Functional Encryption from Secure Enclaves

Sergey Gorbunov* Dhinakaran Vinayagamurthy †

Abstract

Functional encryption (FE) is an emerging paradigm for public-key cryptography that enables fine-grained access control over encrypted data. In FE, each function (program) P is associated with a secret key sk_P. User holding sk_P and a ciphertext ct encrypting a message msg, can learn $P(\mathsf{msg})$ in clear, but nothing else about the message is revealed. Unfortunately, all the existing constructions are either very restrictive in the supported classes of functions, or rely on non-standard mathematical assumptions and satisfy weaker security notions such as indistinguishability-based security, or far from satisfying practical efficiency for general function families.

In this work, we present a construction of functional encryption in a hardware assisted model of computation. We prove the security of our construction under the simulation-based definition. We present an implementation of our construction and show essential evaluation results, which demonstrate that our construction is very practical. In our evaluation, key-generation, encryption and decryption take around 1, 22 and 140 milliseconds for linear regression programs over 4 million sample points. Our construction is motivated by the recent advances in processors that enable creation of encrypted memory containers.

1 Introduction

Functional encryption (FE) is a new vision for public-key encryption proposed by Boneh, Sahai and Waters [BSW12]. On the high level, FE provides fine-grained access control mechanism needed for emerging cloud and mobile applications in today's world. In FE, any program P can be assigned a "program specific" secret key sk_P. A user, holding the secret key sk_P and a ciphertext $\mathsf{ct} = \mathsf{Enc}(\mathsf{msg})$, can learn $P(\mathsf{msg})$. Moreover, informally, the security of functional encryption ensures that no other information about msg can be learned, except $P(\mathsf{msg})$.

Functional encryption generalizes and subsumes many of the previously defined notions including (anonymous) identity-based encryption (IBE) [Sha84, BF01, Coc01, BW06], fuzzy IBE [SW05], attribute-based encryption (ABE) [GPSW06, LOS+10, GVW13], and predicate encryption [KSW08, LOS+10, GVW15]. Many of these notions found extensive applications in protecting medical, financial and personal information in the cloud [APG+11, LYZ+13, PPM+14]. For instance, using FE, a user may issue a secret key to an email server for a program that performs spam detection. Using the secret key, the email server can learn whether or not encrypted emails contain spam, but nothing else about the email is revealed to the server.

Unfortunately, strong security notions for functional encryption (known as simulation-based), very much desired in the real world, are known to be impossible in the standard models for many natural classes of programs [BSW12, AGVW13]. A lot of research effort went into understanding the impossibility results and searching for new directions [CIJ+13, AAP15].

One direction to bypass these impossibility results is to look at a weaker indistinguishability-based notion for FE. However, even this notion is very hard to realize. Currently, existing candidate constructions are built from multi-linear maps and indistinguishability obfuscation [GGH13a, CLT13, GGH15, GGH+13b], which are new cryptographic objects that rely on non-standard mathematical assumptions. Many of these are shown to be broken [CHL+15, HJ16, CLLT15, MSZ16], and significant research effort is needed to understand how to build them securely. Finally, even the existing candidates are very far from practical as they require astronomically large keys and ciphertexts [AHKM14].

Another direction to bypass FE simulation-based impossibility results is to introduce new assumptions in the model. For instance, the work by Chung, Katz and Zhou [CKZ13] bypassed the impossibilities by introducing "small

*University of Waterloo. Email: sgorbunov@uwaterloo.ca.
†University of Waterloo. Email: dvinayag@uwaterloo.ca.

hardware tokens". However, these hardware tokens are simply programmable "deterministic oracles" and not modeled after any concrete real primitive.

Together, existing impossibility results and technical difficulties put functional encryption for general functions on a melting iceberg and in search for a new land.

1.1 Our Contributions

In this work, we present a new construction of functional encryption for arbitrary programs that leverages new advances in secure processor technologies. More explicitly, our contributions can be summarized as follows:

- We describe an oracle assisted model of computation for functional encryption that enhances classical models (and can be used to bypass existing impossibility results). In our model, the oracle is restricted to strict efficiency requirements discussed below.
- We present a construction of functional encryption for arbitrary programs in the oracle assisted model of computation. Our construction leverages a secure hardware component, modeled after existing encrypted memory containers such as Intel Software Guard Extensions (SGX) enclaves. We prove the security of our construction under the simulation-based security definition.
- We present an architecture, implementation and evaluation of our functional encryption scheme.

The hardware component used in our construction can be instantiated with existing processor technology (e.g. Intel SGX [MAB+13], or AMD memory encryption [KPW16]) that enable creation of encrypted memory containers.

Our results set in motion a new direction for functional encryption, where it is possible to simultaneously satisfy strong security definitions and very practical aspects desired from all cryptographic primitives. We now provide a high level overview of our contributions. In the discussion later in this section we highlight additional practical considerations, our assumptions, and future research directions.

Our Model of Computation We define an oracle assisted model of computation and show how to construct a functional encryption scheme in this model. Informally, in this model, a decryptor is given access to an oracle $\mathcal{O}(\cdot)$ (defined below) and a secure hardware component HW. The oracle is restricted to strict efficiency requirements that help prevent trivial and not very useful instantiations. Prior to our work, the works of Chung et al. [CKZ13] and Naveed et al. [NAP+14] used similar albeit more powerful oracles to achieve simulation secure functional encryption and "controlled" functional encryption, respectively. (Please refer to Section 1.3 for a comparison of our work with their works. We also discuss in Section 1.2 why neither HW nor an efficient $\mathcal{O}_{\mathsf{msk}}$, is sufficient by itself for constructing a simulation secure FE scheme).

We envision that the oracle $\mathcal{O}(\cdot)$ is run by an authority or an independent enclave on the system of the decryptor or a third party.[1]. Intuitively, in our construction, this oracle is used to verify/decrypt small tokens for the decryptor. We enforce an efficiency requirement for the authority oracle, which states that it must run in time polynomial in the security parameter, but independent of the input size, program size, and the program runtime.[2]

The secure hardware component HW satisfies the following semantics. It can load an arbitrary program P and run P on any input x. It outputs the result $y = P(x)$ along with a succinct proof that can be used to authenticate the result of computation (with respect to some global public parameters). All intermediary results of the computations are hidden from the outside world.

We built our prototype on a modern Intel CPU enabled with a Software Guard Extensions (SGX). On the high level, these CPUs are equipped with instructions that can be used to create encrypted memory containers. One can create an encrypted container on an arbitrary program P, and run it on arbitrary inputs. No adversary, even with physical

[1] For the latter cases, one can load its code, without the "secret", inside the enclave. Then, use the attestation service to verify that the right code has been loaded. Finally, transmit the secret to the enclave, via a secure channel. All this can be performed during the setup phase of the functional encryption.

[2] Without this condition an FE construction would be trivial and not very meaningful. Given a ciphertext (encrypted under a standard public-key encryption) and a secret key (corresponding to a digital signature), the authority can verify the key, decrypt the input, compute the program on the input and return the result to the decryptor.

access to the system, can see the internal computation state of these containers [MAB+13, AGJS13]. In particular, all program code, data and state values are stored encrypted in computer's memory. A hardware encryption/decryption engine decrypts each instruction and required data and passes it to the CPU. After the CPU finishes executing the instruction, the engine encrypts the results and places them back in the memory. Hence, the memory remains encrypted at all times during the execution.

Construction Overview *Zeroth attempt.* From what we have described till now, a shrewd reader would have noticed a very trivial construction: generate the FE master public and secret keys and then instantiate the secure hardware by inscribing the master secret key inside it. Given the ciphertext, the secure hardware can use the master secret key to decrypt the message, evaluate and output any function on the message. But, this fails for two main reasons. First of all, the setup of secure hardware is done once and for all, independently of the FE setup. That is, we envision that a manufacturer (e.g., Intel) ships computers equipped with the secure hardware. Since anyone should be able to run the FE scheme, there is no method to simply "place" the master secret key inside the secure hardware (without giving it to the manufacturer or leveraging additional crypto protocols, such as secure channels). However, the second reason, is even more troublesome: this construction does not satisfy the simulation-based security notion. In Section 1.2, we discuss why any realistically modelled secure hardware is not sufficient by itself to satisfy simulation-based FE notions.

First attempt. We now describe a simple reasonable starting point for our construction, which is unfortunately insufficient again to satisfy the simulation-based security definition. Let $(\mathsf{pk}_{\mathsf{pke}}, \mathsf{sk}_{\mathsf{pke}})$ denote a public/secret key pair for a semantically secure public-key encryption scheme. Let $(\mathsf{vk}_{\mathsf{sign}}, \mathsf{sk}_{\mathsf{sign}})$ denote a verification/signing key pair for a secure signature scheme. A secret key for a program P is simply a signature σ_P, generated using $\mathsf{sk}_{\mathsf{sign}}$. On an input msg, the ciphertext ct is generated by encrypting msg with the public key $\mathsf{pk}_{\mathsf{pke}}$. Now, consider a decryptor that has oracle access to $\mathcal{O}(\cdot)$ and a trusted hardware component HW. To describe how the decryptor works, we must first define how these oracles are operating.

The authority oracle $\mathcal{O}$ on input $\mathsf{pk}_{\mathrm{tmp}}$ (arbitrary public key), outputs $\mathsf{ct}_{\mathrm{tmp}} \leftarrow \mathsf{Enc}_{\mathsf{pk}_{\mathrm{tmp}}}(\mathsf{sk}_{\mathsf{pke}})$ (that is an encryption of the PKE secret key under the input public key).

Also, consider the hardware component HW that does the following:

- On load, it generates a temporary public/secret key pair $(\mathsf{pk}_{\mathrm{tmp}}, \mathsf{sk}_{\mathrm{tmp}})$ and outputs $\mathsf{pk}_{\mathrm{tmp}}$. It stores $\mathsf{sk}_{\mathrm{tmp}}$ in the internal state (hidden from the outside world).
- On input $(\mathsf{ct}, \mathsf{ct}_{\mathrm{tmp}}, \sigma_P, \mathsf{vk}_{\mathsf{sign}})$, it decrypts $\mathsf{ct}_{\mathrm{tmp}}$ to get $\mathsf{sk}_{\mathsf{pke}} \leftarrow \mathsf{Dec}_{\mathsf{sk}_{\mathrm{tmp}}}(\mathsf{ct}_{\mathrm{tmp}})$. Next, it verifies the signature σ_P. Finally, it decrypts ct to obtain $\mathsf{msg} \leftarrow \mathsf{dec}_{\mathsf{sk}_{\mathsf{pke}}}(\mathsf{ct})$, evaluates $y = P(\mathsf{msg})$ and outputs the result y.

It is now clear how the decryptor works: it loads the secure hardware HW (technically, there is an implicitly defined program Q which it must load into the secure hardware piece and then invoke it on various inputs) and obtains $\mathsf{pk}_{\mathrm{tmp}}$. It calls the $\mathcal{O}$ oracle with $\mathsf{pk}_{\mathrm{tmp}}$ to obtain $\mathsf{ct}_{\mathrm{tmp}} \leftarrow \mathcal{O}(\mathsf{pk}_{\mathrm{tmp}})$. It can then learn the result of the computation $y = P(\mathsf{msg})$ by invoking $\mathsf{HW}(\mathsf{ct}, \mathsf{ct}_{\mathrm{tmp}}, \sigma_P, \mathsf{vk}_{\mathsf{sign}})$. Correctness of the functional encryption scheme follows.

