IRON: Functional Encryption using Intel SGX

Ben Fisch*
Stanford University
bfisch@stanford.edu

Dhinakaran Vinayagamurthy†
University of Waterloo
dvinayag@uwaterloo.ca

Dan Boneh‡
Stanford Univesity
dabo@cs.stanford.edu

Sergey Gorbunov§
University of Waterloo
sgorbunov@uwaterloo.ca

ABSTRACT

Functional encryption (FE) is an extremely powerful cryptographic mechanism that lets an authorized entity compute on encrypted data, and learn the results in the clear. However, all current cryptographic instantiations for general FE are too impractical to be implemented. We construct IRON, a provably secure, and practical 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.

CCS CONCEPTS

• **Security and privacy → Public key (asymmetric) techniques**; **Hardware-based security protocols**;

KEYWORDS

Functional encryption, Intel SGX, secure hardware, provable security.

1 INTRODUCTION

Functional Encryption (FE) is a powerful cryptographic tool that facilitates non-interactive fine-grained access control to encrypted data [11]. 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 (or program) f 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) [24] 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 [24, 28].

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.

In the context of cloud computing, a cloud server 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 server can run the algorithm on the encrypted data to learn the results in the clear, but without learning anything else. The data owner holds the master key, and decides what functional keys to give to the cloud.

Banks could also use FE 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. Similarly, FE could enable spam filters running on a remotely hosted email server to detect spam in encrypted email traffic without fully decrypting the emails.

The problem is that currently there aren't any practical constructions of FE from standard cryptographic assumptions for anything more than simple functionalities (e.g., inner products). Moreover, there is evidence that constructing general-purpose FE is as hard as constructing program obfuscation [4, 9, 23]. However, existing candidate constructions for obfuscation are impractical [39] and rely on very new and unestablished computational hardness assumptions, some of which have been broken [18, 44]. Previous work proposed using secure hardware to instantiate FE, however it relied on simulatable hardware "tokens" which did not model real hardware [19].

*Supported by the NSF Graduate Research Fellowship.
†Supported by the Cheriton Graduate Scholarship from the University of Waterloo.
‡This work is supported by NSF, DARPA, a grant from ONR, and the Simons Foundation. Opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.
§This work is funded by NSERC Discovery grant.

CCS'17, , Oct. 30–Nov. 3, 2017, Dallas, TX, USA.

ACM ISBN ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/http://dx.doi.org/10.1145/3133956.3134106

Our contributions. We propose the first *practical and provably secure FE system that can be instantiated today from real commonly available hardware*. We implemented our proposed system, called Iron, using Intel's Software Guard Extensions (SGX) and performed evaluation to show its practical efficiency compared with alternative cryptographic algorithms. We also propose a formal cryptographic model for analyzing the security of an SGX-based FE system and prove that Iron satisifes our security definitions.

Intel SGX provides hardware support for isolated program execution environments called *enclaves*. Enclaves are encrypted memory containers that protect against operating system, hypervisor, physical, and malware attacks. By default, designing an application to work in an enclave involves partitioning it into trusted and untrusted components and defining a communication API between them. A large component of the SGX architecture is the attestation property. Intuitively, it allows a user to attest that a remote party is running a desired program within an enclave and verify input/output tuples produced by the enclave. Enclaves run at full processor speeds, so it's very intuitive that they introduce minimal performance overheads.

However, designing a *provably secure application* from Intel SGX is a non-trivial task. While a number of works showed how to build cryptographic algorithms and systems from Intel SGX [5–7, 30, 50, 54, 57, 60], only a handful of works have attempted to model and prove systems security from Intel SGX [6, 7, 30, 48]. Reminiscent to secure protocols (such as SSL/TLS), which are easy to construct from basic cryptographic primitives, but are notoriously hard to analyze and prove, doing so requires careful understanding of nuances and techniques. We believe Intel SGX (and similar hardware encryption technologies) will become standard cryptographic tools for building secure systems. Thus, it is important to understand how to build a system with a formal model and guarantees from the beginning.

Establishing a rigorous connection between Iron and the cryptographic notion of FE is also particularly useful since FE is a very general and powerful primitive that can be used to directly construct many other cryptographic primitives, including fully homomorphic encryption (FHE) [3, 15] and obfuscation [4, 9]. Thus, rather than a complete system on its own, we view Iron as a basic framework upon which a family of more application-specific systems can be built in the future, and automatically inherit Iron's rigorous notion of security.

The security of Iron relies on trust in Intel's manufacturing process and the robustness of the SGX system. While we focus on implementing Iron with Intel SGX, in principle the system could be instantiated using other isolated execution environments that also support remote software attestation, such as XOM [40], AEGIS [55, 56], Bastion [16], Ascend [22] and Sanctum [21]. Each of these systems have slightly different trust assumptions and trusted computing bases (TCBs). A detailed comparison of these systems to Intel SGX is covered in [20]. It is important to acknowledge the limitations of basing security on trust in any particular hardware design. For instance, several side-channel attacks have come to light since SGX's initial release [13, 38, 52, 58, 59]. In our system, we ensure that the functionalities we implemented are resistant to known side-channel attacks on SGX. Generic techniques for protection against enclave side channels are also under study in various works [38, 41, 49, 53, 58].

Construction overview. 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, the client 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 client 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.

We implemented Iron and report on its performance for a number of functionalities. For complex functionalities, this implementation is (unsurprisingly) far superior to any cryptographic implementation of FE (which does not rely on hardware assumptions). We show in Section 4 that even for simple functionalities, such as comparison and small logical circuits, our implementation outperforms the best cryptographic schemes by over a 10,000 fold improvement. Furthermore, we discuss how Iron could support more expressive function authorization policies that are not possible with standard FE.

Security analysis. In this work we formalize our trust assumptions and definition of security for hardware-assisted FE, as well as rigorously prove the security of our system in this formal model (Section 6 and Section 7). While our construction of SGX-assisted FE/MIFE is clean and simple, formally proving security turns out to be complicated and non-trivial. For instance, we encounter a TLS-like situation where we have to show that no information is revealed from an encryption of m whose corresponding secret decryption key is transferred from KME to DE to the third enclave using the secure channels established between these enclaves. With an adversary being able to tamper with the inputs and the outputs of these enclaves, the "simulator" that we construct to prove the simulation-security of FE requires more care. Section 7 has more details on this.

1.1 Related Work

A number of papers use SGX to build secure systems. Haven [8] protects unmodified Windows applications from malicious OS by running them in SGX enclaves. Scone [5] and Panoply [54] build secure Linux containers using SGX. VC3 [50] 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. [47] 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. [30] 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. [6] 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 [48] also attempt formal modeling of SGX like we do. We discuss the comparison between the models in Section 5.1. Also concurrent to our work, Nayak et al. [46] designed and implemented a construction for virtual black-box obfuscation (a crypto primitive even stronger than FE) using a version of secure hardware that they design and prototype in an FPGA. In contrast, our work focuses on studying the provable guarantees from a commercially available hardware.

[19] first proposed a way to bypass the impossibility results in functional encryption by the use of "hardware tokens". But, their work is purely theoretical and 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 [19]). 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. [45] 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.

In general, various forms of trusted hardware (real ones like TPM [29] and Intel TXT [31] and theoretical ones like tamper-proof tokens [25, 37]) have enabled applications like one-time programs [25], a contractual anonymity system [51], secure multi-party computation with some strong security guarantees [27] that are either not possible or not practical otherwise.

2 INTEL SGX BACKGROUND

Intel Software Guard Extensions (SGX) [43] 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 all the system software (including OS, hypervisor, etc) 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 [32, 43]. 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.

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.

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.

Attestation. There are two forms of attestation: *local* and *remote*.

- *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, an enclave first local attests to a special enclave called the Quoting Enclave (QE), sending it a report. The QE verifies local reports and convert them 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) [36]. The QE obtains this private key during through a protocol with the Intel Provisioning Server upon device initialization. This protocol includes a symmetric authentication involving Root Provisioning Key that was embedded in the device during the manufacturing process and also shared with the Intel Provisioning Server. Currently, verifying quotes requires contacting the Intel Attestation Server, though in principle this could be done by any verifier that has the group public key.

