mathsf{ct}_{key})$.
    3. Look up the state to obtain the entry $(\mathsf{sid},\mathsf{sk}_{\mathsf{la}})$. If no entry exists for sid, output $\perp$.
    4. Compute $\mathsf{sk}_{\mathsf{pke}} \leftarrow \mathsf{PKE.dec}(\mathsf{sk}_{\mathsf{ra}},\mathsf{ct}_{key})$ and $x \leftarrow \mathsf{PKE.dec}(\mathsf{sk}_{\mathsf{pke}},\mathsf{ct}_{msg})$.
    5. Run P on x and record the output $\mathsf{out} := P(x)$. Output out.

Set $\mathsf{hdl}_{DE} = \mathsf{hdl}$, $\mathsf{sig} = \mathsf{sk}_P$, and $\mathsf{ct}_{msg} = \mathsf{ct}$. Run $\mathsf{hdl}_P \leftarrow \mathsf{HW.Load}(\mathsf{params},Q_{FE}(P))$ and call $\mathsf{report} \leftarrow \mathsf{HW.Run\&Report}_{\mathsf{sk}_{\mathsf{report}}}(\mathsf{hdl}_P,\text{"init"})$. Run $\mathsf{HW.ReportVerify}_{\mathsf{sk}_{\mathsf{report}}}(\mathsf{hdl}_{DE},\mathsf{report})$ and then call $\mathsf{report}_{sk} \leftarrow \mathsf{HW.Run\&Report}(\mathsf{hdl}_{DE},(\text{"provision"},\mathsf{report},\mathsf{sig}))$. Finally, run $\mathsf{HW.ReportVerify}_{\mathsf{sk}_{\mathsf{report}}}(\mathsf{hdl}_P,\mathsf{report}_{sk})$ and call $\mathsf{out} \leftarrow \mathsf{HW.Run}(\mathsf{hdl}_P,\text{"run"},\mathsf{report}_{sk},\mathsf{ct}_{msg})$. Output out.

---

[9]$\mathsf{vk}_{\mathsf{sign}}$ is already in state as part of the outputs of the previous "init setup" phase, but it is useful store and use this tuple as result of a successfully completed setup.

[10]We could use $\mathsf{HW.Run\&Quote}$ here instead of explicitly creating the signature σ_k. If we do that, the verification step in DE would involve using the Intel Attestation Service.

7.2 Security proof

Overview We explain the crux of our security proof here.

We construct a simulator $\mathcal{S}$ which can simulate HW, KM, FE.Keygen oracles and simulate the challenge ciphertext for the challenge message msg^* provided by the adversary $\mathcal{A}$. The only information that $\mathcal{S}$ will get about msg^* other than its length is the access to the U_{msg^*} oracle which reveals $P(\mathsf{msg}^*)$ for the P's queried by $\mathcal{A}$ to FE.Keygen. At a high level, the proof idea is simple: $\mathcal{S}$ encrypts zeros as the the challenge ciphertext ct^* and FE.Keygen is simulated honestly. In the ideal experiment, $\mathcal{S}$ intercepts $\mathcal{A}$'s queries to HW and provides simulated responses. It can use its U_{msg^*} oracle to get $P(\mathsf{msg}^*)$ and simply send this back to $\mathcal{A}$ as the simulated HW output. If $\mathcal{A}$ queries HW on any ciphertexts that do not match the challenge ciphertext ct^*, $\mathcal{S}$ can decrypt them honestly since it possesses msk. Since $\mathcal{S}$ has to modify the program descriptions in enclaves, we provide $\mathcal{S}$ access to the HW keys $\mathsf{sk}_{\mathsf{report}}$ and $\mathsf{sk}_{\mathsf{quote}}$ to produce reports and quotes.

Despite the apparent simplicity, the following subtleties make the proof of security more challenging than on first sight:

1. The simple proof sketch does not account for all of $\mathcal{A}$'s interaction with HW between sending ct^* and receiving back $P(\mathsf{msg}^*)$. HW communicates through $\mathcal{A}$ as a proxy. $\mathcal{A}$ might even tamper with these intermediate messages and observe how HW responds. We need to ensure that anything $\mathcal{A}$ observes in the real experiment can be simulated in the ideal experiment.

2. We use IND-CCA2 public key encryption to secure communication between enclaves that is intercepted by $\mathcal{A}$. $\mathcal{S}$ will need to simulate this communication. Proving that $\mathcal{A}$ cannot distinguish this involves a reduction to the IND-CCA2 security game, showing that if $\mathcal{A}$ can distinguish the real and simulated communication then it would break the IND-CCA2 security. The IND-CCA2 adversary will need to simulate the entire FE system for $\mathcal{A}$ without knowledge of the corresponding secret keys for the public keys that the enclaves are using to secure their communication. In particular, it must see if $\mathcal{A}$ tampers with messages in a way that would cause the system to abort). This is what necessitates an extra layer of authentication on the communication between enclaves

3. The final challenge is that the adversary can also load modified programs of its choice into different enclaves and test their behavior with honest or tampered inputs. This aspect in particular makes the security proof challenging because the FE simulator in the ideal world has to identify whether honest attested programs are running inside the enclaves, and produce simulated outputs only for those enclaves. This gets tricky as there are three enclaves each with multiple entry points.

Theorem 7.1. *If S is an EUF-CMA secure signature scheme, PKE is an IND-CCA2 secure public key encryption scheme and HW is a secure hardware scheme, then FE is a secure functional encryption scheme according to Definition 6.4.*

Proof. We will construct a simulator $\mathcal{S}$ for the FE security game in Definition 6.4. $\mathcal{S}$ is given the length $|\mathsf{msg}^*|$ and an oracle access to $U_{\mathsf{msg}^*}(\cdot)$ (such that $U_{\mathsf{msg}^*}(P) = P(\mathsf{msg}^*)$) after the adversary provides its challenge message msg^*. $\mathcal{S}$ can use this U_{msg^*} oracle on the programs queried by the adversary $\mathcal{A}$ to FE.Keygen. $\mathcal{S}$ has to simulate the pre-processing phase and a ciphertext corresponding to the challenge message msg^* along with answering the adversary's queries to the KeyGen, HW and the KM oracles.

Pre-processing phase: $\mathcal{S}$ simulates the pre-processing phase similar to the real world. $\mathcal{S}$ runs HW.Setup(1^λ) and records $(\mathsf{sk}_{\mathsf{quote}}, \mathsf{sk}_{\mathsf{report}})$ generated during the process. $\mathcal{S}$ measures and stores tag_{DE}. $\mathcal{S}$ also creates empty lists $\mathcal{K}, \mathcal{R}, \mathcal{N}, L_{KM}, L_{DE}, L_{DE2}, L_{FE}$ which will be used later.

FE.Keygen*(msk, P) When $\mathcal{A}$ makes a query to the FE.Keygen oracle, $\mathcal{S}$ responds the same way as in the real world except that $\mathcal{S}$ now stores all the tag_P corresponding to the P's queried in a list $\mathcal{K}$. That is, $\mathcal{S}$ does the following.

1. Parse msk as a handle to HW.Run. Derive tag_P and call $\mathsf{sig} \leftarrow \mathsf{HW.Run}(\mathsf{hdl}_{KME}, (\text{``sign''}, \mathsf{tag}_P))$. Output $\mathsf{sk}_p := \mathsf{sig}$.

2. If tag_P has an entry in $\mathcal{K}$, make the first entry 1. Else, add $(1, \mathsf{tag}_P, \{\})$ to $\mathcal{K}$. We call the first bit in the tuple as the "honest" bit.

FE.Enc*$(\mathsf{mpk}, 1^{|\mathsf{msg}^*|})$ When $\mathcal{A}$ provides the challenge message msg^*, the following algorithm is used by $\mathcal{S}$ to simulate the challenge ciphertext for msg^*.

1. Parse $\mathsf{mpk} = (\mathsf{pk}, \mathsf{vk})$. Compute and output $\mathsf{ct}^* \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}, 0^{|\mathsf{msg}^*|})$.

2. Store ct^* in the list $\mathcal{R}$.

HW oracle   For $\mathcal{A}$'s queries to the algorithms of the HW oracle, $\mathcal{S}$ runs the corresponding HW algorithms honestly and outputs their results except for the following oracle calls.