On the high level, the construction also seems secure: the input is encrypted using a semantically secure encryption scheme which can only be decrypted using $\mathsf{sk}_{\mathsf{pke}}$. Consecutively, $\mathsf{sk}_{\mathsf{pke}}$ is only revealed in the internal state of the hardware component HW, and is never seen by the decryptor in clear. Moreover, secret keys (σ_P's) cannot be forged by the security of the signature scheme. Unfortunately, this intuition is misleading. First of all, there is an obvious "man-in-the-middle" attack that the decryptor can perform by generating a temporary public/secret key pair $(\mathsf{pk}_{\mathrm{tmp}}, \mathsf{sk}_{\mathrm{tmp}})$ by itself and performing the rest of the functionality of the HW oracle on its own. This means that it can learn the message msg in clear. Moreover, using regular public-key encryption is not sufficient since its susceptible to "malleability" attacks, meaning that adversary may also generate related ciphertexts msg' and learn $P(\mathsf{msg}')$.

Second Attempt. To solve the above problems we turn our attention to stronger properties of the secure hardware and switch to using more powerful encryption schemes. First of all, every secure hardware HW is associated with public parameters. On any input, along with the corresponding output, HW also produces a cryptographic proof (signature) that can be used to verify the output of the computation with respect to the public parameters. We can now solve the "man-in-the-middle" attack by enforcing the decryptor to pass the proof to the authority oracle. The proof can be used to authenticate that the temporal public/secret key pair $(\mathsf{pk}_{\mathrm{tmp}}, \mathsf{sk}_{\mathrm{tmp}})$ was produced inside the secure

hardware and its internal state is hidden from the decryptor. We also change all encryption schemes to be secure against chosen-ciphertext attacks. This prevents any possible malleability by the decryptor.

Yet again, the above construction seems insufficient for the security proof. To explain the issue, we explain the intuition behind the simulation proof. In the ideal experiment, we need to simulate the ciphertext and the behavior of the oracles without knowing the input msg. Given $P(\mathsf{msg})$, the authority oracle $\mathcal{O}_{\mathsf{msk}}$ can send an encryption of $P(\mathsf{msg})$ along with $\mathsf{sk}_{\mathsf{pke}}$ (under the public key $\mathsf{pk}_{\mathrm{tmp}}$). We can then define the hardware oracle to output $P(\mathsf{msg})$, instead of computing it. However, to argue indistinguishability of $\mathsf{enc}_{\mathsf{pk}_{\mathrm{tmp}}}(\mathsf{sk}_{\mathsf{pke}}, 0)$[3] and $\mathsf{enc}_{\mathsf{pk}_{\mathrm{tmp}}}(\mathsf{sk}_{\mathsf{pke}}, P(\mathsf{msg}))$, we need to remove the key $\mathsf{sk}_{\mathrm{tmp}}$ from the internal state of the secure hardware. However, if we remove the secret key $\mathsf{sk}_{\mathrm{tmp}}$ from the internal state of the secure hardware, then it cannot decrypt $\mathsf{ct}_{\mathrm{tmp}}$ and we lose the correctness property! To solve this problem, we modify the construction to use a "dual-encryption" paradigm [NY90]. Using it, we create two encryption tracks and a special "mode" which can be used to substitute encryptions and switch computation to a second track that can then be used to satisfy correctness in the simulation experiment. This step also requires the encryption scheme to be (weakly) "robust" [ABN10]. We refer the reader to Section 3.2 for the detailed security proof.

Implementation and Evaluation We implement our construction on a Dell Inspiron laptop with Intel i7 processor supporting Software Guard Extensions (SGX) instruction set. Using Intel SGX CPUs, we created a secure container on the decryptor's node. The secure container is an encrypted memory region of 128 MB. Only the CPU can access the code and the data residing inside this container, and external programs can communicate via an explicitly defined API set. We implemented basic algorithms of FE and performed their benchmarking. Our implementation supports issuing of secret keys for basic arithmetic programs (sum, multiplication, division, modulo), mean and simple linear regression function.

Our results show that our construction is very practical for many real world applications. For our evaluations, we took an input message of 27 MB (so that we do not have to swap chunks of data in and out of the enclave). Setup and key-generation procedures essentially correspond to a few standard encryption/signing invocations, which takes no more than a few milliseconds on a modern laptop. Encryption runtime is proportional to the overhead of a standard encryption over raw data. In our experiments, we encrypted 27 MB messages in 22 milliseconds. Decryption for arithmetic operations and linear regression took between 130 and 180 milliseconds.

1.2 Discussion

Need for secure hardware One could ask what is the best that we can achieve without the secure hardware when the decryptor is just given access to the $\mathcal{O}_{\mathsf{msk}}$ oracle. To our knowledge, the best possible "crypto-only" approach to achieve our goal is to combine fully-homomorphic encryption and SNARKs/NIZKs, similar to the second construction in the work of Chung, Katz and Zhou [CKZ13]. However, in their approach, the decryptor needs to transmit data of size linear in size of the input message (ciphertext) to the oracle and the complexity of verification of SNARKs depends on the instance length. Hence, the oracle *does not* run in time independent of the length of the input message, desired by our model. If the input is large (database of medical records), this approach becomes infeasible. Moreover, the computational overhead of FHE and SNARKs is currently very high for general functions to be used in practice.

Need for the $\mathcal{O}_{\mathsf{msk}}$ oracle An orthogonal question to ask is what is the best that we can achieve with just the secure hardware and without the $\mathcal{O}_{\mathsf{msk}}$ oracle. Or can we even achieve simulation secure functional encryption with just the secure hardware in the form that we have defined (and the model that we believe captures the new generation of trusted computing designs such as SGX)? Informally, the following argument rules out the possibility of this. Only secure hardware seems insufficient because one would need to pre-program (compress) arbitrary number of outputs $P_i(\mathsf{msg})$ into the secure hardware to achieve simulation security. Given that secure hardware has a fixed memory bound (more formally, the secure hardware setup takes as input an aux string of some fixed size), it is impossible to compress many $P_i(\mathsf{msg})$ into this bound ($|\mathsf{aux}|$) for some function families [BSW12, AGVW13]. Moreover, even if one were to assume that the secure hardware does not have a fixed memory bound (and one can initiate it with an arbitrary long aux string), in the security game, the adversary would need to declare all inputs and functions before

[3] We pad an encryption with 0s to match the length.

the game begins so that the simulator can potentially pre-program $P_i(\mathsf{msg})$'s into aux. This results in a much weaker "selective" security notion, undesired in practice.

Bypassing FE Impossibility Results A very similar argument also explains why we are able to bypass the impossibility results of FE using $\mathcal{O}_{\mathsf{msk}}$. The core intuition for why existing simulation-based FE models are impossible [BSW12, AGVW13, CIJ+13] comes down to a "compression argument". Intuitively, it says that it is impossible to compress large number of strings $(x_1, \ldots, x_n)$ into a few succinct strings $(y_1, \ldots, y_k)$ for k much smaller than n. This becomes relevant in the simulation of functional encryption, because a simulator, given many function outputs $(P_1(\mathsf{msg}), \ldots, P_n(\mathsf{msg}))$ adaptively, must be able to produce a few public parameters, secret keys and a ciphertext that can be used to get all n output values. If n output values are pseudorandom (or have a lot of entropy), then it is impossible to compress them into few strings of fixed sizes. In our model, however, the decryptor is equipped with oracle access to the authority. The decryptor makes a single call to this oracle for each decryption invocation. Hence, we can hide the results of the computations $P_i(\mathsf{msg})$ in *each oracle response*, therefore without needing to compress all of them into few strings. This is possible even with the strict efficiency constraints which the oracle is subjected to.

Need for provable security Even with the use of basic crypto primitives as building blocks, in addition to a secure hardware, proving the security of our construction is not straightforward. As history shows, there are many subtleties that come up when composing multiple standard building blocks into complex protocols. For example, many TLS/SSL protocols that seemed secure were later identified with problems. Only recently, formal treatment of these protocols enabled their better understanding. Similarly, quantifying the security formally has become very important and relevant for the cryptographic constructions that enable "computations over encrypted data", especially in light of the recent attacks on systems such as CryptDB [PRZB11, NKW15] and Mylar [PSV+14, GMN+16].

In our construction, we first need to resolve to the "dual-encryption" proof technique, as explained in the above paragraphs. In addition, we face similar problems as those that arise in TLS security proofs: during the protocol execution, some auxiliary information about the shared secret is leaked to the adversary from the handshake protocol. We overcome these problems by a careful selection of the proof hybrid sequence.

Side-channel leakage Our construction assumes black box access to the secure hardware. One limitation of Intel SGX, which we use to instantiate secure hardware, is that it leaks program access patterns at 4KB page granularity [CD16, XCP15]. For us, it means that the programs we run inside the encrypted memory can possibly be subjected to side-channel attacks based on the access pattern. Moreover, since programs may run in different time depending on an input, timing side-channels may be introduced in practice. To prevent these attacks, oblivious algorithms should be designed and implemented within the secure hardware for all the programs. Designing oblivious algorithms that does not leak sensitive information from run-time, memory access pattern and other side-channel information is an active research area [MLS+13, LHM+15, WNL+14, SZEA+16]. The overhead incurred to make a program oblivious is negligible for some programs but it is also orders of magnitude higher for some programs. Also the crypto libraries used inside SGX should also be oblivious. For instance, SGX leverages AES-NI instruction set which is side-channel resistant. In general, one should note that Oblivious RAM does not provide a ready made solution here and some work has to be done on top of it. This is because in ORAM we need a client controller who stores some secret data and interacts with the server based on this secret. To leverage ORAM *non-interactively*, we would need to place both the client and the server code inside secure hardware. While the server component is oblivious, this is not necessarily the case for the client component. One would need to make the client code oblivious, which we believe is an interesting follow-up research direction.

1.3 Other related works

As we mentioned, Chung, Katz and Zhou [CKZ13] proposed a way to bypass the impossibility results in functional encryption by the use of "hardware tokens". They model the hardware tokens as "simply deterministic oracles" (refer Definition 5 of [CKZ13]). In contrast, our construction is based on real-world secure hardware with explicitly defined security properties (as in [BPSW16]). It is not clear if one can replace their deterministic oracles with the secure

hardware and carry through the proof. In particular, our secure hardware by itself does not have the "programmability" property required to achieve simulation security, but the programmability of deterministic oracles is crucially used in CKZ13. One might wonder how our $\mathcal{O}_{\mathsf{msk}}$ oracle compares with their notion of hardware tokens.With an "oracle" being necessary due to the impossibility results, we made the functionality of the $\mathcal{O}_{\mathsf{msk}}$ oracle minimal. In our construction, $\mathcal{O}_{\mathsf{msk}}$ performs minimal crypto functionality: basic signing/encryption. (And it is an independent enclave without access to msk which runs the user-specified programs on user-specified inputs). Hence, it is relatively easier to implement the $\mathcal{O}_{\mathsf{msk}}$ functionality secure against side-channels, when compared to the powerful hardware tokens. Also from a theoretical perspective, $\mathcal{O}_{\mathsf{msk}}$ 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.