SGX TCB. SGX stands out in that its TCB consists only of the CPU microcode and privileged containers, however it also requires the user to trust in Intel's key management infrastructure for signing microcode and various service enclaves. In particular, we must

trust that the root seal keys embedded into devices are not leaked from the manufacturing facility, and that the Intel Provisioning Server safely manages root provisioning keys as well as EPID master secret keys.

SGX side-channel attacks. The security of SGX is still evolving [35] but the current version is susceptible various side-channel attack which 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, although successful physical attacks against SGX have not yet been demonstrated.

Several software attacks have been demonstrated so far, including cache-timing attacks [20], page-fault attacks [59], branch shadowing [38] and synchronization bugs [58]. 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.*

3 SYSTEM DESIGN

3.1 Architecture 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 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 Authority in order to obtain credentials (i.e., a secret key) for a function and will then interact with any decryption node in order to perform functional decryptions of ciphertexts.

Protocol flow. The public parameters that the Authority generates includes a public encryption key for a public key cryptosystem and a public verification key for a cryptographic signature scheme. The Authority manages the corresponding secret decryption key and secret signing key. Through remote attestation, the Authority platform provisions the secret decryption key to a special enclave called a *decryption enclave* (DE) on the decryption node(s). Ciphertexts are encrypted using the public encryption key. To authorize a client application to run a function on ciphertexts, the Authority signs the function code using its secret signing key, and sends this signature to the client. When the client sends a ciphertext, function code, and valid signature on the function code to the decryption node, the DE with access to the secret key checks the signature, decrypt the ciphertext, run the function on the plaintext, and output the result. The enclave aborts on invalid signatures.

Decryption enclaves & function enclaves. Thus far in our simple description of the protocol flow, there is a single enclave on the decryption node (the DE) that manages the secret decryption key, checks function signatures, and performs functional decryption. This requires the DE to receive code as input (after enclave initialization) and to both check a signature on the code as well as execute the code. However, in the current version of SGX, enclaves cannot dynamically allocate new code pages. All enclave memory as well as the Read, Write, and Execute (RWX) permissions of each page must be committed before initialization (i.e., at build time). Therefore, the only way for the DE to execute the function it receives as native code would be to pre-allocate empty pages at build time that are both writeable and executable, and to write the function code it receives to these pages.[1] There are several drawbacks to this approach, namely that it requires the DE to predetermine the maximum size of any function it will support, and conflicts with executable space protection (the function code is more vulnerable to exploits that might overwrite code pages). A second option is to execute the function inside the DE as interpreted code, but this could greatly impact performance for more complex functions.

The third option is to load functions in entirely separate *function enclaves* and take advantage of *local attestation*, which already provides a way for one enclave to verify the code running in another. This is the cleanest design and the simplest to implement. One tradeoff, however, is that creating a new enclave for each authorized function is a relatively expensive operation. This has little impact on applications that run a few functions on many ciphertexts, but would impact applications that run many functions on only a few ciphertexts. We demonstrate in our evaluation (Section 4) 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.

Authorization policies. The Authority has full responsibility over implementing a given function authorization policy, which governs how it decides whether or not to provide a given client with a signed function. The enclaves on the decryption platform do not play any role in implementing this policy. Typically, the details of the authorization policy are beyond the scope of an FE construction and are application specific (we mentioned several examples in the introduction). It is important to note that in classical FE once a client obtains a secret key it can use it arbitrarily. Thus authorization policies are one-time decisions, and cannot cover key revocation, or limits on the number of times a client may run a function, etc. In contrast, more expressive policies may be possible in our SGX-assisted version of FE. For example, the secret key could be tagged with an expiration time that the enclaves on the decryption platform could check before running decryption by utilizing SGX's trusted time service [34]. Enforcing limits on the number of times a client can run a function would require maintaining nonvolatile enclave state, for which SGX does not immediately provide rollback protection (see [42] for a recent system providing rollback protection using SGX's monotonic counters [33]. Additionally, it would require sharing state across all active decryption enclaves with assistance from the Authority.

Key manager enclave. The Authority uses the *key manager enclave* (KME) to generate encryption and signing keys, and uses

[1] This will change in SGX2[35], which adds instructions to dynamically load new code pages into enclaves. We can revisit the design based on this new feature when SGX2 becomes available.

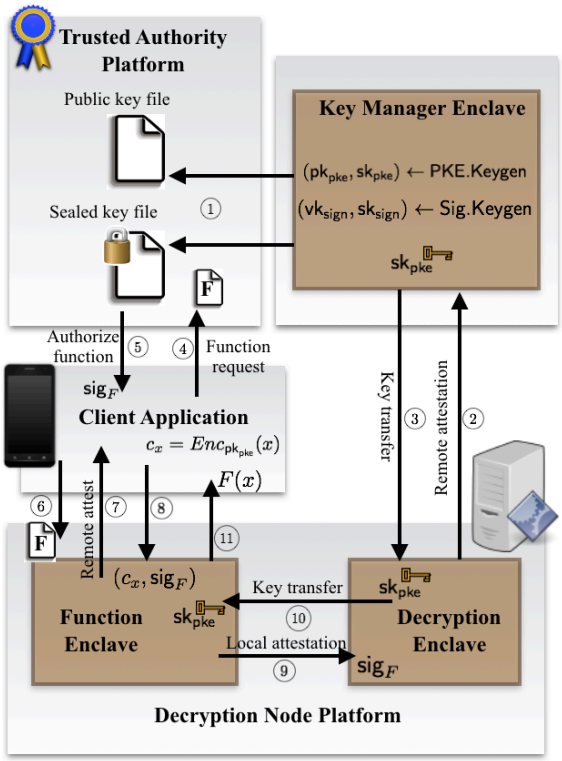

Figure 1: IRON **Architecture and Protocol Flow**

this enclave as an oracle to authorize functions. This might seem unnecessary (in our current implementation) as the Authority can use the KME to sign any function of its choice, however it offers several advantages. First, it serves as a way to protect the FE master key against an attacker that does not have long term access to the machine running the key manager enclave. Furthermore, we can imagine a more general scenario where the authorization policy is run entirely inside a key manager enclave, which only signs functions when provided with suitable proof of authorization which could come from a decentralized authority like a public blockchain or rely on an independent PKI.

3.2 FE Protocols

FE Setup. The Authority platform runs a secure enclave called the *key manager enclave* (KME) that it uses to generate a public/private key pair $(\mathsf{pk}_{\mathsf{pke}}, \mathsf{sk}_{\mathsf{pke}})$ for a CCA2 secure public key cryptosystem and a verification/signing key pair $(\mathsf{vk}_{\mathsf{sign}}, \mathsf{sk}_{\mathsf{sign}})$ for a cryptographic signature scheme. The keys $\mathsf{pk}_{\mathsf{pke}}$ and $\mathsf{vk}_{\mathsf{sign}}$ are published while the keys $\mathsf{sk}_{\mathsf{pke}}$ and $\mathsf{sk}_{\mathsf{sign}}$ are sealed with the KME's sealing key and kept in non-volatile storage. Note that the Authority has full access to the KME and can thus use it to authorize any function, thus the KME is simply used for key management. The handle to the KME's signing function call, which produces signatures using $\mathsf{sk}_{\mathsf{sign}}$, serves as the trusted authority's master secret key.

