Making Public Key Functional Encryption Function Private, Distributively

Xiong Fan[1] Qiang Tang[2]

[1] Cornell University, Ithaca, NY, USA. xfan@cs.cornell.edu
[2] New Jersey Institute of Technology, Newark, NJ, USA. qiang@njit.edu

Abstract. We put forth a new notion of distributed public key functional encryption. In such a functional encryption scheme, the secret key for a function f will be split into shares sk_i^f. Given a ciphertext ct that encrypts a message x, a secret key share sk_i^f, one can evaluate and obtain a shared value y_i. Adding all the shares up can recover the actual value of $f(x)$, while partial shares reveal nothing about the plaintext. More importantly, this new model allows us to establish *function privacy* which was not possible in the setting of regular public key functional encryption. We formalize such notion and construct such a scheme from any public key functional encryption scheme together with learning with error assumption.

We then consider the problem of hosting services in the untrusted cloud. Boneh, Gupta, Mironov, and Sahai (Eurocrypt 2014) first studied such application and gave a construction based on indistinguishability obfuscation. Their construction had the restriction that the number of corrupted clients has to be bounded and known. They left an open problem how to remove such restriction. We resolve this problem by applying our function private (distributed) public key functional encryption to the setting of hosting service in multiple clouds. Furthermore, our construction provides a much simpler and more flexible paradigm which is of both conceptual and practical interests.

Along the way, we strengthen and simplify the security notions of the underlying primitives, including function secret sharing.

1 Introduction

Cloud computing has the advantages that the cloud servers provide infrastructure and resources that can hold data, do computation for the clients, and even host service on behalf of the individual vendor (also called service providers). Despite those appealing features, major concerns of deploying such a computing paradigm are the security and privacy considerations, as data owner does not have control of the outsourced data.

Functional encryption [BSW11,O'N10] provides a powerful tool to enable such versatile "outsourcing" without leaking the actual data. In particular, a data owner can first encrypt his data x and store the ciphertext ct on the cloud server, and then issue a secret key sk_f to the cloud for a functionality f

that the data owner would like the cloud to compute. Decrypting ct using sk_f yields only $f(x)$ and nothing else. For instance, the client would like to request the cloud to apply a transformation T to all his files that satisfies a certain condition described by a predicate P. This can be easily done by defining a following function $g(\cdot)$, where $g(x) = T(x)$, if $P(x) = 1$; otherwise, $g(x) = x$; the data owner can simply send such a decryption key sk_g to the cloud and enable the cloud to carry on the transformation given only the encrypted files. Those mechanisms could potentially enable a very powerful paradigm shift in computing. For example, content providers can simply focus on producing the data while offloading all the content management and delivery functionalities to the cloud provider. Concretely, Netflix streaming services have been migrated to Amazon cloud [net]. In particular, Netflix could codify their algorithm (such as the recommendation system) f to be sk_f and let Amazon cloud process all the subscriber requests expressed as ciphertext.

In many cases, hiding data only is not enough for those applications, as the function itself may already leak critical or proprietary information. In the above example, other content providers such as Hulu also hosts their service in the Amazon cloud [hul], the recommendation system might be one of the competing advantages of those content vendors. If not protected properly, one service provider has great interests to infer information about the competing vendor's proprietary program via the cloud. For this reason, *function privacy* was first proposed by Shen, Shi, Waters in the setting of private key predicate encryption in [SSW09]. It requires that a decryption key sk_f does not leak anything about the function f.

It is easy to see that for a public key functional encryption, standard function privacy cannot be possible as it is. Since the attacker who has a key sk_f, can generate ciphertext on the fly, and thus obtain values of $f(x_1), \ldots, f(x_n)$ for the plaintext $x_1, \ldots, x_n$ of his choices. As a result, majority of research along this line have been carried in two paths: (i) study function privacy in the setting of private key functional encryption such as the elegant work of Brakerski and Segev [BS15]. (ii) study weakened notion of function privacy by requiring that the function comes from high-entropy distribution [AAB+13], and those are studied only in special cases of identity based encryption [BRS13a] and subspace-membership encryption [BRS13b]. [3]

Private key functional encryption is very useful for data owner to do outsourcing, however it is not convenient for sharing applications in which multiple clients may want to freely encode inputs, i.e., a public encryption operation is needed. While putting entropy restriction on the functions is a natural choice for feasibility of function privacy in specific scenarios, it is not clear how could the weakened notion be applied in the general setting. In this paper, we are trying to answer the following question:

Can we find a realistic model that allows us to approach function privacy for general public key functional encryption?

[3] Except the nice work of Agrawal et al.[AAB+13] which considered both above cases.

1.1 Our contributions

Circumventing impossibility via a distributed model. We initiate the study of public key, distributed functional encryption. In such a cryptographic primitive, the secret key of a function f will be split into shares sk_i^f and distributed to different parties. Given a secret key share, and a ciphertext ct that encrypts a message x, one can evaluate locally using sk_i^f and obtains $\mathsf{Dec}(\mathsf{sk}_i^f, \mathsf{Enc}(x)) = y_i$. Once all the evaluation shares $\{y_1, \ldots, y_n\}$ are obtained, everyone can reconstruct the actual evaluation $f(x)$. This new model of distributed functional encryption naturally generalizes the notion of threshold decryption to the setting of functional encryption, and enables the joint efforts to recover an evaluation for a plaintext from a ciphertext (i.e., computing $f(x)$ from the ciphertext ct), and when the number function shares are not enough, nothing will be revealed about $f(x)$.

More interestingly, such a new model offers an opportunity to bypass the impossibility of function privacy in the setting of public key functional encryption. Intuitively, given only a share sk_i^f (or multiple shares as long as it is below the threshold), the adversary can only learn y_i which may not be enough to determine $f(x)$. Formalizing such intuition, we give formal definitions of public key distributed functional encryption, and transform any public key functional encryption into a distributed version supporting both message privacy and function privacy via function secret sharing [BGI15,BGI16,KZ16]. Our construction can be instantiated from any functional encryption together with Learning With Error assumption [Reg05] where the construction of function secret sharing is based on, and reconstruction from shares $\{y_1, \ldots, y_n\}$ can be done by simply summing them up.

We remark here that our notion of distributed functional encryption is different from the decentralized key generation of functional encryption [CGJS15]. The latter mainly considers how to distribute the master key setup; while we consider how to split each function into secret key shares, and use such a model as a basis for studying function privacy. We also emphasize that the goal in this work is to achieve results generically from functional encryption itself directly, instead of from stronger primitives such as indistinguishability obfuscation (iO). With the help of iO or its stronger variant, differing-inputs obfuscation, we know how to construction multi-input functional encryption [GGG+14] and also function secret sharing [KZ16], there might be alternative ways to construct distributed public key functional encryption, which we will not explore in this paper.

Hosting service in multiple clouds securely and efficiently. One of the most appealing and widely deployed applications of cloud computing is to hosting service in the cloud. Boneh, Gupta, Mironov, and Sahai gave the first formal study of such an application [BGMS15]. The security considerations in this application scenario include protecting program (service) information and clients' inputs against a untrusted cloud and protecting program (service) information and authorization procedure against untrusted clients. Their construction relied on indistinguishability obfuscation (iO), and had to restrict the number of colluded/corrupted clients for both security. They left as an open problem how

to get rid of such a restriction. As one major application of our function private functional encryption, we demonstrate how to tackle this challenge in the model of hosting service in multiple clouds.

Let us elaborate via a concrete example: the popular augmented-reality game Pokémon Go server was hosted at Google Cloud [pok]. The whole game as a computer program is deployed in Google cloud servers, and players directly interact with Google cloud to play the game once they are registered. The players try to catch various level Poke Monsters depending on the locations. Thus the level and location of the monsters contained in the game program need to be hidden. At the same time, the business model for such a game is to sell virtual goods, thus the program that hosts the service in the cloud will have to authenticate those in-game equipments. If such function is not protected well, when the cloud is corrupted, such authentication could either be bypassed or even completely reaped. On the other hand, there were also huge number of security concerns about the server collecting user private information when playing the game.

The above example highlights the need of *securely* hosting service in the cloud, and the service may be provided to millions of clients. One simple observation we would like to highlight in the paper is that our public key functional encryption with function privacy is already very close to the powerful notion of virtual black-box obfuscation (VBB) [BGI+01]. Taking a "detour" from using iO as in [BGMS15] to using VBB, and then "instantiating" it using our functional encryption yields a new way of securely hosting service in multiple clouds, and enables us to achieve much stronger security notions that have no restriction on the number of corrupted clients. From a high level, to host a service described as a function f in the cloud, the service provider runs our distributed functional encryption key generation algorithm and generates shares sk_i^f for each cloud.

It is not hard to see from the above description, as our construction following such a paradigm, we can easily extend the functionalities by encode the original functionality f into other program g to support more advanced properties and more complex access control.

Moreover, as our distributed functional encryption only relies on a regular functional encryption instead of a general iO, this new paradigm may potentially lead efficient constructions that can be actually instantiated. For example, if a service provider only hosts a couple of functionalities in the cloud, we do not have to use the full power of general functional encryption, instead we can use the bounded collusion functional encryption [GVW12] which could be further optimized for particular functions.

Last, as our reconstruction procedure only requires an addition, it gives minimum overhead to the client.

Strengthened and simplified security models, and modular constructions. We note that since the application of hosting service in the cloud is complex, several underlying building blocks such as function secret sharing as given are not enough for our applications. We carefully decoupled the complex security notions of [BGMS15] which handles two properties for each notion. This simplification helps us identify necessary enhancements of the security notions

of the underlying building blocks, which in turn, enables us to have a smooth modular construction for the complex object.

Consider the security of the program against untrusted clouds when the service is hosted in two clouds. A corrupted cloud has one share of the program, on the mean time, the cloud may pretend to be a client and send requests to the other cloud for service. This means that considering function privacy against adversaries that has only partial shares is not enough. We should further allow the adversary to query the rest of function shares to reconstruct values for a bunch of points. The desired security notion now is that the adversary should learn nothing more than the values she already obtained as above. For this reason, we propose a CCA-type of definition for function privacy. To tackle this, we revisited the security of function secret sharing and study a CCA-type of security notion for it (the existing work only considered the CPA version).

Consider the security of the program against untrusted clients. Now a legitimate client can send requests and get evaluated at arbitrary points. To ensure the security of the program which comes from the function privacy in our construction, it naturally requires a simulation style definition. While IND style of function privacy was considered in most of previous works, even for private key functional encryption [BS15], we propose to study a simulation based definition with the CCA-type of enhancement mentioned above.

We show that the simple construction of function secret sharing from Spooky Encryption [DHRW16] actually satisfies the stronger notions, and we can safely apply it to construct our distributed functional encryption and eventually lead to the secure service hosting in multiple clouds.

1.2 Related work

As mentioned above, despite the great potential of function privacy, our understanding of it is limited. Shen, Shi and Waters [SSW09] initiated the research on predicate privacy of attribute-based encryption in private key setting. Boneh, Raghunathan and Segev [BRS13a,BRS13b] initiated function privacy research in public key setting. They constructed function-private public-key functional encryption schemes for point functions (identity-based encryption) and for subspace membership (generalization of inner-product encryption). However, their framework assumes that the functions come from a distribution of sufficient entropy.

In an elegant work [AAB+15], Agrawal et al. presented a general framework of security that captures both data and function hiding, both public key and symmetric key settings, and show that it can be achieved in the generic group model for Inner Product FE [KSW08]. Later, in the private-key setting, Brakerski and Segev [BS15] present a generic transformation that yields a function-private functional encryption scheme, starting with any non-function-private scheme for a sufficiently rich function class.