Naveed et al. [NAP+14] propose a related notion of FE called "controlled functional encryption". The main motivation of C-FE is to introduce an additional level of access control. The data source encrypts its data and uploads the "policy" of the ciphertext ct to the authority. A client (playing the role of the decryptor) can make a "key-request" to the authority for computing P over the decryption of a *specific* ciphertext ct. For every key-request for the ciphertext, the authority checks the ciphertext's policy to decide on answering the key-request. The similarity of C-FE with our notion is that there is an "authority" mediating every decryption[4]. Also, our construction could be modified to achieve controlled functional encryption (CFE) primitive, when the efficiency constraints are relaxed for the authority oracle such that they run in time independent on the length of the input but dependent on the function description length. The construction in [NAP+14] requires the authority to run in time proportional to the length of function description and input.

Organization In Section 2, we discuss our functional encryption model, basic cryptographic preliminaries needed for our construction (encryption, signatures), and secure hardware component. In Section 3, we describe our main construction and prove its security. In Section 4, we describe our implementation and evaluation results. In Section 5, we summarize our results and discuss future directions.

2 Preliminaries

2.1 Functional Encryption

We define functional encryption in an oracle assisted model of computation. We discuss the efficiency requirements of the oracle below.

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

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

- $\mathsf{FE.Setup}(1^\lambda)$: The setup algorithm takes as input the unary representation of the security parameter λ and outputs the master public key mpk and the master secret key msk.
- $\mathsf{FE.Keygen}(\mathsf{msk}, P)$: The key generation algorithm takes as input the master secret key msk and a program $P \in \mathcal{P}$ and outputs the secret key sk_P for P.
- $\mathsf{FE.Enc}(\mathsf{mpk}, \mathsf{msg})$: The encryption algorithm takes as input the master public key mpk and an input message $\mathsf{msg} \in \mathcal{M}$ and outputs a ciphertext ct.

[4]We will discuss the implications of this mediation for FE at the end of Section 2.1.

- $\mathsf{FE.Dec}^{\mathcal{O}_{\mathsf{msk}}(\cdot)}(\mathsf{sk}_P, \mathsf{ct})$: The decryption algorithm takes as input a secret key sk_P and a ciphertext ct and outputs $P(\mathsf{msg})$ or $\perp$. It has access to an oracle $\mathcal{O}_{\mathsf{msk}}(\cdot)$, which is subject to strict efficiency constraints defined below.

We envision that the oracle $\mathcal{O}_{\mathsf{msk}}$ is executed by the trusted authority that executes $\mathsf{FE.Setup}$, or a designated party that is given the master secret key.

Oracle Efficiency

We require that the runtime of oracle $\mathcal{O}_{\mathsf{msk}}(\cdot)$ is at most $\mathrm{poly}(\lambda, \ell_{\mathsf{out}})$, where ℓ_{out} is the bit-length of the maximum program output size over all $P \in \mathcal{P}$ and inputs $\mathsf{msg} \in \mathcal{M}$. In particular, the runtime must be independent on the input msg size and program P runtime and description length. It follows that the oracle *may not* receive input msg and program P from the decryptor, compute $P(\mathsf{msg})$ and return the return back to the decryptor.

Correctness

A functional encryption scheme $\mathcal{FE}$ is correct if for all $P \in \mathcal{P}$ and all $\mathsf{msg} \in \mathcal{M}$, the probability for

$$\Pr\left[\mathsf{FE.Dec}^{\mathcal{O}(\cdot)}\Big(\mathsf{FE.Keygen}(\mathsf{msk}, P), \mathsf{FE.Enc}(\mathsf{mpk}, \mathsf{msg})\Big) \neq P(\mathsf{msg})\right] = \mathrm{negl}(\lambda)$$

where $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$ and the probability is taken over the random coins of the probabilistic algorithms $\mathsf{FE.Setup}, \mathsf{FE.Keygen}, \mathsf{FE.Enc}$.

Security

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

Definition 2.1 (Security-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.

$\mathsf{Exp}^{\mathsf{real}}_{\mathcal{FE}}(1^\lambda)$:	$\mathsf{Exp}^{\mathsf{ideal}}_{\mathcal{FE}}(1^\lambda)$:
1. $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$	1. $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$
2. $(\mathsf{msg}) \leftarrow \mathcal{A}^{\mathsf{FE.Keygen}(\mathsf{msk}, \cdot)}(\mathsf{mpk})$	2. $(\mathsf{msg}) \leftarrow \mathcal{A}^{\mathcal{S}(\mathsf{msk}, \cdot)}(\mathsf{mpk})$
3. $\mathsf{ct} \leftarrow \mathsf{FE.Enc}(\mathsf{mpk}, \mathsf{msg})$	3. $\mathsf{ct} \leftarrow \mathcal{S}^{U_{\mathsf{msg}}(\cdot)}(1^\lambda, 1^{\lvert\mathsf{msg}\rvert})$
4. $\alpha \leftarrow \mathcal{A}^{\mathsf{FE.Keygen}(\mathsf{msk}, \cdot), \mathcal{O}_{\mathsf{msk}}(\cdot)}(\mathsf{mpk}, \mathsf{ct})$	4. $\alpha \leftarrow \mathcal{A}^{\mathcal{S}^{U_{\mathsf{msg}}(\cdot)}(\cdot)}(\mathsf{mpk}, \mathsf{ct})$
5. *Output* (msg, α)	5. *Output* (msg, α)

In the above experiment, oracle calls by $\mathcal{A}$ to both the key-generation and $\mathcal{O}_{\mathsf{msk}}$ oracles are simulated by the simulator $\mathcal{S}^{U_{\mathsf{msg}}(\cdot)}(\cdot)$. 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.*

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

There are a lot of subtleties involved in the FE definition. We remark a few here and refer the interested readers to the original papers for more details. In the above definition, the simulator is given access to msk. Most previous works [BSW12, AGVW13] let $\mathcal{S}$ simulate the public parameters. Hence, running the $\mathsf{FE.Setup}$ honestly and providing the msk to $\mathcal{S}$, as in our definition, actually makes the definition stronger. Also, α may contain all the inputs to the oracles and the corresponding outputs. 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 output many messages and providing him the ciphertext for all of them. Here, the simulator will have an oracle $U_{\mathsf{msg}_i}(\cdot)$ for every msg_i.[5]

2.2 Additional Basic Crypto Primitives

2.2.1 Secret key encryption

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

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

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

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

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

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

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

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

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

Definition 2.2 (IND-CPA security of a secret key encryption scheme). *Security is depicted by the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$.*

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

[5] In our model, the decryptor may query the $\mathcal{O}_{\mathsf{msk}}$ oracle during every run of decryption. Hence, the authority running the $\mathcal{O}_{\mathsf{msk}}$ oracle can track some metadata regarding the decryption of a ciphertext. We believe this leakage is minimal and acceptable for many real-world applications, partially due to our efficiency constraints on the oracle.

2.2.2 A signature scheme

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

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

$\mathsf{S.Sign}(\mathsf{sk}, \mathsf{msg})$ The signing algorithm takes in a signing key sk and a message $\mathsf{msg} \in \mathcal{M}$ and outputs the signature σ. We assume that σ also explicitly contains the message msg that is signed.

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

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

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

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

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

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

Definition 2.3 (EUF-CMA). *Consider the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$.*

1. *The challenger runs the* $\mathsf{S.KeyGen}$ *algorithm to obtain the key pair* $(\mathsf{sk}, \mathsf{vk})$*, and provides the verification key* vk *to the adversary.*
2. *Initialize* $\mathrm{query} = \{\}$.
3. *Now, whenever the adversary provides a query with a message* msg*, the challenger replies with* $\mathsf{S.Sign}(\mathsf{sk}, \mathsf{msg})$. *Also,* $\mathrm{query} = \mathrm{query} \cup \mathsf{msg}$.
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

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

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

2.2.3 Public key encryption

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

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

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

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

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

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

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

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

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

Definition 2.4 (IND-CCA2 security of a public key encryption scheme). *Consider the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$.*

1. *$\mathcal{C}$ runs the* $\mathsf{PKE.KeyGen}$ *algorithm to obtain a key pair* $(\mathsf{pk}, \mathsf{sk})$ *and gives* pk *to the adversary.*
2. *$\mathcal{A}$ provides adaptively chosen* ct *and get back* $\mathsf{PKE.Dec}(\mathsf{sk}, \mathsf{ct})$.
3. *$\mathcal{A}$ provides* $\mathsf{msg}_0, \mathsf{msg}_1$ *to $\mathcal{C}$.*
4. *$\mathcal{C}$ then runs* $\mathsf{PKE.Enc}(\mathsf{pk})$ *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* $\mathsf{PKE.Dec}(\mathsf{sk}, \mathsf{ct})$*, with a restriction that* $\mathsf{ct} \neq \mathsf{ct}^*$.
6. *$\mathcal{A}$ outputs its guess b'.*

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

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

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

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

Definition 2.5 ((Weak) robustness property of PKE). *A PKE scheme* PKE *has the (weak) robustness property if for all λ and* $\mathsf{msg} \in \mathcal{M}$*,*

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

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

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

2.3 Collision resistant hash functions

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

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

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

2.4 Secure Hardware

In our model, we assume parties running decryption have access to the secure hardware defined below. Our definition for secure hardware follows the model defined by Barbosa, Portela, Scerri, and Warinschi [BPSW16]. They use this model to construct basic cryptographic components such as secure channels and outsourced computation[6].

A secure hardware scheme HW for a class of programs $\mathcal{Q}$ consists of the following polynomial time algorithms.

- $\mathsf{HW.Setup}(1^\lambda, \mathsf{aux})$: The HW.Setup algorithm takes in the security parameter and an auxiliary initialization parameter aux. It outputs public parameters params along with a secret key $\mathsf{sk}_{\mathsf{HW}}$ and an initialization state init.st.

- $\mathsf{HW.Load}_{\mathsf{init.st}}(\mathsf{params}, Q)$: The HW.Load algorithm loads a program into a secure container. HW.Load takes as input a possibly non-deterministic program $Q \in \mathcal{Q}$ and some global parameters params. It first creates a secure container and loads Q into it with an initial state init.st. It outputs a handle hdl_Q.

- $\mathsf{HW.Run\&Attest}_{\mathsf{sk}_{\mathsf{HW}}}(\mathsf{hdl}_Q, \mathsf{in})$: This is the program execution algorithm which takes in a handle hdl_Q, corresponding to a container running the program Q, and an input in. Given access to the secret key $\mathsf{sk}_{\mathsf{HW}}$, it outputs a tuple $\phi := \big(\mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \pi\big)$, where $\mathsf{out} = Q(\mathsf{in})$, π is a proof that can be used to verify the output of the computation, tag_Q is a program tag that can be used to identify the program running inside the secure container[7].

- $\mathsf{HW.Verify}(\mathsf{params}, \phi)$: This is the attestation verification algorithm. HW.Verify takes as input params and $\phi = \big(\mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \pi\big)$. It outputs 1 if π is a valid proof that $Q(\mathsf{in}) = \mathsf{out}$ when the program Q is run inside a secure container. It outputs 0 otherwise.

All the algorithms except HW.Verify are probabilistic. Note that in the above definition, only HW.Run&Attest has access to the secret key $\mathsf{sk}_{\mathsf{HW}}$. Not even the programs running inside the secure containers have access to $\mathsf{sk}_{\mathsf{HW}}$. Also, note that we have omitted the nonce in the definition of HW.Run&Attest so that the definition is general enough to work for arbitrary attestation protocols/verifiers.