FE Decrypt Setup. When a new decryption node is initialized, the KME establishes a secure channel with a *decryption enclave* (DE) running on the decryption node SGX-enabled platform. The KME receives from the decryption node a *remote attestation*, which demonstrates that the decryption node is running the expected DE software and that the DE has the correct signature verification key $\mathsf{vk}_{\mathsf{sign}}$. The remote attestation also establishes a secure channel, i.e. contains a public key generated inside the DE. After verifying the remote attestation, the KME sends $\mathsf{sk}_{\mathsf{pke}}$ to the DE over the established secure channel, and authenticates this message by signing it with $\mathsf{sk}_{\mathsf{sign}}$.At this point, it is not at all obvious why the KME needs to sign its message to the DE. Indeed, since $\mathsf{sk}_{\mathsf{pke}}$ is encrypted, it seems that there isn't anything a man-in-the-middle attacker could do to harm security. If the message from the KME to the DE is replaced, the decryption node platform would simply fail to decrypt ciphertexts encrypted under $\mathsf{pk}_{\mathsf{pke}}$. However, it turns out that authenticating the KME's messages is necessary for our formal proof of security to work (see Section 6).

FE Keygen. A client application requests from the Authority the "secret key" for a function f.The Authority decides whether the client application is authorized to run the given function f, and if not it rejects the request. Otherwise, it produces a secret key for the function f as follows. The function f is wrapped in a *function enclave* program, described in more details below. The Authority generates an instance of this function enclave and obtains an attestation report for the enclave including the MRENCLAVE value $\mathsf{mrenclave}_f$. It then uses the KME signing handle to sign $\mathsf{mrenclave}_f$ using $\mathsf{sk}_{\mathsf{sign}}$. The signature sig_f is returned to the client application, and serves as the "secret key" sk_f.

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

FE.Decrypt. Decryption begins with a client application connecting to a decryption node that has already been provisioned with the decryption key $\mathsf{sk}_{\mathsf{pke}}$. 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 binary $\mathsf{enclave}_f$ to the decryption node, which the decryption node then runs. Note that the binary $\mathsf{enclave}_f$ it initialized by untrusted code running on the decryption node, not by the DE.
(2) The client initiates a key exchange with the function enclave, and receives a remote attestation that it has successfully established a secure channel with an Intel SGX enclave running $\mathsf{enclave}_f$. (Local client applications skip this step).
(3) The client sends over the established secure channel a vector of ciphertexts and the KME signature sig_f that it obtained from the Authority in FE.Keygen.
(4) The function enclave locally attests to the DE and passes sig_f. The DE validates this signature against $\mathsf{vk}_{\mathsf{sign}}$ and the MRENCLAVE value $\mathsf{mrenclave}_f$, which it obtains during local attestation. If this validation passes, the DE delivers the secret key $\mathsf{sk}_{\mathsf{pke}}$ to the function enclave. The DE authenticates its message to the function enclave by wrapping it

inside its own local attestation report.[2] Finally, the function enclave uses sk_{pke} 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 IMPLEMENTATION AND EVALUATION

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]. All 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, and sgx_tcrypto.lib, which includes EC256-DHKE key exchange, ECDSA signatures over the NIST P-256 elliptic curve, Rijndael AES-GCM encryption on 128-bit key sizes, and SHA256. We implemented a CCA2-secure hybrid encryption scheme using ElGamal, AES-GCM, and SHA256 in the standard way. 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, compiled with 64-bit and Debug mode build configurations.

We evaluate three special cases of functional encryption: *identity based encryption* (IBE), *order revealing encryption* (ORE), and *three input DNF* (3DNF). We chose these primarily to 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). Our evaluation confirms that the SGX-based functional encryption examples we implemented are orders of magnitude faster than cryptographic solutions without secure hardware, even for IBE which is already widely used in practice. 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.

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. 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, which was easy to achieve by implementing data comparisons in x86 assembly with the `setg` and `sete` conditional instructions (similar to [47]).

Performance evaluation. We report on the performance of FE.Decrypt, FE.Setup, and FE.Keygen (Figures 2 and 3). FE.Encrypt in our system is standard public key encryption (our implementation uses ElGamal), and this is done outside of SGX enclaves.[5]

Figure 2 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 2: 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.

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 3 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 3: Breakdown of FE.Decrypt run times for each of our SGX-FE 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.

[2] Authenticating the DE's message to the function enclave serves the same purpose as authenticating the KME's message to the DE in the formal proof of security.
[3] https://software.intel.com/sites/default/files/managed/b4/cf/Intel-SGX-SDK-Developer-Reference-for-Windows-OS.pdf
[4] https://www.microsoft.com/en-us/research/project/msr-elliptic-curve-cryptography-library
[5] 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.

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

$\textbf{ORE}^{\text{SGX}}$	$\textbf{ORE}^{\text{5Gen}}$	× increase
32 bits	32 bits	NA
172 bytes	4.7 GB	$27.3 \cdot 10^6$
23.78 ms	4 m	$10.1 \cdot 10^3$
0.32 ms	4 m	$750 \cdot 10^3$

$\textbf{3DNF}^{\text{SGX}}$	$\textbf{3DNG}^{\text{5Gen}}$	× increase
16 bits	16 bits	NA
170 bytes	2.5 GB	$14.7 \cdot 10^6$
22.76 ms	3 m	$7.9 \cdot 10^3$
0.45 ms	3 m	$400 \cdot 10^3$

Figure 4: Comparison of decryption times and ciphertext sizes for the SGX-FE 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.

Amortized decryption costs. As shown in Figure 3, 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 4.

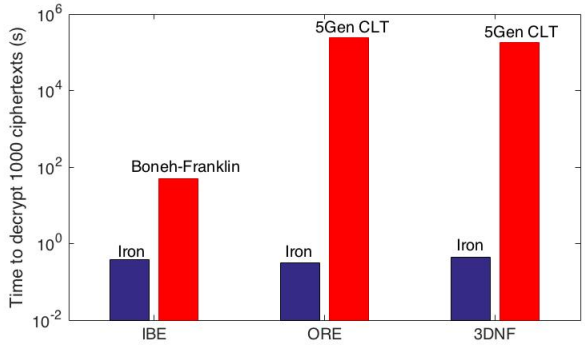

Figure 5: Comparison of time for decrypting 10^3 ciphertext tuples using the SGX-FE implementation of IBE, ORE, 3DNF vs cryptographic implementations from pairings and mmaps respectively.

Comparison to cryptographic implementations. We measured decryption time for an implementation[6] of Boneh-Franklin IBE [10] on our platform. We also include decryption time performance numbers for the 5Gen implementation[7] of mmap-based ORE and 3DNF as reported in [39]. 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 SGX-FE system makes possible primitives that are currently otherwise infeasible to build for practical use without secure hardware.

[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

5 FORMAL MODELS AND DEFINITIONS

5.1 Formal HW model

We describe a black-box program HW that captures the secure hardware's functionality and its interface exposed to the user.

Definition 5.1. The functionality HW for a class of (probabilistic polynomial time) programs $\mathcal{Q}$ consists of HW.Setup, HW.Load, HW.Run, HW.Run&Report, HW.Run&Quote, HW.ReportVerify, HW.QuoteVerify. HW has an internal state state that consists of two variables $\mathsf{HW.sk}_{\mathsf{quote}}$ and $\mathsf{HW.sk}_{\mathsf{report}}$ and a table T consisting of enclave state tuples indexed by enclave handles.

- $\mathsf{HW.Setup}(1^\lambda)$: This takes in a security parameter λ and generates the secret keys $\mathsf{sk}_{\mathsf{quote}}, \mathsf{sk}_{\mathsf{report}}$, and stores these in $\mathsf{HW.sk}_{\mathsf{quote}}$, $\mathsf{HW.sk}_{\mathsf{report}}$ respectively. Finally, it generates and outputs public parameters params.
- HW.Load(params, Q): This loads a stateful program into an enclave. HW.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 $T[\mathsf{hdl}] = \emptyset$.
- HW.Run(hdl, 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.
- $\mathsf{HW.Run\&Report}_{\mathsf{sk}_{\mathsf{report}}}(\mathsf{hdl}, \mathsf{in})$: 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. HW.Run&Report outputs the tuple $\mathsf{report} := \left(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \mathsf{mac}\right)$, where $\mathsf{md}_{\mathsf{hdl}}$ 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 $\mathsf{sk}_{\mathsf{report}}$ on $(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out})$.
- $\mathsf{HW.Run\&Quote}_{\mathsf{sk}_{\mathsf{HW}}}(\mathsf{hdl}, \mathsf{in})$: 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 $\mathsf{sk}_{\mathsf{HW}}$ for using it to sign messages. The algorithm

first executes Q on in to get out, and updates $T[\mathsf{hdl}]$ accordingly. HW.Run&Quote then outputs the tuple quote := $\left(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \sigma\right)$, where $\mathsf{md}_{\mathsf{hdl}}$ is the metadata associated with the enclave, tag_Q is a program tag for Q and σ is a signature on $(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out})$.