In [BGMS15], Boneh et al. provide the first formalizations of security for a secure cloud service scheme. They also provide constructions of secure cloud

service schemes assuming indistinguishability obfuscation, one-way functions, and non-interactive zero-knowledge proofs.

2 Preliminaries

Notation. Let λ be the security parameter, and let PPT denote probabilistic polynomial time. We say a function $\mathsf{negl}(\cdot) : \mathbb{N} \to (0,1)$ is negligible, if for every constant $c \in \mathbb{N}$, $\mathsf{negl}(n) < n^{-c}$ for sufficiently large n. We say two distributions D_1, D_2 over a finite universe $\mathcal{U}$ are ϵ-close if their statistical distance $\frac{1}{2}||D_1 - D_2||_1$ are at most ϵ, and denoted as $D_1 \approx D_2$.

2.1 Signature Scheme

In this part, we recall the syntax and security definition of a signature scheme. A signature scheme $\Sigma = (\mathsf{Setup}, \mathsf{Sign}, \mathsf{Verify})$ can be described as

- $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Setup}(1^\lambda)$: On input security parameter λ, the setup algorithm outputs signing key sk and verification key vk.
- $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$: On input signing key sk and message m, the signing algorithm outputs signature σ for message m.
- 1 or $0 \leftarrow \mathsf{Verify}(\mathsf{vk}, m, \sigma)$: On input verification key vk, message m and signature σ, the verification algorithm outputs 1 if the signature is valid. Otherwise, output 0.

For the security definition of signature scheme, we use the following experiment to describe it. Formally, for any PPT adversary $\mathcal{A}$, we consider the experiment $\mathbf{Expt}^{\mathsf{sig}}_{\mathcal{A}}(1^\lambda)$:

1. Challenger runs $\mathsf{Setup}(1^\lambda)$ to obtain $(\mathsf{vk}, \mathsf{sk})$ and sends vk to adversary $\mathcal{A}$.
2. Adversary $\mathcal{A}$ sends signing queries $\{m_i\}_{i\in[Q]}$ to challenger. For $i \in [Q]$, challenger computes $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{sk}, m_i)$ and sends $\{\sigma_i\}_{i\in[Q]}$ to adversary $\mathcal{A}$.
3. Adversary $\mathcal{A}$ outputs a forgery pair (m^*, σ^*).

We say adversary $\mathcal{A}$ wins experiment $\mathbf{Expt}^{\mathsf{sig}}_{\mathcal{A}}(1^\lambda)$ if m^* is not queried before and $\mathsf{Verify}(\mathsf{vk}, m^*, \sigma^*) = 1$.

Definition 1 (Existential Unforgeability). *We say a signature scheme Σ is existentially unforgeable if no* PPT *adversary $\mathcal{A}$ can win the experiment $\mathbf{Expt}^{\mathsf{sig}}_{\mathcal{A}}(1^\lambda)$ with non-negligible probability.*

2.2 Functional Encryption

We recall the syntax and ind-based security of functional encryption introduced in [BSW11]. A functional encryption scheme FE for function ensemble $\mathcal{F}$ consists of four algorithms defined as follows:

- $(\mathsf{pp}, \mathsf{msk}) \leftarrow \mathsf{Setup}(1^\lambda)$: On input the security parameter λ, the setup algorithm outputs public parameters pp and master secret key msk.

– $\mathsf{sk}_f \leftarrow \mathsf{Keygen}(\mathsf{msk}, f)$: On input the master secret key msk and a function f, the key generation algorithm outputs a secret key sk_f for function f.
– $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pp}, \mu)$: On input the public parameters pp and a message μ, the encryption algorithm outputs a ciphertext ct.
– $f(\mu) \leftarrow \mathsf{Dec}(\mathsf{sk}_f, \mathsf{ct})$: On input a secret key sk_f for function f and a ciphertext ct for plaintext μ, the decryption algorithm outputs $f(\mu)$.

Definition 2 (Correctness). *A functional encryption scheme* FE *is correct if for any* $(\mathsf{pp}, \mathsf{msk}) \leftarrow \mathsf{Setup}(1^\lambda)$*, any* $f \in \mathcal{F}$*, and* $\mu \in \mathsf{domain}(f)$*, it holds that*

$$\Pr[\mathsf{Dec}(\mathsf{Keygen}(\mathsf{msk}, f), \mathsf{Enc}(\mathsf{pp}, \mu)) \neq f(\mu)] = \mathsf{negl}(\lambda)$$

where the probability is taken over the coins in algorithms Keygen *and* Enc*.*

Security Definition. We present the security of functional encryption scheme FE for function ensemble $\mathcal{F}$ by first describing an experiment $\mathbf{Expt}^{\mathsf{FE}}_{\mathcal{A}}(1^\lambda)$ between an adversary $\mathcal{A}$ and a challenger in the following:

Setup: The challenger runs $(\mathsf{msk}, \mathsf{pp}) \leftarrow \mathsf{Setup}(1^\lambda)$ and sends pp to adversary $\mathcal{A}$.
Key query phase I: Proceeding adaptively, the adversary $\mathcal{A}$ submits function $f_i \in \mathcal{F}$ to challenger. The challenger then sends back $\mathsf{sk}_f \leftarrow \mathsf{Keygen}(\mathsf{msk}, f_i)$ to adversary $\mathcal{A}$.
Challenge phase: The adversary submits the challenge pair (μ_0^*, μ_1^*), with the restriction that $f_i(\mu_0^*) = f_i(\mu_1^*)$ for all functions f_i queried before. The challenger first chooses a random bit $b \in \{0, 1\}$ and sends back $\mathsf{ct}^* \leftarrow \mathsf{Enc}(\mathsf{pp}, \mu_b)$ to adversary $\mathcal{A}$.
Key query phase II: The adversary $\mathcal{A}$ may continue his function queries $f_i \in \mathcal{F}$ adaptively with the restriction that $f_i(\mu_0^*) = f_i(\mu_1^*)$ for all function queries f_i.
Guess: Finally, the adversary $\mathcal{A}$ outputs his guess b' for the bit b.

We say the adversary wins the experiment if $b' = b$.

Definition 3 (Ind-based Data Privacy). *A functional encryption scheme* $\mathsf{FE} = (\mathsf{Setup}, \mathsf{Keygen}, \mathsf{Enc}, \mathsf{Dec})$ *for a family of function* $\mathcal{F}$ *is secure if no* PPT *adversary* $\mathcal{A}$ *can win the experiment* $\mathbf{Expt}^{\mathsf{FE}}_{\mathcal{A}}(1^\lambda)$ *with non-negligible probability.*

2.3 Spooky Encryption

We recall the definition of spooky encryption, introduced in [DHRW16] in this part. A public key encryption scheme consists a tuple $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ of polynomial-time algorithms. The key-generation algorithm Gen gets as input a security parameter λ and outputs a pair of public/secret keys $(\mathsf{pk}, \mathsf{sk})$. The encryption algorithm Enc takes as input the public key pk and a bit m and output a ciphertext ct, whereas the decryption algorithm Dec gets as input the secret key sk and ciphertext ct, and outputs the plaintext m. The basic correctness guarantee is that $\Pr[\mathsf{Dec}_{\mathsf{sk}}(\mathsf{Enc}(\mathsf{pk}, m)) = m] \geq 1 - \mathsf{negl}(\lambda)$, where the probability is over the

randomness of all these algorithms. The security requirement is that for any PPT adversary $(\mathcal{A}_1, \mathcal{A}_2)$ it holds that

$$\mathsf{Pr}_{b\leftarrow\{0,1\}}[(m_0, m_1) \leftarrow \mathcal{A}_1(\mathsf{pk}), \mathcal{A}_2(\mathsf{pk}, \mathsf{ct}_b) = 1] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$$

where $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda), \mathsf{ct}_b \leftarrow \mathsf{Enc}(\mathsf{pk}, m_b)$ and require $|m_0| = |m_1|$.

Definition 4 (Spooky Encryption). *Let* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a public key encryption and* Eval *be a polynomial-time algorithm that takes as input a (possibly randomized) circuit* C *with* $n = n(\lambda)$ *inputs and* n *outputs,* $C : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$*, and also* n *pairs of (public key, ciphertext), and outputs* n *ciphertext.*

Let $\mathcal{C}$ *be a class of such circuits, we say that* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ *is a* $\mathcal{C}$*-spooky encryption scheme if for any security parameter* λ*, any randomized circuit* $C \in \mathcal{C}$*, and any input* $\boldsymbol{x} = (x_1, ..., x_n)$ *for* C*, the following distributions are close upto a negligible distance in* λ

$$C(x_1, ..., x_n) \approx \mathsf{SPOOK}[C, \boldsymbol{x}] \stackrel{\Delta}{=}$$
$$\{(\mathsf{Dec}(\mathsf{sk}_1, \mathsf{ct}'_1), ..., \mathsf{Dec}(\mathsf{sk}_n, \mathsf{ct}'_n)) : (\mathsf{ct}'_1, ..., \mathsf{ct}'_n) \leftarrow \mathsf{Eval}(C, \{(\mathsf{pk}_i, \mathsf{ct}_i)\}_i)\}$$

where for $i \in [n]$*,* $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{Gen}(1^\lambda), \mathsf{ct}_i \leftarrow \mathsf{Enc}(\mathsf{pk}_i, m_i)$*.*

A special case of spooky encryption, named additive-function-sharing (AFS) spooky encryption, allows us to take encryptions $\mathsf{ct}_i \leftarrow \mathsf{Enc}(\mathsf{pk}_i, x_i)$ under n independent keys of inputs $x_1, ..., x_n$ to an n-argument function f, and produce new ciphertext under the same n keys that decrypts to additive secret shares of $y = f(x_1, ..., x_n)$. Formally, the definition is the following

Definition 5 (AFS-Spooky). *Let* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ *be a scheme where* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is a semantically secure public key encryption. We say* Π *is leveled* ϵ*-AFS-spooky if* Π *satisfies*

- *If for any boolean circuit* C *computing an* n*-argument function* $f : (\{0,1\}^*)^n \to \{0,1\}$*, and any input* $(x_1, ..., x_n)$ *for* C*, it holds that*

$$\mathsf{Pr}[\sum_{i=1}^{n} y_i = C(x_1, ..., x_n) : (\mathsf{ct}'_1, ..., \mathsf{ct}'_n) \leftarrow \mathsf{Eval}(C, \{(\mathsf{pk}_i, \mathsf{ct}_i)\}_i)]$$

  where for $i \in [n]$*,* $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{Gen}(1^\lambda), \mathsf{ct}_i \leftarrow \mathsf{Enc}(\mathsf{pk}_i, x_i), y_i = \mathsf{Dec}(\mathsf{sk}_i, \mathsf{ct}'_i)$*.*
- *Any* $n - 1$ *of the shares* y_i *above are distributed* ϵ*-close to uniform.*
- *We say* Π *is leveled if the* Gen *algorithm receives an additional depth parameter* 1^d*, and the conditions above hold only for circuit of depth upto* d*.*

Spooky Encryption with CRS. We say that $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Spooky.Eval})$ is a $\mathcal{C}$*-spooky encryption scheme with CRS*, if Definition 4 and 5 are satisfied if we allow all algorithms (and the adversary) to get as input also a public uniformly distributed common random string.

In [DHRW16], the authors showed how to construction ϵ-AFS-Spooky Encryption with CRS from Learning With Error assumption (LWE) [Reg05]. Their results can be summarized below:

Theorem 1 ([DHRW16]). *Assuming the hardness of LWE assumption, there exists a leveled ϵ-AFS-spooky encryption scheme.*

3 Distributed Public Key FE with Function Privacy

In this section, we give a detailed study of distributed functional encryption (DFE), and specifically a simplified DFE notion, n-out-of-n threshold functional encryption. In an (n, n)-DFE scheme, during key generation, we split a secret key corresponding to the function into n secret key shares $\{\mathsf{sk}_i^f\}_{i=1}^n$, and by running partial decryption on sk_i^f and a ciphertext ct, we can obtain a share s_i of $f(x)$, where ct is an encryption of message x. There is also a reconstruction process that outputs $f(x)$ on n shares $\{s_i\}_{i=1}^n$. We then define security, including *function privacy* and *data privacy*, with respect to (n, n)-DFE.

