Hosting Services on an Untrusted Cloud

Dan Boneh[1⋆], Divya Gupta[2⋆⋆], Ilya Mironov[3⋆⋆⋆], and Amit Sahai[2⋆⋆]

[1] Stanford University, dabo@cs.stanford.edu
[2] UCLA and Center for Encrypted Functionalities, {divyag, asahai}@cs.ucla.edu
[3] Google, mironov@gmail.com

Abstract. We consider a scenario where a service provider has created a software service S and desires to outsource the execution of this service to an untrusted cloud. The software service contains secrets that the provider would like to keep hidden from the cloud. For example, the software might contain a secret database, and the service could allow users to make queries to different slices of this database depending on the user's identity.

This setting presents significant challenges not present in previous works on outsourcing or secure computation. Because secrets in the software itself must be protected against an adversary that has full control over the cloud that is executing this software, our notion implies indistinguishability obfuscation. Furthermore, we seek to protect knowledge of the software S to the maximum extent possible even if the cloud can collude with several corrupted users.

In this work, we provide the first formalizations of security for this setting, yielding our definition of a *secure cloud service scheme*. We provide constructions of secure cloud service schemes assuming indistinguishability obfuscation, one-way functions, and non-interactive zero-knowledge proofs.

At the heart of our paper are novel techniques to allow parties to simultaneously authenticate and securely communicate with an obfuscated program, while hiding this authentication and communication from the entity in possession of the obfuscated program.

1 Introduction

Consider a service provider that has created some software service S that he wants to make accessible to a collection of users. However, the service provider is computationally weak and wants to outsource the computation of S to an untrusted cloud. Nevertheless, the software is greatly valuable and he does not want the cloud to learn

⋆ Supported by NSF and DARPA.

⋆⋆ Research supported in part from a DARPA/ONR PROCEED award, NSF Frontier Award 1413955, NSF grants 1228984, 1136174, 1118096, and 1065276, a Xerox Faculty Research Award, a Google Faculty Research Award, an equipment grant from Intel, and an Okawa Foundation Research Grant. This material is based upon work supported by the Defense Advanced Research Projects Agency through the U.S. Office of Naval Research under Contract N00014-11-1-0389. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense, the NSF, or the U.S. Government.

⋆⋆⋆ Work done in Microsoft Research.

what secrets are embedded in the software S. There are many concrete examples of such a scenario; for example, the software could contain a secret database, and the service could allow users to make queries to different slices of this database depending on the user's identity.

At first glance, such a scenario seems like a perfect application of obfuscation and can be thought to be solved as follows: The provider could obfuscate the software $O(S)$ and send this directly to the cloud. Now, the cloud could receive an input (id, x) directly from a user with identity id, and respond with the computed output $O(S)(\mathsf{id}, x) = S(\mathsf{id}, x)$. Secure obfuscation would ensure that the cloud would never learn the secrets built inside the software, *except* for what is efficiently revealed by the input-output behavior of the software. But this approach does not provide any privacy to the users. In such a setting, the cloud will be able to learn the inputs x and the outputs $S(\mathsf{id}, x)$ of the multitude of users which use this service. This is clearly undesirable in most applications. Worse still, the cloud will be able to query the software on arbitrary inputs and identities of its choice. In our scheme, we want to guarantee input and output privacy for the users. Moreover, we want that only a user who pays for and subscribes to the service is able to access the functionality that the service provides for that particular user.

Ideally, we would like that, first, a user with identity id performs some simple one-time set-up interaction with the service provider to obtain a key K_{id}. This key K_{id} would also serve as authentication information for the user. Later, in order to run the software on input x of his choice, he would encrypt x to $Enc_{K_{\mathsf{id}}}(x)$ and send it to the cloud. The cloud would run the software to obtain an encryption of $S(\mathsf{id}, x)$, which is sent back to the user while still in encrypted form. Finally, the user can decrypt in order to obtain its output.

Let us step back and specify a bit more precisely the security properties we desire from such a secure cloud service.

1. **Security against malicious cloud.** In our setting, if the cloud is the only malicious party, then we require that it cannot learn anything about the nature of the computation except a bound on the running time. In particular, it learns nothing about the code of the software or the input/output of users.
2. **Security against malicious clients.** If a collection of users is malicious, they cannot learn anything beyond what is learnable via specific input/output that the malicious users see. Furthermore, if a client is not authenticated by the service provider, it cannot learn anything at all.
3. **Security against a malicious cloud and clients.** Moreover, even when a malicious cloud colludes with a collection of malicious users, the adversary cannot learn anything beyond the functionality provided to the malicious users. That is, the adversary does not learn anything about the input/output of the honest users or the slice of service provided to them. More precisely, consider two software services S and S' which are functionally equivalent when restricted to corrupt users. Then the adversary cannot distinguish between the instantiations of the scheme with S and S'.
4. **Efficiency.** Since the service provider and the users are computationally weak parties, we want to make their online computation highly efficient. The interaction in the set-up phase between the provider and a user should be independent of the

complexity of the service being provided. For the provider, only its one-time encoding of the software service should depend polynomially on the complexity of the software. The work of the client in encrypting his inputs should only depend polynomially on the size of his inputs and a security parameter. And finally, the running time of the encoded software on the cloud should be bounded by a fixed polynomial of the running time of the software.

Note that since the scheme is for the benefit of the service provider, who could choose to provide whatever service it desires, we assume that the service provider itself is uncompromised.

We call a scheme that satisfies the above listed properties, a *Secure Cloud Service Scheme* (SCSS). In this work, we provide the first construction of a secure cloud service scheme, based on indistinguishability obfuscation, one-way functions, and non-interactive zero-knowledge proofs. At the heart of our paper are novel techniques to allow parties to simultaneously authenticate and securely communicate with an obfuscated program, while hiding this authentication and communication from the entity in possession of the obfuscated program.

Relationships to other models. At first glance, the setting we consider may seem similar to notions considered in earlier works. However, as we describe below, there are substantial gaps between these notions and our setting. As an initial observation, we note that a secure cloud service scheme is fundamentally about protecting secrets within software run by a single entity (the cloud), and therefore is intimately tied to obfuscation. Indeed, our definition of a secure cloud service scheme immediately implies indistinguishability obfuscation. Thus, our notion is separated from notions that do not imply obfuscation. We now elaborate further, comparing our setting to two prominent previously considered notions.

- **Delegation of Computation.** A widely studied topic in cryptography is secure delegation or outsourcing of computation (e.g., [14,11,7,16]), where a *single user* wishes to delegate a computation to the cloud. The most significant difference between delegation and our scheme is that in delegation the role of the provider and the user is combined into a single entity. In contrast, in our setting the entity that decides the function S is the provider, and this entity is completely separate from the entities (users) that receive outputs. Indeed, a user should learn nothing about the function being computed by the cloud beyond what the specific input/output pairs that the user sees. Moreover, the vast majority of delegation notions in literature do not require any kind of obfuscation.
  Furthermore, we consider a setting where multiple unique users have access to a different slice of service on the cloud (based on their identities), whereas in standard formulations of delegation, only one computation is outsourced from client to the cloud. There is a recent work on delegation that does consider multiple users: the work of [8] on outsourcing RAM computations goes beyond the standard setting of delegation to consider a multi-user setting. But as pointed out by the authors themselves, in this setting, the cloud can learn arbitrary information about the description of the software. Their notion of privacy only guarantees that the cloud

learns nothing about the inputs and outputs of the users, but not about the nature of the computation – which is the focus of our work. Moreover, in their setting, no security is promised in the case of a collusion between a malicious cloud and a malicious client. The primary technical contributions of our work revolve around guaranteeing security in this challenging setting.

- **Multi-Input Functional Encryption (MIFE).** Recently, the work of [10] introduced the extremely general notion of multi-input functional encryption (MIFE), whose setting can capture a vast range of scenarios. Nevertheless, MIFE does not directly apply to our scenario: In our setting, there are an unbounded number of possible clients, each of which gets a unique *encryption* key that is used to prepare its input for the cloud. MIFE has been defined with respect to a fixed number of possible encryption keys [10], but even if it were extended to an unbounded number of encryption keys, each function evaluation key in an MIFE would necessarily be bound to a fixed number of encryption keys. This would lead to a combinatorial explosion of exponentially many function evaluation keys needed for the cloud.
  Alternatively, one could try to build a secure cloud service scheme by "jury-rigging" MIFE to nevertheless apply to our scenario. Fundamentally, because MIFE does imply indistinguishability obfuscation [10], this must be possible. But, as far we know, the only way to use MIFE to build a secure cloud service scheme is by essentially carrying out our entire construction, but replacing our use of indistinguishability obfuscation with calls to MIFE. At a very high level, the key challenges in applying MIFE to our setting arise from the IND-definition of MIFE security [10], which largely mirrors the definition of indistinguishability obfuscation security. We elaborate on these challenges below, when we discuss our techniques.

1.1 Our Results.

In this work, we formalize the notion of secure cloud service scheme (Section 3) and give the first scheme which achieves this notion. In our formal notion, we consider potential collusions involving the cloud and up to k corrupt users, where k is a bound fixed in advance. (Note again that even with a single corrupt user, our notion implies indistinguishability obfuscation.) We then give a protocol which implements a secure cloud service scheme. More formally,

Theorem 1. *Assuming the existence of indistinguishability obfuscation, statistically simulation-sound non-interactive zero-knowledge proof systems and one-way functions, for any bound k on the number of corrupt users that is polynomially related to the security parameter, there exists a secure cloud service scheme.*

Note that we only require a bound on the number of corrupt clients, and not on the total number of users in the system. Our scheme provides an exponential space of possible identities for users. We note that the need to bound the number of corrupt users when using indistinguishability obfuscation is related to several other such bounds that are needed in other applications of indistinguishability obfuscation, such as the number of adversarial ciphertexts in functional encryption [6] and multi-input functional encryption [10] schemes. We consider the removal of such a bound using indistinguishability obfuscation to be a major open problem posed by our work.