Correctness A HW scheme is correct if the following things hold: For all $Q \in \mathcal{Q}$ and all in in the input domain of Q,

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

- Correctness of Attest and Verify:

$$\Pr\Big[\mathsf{HW.Verify}\Big(\mathsf{params}, \phi\Big) = 0\Big] = \mathrm{negl}(\lambda)$$

where $(\mathsf{params}, \mathsf{sk}_{\mathsf{HW}}, \mathsf{init.st}) \leftarrow \mathsf{HW.Setup}(1^\lambda, \mathsf{aux})$ with any aux, $\mathsf{hdl}_Q \leftarrow \mathsf{HW.Load}_{\mathsf{init.st}}(\mathsf{params}, Q)$ and $\phi \leftarrow \mathsf{HW.Run\&Attest}_{\mathsf{sk}_{\mathsf{HW}}}(\mathsf{hdl}_Q, \mathsf{in})$ for $\phi = \big(\mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \pi\big)$. The probability is taken over the random coins of the probabilistic algorithms HW.Setup, HW.Load and HW.Run&Attest.

Security The security of the hardware, denoted by attestation unforgeability (AttUnf), is defined similarly to the unforgeability security of a signature scheme. Informally, it says that no adversary can produce a tuple $\phi = \big(\mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \pi\big)$ that verifies correctly and $\mathsf{out} = Q(\mathsf{in})$, when the inputs $(\mathsf{hdl}_Q, \mathsf{in})$ were queried never by it. The security of HW is formally defined by the following security game.

Definition 2.6 (AttUnf-HW). *Consider the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$.*

[6] Barbosa et al. defined a slightly weaker syntax for secure hardware (based on Intel SGX). From it, they built a more powerful remote attestation functionality which we use in our definition. We believe our definition resembles functionality/syntax provided by Intel SGX and Intel remote attestation service, combined.

[7] One may think of tag_Q as a cryptographic hash of the program code Q.

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 key* $\mathsf{sk_{HW}}$ *and an initialization string* $\mathsf{init.st}$*. It gives* params *to $\mathcal{A}$, and keeps* $\mathsf{sk_{HW}}$ *and* $\mathsf{init.st}$ *secret in the secure hardware.*
3. *$\mathcal{C}$ initializes a list* $\mathrm{query} = \{\}$.
4. *$\mathcal{A}$ can run* $\mathsf{HW.Load}$ *on any input* (params, Q) *of its choice and get back* hdl_Q.
5. *Also, $\mathcal{A}$ can run* $\mathsf{HW.Run\&Attest}$ *on input* $(\mathsf{hdl}_Q, \mathsf{in})$ *of its choice and get* $\phi := \big(\mathsf{tag}_Q, \mathsf{in}, \mathsf{out}, \pi\big)$*. For every run, $\mathcal{C}$ adds the tuple* $(\mathsf{tag}_Q, \mathsf{in}, \mathsf{out})$ *to the list* query.
6. *Finally, the adversary outputs* $\phi^* = (\mathsf{tag}^*_Q, \mathsf{in}^*, \mathsf{out}^*, \pi^*)$.

We say the adversary wins the above experiment if:

1. $\mathsf{HW.Verify}(\mathsf{params}, \phi^*) = 1$,
2. $(\mathsf{tag}^*_Q, \mathsf{in}^*, \mathsf{out}^*) \notin \mathrm{query}$

The HW *scheme is 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 π^* different from the query outputs, but it cannot be a proof for a different program or input or output. This definition resembles an existential unforgeability like notions.

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

- Any user only has black box access to these algorithms and hence hidden from the internal secret key $\mathsf{sk_{HW}}$, initial state $\mathsf{init.st}$ or intermediary states of the programs running inside secure containers.
- The output of the $\mathsf{HW.Run\&Attest}$ algorithm is succinct: it does not include the full program description, for instance.
- We also require the params and the handles hdl_Q to be *independent* of aux. In particular, for all $\mathsf{aux}, \mathsf{aux}'$,

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

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

A few aspects of our FE model Now that we have described the required preliminaries, we would like to discuss and reiterate a few aspects of our model. Our model has three pieces that differ from the classical FE crypto models defined. First, we equip decryptor nodes with a secure hardware component. Given that all newer Intel processors are equipped with such a component (SGX), we anticipate that this assumption will be real in the next few years. AMD and other research and industry teams are also working on enabling secure hardware (encrypted memory) containers in all future processors. Second, we allow a pre-processing phase for all the decryptor nodes. This phase is used to setup public parameters for each secure component, which can be used to authenticate inputs/outputs from programs running within it. Intel, for instance, already provides a Intel Attestation Service, that is used to distribute these public parameters and during verification/attestation [JSR+16]. Finally, we allow the decryptor to communicate with the authority oracle during decryption. Communication is restricted to short messages and also the authority oracle cannot compute the function on behalf of the decryptor. This communication is cheap in the real world and, as mentioned earlier, leveraged by us to bypass impossibility results.

3 FE from Secure Hardware

In this section we will provide our construction for functional encryption FE which supports a message domain $\mathcal{M}$ and a class of programs $\mathcal{P}$ producing outputs of some fixed length ℓ_{out} (programs with smaller outputs can always be padded). We also require that each program's runtime is a function of the length of the input i.e. for all inputs of a particular length a program takes the same time to produce an output. Our FE scheme makes use of the following primitives:

1. an IND-CPA secure secret key encryption scheme $\mathsf{E} = (\mathsf{E.KeyGen}, \mathsf{E.Enc}, \mathsf{E.Dec})$ with message domain $\mathcal{M}$,
2. an EUF-CMA secure signature scheme $\mathsf{S} = (\mathsf{S.KeyGen}, \mathsf{S.Sign}, \mathsf{S.Verify})$,
3. a collision resistant hash function family H,
4. an IND-CCA2 secure and weakly robust public key encryption scheme $\mathsf{PKE} = (\mathsf{PKE.KeyGen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$[8],
5. an AttUnf secure hardware scheme $\mathsf{HW} = (\mathsf{HW.Setup}, \mathsf{HW.Load}, \mathsf{HW.Run\&Attest}, \mathsf{HW.Verify})$.

First we will explain the pre-processing phase.

Pre-processing: A decryptor node, sets up its secure hardware components by running:

1. Run $\mathsf{HW.Setup}(1^\lambda, \perp)$ to get params, a secret key $\mathsf{sk}_{\mathsf{HW}}$ and the initialization state $\mathsf{init.st}$.
2. Parameters $\mathsf{sk}_{\mathsf{HW}}$ and $\mathsf{init.st}$ remain secretly stored inside the secure hardware, but params is made public.

We are now ready to define the main FE algorithms. params is implicitly given as input to all the FE algorithms.[9]

$\mathsf{FE.Setup}(1^\lambda)$**:** The setup algorithm takes in the security parameter and does the following:

1. Sample a pair of PKE public/secret keys.

$$(\mathsf{pk}_{\mathsf{pke}}, \mathsf{sk}_{\mathsf{pke}}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$$

2. Sample a pair of signing/verification keys.

$$(\mathsf{vk}_{\mathsf{sign}}, \mathsf{sk}_{\mathsf{sign}}) \leftarrow \mathsf{S.KeyGen}(1^\lambda)$$

3. Sample a hash function $H \leftarrow \mathsf{H.Gen}(1^\lambda)$.
4. Output the master public key $\mathsf{mpk} := (\mathsf{pk}_{\mathsf{pke}}, \mathsf{vk}_{\mathsf{sign}}, H)$ and the master secret key $\mathsf{msk} := (\mathsf{sk}_{\mathsf{pke}}, \mathsf{sk}_{\mathsf{sign}})$.

$\mathsf{FE.Enc}(\mathsf{mpk}, \mathsf{msg})$**:** The encryption algorithm takes as input the master public key mpk and a message $\mathsf{msg} \in \mathcal{M}$, and does the following:

1. Sample an ephemeral key $\mathsf{ek} \leftarrow \mathsf{E.KeyGen}(1^\lambda)$ for encrypting the message.
2. Encrypt the message using the ephemeral key.

$$\mathsf{ct}_m \leftarrow \mathsf{E.Enc}(\mathsf{ek}, \mathsf{msg})$$

3. Encrypt the ephemeral key and the hash of the ciphertext ct_m.

$$\mathsf{ct}_k \leftarrow \mathsf{PKE.Enc}\left(\mathsf{pk}_{\mathsf{pke}}, [\mathsf{ek}, H(\mathsf{ct}_m)]\right)$$

4. Output $\mathsf{ct} := (\mathsf{ct}_k, \mathsf{ct}_m)$.

[8] Technically, two different PKE schemes each having one of the two properties would suffice for our construction (i.e., an IND-CCA2 secure scheme and another IND-CPA secure scheme with the weak robustness property).

[9] Note that when multiple users will need run decryption, then each one of them needs to make its params public. The setup of the secure hardware may also come at any time during the execution of FE algorithms. We define it explicitly in the pre-processing phase for clarity.

$\mathsf{FE.Keygen}(\mathsf{msk}, P)$**:** The key generation algorithm takes in the master secret key msk and a program P and does the following:

1. Generate a random tag $\tau_P \xleftarrow{\$} \{0,1\}^\lambda$.
2. Obtain a signature σ_P of the program P along with its tag.

$$\sigma_P \leftarrow \mathsf{S.Sign}(\mathsf{sk}_{\mathsf{sign}}, [P, \tau_P])$$

3. Output $\mathsf{sk}_P := (\sigma_P, P, \tau_P)$.

$\mathsf{FE.Dec}^{\mathcal{O}_{\mathsf{msk}}(\cdot)}(\mathsf{sk}_P, \mathsf{ct})$**:** The decryption algorithm takes as input a secret key $\mathsf{sk}_P = (\sigma_P, P, \tau_P)$ and a ciphertext $\mathsf{ct} = (\mathsf{ct}_k, \mathsf{ct}_m)$. It has access to oracle $\mathcal{O}_{\mathsf{msk}}(\cdot)$, and secure hardware component HW. It proceeds as follows.

1. Create a secure container for the program $\mathsf{Prog}_{\mathsf{Dec}}$ by running $\mathsf{HW.Load}_{\mathsf{init.st}}(\mathsf{params}, \mathsf{Prog}_{\mathsf{Dec}})$. A succinct handle hdl is obtained after its successful execution. Informally from correctness perspective, $\mathsf{Prog}_{\mathsf{Dec}}$ performs two main tasks. First it generates ephemeral public keys ($\mathsf{pk}_{\mathrm{tmp}}$) and sends them along with ct_k to the $\mathcal{O}_{\mathsf{msk}}$ oracle. When the $\mathcal{O}_{\mathsf{msk}}$ oracle returns the encryption of ek under the ephemeral public keys, $\mathsf{Prog}_{\mathsf{Dec}}$ also takes in ct_m and P and outputs $P(\mathsf{msg})$. A formal description of $\mathsf{Prog}_{\mathsf{Dec}}$ is provided after $\mathsf{FE.Dec}$.
2. Start the decryption by invoking $\mathsf{Prog}_{\mathsf{Dec}}$ through

$$\phi \leftarrow \mathsf{HW.Run\&Attest}_{\mathsf{sk}_{\mathsf{HW}}}(\mathsf{hdl}, [\tau_P, \mathsf{ct}_k])$$

where

$$\phi = \big(H(\mathsf{Prog}_{\mathsf{Dec}}), [\tau_P, \mathsf{ct}_k], [\mathsf{pk}^1_{\mathrm{tmp}}, \mathsf{pk}^2_{\mathrm{tmp}}, \mathrm{ID}], \pi\big)$$

Here, π is a proof that $\mathsf{Prog}_{\mathsf{Dec}}$ is run inside the secure hardware and on input $[\tau_P, \mathsf{ct}_k]$ has produced the output $[\mathsf{pk}^1_{\mathrm{tmp}}, \mathsf{pk}^2_{\mathrm{tmp}}, \mathrm{ID}]$.