To achieve a secure DFE satisfying our security definitions, we rely on a building block, named function secret sharing [BGI15,BGI16]. We strengthen the security definition of FSS in comparison with that in [BGI15,BGI16], and show that a construction[4] based on spooky encryption satisfies our generalized security definition.

3.1 Syntax and Security Definition

We first describe the syntax $\mathsf{DFE} = (\mathsf{DFE.Setup}, \mathsf{DFE.Keygen}, \mathsf{DFE.Enc}, \mathsf{DFE.PartDec}, \mathsf{DFE.Reconstruct})$:

- $\mathsf{DFE.Setup}(1^\lambda, n, \mathcal{F})$: On input security parameter λ, threshold parameter n and function ensemble $\mathcal{F}$, the setup algorithm produces $(\mathsf{pp}, \mathsf{msk})$ for the whole system.
- $\mathsf{DFE.Keygen}(f, \mathsf{msk})$: On input a function $f \in \mathcal{F}$ and the secret key sk_i of this authority, the key generation algorithm outputs n secret key shares $\{\mathsf{sk}_i^f\}_{i\in[n]}$ for the function f.
- $\mathsf{DFE.Enc}(\mathsf{pp}, m)$: On input the public parameters pp and a message m, the encryption algorithm outputs a ciphertext ct.
- $\mathsf{DFE.PartDec}(\mathsf{ct}, \mathsf{sk}_i^f)$: On input a ciphertext ct and a secret key share sk_i^f for function f, the partial decryption algorithm outputs a decryption share s_i.
- $\mathsf{DFE.Reconstruct}(\mathsf{pp}, \{s_i\}_{i=1}^n)$: On input the public parameters pp and decryption shares $\{s_i\}_{i=1}^n$ for the same ciphertext, the reconstruction algorithm outputs $f(m)$.

[4] We remark that the construction was first sketched in [DHRW16]. Here we generalize it and provide a formal security proof for the stronger notions.

Definition 6 (Correctness). *An* (n, n)-DFE *scheme is correct if for any* $(\mathsf{pp}, \mathsf{msk}) \leftarrow \mathsf{DFE.Setup}(1^\lambda, 1^n)$*, any* $f \in \mathcal{F}$*, and any* $m \in \mathsf{domain}(f)$*, it holds*

$$\Pr[\mathsf{DFE.Reconstruct}(\mathsf{pp}, \{\mathsf{DFE.PartDec}(\mathsf{ct}, \mathsf{sk}_i^f)\}_{i=1}^n) \neq f(m)] = \mathsf{negl}(\lambda)$$

where $\mathsf{ct} \leftarrow \mathsf{DFE.Enc}(\mathsf{pp}, m), \mathsf{sk}_i^f \leftarrow \mathsf{DFE.Keygen}(f, \mathsf{msk})$ *and the probability is taken over the coins in algorithms* DFE.Keygen *and* DFE.Enc*.*

Security Definition of DFE. As mentioned before, we consider both the data privacy and function privacy for DFE. For completeness, we give both IND-based and simulation based notions for function privacy. As we know, simulation based data privacy is infeasible [AGVW13], thus we only give a Ind based definition. It would be an interesting open problem to consider an alternative model that simulation based data privacy for functional encryption become feasible, e.g., [GVW12] The detailed definitions are below.

Definition 7 (Ind-based function privacy). *We first describe an experiment* $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{DFE\text{-}func}}(1^\lambda)$ *between an adversary* $\mathcal{A}$ *and a challenger as follows:*

- ***Setup****: The challenger runs* $(\mathsf{msk}, \mathsf{pp}) \leftarrow \mathsf{DFE.Setup}(1^\lambda, 1^n)$ *and sends* pp *to adversary* $\mathcal{A}$*.*
- ***Key query phase I****: Proceeding adaptively, the adversary* $\mathcal{A}$ *submits function* $f_j \in \mathcal{F}$ *to challenger. The challenger then sends back* $\{\mathsf{sk}_i^{f_j}\}_{i=1}^n \leftarrow \mathsf{DFE.Keygen}(\mathsf{msk}, f_j)$ *to adversary* $\mathcal{A}$*.*
- ***Challenge phase****: The adversary submits the challenge function pair* (f_0^*, f_1^*) *that are not queried before. The challenger first chooses a random bit* $b \in \{0, 1\}$ *and computes* $\{\mathsf{sk}_i^{f_b^*}\}_{i\in[n]} \leftarrow \mathsf{DFE.Keygen}(\mathsf{msk}, f_b^*)$*. Then challenge selects random* $n-1$ *keys* $\{\mathsf{sk}_i^{f_b^*}\}_{i\in S}$ *and sends them to adversary* $\mathcal{A}$*.*
- ***Key query phase II****: Proceeding adaptively, the adversary* $\mathcal{A}$ *continues querying function* $f_j \in \mathcal{F}$ *with the restriction that* $f_j \neq f_0^*$ *and* $f_j \neq f_1^*$*. The challenger then sends back* $\{\mathsf{sk}_i^{f_j}\}_{i=1}^n \leftarrow \mathsf{DFE.Keygen}(\mathsf{msk}, f_j)$ *to adversary* $\mathcal{A}$*.*
- ***Guess****: Finally, the adversary* $\mathcal{A}$ *outputs his guess* b' *for the bit* b*.*

We say the adversary wins the experiment if $b' = b$*.*

A distributed functional encryption scheme Π *for a family of function* $\mathcal{F}$ *is function private if no* PPT *adversary* $\mathcal{A}$ *can win the experiment* $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{DFE\text{-}func}}(1^\lambda)$ *with non-negligible probability.*

In the simulation-based definition of function privacy, we additionally allow adversary to query oracle $\mathsf{DFE.Dec}(\mathsf{sk}_n^{f^*}, \cdot)$, where $\mathsf{sk}_n^{f^*}$ is the only secret key share for challenge function f^* that is not given to adversary. We then show that our sim-based function privacy implies ind-based function privacy as defined above. The detail is as follows:

Definition 8 (Sim-based function privacy). *Let* Π *be a distributed functional encryption scheme for a function family* $\mathcal{F}$*. Consider a* PPT *adversary*

$\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *and simulator* $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3)$[5]*. We say the function secret sharing scheme* Π *is simulation-secure if the following two distribution ensembles (over the security parameter* λ*) are computationally indistinguishable:*

Real Distribution*:*

1. $(\mathsf{pp}, \mathsf{msk}) \leftarrow \mathsf{DFE.Setup}(1^\lambda, n)$.
2. $(f^*, \tau) \leftarrow \mathcal{A}_1^{\mathsf{DFE.Keygen}(\mathsf{msk},\cdot)}(\mathsf{pp})$.
3. $\{\mathsf{sk}_i^{f^*}\}_{i=1}^n \leftarrow \mathsf{DFE.Keygen}(\mathsf{msk}, f^*)$.
4. $\alpha \leftarrow \mathcal{A}_2^{\mathsf{DFE.Keygen}(\mathsf{msk},\cdot),\mathsf{DFE.Dec}(\mathsf{sk}_n^{f^*},\cdot)}(\mathsf{pp}, \{\mathsf{sk}_i^{f^*}\}_{i=S}^{n-1}, \tau)$.
5. *Output* $(\mathsf{pp}, f^*, \tau, \alpha)$.

Ideal Distribution*:*

1. $\mathsf{pp} \leftarrow \mathcal{S}_1(1^\lambda, n)$.
2. $(f^*, \tau) \leftarrow \mathcal{A}_1^{\mathcal{S}_2(\cdot)}(\mathsf{pp})$.
3. $\{\mathsf{sk}_{f_i^*}\}_{i=1}^n \leftarrow \mathcal{S}_2(|f^*|)$.
4. $\alpha \leftarrow \mathcal{A}_2^{\mathcal{S}_2(\cdot),\mathcal{S}_3^{f^*}(\cdot)}(\mathsf{pp}, \{\mathsf{sk}_i^{f^*}\}_{i=1}^{n-1}, \tau)$.
5. *Output* $(\mathsf{pp}, f^*, \tau, \alpha)$.

where on query $\mathsf{ct} = \mathsf{Enc}(\mathsf{pp}, x)$ *made by adversary* $\mathcal{A}_2$*, simulator* $\mathcal{S}_3^{f^*}(\cdot)$ *makes a query to the oracle* f^**.*

Remark 1. We note that if a DFE construction satisfies sim-based function privacy, then we can show that it also satisfies ind-based function privacy. The challenger in the ind-based experiment $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{DFE\text{-}func}}(1^\lambda)$ first uses simulation $\mathcal{S}_1(1^\lambda, n)$ to generate pp. For key queries f_i, challenger responses by computing $\{\mathsf{sk}_{f_{ij}}\}_{j=1}^n \leftarrow \mathcal{S}_2(f_i)$. For challenge function (f_0^*, f_1^*), the challenger chooses a random bit b (let $f^* = f_b^*$) and computes $\{\mathsf{sk}_{f_i^*}\}_{i=1}^n \leftarrow \mathcal{S}_2(f^*)$. Then by sim-based function privacy as defined above, the responses for key queries simulated by $\mathcal{S}_2$ are indistinguishable from real execution and the bit b is chosen from random, thus we show that it also satisfies ind-based function privacy.

Next, we adapt the standard ind-based data privacy for a DFE scheme.

Definition 9 (Ind-based data privacy). *We first describe an experiment* $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{DFE\text{-}data}}(1^\lambda)$ *between a challenger and an adversary* $\mathcal{A}$ *as below:*

- ***Setup****: The challenger runs* $(\mathsf{msk}, \mathsf{pp}) \leftarrow \mathsf{DFE.Setup}(1^\lambda, 1^n)$ *and sends* pp *to adversary* $\mathcal{A}$*.*
- ***Key query phase I****: Proceeding adaptively, the adversary* $\mathcal{A}$ *submits function* $f_j \in \mathcal{F}$ *to challenger. The challenger then sends back* $\{\mathsf{sk}_i^{f_j}\}_{i=1}^n \leftarrow \mathsf{DFE.Keygen}(\mathsf{msk}, f_j)$ *to adversary* $\mathcal{A}$*.*
- ***Challenge phase****: Adversary submits the challenge message pair* (m_0^*, m_1^*) *with the restriction that* $f_i(m_0) = f_i(m_1)$ *for all queried* f_i*. The challenger first chooses a random bit* $b \in \{0, 1\}$ *and computes* $\mathsf{ct} \leftarrow \mathsf{DFE.enc}(\mathsf{pp}, m_b)$*. Then send* ct *to adversary.*

[5] Looking ahead, we abuse the notation of S_2 in the ideal distribution, by allowing it taking two kinds of inputs: 1. the description of function f, 2. the size of function f.

– ***Key query phase II***: *The same as* ***Key query phase I*** *with the restriction that the query f_i satisfies $f_i(m_0) = f_i(m_1)$.*
– ***Guess***: *Finally, the adversary $\mathcal{A}$ outputs his guess b' for the bit b.*

We say the adversary wins the experiment if $b' = b$.

A distributed functional encryption scheme Π for a family of function $\mathcal{F}$ is data private if no PPT *adversary $\mathcal{A}$ can win the experiment $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{DFE\text{-}data}}(1^\lambda)$ with non-negligible probability.*

3.2 Building Block: Function Secret Sharing

A function secret sharing scheme provides a method to split this function into a set of separate keys, where each key enable it to efficiently generate a share of evaluation $f(x)$, and yet each key individually does not reveal information about the details of function f. In [BGI15,BGI16], Boyle et al. formalized the syntax and security definition of function secret sharing. In this part, we first revisit the definition of function secret sharing along with a new security definition.