Furthermore, we also consider the case when the software service takes two inputs: one from the user and other from the cloud. We call this setting a secure cloud service scheme with cloud inputs. This setting presents an interesting technical challenge because it opens up exponential number of possible functions that could have been provided to a client. We resolve this issue using a technically interesting sequence of 2^ℓ hybrids, where ℓ is the length of the cloud's input (see Our Techniques below for further details). To prove security, we need to assume sub-exponential hardness of indistinguishability obfuscation. More formally, we have the following result.

Theorem 2. *Assuming the existence of sub-exponentially hard indistinguishability obfuscation, statistically simulation-sound non-interactive zero-knowledge proof systems and sub-exponentially hard one-way functions, for any bound k on the number of corrupt users that is polynomially related to the security parameter, there exists a secure cloud service scheme with cloud inputs.*

1.2 Our Techniques.

Since a secure cloud service scheme implies indistinguishability obfuscation ($i\mathcal{O}$), let us begin by considering how we may apply obfuscation to solve our problem, and use this to identify the technical obstacles that we will face.

The central goal of a secure cloud service scheme is to hide the nature of the service software S from the cloud. Thus, we would certainly use $i\mathcal{O}$ to obfuscate the software S before providing it to the cloud. However, as we have already mentioned, this is not enough, as we also want to provide privacy to honest users. Our scheme must also give a user the ability to encrypt its input x in such a way that the cloud cannot decrypt it, but the obfuscated software can. After choosing a public key PK and decryption key SK for a public-key encryption scheme, we could provide PK to the user, and build SK into the obfuscated software to decrypt inputs. Finally, each user should obtain its output in encrypted form, so that the cloud cannot decrypt it. In particular, each user can choose a secret key K_{id}, and then to issue a query, it can create the ciphertext $c = Enc_{PK}(x, K_{\mathsf{id}})$. Thus, we need to build a program $\hat{S}$ that does the following: It takes as input the user id id and a ciphertext c. It then decrypts c using SK to yield (x, K_{id}). It then computes the output $y = S(\mathsf{id}, x)$. Finally, it outputs the ciphertext $d = Enc(K_{\mathsf{id}}, y)$. The user can decrypt this to obtain y. The cloud should obtain an obfuscated version of this software $\hat{S}$.

At first glance, it may appear that this scheme would already be secure, at least if given an "ideal obfuscation" akin to Virtual Black-Box obfuscation [1]. However, this is not true. In particular, there is a malleability attack that arises: Consider the scenario where the cloud can malleate the ciphertext sent by the user, which contains his input x and key K_{id}, to an encryption of x and K^*, where K^* is maliciously chosen by the cloud. If this were possible, the cloud could use its knowledge of K^* to decrypt the output $d = Enc(K_{\mathsf{id}}, y)$ produced by the obfuscated version of $\hat{S}$. But this is not all. Another problem we have not yet handled is authentication: a malicious user could pretend to have a different identity id than the one that it is actually given, thereby obtaining outputs from S that it is not allowed to access. We must address both the

malleability concern and the authentication concern, but also do this in a way that works with indistinguishability obfuscation, not just an ideal obfuscation.

Indeed, once we constrain ourselves to only using indistinguishability obfuscation, additional concerns arise. Here, we will describe the two most prominent issues, and describe how we deal with them.

Recall that our security notion requires that if an adversary corrupts the cloud and a user id^*, then the view of the adversary is indistinguishable for any two softwares S and S' such that $S(\mathsf{id}^*, x) = S'(\mathsf{id}^*, x)$ for all possible inputs x. However, S and S' could differ completely on inputs for several other identities id. Ideally, in our proof, we would like to use the security of $i\mathcal{O}$ while making the change from S to S' in the obfuscated program. In order to use the security of $i\mathcal{O}$, the two programs being obfuscated must be equivalent for all inputs, and not just the inputs of the malicious client with identity id^*. However, we are given no such guarantee for S and S'. So in our proof of security, we have to construct a hybrid (indistinguishable from real execution on S) in which S can *only* be invoked for the malicious client identity id^*. Since we have functional equivalence for this client, we will then be able to make the switch from $\hat{S}$ to $\hat{S}'$ by security of $i\mathcal{O}$. We stress that the requirement to make this switch is that there does not exist any input to the obfuscated program which give different outputs for $\hat{S}$ and $\hat{S}'$. It does not suffice to ensure that a differing input cannot be computed efficiently. To achieve this, in this hybrid, we must ensure that there does not exist any valid authentication for all the honest users. Thus, since no honest user can actually get a useful output from $\hat{S}$ or $\hat{S}'$, they will be functionally equivalent. In contrast, all the malicious users should still be able to get authenticated and obtain outputs from the cloud; otherwise the adversary would notice that something is wrong. We achieve this using a carefully designed authentication scheme that we describe next.

At a high level, we require the following: Let k be the bound on the number of malicious clients. The authentication scheme should be such that in the "fake mode" it is possible to authenticate the k corrupt user identities and there does not exist (even information-theoretically) any valid authentication for any other identity. We achieve this notion by leveraging k-cover-free sets of [4,13] where there are a super-polynomial number of sets over a polynomial sized universe such that the union of *any* k sets does not cover any other set. We use these sets along with length doubling PRGs to build our authentication scheme.

Another problem that arises with the use of indistinguishability obfuscation concerns how outputs are encrypted within $\hat{S}$. The output of the obfuscated program is a ciphertext which encrypts the actual output of the software. We are guaranteed that the outputs of S and S' are identical for the corrupt clients, but we still need to ensure that the corresponding encryptions are also identical (in order to apply the security of $i\mathcal{O}$.) We ensure this by using an encryption scheme which satisfies the following: If two obfuscated programs using S and S', respectively, are given a ciphertext as input, then if S and S' produce the same output, then the obfuscated programs will produce *identical* encryptions as output. In particular, our scheme works as follows: the user sends a pseudo-random function (PRF) key K_{id} and the program outputs $y = \mathsf{PRF}(K_{\mathsf{id}}, r) \oplus S(x, \mathsf{id})$, where the r value is computed using another PRF applied to the ciphertext c itself. Thus we ensure that for identical ciphertexts as inputs, both

programs produce the same r, and hence the same y. This method allows us to switch S to S', but the new challenge then becomes how to argue the *security* of this encryption scheme. To accomplish this, we use the punctured programming paradigm of [18] to build a careful sequence of hybrids using punctured PRF keys to argue security.

We need several other technical ideas to make the security proof work. Please see our protocol in Section 4 and proof in Section 4.1 for details.

When considering the case where the cloud can also provide an input to the computation, the analysis becomes significantly more complex because of a new attack: The cloud can take an input from an honest party, and then try to vary the cloud's own input, and observe the impact this has on the output of the computation. Recall that in our proof of security, in one hybrid, we will need to "cut off" honest parties from the computation – but we need to do this in a way that is indistinguishable from the cloud's point of view. But an honest party that has been cut off will no longer have an output that can depend on the cloud's input. If the cloud can detect this, the proof of security fails. In order to deal with this, we must change the way that our encryption of the output works, in order to include the cloud input in the computation of the r value. But once we do this, the punctured programming methods of [18] become problematic. To deal with this issue, we create a sequence of exponentially many hybrids, where we puncture out exactly one possible cloud input at a time. This lets us avoid a situation where the direct punctured programming approach would have required an exponential amount of puncturing, which would cause the programs being obfuscated to blow up to an exponential size.

2 Prelims

Let λ be the security parameter. Below, we describe the primitives used in our scheme.

2.1 Public Key Encryption Scheme

A public key encryption scheme pke over a message space $\mathcal{M} = \mathcal{M}_\lambda$ consists of three algorithms $\mathsf{PKGen}, \mathsf{PKEnc}, \mathsf{PKDec}$. The algorithm PKGen takes security parameter 1^λ and outputs the public key pk and secret key sk. The algorithm PKEnc takes public key pk and a message $\mu \in \mathcal{M}$ as input and outputs the ciphertext c that encrypts μ. The algorithm PKDec takes the secret key sk and ciphertext c and outputs a message μ.

A public key encryption scheme pke is said to be correct if for all messages $\mu \in \mathcal{M}$:

$$\Pr[(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKGen}(1^\lambda); \mathsf{PKDec}(\mathsf{sk}, \mathsf{PKEnc}(\mathsf{pk}, \mu; u)) \neq \mu] \leqslant \mathsf{negl}(\lambda)$$

A public key encryption scheme pke is said to be $\mathsf{IND\text{-}CPA}$ secure if for all PPT adversaries $\mathcal{A}$ following holds:

$$\Pr\left[b = b' \,\middle|\, \begin{array}{l}(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKGen}(1^\lambda); (\mu_0, \mu_1, \mathsf{st}) \leftarrow \mathcal{A}(1^\lambda, \mathsf{pk}); \\ b \xleftarrow{\$} \{0,1\}; c = \mathsf{PKEnc}(\mathsf{pk}, \mu_b; u); b' \leftarrow \mathcal{A}(c, \mathsf{st})\end{array}\right] \leqslant \frac{1}{2} + \mathsf{negl}(\lambda)$$

2.2 Indistinguishability Obfuscation

The definition below is from [6]; there it is called a "family-indistinguishable obfuscator", however they show that this notion follows immediately from their standard definition of indistinguishability obfuscator using a non-uniform argument.