- $\mathsf{HW.ReportVerify}_{\mathsf{sk}_{\mathsf{report}}}(\mathsf{hdl}', \mathsf{report})$: This is the report verification algorithm. It takes as inputs, a handle hdl' for an enclave and a report = $\left(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \mathsf{mac}\right)$. It uses $\mathsf{sk}_{\mathsf{report}}$ 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 $T[\mathsf{hdl}']$.
- HW.QuoteVerify(params, quote): This is the quote verification algorithm. This takes params and quote = $\left(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \sigma\right)$ as input. It outputs 1 if the signature verification of σ succeeds. It outputs 0 otherwise.

In Appendix A, we formally define the correctness of HW as well as the security properties of HW.Run&Report, HW.Run&Quote, HW.ReportVerify, and HW.QuoteVerify as local attestation unforgeability (LocAttUnf) and remote attestation unforgeability (RemAttUnf).

Oracles and handles. HW models a single SGX chip. Our system involves multiple HW platforms, and each is modeled by a separate HW instance. When a particular process, e.g. FE.Decrypt, needs to interact with multiple platforms, the remote interactions are modeled through oracle calls, which in the real world corresponds to communicating with a process running on the relevant remote machine. The handles in the model generated by HW.Load do not need to secret or unpredictable. They are only relevant to the interfaces described in HW, which by definition can only be accessed by the HW instance itself. More concretely, in the real world SGX instantiation, these enclave handles are used only by processes running on the same machine as the enclave(s).

Modeling assumptions. 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), and that an adversary shouldn't be able to distinguish between interacting with the real world hardware and the ideal functionality. This allows us to simulate the adversary's interaction with HW in a proof of security, but it 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, stated informally, is simply that the adversary gains no more "useful" information from querying the real hardware on some input beyond the outputs specified by HW, without requiring that 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 Appendix D 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. [7] 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 the Appendix.

Pass, Shi, and Tramer [48] also define an ideal functionality for attested computation in the Universal Composability framework [14]. The goal of their model is to explore *composable* security for protocols using secure processors performing attested computation. Similar to [7] 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 [6] adapts the SGX model of [7] 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.

5.2 Functional Encryption

We adapt the definition of functional encryption to fit the computational model of our system. Interaction with local enclaves is modeled as calls to the HW functionality defined in Definition 5.1. Communication with the remote KME is modeled with a separate oracle $\mathsf{KM}(\cdot)$. We allow for a preprocessing phase which runs the setup for all HW instances. A functional encryption scheme $\mathcal{FE}$ for a family of programs $\mathcal{P}$ and message space $\mathcal{M}$ consists of algorithms $\mathcal{FE}$ = (FE.Setup, FE.Keygen, FE.Enc, FE.DecSetup, FE.Dec) defined as follows.

- $\mathsf{FE.Setup}(1^\lambda)$: On input security parameter λ (in unary), output the master public key mpk and the master secret key msk.
- FE.Keygen(msk, P): On input the master secret key msk and a program $P \in \mathcal{P}$, output the secret key sk_P for P.
- FE.Enc(mpk, msg): On input the master public key mpk and an input message msg $\in \mathcal{M}$, output a ciphertext ct.
- $\mathsf{FE.DecSetup}^{\mathsf{KM}(\cdot),\mathsf{HW}(\cdot)}(\mathsf{mpk})$: The decryption node setup algorithm has access to the KM oracle and the HW oracles. On input the master public key mpk, output a handle hdl to be used by the actual decryption algorithm.
- $\mathsf{FE.Dec}^{\mathsf{HW}(\cdot)}(\mathsf{hdl}, \mathsf{sk}_P, \mathsf{ct})$: On 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 msg $\in \mathcal{M}$, the probability for $\mathsf{FE.Dec}^{\mathsf{HW}(\cdot)}\left(\mathsf{hdl}, \mathsf{sk}_P, \mathsf{ct}\right)$ 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 FE.Setup, FE.Keygen, FE.Enc, FE.DecSetup.

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 KM(·). The decryption algorithm FE.Dec is only given access to HW.

Security definition. Here, we define a strong simulation-based security of *FE* similar to [2, 11, 26]. 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 $P(\text{msg})$.

Definition 5.2 (SimSecurity-FE). Consider a stateful simulator $\mathcal{S}$ and a stateful adversary $\mathcal{A}$. Let $U_{\text{msg}}(\cdot)$ denote a universal oracle, such that $U_{\text{msg}}(P) = P(\text{msg})$.

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.

$\mathbf{Exp}^{\mathbf{real}}_{\mathcal{FE}}(1^\lambda)$:	$\mathbf{Exp}^{\mathbf{ideal}}_{\mathcal{FE}}(1^\lambda)$:
$(\text{mpk}, \text{msk}) \leftarrow \text{FE.Setup}(1^\lambda)$	$\text{mpk} \leftarrow \mathcal{S}(1^\lambda)$
$(\text{msg}) \leftarrow \mathcal{A}^{\text{FE.Keygen}(\text{msk},\cdot)}(\text{mpk})$	$\text{msg} \leftarrow \mathcal{A}^{\mathcal{S}(\cdot)}(\text{mpk})$
$\text{ct} \leftarrow \text{FE.Enc}(\text{mpk}, \text{msg})$	$\text{ct} \leftarrow \mathcal{S}^{U_{\text{msg}}(\cdot)}(1^\lambda, 1^{\lvert\text{msg}\rvert})$
$\alpha \leftarrow \mathcal{A}^{\text{FE.Keygen}(\text{msk},\cdot),\text{HW},\text{KM}(\cdot)}(\text{mpk}, \text{ct})$	$\alpha \leftarrow \mathcal{A}^{\mathcal{S}^{U_{\text{msg}}(\cdot)}(\cdot)}(\text{mpk}, \text{ct})$
Output (msg, α)	Output (msg, α)

In the above experiment, oracle calls by $\mathcal{A}$ to the key-generation, HW and KM oracles are all simulated by the simulator $\mathcal{S}^{U_{\text{msg}}(\cdot)}(\cdot)$. 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 $U_{\text{msg}}(\cdot)$ only on the same P (and hence learn just $P(\text{msg})$), such that for every probabilistic polynomial time adversary $\mathcal{A}$ the following distributions are computationally indistinguishable.

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

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 $U_{\text{msg}_i}(\cdot)$ 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 [19]. If we do not allow simulation of HW, it is impossible to achieve Definition 5.2. In Appendix D 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 FORMAL CONSTRUCTION

We present here the formal description of our FE system using the syntax of the HW model from Definition 5.1. 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 B.3) and let S denote an existentially unforgeable signature scheme (Definition B.2).

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

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

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

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

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

FE.DecSetup$^{\text{HW},\text{KM}(\cdot)}(\text{sk}_P, \text{ct})$. 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", vk_{sign}):
  (1) Run $(\text{pk}_{\text{ra}}, \text{sk}_{\text{ra}}) \leftarrow \text{PKE.KeyGen}(1^\lambda)$.
  (2) Generate a session ID, $\text{sid} \leftarrow \{0,1\}^\lambda$.
  (3) Update state to $(\text{sid}, \text{sk}_{\text{ra}}, \text{vk}_{\text{sign}})$, and output $(\text{sid}, \text{pk}_{\text{ra}})$.