Syntax and Security Definition A (n, n)-function secret sharing scheme for a function family $\mathcal{F}$ consists of algorithms $(\mathsf{FSS.Setup}, \mathsf{FSS.ShareGen}, \mathsf{FSS.Reconstruct})$ described as follows:

– $\mathsf{FSS.Setup}(1^\lambda, n, \mathcal{F})$: Given the security parameter λ, the parameter n of the secret sharing system and the description of function family $\mathcal{F}$, the setup outputs the public parameters pp.
– $\mathsf{FSS.ShareGen}(\mathsf{pp}, f)$: Given pp and a function $f \in \mathcal{F}$, the share generation algorithm outputs n shares of function f as $\{f_i\}_{i=1}^n$.
– $\mathsf{FSS.Reconstruct}(\mathsf{pp}, \{f_i(x)\}_{i=1}^n)$: Given an input x, evaluating each function share f_i on x, we obtain n output shares $\{f_i(x)\}_{i=1}^n$. The reconstruction algorithm then aggregates all the share values $\{f_i(x)\}_{i=1}^n$ and outputs $f(x)$.

Definition 10 (Correctness). *We say that an (n, n)-function secret sharing scheme FSS for function family $\mathcal{F}$ is correct, if for any function $f \in \mathcal{F}$, $\forall x \in \mathsf{dom}(f)$, $\mathsf{pp} \leftarrow \mathsf{FSS.Setup}(1^\lambda, n, \mathcal{F})$, we have*

$$f(x) = \mathsf{FSS.Reconstruct}(\mathsf{pp}, \{f_i(x)\}_{i=1}^n)$$

where $\{f_i\}_{i=1}^n \leftarrow \mathsf{FSS.ShareGen}(\mathsf{pp}, f)$.

Security definition of FSS. In [BGI15,BGI16], Boyle et al. proposed a ind-based security definition. In their security definition, adversary is given $n-1$ shares of function f_b, where f_b is chosen randomly from (f_0, f_1) of adversary's choice. It requires that adversary cannot guess bit b correctly with overwhelming probability. We enhance the security of FSS by modeling it as simulation-based CCA-type one. More specifically, in additional to the $n-1$ shares of challenge function f^*, the adversary is given oracle access to the function share generation algorithm of his choice (different from challenge function f^*). Moreover, the adversary is given

oracle access to the share that she is not holding for f^*. The security requires that adversary cannot tell real execution from simulated one. The detailed definition is below.

Definition 11 (CCA-type of Security, Sim-based). *Let Π be a function secret sharing scheme for a function family $\mathcal{F}$. Consider a* PPT *adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ and simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3)$[6]. We say the function secret sharing scheme Π is simulation-secure if the following two distribution ensembles (over the security parameter λ) are computationally indistinguishable:*

***Real Distribution**:*

1. $\mathsf{pp} \leftarrow \mathsf{FSS.Setup}(1^\lambda, n, \mathcal{F})$.
2. $(f^*, \tau) \leftarrow \mathcal{A}_1^{\mathsf{FSS.ShareGen}(\mathsf{pp}, \cdot)}(\mathsf{pp})$
3. $\{f_i^*\}_{i=1}^n \leftarrow \mathsf{FSS.ShareGen}(\mathsf{pp}, f^*)$
4. $\alpha \leftarrow \mathcal{A}_2^{\mathsf{FSS.ShareGen}(\mathsf{pp}, \cdot), f_n^*(\cdot)}(\mathsf{pp}, \{f_i^*\}_{i=1}^{n-1}, \tau)$
5. *Output* $(\mathsf{pp}, f^*, \tau, \alpha)$.

***Ideal Distribution**:*

1. $\mathsf{pp} \leftarrow \mathcal{S}_1(1^\lambda, n, \mathcal{F})$.
2. $(f^*, \tau) \leftarrow \mathcal{A}_1^{\mathcal{S}_2(\cdot)}(\mathsf{pp})$.
3. $\{f_i^*\}_{i=1}^{n-1} \leftarrow \mathcal{S}_2(|f^*|)$
4. $\alpha \leftarrow \mathcal{A}_2^{\mathcal{S}_2(\cdot), \mathcal{S}_3^{f^*}(\{f_i^*\}_{i=1}^{n-1}, \cdot)}(\mathsf{pp}, \{f_i^*\}_{i=1}^{n-1}, \tau)$.
5. *Output* $(\mathsf{pp}, f^*, \tau, \alpha)$.

where on query x made by adversary $\mathcal{A}_2$, simulator $\mathcal{S}_3^{f^*(\cdot)}(\cdot)$ makes a single query to oracle $f^*(\cdot)$ on x.

FSS Construction. Let $\mathsf{SP} = (\mathsf{SP.Gen}, \mathsf{SP.Enc}, \mathsf{SP.Dec}, \mathsf{SP.Eval})$ be a $\mathcal{F}$-AFS-spooky encryption as defined in Definition 4. To make the description simpler, we add a temporary algorithm, $\hat{f}_i(x) \leftarrow \mathsf{LocalEval}(\hat{f}_i, x)$, which locally evaluates x using the i-th share $\hat{f}_i$. The construction of function secret sharing scheme $\Pi = (\mathsf{FSS.Setup}, \mathsf{FSS.ShareGen}, \mathsf{FSS.Reconstruct})$ for poly(λ)-depth circuit family $\mathcal{F}$ is the following:

- $\mathsf{FSS.Setup}(1^\lambda, n, \mathcal{F})$: The setup algorithm outputs public parameter $\mathsf{pp} = (n, \mathcal{F})$ for the system.
- $\mathsf{FSS.ShareGen}(\mathsf{pp}, f)$: On input a function $f \in \mathcal{F}$, the share generation algorithm first generates a n-out-of-n secret sharing $\{f_i\}_{i=1}^n$ of the description of f, and for $i \in [n]$ computes $(\mathsf{SP.pk}_i, \mathsf{SP.sk}_i) \leftarrow \mathsf{SP.Gen}(1^\lambda)$. Then for $i \in [n]$, encrypt the description share using spooky encryption $\mathsf{SP.Enc}(\mathsf{pk}_i, f_i)$. Output the i-th share of function f as $f_i = (\mathsf{SP.sk}_i, \{\mathsf{SP.pk}_i\}_{i=1}^n, \{\mathsf{SP.Enc}(\mathsf{SP.pk}_i, f_i)\}_{i=1}^n)$.

[6] Looking ahead, we overload the notation of S_2 in the ideal distribution, by allowing it to take two kinds of inputs: 1. the description of a function f; 2. the size of a function f.

– $\mathsf{FSS.LocalEval}(f_i, x)$: On input the i-th share f_i, which is composed of the items $(\mathsf{SP.sk}_i, \{\mathsf{SP.pk}_i\}_{i=1}^n, \{\mathsf{SP.Enc}(\mathsf{SP.pk}_i, f_i)\}_{i=1}^n)$, and a value x, run the spooky evaluation $\{c_i\}_{i=1}^n = \mathsf{SP.Eval}(C_x, \{\mathsf{SP.Enc}(\mathsf{SP.pk}_i, f_i)\}_{i=1}^n)$, where the circuit $C_x(\cdot)$ is defined as

Hardcode: value x. **Input**: $\{\mathsf{SP.Enc}(\mathsf{SP.pk}_i, f_i)\}_{i=1}^n$.
1. Compute $\hat{f} = \sum_{i=1}^n \mathsf{SP.Enc}(\mathsf{SP.pk}_i, f_i)$.
2. Compute $\hat{f}(x)$.

Fig. 1. Description of function $C_x(\cdot)$

Then output $s_i = \mathsf{SP.Dec}(\mathsf{SP.sk}_i, c_i)$.
– $\mathsf{FSS.Reconstruct}(\mathsf{pp}, \{s_i\}_{i=1}^n)$: Given the n shares $\{s_i\}_{i=1}^n$ of function $f(x)$, the reconstruction algorithm outputs $f(x) = \sum_{i=1}^n s_i$.

Correctness Proof. The correctness of our FSS construction is proved using properties of $\mathcal{F}$-AFS-spooky encryption as defined in Definition 4.

Lemma 1. *Our FSS construction described above is correct (c.f. Definition 10).*

Proof. Assuming wlog that the evaluate algorithm is deterministic, we obtain the same $\{c_i\}_{i=1}^n = \mathsf{SP.Eval}(C_x, \{\mathsf{SP.Enc}(\mathsf{SP.pk}_i, f_i)\}_{i=1}^n)$ in algorithm $\mathsf{FSS.LocalEval}(\hat{f}_i, x)$, for $i \in [n]$. By the correctness of $\mathcal{F}$-AFS-spooky encryption as stated in Definition 4, we have $\sum_{i=1}^n s_i = C_x(\{f_i\}_{i=1}^n) = f(x)$, where $s_i = \mathsf{SP.Dec}(\mathsf{SP.sk}_i, c_i)$.

Security Proof. In this part, we show that our construction of FSS is secure as defined in Definition 11. Intuitively, for function queries other than the challenge one, the simulation computes in the exactly same method as the real execution. For the challenge function, we rely on the semantic security and psedurandomness of $n-1$ evaluations of challenge function shares on any input, provided by the underlying spooky encryption to show the indistinguishability between real and simulated executions. The proof detail is the following.

Theorem 2. *Let* SP *be a secure $\mathcal{F}$-AFS-spooky encryption as defined in Definition 4. Our construction of FSS described above is secure (c.f. Definition 11).*

Proof. We first describe the simulation algorithm $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3)$ that are used in the proof.

– $\mathcal{S}_1(1^\lambda, n, \mathcal{F})$: Run $\mathsf{FSS.Setup}(1^\lambda, n, \mathcal{F})$ to obtain pp and output pp.
– $\mathcal{S}_2(\mathsf{inp})$: On input $\mathsf{inp} = f_i$ or $\mathsf{inp} = |f^*|$:
  • On input function f_i, first look for $(f_i, \{f_ij\}_{j=1}^n)$ in local storage. If found, output $(f_i, \{f_{ij}\}_{j=1}^n)$. Otherwise, compute $\{f_{ij}\}_{j=1}^n \leftarrow \mathsf{FSS.ShareGen}(\mathsf{pp}, f_i)$ and store $(f_i, \{f_ij\}_{j=1}^n)$ locally. Then output $(f_i, \{f_{ij}\}_{j=1}^n)$.

- On input size $|f^*|$, first choose $n-1$ bit strings t_i of size $|f^*|$. For $i \in [n]$ computes $(\mathsf{SP.pk}_i, \mathsf{SP.sk}_i) \leftarrow \mathsf{SP.Gen}(1^\lambda)$. Then for $i \in [n-1]$, encrypt the description share using spooky encryption $\mathsf{ct}_i \leftarrow \mathsf{SP.Enc}(\mathsf{pk}_i, t_i)$, and $\mathsf{ct}_n \leftarrow \mathsf{SP.Enc}(\mathsf{pk}_i, 0^{|f^*|})$. Output $f_j^* = (\mathsf{SP.sk}_j, \{\mathsf{SP.pk}_i\}_{i=1}^n, \{\mathsf{ct}_i\}_{i=1}^n)$ for $j \in [n-1]$.

– $\mathcal{S}_3^{f^*}(\{f_i^*\}_{i\in S}, x)$: On input $n-1$ shares $\{f_i^*\}_{i\in S}$ and x, for $i \in S$, compute $y_i \leftarrow \mathsf{FSS.LocalEval}(f_i^*, x)$. Then call the oracle f on input x to obtain $y = f^*(x)$. Output $y_n = y - \sum_{i\in S} y_i$.