Definition 1 (Indistinguishability Obfuscator ($i\mathcal{O}$)). *A uniform PPT machine $i\mathcal{O}$ is called an* indistinguishability obfuscator *for acircuit class $\{\mathcal{C}_\lambda\}$ if the following conditions are satisfied:*

- *For all security parameters $\lambda \in \mathbb{N}$, for all $C \in \mathcal{C}_\lambda$, for all inputs x, we have that*

$$\Pr[C'(x) = C(x) : C' \leftarrow i\mathcal{O}(\lambda, C)] = 1$$

- *For any (not necessarily uniform) PPT adversaries $Samp$, D, there exists a negligible function α such that the following holds: if $\Pr[\forall x, C_0(x) = C_1(x) : (C_0, C_1, \sigma) \leftarrow Samp(1^\lambda)] > 1 - \alpha(\lambda)$, then we have:*

$$\Big|\Pr\left[D(\sigma, i\mathcal{O}(\lambda, C_0)) = 1 : (C_0, C_1, \sigma) \leftarrow Samp(1^\lambda)\right] - \Pr\left[D(\sigma, i\mathcal{O}(\lambda, C_1)) = 1 : (C_0, C_1, \sigma) \leftarrow Samp(1^\lambda)\right]\Big| \leq \alpha(\lambda)$$

In this paper, we will make use of such indistinguishability obfuscators for all polynomial-size circuits:

Definition 2 (Indistinguishability Obfuscator for $P/poly$). *A uniform PPT machine $i\mathcal{O}$ is called an* indistinguishability obfuscator *for $P/poly$ if the following holds: Let $\mathcal{C}_\lambda$ be the class of circuits of size at most λ. Then $i\mathcal{O}$ is an indistinguishability obfuscator for the class $\{\mathcal{C}_\lambda\}$.*

Such indistinguishability obfuscators for all polynomial-size circuits were constructed under novel algebraic hardness assumptions in [6].

2.3 Puncturable PRF

Puncturable PRFs are a simple types of constrained PRFs [2,12,3]. These are PRFs that can be defined on all bit strings of a certain length, except for any polynomial-size set of inputs. Following definition has been taken verbatim from [18].

Definition 3. *A puncturable family of PRFs F is given by a triple of turing machines $\mathsf{PRFKey_F}, \mathsf{Puncture_F}, \mathsf{Eval_F}$, and a pair of computable functions $n(\cdot)$ and $m(\cdot)$, satisfying the following conditions.*

- **Functionality preserved under puncturing.** *For every PPT adversary $\mathcal{A}$ such that $\mathcal{A}(1^\lambda)$ outputs a set $S \subseteq \{0,1\}^{n(\lambda)}$, then for all $x \in \{0,1\}^{n(\lambda)}$ where $x \notin S$, we have that:*

$$\Pr\left[\mathsf{Eval_F}(K, x) = \mathsf{Eval_F}(K_S, x) : K \leftarrow \mathsf{PRFKey_F}(1^\lambda), K_S = \mathsf{Puncture_F}(K, S)\right] = 1$$

- **Pseudorandom at punctured points.** *For every PPT adversary* $(\mathcal{A}_1, \mathcal{A}_2)$ *such that* $\mathcal{A}_1(1^\lambda)$ *outputs a set* $S \subseteq \{0,1\}^{n(\lambda)}$ *and state* st*, consider an experiment where* $K \leftarrow \mathsf{PRFKey}_\mathsf{F}(1^\lambda)$ *and* $K_S = \mathsf{Puncture}_\mathsf{F}(K, S)$*. Then we have*

$$\left| \Pr\left[\mathcal{A}_2(\sigma, K_S, S, \mathsf{Eval}_\mathsf{F}(K, S)) = 1\right] - \Pr\left[\mathcal{A}_2(\mathsf{st}, K_S, S, U_{m(\lambda)\cdot|S|}) = 1\right] \right| = \mathsf{negl}(\lambda)$$

where $\mathsf{Eval}_\mathsf{F}(K, S)$ *denotes the concatenation of* $\mathsf{Eval}_\mathsf{F}(K, x_1)), \ldots, \mathsf{Eval}_\mathsf{F}(K, x_k))$ *where* $S = \{x_1, \ldots, x_k\}$ *is the enumeration of the elements of* S *in lexicographic order,* $\mathsf{negl}(\cdot)$ *is a negligible function, and* U_ℓ *denotes the uniform distribution over* ℓ *bits.*

For ease of notation, we write $\mathsf{PRF}(K, x)$ *to represent* $\mathsf{Eval}_\mathsf{F}(K, x)$*. We also represent the punctured key* $\mathsf{Puncture}_\mathsf{F}(K, S)$ *by* $K(S)$*.*

The GGM tree-based construction of PRFs [9] from one-way functions are easily seen to yield puncturable PRFs, as recently observed by [2,12,3]. Thus we have:

Theorem 3. *[9,2,12,3] If one-way functions exist, then for all efficiently computable functions* $n(\lambda)$ *and* $m(\lambda)$*, there exists a puncturable PRF family that maps* $n(\lambda)$ *bits to* $m(\lambda)$ *bits.*

2.4 Statistical Simulation-Sound Non-Interactive Zero-Knowledge

This primitive was introduced in [6] and was constructed from standard NIZKs using a commitment scheme. A statistically simulation-sound NIZK proof system for a relation R consists of three algorithms: $\mathsf{NIZKSetup}$, $\mathsf{NIZKProve}$, and $\mathsf{NIZKVerify}$ and satisfies the following properties.

Perfect completeness. An honest prover holding a valid witness can always convince an honest verifier. Formally,

$$\Pr\left[\mathsf{NIZKVerify}(\mathsf{crs}, x, \pi) = 1 \,\middle|\, \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{NIZKSetup}(1^\lambda); (x, w) \in R; \\ \pi \leftarrow \mathsf{NIZKProve}(\mathsf{crs}, x, w) \end{array}\right] = 1$$

Statistical soundness. A proof system is sound if it is infeasible to convince an honest verifier when the statement is false. Formally, for all (even unbounded) adversaries $\mathcal{A}$,

$$\Pr\left[\mathsf{NIZKVerify}(\mathsf{crs}, x, \pi) = 1 \,\middle|\, \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{NIZKSetup}(1^\lambda); \\ (x, \pi) \leftarrow \mathcal{A}(\mathsf{crs}); x \notin L \end{array}\right] \leqslant \mathsf{negl}(\lambda)$$

Computational zero-knowledge [5]. A proof system is zero-knowledge if a proof does not reveal anything beyond the validity of the statement. In particular, it does not reveal anything about the witness used by an honest prover. We say that a non-interactive proof system is zero-knowledge if there exists a PPT simulator $\mathcal{S} = (\mathsf{S}_1, \mathsf{S}_2)$ such that

S_1 outputs a simulated CRS and a trapdoor τ for proving x and S_2 produces a simulated proof which is indistinguishable from an honest proof. Formally, for all PPT adversaries $\mathcal{A}$, for all $x \in L$ such w is witness, following holds.

$$\Pr\left[\mathcal{A}(\mathsf{crs}, x, \pi) = 1 \,\middle|\, \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{NIZKSetup}(1^\lambda); \\ \pi \leftarrow \mathsf{NIZKProve}(\mathsf{crs}, x, w) \end{array}\right] \approx$$
$$\Pr\left[\mathcal{A}(\mathsf{crs}, x, \pi) = 1 \,\middle|\, \begin{array}{l} (\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, x); \\ \pi \leftarrow \mathsf{S}_2(\mathsf{crs}, \tau, x) \end{array}\right]$$

Statistical simulation-soundness. A proof system is said to be statistical simulation sound if is infeasible to convince an honest verifier when the statement is false even when the adversary is provided with a simulated proof (of a possibly false statement.) Formally, for all (even unbounded) adversaries $\mathcal{A}$, for all statements x, following holds.

$$\Pr\left[\mathsf{NIZKVerify}(\mathsf{crs}, x', \pi') = 1 \,\middle|\, \begin{array}{l} (\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, x); \pi \leftarrow \mathsf{S}_2(\mathsf{crs}, \tau, x); \\ (x', \pi') \leftarrow \mathcal{A}(\mathsf{crs}, x, \pi); x' \notin L \end{array}\right] \leqslant \mathsf{negl}(\lambda)$$

2.5 Cover-Free Set Systems and Authentication Schemes.

The authentication system we will use in our scheme will crucially use the notion of a cover-free set systems. Such systems were considered and build in [4,13]. Our definitions and constructions are inspired by those in [13].

Definition 4 (k**-cover-free set system).** *Let U be the universe and $n{:=}|U|$. A family of sets $\mathcal{T} = \{T_1, \ldots, T_N\}$, where each $T_i \subseteq U$ is a k-cover-free set family if for all $T_1, \ldots, T_k \in \mathcal{T}$ and $T \in \mathcal{T}$ such that $T \neq T_i$ for all $i \in [k]$ following holds: $T \setminus \cup_{i \in [k]} T_i \neq \emptyset$.*

[13] constructed such a set system using Reed-Solomon codes. We define these next. Let $\mathbb{F}_q$ be a finite field of size q. Let $F_{q,k}$ denote the set of polynomials on $\mathbb{F}_q$ of degree at most k.