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

- On input ("complete setup", sid, ct_{sk}, σ_{sk}):
  (1) Look up the state to obtain the entry (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", report, 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, report). If not, output $\perp$.
  (3) Parse report = $\left(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \mathsf{mac}\right)$ 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 (sid, pk). If $b = 1$ output (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 mpk = ($\mathsf{sk}_{\mathsf{pke}}$, $\mathsf{vk}_{\mathsf{sign}}$) and call quote $\leftarrow \mathsf{HW.Run\&Quote}_{\mathsf{sk}_{\mathsf{HW}}}(\mathsf{hdl}_{DE}, \text{"init setup"}, \mathsf{vk}_{\mathsf{sign}})$. Query KM(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]. And now, 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 (sid, $\mathsf{sk}_{\mathsf{la}}$), and output (sid, $\mathsf{pk}_{\mathsf{la}}$).
- On input ("run", report_{sk}, ct_{msg}):
  (1) Check to see that the report has been verified, i.e. that state contains the tuple (1, report_{sk}). If not, output $\perp$.
  (2) Parse $\mathsf{report}_{sk} = \left(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \mathsf{mac}\right)$. Parse out as (sid, ct_{key}).
  (3) Look up the state to obtain the entry (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 use it to decrypt $x \leftarrow \mathsf{PKE.dec}(\mathsf{sk}_{\mathsf{pke}}, \mathsf{ct}_{msg})$.
  (5) Run P on x and record the output out := $P(x)$. Output out.

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

7 SECURITY

We first explain the crux of our security proof here. More details will follow.

We construct a simulator $\mathcal{S}$ which can simulate FE.Keygen, HW, KM 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.

7.1 Security proof

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 5.2.*

Proof. We will construct a simulator $\mathcal{S}$ for the FE security game in Definition 5.2. $\mathcal{S}$ 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

[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 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.

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 $\mathsf{HW.Setup}(1^\lambda)$ 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 $\mathcal{K}$.

FE.Enc* $(\mathsf{mpk}, 1^{|\mathsf{msg}^*|})$. $\mathcal{S}$ outputs $\mathsf{ct}^* \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}, 0^{|\mathsf{msg}^*|})$ and stores 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.

- $\mathsf{HW.Run}(\mathsf{hdl}_{KME}, \text{"provision"}, \mathsf{quote}, \mathsf{params})$: When a provision query is made to KME, $\mathcal{S}$ parses $\mathsf{quote} = \left(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \sigma\right)$ 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}.
- $\mathsf{HW.Load}(\mathsf{params}, Q)$: 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 $\mathcal{K}$.
- $\mathsf{HW.Run}(\mathsf{hdl}_{DE}, \text{"init setup"}, \mathsf{vk}_{\mathsf{sign}})$: 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}.
- $\mathsf{HW.Run}(\mathsf{hdl}_{DE}, \text{"complete setup"}, \mathsf{sid}, \mathsf{ct}_{sk}, \sigma_{sk})$: When a complete setup query is made to a $\mathsf{hdl}_{DE} \in \mathcal{D}$, $\mathcal{S}$ outputs $\perp$ 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.
- $\mathsf{HW.Run}(\mathsf{hdl}_{DE}, \text{"provision"}, \mathsf{report}, \mathsf{sig})$: When a provision query is made to a $\mathsf{hdl}_{DE} \in \mathcal{D}$, $\mathcal{S}$ parses $\mathsf{report} = \left(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \mathsf{mac}\right)$ 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}.
- $\mathsf{HW.Run}(\mathsf{hdl}_P, \text{"init"})$: 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}.
- $\mathsf{HW.Run}(\mathsf{hdl}_P, \text{"run"}, \mathsf{report}_{sk}, \mathsf{ct}_{msg})$: 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 $\mathsf{report}_{sk} = \left(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \mathsf{mac}\right)$ 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.
- For the HW.Run&Report and HW.Run&Quote queries, similar changes are made as in the respective HW.Runs above. But, 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 5.2,

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

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 5.2 (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.

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

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 $\mathsf{HW.Load}(\mathsf{params}, Q)$ is run for the Q of a DE, store the output in the list $\mathcal{D}$, and when $\mathsf{HW.Run}(\mathsf{hdl}_{DE}, \text{"init setup"}, \mathsf{vk}_{\mathsf{sign}})$ 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.

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 $\perp$ 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 $\mathsf{HW.Run\&Report}(\mathsf{hdl}_{DE}, \cdot)$ would make it output $\perp$. Hence, **Hybrid** 2 differs from **Hybrid** 1 only when a valid signature sig for tag_P is part of the "provision" query to $\mathsf{HW.Run\&Report}(\mathsf{hdl}_{DE}, \cdot)$ 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, Lemma C.1 shows that this can be used to break the existential unforgeability of the signature scheme S.

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 $(\text{sid}, \text{ct}_{sk})$ tuples. Then, on any call to HW.Run(hdl_{DE}, "complete setup", sid, ct_k, σ_k) for $\text{hdl}_{DE} \in \mathcal{D}$, if $(\text{sid}, \text{ct}_{sk}) \notin L_{KM}$, $\mathcal{S}$ outputs $\perp$.

The proof at a high level will be similar to the previous one. HW.Run(hdl_{DE}, "complete setup", $\cdot$) already outputs $\perp$ in **Hybrid** 2 if σ_{sk} is not a valid signature of $(\text{sid}, \text{ct}_{sk})$ 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 $(\text{sid}, \text{ct}_{sk})$ pair for a sid which it has seen before in the communication between KM and a DE whose handle is in $\mathcal{D}$. This is proved in Lemma C.2.

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

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

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

- On input ("run", $\text{report}_{sk}, \text{ct}_{msg}$):
  (4) If $\text{ct}_{msg} \notin \mathcal{R}$, retrieve sk_{pke} from msk. Compute $x \leftarrow \text{PKE.dec}(\text{sk}_{\text{pke}}, \text{ct}_{msg})$. Run P on x and record the output out := $P(x)$. Output out.
  (5) If $\text{ct}_{msg} \in \mathcal{R}$, query $U_{\text{msg}^*}(P)$ and output the response.

In **Hybrid** 3.1, the decryption of ct_{key} is used by $\mathcal{S}$ to decrypt ct_{msg} while running HW.Run($\text{hdl}_P, \cdot$). This ct_{key} is a valid encryption of sk_{pke} because **Hybrid** 3.0 and **Hybrid** 3.1 ensure that the encryption of sk_{pke} sent from KME to DE and then the one from DE to FE both reach FE unmodified. Hence, the sk_{pke} got by decrypting ct_{msg} is same as the one from msk. Thus, **Hybrid** 4 is indistinguishable from **Hybrid** 3.1 for any $\text{ct}_{msg} \notin \mathcal{R}$. Now, let us consider the case of $\text{ct}_{msg} \in \mathcal{R}$. $\mathcal{S}$ has the restriction that it can use the U_{msg^*} oracle only for a P for which $\text{tag}_P \in \mathcal{K}$. From **Hybrid** 3.1, we know that HW.Run($\text{hdl}_P, \cdot$) does not output $\perp$ only when run with a valid $\text{report}_{sk} = \big(\text{md}_{\text{hdl}}, \text{tag}_{DE}, (\text{report}, \text{sig}), (\text{sid}, \text{ct}_{key}), \text{mac}\big)$ which is output by a DE "provision" query. Hence, sig is a valid signature of the tag_P contained in report. Also, $\text{tag}_P \in \mathcal{K}$ with the honest bit set, as ensured in **Hybrid** 2. So, when a HW.Run "run" query is made for hdl_P, $\mathcal{S}$ is allowed use its U_{msg^*} oracle to output the 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}.

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

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 C.4). Thus 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 HW.Run(hdl_{KME}, " provision", quote, params) is called with a valid quote output by a valid instance of DE.

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

This is ensured by the Local Attestation security (Lemma C.5). And, this shows that Q_{DE} only outputs sk_{pke} encrypted under some pk_{la} that was generated by a $Q_{FE}(\text{hdl}_P, \cdot)$ running a program P that has been queried to FE.Keygen.