The view of adversary includes $(\mathsf{pp}, f^*, \tau, \alpha)$, where (τ, α) are states that incorporate adversary's queries to $\mathsf{FSS.ShareGen}(\mathsf{pp}, \cdot)$ (or $\mathcal{S}_2$) and $f^*_{i\notin[S]}$ (or $\mathcal{S}_3$). As we described above, $\mathcal{S}_1(1^\lambda, n, \mathcal{F})$ computes $\mathsf{FSS.Setup}(1^\lambda, n, \mathcal{F})$ as a subroutine, so the output pp is identical in these two procedures. For each function query f_i, $\mathcal{S}_2(f_i)$ calls $\mathsf{FSS.ShareGen}(\mathsf{pp}, f_i)$ as a subroutine, so the output of $\mathcal{S}_2(f_i)$ is identical to that of $\mathsf{FSS.ShareGen}(\mathsf{pp}, f_i)$. For challenge function query $\mathcal{S}_2(|f^*|)$, the shares given to adversary are $f_j^* = (\mathsf{SP.sk}_j, \{\mathsf{SP.pk}_i\}_{i=1}^n, \{\mathsf{ct}_i\}_{i=1}^n)$ for $j \in [n-1]$. By the semantic security of underlying spooky encryption, ct_n remains secure. By the second property of spooky encryption (c.f. Definition 4), any $n-1$ of the shares y_i above are distributed ϵ-close to uniform, where $y_i = \mathsf{FSS.LocalEval}(f_i^*, x)$ for any x.

Lastly, on query x, in the real execution, adversary gets back $y_n = f_n^*(x)$, while in the ideal execution, he gets back $y_n = y - \sum_{i\in S} y_i$, where $y_i \leftarrow \mathsf{FSS.LocalEval}(f_i^*, x)$. Also by property of spooky encryption as stated in Definition 5, the $n-1$ shares $\{y_i\}$ are distributed ϵ-close to uniform. Thus, y_n in the ideal execution is a valid share and is identical to that in real execution.

3.3 Instantiation of DFE from FSS

Let $\mathsf{FSS} = (\mathsf{FSS.Setup}, \mathsf{FSS.ShareGen}, \mathsf{FSS.LocalEval}, \mathsf{FSS.Reconstruct})$ be a function secret sharing scheme for function ensemble $\mathcal{F}$, and $\mathsf{FE} = (\mathsf{FE.Setup}, \mathsf{FE.Keygen}, \mathsf{FE.Enc}, \mathsf{FE.Dec})$ be a functional encryption. The description of DFE scheme $\mathsf{DFE} = (\mathsf{Setup}, \mathsf{Keygen}, \mathsf{Enc}, \mathsf{PartDec}, \mathsf{Reconstruct})$ is as follows:

– $\mathsf{DFE.Setup}(1^\lambda, n)$: Run the FSS setup algorithm $\mathsf{FSS.pp} \leftarrow \mathsf{FSS..Setup}(1^\lambda, n)$ and the FE setup algorithm $(\mathsf{FE.pp}, \mathsf{FE.msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$. Output the public parameters pp and master secret key msk as

$$\mathsf{pp} = (\mathsf{FSS.pp}, \mathsf{FE.pp}), \quad \mathsf{msk} = \mathsf{FE.msk}$$

– $\mathsf{DFE.Enc}(\mathsf{pp}, m)$: Run the FE encryption algorithm $\mathsf{ct} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.pp}, m)$. Output ciphertext ct.

– $\mathsf{DFE.Keygen}(\mathsf{msk}, f)$: Given a function $f \in \mathcal{F}$ and msk, the key generation algorithm first runs the share generation algorithm in FSS as $\{f_i\}_{i=1}^n \leftarrow \mathsf{FSS.ShareGen}(\mathsf{FSS.msk}, f)$, and then compute the key shares by running the FE key generation as $\mathsf{sk}_i^f \leftarrow \mathsf{FE.Keygen}(\mathsf{FE.msk}, C_i)$, for $i \in [n]$, where the function $C_i(\cdot)$ is defined as

Hardcode: function share f_i	**Input**: value x.
Compute and output $c_i = \mathsf{FSS.LocalEval}(f_i, x)$	

Fig. 2. Description of function $C_i(\cdot)$

Output the secret key shares $\{\mathsf{sk}_i^f\}_{i=1}^n$.

- $\mathsf{DFE.PartDec}(\mathsf{ct}, \mathsf{sk}_i^f)$: Given the i-th secret key share sk_i^f, compute and output $s_i = \mathsf{FE.Dec}(\mathsf{sk}_i^f, \mathsf{ct})$.
- $\mathsf{DFE.Reconstruct}(\mathsf{pp}, \{s_i\}_{i=1}^n)$: Output the reconstructed result as $f(m) = \sum_{i=1}^n s_i$.

Correctness Proof. The correctness proof of our DFE construction follows directly from the correctness of FSS and FE. First by the correctness of FSS scheme FSS, the output of circuit C_i (c.f. Figure 3.3) satisfies $\sum_{i=1}^n c_i = f(m)$. Secondly, by correctness of functional encryption scheme FE, the output of $s_i = \mathsf{FE.Dec}(\mathsf{sk}_f^i, \mathsf{ct})$, where sk_f^i is secret key for circuit C_i satisfies $s_i = c_i$. Therefore in $\mathsf{DFE.Keygen}(\mathsf{msk}, f)$, we also get $f(m) = \sum_{i=1}^n s_i$.

Security Proof. In this part, we show that our construction of DFE satisfies (sim-based) function privacy and data privacy as defined above. The function privacy of our DFE construction mainly is based on the sim-based security of FSS (c.f. Definition 11), thus in our proof below, we use the simulation algorithm of FSS to setup the system and answer adversary's queries. The data privacy of our DFE construction directly follows the ind-based data privacy of underlying functional encryption (c.f. Definition 3).

Theorem 3. *Let* FSS *be function secret sharing scheme satisfying security as defined in Definition 11, our construction of DFE described above is function private (c.f. Definition 8).*

Proof. We first describe the simulation algorithm $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3)$ based on the simulation algorithms of FSS, $(\mathsf{FSS}.\mathcal{S}_1, \mathsf{FSS}.\mathcal{S}_2)$ (as described in the proof of Theorem 2), that are used in the proof.

- $\mathcal{S}_1(1^\lambda, n)$: Run the FSS simulated setup algorithm $\mathsf{FSS.pp} \leftarrow \mathsf{FSS}.\mathcal{S}_1(1^\lambda, n)$ and the FE setup algorithm $(\mathsf{FE.pp}, \mathsf{FE.msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$. Send $\mathsf{pp} = (\mathsf{FSS.pp}, \mathsf{FE.pp})$ to adversary.
- $\mathcal{S}_2(f)$: On input function query f, first look for $(f, \{\mathsf{sk}_i^f\}_{i=1}^n)$ in local storage. If found, send $(f_i, \{\mathsf{sk}_i^f\}_{i=1}^n)$ to adversary. Otherwise, $\mathcal{S}_2$ runs the simulation algorithm $\mathsf{FSS}.\mathcal{S}_2(f)$ of FSS to obtain $\{f_i\}_{i=1}^n$ as shares of function f. Then for $i \in [n]$, compute $\mathsf{sk}_i^f \leftarrow \mathsf{FE.Keygen}(\mathsf{msk}, f_i)$ and store $(f, \{\mathsf{sk}_i^f\}_{i=1}^n)$ locally. Send $(f, \{\mathsf{sk}_i^f\}_{i=1}^n)$ to adversary.
- $\mathcal{S}_3^{f^*}(\{\mathsf{sk}_i^{f^*}\}_{i=1}^{n-1}, \mathsf{ct})$: On input ciphertext query ct, first compute $x = \mathsf{FE.Dec}(\mathsf{sk}_{\mathsf{id}}, \mathsf{ct})$, where $\mathsf{sk}_{\mathsf{id}} \leftarrow \mathsf{FE.Keygen}(\mathsf{msk}, \mathsf{id})$ and id denotes the identity function. Then for $i \in [n-1]$, compute $s_i = \mathsf{FE.Dec}(\mathsf{sk}_i^{f^*}, \mathsf{ct})$. Output $s_n = f(x) - \sum_{i=1}^{n-1} s_i$.

In the following, we show, that adversary's view $(\mathsf{pp}, f^*, \tau, \alpha)$, where (τ, α) are states that incorporate adversary's queries to $\mathsf{DFE.Keygen}$ (or $\mathcal{S}_2$) and $\mathsf{DFE.Dec}(\mathsf{sk}_n^{f^*}, \cdot)$ (or $\mathcal{S}_3$), are indistinguishable in the two executions. As described above, $\mathcal{S}_1$ computes the FSS simulated setup $\mathsf{FSS}.\mathcal{S}_1$ and a real $\mathsf{FE.Setup}$ as subroutines, by the security of underlying FSS scheme, we have the distribution of public parameters in real and ideal executions are statistically close. Similarly, by the security of underlying FSS scheme, the function shares $\{f_i\}_{i=1}^n \leftarrow \mathsf{FSS}.\mathcal{S}_2(f)$ computed in simulation $\mathcal{S}_2(f)$ is indistinguishable from that in the real execution $\mathsf{DFE.Keygen}(f)$, thus the responses for key queries in the real and ideal executions are indistinguishable. Lastly, the output $s_n = \mathcal{S}_3^{f^*}(\{\mathsf{sk}_i^{f^*}\}_{i=1}^{n-1}, \mathsf{ct})$, where $\mathsf{ct} = \mathsf{FE.enc}(\mathsf{pp}, x)$, satisfies $\sum_{i=1}^n s_i = f(x)$, where $s_i = \mathsf{FE.Dec}(\mathsf{sk}_i^{f^*}, \mathsf{ct})$ can be computed by the adversary himself. In conclusion, the view of adversary in real execution is indistinguishable from that in the ideal execution.

Theorem 4. *Let* FE *be functional encryption scheme satisfying ind-based data privacy as defined in Definition 3, our construction of DFE described above is data private (c.f. Definition 9).*

Proof. The ciphertext in our DFE construction is indeed a FE ciphertext, thus by ind-based data privacy of FE scheme, our construction of DFE is data private.

4 Hosting Services Securely in Multiple Clouds

In [BGMS15], the authors consider a setting of hosting service in untrusted clouds: there exist three parties: *Service provider* who owns a program and setups the whole system, *cloud server* where the program is hosted, and arbitrary many *clients*. Intuitively speaking, the service provider wants to host the program P on a cloud server, and additionally it wants to authenticate clients who pay for the service provided by program P. This authentication should allow a legitimate user to access the program hosted on the cloud server and compute output on inputs of his choice. Moreover, the program P could contain proprietary information, thus needs to be kept confidential. The authors in [BGMS15] also require that the scheme satisfies some essential properties:

Weak client: The amount of work performed by client should only depends on the size of input and security parameter, but independent of the running time of program P.

Delegation: The work performed by the service provider includes one-time setup of the whole system and authentication clients. The amount of work in one-time setup phase should be bounded by a fixed polynomial in the program size, while the amount of work incurred in authentication should only depend on the security parameter.

Polynomial slowdown: The running time of encoded program (running on cloud server) is bounded by a fixed polynomial in the running time of program P.

Boneh et al give a construction based on indistinguishability obfuscation, and their construction suffers from a restriction that the number of corrupted clients should be pre-fixed [BGMS15]. In this section, we generalize the above model by distributing encoded program shares to multiple cloud servers and resolve the open problem that to remove the restriction on number of corrupted clients from [BGMS15].