- HW.Run(hdl_{KME}, "provision", quote, params): When a provision query is made to KME, $\mathcal{S}$ parses $\mathsf{quote} = (\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \sigma)$ and outputs $\perp$ if $\mathsf{out} \notin L_{DE2}$. Else, it honestly runs the HW algorithm and then replaces ct_{sk} with $\mathsf{PKE.Enc}(\mathsf{pk}, 0^{|\mathsf{sk}_{\mathsf{pke}}|})$. $\mathcal{S}$ also generates and replaces σ_{sk} for the modified ct_{sk}. Finally, $\mathcal{S}$ stores $(\mathsf{sid}, \mathsf{ct}_{sk})$ in L_{KM}. $\mathcal{S}$ does the same for HW.Run&Report and HW.Run&Quote with $(\mathsf{hdl}_{KME}, \text{``provision''}, \mathsf{quote}, \mathsf{params})$ as input.

- HW.Load(params, Q): When the load algorithm is run for a Q corresponding to that of a DE, $\mathcal{S}$ runs the load algorithm honestly and outputs hdl_{DE}. In addition, it stores hdl_{DE} in the list $\mathcal{D}$. When the load algorithm is run for a Q of the form $Q_{FE}(P)$, $\mathcal{S}$ adds the output handle hdl_P to the list $\mathcal{K}$ as follows. $\mathcal{S}$ first checks if the tag_P corresponding to this has an entry in $\mathcal{K}$, and if it exists $\mathcal{S}$ appends hdl_P to its handle list. Else, $\mathcal{S}$ adds the tuple $(0, \mathsf{tag}_P, \mathsf{hdl}_P)$ to $\mathcal{K}$.

- HW.Run(hdl_{DE}, "init setup", $\mathsf{vk}_{\mathsf{sign}}$): When an init setup query is made to a $\mathsf{hdl}_{DE} \in \mathcal{D}$, $\mathcal{S}$ checks if $\mathsf{vk}_{\mathsf{sign}}$ matches with the one in mpk. Else, it removes hdl_{DE} from $\mathcal{D}$. $\mathcal{D}$ will remain as the list of handles for DEs with the correct $\mathsf{vk}_{\mathsf{sign}}$ fed as input. Then, $\mathcal{S}$ runs HW.Run honestly on the given input and outputs the result. It also adds $(\mathsf{sid}, \mathsf{pk}_{\mathsf{ra}})$ to the list L_{DE2}.

- HW.Run(hdl_{DE}, "complete setup", $\mathsf{sid}, \mathsf{ct}_{sk}, \sigma_{sk}$): When a complete setup query is made to a $\mathsf{hdl}_{DE} \in \mathcal{D}$, $\mathcal{S}$ output $\perp$ if $(\mathsf{sid}, \mathsf{ct}_{sk}) \notin L_{KM}$. Else, it honestly executes HW.Run. Similar changes are made for HW.Run&Report and HW.Run&Quote on this set of inputs.

- HW.Run(hdl_{DE}, "provision", $\mathsf{report}, \mathsf{sig}$): When a provision query is made to a $\mathsf{hdl}_{DE} \in \mathcal{D}$, $\mathcal{S}$ parses $\mathsf{report} = (\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \mathsf{mac})$ and outputs $\perp$ if $\mathsf{out} \notin L_{FE}$. Else, it honestly executes HW.Run. At the end, $\mathcal{S}$ adds the output $(\mathsf{sid}, \mathsf{ct}_{key})$ to L_{DE}. Similar changes are made for HW.Run&Report and HW.Run&Quote on this set of inputs.

- HW.Run(hdl_P, "init"): When an init query is made to a $\mathsf{hdl}_P \in \mathcal{K}$ whose tuple in $\mathcal{K}$ has the honest bit set, $\mathcal{S}$ runs HW.Run&Report honestly and outputs the result. It also adds $(\mathsf{sid}, \mathsf{pk}_{\mathsf{la}})$ to the list L_{FE}. Similar changes are made for HW.Run&Report and HW.Run&Quote on this set of inputs.

- HW.Run(hdl_P, "run", $\mathsf{report}_{sk}, \mathsf{ct}_{msg}$): When a run query is made to $\mathsf{hdl}_P \in \mathcal{K}$ whose tuple in $\mathcal{K}$ has the honest bit set, $\mathcal{S}$ first parses $\mathsf{report}_{sk} = (\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \mathsf{mac})$ and outputs $\perp$ if $\mathsf{out} \notin L_{DE}$. Else, it runs HW.Run on the given inputs. If the output is $\perp$, $\mathcal{S}$ outputs $\perp$. Else, it parses out as $(\mathsf{sid}, \mathsf{ct}_{key})$ and retrieves $\mathsf{sk}_{\mathsf{pke}}$ from msk. If $\mathsf{ct}_{msg} \notin \mathcal{R}$, $\mathcal{S}$ computes $x \leftarrow \mathsf{PKE.dec}(\mathsf{sk}_{\mathsf{pke}}, \mathsf{ct}_{msg})$, runs P on x and outputs $\mathsf{out} := P(x)$. If $\mathsf{ct}_{msg} \in \mathcal{R}$, $\mathcal{S}$ queries its U_{msg^*} oracle on P and outputs the response. Similar changes are made for HW.Run&Report and HW.Run&Quote on this set of inputs.

- For HW.Run&Report and HW.Run&Quote queries, even when changes are made according to the above changes, report and quote are generated for unmodified tag's of the unmodified programs descriptions. (This is to prevent the adversary from being able to distinguish the change in hybrids just by looking at the report or quote.)

KM oracle For $\mathcal{A}$'s queries to the KM oracle with input quote, $\mathcal{S}$ uses the provision queries to HW.Run for KME with the changes mentioned above.

Now, for this polynomial time simulator $\mathcal{S}$ described above, we will show that for experiments in Definition 6.4,

$$(\mathsf{msg}, \alpha)_{\mathsf{real}} \overset{c}{\approx} (\mathsf{msg}, \alpha)_{\mathsf{ideal}} \tag{1}$$

We prove this by showing that the view of the adversary $\mathcal{A}$ in the real world is computationally indistinguishable from its view in the ideal world. It can be easily checked that the algorithms KeyGen*, Enc* and oracle KM* simulated by S correspond to the ideal world specifications of Definition 6.4 (because the only information that $\mathcal{S}$ obtains about msg* is through the $U_{\mathsf{msg}^*}(\cdot)$ oracle which it queries on the FE.Keygen queries made by $\mathcal{A}$). We will prove through a series of hybrids that $\mathcal{A}$ cannot distinguish between the real and the ideal world algorithms and oracles.

$\boxed{\textbf{Hybrid } 0}$ $\mathsf{Exp}_{\mathbf{FE}}^{\mathsf{real}}(1^\lambda)$ is run.

$\boxed{\textbf{Hybrid } 1}$ As in **Hybrid** 0, except that FE.Keygen* run by $\mathcal{S}$ is used to generate secret keys instead of FE.Keygen. Also, the ct* returned by FE.Enc for the encryption of the challenge message msg* is stored in the list $\mathcal{R}$. Also, when HW.Load(params, Q) is run for the Q of a DE, store the output in the list $\mathcal{D}$, and when HW.Run(hdl_{DE}, "init setup", $\mathsf{vk}_{\mathsf{sign}}$) is run with a $\mathsf{vk}_{\mathsf{sign}}$ different from that in mpk, remove hdl_{DE} from $\mathcal{D}$. Also, when HW.Load is run for a Q of the form $Q_{FE}(P)$, the output handle hdl_P is added to the list $\mathcal{K}$ in the tuple corresponding to tag_P. If tag_P does not have an entry in $\mathcal{K}$, the entire tuple $(0, \mathsf{tag}_P, \mathsf{hdl}_P)$ is added to $\mathcal{K}$.

Here, FE.Keygen* and FE.Keygen are identical. And storing in lists does not affect the view of $\mathcal{A}$. Hence, **Hybrid** 1 is indistinguishable from **Hybrid** 0.

$\boxed{\textbf{Hybrid } 2}$ As in **Hybrid** 1, except that when the HW.Run&Report is queried with $(\mathsf{hdl}_{DE}, (\text{"provision"}, \mathsf{report}, \mathsf{sig}))$ for $\mathsf{hdl}_{DE} \in \mathcal{D}$, $\mathcal{S}$ outputs $\bot$ if tag_P that is part of report does not have an entry in $\mathcal{K}$ with the honest bit set.