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

Lemma C.4 and Lemma C.5 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, Lemma C.6 will use the IND-CCA2 security gameto 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.

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, as shown in Lemma C.7.

8 EXTENSIONS AND FUTURE WORK

Private Key MIFE. There is a private key variant of MIFE where producing a valid ciphertext for the ith input to a function requires a secret encryption key ek_i. Invoking the decryption algorithm on inputs produced with an invalid key does not reveal any information about the plaintext data. For some multi-input functionalities, private key MIFE is necessary to achieve meaningful security. For example, consider the order function $ord(x, y) = 1$ iff $x > y$. In the public key setting, given an encryption c_x of x and a functional key for ord the decryptor can produce valid ciphertexts for any arbitrary integer y in order to learn $ord(x, y)$, and can recover x by binary search. Iron supports private key MIFE. In this mode, the Authority appends a signature on the appropriate index to the public encryption key, i.e. $ek_i = sig_i||\mathsf{pk}_{\mathsf{pke}}$ where sig_i is a signature on the integer i using $\mathsf{sk}_{\mathsf{sign}}$. To encrypt a message m with ek_i, the encryptor uses $\mathsf{pk}_{\mathsf{pke}}$ to produce a public key encryption $c_{i,m}$ of $sig_i||m$. When an enclave on the decryption node receives $c_{i,m}$ as the ith input to a function, it uses $\mathsf{sk}_{\mathsf{pke}}$ to decrypt $c_{i,m}$ and validates the signature appended to the message using $\mathsf{vk}_{\mathsf{sign}}$. If this is not a valid signature on the index i then the enclave aborts the operation, and otherwise it proceeds with m.

Function Private FE. Currently, Iron supports a version of FE where the function to be evaluated is not hidden from the decryptor, and moreover, it is not hidden from the decryption node. Function private FE [12] could be supported by running a single enclave on the decryption node that receives encrypted and signed function code, decrypts the function code, checks the signature, and executes the decrypted code either through an interpreter or by writing the code to pre-allocated WX enabled pages. However, doing this securely would require the capability of full program obfuscation in SGX. It has not yet been demonstrated that this is possible to achieve *practically* for generic programs given the current side-channel attacks on SGX, though some effort in this direction was made in [49] and demonstrated on SGX-like special purpose hardware in [46].

Multi-Authority FE. In multi-authority FE [17], the trust is distributed among multiple authorities instead of having a single authority manage all the credentials. Clients must obtain secret keys from all (or a suitably large subset) of the authorities in order to be able to decrypt ciphertexts. Since the secret keys in Iron are simply signatures, it would be easy to augment Iron to support this feature by using threshold-signatures and multiple KMEs.

Application-specific implementations. In addition to the above general purpose extensions, we envision that future work can build more application-specific FE systems on top of Iron. This could involve supporting more complex functionalities (and measuring their performance) as well as more expressive authorization policies, such as utilizing SGX's trusted time and monotonic counters as discussed earlier.

ACKNOWLEDGMENTS

This work was funded by NSF, DARPA, a grant from ONR, and the Simons Foundation. Opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

A HW CORRECTNESS AND SECURITY DEFINITIONS

Correctness. A HW scheme is correct if the following things hold (using the syntax from Definition 5.1): For all aux, $Q \in \mathcal{Q}$, all in in the input domain of Q and all handles $\mathsf{hdl}' \in \mathcal{H}$,

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

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

- Correctness of Quote and QuoteVerify:

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

A.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 report = $\left(\mathsf{md}'_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \mathsf{mac}\right)$ that verifies correctly for any $\mathsf{hdl}' \in \mathcal{H}$ and out = Q(in), without querying the inputs (hdl, in).

This is formally defined by the following security game.

Definition A.1. (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 HW.Setup(1^λ, 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 query = {}.
(4) $\mathcal{A}$ can run HW.Load on any input (params, Q) of its choice and get back hdl.
(5) $\mathcal{A}$ can run HW.Run&Report on input (hdl, in) of its choice and get report := $\left(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \mathsf{mac}\right)$. 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 HW.ReportVerify on input (hdl′, report) of its choice and gets back the result.

We say the adversary wins the above experiment if:

(1) HW.ReportVerify(hdl′*, report*) = 1, where report* = $(\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_Q, \mathsf{in}^*, \mathsf{out}^*, \mathsf{mac}^*)$ and
(2) $(\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_Q, \mathsf{in}^*, \mathsf{out}^*, \mathsf{mac}^*)$ was not added to query before $\mathcal{A}$ queried HW.ReportVerify on (hdl′*, report*).

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

A.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 quote $= \left(\mathsf{hdl}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \pi\right)$ 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 A.2. (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 query $= \{\}$.
(4) $\mathcal{A}$ can run HW.Load on any input (params, Q) of its choice and get back hdl.
(5) Also, $\mathcal{A}$ can run HW.Run&Quote on input $(\mathsf{hdl}, \mathsf{in})$ of its choice and get quote $:= \left(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \pi\right)$. 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) $(\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_Q, \mathsf{in}^*, \mathsf{out}^*) \notin$ 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 $(\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_Q, \mathsf{in}^*, \mathsf{out}^*) \in$ 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 $\mathsf{sk}_{\mathsf{HW}}$, initial state state or intermediary states of the programs running inside secure containers.
- The output of the 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 aux, aux′,

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

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

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

B CRYPTO PRIMITIVE DEFINITIONS

Secret key encryption. A secret key encryption scheme E supporting a message domain $\mathcal{M}$ consists of a probabilistic polynomial time key generation algorithm $\mathsf{E.KeyGen}(1^\lambda)$ that takes in a security parameter and outputs a key sk from the key space $\mathcal{K}$, a probabilistic polynomial time encryption algorithm $\mathsf{E.Enc}(\mathsf{sk}, \mathsf{msg})$ that takes in a key sk and a message $\mathsf{msg} \in \mathcal{M}$ and outputs the ciphertext ct, and a deterministic polynomial time decryption algorithm $\mathsf{E.Dec}(\mathsf{sk}, \mathsf{ct})$ that takes in a key sk and a ciphertext ct and outputs the decryption msg.

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

$$\Pr\left[\mathsf{E.Dec}\Big(\mathsf{sk}, \mathsf{E.Enc}(\mathsf{sk}, \mathsf{msg})\Big) \neq \mathsf{msg} \,\middle|\, \mathsf{sk} \leftarrow \mathsf{E.KeyGen}(1^\lambda)\right] = \mathsf{negl}(\lambda) \quad (2)$$

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

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

Definition B.1. (IND-CPA security of E). We define 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 $b \in \{0, 1\}$.
(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 signature scheme. A digital signature scheme S supporting a message domain $\mathcal{M}$ consists of a probabilistic polynomial time algorithm $\mathsf{S.KeyGen}(1^\lambda)$ that takes in a security parameter and outputs the signing key sk and a verification key vk, a probabilistic polynomial time signing algorithm $\mathsf{S.Sign}(\mathsf{sk}, \mathsf{msg})$ that takes in a signing key sk and a message $\mathsf{msg} \in \mathcal{M}$ and outputs the signature σ, and a deterministic verification algorithm $\mathsf{S.Verify}(\mathsf{vk}, \sigma, \mathsf{msg})$ that takes in a verification key vk, a signature σ and a message msg and outputs 0 or 1.

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

$$\Pr\left[\mathsf{S.Verify}\Big(\mathsf{vk}, \mathsf{S.Sign}(\mathsf{sk}, \mathsf{msg}), \mathsf{msg}\Big) = 0 \,\middle|\, (\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{S.KeyGen}(1^\lambda)\right] = \mathsf{negl}(\lambda) \quad (3)$$

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

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

Definition B.2. (EUF-CMA security of $\mathcal{S}$). We define the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$.