In our Distributed Secure Cloud Service (DSCS) scheme, the service provider generates a set of encoded program shares for program P, and then hosts each encoded program share on one cloud server. Any authenticated users can access the encoded program shares hosted multiple cloud servers and compute output on inputs of his choice. We also require that our DSCS scheme satisfied the above three properties.

4.1 Syntax and Security Definitions

The Distributed Secure Cloud Service scheme consists of algorithms $\mathsf{DSCS} = (\mathsf{DSCS.Prog}, \mathsf{DSCS.Auth}, \mathsf{DSCS.Inp}, \mathsf{DSCS.Eval}, \mathsf{DSCS.Reconstruct})$ with details as follows:

- $\mathsf{DSCS.Prog}(1^\lambda, n, P)$: On input the security parameter λ, the threshold parameter n and a program P, it returns the distributed encoded program $\{\tilde{P}_i\}_{i=1}^n$ and a secret sk to be useful in authentication.
- $\mathsf{DSCS.Auth}(\mathsf{id}, \mathsf{sk})$: On input the identity id of a client and the secret sk, it produces an authentication token $\mathsf{token}_{\mathsf{id}}$ for the client.
- $\mathsf{DSCS.Inp}(\mathsf{token}_{\mathsf{id}}, x)$: On input the authentication token $\mathsf{token}_{\mathsf{id}}$ and an input x, it outputs an encoded input $\tilde{x}$ and α which is used by the client to later decode the evaluated results.
- $\mathsf{DSCS.Eval}(\tilde{P}_i, \tilde{x})$: On input the encoded program $\tilde{P}_i$ and input $\tilde{x}$, it produces the encoded distributed result $\tilde{y}_i = \tilde{P}_i(\tilde{x})$.
- $\mathsf{DSCS.Reconstruct}(\{\tilde{P}_i(\tilde{x})\}_{i=1}^n)$: On input the evaluated result $\{\tilde{P}_i(\tilde{x})\}_{i=1}^n$, it reconstructs the result $P(x)$.

Similar to the analysis in [BGMS15], the procedure goes as follows: the service provider first runs the procedure $\mathsf{Prog}(1^\lambda, P)$ to obtain the distributed encoded program $\{\tilde{P}_i\}_{i=1}^n$ and the secret σ. Then for $i \in [n]$, it will send $\tilde{P}_i$ to cloud server i. Later, the service provider will authenticate users using σ. A client with identity id, who has been authenticated, will encode his input using procedure $\mathsf{Inp}(1^\lambda, \sigma_{\mathsf{id}}, x)$. The client will send $\tilde{x}$ to cloud i, for $i \in [n]$. For $i \in [n]$, the cloud will evaluate the program $\tilde{P}_i$ on encoded input $\tilde{x}$ and return the result $P_i(x)$. Finally, the client can run $\mathsf{Reconstruct}(\{P_i(x)\}_{i=1}^n)$ to obtain the result $P(x)$.

Security definitions. In [BGMS15], the authors consider two cases for security definition, namely *untrusted cloud security* and *untrusted client security*. We generalize their security definition to the DSCS setting. More specifically, we decouple the case of untrusted cloud security into two subcases, *program privacy* and *input privacy* in untrusted cloud security. And in untrusted client security,

we enhance it by allowing the set of corrupt clients colluding with some corrupt servers. In various security definitions below, we assume that the service provider is uncompromisd.

For program privacy in untrusted cloud case, the service provider first setup the whole system based on program (P_0, P_1) submitted by adversary. In the system, the adversary can corrupts a set of servers and also has access to authentication and encoding oracles, but he cannot tell which program P_b is used to setup the system. The only restriction here is that P_0 and P_1 are of the same size.

Definition 12 (Untrusted Cloud Security – Program Privacy). *For the program privacy case in untrusted cloud setting, we first describe the following experiment* $\mathbf{Expt}^{\mathsf{prog}}(1^\lambda)$ *between a challenger and adversary* $\mathcal{A}$*:*

- ***Setup****: The adversary sends challenge programs* (P_0, P_1) *to challenger. The challenger choose a random bit* $b \in \{0,1\}$ *and obtains the challenge encoded program* $(\{\tilde{P}_i\}_{i=1}^n, \mathsf{sk}) \leftarrow \mathsf{DSCS.Prog}(1^\lambda, n, P_b)$ *and sends* $\{\tilde{P}_i\}_{i=1}^{n-1}$ *to adversary* $\mathcal{A}$*.*
- ***Query phase****: Proceeding adaptively, the adversary* $\mathcal{A}$ *can submit the following two kinds of queries:*
  - ***Authentication query****:* $\mathcal{A}$ *sends identity* id_i *to challenger. The challenger computes* $\mathsf{token}_{\mathsf{id}_i} \leftarrow \mathsf{DSCS.Auth}(\mathsf{id}, \mathsf{sk})$ *and sends back* $\mathsf{token}_{\mathsf{id}_i}$*.*
  - ***Input query****:* $\mathcal{A}$ *sends* (x, id) *to challenger. The challenger computes* $\mathsf{ct} \leftarrow \mathsf{DSCS.Inp}(\mathsf{token}_{\mathsf{id}}, x)$*, where* $\mathsf{token}_{\mathsf{id}_i} \leftarrow \mathsf{DSCS.Auth}(\mathsf{id}, \mathsf{sk})$*. Then send back* ct*.*
- ***Guess****: Finally, the adversary* $\mathcal{A}$ *outputs his guess* b' *for the bit* b*.*

We say the adversary wins the experiment if $b' = b$*.*

A DSCS scheme is program private in untrusted cloud setting if no PPT *adversary* $\mathcal{A}$ *can win the experiment* $\mathbf{Expt}^{\mathsf{prog}}(1^\lambda)$ *with non-negligible probability.*

For input privacy in untrusted cloud security, the service provider first sets up the whole system using program P submitted by adversary. Then in the system, the adversary corrupts all servers, and additionally has access to authentication oracles, but he cannot distinguish the encryption of two message (m_0, m_1), where $P(m_0) = P(m_1)$. Put simply, beyond the evaluation of program, he learns nothing about the underlying message.

Definition 13 (Untrusted Cloud Security – Input Privacy). *For the input privacy case in untrusted cloud setting, we first describe the following experiment* $\mathbf{Expt}^{\mathsf{inp}}(1^\lambda)$ *between a challenger and adversary* $\mathcal{A}$*:*

- ***Setup****: The adversary sends challenge program* P *to challenger. The challenger runs* $(\{\tilde{P}_i\}_{i=1}^n, \mathsf{sk}) \leftarrow \mathsf{DSCS.Prog}(1^\lambda, n, P)$ *and sends* $\{\tilde{P}_i\}_{i=1}^n$ *to adversary* $\mathcal{A}$*.*
- ***Authentication query phase I****: Proceeding adaptively, the adversary* $\mathcal{A}$ *sends identity* id_i *to challenger. The challenger computes* $\mathsf{token}_{\mathsf{id}_i} \leftarrow \mathsf{DSCS.Auth}(\mathsf{id}, \mathsf{sk})$ *and sends back* $\mathsf{token}_{\mathsf{id}_i}$*.*

– ***Challenge phase***: *Adversary submits the challenge message pair* $(m_0, m_1, \mathsf{id}^*)$ *with the constraint that* $P(m_0) = P(m_1)$. *The challenger first chooses a random bit* $b \in \{0,1\}$ *and computes* $\mathsf{ct} \leftarrow \mathsf{DSCS.Inp}(\mathsf{token}_{\mathsf{id}}, m_b)$, *where* $\mathsf{token}_{\mathsf{id}_i^*} \leftarrow \mathsf{DSCS.Auth}(\mathsf{id}^*, \mathsf{sk})$. *Then send* ct *to adversary.*
– ***Authentication query phase II***: *The same as* ***Authentication query phase I*** *with the restriction that the query* id_i *does not equal* id^* *in the challenge phase*
– ***Guess***: *Finally, the adversary* $\mathcal{A}$ *outputs his guess* b' *for the bit* b.

We say the adversary wins the experiment if $b' = b$.

A DSCS scheme is data private in untrusted cloud setting if no PPT *adversary* $\mathcal{A}$ *can win the experiment* $\mathbf{Expt}^{\mathsf{inp}}(1^\lambda)$ *with non-negligible probability.*

Remark 2. We note that in the above data privacy definition, the challenge phase can be access multiple times as long as the query pair $(m_0, m_1, \mathsf{id}_i^*)$ satisfies $P(m_0) = P(m_1)$, and challenger use the same random bit b in generating the challenge ciphertext.

For untrusted client security, a collection of corrupt clients with the help of a subset of corrupt servers do not learn anything beyond the program's output with respect to their identities on certain inputs of their choice, and if a client is not authenticated, it learns nothing.

Definition 14 (Untrusted Client Security). *Let* DSCS *be the secure Distributed Secure Cloud Service scheme as described above. We say the scheme satisfies untrusted client security if the following holds. Let* $\mathcal{A}$ *be a* PPT *adversary who corrupts* ℓ *clients* $I = \{\mathsf{id}_1, \ldots, \mathsf{id}_\ell\}$. *Consider any program* P, *let* $Q = \mathrm{poly}(\lambda)$. *The experiment described below requires one additional procedure* decode. *Based on these two procedures, we define simulator* $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$. *Consider the following two experiments:*

The experiment $\mathsf{Real}(1^\lambda)$ *is as follows:*
1. $(\{\tilde{P}\}_{i=1}^n, \mathsf{sk}) \leftarrow \mathsf{DSCS.Prog}(1^\lambda, n, P)$.
2. *For all* $i \in [\ell]$, $\mathsf{token}_{\mathsf{id}_i} \leftarrow \mathsf{DSCS.Auth}(\mathsf{id}_i, \mathsf{sk})$.
3. *For* $i \in [Q]$, $\mathcal{A}(\{\tilde{P}\}_{i=1}^{n-1})$ *adaptively sends an encoding* $\tilde{x}_i$, *using identity* id, *and gets back response*

$$\tilde{y}_{ij} = \tilde{P}_j(x_i) \leftarrow \mathsf{DSCS.Eval}(\tilde{P}_j, \tilde{x}_i), \forall j \in [n]$$

4. *Output* $(\{\tilde{P}\}_{i=1}^{n-1}, \{\mathsf{token}_{\mathsf{id}_i}\}_{i\in[\ell]}, \{\tilde{y}_{ij}\})$.

The experiment $\mathsf{Ideal}^{\mathsf{P}}(1^\lambda)$ *is as follows:*
1. $\{\tilde{P}_i'\}_{i=1}^n \leftarrow \mathcal{S}_1(1^\lambda, n)$.
2. *For all* $i \in [\ell]$, $\mathsf{token}_{\mathsf{id}_i} \leftarrow \mathcal{S}_2(\mathsf{id}_i)$
3. *For* $i \in [Q]$, $\mathcal{A}(\{\tilde{P}\}_{i=1}^{n-1})$ *adaptively sends an encoding* $\tilde{x}_i$, *using identity* id,
   – *If* $\mathsf{id} \notin I$, *then return* $\tilde{y}_{ij} = \perp$ *for* $j \in [n]$.

– *Otherwise, compute* $x_i = \mathsf{decode}(\sigma, \tilde{\mathsf{x}}_\mathsf{i})$. *Then the simulator sends* (id, x_i) *to oracle* P *and obtains* $y_i = P(\mathsf{id}, x_i)$. *Simulator then sends shares* $\{y_{ij}\}_{j\in[n]}$ *of* y_i *to adversary* $\mathcal{A}$.

4. *Output* $(\{\tilde{P}\}_{i=1}^{n-1}, \{\mathsf{token}_{\mathsf{id}_i}\}_{i\in[\ell]}, \{\tilde{y}_{ij}\})$.