Definition 5 (Reed-Solomon code). *Let $x_1, \ldots, x_n \in \mathbb{F}_q$ be distinct and $k > 0$. The $(n,k)_q$-Reed-Solomon code is given by the subspace $\{\langle f(x_1), \ldots, f(x_n)\rangle \mid f \in F_{q,k}\}$.*

It is well-known that any two distinct polynomials of degree at most k can agree on at most k points.

Construction of k-cover-free sets. Let $\mathbb{F}_q = \{x_1, \ldots, x_q\}$ be a finite field of size q. We will set q in terms of security parameter λ and k later. Let universe be $U = \mathbb{F}_q \times \mathbb{F}_q$. Define $d{:=}\frac{q-1}{k}$. The k-cover-free set system is as follows: $\mathcal{T} = \{T_f \mid f \in F_{q,d}\}$, where $T_f = \{\langle x_1, f(x_1)\rangle, \ldots \langle x_q, f(x_q)\rangle\} \subset U$.

Note that $N{:=}|\mathcal{T}| = q^{d+1}$. For example, by putting $q = k \log \lambda$, we get $N = \lambda^{\omega(1)}$. In our scheme, we will set $q = k\lambda$ to obtain $N \geqslant 2^\lambda$.

Claim. The set system $\mathcal{T}$ is k-cover-free.

Proof. Note that each set T_f is a $(q,d)_q$-Reed-Solomon code. As pointed out earlier, any two distinct Reed-Solomon codes of degree d can agree on at most d points. Hence, $|T_i \cap T_j| \leqslant d$ for all $T_i, T_j \in \mathcal{T}$. Using this we get, for any $T, T_1, \ldots, T_k \in \mathcal{T}$ such that $T \neq T_i$ for all $i \in [k]$,

$$\left|T \setminus \cup_{i\in[k]} T_i\right| \geqslant q - kd = 1$$

Authentication Scheme based on k-cover-free sets. At a high level, there is an honest authenticator $\mathcal{H}$ who posses a secret authentication key ask and announces the public verification key avk. There are (possibly unbounded) polynomial number of users and each user has an identity. We want to design a primitive such that $\mathcal{H}$ can authenticate a user depending on his identity. The authentication t_{id} can be publicly verified using the public verification key.

Let $\mathsf{PRG} : Y \to Z$ be a pseudorandom generator. with $Y = \{0,1\}^\lambda$ and $Z = \{0,1\}^{2\lambda}$. Let the number of corrupted users be bounded by k. Let $\mathbb{F}_q = \{x_1, \ldots, x_q\}$ be a finite field with $q \geqslant k\lambda$. In the scheme below we will use the k-cover-free sets described above. Let $d = \frac{q-1}{k}$. Let $\mathcal{T}$ be the family of cover-free sets over the universe $\mathbb{F}_q^2$ such that each set is indexed by an element in $\mathbb{F}_q^{d+1}$.

The authentication schemes has three algorithms $\mathsf{AuthGen}$, $\mathsf{AuthProve}$ and $\mathsf{Authverify}$ described as follows.

- Setup: The algorithm $\mathsf{AuthGen}(1^\lambda)$ works follows: For all $i, j \in [q]$, picks $s_{ij} \xleftarrow{\$} Y$. Set $\mathsf{ask} = \{s_{ij}\}_{i,j\in[q]}$ and $\mathsf{avk} = \{\mathsf{PRG}(s_{ij})\}_{i,j\in[q]} = \{z_{ij}\}_{i,j\in[q]}$. Returns $(\mathsf{avk}, \mathsf{ask})$. The keys will also contain the set-system $\mathcal{T}$. We assume this implicitly, and omit writing it.
- Authentication: The algorithm $\mathsf{AuthProve}(\mathsf{ask}, \mathsf{id})$ works as follows for a user id. Interpret id as a polynomial in $F_{q,d}$ for $d = \frac{q-1}{k}$, i.e., $\mathsf{id} \in \mathbb{F}_q^{d+1}$. Let T_{id} be the corresponding set in $\mathcal{T}$. For all $i \in [q]$, if $\mathsf{id}(x_i) = x_j$ for some $j \in [q]$, then set $y_i = s_{ij}$. It returns $t_{\mathsf{id}} = \{y_i\}$ for all $i \in [q]$.
- Verification: The algorithm $\mathsf{Authverify}(\mathsf{avk}, \mathsf{id}, t_{\mathsf{id}})$ works as follows: Interpret id as a polynomial in $F_{q,d}$ for $d = \frac{q-1}{k}$, i.e., $\mathsf{id} \in \mathbb{F}_q^{d+1}$. Let T_{id} be the corresponding set in $\mathcal{T}$. Let $t_{\mathsf{id}} = \{y_1, \ldots, y_q\}$. For all $i \in [q]$, if $\mathsf{id}(x_i) = x_j$ for some $j \in [q]$, then check whether $\mathsf{PRG}(y_i) = z_{ij}$. Accept t_{id} if and only if all the checks pass.

The security properties this scheme satisfies are as follows:

Correctness. Honestly generated authentications always verify under the verification key. Formally, for any id, following holds.

$$\Pr[\mathsf{Authverify}(\mathsf{avk}, \mathsf{id}, t_{\mathsf{id}}) = 1 \mid (\mathsf{avk}, \mathsf{ask}) \leftarrow \mathsf{AuthGen}(1^\lambda); t_{\mathsf{id}} \leftarrow \mathsf{AuthProve}(\mathsf{ask}, \mathsf{id})] = 1$$

k-Unforgeability. Given authentication of any k users $\{\mathsf{id}_1, \ldots, \mathsf{id}_k\}$, for any PPT adversary $\mathcal{A}$, it is infeasible to compute t_{id^*} for any $\mathsf{id}^* \neq \mathsf{id}_i$ for all $i \in [k]$. More formally, we have that for PPT adversary $\mathcal{A}$ and any set of at most k corrupt ids $\mathcal{I}$ such that $|\mathcal{I}| \leqslant k$, following holds.

$$\Pr\left[\begin{array}{c}\mathsf{id}^* \notin \mathcal{I} \wedge \\ \mathsf{Authverify}(\mathsf{avk}, \mathsf{id}^*, t_{\mathsf{id}^*}) = 1\end{array}\middle|\begin{array}{l}(\mathsf{ask}, \mathsf{avk}) \leftarrow \mathsf{AuthGen}(1^\lambda); \\ t_{\mathsf{id}_i} \leftarrow \mathsf{AuthProve}(\mathsf{ask}, \mathsf{id}_i) \forall \mathsf{id}_i \in \mathcal{I}; \\ (\mathsf{id}^*, t_{\mathsf{id}^*}) \leftarrow \mathcal{A}(\mathsf{avk}, \{\mathsf{id}_i, t_{\mathsf{id}_i}\}_{\mathsf{id}_i \in \mathcal{I}})\end{array}\right] \leqslant \mathsf{negl}(\lambda)$$

Our scheme satisfies unforgeability as follows: Since $\mathcal{T}$ is a k-cover-free set system, there exists an element in T_{id^*} which is not present in $\cup_{\mathsf{id}_i \in \mathcal{I}} T_{\mathsf{id}_i}$. Hence, we can use an adversary $\mathcal{A}$ who breaks unforgeability to break the pseudorandomness of PRG.

Fake Setup: In our hybrids, we will also use a fake algorithm of setup. Consider a scenario where a PPT adversary $\mathcal{A}$ controls k corrupt users with identities $\mathsf{id}_1, \ldots, \mathsf{id}_k$, without loss of generality. The fake setup algorithm we describe below will generate keys $(\mathsf{ask}, \mathsf{avk})$ such that it is only possible to authenticate the corrupt users and there does not exist any authentication which verifies under avk for honest users. Moreover, these two settings should be indistinguishable to the adversary. Below, we describe this setup procedure and then state and prove the security property.

The algorithm $\mathsf{FakeAuthGen}(1^\lambda, \mathsf{id}_1, \ldots, \mathsf{id}_k)$ works follows: For each $i \in [k]$, interpret id_i as a polynomial in $F_{q,d}$ for $d = \frac{q-1}{k}$, i.e., $\mathsf{id}_i \in \mathbb{F}_q^{d+1}$. Let T_{id_i} be the corresponding set in $\mathcal{T}$. Define $T^* = \cup_i T_{\mathsf{id}_i}$. Recall that the universe is $\mathbb{F}_q^2$.

Start with $\mathsf{ask} = \emptyset$. For all $i, j \in [q]$, if $(x_i, x_j) \in T^*$, pick $s_{ij} \xleftarrow{\$} Y$ and add (i, j, s_{ij}) to ask. For all $i, j \in [q]$, if $(x_i, x_j) \in T^*$, set $z_{ij} = \mathsf{PRG}(s_{ij})$ else set $z_{ij} \xleftarrow{\$} Z$. Define $\mathsf{avk} = \{\mathsf{PRG}(s_{ij})\}_{i,j \in [q]}$. Return $(\mathsf{avk}, \mathsf{ask})$.