(1) The challenger runs the S.KeyGen algorithm to obtain the key pair (sk, 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 S.Sign(sk, msg). Also, query = query ∪ 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

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

Public key encryption. A public key encryption (PKE) scheme supporting a message domain $\mathcal{M}$ consists of a probabilistic polynomial time algorithm $\mathsf{PKE.KeyGen}(1^\lambda)$ that takes in a security parameter and outputs a key pair (pk, sk), a probabilistic encryption algorithm PKE.Enc(pk, msg) that takes in a public key pk and a message $\mathsf{msg} \in \mathcal{M}$ and outputs a ciphertext ct, and a deterministic decryption algorithm PKE.Dec(sk, ct) that takes in a secret key sk and a ciphertext ct and outputs the decryption msg or $\perp$.

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

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

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

A PKE scheme provides confidentiality to the encrypted message. Formally, a PKE scheme PKE is said to have *indistinguishability security under adaptively chosen ciphertext attack* (IND-CCA2) if there is no polynomial time adversary $\mathcal{A}$ which can guess $b' = b$ in the following game with probability non-negligible in λ, *plus* half.

Definition B.3. (IND-CCA2 security of PKE). We define 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 (pk, 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'.

A PKE scheme may also be "weakly robust" [1]. Informally, this means that a ciphertext when decrypted with an "incorrect" secret key should output $\perp$ when all the algorithms are honestly run.

Definition B.4. ((Weak) robustness property of PKE). A PKE scheme PKE has the (weak) robustness property if for all λ and $\mathsf{msg} \in \mathcal{M}$,

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

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

C SECURITY PROOF LEMMATA

Lemma C.1. *If the signature scheme* S *is existentially unforgeable as in Definition B.2, 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 $\mathsf{S.Sign}(\mathsf{sk}^*_{\mathsf{sign}}, \cdot)$ 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 using $\mathsf{sk}^*_{\mathsf{sign}}$, it uses the $\mathsf{S.Sign}(\mathsf{sk}^*_{\mathsf{sign}}, \cdot)$ 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 $\mathsf{HW.Run\&Report}(\mathsf{hdl}_{DE}, \cdot)$ oracle with a $\mathsf{hdl}_{DE} \in \mathcal{D}$ that has $\mathsf{vk}^*_{\mathsf{sign}}$ in its , 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. □

Lemma C.2. *If the signature scheme* S *is existentially unforgeable as in Definition B.2, 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 C.1, $\mathcal{S}$ 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 , . Also, $\mathsf{hdl}_{DE} \in \mathcal{D}$ and hence has $\mathsf{vk}^*_{\mathsf{sign}}$ in its , . We will output $((\mathsf{sid}, \mathsf{ct}_{sk}), \sigma_{sk})$ as our forgery to the EUF-CMA challenger. □

Lemma C.3. *If the Local Attestation process of* HW *is secure as in Definition A.1, then* **Hybrid** 3.1 *is indistinguishable from* **Hybrid** 3.0.

The proof of this lemma is similar to Lemma C.2, since $\mathsf{sk}_{\mathsf{report}}$ is not used by $\mathcal{S}$ other than to produce a report.

Lemma C.4. *If Remote Attestation is secure as in Definition A.2, then* **Hybrid** 5.0 *is indistinguishable from* **Hybrid** 4.

The proof of this lemma is similar to Lemma C.2 since $\mathsf{sk}_{\mathsf{quote}}$ is not used by $\mathcal{S}$ except for producing a quote.

Lemma C.5. *If Local Attestation is secure as in Definition A.1, then* **Hybrid** 5.1 *is indistinguishable from* **Hybrid** 5.0.

The proof of this lemma is again similar to Lemma C.2 since $\mathsf{sk}_{\mathsf{report}}$ is not used by $\mathcal{S}$ except for producing a report.

Lemma C.6. *If* PKE *is an IND-CCA2 secure encryption scheme, then* **Hybrid** 5.2 *is indistinguishable from* **Hybrid** 5.1.

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. □

Lemma C.7. *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 $\mathsf{HW.Run}(\mathsf{hdl}_P, \text{"run"}, \mathsf{report}_{sk}, \mathsf{ct}_{msg})$ 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 $\mathsf{HW.Run}(\mathsf{hdl}_P, \text{"run"}, \mathsf{report}_{sk}, \mathsf{ct}_{msg})$ 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. □

D STRONGER HW SIMULATION MODEL

Definition D.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 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.

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

E FE CONSTRUCTION IN THE STRONGER SECURITY MODEL

We 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 B.3) with the weak robustness property[11], let S denote an existentially unforgeable signature scheme (Definition B.2) and E denote an IND-CPA secure secret key encryption scheme (Definition B.1).

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", quote, params):
  (1) Parse quote = $(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_P, \mathsf{in}, \mathsf{out}, \sigma)$, and parse 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 $\mathsf{ct}^2_{sk} = \mathsf{PKE.Enc}(\mathsf{pk}^2, \mathsf{ek}||\mathsf{vk}_{\mathsf{sign}})$

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

(6) Compute σ_{sk} = S.Sign($\mathsf{sk}_{\mathsf{sign}}$, (sid, ct^1_{sk}, ct^2_{sk}, h)) and output (sid, ct^1_{sk}, ct^2_{sk}, h, σ_{sk}).

- On input ("sign", msg):
  Compute sig ← S.Sign($\mathsf{sk}_{\mathsf{sign}}$, msg) and output sig.

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

FE.Keygen(msk, P). Parse msk as a handle to HW.Run. Derive tag_P and call sig ← HW.Run(hdl_{KME}, ("sign", tag_P)). Output sk_P := sig.

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

FE.Dec$^{\mathsf{HW},\mathsf{KM}(\cdot)}(\mathsf{sk}_P, \mathsf{ct})$. 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 ($\mathsf{pk}^1_{\mathsf{ra}}$, $\mathsf{sk}^1_{\mathsf{ra}}$) and ($\mathsf{pk}^2_{\mathsf{ra}}$, $\mathsf{sk}^2_{\mathsf{ra}}$).
  (2) Generate a session ID, sid ← $\{0,1\}^\lambda$.
  (3) Update state to (sid, $\mathsf{sk}^1_{\mathsf{ra}}$, $\mathsf{sk}^2_{\mathsf{ra}}$), and output (sid, $\mathsf{pk}^1_{\mathsf{ra}}$, $\mathsf{pk}^2_{\mathsf{ra}}$, sk_P, ct_k).
- On input ("complete dec", (sid, ct^1_{sk}, ct^2_{sk}, h), σ_{sk}):
  (1) Look up the state to obtain the entry (sid, $\mathsf{sk}^1_{\mathsf{ra}}$, $\mathsf{sk}^2_{\mathsf{ra}}$). If no entry exists for sid, output ⊥.
  (2) Verify the signature b ← S.Verify($\mathsf{vk}_{\mathsf{sign}}$, σ_k, (sid, ct^1_{sk}, ct^2_{sk}, h)). If b = 0, output ⊥.
  (3) Check that $h = H(\mathsf{ct}_m)$. If not, output ⊥.
  (4) Decrypt m ← PKE.dec($\mathsf{sk}^1_{\mathsf{ra}}$, ct^1_{sk}).
  (5) If m = ⊥, decrypt and output out ← PKE.dec($\mathsf{sk}^2_{\mathsf{ra}}$, ct^2_{sk}).
  (6) Parse m = (ek, $\mathsf{vk}_{\mathsf{sign}}$) and compute x ← E.dec(ek, ct_m).
  (7) Run P on x and output out := $P(x)$.

Run hdl_{DE} ← HW.Load(params, Q_{DE}) and call quote ← $\mathsf{HW.Run\&Quote}_{\mathsf{sk}_{\mathsf{HW}}}(\mathsf{hdl}_{DE}$, "init dec", sk_P, ct_k). Query KM(quote), which internally runs (sid, ct^1_{sk}, ct^2_{sk}, h, σ_{sk}) ← HW.Run(hdl_{KME}, ("provision", quote, params))[12]. Call HW.Run(hdl_{DE}, ("complete dec", sid, ct^1_{sk}, ct^2_{sk}, h, σ_{sk})) and output its result out.