Then we have $\mathsf{Real}(1^\lambda) \overset{\mathsf{c}}{\approx} \mathsf{Sim}^{\mathsf{P}}(1^\lambda)$.

4.2 Our DSCS Construction

Let $\mathsf{DFE} = (\mathsf{DFE.Setup}, \mathsf{DFE.Keygen}, \mathsf{DFE.Enc}, \mathsf{DFE.PartDec}, \mathsf{DFE.Reconstruct})$ be a distributed functional encryption and $\Sigma = (\Sigma.\mathsf{Setup}, \Sigma.\mathsf{Sign}, \Sigma.\mathsf{Verify})$ be an existential unforgeable signature scheme. We describe our construction for DSCS as follows:

– $\mathsf{DSCS.Prog}(1^\lambda, n, P)$: First run

$$(\mathsf{DFE.pp}, \mathsf{DFE.msk}) \leftarrow \mathsf{DFE.Setup}(1^\lambda, n), \quad (\Sigma.\mathsf{sk}, \Sigma.\mathsf{vk}) \leftarrow \Sigma.\mathsf{Setup}(1^\lambda)$$

Then let the augmented program P_{aug} be

Hardcode: $\mathsf{DFE.pp}, \Sigma.\mathsf{vk}$ and program P **Input**: signature σ, value x and id.

1. If $\Sigma.\mathsf{Verify}(\sigma, \mathsf{id}, \Sigma.\mathsf{vk}) = 0$, output $\perp$.
2. Compute and output $P(x)$.

Fig. 3. Description of augmented program P_{aug}

And compute $\{\mathsf{sk}_i^{P_{\mathsf{aug}}}\}_{i=1}^n \leftarrow \mathsf{DFE.Keygen}(\mathsf{DFE.msk}, P_{\mathsf{aug}})$. For $i \in [n]$, define the distributed encoded program $\tilde{P}_i$[7] as

Hardcode: $\mathsf{sk}_i^{P_{\mathsf{aug}}}$ and algorithm $\mathsf{DFE.Dec}$. **Input**: ciphertext ct.

Compute and output $\mathsf{DFE.PartDec}(\mathsf{sk}_i^{P_{\mathsf{aug}}}, \mathsf{ct})$.

Fig. 4. Description of distributed encoded program $\tilde{P}_i$

Output $\{\tilde{P}_i\}_{i=1}^n$ and secret $\mathsf{sk} = (\Sigma.\mathsf{sk}, \mathsf{DFE.pp})$.

– $\mathsf{DSCS.Auth}(\mathsf{id}, \mathsf{sk})$: First parse $\mathsf{sk} = (\Sigma.\mathsf{sk}, \mathsf{DFE.pp})$, and then compute $\sigma_{\mathsf{id}} \leftarrow \Sigma.\mathsf{Sign}(\Sigma.\mathsf{sk}, \mathsf{id})$. Output $\mathsf{token}_{\mathsf{id}} = (\sigma_{\mathsf{id}}, \mathsf{DFE.pp})$.
– $\mathsf{DSCS.Inp}(\mathsf{token}_{\mathsf{id}}, x)$: First parse $\mathsf{token}_{\mathsf{id}} = (\sigma_{\mathsf{id}}, \mathsf{DFE.pp})$, then compute $\mathsf{ct} \leftarrow \mathsf{DFE.Enc}(\mathsf{DFE.pp}, \sigma_{\mathsf{id}} || x)$. Output ciphertext $\tilde{x} = \mathsf{ct}$.
– $\mathsf{DSCS.Eval}(\tilde{P}_i, \tilde{x})$: Compute and output $\tilde{y}_i = \tilde{P}_i(\tilde{x})$.
– $\mathsf{DSCS.Reconstruct}(\{\tilde{y}_i\}_{i=1}^n)$: Compute and output $y = \mathsf{DFE.Reconstruct}(\{\tilde{y}_i\}_{i\in[n]})$.

[7] We note that the distributed encoded program $\tilde{P}_i$ does not require obfuscation

Correctness Proof. The correctness proof of our DSCS construction follows directly from the correctness of underlying distributed functional encryption scheme DFE and signature scheme Σ. As we described above, in the distributed encoded program $\tilde{P}_i$, it outputs $\perp$ for an invalid signature, otherwise outputs $\tilde{y}_i = \mathsf{DFE.PartDec}(\mathsf{sk}_i^{P_{\mathsf{aug}}}, \mathsf{ct})$, where $\mathsf{ct} = \mathsf{DFE.Enc}(\mathsf{DFE.pp}, \sigma_{\mathsf{id}}||x)$. By correctness of DFE, the output of $\mathsf{DSCS.Reconstruct}(\{\tilde{y}_i\}_{i=1}^n)$ is $P(x)$.

Security Proof. In this part, we show that our DSCS construction satisfies untrusted cloud security (program and data privacy) and untrusted client security as defined above. Intuitively, the program privacy in untrusted cloud setting can reduce to the ind-based function privacy of underlying DFE scheme, thus in the proof we construction a reduction that reduces the program privacy property to the ind-based function privacy of DFE scheme. The data privacy in untrusted cloud setting can be based on ind-based data privacy of DFE scheme, so similarly we show a reduction that bounds this two properties together. Lastly, the untrusted client security is based on the sim-based function privacy of DFE scheme. Therefore, we use the simulation algorithms of DFE to do the simulation for our DSCS construction. The detailed proofs are as follows.

Theorem 5. *Let distributed functional encryption* DFE *satisfy ind-based function privacy (c.f. Definiton 7), then our DSCS construction described above satisfies program privacy in untrusted cloud setting (c.f. Definition 12).*

Proof. We describe a reduction $\mathcal{B}$ against the ind-based function privacy of underlying DFE scheme. If the adversary $\mathcal{A}$ can win the experiment $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{DFE\text{-}func}}(1^\lambda)$ as defined in Definition 7, then reduction $\mathcal{B}$ can also win the experiment $\mathbf{Expt}^{\mathsf{prog}}(1^\lambda)$ as defined in Definition 12. The description of reduction $\mathcal{B}$ is as follows:

- **Setup**: $\mathcal{B}$ interacts with the challenger of DFE to obtain DFE.pp and computes $(\Sigma.\mathsf{vk}, \Sigma.\mathsf{sk}) \leftarrow \Sigma.\mathsf{Setup}(1^\lambda)$. Then $\mathcal{B}$ invokes adversary $\mathcal{A}$ to get the challenge programs (P^0, P^1). Next, $\mathcal{B}$ sends the augmented program $(P^0_{\mathsf{aug}}, P^1_{\mathsf{aug}})$ (as described in Figure 3) to the challenger of DFE, and the challenger sends back $\{\mathsf{sk}_i^{P^b_{\mathsf{aug}}}\}_{i=1}^n$. Lastly, $\mathcal{B}$ sends $\{\tilde{P}_i\}_{i=1}^{n-1}$ to adversary, where $\{\tilde{P}_i\}_{i=1}^n$ are constructed as in Figure 4 using $\{\mathsf{sk}_i^{P^b_{\mathsf{aug}}}\}_{i=1}^n$ as input.
- **Identity query**: On input identity query id_i, $\mathcal{B}$ computes $\sigma_{\mathsf{id}} \leftarrow \Sigma.\mathsf{Sign}(\Sigma.\mathsf{sk}, \mathsf{id})$ and sends back $\mathsf{token}_{\mathsf{id}} = (\sigma_{\mathsf{id}}, \mathsf{DFE.pp})$ to adversary $\mathcal{A}$.
- **Input query**: On input (x, id), $\mathcal{B}$ computes $\mathsf{ct} \leftarrow \mathsf{DSCS.Inp}(\mathsf{token}_{\mathsf{id}}, x)$, where $\mathsf{token}_{\mathsf{id}_i} \leftarrow \mathsf{DSCS.Auth}(\mathsf{id}, \mathsf{sk})$. Then send back ct.
- **Guess**: $\mathcal{B}$ receives adversary $\mathcal{A}$'s guess b'. And $\mathcal{B}$ outputs b' as his guess for the DFE experiment $\mathbf{Expt}^{\mathsf{DFE\text{-}func}}(1^\lambda)$.

We now argue that the adversary's view $(\{\tilde{P}_i\}_{i=1}^{n-1}, \{\mathsf{token}_{\mathsf{id}}\}, \{\mathsf{ct}\})$ in real execution is identical here as produced by reduction $\mathcal{B}$. This follows obviously, as $\{\tilde{P}_i\}_{i=1}^n$ is generated in the same way with the help of the challenge of DFE, $\mathsf{token}_{\mathsf{id}}$ is valid signature of identity id. The ciphertexts $\{\mathsf{ct}\}$ are generated in the same way in both executions. Therefore, a correct guess b' from adversary $\mathcal{A}$ is a correct guess for experiment $\mathbf{Expt}^{\mathsf{DFE\text{-}func}}(1^\lambda)$.

Theorem 6. *Let distributed functional encryption* DFE *satisfy ind-based data privacy (c.f. Definiton 9), then our DSCS construction described above satisfies input privacy in untrusted cloud setting (c.f. Definition 13).*

Proof. We describe a reduction $\mathcal{B}$ against the ind-based data privacy of underlying DFE scheme. If the adversary $\mathcal{A}$ can win the experiment $\mathbf{Expt}^{\mathsf{DFE\text{-}data}}(1^\lambda)$ as defined in Definition 9, then reduction $\mathcal{B}$ can also win the experiment $\mathbf{Expt}^{\mathsf{inp}}(1^\lambda)$ as defined in Definition 13. The description of reduction $\mathcal{B}$ is as follows:

- **Setup**; $\mathcal{B}$ interacts with the challenger of DFE to obtain $\mathsf{DFE.pp}$ and computes $(\Sigma.\mathsf{vk}, \Sigma.\mathsf{sk}) \leftarrow \Sigma.\mathsf{Setup}(1^\lambda)$. Then $\mathcal{B}$ invokes adversary $\mathcal{A}$ to get the program P. Next, $\mathcal{B}$ sends the augmented program P_{aug} (as described in Figure 3) to the challenger of DFE, and the challenger sends back $\{\mathsf{sk}_i^{P_{\mathsf{aug}}}\}_{i=1}^n$. Lastly, $\mathcal{B}$ sends $\{\tilde{P}_i\}_{i=1}^n$ to adversary, where $\{\tilde{P}_i\}_{i=1}^n$ are constructed as in Figure 4.
- **Authentication query phase I**: On input identity query id_i, $\mathcal{B}$ computes $\sigma_{\mathsf{id}} \leftarrow \Sigma.\mathsf{Sign}(\Sigma.\mathsf{sk}, \mathsf{id})$ and sends back $\mathsf{token}_{\mathsf{id}} = (\sigma_{\mathsf{id}}, \mathsf{DFE.pp})$ to adversary $\mathcal{A}$.
- **Challenge phase**: On input $(m_0, m_1, \mathsf{id}^*)$ from adversary, where $P(m_0) = P(m_1)$, $\mathcal{B}$ first computes $\sigma_{\mathsf{id}^*} \leftarrow \Sigma.\mathsf{Sign}(\Sigma.\mathsf{sk}, \mathsf{id}^*)$, and sends $(m_0||\sigma_{\mathsf{id}^*}, m_1||\sigma_{\mathsf{id}^*})$ to challenger, and receives challenge ciphertext ct^*.
- **Authentication query phase II**: Same as Authentication query phase I.
- **Guess**: $\mathcal{B}$ receives adversary $\mathcal{A}$'s guess b'. And $\mathcal{B}$ outputs b' as his guess for the DFE experiment $\mathbf{Expt}^{\mathsf{DFE\text{-}data}}(1^\lambda)$.