Let $\mathcal{I} = \{\mathsf{id}_1, \ldots, \mathsf{id}_k\}$. The security properties of algorithm FakeAuthGen are:

- Correct authentication for all $\mathsf{id} \in \mathcal{I}$: It is easy to see that for any corrupt user $\mathsf{id} \in \mathcal{I}$, AuthProve will produce a t_{id} which will verify under avk.
- No authentication for all $\mathsf{id} \notin \mathcal{I}$: For any $\mathsf{id} \notin \mathcal{I}$, by property of k-cover-free sets, there exists a $(x_i, x_j) \in T_{\mathsf{id}}$ such that $(x_i, x_j) \notin T^*$. Moreover, a random element $z \xleftarrow{\$} Z$ does not lie in $\mathsf{im}(\mathsf{PRG})$ with probability $1 - \mathsf{negl}(\lambda)$. Hence, with probability $1 - \mathsf{negl}(\lambda)$, z_{ij} has no pre-image under PRG. This ensures that no t_{id} can verify under avk using algorithm Authverify.
- Indistinguishability: This implies that any PPT adversary given avk and t_{id} for all corrupt users cannot distinguish between real setup and fake setup. More formally, we have that for any PPT adversary $\mathcal{A}$, and any set of at most k corrupt ids $\mathcal{I} = \{\mathsf{id}_i\}_{i \in [k]}$, following holds.

$$\Pr\left[\mathcal{A}(\mathsf{avk}, \{t_{\mathsf{id}_i}\}_{i \in [k]}) = 1 \middle| \begin{array}{l}(\mathsf{ask}, \mathsf{avk}) \leftarrow \mathsf{AuthGen}(1^\lambda); \\ t_{\mathsf{id}_i} \leftarrow \mathsf{AuthProve}(\mathsf{ask}, \mathsf{id}_i) \\ \forall i \in [k]\end{array}\right] \approx$$
$$\Pr\left[\mathcal{A}(\mathsf{avk}, \{t_{\mathsf{id}_i}\}_{i \in [k]}) = 1 \middle| \begin{array}{l}(\mathsf{ask}, \mathsf{avk}) \leftarrow \mathsf{AuthGen}(1^\lambda, \mathcal{I}); \\ t_{\mathsf{id}_i} \leftarrow \mathsf{AuthProve}(\mathsf{ask}, \mathsf{id}_i) \\ \forall i \in [k]\end{array}\right]$$

We can prove this via a sequence of $q^2 - |T^*|$ hybrids. In the first hybrid, we use the algorithm $\mathsf{AuthGen}$ to produce the keys. In each subsequent hybrid, we pick a new i, j such that $(x_i, x_j) \notin T^*$ and change z_{ij} to a random element in Z instead of $\mathsf{PRG}(s_{ij}$. Indistinguishability of any two consecutive hybrids can be reduced to the pseudorandomness of PRG.

3 Secure Cloud Service Scheme (SCSS) Model

In this section, we first describe the setting of the secure cloud service, followed by various algorithms associated with the scheme and finally the desired security properties.

In this setting, we have three parties: *The provider*, who owns a program P, the *cloud*, where the program is hosted, and arbitrary many collection of *users*. At a very high level, the provider wants to hosts the program P on a cloud. Additionally, it wants to authenticate users who pay for the service. This authentication should allow a legitimate user to access the program hosted on the cloud and compute output on inputs of his choice. To be useful, we require the scheme to satisfy the following efficiency properties:

Weak Client. The amount of work done by the client should depend only on the size of the input and the security parameter and should be completely independent of the running time of the program P. In other words, the client should perform significantly less work than executing the program himself. This implies that both the initial set up phase with the provider and the subsequent encoding of inputs to the cloud are both highly efficient.

Delegation. The one-time work done by the provider in hosting the program should be bounded by a fixed polynomial in the program size. But, henceforth, we can assume that the work load of the provider in authenticating users only depends on the security parameter.

Polynomial Slowdown. The running time of the cloud on encoded program is bounded by a fixed polynomial in the running time of the actual program.

Next, we describe the different procedures associated with the scheme formally.

Definition 6 (Secure Cloud Service Scheme (SCSS)). *A secure cloud service scheme consists of following procedures* $\mathsf{SCSS} = (\mathsf{SCSS.prog}, \mathsf{SCSS.auth}, \mathsf{SCSS.inp}, \mathsf{SCSS.eval})$*:*

- $(\tilde{P}, \sigma) \leftarrow \mathsf{SCSS.prog}(1^\lambda, P, k)$*: Takes as input security parameter* λ*, program* P *and a bound* k *on the number of corrupt users and returns encoded program* $\tilde{P}$ *and a secret* σ *to be useful in authentication.*
- $\mathsf{auth_{id}} \leftarrow \mathsf{SCSS.auth}(\mathsf{id}, \sigma)$*: Takes the identity of a client and the secret* σ *and produces an authentication* $\mathsf{auth_{id}}$ *for the client.*
- $(\tilde{x}, \alpha) \leftarrow \mathsf{SCSS.inp}(1^\lambda, \mathsf{auth_{id}}, x)$*: Takes as input the security parameter, authentication for the identity and the input* x *to produce encoded input* $\tilde{x}$*. It also outputs* α *which is used by the client later to decode the output obtained.*
- $\tilde{y} \leftarrow \mathsf{SCSS.eval}(\tilde{P}, \tilde{x})$*: Takes as input encoded program and encoded input and produces encoded output. This can be later decoded by the client using* α *produced in the previous phase.*

In our scheme, the provider will run the procedure $\mathsf{SCSS.prog}$ to obtain the encoded program $\tilde{P}$ and the secret σ. It will then send $\tilde{P}$ to the cloud. Later, it will authenticate users using σ. A user with identity id who has a authentication $\mathsf{auth}_{\mathsf{id}}$, will encode his input x using procedure $\mathsf{SCSS.inp}$ to produce encoded input $\tilde{x}$ and secret α. He will send $\tilde{x}$ to the cloud. The cloud will evaluate the encoded program $\tilde{P}$ on encoded input $\tilde{x}$ and return encoded output $\tilde{y}$ to the user. The user can now decode the output using α.

Security properties. Our scheme is for the benefit of the provider and hence we assume that the provider is uncompromised. The various security properties desired are as follows:

Definition 7 (Untrusted Cloud Security). *Let* SCSS *be the secure cloud service scheme as described above. This scheme satisfies untrusted cloud security if the following holds. We consider an adversary who corrupts the cloud as well as k clients $\mathcal{I}' = \{\mathsf{id}'_1, \ldots, \mathsf{id}'_k\}$. Consider two programs P and P' such that $P(\mathsf{id}'_i, x) = P'(\mathsf{id}'_i, x)$ for all $i \in [k]$ and all inputs x. Let $m(\lambda)$ be an efficiently computable polynomial. For any m honest users identities $\mathcal{I} = \{\mathsf{id}_1, \ldots, \mathsf{id}_m\}$ such that $\mathcal{I} \cap \mathcal{I}' = \emptyset$ and for any sequence of pairs of inputs for honest users $\{(x_1, x'_1), \ldots, (x_m, x'_m)\}$, consider the following two experiments:*

The experiment $\mathsf{Real}(1^\lambda)$ *is as follows:*

1. $(\tilde{P}, \sigma) \leftarrow \mathsf{SCSS.prog}(1^\lambda, P, k)$.
2. *For all* $i \in [m]$, $\mathsf{auth}_{\mathsf{id}_i} \leftarrow \mathsf{SCSS.auth}(\mathsf{id}_i, \sigma)$.
3. *For all* $i \in [m]$, $(\tilde{x}_i, \alpha_i) \leftarrow \mathsf{SCSS.inp}(1^\lambda, \mathsf{id}_i, \mathsf{auth}_{\mathsf{id}_i}, x_i)$.
4. *For all* $j \in [k]$, $\mathsf{auth}_{\mathsf{id}'_j} \leftarrow \mathsf{SCSS.auth}(\mathsf{id}'_j, \sigma)$.
5. *Output* $(\tilde{P}, \{\mathsf{auth}_{\mathsf{id}'_j}\}_{j \in [k]}, \{\tilde{x}_i\}_{i \in [m]})$.

The experiment $\mathsf{Real}'(1^\lambda)$ *is as follows:*

1. $(\tilde{P}', \sigma) \leftarrow \mathsf{SCSS.prog}(1^\lambda, P', k)$.
2. *For all* $i \in [m]$, $\mathsf{auth}_{\mathsf{id}_i} \leftarrow \mathsf{SCSS.auth}(\mathsf{id}_i, \sigma)$.
3. *For all* $i \in [m]$, $(\tilde{x}'_i, \alpha_i) \leftarrow \mathsf{SCSS.inp}(1^\lambda, \mathsf{id}_i, \mathsf{auth}_{\mathsf{id}_i}, x'_i)$.
4. *For all* $j \in [k]$, $\mathsf{auth}_{\mathsf{id}'_j} \leftarrow \mathsf{SCSS.auth}(\mathsf{id}'_j, \sigma)$.
5. *Output* $(\tilde{P}', \{\mathsf{auth}_{\mathsf{id}'_j}\}_{j \in [k]}, \{\tilde{x}'_i\}_{i \in [m]})$.

Then we have,

$$\mathsf{Real}(1^\lambda) \approx_c \mathsf{Real}'(1^\lambda)$$

Remark: In the above definition, the only difference between two experiments is that Real uses the program P and honest users inputs $\{x_1, \ldots, x_m\}$ and Real' uses program P' and honest users inputs $\{x'_1, \ldots, x'_m\}$. Note that no relationship is required to exist between the set of inputs $\{x_1, \ldots, x_m\}$ and the set of inputs $\{x'_1, \ldots, x'_m\}$.