If sig is not a valid signature of tag_P, then the S.Verify step during the execution of HW.Run&Report($\mathsf{hdl}_{DE}, \cdot$) would make it output $\bot$. Hence, **Hybrid** 2 differs from **Hybrid** 1 only when a valid signature sig for tag_P is part of the "provision" query to HW.Run&Report($\mathsf{hdl}_{DE}, \cdot$) with a hdl_{DE} that has the correct $\mathsf{vk}_{\mathsf{sign}}$ in its state and with a P that $\mathcal{A}$ has not queried to FE.Keygen*. But, if $\mathcal{A}$ does make a query of this kind to HW.Run&Report with a valid sig, we will show that this can be used to break the existential unforgeability of the signature scheme S.

Lemma 7.2. *If the signature scheme* S *is existentially unforgeable as in Definition 6.6, then* **Hybrid** 2 *is indistinguishable from* **Hybrid** 1.

Proof. Let $\mathcal{A}$ be an adversary which distinguishes between **Hybrid** 1 and **Hybrid** 2. We will use it to break the EUF-CMA security of S. We will get a verification key $\mathsf{vk}_{\mathsf{sign}}^*$ and an access to S.Sign($\mathsf{sk}_{\mathsf{sign}}^*, \cdot$) oracle from the EUF-CMA challenger. $\mathcal{S}$ sets this $\mathsf{vk}_{\mathsf{sign}}^*$ as part of the mpk. Whenever $\mathcal{S}$ has to sign a message using $\mathsf{sk}_{\mathsf{sign}}^*$, it uses the S.Sign($\mathsf{sk}_{\mathsf{sign}}^*, \cdot$) oracle. Also, our construction does not ever need a direct access to $\mathsf{sk}_{\mathsf{sign}}^*$; it is used only to sign messages for which the oracle provided by the challenger can be used. Now, if $\mathcal{A}$ can distinguish between the two hybrids, as we argued earlier, it is only because $\mathcal{A}$ makes a "provision" query to the HW.Run&Report($\mathsf{hdl}_{DE}, \cdot$) oracle with a $\mathsf{hdl}_{DE} \in \mathcal{D}$ that has $\mathsf{vk}_{\mathsf{sign}}^*$ in its state and with a valid signature sig on a $\mathsf{tag}_P \notin \mathcal{K}$. We will output $(\mathsf{tag}_P, \mathsf{sig})$ as our forgery to the EUF-CMA challenger. $\qquad\square$

$\boxed{\textbf{Hybrid } 3.0}$ As in **Hybrid** 2, except that $\mathcal{S}$ maintains a list L_{KM} of all the "provision" query responses from KM i.e., the $(\mathsf{sid}, \mathsf{ct}_{sk})$ tuples. Then, on any call to HW.Run(hdl_{DE}, "complete setup", $\mathsf{sid}, \mathsf{ct}_k, \sigma_k$) for $\mathsf{hdl}_{DE} \in \mathcal{D}$, if $(\mathsf{sid}, \mathsf{ct}_{sk}) \notin L_{KM}$, $\mathcal{S}$ outputs $\bot$.

The proof at a high level will be similar to the previous one. $\mathsf{HW.Run}(\mathsf{hdl}_{DE}, \text{"complete setup"}, \cdot)$ already outputs $\perp$ in **Hybrid** 2 if σ_{sk} is not a valid signature of $(\mathsf{sid}, \mathsf{ct}_{sk})$ or if an entry for the session ID sid is not in state. So, **Hybrid** 3.0 differs from **Hybrid** 2 only when $\mathcal{A}$ can produce a valid signature σ_{sk} on a $(\mathsf{sid}, \mathsf{ct}_{sk})$ pair for a sid which it has seen before in the communication between KM and a DE whose handle is in $\mathcal{D}$.

Lemma 7.3. *If the signature scheme S is existentially unforgeable as in Definition 6.6, then* **Hybrid** *3.0 is indistinguishable from* **Hybrid** *2.*

Proof. Let $\mathcal{A}$ be an adversary which distinguishes between **Hybrid** 2 and **Hybrid** 3.0. We will use it to break the EUF-CMA security of S. We will get a verification key $\mathsf{vk}^*_{\mathsf{sign}}$ and an access to $\mathsf{S.Sign}(\mathsf{sk}^*_{\mathsf{sign}}, \cdot)$ oracle from the EUF-CMA challenger. $\mathcal{S}$ sets this $\mathsf{vk}^*_{\mathsf{sign}}$ as part of the mpk. Whenever $\mathcal{S}$ has to sign a message with $\mathsf{sk}^*_{\mathsf{sign}}$, it uses the $\mathsf{S.Sign}(\mathsf{sk}^*_{\mathsf{sign}}, \cdot)$ oracle. As mentioned in the proof of Lemma 7.2, $\mathcal{S}$ never needs a direct access to $\mathsf{sk}^*_{\mathsf{sign}}$. Now, if $\mathcal{A}$ can distinguish between the two hybrids, as we argued earlier, it is only because $\mathcal{A}$ makes a "complete setup" query to the $\mathsf{HW.Run}(\mathsf{hdl}_{DE}, \cdot)$ oracle with a valid signature σ_{sk} for $(\mathsf{sid}, \mathsf{ct}_{sk}) \notin L_{KM}$ but sid has an entry in state. Also, $\mathsf{hdl}_{DE} \in \mathcal{D}$ and hence has $\mathsf{vk}^*_{\mathsf{sign}}$ in its state. We will output $((\mathsf{sid}, \mathsf{ct}_{sk}), \sigma_{sk})$ as our forgery to the EUF-CMA challenger. $\square$

Hybrid 3.1 As in **Hybrid** 3.0, except that $\mathcal{S}$ maintains a list L_{DE} of all the "provision" query responses from $\mathsf{hdl}_{DE} \in \mathcal{D}$ i.e., the $(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_{Q_{DE}}, (\mathsf{report}, \mathsf{sig}), (\mathsf{sid}, \mathsf{ct}_{key}))$ tuples. And, on call to $\mathsf{HW.Run}(\mathsf{hdl}_P, \mathsf{report}_{sk}, \mathsf{ct}_{msg})$ with hdl_P having an entry in $\mathcal{K}$ with its honest bit set, $\mathcal{S}$ outputs $\perp$ if $\mathsf{report}_{sk} = (\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, (\mathsf{sid}, \mathsf{ct}_{key}), \mathsf{mac})$ with $\mathsf{tag}_Q = \mathsf{tag}_{DE}$, sid having an entry in state and $(\mathsf{sid}, \mathsf{ct}_{key}) \notin L_{DE}$.

The security of local attestation is used to prove the indistinguishability of the hybrids. For honest hdl_Ps, $\mathsf{HW.Run}(\mathsf{hdl}_P, \mathsf{report}_{sk}, \mathsf{ct}_{msg})$ already outputs $\perp$ in **Hybrid** 3.0 if for $\mathsf{report}_{sk} = (\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_{DE}, (\mathsf{report}, \mathsf{sig}), (\mathsf{sid}, \mathsf{ct}_{key}), \mathsf{mac})$, mac is not a valid MAC on $(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_{DE}, (\mathsf{report}, \mathsf{sig}), (\mathsf{sid}, \mathsf{ct}_{key}))$, or if sid does not have an entry in state. So, the only change in **Hybrid** 3.1 is that $\mathsf{HW.Run}$ also outputs $\perp$ if mac is a valid MAC but on a $(\mathsf{sid}, \mathsf{ct}_{key}) \notin L_{DE}$. Hence, $\mathcal{A}$ can distinguish between the hybrids only when it produces a valid mac on a tuple with $(\mathsf{sid}, \mathsf{ct}_{sk})$ not in L_{DE}. But this happens with negligible probability due to the security of Local Attestation.

Lemma 7.4. *If the Local Attestation process of HW is secure as in Definition 6.2, then* **Hybrid** *3.1 is indistinguishable from* **Hybrid** *3.0.*

The proof of this lemma is similar to Lemma 7.3, since $\mathsf{sk}_{\mathsf{report}}$ is not used by $\mathcal{S}$ other than to produce a report.