We now argue that the adversary's view $(\{\tilde{P}_i\}_{i=1}^n, \{\mathsf{token}_{\mathsf{id}}\}, \mathsf{ct}^*)$ in real execution is identical here as produced by reduction $\mathcal{B}$. This follows obviously, as $\{\tilde{P}_i\}_{i=1}^n$ is generated in the same way with the help of the challenge of DFE, $\mathsf{token}_{\mathsf{id}}$ is valid signature of identity id. For the challenge ciphertext ct^*, since $P'(m_0||\sigma_{\mathsf{id}^*}) = P'(m_1||\sigma_{\mathsf{id}^*})$, so the query $(m_0||\sigma_{\mathsf{id}^*}, m_1||\sigma_{\mathsf{id}^*})$ is a valid one. Therefore, a correct guess b' from adversary $\mathcal{A}$ is a correct guess for experiment $\mathbf{Expt}^{\mathsf{DFE\text{-}data}}(1^\lambda)$.

Theorem 7. *Let distributed functional encryption* DFE *satisfy sim-based function privacy (c.f. Definition 8) and* Σ *be an existential unforgeable signature scheme (c.f. Definition 1), then our DSCS construction described above satisfies untrusted client security (c.f. Definition 14).*

Proof. Based on the simulation algorithms of distributed functional encryption $(\mathsf{DFE}.\mathcal{S}_1, \mathsf{DFE}.\mathcal{S}_2, \mathsf{DFE}.\mathcal{S}_3)$ (c.f. Definition 8), we first describe the simulation algorithms $(\mathcal{S}_1, \mathcal{S}_2)$ and procedure decode as follows:

- $\mathcal{S}_1(1^\lambda, n)$: The simulation $\mathcal{S}_1$ first runs $\mathsf{DFE}.\mathcal{S}_1(1^\lambda, n)$ to obtain $\mathsf{DFE.pp}$, and $\Sigma.\mathsf{Setup}(1^\lambda)$ to obtain $(\Sigma.\mathsf{vk}, \Sigma.\mathsf{sk})$. Then $\mathcal{S}_1$ chooses a random program P' of the same size as P, and computes $\{\mathsf{sk}_i^{P'_{\mathsf{aug}}}\}_{i=1}^n \leftarrow \mathsf{DFE}.\mathcal{S}_2(P'_{\mathsf{aug}})$, where P'_{aug} is the augmented program of P' (c.f. Figure 3). Then compute $\{\tilde{P}'_i\}_{i=1}^n$ as described in Figure 4 and send back $\{\tilde{P}'_i\}_{i=1}^{n-1}$ to adversary.
- $\mathcal{S}_2(\mathsf{id}_i)$: On input identity id_i, $\mathcal{S}_2$ computes $\sigma_{\mathsf{id}_i} \leftarrow \Sigma.\mathsf{Sign}(\Sigma.\mathsf{sk}, \mathsf{id}_i)$. Send back $\mathsf{token}_{\mathsf{id}_i} = (\sigma_{\mathsf{id}_i}, \mathsf{DFE.pp})$.

– $\mathsf{decode}(\tilde{\mathsf{x}}_\mathsf{i}, \mathsf{id})$: On input ciphertext $\tilde{x}$ and identity id, it first computes $\{\mathsf{sk}_i^{\mathsf{Ind}}\}_{i=1}^n \leftarrow \mathsf{DFE}.\mathcal{S}_2(\mathsf{Ind})$, where Ind denotes the identity function, i.e. $\mathsf{Ind}(\mathsf{x}) = \mathsf{x}$, for any x. Then compute $x_i||\sigma_{\mathsf{id}} = \mathsf{DFE.Reconstruct}(\mathsf{DFE.pp}, \{s_j\}_{j=1}^n)$, where $s_j = \mathsf{DFE.PartDec}(\tilde{x}_i, \mathsf{sk}_j^{\mathsf{Ind}})$ for $j \in [n]$. Output $\perp$ if $\Sigma.\mathsf{Verify}(\Sigma.\mathsf{vk}, \mathsf{id}, \sigma_{\mathsf{id}}) = 0$. Otherwise, choose $n-1$ random values $\{y_{ij}\}_{j=1}^{n-1}$, then query $\mathsf{DFE}.\mathcal{S}_3^P$ on input $(x_i, \{y_{ij}\}_{j=1}^{n-1})$ to get y_{in}. Lastly, send back $\{y_{ij}\}_{j=1}^n$ back to adversary.

In the following, we show that adversary's view $(\{\tilde{P}\}_{i=1}^{n-1}, \{\mathsf{token}_{\mathsf{id}_i}\}_{i\in[\ell]}, \{\tilde{y}_{ij}\})$ in the two executions are indistinguishable. By the sim-based function privacy of DFE, $\{\tilde{P}\}_{i=1}^{n-1}$ in the two executions are indistinguishable. The tokens $(\{\mathsf{token}_{\mathsf{id}_i}\}_{i\in[\ell]}$ are computed identically in the two execution, thus they are indistinguishable in two executions. For query $(\tilde{x}_i, \mathsf{id})$, by the unforgeability of signature scheme Σ, if the underlying plaintext of $\tilde{x}_i$ does not contain a valid signature of id, then both executions output $\perp$ for query $(\tilde{x}_i, \mathsf{id})$. Otherwise, by the sim-based function privacy of DFE, the output $\{y_{ij}\}_{j=1}^n$ returned by simulation $\mathsf{DFE}.\mathcal{S}_3^P$ is indistinguishable from that in the real execution. Therefore, we reach the conclusion that our DSCS construction satisfies untrusted client security.

Additional properties. We remark that there are multiple additional properties we can consider for the application of hosting service in a cloud.

First, we can inherit the two extra properties of *verifiability* and *persistent memory* mentioned briefly in [BGMS15]. The *verifiability* requires that the validity of the results returned by the server can be checked, same as [BGMS15], we can rely on the technology of verifiable computation [GHRW14]. *Persistent memory* property is to consider the server can maintain a state for each client across different invocations. We can do the same as in [BGMS15] except the client needs to return the aggregated value back after each invocation.

Furthermore, we remark that as our construction is very simple and easy to extend, we can further support many other properties as well. Here we only list two examples. (i) Our current version and previous work [BGMS15] only puts the authentication on the client, once authorized, the client can query the cloud unlimitedly. If the service provider wants to post more fine grained control on each input data, we can further enable this by embedding the access structure into the function. (ii) We can further support *client anonymity* that the server (even all of them collude) cannot recognize whether two queries are from the same client. Currently, the client will submit the authentication token together with the data. It is easy to see that if we replace the token with an anonymous credential, we can have the additional anonymity.

5 Conclusion and Open Problems

We study the problem of public key functional encryption in a distributed model. Such a model enables us to circumvent the impossibility of function privacy in public key functional encryption. We formulated such a new primitive and gave a construction from functional secret sharing, which can be obtained from learning

with error assumption. We showcased the power of our new primitive by applying it to host services in multiple clouds.

One important observation of our distributed public key functional encryption is that achieving function privacy in this alternative model yields the power of virtual black-box obfuscation (essentially), which could potentially help circumvent other theoretic impossibilities in a distributed model. Another observation that may benefit application is that our construction is generic that upgrades any functional encryption. In some applications, we may only need a functional encryption for a special class of functions, which could have efficient constructions. This in turn yields potentially practical solutions for a class of important problems, such as encrypted search [SSW09], and copyright protection [KT15].

We leave the exploration of those interesting questions as open problems.

References

AAB+13. Shashank Agrawal, Shweta Agrawal, Saikrishna Badrinarayanan, Abishek Kumarasubramanian, Manoj Prabhakaran, and Amit Sahai. Functional encryption and property preserving encryption: New definitions and positive results. Cryptology ePrint Archive, Report 2013/744, 2013. http://eprint.iacr.org/2013/744.

AAB+15. Shashank Agrawal, Shweta Agrawal, Saikrishna Badrinarayanan, Abishek Kumarasubramanian, Manoj Prabhakaran, and Amit Sahai. On the practical security of inner product functional encryption. In Jonathan Katz, editor, *PKC 2015*, volume 9020 of *LNCS*, pages 777–798. Springer, Heidelberg, March / April 2015.

AGVW13. Shweta Agrawal, Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption: New perspectives and lower bounds. In Canetti and Garay [CG13], pages 500–518.

BGI+01. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 1–18. Springer, Heidelberg, August 2001.

BGI15. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Oswald and Fischlin [OF15], pages 337–367.

BGI16. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 1292–1303. ACM Press, October 2016.

BGMS15. Dan Boneh, Divya Gupta, Ilya Mironov, and Amit Sahai. Hosting services on an untrusted cloud. In Oswald and Fischlin [OF15], pages 404–436.

BRS13a. Dan Boneh, Ananth Raghunathan, and Gil Segev. Function-private identity-based encryption: Hiding the function in functional encryption. In Canetti and Garay [CG13], pages 461–478.

BRS13b. Dan Boneh, Ananth Raghunathan, and Gil Segev. Function-private subspace-membership encryption and its applications. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part I*, volume 8269 of *LNCS*, pages 255–275. Springer, Heidelberg, December 2013.

BS15. Zvika Brakerski and Gil Segev. Function-private functional encryption in the private-key setting. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part II*, volume 9015 of *LNCS*, pages 306–324. Springer, Heidelberg, March 2015.

BSW11. Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In Yuval Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 253–273. Springer, Heidelberg, March 2011.

CG13. Ran Canetti and Juan A. Garay, editors. *CRYPTO 2013, Part II*, volume 8043 of *LNCS*. Springer, Heidelberg, August 2013.

CGJS15. Nishanth Chandran, Vipul Goyal, Aayush Jain, and Amit Sahai. Functional encryption: Decentralised and delegatable. Cryptology ePrint Archive, Report 2015/1017, 2015. http://eprint.iacr.org/2015/1017.

DHRW16. Yevgeniy Dodis, Shai Halevi, Ron D. Rothblum, and Daniel Wichs. Spooky encryption and its applications. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 93–122. Springer, Heidelberg, August 2016.

GGG+14. Shafi Goldwasser, S Dov Gordon, Vipul Goyal, Abhishek Jain, Jonathan Katz, Feng-Hao Liu, Amit Sahai, Elaine Shi, and Hong-Sheng Zhou. Multi-input functional encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 578–602. Springer, 2014.

GHRW14. Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private RAM computation. In *55th FOCS*, pages 404–413. IEEE Computer Society Press, October 2014.

GVW12. Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption with bounded collusions via multi-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 162–179. Springer, Heidelberg, August 2012.

hul. Hulu's move into live television makes amazon a surprise winner. http://fortune.com/2017/08/15/hulu-live-tv-amazon-aws/.

KSW08. Jonathan Katz, Amit Sahai, and Brent Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 146–162. Springer, Heidelberg, April 2008.

KT15. Aggelos Kiayias and Qiang Tang. Traitor deterring schemes: Using bitcoin as collateral for digital content. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 231–242. ACM Press, October 2015.

KZ16. Ilan Komargodski and Mark Zhandry. Cutting-edge cryptography through the lens of secret sharing. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part II*, volume 9563 of *LNCS*, pages 449–479. Springer, Heidelberg, January 2016.

net. Completing the netflix cloud migration. https://media.netflix.com/en/company-blog/completing-the-netflix-cloud-migration.

OF15. Elisabeth Oswald and Marc Fischlin, editors. *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*. Springer, Heidelberg, April 2015.

O'N10. Adam O'Neill. Definitional issues in functional encryption. *IACR Cryptology ePrint Archive*, 2010:556, 2010.

pok. Bringing pokmon go to life on google cloud. https://cloudplatform.googleblog.com/2016/09/bringing-Pokemon-GO-to-life-on-Google-Cloud.html.

Reg05. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.

SSW09. Emily Shen, Elaine Shi, and Brent Waters. Predicate privacy in encryption systems. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 457–473. Springer, Heidelberg, March 2009.