Definition 8 (Untrusted Client Security). *Let* SCSS *be the secure cloud service scheme as described above. This scheme satisfies untrusted client security if the following holds.*

Let $\mathcal{A}$ be a PPT adversary who corrupts at most k clients $\mathcal{I}' = \{\mathsf{id}'_1, \ldots, \mathsf{id}'_k\}$. Consider any program P. Let $n(\lambda)$ be an efficiently computable polynomial. Consider the following two experiments:

The experiment $\mathsf{Real}(1^\lambda)$ *is as follows:*

1. $(\tilde{P}, \sigma) \leftarrow \mathsf{SCSS.prog}(1^\lambda, P, k)$.
2. *For all* $i \in [k]$, $\mathsf{auth}_{\mathsf{id}'_i} \leftarrow \mathsf{SCSS.auth}(\mathsf{id}'_i, \sigma)$. *Send* $\{\mathsf{auth}_{\mathsf{id}'_i}\}_{i\in[k]}$ *to* $\mathcal{A}$.
3. *For each* $i \in [n]$,
   - $\mathcal{A}$ *(adaptively) sends an encoding* $\tilde{x}_i$ *using identity* id.
   - *Run* $\mathsf{SCSS.eval}(\tilde{P}, \tilde{x}_i)$ *to compute* $\tilde{y}_i$. *Send this to* $\mathcal{A}$.
4. *Output* $(\{\mathsf{auth}_{\mathsf{id}'_i}\}_{i\in[k]}, \{\tilde{y}_i\}_{i\in[n]})$.

We require that there The definition requires that there exist two procedures decode *and* $\mathsf{response}$. *Based on these procedures, we define* $\mathsf{Sim}^P(1^\lambda)$ *w.r.t. an oracle for the program P. Below,* dummy *is any program of the same size as P.*

1. $(\widetilde{\mathsf{dummy}}, \sigma) \leftarrow \mathsf{SCSS.prog}(1^\lambda, \mathsf{dummy}, k)$.
2. *For all* $i \in [k]$, $\mathsf{auth}_{\mathsf{id}'_i} \leftarrow \mathsf{SCSS.auth}(\mathsf{id}'_i, \sigma)$. *Send* $\{\mathsf{auth}_{\mathsf{id}'_i}\}_{i\in[k]}$ *to* $\mathcal{A}$.
3. *For each* $i \in [n]$,
   - $\mathcal{A}$ *(adaptively) sends an encoding* $\tilde{x}_i$ *using some identity* id.
   - *If* $\mathsf{id} \notin \mathcal{I}'$ *set* $\tilde{y} = \perp$. *Otherwise, run* $\mathsf{decode}(\sigma, \tilde{x}_i)$ *which either outputs* (x_i, τ_i) *or* $\perp$. *If it outputs* $\perp$, *set* $\tilde{y} = \perp$. *Else, the simulator sends* (id, x_i) *to the oracle and obtains* $y_i = P(\mathsf{id}, x_i)$. *Finally, it computes* $\tilde{y}_i \leftarrow \mathsf{response}(y_i, \tau_i, \sigma)$. *Send* $\tilde{y}_i$ *to* $\mathcal{A}$.
4. *Output* $(\{\mathsf{auth}_{\mathsf{id}'_i}\}_{i\in[k]}, \{\tilde{y}_i\}_{i\in[n]})$.

Then we have,

$$\mathsf{Real}(1^\lambda) \approx_c \mathsf{Sim}^P(1^\lambda)$$

Intuitively, the above security definition says that a collection of corrupt clients do not learn anything beyond the program's output w.r.t. to their identities on certain inputs of their choice. Moreover, it says that if a client is not authenticated, it learns nothing.

We describe a scheme which is a secure cloud service scheme in Section 4 and prove its security in Section 4.1.

3.1 Additional Properties.

We believe our scheme can also be modified to achieve some additional properties which are not the focus of this work. We use this section to mention them below.

Verifiability. In the above scenario, where the cloud outputs $\tilde{y}$ intended for the client, we may also want to add verifiability, where the client is sure that the received output $\tilde{y}$ is indeed the correct output of the computation. We stress that verifiability is not the focus of this work. The scheme we present in Section 4 can be augmented with known techniques to get verifiability. One such method is to use one-time MACs as suggested in [8].

Persistent Memory. An interesting setting to consider is the one where the the cloud also holds a user-specific persistent memory that maintains state across different invocations of the service by the user. In this setting we must ensure that for each invocation of the functionality by a user, there only exists one valid state for the persistent memory that can be used for computing the user's output and the next state for the persistent memory. Such a result would only require the assumptions present in Theorem 2, and would not require any complexity leveraging. We believe that techniques developed in this paper along with those in [8] should be helpful to realize this setting as well.

3.2 Secure Cloud Service Scheme with Cloud Inputs.

Here we consider a more general scenario, where the program takes two inputs: one from the user and another from the cloud.

This setting is technically more challenging since the cloud can use any input in each invocation of the program. In particular, it allows users to access super-polynomially potentially different functionalities on the cloud based on cloud's input.

Notationally, this scheme is same as the previous scheme except that the procedure $\mathsf{SCSS.eval}(\tilde{P}, \tilde{x}, z) \rightarrow \tilde{y}$ takes additional input z from the cloud. The efficiency and security requirements for this scheme are essentially the same as the simple scheme without the cloud inputs.

There is absolutely no change required in Definition 7. This is because it talks about the view of a malicious cloud. There is a minor change in untrusted client security (Definition 8). The oracle on query (id, x_i), returns $P(\mathsf{id}'_i, x_i, z_i)$, where $z_1, \ldots, z_n$ are arbitrarily chosen choice for cloud's inputs. Note that the security guarantee for an honest cloud is captured in this definition.

We provide a scheme which is secure cloud service scheme with cloud inputs in Section 5.

4 Our Secure Cloud Service Scheme

In this section, we describe our scheme for hosting on the cloud. We have three different parties: The provider who owns the program, the cloud where the program is hosted, and the users. Recall that we assume that the provider of the service is honest.

Let λ be the security parameter. Note that the number of users can be any (unbounded) polynomial in λ. Let k be the bound on the number of corrupt users. In our security game, we allow the cloud as well as any subset of users to be controlled by the adversary as long as the number of such users is at most k.

In order to describe our construction, we first recall the primitives and their notation that we use in our protocol. Let $\mathcal{T}$ be a k-cover-free set system using a finite field $\mathbb{F}_q$ and polynomials of degee $d = (q-1)/k$ described in Section 2.5. Let $(\mathsf{AuthGen}, \mathsf{AuthProve}, \mathsf{Authverify})$ be the authentication scheme based on this k-cover-free set system. As mentioned before, we will use $q = k\lambda$, so that the number of sets/users is at least 2^λ. We will interpret the user's identity id as the coefficients of a

polynomial over $\mathbb{F}_q$ of degree at most d. Let the length of the identity be $\ell_{\mathsf{id}} := (d+1)\lg q$ and length of the authentication be ℓ_{auth}. Note that in our scheme $\ell_{\mathsf{auth}} = 2\lambda q$.

Let $\mathsf{pke} = (\mathsf{PKGen}, \mathsf{PKEnc}, \mathsf{PKDec})$ be public key encryption scheme which accepts messages of length $\ell_e = (\ell_{\mathsf{id}} + \ell_{\mathsf{in}} + \ell_{\mathsf{auth}} + \ell_{\mathsf{kout}} + 1)$ and returns ciphertexts of length ℓ_c. Here ℓ_{in} is the length of the input of the user and ℓ_{kout} is the length of the key for PRF_2 described below.

Let $(\mathsf{NIZKSetup}, \mathsf{NIZKProve}, \mathsf{NIZKVerify})$ be the statistical simulation-sound non-interactive zero-knowledge proof system with simulator $(\mathsf{S}_1, \mathsf{S}_2)$. In our scheme we use the two-key paradigm along with statistically simulation-sound non-interactive zero-knowledge for non-malleability inspired from [15,17,6].

We will make use of two different family of puncturable PRFs. a) $\mathsf{PRF}_1(K, \cdot)$ that accepts inputs of length $(\ell_{\mathsf{id}} + \ell_c)$ and returns strings of length ℓ_r. b) $\mathsf{PRF}_2(K_{\mathsf{id}}, \cdot)$ that accepts inputs of length ℓ_r and returns strings of length ℓ_{out}, where ℓ_{out} is the length of the output of program. Such PRFs exist by Theorem 3.

Now we describe our scheme.

Consider an honest provider $\mathcal{H}$ who holds a program F which he wants to hosts on the cloud $\mathcal{C}$. Also, there will be a collection of users who will interact with the provider to obtain authentication which will enable them to run the program stored on the cloud. We first describe the procedure $\mathsf{SCSS.prog}(1^\lambda, \mathsf{F}, k)$ run by the provider.

1. Chooses PRF key K at random for PRF_1.
2. Picks $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{PKGen}(1^\lambda)$, $(\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow \mathsf{PKGen}(1^\lambda)$.
3. Picks $(\mathsf{avk}, \mathsf{ask}) \leftarrow \mathsf{AuthGen}(1^\lambda)$ with respect to k-cover-free set system $\mathcal{T}$ and pseudorandom generator $\mathsf{PRG} : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$.
4. Picks $\mathsf{crs} \leftarrow \mathsf{NIZKSetup}(1^\lambda)$.
5. Creates an indistinguishability obfuscation $P_{comp} = i\mathcal{O}(\text{Compute})$, where Compute is the program described in Figure 1.

Here $\tilde{F} = P_{comp}$ and $\sigma = (\mathsf{ask}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs}, K)$. Note that K is not used by the honest provider in any of the future steps, but we include it as part of secret for completion. This would be useful in proving untrusted client security later.