E.1 Security overview

Theorem E.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 D.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 ⊥ when a ciphertext is decrypted with a "wrong" key.

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 [19]. 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 [45] requires the authority to run in time proportional to the length of function description and input.

[12] 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.

REFERENCES

[1] Michel Abdalla, Mihir Bellare, and Gregory Neven. 2010. Robust Encryption. In *TCC*. 480–497.

[2] Shweta Agrawal, Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. 2013. Functional Encryption: New Perspectives and Lower Bounds. In *CRYPTO*. 500–518.

[3] Joël Alwen, Manuel Barbosa, Pooya Farshim, Rosario Gennaro, S. Dov Gordon, Stefano Tessaro, and David A. Wilson. 2013. On the Relationship between Functional Encryption, Obfuscation, and Fully Homomorphic Encryption. In *IMACC*. 65–84.

[4] Prabhanjan Ananth and Abhishek Jain. 2015. Indistinguishability Obfuscation from Compact Functional Encryption. In *CRYPTO I*. 308–326.

[5] 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. 2016. SCONE: Secure Linux Containers with Intel SGX. In *OSDI*. 689–703.

[6] Raad Bahmani, Manuel Barbosa, Ferdinand Brasser, Bernardo Portela, Ahmad-Reza Sadeghi, Guillaume Scerri, and Bogdan Warinschi. 2017. Secure Multiparty Computation from SGX. In *FC*.

[7] Manuel Barbosa, Bernardo Portela, Guillaume Scerri, and Bogdan Warinschi. 2016. Foundations of Hardware-Based Attested Computation and Application to SGX. In *EuroS&P*. 245–260.

[8] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *OSDI*. 267–283.

[9] Nir Bitansky and Vinod Vaikuntanathan. 2015. Indistinguishability Obfuscation from Functional Encryption. In *FOCS*. 171–190.

[10] Dan Boneh and Matthew K. Franklin. 2001. Identity-Based Encryption from the Weil Pairing. In *CRYPTO*. 213–229.

[11] Dan Boneh, Amit Sahai, and Brent Waters. 2012. Functional Encryption: A New Vision for Public-key Cryptography. *Commun. ACM* 55, 11 (Nov. 2012), 56–64. https://doi.org/10.1145/2366316.2366333

[12] Zvika Brakerski and Gil Segev. 2015. Function-Private Functional Encryption in the Private-Key Setting. In *TCC II*. 306–324.

[13] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. *CoRR* abs/1702.07521 (2017).

[14] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *FOCS*. 136–145.

[15] Ran Canetti, Huijia Lin, Stefano Tessaro, and Vinod Vaikuntanathan. 2015. Obfuscation of Probabilistic Circuits and Applications. In *TCC II*. 468–497.

[16] David Champagne and Ruby B. Lee. 2010. Scalable architectural support for trusted software. In *HPCA*. 1–12.

[17] Nishanth Chandran, Vipul Goyal, Aayush Jain, and Amit Sahai. 2015. Functional Encryption: Decentralised and Delegatable. Cryptology ePrint Archive, Report 2015/1017. (2015). http://eprint.iacr.org/2015/1017.

[18] Yilei Chen, Craig Gentry, and Shai Halevi. 2017. Cryptanalyses of Candidate Branching Program Obfuscators. In *EUROCRYPT*. 278–307.

[19] Kai-Min Chung, Jonathan Katz, and Hong-Sheng Zhou. 2013. Functional Encryption from (Small) Hardware Tokens. In *ASIACRYPT II*. 120–139.

[20] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 086.

[21] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security*. 857–874.

[22] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. 2012. A secure processor architecture for encrypted computation on untrusted programs. In *STC*. ACM, 3–8.

[23] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. 2013. Candidate Indistinguishability Obfuscation and Functional Encryption for all Circuits. In *FOCS*. 40–49.

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

[25] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. 2008. One-Time Programs. In *CRYPTO*. 39–56.

[26] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. 2012. Functional Encryption with Bounded Collusions via Multi-party Computation. In *CRYPTO*. 162–179.

[27] Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. 2010. Founding Cryptography on Tamper-Proof Hardware Tokens. In *TCC*. 308–326.

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

[29] Trusted Computing Group. 2009. Trusted Platform Module. https://trustedcomputinggroup.org/. (2009).

[30] Debayan Gupta, Benjamin Mood, Joan Feigenbaum, Kevin R. B. Butler, and Patrick Traynor. 2016. Using Intel Software Guard Extensions for Efficient Two-Party Secure Function Evaluation. In *FC Workshops*. 302–318.

[31] Intel. 2009. Intel Trusted Execution Technology. (2009).

[32] Intel. 2016. Intel Software Guard Extensions Programming Reference. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf

[33] Intel. 2016. SGX documentation: `sgx_create_monotonic_counter`. https://software.intel.com/en-us/node/696638. (2016).

[34] Intel. 2016. SGX documentation: `sgx_get_trusted_time`. https://software.intel.com/en-us/node/696636. (2016).

[35] Intel. 2017. 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. (2017). Accessed: 2017-02-16.

[36] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen. 2016. Intel Software Guard Extensions: EPID provisioning and attestation services.

[37] Jonathan Katz. 2007. Universally Composable Multi-party Computation Using Tamper-Proof Hardware. In *EUROCRYPT*. 115–128.

[38] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security*.

[39] Kevin Lewi, Alex J. Malozemoff, Daniel Apon, Brent Carmer, Adam Foltzer, Daniel Wagner, David W. Archer, Dan Boneh, Jonathan Katz, and Mariana Raykova. 2016. 5Gen: A Framework for Prototyping Applications Using Multilinear Maps and Matrix Branching Programs. In *CCS*. 981–992.

[40] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. 2000. Architectural Support for Copy and Tamper Resistant Software. In *ASPLOS*. 168–177.

[41] Chang Liu, Austin Harris, Martin Maas, Michael W. Hicks, Mohit Tiwari, and Elaine Shi. 2015. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *ASPLOS*. 87–101.

[42] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. Cryptology ePrint Archive, Report 2017/048. (2017). http://eprint.iacr.org/2017/048.

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

[44] Eric Miles, Amit Sahai, and Mark Zhandry. 2016. Annihilation Attacks for Multilinear Maps: Cryptanalysis of Indistinguishability Obfuscation over GGH13. In *CRYPTO*.

[45] Muhammad Naveed, Shashank Agrawal, Manoj Prabhakaran, XiaoFeng Wang, Erman Ayday, Jean-Pierre Hubaux, and Carl A. Gunter. 2014. Controlled Functional Encryption. In *CCS*. 1280–1291.

[46] Kartik Nayak, Christopher Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi, and Vipul Goyal. 2017. Hop: Hardware makes obfuscation practical. In *NDSS*.

[47] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security*. 619–636.

[48] Rafael Pass, Elaine Shi, and Florian Tramèr. 2017. Formal Abstractions for Attested Execution Secure Processors. In *EUROCRYPT*.

[49] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security*. 431–446.

[50] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *IEEE SP*. 38–54.

[51] Edward J. Schwartz, David Brumley, and Jonathan M. McCune. 2010. Contractual Anonymity. In *NDSS*.

[52] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. *CoRR* abs/1702.08719 (2017).

[53] Jaebaek Seo, Byoungyoung Lee, Sungmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *NDSS*.

[54] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. PANOPLY: Low-TCB Linux Applications with SGX Enclaves. In *NDSS*.

[55] G. Edward Suh, Dwaine E. Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *ICS*. 160–171.

[56] G. Edward Suh, Charles W. O'Donnell, and Srinivas Devadas. 2007. Aegis: A Single-Chip Secure Processor. *IEEE Design & Test of Computers* 24, 6 (2007), 570–580.

[57] Chia-che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. 2014. Cooperation and security isolation of library OSes for multi-process applications. In *EuroSys*. 9:1–9:14.

[58] Nico Weichbrodt, Anil Kurmus, Peter R. Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *ESORICS I*. 440–457.

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

[60] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI*. 283–298.