3. Transform ct_k into an encryption of ek under the $\mathsf{pk}_{\mathrm{tmp}}$'s by using the $\mathcal{O}_{\mathsf{msk}}$ oracle.[10]

$$(\psi, \sigma_\psi) \leftarrow \mathcal{O}_{\mathsf{msk}}(\phi)$$

4. Complete the decryption process by invoking $\mathsf{Prog}_{\mathsf{Dec}}$ again through

$$\phi' \leftarrow \mathsf{HW.Run\&Attest}\,(\mathsf{hdl}, [\mathsf{sk}_P, \mathsf{ct}_m, \psi, \sigma_\psi])$$

where

$$\phi' = (H(\mathsf{Prog}_{\mathsf{Dec}}), [\mathsf{sk}_P, \mathsf{ct}_m, \psi, \sigma_\psi], [\mathsf{result}], \pi')$$

5. Output result.

We will now formally define the program $\mathsf{Prog}_{\mathsf{Dec}}$ which is loaded inside the secure container. Note that the $\mathsf{Prog}_{\mathsf{Dec}}$ program description is not dependent on any of the FE parameters.

$\mathsf{Prog}_{\mathsf{Dec}}$: $\mathsf{Prog}_{\mathsf{Dec}}$ is a stateful algorithm and is initialized with the initial state $\mathsf{state} \triangleq \mathsf{init.st}$. $\mathsf{Prog}_{\mathsf{Dec}}$ has two possible entry points.[11]

- Initializing decryption: On input (τ_P, ct_k),

[10] Note that ϕ is succinct and does not include program description or the ciphertext ct. Hence, the interaction with $\mathcal{O}_{\mathsf{msk}}$ oracle is minimal.

[11] In our presentation, we give the non-oblivious version of $\mathsf{Prog}_{\mathsf{Dec}}$ for simplicity. Every step (or collection of steps) run by $\mathsf{Prog}_{\mathsf{Dec}}$ can be made oblivious to memory access pattern and timing leaks. The important thing that an authority ($\mathsf{FE.Keygen}$ algorithm) should ensure is that it should issue secret keys only for the oblivious versions of the programs for which secret key is requested.

1. If $\mathsf{pk}^1_{\mathrm{tmp}} = \perp$ in state, generate an ephemeral PKE public/secret key pair.

$$(\mathsf{pk}^1_{\mathrm{tmp}}, \mathsf{sk}^1_{\mathrm{tmp}}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$$

   Else, set $\mathsf{sk}^1_{\mathrm{tmp}} = \perp$.
2. If $\mathsf{pk}^2_{\mathrm{tmp}} = \perp$ in state, generate another pair.

$$(\mathsf{pk}^2_{\mathrm{tmp}}, \mathsf{sk}^2_{\mathrm{tmp}}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$$

   Else, set $\mathsf{sk}^2_{\mathrm{tmp}} = \perp$.
3. Output $(\mathsf{pk}^1_{\mathrm{tmp}}, \mathsf{pk}^2_{\mathrm{tmp}}, \mathrm{ID})$, where $\mathrm{ID} \xleftarrow{\$} \{0,1\}^\lambda$ identifies this run of $\mathsf{Prog}_{\mathsf{Dec}}$.
4. The state state that is passed onto the second step is $(\mathsf{sk}^1_{\mathrm{tmp}}, \mathsf{sk}^2_{\mathrm{tmp}}, \mathrm{ID})$.

- Completing decryption: On input $(\mathsf{sk}_P, \mathsf{ct}_m, \psi, \sigma_\psi)$ for $\psi = (\mathrm{ID}, H(\mathsf{ct}_m), \mathsf{ct}^1_{\mathrm{tmp}}, \mathsf{ct}^2_{\mathrm{tmp}})$,

1. Retrieve $\mathsf{state} = (\mathsf{sk}^1_{\mathrm{tmp}}, \mathsf{sk}^2_{\mathrm{tmp}}, \mathrm{ID})$.
2. Verify the secret key signature sk_P by running $\mathsf{S.Verify}(\mathsf{vk}_{\mathsf{sign}}, \sigma_P, [P, \tau_P])$. If the verification fails output $\perp$, else proceed.
3. Verify the signature from $\mathcal{O}_{\mathsf{msk}}$: $\mathsf{S.Verify}(\mathsf{vk}_{\mathsf{sign}}, \sigma_\psi, \psi)$. If the verification fails output $\perp$, else proceed.
4. Compare the ID in state and ψ. If they do not match output $\perp$, else proceed.
5. Check the validity of ct_m by first computing $H(\mathsf{ct}_m)$ with ct_m from the input and then comparing it with the $H(\mathsf{ct}_m)$ in ψ. If they do not match output $\perp$, else proceed.
6. Decrypt $\mathsf{ct}^1_{\mathrm{tmp}}$.

$$[\mathsf{ek}, \mathrm{val}, \mathsf{mode}]/\perp \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}^1_{\mathrm{tmp}}, \mathsf{ct}^1_{\mathrm{tmp}})$$

7. If $\perp$, decrypt $\mathsf{ct}^2_{\mathrm{tmp}}$.

$$[\mathsf{ek}, \mathrm{val}, \mathsf{mode}]/\perp \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}^2_{\mathrm{tmp}}, \mathsf{ct}^2_{\mathrm{tmp}})$$

   If $\perp$ output $\perp$, else proceed.
8. Now use the ek obtained to decrypt ct_m.

$$\mathsf{msg} \leftarrow \mathsf{E.Dec}(\mathsf{ek}, \mathsf{ct}_m)$$

9. Compute the function evaluation $P(\mathsf{msg})$.
10. If $\mathsf{mode} = 0$, output $P(\mathsf{msg})$, else output val.

Now we will describe the $\mathcal{O}_{\mathsf{msk}}$ oracle.

$\mathcal{O}_{\mathsf{msk}}$ oracle: The $\mathcal{O}_{\mathsf{msk}}$ oracle is run by the authority primarily to help the decryption algorithm by providing the ephemeral key ek. It takes as input

$$\phi = \left(H(\mathsf{Prog}_{\mathsf{Dec}}), [\tau_P, \mathsf{ct}_k], [\mathsf{pk}^1_{\mathrm{tmp}}, \mathsf{pk}^2_{\mathrm{tmp}}, \mathrm{ID}], \pi\right)$$

and works as follows:

1. Verify the attestation π by running $\mathsf{HW.Verify}(\phi)$. If the output is 0 output $\perp$, else proceed.
2. Decrypt ct_k using the $\mathsf{sk}_{\mathsf{pke}}$ from msk.

$$[\mathsf{ek}, H(\mathsf{ct}_m)] \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}_{\mathsf{pke}}, \mathsf{ct}_k)$$

3. Encrypt $[\mathsf{ek}, \mathrm{val}, \mathsf{mode}]$ under $\mathsf{pk}^1_{\mathrm{tmp}}$ for $\mathrm{val} = 0^{\ell_{\mathsf{out}}}$ and $\mathsf{mode} = 0$.

$$\mathsf{ct}^1_{\mathrm{tmp}} \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}^1_{\mathrm{tmp}}, [\mathsf{ek}, 0^{\ell_{\mathsf{out}}}, 0])$$

4. Encrypt $[\mathsf{ek}, 0^{\ell_{\mathsf{out}}}, 0]$ under $\mathsf{pk}^2_{\mathrm{tmp}}$.

$$\mathsf{ct}^2_{\mathrm{tmp}} \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}^2_{\mathrm{tmp}}, [\mathsf{ek}, 0^{\ell_{\mathsf{out}}}, 0])$$

5. Obtain a signature on $\psi = (\mathrm{ID}, H(\mathsf{ct}_m), \mathsf{ct}^1_{\mathrm{tmp}}, \mathsf{ct}^2_{\mathrm{tmp}})$.

$$\sigma_\psi \leftarrow \mathsf{S.Sign}(\mathsf{sk}_{\mathsf{sign}}, \psi)$$

6. Output σ_ψ.

3.1 Correctness

We will informally argue the correctness of our scheme for an honest decryptor. On input $(\mathsf{sk}_P, \mathsf{ct})$ to FE.Dec, where $\mathsf{ct} \leftarrow \mathsf{FE.Enc}(\mathsf{mpk}, \mathsf{msg})$ and $\mathsf{sk}_P \leftarrow \mathsf{FE.Keygen}(\mathsf{msk}, P)$,

1. The correctness of HW will ensure that the correct ct_k is passed to the $\mathcal{O}_{\mathsf{msk}}$ oracle.
2. Then, the correctness of PKE will ensure that $\mathcal{O}_{\mathsf{msk}}$ on decryption gets the correct ek which is returned back encrypted under $\mathsf{pk}^1_{\mathrm{tmp}}$ (and $\mathsf{pk}^2_{\mathrm{tmp}}$).
3. Again, the correctness of HW and PKE will ensure that the second run of $\mathsf{Prog}_{\mathsf{Dec}}$ will get back ek on decrypting $\mathsf{ct}^1_{\mathrm{tmp}}$. This can be used to get the msg due to the correctness of E, and from msg, $P(\mathsf{msg})$ can be calculated.

Also note that the $\mathcal{O}_{\mathsf{msk}}$ oracle satisfies the efficiency requirements. Each component of its input ϕ has length $\mathrm{poly}(\lambda)$. Also, every step of $\mathcal{O}_{\mathsf{msk}}$ runs in time polynomial in the length of its inputs, other than the two PKE.Enc steps whose running time additionally depends on ℓ_{out}. Hence, the total running time of $\mathcal{O}_{\mathsf{msk}}$ does not depend on the length of P or in, or the running time of P.

3.2 Security

Theorem 3.1 *If* E *is an IND-CPA secure secret key encryption scheme,* S *is an EUF-CMA secure signature scheme,* PKE *is an IND-CCA2 secure, weakly robust public key encryption scheme and* HW *is an AttUNF secure hardware scheme, then FE is a secure functional encryption scheme according to Definition 2.1.*

Proof. We will construct a simulator $\mathcal{S}$ for the FE security game in Definition 2.1. $\mathcal{S}$ 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^*. $\mathcal{S}$ has to simulate a ciphertext corresponding to the challenge message msg^* along with the pre-processing phase, KeyGen algorithm and the $\mathcal{O}_{\mathsf{msk}}$ oracle. It does them as follows:

Pre-processing phase: $\mathcal{S}$ simulates the pre-processing phase similar to the real world except that it inputs a pk sampled using PKE.KeyGen in aux which will be set as $\mathsf{pk}^1_{\mathrm{tmp}}$ in init.st.

1. Run $\mathsf{HW.Setup}(1^\lambda, [\mathsf{pk}, \perp])$ to get params, a secret key $\mathsf{sk}_{\mathsf{HW}}$ and the initialization state init.st, where pk is provided by environment. Here, init.st has $\mathsf{pk}^1_{\mathrm{tmp}} = \mathsf{pk}$ and $\mathsf{pk}^2_{\mathrm{tmp}} = \perp$.
2. Parameters $\mathsf{sk}_{\mathsf{HW}}$ and init.st remain secretly stored inside the secure hardware, and params is made public.