Hybrid 4 As in **Hybrid** 3.1, except that when $\mathsf{HW.Run}$ is queried with $(\mathsf{hdl}_P, \text{"run"}, \mathsf{report}_{sk}, \mathsf{ct}_{msg})$ where report_{sk} is a valid MAC of a tuple containing an entry in L_{DE} and $\mathsf{hdl}_P \in \mathcal{K}$ with the honest bit set. If $\mathsf{ct}_{msg} \in \mathcal{R}$, $\mathcal{S}$ uses the U_{msg^*} oracle to answer the $\mathsf{HW.Run}$ query. If $\mathsf{ct}_{msg} \notin \mathcal{R}$, $\mathcal{S}$ uses the $\mathsf{sk}_{\mathsf{pke}}$ from $\mathsf{FE.Setup}$ to decrypt ct_{msg} instead of the one got by decrypting ct_{key} i.e.,

- On input $(\text{"run"}, \mathsf{report}_{sk}, \mathsf{ct}_{msg})$:

  4. If $\mathsf{ct}_{msg} \notin \mathcal{R}$, retrieve $\mathsf{sk}_{\mathsf{pke}}$ from msk. Compute $x \leftarrow \mathsf{PKE.dec}(\mathsf{sk}_{\mathsf{pke}}, \mathsf{ct}_{msg})$. Run P on x and record the output $\mathsf{out} := P(x)$. Output out.

  5. If $\mathsf{ct}_{msg} \in \mathcal{R}$, query $U_{\mathsf{msg}^*}(P)$ and output the response.

In **Hybrid** 3.1, the decryption of ct_{key} is used by $\mathcal{S}$ to decrypt ct_{msg} while running $\mathsf{HW.Run}(\mathsf{hdl}_P, \cdot)$. This ct_{key} is a valid encryption of $\mathsf{sk}_{\mathsf{pke}}$ because **Hybrid** 3.0 and **Hybrid** 3.1 ensure that the encryption of $\mathsf{sk}_{\mathsf{pke}}$ sent from KME to DE and then the one from DE to FE both reach FE unmodified. Hence, the $\mathsf{sk}_{\mathsf{pke}}$ got by decrypting ct_{msg} is same as the one from msk. Thus, **Hybrid** 4 is indistinguishable from **Hybrid** 3.1 for any $\mathsf{ct}_{msg} \notin \mathcal{R}$. Now, let us consider the case of $\mathsf{ct}_{msg} \in \mathcal{R}$. $\mathcal{S}$ has the restriction that it can use the U_{msg^*}

oracle only for a P for which $\mathsf{tag}_P \in \mathcal{K}$. From **Hybrid** 3.1, we know that $\mathsf{HW.Run}(\mathsf{hdl}_P, \cdot)$ does not output $\perp$ only when run with a valid $\mathsf{report}_{sk} = (\mathsf{md}_\mathsf{hdl}, \mathsf{tag}_{DE}, (\mathsf{report}, \mathsf{sig}), (\mathsf{sid}, \mathsf{ct}_{key}), \mathsf{mac})$ which is output by a DE "provision" query. Hence, sig is a valid signature of the tag_P contained in report. Also, $\mathsf{tag}_P \in \mathcal{K}$ with the honest bit set, as ensured in **Hybrid** 2. So, when a $\mathsf{HW.Run}$ "run" query is made for hdl_P, $\mathcal{S}$ is allowed use its U_{msg^*} oracle to output the $\mathsf{FE.Dec}$ result. Thus, **Hybrid** 4 is indistinguishable from **Hybrid** 3.1 for any ct_{msg}.

The following set of hybrids will help $\mathcal{S}$ replace an encryption of sk_pke with an encryption of zeros. In order to prove the indistinguishability, we will argue that all the FE algorithms run independent of the sk_pke encrypted in ct_{sk}, and that $\mathcal{A}$ does not get any information about the value encrypted in ct_{sk}.

$\boxed{\textbf{Hybrid } 5.0}$ As in **Hybrid** 4, except that $\mathcal{S}$ maintains a list L_{DE2} of all $(\mathsf{sid}, \mathsf{pk}_\mathsf{ra})$ that are part of $\mathsf{quote} = (\mathsf{md}_\mathsf{hdl}, \mathsf{tag}_{DE}, \text{"init setup"}, (\mathsf{sid}, \mathsf{pk}_\mathsf{ra}), \sigma)$ output by $\mathsf{HW.Run\&Quote}(\mathsf{hdl}_{DE}, \text{"init setup"}, \cdot)$ for $\mathsf{hdl}_{DE} \in \mathcal{D}$. And, when $\mathsf{HW.Run}(\mathsf{hdl}_{KME}, \text{"provision"}, \mathsf{quote}, \mathsf{params})$ is called $\mathcal{S}$ outputs $\perp$ when $(\mathsf{sid}, \mathsf{pk}_\mathsf{ra}) \notin L_{DE2}$.

The Remote Attestation security ensures that $\mathcal{A}$ can provide a fake quote on a pk_ra not provided by DE only with negligible probability.

Lemma 7.5. *If Remote Attestation is secure as in Definition 6.3, then* **Hybrid** *5.0 is indistinguishable from* **Hybrid** *4.*

The proof of this lemma is similar to Lemma 7.3 since $\mathsf{sk}_\mathsf{quote}$ is not used by $\mathcal{S}$ except for producing a quote. And, this lemma ensures that KME provides an encryption of sk_pke only under a public key pk_ra generated inside $Q_{DE} \in \mathcal{D}$ i.e., when $\mathsf{HW.Run}(\mathsf{hdl}_{KME}, \text{" provision"}, \mathsf{quote}, \mathsf{params})$ is called with a valid quote output by a valid instance of DE.

$\boxed{\textbf{Hybrid } 5.1}$ As in **Hybrid** 5.0, except that $\mathcal{S}$ maintains a list L_{FE} of all $(\mathsf{sid}, \mathsf{pk}_\mathsf{la})$ that are part of $\mathsf{report} = (\mathsf{md}_\mathsf{hdl}, \mathsf{tag}_P, (\text{"init"}, \mathsf{sid}, \mathsf{pk}_\mathsf{la}), \mathsf{mac})$ output by $\mathsf{HW.Run\&Report}(\mathsf{hdl}_P, \text{"init"}, \cdot)$ for $\mathsf{hdl}_P \in \mathcal{K}$ with the honest bit set. And when $\mathsf{HW.Run\&Report}(\mathsf{hdl}_{DE}, \text{"provision"}, \mathsf{report}, \mathsf{sig})$ is called for a $\mathsf{hdl}_{DE} \in \mathcal{D}$, $\mathcal{S}$ outputs $\perp$ when report contains $\mathsf{tag}_P \in \mathcal{K}$ but $(\mathsf{sid}, \mathsf{pk}_\mathsf{la}) \notin L_{FE}$.

This is ensured by the Local Attestation security. And, this shows that Q_{DE} only outputs sk_pke encrypted under some pk_la that was generated by a $Q_{FE}(\mathsf{hdl}_P, \cdot)$ running a program P that has been queried to $\mathsf{FE.Keygen}$.

Lemma 7.6. *If Local Attestation is secure as in Definition 6.2, then* **Hybrid** *5.1 is indistinguishable from* **Hybrid** *5.0.*

The proof of this lemma is again similar to Lemma 7.3 since $\mathsf{sk}_\mathsf{report}$ is not used by $\mathcal{S}$ except for producing a report.

$\boxed{\textbf{Hybrid } 5.2}$ As in **Hybrid** 5.1, except that when the KM oracle calls $\mathsf{HW.Run}(\mathsf{hdl}_{KME}, (\text{"provision"}, \cdot, \cdot))$, $\mathcal{S}$ replaces ct_{sk} in the output with $\mathsf{PKE.Enc}(0^{|\mathsf{sk}_\mathsf{pke}|})$.

Lemma 7.5 and Lemma 7.6 ensure that sk_pke is encrypted only under pk_ra and pk_la generated by valid enclaves and $\mathcal{A}$ has no access to the corresponding secret keys. Now we will use the IND-CCA2 security game[11] to argue that $\mathcal{A}$ cannot distinguish whether ct_{sk} has an encryption of zeros or sk_pke under pk_ra of the DE, and whether ct_{key} is an encryption of zeros or sk_pke under pk_la of a valid FE.

Lemma 7.7. *If* PKE *is an IND-CCA2 secure encryption scheme, then* **Hybrid** *5.2 is indistinguishable from* **Hybrid** *5.1.*

---

[11] Actually, IND-CPA security of PKE is enough here.