Next, we describe the procedure $\mathsf{SCSS.auth}(\mathsf{id}, \sigma = (\mathsf{ask}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs}))$, where a user sends his id to the provider for authentication. The provider sends back $\mathsf{auth}_{\mathsf{id}} = (t_{\mathsf{id}}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs})$, where $t_{\mathsf{id}} = \mathsf{AuthProve}(\mathsf{ask}, \mathsf{id})$. We also describe this interaction in Figure 2.

Finally, we describe the procedures SCSS.inp and SCSS.eval. This interaction between the user and the cloud is also described in Figure 3.

Procedure $\mathsf{SCSS.inp}(1^\lambda, \mathsf{auth}_{\mathsf{id}} = (t_{\mathsf{id}}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs}), x)$: The user chooses a key $K_{\mathsf{id,out}}$ for PRF_2. Let $m = (\mathsf{id}||x||t_{\mathsf{id}}||K_{\mathsf{id,out}}||0)$. It then computes $c_1 = \mathsf{PKEnc}(\mathsf{pk}_1, m; r_1)$, $c_2 = \mathsf{PKEnc}(\mathsf{pk}_2, m; r_2)$ and a SSS-NIZK proof π for

$$\exists m, t_1, t_2 \text{ s.t. } (c_1 = \mathsf{PKEnc}(\mathsf{pk}_1, m; t_1) \wedge c_2 = \mathsf{PKEnc}(\mathsf{pk}_2, m; t_2))$$

It outputs $\tilde{x} = (\mathsf{id}, c = (c_1, c_2, \pi))$ and $\alpha = K_{\mathsf{id,out}}$.

Procedure $\mathsf{SCSS.eval}(\tilde{F} = P_{comp}, \tilde{x})$: Run $\tilde{F}$ on $\tilde{x}$ to obtain $\tilde{y}$. The user parses $\tilde{y}$ as $\tilde{y}_1, \tilde{y}_2$ and computes $y = \mathsf{PRF}_2(\alpha, \tilde{y}_1) \oplus \tilde{y}_2$.

Compute

Constants: Secret key sk_1, puncturable PRF key K, verification key avk and common reference string crs.
Input: Identity id and ciphertext $c = (c_1, c_2, \pi)$.

1. If $\mathsf{NIZKVerify}(\mathsf{crs}, (c_1, c_2), \pi) = 0$, output $\perp$ and end.
2. Let $(\mathsf{id}'||x||t_{\mathsf{id}}||K_{\mathsf{id,out}}||\mathsf{flag}) = \mathsf{PKDec}(\mathsf{sk}_1, c_1)$ for appropriate length strings. If $\mathsf{id} \neq \mathsf{id}'$, output $\perp$ and end.
3. Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}, c))$.
4. If $\mathsf{flag} = 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id,out}}, r))$ and end.
5. Else if $\mathsf{flag} = 0$, and $\mathsf{Authverify}(\mathsf{avk}, \mathsf{id}, t_{\mathsf{id}}) = 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id,out}}, r) \oplus \mathsf{F}(\mathsf{id}, x))$ and end.
6. Output $y = \perp$ and end.

Fig. 1: Program Compute

Provider and User

Inputs: Let the user's identity be id. The provider has two public keys $\mathsf{pk}_1, \mathsf{pk}_2$, common reference string crs and secret key ask for authentication.

1. The user sends his identity id to the provider.
2. The provider computes $t_{\mathsf{id}} \leftarrow \mathsf{AuthProve}(\mathsf{ask}, \mathsf{id})$ and sends $\mathsf{auth}_{\mathsf{id}} = (t_{\mathsf{id}}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs})$ to the user.

Fig. 2: Authentication phase between the provider and the user.

User and Cloud

Inputs: Let the user's identity be id. Let the user's input to the function be x. An authenticated user has the authentication $\mathsf{auth}_{\mathsf{id}} = (t_{\mathsf{id}}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs})$ obtained from the provider. The cloud has obfuscated program P_{comp}. The user encodes his input for the cloud using $\mathsf{SCSS.inp}(1^\lambda, \mathsf{auth}_{\mathsf{id}}, x)$ as follows:

1. Pick a key $K_{\mathsf{id,out}}$ for PRF_2. Set $\mathsf{flag} = 0$.
2. Let $m = (\mathsf{id}||x||t_{\mathsf{id}}||K_{\mathsf{id,out}}||\mathsf{flag})$. Compute $c_1 = \mathsf{PKEnc}(\mathsf{pk}_1, m; r_1)$, $c_2 = \mathsf{PKEnc}(\mathsf{pk}_2, m; r_2)$ and a SSS-NIZK proof $\pi = \mathsf{NIZKProve}(\mathsf{crs}, \mathsf{stmt}, (m, r_1, r_2))$, where stmt is

   $$\exists m, t_1, t_2 \text{ s.t. } (c_1 = \mathsf{PKEnc}(\mathsf{pk}_1, m; t_1) \wedge c_2 = \mathsf{PKEnc}(\mathsf{pk}_2, m; t_2))$$

3. $\tilde{x} = (\mathsf{id}, c = (c_1, c_2, \pi))$ and $\alpha = K_{\mathsf{id,out}}$.

The cloud runs the program P_{comp} on the input $\tilde{x}$ and obtains output $\tilde{y}$. It sends $\tilde{y}$ to the user.
The user parses $\tilde{y}$ as $\tilde{y}_1, \tilde{y}_2$ and computes $y = \mathsf{PRF}_2(\alpha, \tilde{y}_1) \oplus \tilde{y}_2$.

Fig. 3: Encoding and evaluation phase between an authenticated user and the cloud

4.1 Security Proof

In this section, we give a proof overview for Theorem 1 for the scheme described above.

Untrusted client security. In our scheme, the secret information σ created after running the procedure $\mathsf{SCSS.prog}$ is $\sigma = (\mathsf{ask}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs}, K)$. Hence, on obtaining a encoded $\tilde{x}$ from the adversary the decode procedure can work identically to the program Compute to extract an input x, authentication t_{id}, a key $K_{\mathsf{id},\mathsf{out}}$, and flag from $\tilde{x}$. If $\mathsf{flag} = 1$, it gives $y = 0$ to $\mathsf{response}$ procedure. Else, if authentication t_{id} verifies using avk, it sends the (id, x) to the oracle implementing P and obtains y which is sent to $\mathsf{response}$. The $\mathsf{response}$ procedure finally encodes the output y using $\tau = K_{\mathsf{id},\mathsf{out}}$ and $K \in \sigma$ and sends it to the corrupt client. if $\mathsf{flag} = 0$ and t_{id} is invalid, send $\perp$ to the client. This is exactly what the obfuscated program would have done. Hence, the real and simulated experiments are indistinguishable as is required by this security property.

To prove security against unauthenticated clients, we need to prove the following: Any PPT malicious client id who has not done the set up phase to obtain $\mathsf{auth}_{\mathsf{id}}$ should not be able to learn the output of F on any input using interaction with the honest cloud. Note that in our scheme F is invoked only if the authentication extracted by the program verifies under avk. Hence, the security will follow from the k-unforgeability property of our scheme (see Section 2.5).

Untrusted cloud security. Consider a PPT adversary $\mathcal{A}$ who controls the cloud and a collection of at most k users. Let F and G be two functions such that F and G are functionally equivalent for corrupt users. Then, we will prove that $\mathcal{A}$ can not distinguish between the cases when the provider uses the function F or G. We will prove this via a sequence of hybrids. Below, we first give a high level overview of these hybrids.

Let m be the number of honest users in the scheme. Without loss of generality, let their identities be $\mathsf{id}_1, \ldots, \mathsf{id}_m$ and inputs be $x_1, \ldots, x_m$. In the first sequence of hybrids, we will change the interaction of the honest users with the cloud such that all honest user queries will use $\mathsf{flag} = 1$ and input $0^{\ell_{\mathsf{in}}}$. This will ensure that in the final hybrid of this sequence, function F is not being invoked for any of the honest users.

In the next sequence of hybrids, we will change the output of the procedure $\mathsf{AuthGen}$ such that there does not exist any valid authentication for honest users. Now, we can be absolutely certain that the program does not invoke the function F on any of the honest ids. We also know that the functions F and G are functionally equivalent for all the corrupt ids. At this point, we can rely on the indistinguishability of obfuscations of program Compute using F and program Compute using G.

Finally, we can reverse the sequence of all the hybrids used so far so that the final hybrid is the real execution with G with honest user inputs $x'_1, \ldots, x'_m$.

Overview of hybrids. Below we describe the important steps in the hybrid arguments. For detailed and formal security proof, refer to the full version. We denote changes between subsequent hybrids using underlined font.

Each hybrid below is an experiment that takes as input 1^λ. The final output of each hybrid experiment is the output produced by the adversary when it terminates. More-

over, in each of these hybrids, note that the adversary also receives authentication identities for all the corrupt users.

The first hybrid Hyb_0 is the real execution with F.

1. Choose PRF key K at random for PRF_1.
2. Pick $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{PKGen}(1^\lambda)$, $(\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow \mathsf{PKGen}(1^\lambda)$.
3. Pick $(\mathsf{avk}, \mathsf{ask}) \leftarrow \mathsf{AuthGen}(1^\lambda)$ with respect to cover-free set system $\mathcal{T}$ and pseudorandom generator PRG.
4. Pick $\mathsf{crs} \leftarrow \mathsf{NIZKSetup}(1^\lambda)$.
5. Let $P_{comp} = i\mathcal{O}(\text{Compute})$.
6. On receiving a corrupt user's identity id, return the authentication as in real execution. That is, compute $t_{\mathsf{id}} \leftarrow \mathsf{AuthProve}(\mathsf{ask}, \mathsf{id})$ and send $(t_{\mathsf{id}}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs})$ to the user.
7. Authentication for honest users and queries of honest users are also computed as in real execution. See Figure 3 for details.
8. Output P_{comp} and queries of all honest users $\mathsf{id}_1, \ldots, \mathsf{id}_m$.