$\mathsf{FE.Enc}^*(\mathsf{mpk})$**:** This algorithm is used by $\mathcal{S}$ to simulate the challenge ciphertext for the challenge message msg^* provided by the adversary $\mathcal{A}$. Enc^* maintains a list $\mathcal{K}$ and does the following:

1. Obtain ek by running $\mathsf{E.KeyGen}(1^\lambda)$.
2. Obtain the ciphertext ct^*_m by encrypting a string of zeros of length $|\mathsf{msg}^*|$.

$$\mathsf{ct}^*_m \leftarrow \mathsf{E.Enc}(\mathsf{ek}, 0^{|\mathsf{msg}^*|})$$

3. Encrypt a string of zeros again along with $H(\mathsf{ct}^*_m)$.

$$\mathsf{ct}^*_k \leftarrow \mathsf{PKE.Enc}\left(\mathsf{pk}_{\mathsf{pke}}, [0^{|\mathsf{ek}|}, H(\mathsf{ct}^*_m)]\right)$$

4. Output $\mathsf{ct}^* := (\mathsf{ct}^*_k, \mathsf{ct}^*_m)$.

In addition, $\mathcal{S}$ stores $(\mathsf{ct}^*_k, H(\mathsf{ct}^*_m))$ in the list $\mathcal{K}$.

$\mathsf{FE.Keygen}^*(\mathsf{msk}, P)$**:** $\mathcal{S}$ has access to the master secret key msk. So the simulated KeyGen^* is run the same way as the real one as follows:

1. Generate a random tag $\tau_P \xleftarrow{\$} \{0,1\}^\lambda$.
2. Obtain a signature of the program P along with its tag.

$$\sigma_P \leftarrow \mathsf{S.Sign}(\mathsf{sk}_{\mathsf{sign}}, [P, \tau_P])$$

3. Output $\mathsf{sk}_P := (\sigma_P, P, \tau_P)$.

In addition, $\mathcal{S}$ queries $U_{\mathsf{msg}^*}(P)$ to get $P(\mathsf{msg}^*)$ and store the tuple $(\tau_P, P(\mathsf{msg}^*))$ in a list $\mathcal{R}$. For the queries made before $\mathcal{A}$ provides msg^*, $\mathcal{S}$ stores the programs along with their tags and later fill their entries in $\mathcal{R}$ after $\mathcal{A}$ provides msg^*.

$\mathcal{O}^*_{\mathsf{msk}}(\phi)$**:** $\mathcal{S}$ simulates the $\mathcal{O}^*_{\mathsf{msk}}$ oracle using its oracle access to $U_{\mathsf{msg}^*}(\cdot)$. Remember that $U_{\mathsf{msg}^*}(P) = P(\mathsf{msg}^*)$. On an input

$$\phi = \left(H(\mathsf{Prog}_{\mathsf{Dec}}), [\tau_P, \mathsf{ct}_k], [\mathsf{pk}^1_{\mathrm{tmp}}, \mathsf{pk}^2_{\mathrm{tmp}}, \mathrm{ID}], \pi\right)$$

$\mathcal{S}$ runs the following steps:

1. Run the $\mathsf{HW.Verify}$ oracle on π. If its output is 0 output $\perp$, else proceed.
2. If $\mathsf{ct}_k \in \mathcal{K}$ (which, as explained later, with very high probability captures the case that decryption is being run on an encryption of msg^*):
   (a) Search $\mathcal{R}$ for τ_P to obtain $P(\mathsf{msg}^*)$ and let $\mathrm{val} = P(\mathsf{msg}^*)$.
   (b) Encrypt a string of zeros of length $|\mathsf{ek}|$ with $\mathsf{mode} = 1$ under $\mathsf{pk}^1_{\mathrm{tmp}}$.

$$\mathsf{ct}^1_{\mathrm{tmp}} \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}^1_{\mathrm{tmp}}, [0^{|\mathsf{ek}|}, \mathrm{val}, 1])$$

   (c) Encrypt the same string under $\mathsf{pk}^2_{\mathrm{tmp}}$.

$$\mathsf{ct}^2_{\mathrm{tmp}} \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}^2_{\mathrm{tmp}}, [0^{|\mathsf{ek}|}, \mathrm{val}, 1])$$

   Else, the decryption is being run on an encryption of some $\mathsf{msg} \neq \mathsf{msg}^*$. Here, do as in the real world:
   (a) Decrypt ct_k using the $\mathsf{sk}_{\mathsf{pke}}$.

$$[\mathsf{ek}, H(\mathsf{ct}_m)] \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}_{\mathsf{pke}}, \mathsf{ct}_k)$$

   (b) Encrypt $[\mathsf{ek}, 0^{\ell_{\mathsf{out}}}, 0]$ under $\mathsf{pk}^1_{\mathrm{tmp}}$.

$$\mathsf{ct}^1_{\mathrm{tmp}} \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}^1_{\mathrm{tmp}}, [\mathsf{ek}, 0^{\ell_{\mathsf{out}}}, 0])$$

   (c) Encrypt $[\mathsf{ek}, 0^{\ell_{\mathsf{out}}}, 0]$ under $\mathsf{pk}^2_{\mathrm{tmp}}$.

$$\mathsf{ct}^2_{\mathrm{tmp}} \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}^2_{\mathrm{tmp}}, [\mathsf{ek}, 0^{\ell_{\mathsf{out}}}, 0])$$

3. Obtain a signature on $\psi = (\mathrm{ID}, H(\mathsf{ct}_m), \mathsf{ct}^1_{\mathrm{tmp}}, \mathsf{ct}^2_{\mathrm{tmp}})$.

$$\sigma_\psi \leftarrow \mathsf{S.Sign}(\mathsf{sk}_{\mathsf{sign}}, \psi)$$

4. Output σ_ψ.

Now, for this polynomial time simulator $\mathcal{S}$ described above, we have to show that for experiments in Definition 2.1,

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

We prove this by showing that the view of the adversary $\mathcal{A}$ in the real world is computationally indistinguishable from the view in the ideal world. It can be easily checked that the algorithms KeyGen^*, Enc^* and oracle $\mathcal{O}^*_{\mathsf{msk}}$ simulated by S correspond to the ideal world specifications of Definition 2.1. The most important thing to notice is that no information about msg^* is used by S other than those provided by the $U_{\mathsf{msg}^*}(\cdot)$ oracle. We will prove through a series of hybrids that $\mathcal{A}$ cannot distinguish between the real and the ideal world algorithms and oracles. In this paper, we have removed some details which can be easily worked out. A more detailed proof is available in the full version.

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

Hybrid 1 As in **Hybrid** 0, except that $\mathsf{FE.Keygen}^*$ run by $\mathcal{S}$ is used to generate secret keys instead of $\mathsf{FE.Keygen}$. And Enc stores $(\mathsf{ct}^*_k, H(\mathsf{ct}^*_m))$ used in the challenge ciphertext for msg^* in the list $\mathcal{K}$.

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

Hybrid 2 As in **Hybrid** 1, except that during the pre-processing phase $\mathcal{S}$ sets $\mathsf{pk}^2_{\mathrm{tmp}}$ to a value provided by the environment.

The following claims will be useful in proving the indistinguishability of these two hybrids and in the rest of the proof.

Claim 3.1.1 *Any σ_ψ input to the second run of $\mathsf{Prog}_{\mathsf{Dec}}$, for which $\mathsf{S.Verify}$ does not output 0, is a valid signature on $\psi = (\mathrm{ID}, H(\mathsf{ct}_m), \mathsf{ct}^1_{\mathrm{tmp}}, \mathsf{ct}^2_{\mathrm{tmp}})$ generated by $\mathcal{O}_{\mathsf{msk}}$.*

- We will use the EUF-CMA security of the signature scheme S to prove this claim.
- If $\mathcal{A}$ violates this claim, we will construct an adversary $\mathcal{A}^*$ which uses $\mathcal{A}$ to break the EUF-CMA security. $\mathcal{A}^*$ gets vk^* from its EUF-CMA challenger and it makes $\mathcal{S}$ set $\mathsf{vk}_{\mathsf{sign}} = \mathsf{vk}^*$ in mpk. When $\mathcal{S}$ needs to sign $[P, \tau_P]$ in $\mathsf{FE.Keygen}^*$ or ψ in $\mathcal{O}_{\mathsf{msk}}$, $\mathcal{A}^*$ uses the signing oracle provided by its challenger to get the signature. $\mathcal{S}$ can simulate the other components of the FE game since they do not require any information about $\mathsf{sk}_{\mathsf{sign}}$. Now, if $\mathcal{A}$ produces a forged σ_ψ $\mathcal{A}^*$ forwards it to its EUF-CMA challenger as its forgery.
- This claim ensures any ψ input to the second part of $\mathsf{Prog}_{\mathsf{Dec}}$ algorithm originates from the $\mathcal{O}_{\mathsf{msk}}$ oracle.

Claim 3.1.2 *Any ϕ input to the $\mathcal{O}_{\mathsf{msk}}$ oracle*

$$\phi = \left(H(\mathsf{Prog}_{\mathsf{Dec}}), [\tau_P, \mathsf{ct}_k], [\mathsf{pk}^1_{\mathrm{tmp}}, \mathsf{pk}^2_{\mathrm{tmp}}, \mathrm{ID}], \pi\right)$$

for which $\mathsf{HW.Verify}$ does not output 0 has the program hash $H(\mathsf{Prog}_{\mathsf{Dec}})$, inputs $[\tau_P, \mathsf{ct}_k]$ and outputs $[\mathsf{pk}^1_{\mathrm{tmp}}, \mathsf{pk}^2_{\mathrm{tmp}}, \mathrm{ID}]$ as output by an instance of the $\mathsf{HW.Run\&Attest}$ oracle for $\mathsf{Prog}_{\mathsf{Dec}}$.

- We will use the AttUNF security of HW to prove this claim.

- If $\mathcal{A}$ has forged a π to attest a non-queried

$$\big(H(\mathsf{Prog}_{\mathsf{Dec}}), [\tau_P, \mathsf{ct}_k], [\mathsf{pk}^1_{\mathrm{tmp}}, \mathsf{pk}^2_{\mathrm{tmp}}, \mathrm{ID}]\big)$$

  which the HW.Verify oracle approves, we will construct an adversary $\mathcal{A}^*$ for AttUNF security game. First, when $\mathcal{S}$ has to run the pre-processing phase with aux, $\mathcal{A}^*$ provides the aux to its AttUNF challenger and gets $\mathsf{sk}^*_{\mathsf{HW}}$ and $\mathsf{init.st}^*$ set in the hardware for $\mathcal{A}$. Now $\mathcal{A}$ can run the other algorithms of HW with the challenge parameters and when $\mathcal{A}$ forges a π $\mathcal{A}^*$ can look it up from the transcript α and forward it to its challenger.

- This claim ensures that any valid query to $\mathcal{O}_{\mathsf{msk}}$ originates from running the $\mathsf{Prog}_{\mathsf{Dec}}$ algorithm.