Proof. We will run two IND-CCA2 games in parallel, one for ct_{sk} and another for ct_{key}. It can be easily shown that this variant is equivalent to the regular IND-CCA2 security game. The IND-CCA2 challenger provides two challenge public keys pk_1^* and pk_2^*. $\mathcal{S}$ sets $\mathsf{pk}_{\mathsf{ra}} = \mathsf{pk}_1^*$ and $\mathsf{pk}_{\mathsf{la}} = \mathsf{pk}_2^*$. Now, $\left\{\mathsf{sk}_{\mathsf{pke}}, 0^{|\mathsf{sk}_{\mathsf{pke}}|}\right\}$ is provided as the challenge message pair for both the games. The challenger returns ct_1^* and ct_2^*, which are encryptions of either the left messages or the right messages from the each pair. Note that we use the same challenge bit for both the games. $\mathcal{S}$ sets $\mathsf{ct}_{sk} = \mathsf{ct}_1^*$ and $\mathsf{ct}_{key} = \mathsf{ct}_2^*$.

Now we argue that when the left messages are encrypted, the view of $\mathcal{A}$ is equivalent to **Hybrid** 5.1, and when the right messages are encrypted, the view is equivalent to **Hybrid** 5.2. This is because the other information that $\mathcal{A}$ gets do not depend on the value encoded in ct_{sk} or ct_{key}. We argue this as follows. We have already established that $\mathcal{A}$ only gets ct_{sk} encrypted with a $\mathsf{pk}_{\mathsf{ra}}$ generated in DE from KME. Similarly, $\mathcal{A}$ only gets ct_{key} encrypted with a $\mathsf{pk}_{\mathsf{la}}$ generated in a valid FE from DE. In addition to these, when interacting with messages from a valid Q_{DE} or $Q_{FE}(\cdot)$, $\mathcal{S}$ either uses the $\mathsf{sk}_{\mathsf{pke}}$ from msk or the U_{msg} oracle to answer the queries and not the decryption of ct_{key}.

Hence, when $\mathcal{A}$ decides between the two hybrids we forward the corresponding answer to the IND-CCA2 challenger. If $\mathcal{A}$ can distinguish between these two hybrids with non-negligible probability, then the IND-CCA2 security of PKE can be broken with non-negligible probability. □

Hybrid 6 As in **Hybrid** 5.2, except that FE.Enc* is used instead of FE.Enc.

We are now ready to use the IND-CCA2 security property of PKE to replace ct_{msg} which was an encryption of msg) with an encryption of zeros.

Lemma 7.8. *If* PKE *is an IND-CCA2 secure encryption scheme, then* **Hybrid** 6 *is indistinguishable from* **Hybrid** 5.2.

Proof. The IND-CCA2 challenger provides the challenge public key pk^*. During FE.Setup $\mathcal{S}$ sets $\mathsf{pk}_{\mathsf{pke}} = \mathsf{pk}^*$. Now, msg and $0^{|\mathsf{msg}|}$ are provided as the challenge messages. The challenger returns ct^*, which is an encryption of either of those with equal probability. $\mathcal{S}$ sets $\mathsf{ct}_{msg} = \mathsf{ct}^*$. When HW.Run($\mathsf{hdl}_P$, "run", report_{sk}, ct_{msg}) is called with a valid report_{sk} to $\mathsf{hdl}_P \in \mathcal{K}$ with the honest bit set, $\mathcal{S}$ uses the U_{msg^*} oracle for a challenge ciphertext $\mathsf{ct}_{msg} \in \mathcal{R}$ from **Hybrid** 4. Now, for any $\mathsf{ct}_{msg} \notin \mathcal{R}$, $\mathcal{S}$ neither has the oracles nor has the sk^* corresponding to pk^* in msk. But, the decryption oracle provided by the IND-CCA2 challenger can be used for any $\mathsf{ct}_{msg} \notin \mathcal{R}$. Hence, $\mathcal{S}$ can answer all the HW.Run(hdl_P, "run", report_{sk}, ct_{msg}) queries. Thus, the view of $\mathcal{A}$ is identical to **Hybrid** 5 when msg is encrypted in ct^* and **Hybrid** 6 when zeros are encrypted in ct^*. So we can forward the answer corresponding to $\mathcal{A}$'s answer to the IND-CCA2 challenger. If $\mathcal{A}$ can distinguish between these two hybrids with non-negligible probability, the IND-CCA2 security of PKE can be broken with non-negligible probability. □

8 FE construction in the stronger security model

First we will present the stronger simulation model for HW and then present our construction proven secure in this model.

8.1 Stronger HW simulation model

Definition 8.1 (StrongSimSecurity-FE). *Consider a stateful simulator $\mathcal{S}$ and a stateful adversary $\mathcal{A}$. Let $U_{\mathsf{msg}}(\cdot)$ denote a universal oracle, such that $U_{\mathsf{msg}}(P) = P(\mathsf{msg})$.*

Both games begin with a pre-processing phase executed by the environment. In the ideal game, pre-processing is simulated by $\mathcal{S}$. Now, consider the following experiments.

$$\mathsf{Exp}^{\mathsf{real}}_{\mathcal{FE}}(1^\lambda):$$

    1. $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$

    2. $(\mathsf{msg}) \leftarrow \mathcal{A}^{\mathsf{FE.Keygen}(\mathsf{msk},\cdot)}(\mathsf{mpk})$

    3. $\mathsf{ct} \leftarrow \mathsf{FE.Enc}(\mathsf{mpk}, \mathsf{msg})$

    4. $\alpha \leftarrow \mathcal{A}^{\mathsf{FE.Keygen}(\mathsf{msk},\cdot),\mathcal{O}_{\mathsf{msk}}(\cdot)}(\mathsf{mpk}, \mathsf{ct})$

    5. Output (msg, α)

$$\mathsf{Exp}^{\mathsf{ideal}}_{\mathcal{FE}}(1^\lambda):$$

    1. $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$

    2. $(\mathsf{msg}) \leftarrow \mathcal{A}^{\mathcal{S}(\mathsf{msk},\cdot)}(\mathsf{mpk})$

    3. $\mathsf{ct} \leftarrow \mathcal{S}^{U_{\mathsf{msg}}(\cdot)}(1^\lambda, 1^{|\mathsf{msg}|})$

    4. $\alpha \leftarrow \mathcal{A}^{\mathsf{HW},\mathcal{S}^{U_{\mathsf{msg}}(\cdot)}(\cdot)}(\mathsf{mpk}, \mathsf{ct})$

    5. Output (msg, α)

In the above experiment, oracle calls by $\mathcal{A}$ to the key-generation and KM oracles are simulated by the simulator $\mathcal{S}^{U_{\mathsf{msg}}(\cdot)}(\cdot)$. But the simulator does not simulate the HW algorithms, except $\mathsf{HW.Setup}$. We call a simulator admissible *if on each input P, it just queries its oracle $U_{\mathsf{msg}}(\cdot)$ on P (and hence learn just $P(\mathsf{msg})$).*

The FE scheme is said to be simulation-secure against adaptive adversaries if there is an admissible *stateful probabilistic polynomial time simulator $\mathcal{S}$ such that for every probabilistic polynomial time adversary $\mathcal{A}$ the following distributions are computationally indistinguishable.*

$$\mathsf{Exp}^{\mathsf{real}}_{\mathcal{FE}}(1^\lambda) \overset{c}{\approx} \mathsf{Exp}^{\mathsf{ideal}}_{\mathcal{FE}}(1^\lambda)$$

We now present here the formal description of our second FE construction which can be proven secure in the stronger security models of HW and FE. The trusted authority platform TA and decryption node platform DN each have access to instances of HW. We assume $\mathsf{HW.Setup}(1^\lambda)$ has been called for each of these instances before they are used in the protocol and the output params was recorded. Let PKE denote an IND-CCA2 secure public key encryption scheme (Definition 6.7) with the weak robustness property[12], let S denote an existentially unforgeable signature scheme (Definition 6.6) and E denote an IND-CPA secure secret key encryption scheme (Definition 6.5).

FE.Setup(1^λ) The key manager enclave program Q_{KME} is defined as follows. Let state denote an internal state variable.

    Q_{KME}:
- On input ("init", 1^λ):
    1. Run $(\mathsf{pk}_{\mathsf{pke}}, \mathsf{sk}_{\mathsf{pke}}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$ and $(\mathsf{vk}_{\mathsf{sign}}, \mathsf{sk}_{\mathsf{sign}}) \leftarrow \mathsf{S.KeyGen}(1^\lambda)$
    2. Update state to $(\mathsf{sk}_{\mathsf{pke}}, \mathsf{sk}_{\mathsf{sign}}, \mathsf{vk}_{\mathsf{sign}})$ and output $(\mathsf{pk}_{\mathsf{pke}}, \mathsf{vk}_{\mathsf{sign}})$
- On input ("provision", $\mathsf{quote}, \mathsf{params}$):
    1. Parse $\mathsf{quote} = (\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_P, \mathsf{in}, \mathsf{out}, \sigma)$, and parse $\mathsf{out} = (\mathsf{sid}, \mathsf{pk}^1, \mathsf{pk}^2, \mathsf{sk}_P, \mathsf{ct}_k)$.
    2. Run $b \leftarrow \mathsf{HW.QuoteVerify}(\mathsf{params}, \mathsf{quote})$ on quote. If $b = 1$, retrieve $\mathsf{sk}_{\mathsf{pke}}$ and $\mathsf{vk}_{\mathsf{sign}}$ from state. If $b = 0$ output $\perp$.
    3. Run $b \leftarrow \mathsf{S.Verify}(\mathsf{vk}_{\mathsf{sign}}, \mathsf{sk}_P, \mathsf{tag}_P)$. If $b = 0$, output $\perp$.
    4. Run $(\mathsf{ek}, h) \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}_{\mathsf{pke}}, \mathsf{ct}_k)$
    5. Compute $\mathsf{ct}^1_{sk} = \mathsf{PKE.Enc}(\mathsf{pk}^1, \mathsf{ek}||\mathsf{vk}_{\mathsf{sign}})$ and $\mathsf{ct}^2_{sk} = \mathsf{PKE.Enc}(\mathsf{pk}^2, \mathsf{ek}||\mathsf{vk}_{\mathsf{sign}})$
    6. Compute $\sigma_{sk} = \mathsf{S.Sign}(\mathsf{sk}_{\mathsf{sign}}, (\mathsf{sid}, \mathsf{ct}^1_{sk}, \mathsf{ct}^2_{sk}, h))$ and output $(\mathsf{sid}, \mathsf{ct}^1_{sk}, \mathsf{ct}^2_{sk}, h, \sigma_{sk})$.
- On input ("sign", msg):
    Compute $\mathsf{sig} \leftarrow \mathsf{S.Sign}(\mathsf{sk}_{\mathsf{sign}}, \mathsf{msg})$ and output sig.

    Run $\mathsf{hdl}_{KME} \leftarrow \mathsf{HW.Load}(\mathsf{params}, Q_{KME})$ and $(\mathsf{pk}_{\mathsf{pke}}, \mathsf{vk}_{\mathsf{sign}}) \leftarrow \mathsf{HW.Run}(\mathsf{hdl}_{KME}, (\text{"init"}, 1^\lambda))$. Output the master public key $\mathsf{mpk} := \mathsf{pk}_{\mathsf{pke}}$ and the master secret key $\mathsf{msk} := \mathsf{hdl}_{KME}$.

FE.Keygen(msk, P) Parse msk as a handle to $\mathsf{HW.Run}$. Derive tag_P and call $\mathsf{sig} \leftarrow \mathsf{HW.Run}(\mathsf{hdl}_{KME}, (\text{"sign"}, \mathsf{tag}_P))$. Output $\mathsf{sk}_p := \mathsf{sig}$.

---

[12]We actually need one PKE scheme with IND-CPA security and weak robustness property and another PKE scheme with IND-CCA2 security

FE.Enc(mpk, msg) Parse mpk = (pk, vk). Sample an ephemeral key ek ← E.KeyGen(1^λ) and use it to encrypt the message $\mathsf{ct}_m \leftarrow$ E.Enc(ek, msg). Then, encrypt the ephemeral key under pk along with the hash of ct_m: $\mathsf{ct}_k \leftarrow$ PKE.Enc(pk, [ek, $H(\mathsf{ct}_m)$]). Output ct := (ct_k, ct_m).

FE.Dec$^{\mathsf{HW}, \mathsf{KM}(\cdot)}$(sk$_P$, ct) The decryption enclave program Q_{DE} parametrized by P is defined as follows. The security parameter λ is hardcoded into the program. The Q_{DE} here can be seen as the merge of the Q_{DE} and Q_{FE} in our first construction.

$Q_{DE}(P)$:

- On input ("init dec", sk$_P$, ct_k):
    1. Run PKE.KeyGen(1^λ) twice to get ($\mathsf{pk}_{\mathsf{ra}}^1$, $\mathsf{sk}_{\mathsf{ra}}^1$) and ($\mathsf{pk}_{\mathsf{ra}}^2$, $\mathsf{sk}_{\mathsf{ra}}^2$).
    2. Generate a session ID, sid ← $\{0,1\}^\lambda$.
    3. Update state to (sid, $\mathsf{sk}_{\mathsf{ra}}^1$, $\mathsf{sk}_{\mathsf{ra}}^2$), and output (sid, $\mathsf{pk}_{\mathsf{ra}}^1$, $\mathsf{pk}_{\mathsf{ra}}^2$, sk$_P$, ct_k).

- On input ("complete dec", (sid, ct_{sk}^1, ct_{sk}^2, h), σ_{sk}):
    1. Look up the state to obtain the entry (sid, $\mathsf{sk}_{\mathsf{ra}}^1$, $\mathsf{sk}_{\mathsf{ra}}^2$). If no entry exists for sid, output $\perp$.
    2. Verify the signature $b \leftarrow$ S.Verify($\mathsf{vk}_{\mathsf{sign}}$, σ_k, (sid, ct_{sk}^1, ct_{sk}^2, h)). If $b = 0$, output $\perp$.
    3. Check that $h = H(\mathsf{ct}_m)$. If not, output $\perp$.
    4. Decrypt $m \leftarrow$ PKE.dec($\mathsf{sk}_{\mathsf{ra}}^1$, ct_{sk}^1).
    5. If $m = \perp$, decrypt and output $m \leftarrow$ PKE.dec($\mathsf{sk}_{\mathsf{ra}}^2$, ct_{sk}^2).
    6. Parse $m = $ (ek, $\mathsf{vk}_{\mathsf{sign}}$) and compute $x \leftarrow$ E.dec(ek, ct_m).
    7. Run P on x and output out := $P(x)$.

Run $\mathsf{hdl}_{DE} \leftarrow$ HW.Load(params, Q_{DE}) and call quote ← HW.Run&Quote$_{\mathsf{sk}_{\mathsf{HW}}}$($\mathsf{hdl}_{DE}$, "init dec", sk$_P$, ct_k). Query KM(quote), which internally runs (sid, ct_{sk}^1, ct_{sk}^2, h, σ_{sk}) ← HW.Run(hdl_{KME}, ("provision", quote, params))[13]. Call HW.Run(hdl_{DE}, ("complete dec", sid, ct_{sk}^1, ct_{sk}^2, h, σ_{sk})) and output its result out.

8.2   Security overview

Theorem 8.1. *If* E *is an IND-CPA secret key encryption scheme,* S *is an EUF-CMA secure signature scheme,* PKE *is an IND-CCA2 secure public key encryption scheme with weak robustness property and* HW *is a secure hardware scheme, then FE is a secure functional encryption scheme according to Definition 8.1.*

We will mention here some of the challenges faced while proving the security of our construction and refer the interested readers to the full version of the paper for a detailed security proof. The main difference from the proof of our first construction is that the HW algorithms are not simulated but are run as in the the real world. Hence, when we use the IND-CCA2 security of PKE to prove that the adversary does not learn any information from the communication between the enclaves, the decryption enclave will not have the correct secret key to decrypt the PKE ciphertext and hence cannot proceed to generate the correct output. To remedy that situation, DE sends two public keys and KME sends two ciphertexts during that step so that when the IND-CCA2 game is run for one of ciphertexts, the other ciphertext can be decrypted by DE to satisfy the correctness of the FE scheme. During this step, we will also use the indistinguishability of ciphertexts when the same messages are encrypted under different public keys. Also during this step, to help the programs decide whether the message got after decryption is correct or not, we require the robustness property from our PKE scheme which ensures that decryption outputs $\perp$ when a ciphertext is decrypted with a "wrong" key.

---

[13]We could again use HW.Run&Quote here instead of explicitly creating the signature σ_k. If we do that, the verification step in DE would involve using the Intel Attestation Service.