Due to Claim 3.1.1 and Claim 3.1.2, $\mathcal{A}$ cannot modify any communication between $\mathsf{Prog}_{\mathsf{Dec}}$ and $\mathcal{O}_{\mathsf{msk}}$. Here, the first part of $\mathsf{Prog}_{\mathsf{Dec}}$ generates an ID and sends it to the $\mathcal{O}_{\mathsf{msk}}$ oracle as a part of ϕ, and the oracle sends it back to the second part of $\mathsf{Prog}_{\mathsf{Dec}}$ as a part of ψ. Hence, $\mathcal{A}$ cannot invoke $\mathsf{Prog}_{\mathsf{Dec}}$ (without aborting during verification) with an output of $\mathcal{O}_{\mathsf{msk}}$ intended for a different instance of $\mathsf{Prog}_{\mathsf{Dec}}$.

Now, let us complete the proof of indistinguishability between **Hybrid** 1 and **Hybrid** 2. In **Hybrid** 1, $\mathsf{Prog}_{\mathsf{Dec}}$ has access to the "correct" $\mathsf{sk}^2_{\mathrm{tmp}}$ but in **Hybrid** 2 $\mathsf{sk}^2_{\mathrm{tmp}} = \perp$. Hence, the hybrids differ only in the decryption of $\mathsf{ct}^2_{\mathrm{tmp}}$, which will be run by the second part of $\mathsf{Prog}_{\mathsf{Dec}}$ only when $\mathsf{PKE.Dec}(\mathsf{sk}^1_{\mathrm{tmp}}, \mathsf{ct}^1_{\mathrm{tmp}})$ outputs $\perp$. But, according to Claim 3.1.1 and Claim 3.1.2, $\mathcal{A}$ cannot modify the $\mathsf{pk}^1_{\mathrm{tmp}}$ in ϕ and the $\mathsf{ct}^1_{\mathrm{tmp}}$ in ψ. Hence, $\mathsf{Prog}_{\mathsf{Dec}}$ will have the correct $\mathsf{sk}^1_{\mathrm{tmp}}$ corresponding to $\mathsf{pk}^1_{\mathrm{tmp}}$ to decrypt $\mathsf{ct}^1_{\mathrm{tmp}}$, and hence $\mathsf{ct}^2_{\mathrm{tmp}}$ will not be used in both the hybrids. Thus, **Hybrid** 1 and **Hybrid** 2 are indistinguishable.

Hybrid 3 As in **Hybrid** 2, except that $\mathcal{O}_{\mathsf{msk}}$ oracle encrypts $[0^{|\mathsf{ek}|}, P(\mathsf{msg}^*), 1]$ to generate $\mathsf{ct}^2_{\mathrm{tmp}}$ instead of the $[\mathsf{ek}, 0^{\ell_{\mathsf{out}}}, 0]$ if the ct_k in ϕ is $\mathsf{ct}^*_k \in \mathcal{K}$. Here, $P(\mathsf{msg}^*)$ is obtained from the list $\mathcal{R}$.

The indistinguishability between the hybrids depend on the IND-CPA security of PKE. The hybrids differ only in the messages encrypted in $\mathsf{ct}^2_{\mathrm{tmp}}$, which will be used by the second part of $\mathsf{Prog}_{\mathsf{Dec}}$ only when $\mathsf{PKE.Dec}(\mathsf{sk}^1_{\mathrm{tmp}}, \mathsf{ct}^1_{\mathrm{tmp}})$ outputs $\perp$. But, by the same argument as above we can show that $\mathsf{ct}^2_{\mathrm{tmp}}$ will not be used in both the hybrids.

Now, we will show that if $\mathcal{A}$ distinguishes between **Hybrid** 2 and **Hybrid** 3, we can construct an adversary $\mathcal{A}^*$ for the IND-CPA security game. $\mathcal{A}^*$ first gets a public key pk^* from its challenger $\mathcal{C}_{\mathsf{PKE}}$. During the pre-processing stage, $\mathcal{S}$ sets $\mathsf{aux} = [\perp, \mathsf{pk}^*]$. Then, $\mathcal{A}^*$ provides two messages $[\mathsf{ek}, 0^{\ell_{\mathsf{out}}}, 0]$ and $[0^{|\mathsf{ek}|}, P(\mathsf{msg}^*), 1]$ to $\mathcal{C}_{\mathsf{PKE}}$ where ek is got by decrypting ct_k. $\mathcal{A}^*$ gets back the challenge ciphertext ct^*. $\mathcal{S}$ sets $\mathsf{ct}^2_{\mathrm{tmp}} = \mathsf{ct}^*$. $\mathcal{S}$ can simulate the other elements of the FE security game without knowing whether ct^* encrypts the first message or the second. If ct^* is an encryption of $[\mathsf{ek}, 0^{\ell_{\mathsf{out}}}, 0]$, then the view of $\mathcal{A}$ is identical to the view in **Hybrid** 2 and if ct^* is an encryption of $[0^{|\mathsf{ek}|}, P(\mathsf{msg}^*), 1]$, then the view of $\mathcal{A}$ is identical the view in **Hybrid** 3. Thus, if $\mathcal{A}$ distinguishes between **Hybrid** 2 and **Hybrid** 3, $\mathcal{A}^*$ breaks the IND-CPA security of PKE.

Hybrid 4 As in **Hybrid** 3, except that during the pre-processing phase $\mathcal{S}$ sets $\mathsf{pk}^2_{\mathrm{tmp}}$ back to $\perp$ and $\mathsf{pk}^1_{\mathrm{tmp}}$ to a value provided by environment.

Due to these changes, the second part of $\mathsf{Prog}_{\mathsf{Dec}}$ will output $\perp$ when $\mathsf{ct}^1_{\mathrm{tmp}}$ is decrypted, by the weak robustness property of PKE and the authenticity of the communication between $\mathsf{Prog}_{\mathsf{Dec}}$ and $\mathcal{O}_{\mathsf{msk}}$ (Claims 3.1.1 and 3.1.2). It will start decrypting $\mathsf{ct}^2_{\mathrm{tmp}}$ and it will use the correct $\mathsf{sk}^2_{\mathrm{tmp}}$ to do it. But, when $\mathsf{ct}_k \in \mathcal{K}$ (corresponding to the challenge msg^*) $\mathsf{ct}^2_{\mathrm{tmp}}$ will be an encryption of $[0^{|\mathsf{ek}|}, P(\mathsf{msg}^*), 1]$, else $\mathsf{ct}^2_{\mathrm{tmp}}$ will be an encryption of $[\mathsf{ek}, 0^{\ell_{\mathsf{out}}}, 0]$. Hence, when $\mathsf{ct}_k \notin \mathcal{K}$ the second part of $\mathsf{Prog}_{\mathsf{Dec}}$ will follow similar procedures to produce the output in both the hybrids (though **Hybrid** 3 will use $\mathsf{ct}^1_{\mathrm{tmp}}$ and **Hybrid** 4 will use $\mathsf{ct}^2_{\mathrm{tmp}}$). But when $\mathsf{ct}_k \in \mathcal{K}$, it will have different modes to work with. In **Hybrid** 4, $\mathsf{Prog}_{\mathsf{Dec}}$ will just output $P(\mathsf{msg}^*)$ got from $\mathsf{ct}^2_{\mathrm{tmp}}$ since $\mathsf{mode} = 1$. But in **Hybrid** 3, $\mathsf{Prog}_{\mathsf{Dec}}$ will do as in the real world: find ek from $\mathsf{ct}^1_{\mathrm{tmp}}$ and then use it to get msg and then find $P(\mathsf{msg})$. Due to the authenticity of the communication between $\mathsf{Prog}_{\mathsf{Dec}}$ and $\mathcal{O}_{\mathsf{msk}}$, the only possibility for $\mathcal{A}$ to make $\mathsf{Prog}_{\mathsf{Dec}}$ output a $P(\mathsf{msg})$ different from $P(\mathsf{msg}^*)$ is by finding an $H(\mathsf{ct}_m)$ corresponding to some message $\mathsf{msg} \neq \mathsf{msg}^*$ s.t. such that $H(\mathsf{ct}_m) = H(\mathsf{ct}^*_m)$. But if $\mathcal{A}$ can do this, we can construct an $\mathcal{A}^*$ which uses $\mathcal{A}$ to break the collision resistance property of $\mathcal{H}$.

Hybrid 5 As in **Hybrid** 4, except that $\mathcal{O}_{\mathsf{msk}}$ oracle encrypts $[0^{|\mathsf{ek}|}, P(\mathsf{msg}^*), 1]$ to generate $\mathsf{ct}^1_{\mathrm{tmp}}$ instead of $[\mathsf{ek}, 0^{\ell_{\mathsf{out}}}, 0]$ if the ct_k in ϕ is $\mathsf{ct}^*_k \in \mathcal{K}$.

The indistinguishability between these two hybrids can be proven similar to the proof of indistinguishability between **Hybrid** 2 and **Hybrid** 3. During the pre-processing stage, $\mathcal{S}$ sets $\mathsf{pk}^1_{\mathrm{tmp}} = \mathsf{pk}^*$ and then later during the simulation of $\mathcal{O}^*_{\mathsf{msk}}$ $\mathcal{S}$ sets $\mathsf{ct}^1_{\mathrm{tmp}} = \mathsf{ct}^*$.

Hybrid 6 As in **Hybrid** 5, except that FE.Enc encrypts $[0^{|\mathsf{ek}|}, H(\mathsf{ct}^*_m)]$ to get ct^*_k instead of $[\mathsf{ek}, H(\mathsf{ct}^*_m)]$ while encrypting the challenge message msg^*. Also, when ct^*_k is queried to the $\mathcal{O}_{\mathsf{msk}}$ oracle, $H(\mathsf{ct}^*_m)$ is obtained from the list $\mathcal{K}$. This hybrid can be seen as using the $\mathcal{O}^*_{\mathsf{msk}}$ oracle.

We will use the IND-CCA2 security of PKE to prove the indistinguishability between these two hybrids. If $\mathcal{A}$ distinguishes between them, we will construct an adversary $\mathcal{A}^*$ for the IND-CCA2 security game. $\mathcal{A}^*$ first gets pk^* from its challenger and $\mathcal{S}$ sets $\mathsf{pk}_{\mathsf{pke}} = \mathsf{pk}^*$ in mpk. $\mathcal{A}^*$ provides $[\mathsf{ek}, H(\mathsf{ct}^*_m)]$ and $[0^{|\mathsf{ek}|}, H(\mathsf{ct}^*_m)]$ to its challenger as its two challenge messages, where ek is the ephemeral key used to encrypt msg^*. And it gets back ct^* which is an encryption of either of these two. $\mathcal{S}$ will set $\mathsf{ct}^*_k = \mathsf{ct}^*$ in FE.Enc. Whenever $\mathcal{A}$ queries the $\mathcal{O}_{\mathsf{msk}}$ oracle with a $\mathsf{ct}_k \notin \mathcal{K}$ (and hence $\mathsf{ct}_k \neq \mathsf{ct}^*$), the decryption oracle provided by the IND-CCA2 challenger is used to get $[\mathsf{ek}, H(\mathsf{ct}_m)]$ for $\mathcal{S}$. Clearly, if ct^* is an encryption of $[\mathsf{ek}, H(\mathsf{ct}^*_m)]$ the view of $\mathcal{A}$ is as in **Hybrid** 5 and if ct^* is an encryption of $[0^{|\mathsf{ek}|}, H(\mathsf{ct}^*_m)]$ the view of $\mathcal{A}$ is an in **Hybrid** 6. Hence, if $\mathcal{A}$ distinguishes between these two hybrids, $\mathcal{A}^*$ breaks the IND-CCA2 security of PKE.

Hybrid 7 As in **Hybrid** 6, except that FE.Enc encrypts $0^{|\mathsf{msg}^*|}$ to get ct^*_m instead of msg^* while encrypting the challenge message msg^*. This hybrid can be seen as using the $\mathsf{FE.Enc}^*$ algorithm to encrypt msg^*.

We will use the IND-CPA security of E here. If $\mathcal{A}$ distinguishes between these two hybrids, we will construct an adversary $\mathcal{A}^*$ for the IND-CPA security game. $\mathcal{A}^*$ provides msg^* and $0^{|\mathsf{msg}^*|}$ to its challenger as its two challenge messages and gets back ct^* which is an encryption of either of these two. $\mathcal{S}$ sets $\mathsf{ct}_m = \mathsf{ct}^*$ in the challenge ciphertext. The other components in the FE security game do not depend on the key used in encrypting ct^* anymore and hence $\mathcal{S}$ can simulate them. Thus, if ct^* is an encryption of msg^*, the view of $\mathcal{A}$ is an in **Hybrid** 6 and if ct^* is an encryption of $0^{|\mathsf{msg}^*|}$, the view of $\mathcal{A}$ is as in **Hybrid** 7. Hence, if $\mathcal{A}$ distinguishes between these two hybrids, $\mathcal{A}^*$ breaks the IND-CPA security of E.

4 Implementation and Evaluation

As we mentioned, the motivation for this hardware assisted model and our work is the recent progress in the Intel processor support for SGX which enables the use of "protected" areas of execution. We perform all our experiments on a Dell Inspiron 13 laptop with an Intel i7 processor and 8 GB RAM. The laptop has Intel Software Guard Extensions (SGX) instruction set. A total of 128 MB is allotted by Intel for creating and running secure containers a.k.a. encrypted enclaves in the main memory. All our experiments are run in the debug mode, hence no remote attestation and verification are performed. Production mode, where which remote attestation is possible, requires commercial licenses [Int] which we did not purchase.

We implemented all FE algorithms. Decryption algorithm was loaded inside the encrypted enclave. In SGX terminology, HW.Load involves running the instructions ECREATE, EADD, EEXTEND and EINIT [MAB+13]. HW.Run&Attest involves first executing the instructions EENTER/ERESUME and EEXIT/AEX and then executing EREPORT to create the attestation. And, HW.Verify is done by contacting the Intel Attestation service [JSR+16]. In the actual SGX implementation $\mathsf{sk}_{\mathsf{HW}}$ does not contain the "signing key". The hardware talks to an external key repository proving its knowledge of $\mathsf{sk}_{\mathsf{HW}}$ to get the signing key.

We run our experiments on 27 MB data files. We designed FE to support issuing of secret keys for basic arithmetic operations (addition, multiplication, division, modulo), mean and simple linear regression programs. We implement a simplified form of our FE scheme in which we define these functions in the decryption program itself, rather than

taking the function (and the corresponding secret key) as input to the main decryption algorithm and then executing it. We run our experiment over three different randomly generated data sets: one structured (in linear regression terms), one with values between 0 and 100 and one with randomly generated unsigned int values. But our implementation runs the same way for any dataset (with individual data points represented as floats).

We use AES-GCM-128 for our secret key encryption. The crypto library provided for SGX enclaves[12] does not include a public key encryption or a key encapsulation mechanism scheme. However, the library does have an implementation for Elliptic curve parameters generation algorithms to be used for key exchange and ECDSA signatures. We build public key encryption on top of these basic operations.

The FE.Setup and FE.Keygen algorithms take about 1 millisecond. We used the EVP interface to OpenSSL to AES-GCM-128 encrypt our dataset outside the enclave. This encryption of a 27 MB data takes around 22 ms. Now, we compare the running time of our FE decryption algorithm with that of a simple implementation of equivalent programs over plaintext data. We first do for simple arithmetic operations: $a_i \odot b_i$ where $\odot = \{+, *, /\}$ over (about 4 million) pairs of numbers. We then do simple linear regression, which involves finding the "best-fit" (α, β) such that $b_i = \alpha + \beta a_i$, over datasets of the same size. We also do the operations, modulus ($a_i \bmod p$ using a fixed p) and mean, over the entire dataset. We exclude the file I/O time in our analysis. We also exclude network delay and the time taken for the one time enclave creation process (around 550 ms) which become relevant only when using FE. Our results are presented in Table 1. This shows that the overhead involved with our scheme is very low. Actually, most of this overhead is in the "context-switch" during the enclave function call. This process took 70 ms for mean and linear regression functions and 50 ms for the others[13].

	Plaintext computation	FE Decryption
Add	15	99
Multiply	15	99
Divide	15	98
Modulus	30	100
Mean	30	133
Lin Reg	52	140

Table 1: Running time in milliseconds first for computations over plaintext data, and then for FE decryptions over FE encrypted data.

Scaling For larger datasets, the program running inside the enclave has to process the data in chunks. Hence, for memory intensive programs (eg. data mining algorithms), Intel's current 128 MB restriction in Windows will have an impact on the performance. The Linux SGX SDK does not have this 128MB restriction. So the program need not have to process data in chunks. The catch is that the performance degrades with the amount of memory allotted for an enclave. It is an interesting direction for future research to explore the performance impact for memory intensive programs.

5 Summary and Future work

In this work, we presented a construction of functional encryption in an oracle assisted model of computation, where the decryption nodes are also equipped with a secure hardware component. We proved the security of our construction under the simulation-based security notion. Our experimental results show that our construction is very practical for the real-world applications.

[12] Only a limited set of libraries are available inside SGX. And, that does not include extensive crypto libraries like OpenSSL.

[13] We measured it by calling a function which has the same arguments but does nothing.

A very important future work is to develop efficient oblivious algorithms resistant to side-channel attacks. It will also be interesting to design oblivious programs that conform to the 4KB page granularity in memory access pattern leaks. On the other hand, from a theoretical perspective, it will also be interesting to develop a security model to formally quantify the access pattern and other side-channel leaks by making the FE simulator also take the side-channel information as input.

References

[AAP15] Shashank Agrawal, Shweta Agrawal, and Manoj Prabhakaran. Cryptographic agents: Towards a unified theory of computing on encrypted data. In *EUROCRYPT II*, pages 501–531, 2015.

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

[AGJS13] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *HASP*, page 13, 2013.

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

[AHKM14] Daniel Apon, Yan Huang, Jonathan Katz, and Alex J. Malozemoff. Implementing cryptographic program obfuscation. Cryptology ePrint Archive, Report 2014/779, 2014. `http://eprint.iacr.org/`.

[APG+11] Joseph A. Akinyele, Matthew W. Pagano, Matthew D. Green, Christoph U. Lehmann, Zachary N.J. Peterson, and Aviel D. Rubin. Securing electronic medical records using attribute-based encryption on mobile devices. In *SPSM*, pages 75–86, 2011.

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

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

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

[BW06] Xavier Boyen and Brent Waters. Anonymous hierarchical identity-based encryption (without random oracles). In *CRYPTO*, pages 290–307, 2006.

[CD16] Victor Costan and Srinivas Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016. `http://eprint.iacr.org/`.

[CHL+15] Jung Hee Cheon, Kyoohyung Han, Changmin Lee, Hansol Ryu, and Damien Stehlé. Cryptanalysis of the multilinear map over the integers. In *EUROCRYPT I*, pages 3–12, 2015.

[CIJ+13] Angelo De Caro, Vincenzo Iovino, Abhishek Jain, Adam O'Neill, Omer Paneth, and Giuseppe Persiano. On the achievability of simulation-based security for functional encryption. In *CRYPTO II*, pages 519–535, 2013.

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

[CLLT15] Jean-Sebastien Coron, Moon Sung Lee, Tancrede Lepoint, and Mehdi Tibouchi. Cryptanalysis of ggh15 multilinear maps. Cryptology ePrint Archive, Report 2015/1037, 2015. `http://eprint.iacr.org/`.

[CLT13] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Practical multilinear maps over the integers. In *CRYPTO (1)*, pages 476–493, 2013.

[Coc01] Clifford Cocks. An identity based encryption scheme based on quadratic residues. In *IMA Int. Conf.*, pages 360–363, 2001.

[GGH13a] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In *EUROCRYPT*, pages 1–17, 2013.

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

[GGH15] Craig Gentry, Sergey Gorbunov, and Shai Halevi. Graph-induced multilinear maps from lattices. In *TCC*, 2015.

[GMN+16] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *CCS*, pages 1353–1364, 2016.

[GPSW06] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *ACM CCS*, pages 89–98, 2006.

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

[GVW13] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Attribute-based encryption for circuits. In *STOC*, pages 545–554, 2013.

[GVW15] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Predicate encryption for circuits from LWE. In *CRYPTO II*, pages 503–523, 2015.

[HJ16] Yupu Hu and Huiwen Jia. Cryptanalysis of GGH map. In *EUROCRYPT I*, pages 537–565, 2016.

[Int] Intel SGX product licensing. `https://software.intel.com/en-us/articles/intel-sgx-product-licensing`. Accessed: 2016-05-20.

[JSR+16] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel software guard extensions: EPID provisioning and attestation services. 2016.

[KPW16] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. Whitepaper, 2016.

[KSW08] Jonathan Katz, Amit Sahai, and Brent Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *EUROCRYPT*, pages 146–162, 2008.

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

[LOS+10] Allison B. Lewko, Tatsuaki Okamoto, Amit Sahai, Katsuyuki Takashima, and Brent Waters. Fully secure functional encryption: Attribute-based encryption and (hierarchical) inner product encryption. In *EUROCRYPT*, pages 62–91, 2010.

[LYZ+13] Ming Li, Shucheng Yu, Yao Zheng, Kui Ren, and Wenjing Lou. Scalable and secure sharing of personal health records in cloud computing using attribute-based encryption. *IEEE Transactions on Parallel and Distributed Systems*, 24(1):131–143, 2013.

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

[MLS+13] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: practical oblivious computation in a secure processor. In *CCS*, pages 311–324, 2013.

[MSZ16] Eric Miles, Amit Sahai, and Mark Zhandry. Annihilation attacks for multilinear maps: Cryptanalysis of indistinguishability obfuscation over GGH13. In *CRYPTO*, 2016.

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

[NKW15] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference attacks on property-preserving encrypted databases. In *CCS*, pages 644–655, 2015.

[NY90] Moni Naor and Moti Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *STOC*, pages 427–437, 1990.

[PPM+14] John P. Papanis, Stavros I. Papapanagiotou, Aziz S. Mousas, Georgios V. Lioudakis, Dimitra I. Kaklamani, and Iakovos S. Venieris. On the use of attribute-based encryption for multimedia content protection over information-centric networks. *Transactions on Emerging Telecommunications Technologies*, 25(4):422–435, 2014.

[PRZB11] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.

[PSV+14] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. Building web applications on top of encrypted data using mylar. In *NSDI*, pages 157–172, 2014.

[Sha84] Adi Shamir. Identity-based cryptosystems and signature schemes. In *CRYPTO*, pages 47–53, 1984.

[SW05] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In *EUROCRYPT*, pages 457–473, 2005.

[SZEA+16] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Rachel Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage, 2016.

[WNL+14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *CCS*, pages 215–226, 2014.

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