Discussion This construction can be modified to work like the first construction, where the decryption enclave is separated from the function enclave written by the user programmer.

This construction allows us to achieve the stronger security notions of FE and HW. But, one might wonder how our KM oracle compares with the notion of hardware tokens in [CKZ13]. With an "oracle" being necessary due to the FE impossibility results, we made the functionality of the KM oracle minimal. In our construction, KM performs minimal crypto functionality: basic signing/encryption. (And it is an independent enclave DE without access to msk which runs the user-specified programs on user-specified inputs). Hence, it is relatively easier to implement the KM functionality secure against side-channels, when compared to the powerful hardware tokens. Also from a theoretical perspective, KM runs in time independent of the runtime of program and the length of msg, in contrast to the hardware tokens whose runtime depends on both the program and msg.

The similarity of C-FE with our notion is that there is an "authority" mediating every decryption. If mediation by KM were a concern to an application of FE, the message sent by DE to the KME can be encrypted and anonymous communication mechanisms like TOR can be used to communicate to KM so that KM cannot discriminate against specific decryptor nodes (also helped by remote attestation using blind signatures). Also, our construction could be modified to achieve C-FE when the efficiency constraints are relaxed for the authority oracle such that they run in time independent on the length of the input but dependent on the function description length. The construction in [NAP$^+$14] requires the authority to run in time proportional to the length of function description and input.

9 Related Work

A number of papers leverage SGX to build secure systems. Haven [BPH14] protects unmodified Windows applications from malicious OS by running them in SGX enclaves. Scone [ATG$^+$16] and Panoply [STTS17] build secure Linux containers using SGX. VC3 [SCF$^+$15] enables secure MapReduce computations while keeping both the code and the data secret using SGX. A complete security analysis of the system was also presented but the system evaluation was performed using their own SGX performance model based on the Intel whitepapers. Ohrimenko et al. [OSF$^+$16] present data-oblivious algorithms for some popular machine learning algorithms. These algorithms can be used in conjunction with our system if one wants an FE scheme supporting machine learning functionalities. Gupta et al. [GMF$^+$16] proposed protocols and theoretical estimates for performing secure two-party computation using SGX based on the SGX specifications provided in Intel whitepapers. Concurrent to our work, Bahmani et al. [BBB$^+$16] proposed a secure multi-party computation protocol where one of the parties has access to SGX hardware and performs the bulk of the computation. They evaluate their protocol for Hamming distance, Private Set Intersection and AES. This work and [PST17] also attempt formal modeling of SGX like we do. We discuss the comparison between the models in Section 6.1.

[CKZ13] first proposed a way to bypass the impossibility results in functional encryption by the use of "hardware tokens". First, their work is purely theoretical. They model secure hardware as a single stateless deterministic token, which does not capture how SGX works because their hardware token is initialized during FE.Setup (refer Definition 5 of [CKZ13]). But in SGX, and hence in our model, the secure hardware HW is setup and initialized *independent* of FE.Setup by the trusted hardware manufacturer, Intel. After this point, an adversary who is in possession of the hardware can monitor and tamper with all the input coming in to the hardware and the corresponding outputs. Naveed et al. [NAP$^+$14] propose a related notion of FE called "controlled functional encryption". The main motivation of C-FE is to introduce an additional level of access control, where the authority mediates every decryption request.

References

[ABN10]   Michel Abdalla, Mihir Bellare, and Gregory Neven. Robust encryption. In *TCC*, pages 480–497, 2010.

[AGVW13]   Shweta Agrawal, Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption: New perspectives and lower bounds. In *CRYPTO*, pages 500–518, 2013.

[ATG⁺16]   Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. SCONE: secure linux containers with intel SGX. In *OSDI*, pages 689–703, 2016.

[BBB⁺16]   Raad Bahmani, Manuel Barbosa, Ferdinand Brasser, Bernardo Portela, Ahmad-Reza Sadeghi, Guillaume Scerri, and Bogdan Warinschi. Secure multiparty computation from SGX. *IACR Cryptology ePrint Archive*, 2016:1057, 2016.

[BF01]   Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In *CRYPTO*, pages 213–229, 2001.

[BPH14]   Andrew Baumann, Marcus Peinado, and Galen C. Hunt. Shielding applications from an untrusted cloud with haven. In *OSDI*, pages 267–283, 2014.

[BPSW16]   Manuel Barbosa, Bernardo Portela, Guillaume Scerri, and Bogdan Warinschi. Foundations of hardware-based attested computation and application to SGX. In *EuroS&P*, pages 245–260, 2016.

[BSW12]   Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: A new vision for public-key cryptography. *Commun. ACM*, 55(11):56–64, November 2012.

[Can01]   Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[CD16]   Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:086, 2016.

[CKZ13]   Kai-Min Chung, Jonathan Katz, and Hong-Sheng Zhou. Functional encryption from (small) hardware tokens. In *ASIACRYPT II*, pages 120–139, 2013.

[CLD16]   Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, pages 857–874, 2016.

[GGG⁺14]   Shafi Goldwasser, S. Dov Gordon, Vipul Goyal, Abhishek Jain, Jonathan Katz, Feng-Hao Liu, Amit Sahai, Elaine Shi, and Hong-Sheng Zhou. Multi-input functional encryption. In *EUROCRYPT 2014*, pages 578–602, 2014.

[GGH⁺13]   Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *FOCS*, pages 40–49, 2013.

[GJKS15]   Vipul Goyal, Abhishek Jain, Venkata Koppula, and Amit Sahai. *Functional Encryption for Randomized Functionalities*, pages 325–351. 2015.

[GMF⁺16]   Debayan Gupta, Benjamin Mood, Joan Feigenbaum, Kevin R. B. Butler, and Patrick Traynor. Using intel software guard extensions for efficient two-party secure function evaluation. In *FC Workshops*, pages 302–318, 2016.

[GVW12]   Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption with bounded collusions via multi-party computation. In *CRYPTO*, pages 162–179, 2012.

[JSR⁺16]   S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen. Intel software guard extensions: Epid provisioning and attestation services. 2016.

[Kat07]   Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In *EUROCRYPT*, pages 115–128, 2007.

[LHM⁺15]   Chang Liu, Austin Harris, Martin Maas, Michael W. Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS*, pages 87–101, 2015.

[LMA⁺16]   Kevin Lewi, Alex J. Malozemoff, Daniel Apon, Brent Carmer, Adam Foltzer, Daniel Wagner, David W. Archer, Dan Boneh, Jonathan Katz, and Mariana Raykova. 5gen: A framework for prototyping applications using multilinear maps and matrix branching programs. In *CCS*, pages 981–992, 2016.

[LSG⁺16]   Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. *CoRR*, abs/1611.06952, 2016.

[MAB⁺13]   Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, page 10, 2013.

[NAP⁺14]   Muhammad Naveed, Shashank Agrawal, Manoj Prabhakaran, XiaoFeng Wang, Erman Ayday, Jean-Pierre Hubaux, and Carl A. Gunter. Controlled functional encryption. In *CCS*, pages 1280–1291, 2014.

[OSF⁺16]   Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, pages 619–636, 2016.

[PST17]   Rafael Pass, Elaine Shi, and Florian Tramèr. Formal abstractions for attested execution secure processors. In *EUROCRYPT*, 2017.

[RLT15]   Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security*, pages 431–446, 2015.

[SCF⁺15]   Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *IEEE SP*, pages 38–54, 2015.

[SGX]   Intel SGX version 2. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3d-part-4-manual.pdf. Accessed: 2017-02-16.

[SGX16]   Intel software guard extensions programming reference. 2016.

[SLK⁺17]   Jaebaek Seo, Byoungyoung Lee, Sungmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-shield: Enabling address space layout randomization for sgx programs. In *NDSS*, 2017.

[SLKP17]   Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.

[STTS17]   Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. PANOPLY: Low-TCB linux applications with sgx enclaves. In *NDSS*, 2017.

[WKPK16] Nico Weichbrodt, Anil Kurmus, Peter R. Pietzuch, and Rüdiger Kapitza. Asyncshock: Exploiting synchronisation bugs in intel SGX enclaves. In *ESORICS I*, pages 440–457, 2016.

[XCP15]   Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE SP*, pages 640–656, 2015.

A   Implemented ECALLS

KeyManager.dll ECALLs

- `ecdsa_setup` generates a public and private key for 256-bit ECDSA and seals the private key using the SDK function `sgx_seal_data` (this retrieves the enclave's *seal key* via the EGETKEY instruction and AES encrypts the data). The public verification key `vk` and sealed signing key `sk` are returned to the application and written to a file. The ECDSA key generation `sgx_ecc256_create_key_pair` is implemented in `sgx_tcrypto.lib`.

- `elgamal_setup` generates an ElGamal public key `pubkey` and private key `privkey`. It signs the ElGamal public key with the TA's ECDSA private key and seals the ElGamal private key with the SDK function `sgx_seal_data` (this wraps the EGETKEY instruction to retrieve the Seal Key and AES-GCM encrypts the data). The signed public key and sealed private key are returned to the application and written to a file.

- `sign_function` takes an input array of data (a 256 bit measurement MRENCLAVE) and outputs a 256-bit ECDSA signature on this input using the key `sk` generated in `ecdsa_setup`.

- `km_ra_proc` receives as input a EC256-DHKE key share `ga`, an enclave quote structure `de_quote`. It checks that the report inside `de_quote` has the appropriate HW configuration, that its `mr_enclave` field matches the expected MRENCLAVE value of the DE, and that it also includes a 512 byte field `report_data` containing the value `ga`. The quote structure also includes an EPID signature on the report, which must be verified with the Intel Attestation Service (see [JSR$^+$16]). (This additional procedure is not implemented in our prototype). If all the checks pass, the function generates a EC256-DHKE key share `gb`, computes the shared EC256-DHKE key, derives from it a 128-bit session key `ss_key`, and encrypts both the ElGamal private key `privkey` and the ECDSA verification key `vk` under `ss_key` with Rijndael AES-GCM. Finally, it returns `gb` and the encrypted secret.

FEDecryption.dll ECALLs

- `sgx_ra_get_msg` calls the SDK function `sgx_ecc256_create_key_pair` to generate an EC256-DHKE key share `ga`. Next it calls the SDK function `sgx_init_quote` (this contacts the Intel Provisioning Server if the processor has not yet been provisioned with an EPID key). It calls the SDK function `sgx_get_quote` to obtain the quote structure `de_quote` (through a local attestation with the Quoting Enclave). It outputs `ga` and `de_quote`.

- `proc_ra_response` receives as input a EC256-DHKE key share `gb` and an encrypted ElGamal private key `privkey`. It computes the shared EC256-DHKE key with the SDK function `sgx_ecc256_compute_shared_dhkey`, derives the 128-bit session key `ss_key`, and uses it to decrypt `privkey` with Rijndael AES-GCM. Finally, it seals the decrypted key with `sgx_seal_data` and outputs the sealed key.

- `proc_local_attest` receives inputs a EC256-DHKE key share `ga`, a CMACed enclave report `fe_report`, and an ECDSA signature `fe_report_signature`. It verifies the CMAC on `fe_report` with `sgx_verify_report` (an SDK function that wraps the EGETKEY instruction to retrieve the Report Key and computes the

CMAC). It then verifies (with the KME's verification key) that `fe_report_signature` is a valid signature on the `mr_enclave` field of `fe_report`. If these verifications pass, it generates a EC256-DHKE key share `gb`, computes the shared EC256-DHKE key, derives a 128-bit shared key `aek` and encrypts the ElGamal private key `privkey` under `aek` with Rijndael AES-GCM. It returns `gb` and the encrypted `privkey`.

FEFunction.dll ECALLs

- `local_attest_to_decryption_enclave` generates a EC256-DHKE key share `ga` and calls `sgx_create_report` (an SDK function that wraps the EREPORT instruction) to generate `fe_report`, a CMACed enclave report. The values `ga`, `fe_report`, and `fe_report_signature` are passed to the OCALL `request_local_dh_session_ocall`. It receives back a EC256-DHKE key share `gb` and encrypted ElGamal private key `privkey`. It computes the shared EC256-DHKE key, derives a 128-bit shared key `aek` and decrypts `privkey`. The decrypted key is stored in a static variable.

B   SGX Attestation

Local attestation   Local attestation is between two enclaves on the same platform. The program in the enclave generating the attestation specifies a target enclave that will verify the attestation and invokes the EREPORT instruction. EREPORT first generates a *report* containing the MRENCLAVE and metadata of the calling enclave, fetching this information directly from protected memory and registers. The report may also include additional data provided by the calling enclave. Second, EREPORT uses the target enclave specification (measurements and metatada) to derive the target enclave's *Report Key* from the Root Seal Key. It then uses this Report Key to compute a MAC over the *report*. Any enclave can use the EGETKEY instruction to fetch its own Report Key, derived from the Root Seal Key and measurements/metadata linked to the enclave. Thus, the target enclave, which resides on the same platform as the attesting enclave and shares the same Root Seal Key, will be able to derive the Report Key it needs to verify the MAC on *report*.

Remote attestation   Local attestation is leveraged in remote attestation, which generates enclave reports that can be verified by remote parties. Roughly, a special enclave called the Quoting Enclave will process local attestations from other enclaves and convert these into remote attestations called *quotes*. More specifically, the Quoting Enclave possesses a private member key for an anonymous group signature scheme called Intel Enhanced Privacy ID (EPID) [JSR⁺16] that is uses to sign reports received from other locally attesting enclaves. In EPID, an *issuer* (in this case Intel) generates a group public key `gpk`, and registers members of the group by issuing member private keys. Member keys are issued through a *blind join* protocol and are unknown to the issuer. Signatures generated from private member keys can be publicly verified using `gpk`, but cannot be linked to any particular member key.[14]

EPID key provisioning   The Quoting Enclave obtains the EPID private key through an involved process with the Intel Provisioning Server. Every SGX CPU has another embedded key called the Root Provisioning Key. Unlike the Root Seal Key, the Root Provisioning Key is also given to the Intel Provisioning Server. Another special enclave called the Provisioning Enclave calls EGETKEY to derive a Provisioning Key from the Root Provisioning Key incorporating TCB specific information, enclave measurements, and metadata. Since the Intel Provisioning Server can derive the same key, the Provisioning Enclave symmetrically authenticates to the IPS, demonstrating that it is a valid Provisioning Enclave running on a genuine Intel SGX CPU at a specific TCB. Finally, an EPID private member key is delivered to the Provisioning Enclave through the EPID blind join protocol, and this key is passed to the QE.

---

[14]Currently, EPID signatures need to be verified by contacting the Intel Attestation Server.

C SGX side-channel attacks and defenses

Cache-timing attacks [CD16] cause cache misses and thus may observe enclave memory access patterns at cache-line granularity. Similarly, page-fault attacks [XCP15] can cause enclave page lookups to result in page-faults and thus may observe enclave memory access patterns at 4KB page granularity. Next, branch shadowing may directly infer control flow (i.e branches) in an enclave process. Branch shadowing exploits the fact that SGX does not erase branch history, which is used by the CPU for branch prediction, and is important for performance of the instruction pipeline. The attack infers from timing differences in branch prediction whether a target branch is stored in the branch history. And, finally synchronization bugs in the multi-threaded code running in SGX could potentially lead to even circumventing the Intel licensing procedure in creating SGX production enclaves [WKPK16]. These bugs are relatively easier to exploit in SGX than outside because the attacker model allows an untrusted OS which can control the thread scheduling of enclaves.

One defense against all the above software attacks is to ensure that enclave programs are *data-oblivious*, i.e. do not have memory access patterns or control flow branches that depend on the values of sensitive data. Ohrimenko et. al. [OSF+16] take this approach in their design of privacy-preserving multi-party machine learning using SGX. T-SGX [SLKP17] also provides countermeasures against controlled-channel attacks. A more general approach is to use ORAM techniques, as in [RLT15, LHM+15], though this can result in a considerable performance overhead. Several countermeasures to the branch shadowing attack, both hardware and software based, were proposed in [LSG+16]. Hardware countermeasures would require changes to SGX architecture. Defense mechanisms against different kinds of synchronization bugs already exist as listed by [WKPK16]. SGX-Shield [SLK+17] enables ASLR for SGX, which helps defend against these attacks in general.

Sanctum [CLD16] is an academic SGX-like system that is resilient to both cache-timing and page-fault attacks, demonstrating that these attacks are not inherent in SGX-like systems. SGX is an evolving technology, and so we can expect that even hardware based countermeasures could be incorporated into future SGX versions (see changes already in SGX2 [SGX]).