Compute

Constants: Secret key sk_1, puncturable PRF key K, verification key avk and common reference string crs.
Input: Identity id and ciphertext $c = (c_1, c_2, \pi)$.

1. If $\mathsf{NIZKVerify}(\mathsf{crs}, (c_1, c_2), \pi) = 0$, output $\perp$ and end.
2. Let $(\mathsf{id}'||x||t_{\mathsf{id}}||K_{\mathsf{id},\mathsf{out}}||\mathsf{flag}) = \mathsf{PKDec}(\mathsf{sk}_1, c_1)$ for appropriate length strings. If $\mathsf{id} \neq \mathsf{id}'$, output $\perp$ and end.
3. Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}, c))$.
4. If $\mathsf{flag} = 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r))$ and end.
5. Else if $\mathsf{flag} = 0$, and $\mathsf{Authverify}(\mathsf{avk}, \mathsf{id}, t_{\mathsf{id}}) = 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x))$ and end.
6. Output $y = \perp$ and end.

Fig. 4: Program Compute

Next we describe the first sequence of hybrids $\mathsf{Hyb}_{1:1}, \ldots, \mathsf{Hyb}_{1:6}, \ldots, \mathsf{Hyb}_{m:1}, \ldots, \mathsf{Hyb}_{m:6}$. In the sub-sequence of hybrids $\mathsf{Hyb}_{i:1}, \ldots, \mathsf{Hyb}_{i:6}$, we only change the behavior of the honest user id_i. All the other honest users id_j such that $j \neq i$ behave identically as in $\mathsf{Hyb}_{i-1:6}$. Hence, we omit their behavior from the description of the hybrids for ease of notation.

Also, we denote id_i by id^*.

Let $\mathsf{Hyb}_{0:6} = \mathsf{Hyb}_0$.

$\mathsf{Hyb}_{i:1}$. This is same as $\mathsf{Hyb}_{i-1:6}$. We use this hybrid as a way to write how the user id^* behaves in the real execution explicitly. This would make it easier to describe the changes next.

It is obvious that the output of the adversary in $\mathsf{Hyb}_{i-1:6}$ and $\mathsf{Hyb}_{i:1}$ is identical.

1. Choose PRF key K at random for PRF_1.
2. Pick $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{PKGen}(1^\lambda)$, $(\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow \mathsf{PKGen}(1^\lambda)$.
3. Pick $(\mathsf{avk}, \mathsf{ask}) \leftarrow \mathsf{AuthGen}(1^\lambda)$ with respect to cover-free set system $\mathcal{T}$ and PRG.
4. On receiving a corrupt user's identity id, return the authentication as in real execution. That is, compute $t_{\mathsf{id}} \leftarrow \mathsf{AuthProve}(\mathsf{ask}, \mathsf{id})$ and send $(t_{\mathsf{id}}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs})$ to the user.
5. Set $t_{\mathsf{id}^*} = \mathsf{AuthGen}(\mathsf{ask}, \mathsf{id}^*)$.
6. Set $\mathsf{flag}^* = 0$.
7. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for PRF_2.
8. Let x^* be the input. Let $m_1^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}||\mathsf{flag}^*)$ and $m_2^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}||\mathsf{flag}^*)$.
9. Compute c_1^*, c_2^* as follows: $c_1^* = \mathsf{PKEnc}(\mathsf{pk}_1, m_1^*; r_1)$, $c_2^* = \mathsf{PKEnc}(\mathsf{pk}_2, m_2^*; r_2)$.
10. Pick $\mathsf{crs} \leftarrow \mathsf{NIZKSetup}(1^\lambda)$.
11. Compute $\pi^* = \mathsf{NIZKProve}(\mathsf{crs}, \mathsf{stmt}, (m_1^*, r_1, r_2))$.
12. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
13. Let $r^* = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*))$.
14. Let $y^* = (r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*) \oplus \mathsf{F}(\mathsf{id}^*, x^*))$.
15. Let $P_{comp} = i\mathcal{O}(\text{Compute})$.
16. Output $(P_{comp}, (\mathsf{id}^*, c^*))$ and the queries of other honest users.

Compute

Constants: Secret key sk_1, puncturable PRF key K, verification key avk and common reference string crs.
Input: Identity id and ciphertext $c = (c_1, c_2, \pi)$.

1. If $\mathsf{NIZKVerify}(\mathsf{crs}, (c_1, c_2), \pi) = 0$, output $\perp$ and end.
2. Let $(\mathsf{id}'||x||t_{\mathsf{id}}||K_{\mathsf{id},\mathsf{out}}||\mathsf{flag}) = \mathsf{PKDec}(\mathsf{sk}_1, c_1)$. If $\mathsf{id} \neq \mathsf{id}'$, output $\perp$.
3. Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}, c))$.
4. If $\mathsf{flag} = 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r))$ and end.
5. Else if $\mathsf{flag} = 0$, and $\mathsf{Authverify}(\mathsf{avk}, \mathsf{id}, t_{\mathsf{id}}) = 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x))$ and end.
6. Output $y = \perp$ and end.

Fig. 5: Program Compute

$\mathsf{Hyb}_{i:2}$. We modify the Compute program as follows. First, we add the constants id^*, c^*, y^* to the program and add an if statement that outputs y^* if the input is (id^*, c^*). Now, because the "if" statement is in place, we know that $\mathsf{PRF}_1(K, \cdot)$ cannot be evaluated at (id^*, c^*) within the program. Hence, we can safely puncture key K at this point. By construction, the Compute program in this hybrid is functionally equivalent to the Compute program in the previous hybrid. Hence, indistinguishability follows by the $i\mathcal{O}$ security.

Next, the value r^* is chosen randomly (at the beginning), instead of as the output of $\mathsf{PRF}_1(K, (\mathsf{id}^*, c^*))$. The indistinguishability of two hybrids follows by pseudorandomness property of the punctured PRF PRF_1.

1. Choose PRF key K at random for PRF_1.
2. Pick $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{PKGen}(1^\lambda)$, $(\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow \mathsf{PKGen}(1^\lambda)$.
3. Pick $(\mathsf{avk}, \mathsf{ask}) \leftarrow \mathsf{AuthGen}(1^\lambda)$ with respect to cover-free set system $\mathcal{T}$ and PRG.
4. On receiving a corrupt user's identity id, return the authentication as in real execution. That is, compute $t_{\mathsf{id}} \leftarrow \mathsf{AuthProve}(\mathsf{ask}, \mathsf{id})$ and send $(t_{\mathsf{id}}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs})$.
5. Set $t_{\mathsf{id}^*} = \mathsf{AuthGen}(\mathsf{ask}, \mathsf{id}^*)$.
6. Set $\mathsf{flag}^* = 0$.
7. Let r^* be chosen randomly.
8. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for PRF_2.
9. Let x^* be the input. Let $m_1^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}||\mathsf{flag}^*)$ and $m_2^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}||\mathsf{flag}^*)$.
10. Compute c_1^*, c_2^* as follows: $c_1^* = \mathsf{PKEnc}(\mathsf{pk}_1, m_1^*; r_1)$, $c_2^* = \mathsf{PKEnc}(\mathsf{pk}_2, m_2^*; r_2)$.
11. Pick $\mathsf{crs} \leftarrow \mathsf{NIZKSetup}(1^\lambda)$.
12. Compute $\pi^* = \mathsf{NIZKProve}(\mathsf{crs}, \mathsf{stmt}, (m_1^*, r_1, r_2))$.
13. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
14. Let $y^* = (r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*) \oplus \mathsf{F}(\mathsf{id}^*, x^*))$.
15. Let $P_{comp} = i\mathcal{O}(\text{Compute})$.
16. Output $(P_{comp}, (\mathsf{id}^*, c^*))$ and the queries of other honest users.

Compute

Constants: $(\mathsf{id}^*, c^*, y^*)$, Secret key sk_1, puncturable PRF key $K(\{(\mathsf{id}^*, c^*)\})$, verification key avk and common reference string crs.
Input: Identity id and ciphertext $c = (c_1, c_2, \pi)$.

1. If $(\mathsf{id}, c) = (\mathsf{id}^*, c^*)$ output y^* and end.
2. If $\mathsf{NIZKVerify}(\mathsf{crs}, (c_1, c_2), \pi) = 0$, output $\perp$ and end.
3. Let $(\mathsf{id}'||x||t_{\mathsf{id}}||K_{\mathsf{id},\mathsf{out}}||\mathsf{flag}) = \mathsf{PKDec}(\mathsf{sk}_1, c_1)$. If $\mathsf{id} \neq \mathsf{id}'$, output $\perp$.
4. Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}, c))$.
5. If $\mathsf{flag} = 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r))$ and end.
6. Else if $\mathsf{flag} = 0$, and $\mathsf{Authverify}(\mathsf{avk}, \mathsf{id}, t_{\mathsf{id}}) = 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x))$ and end.
7. Output $y = \perp$ and end.

Fig. 6: Program Compute

$\mathsf{Hyb}_{i:3}$. In this sequence of hybrids, first, instead of generating crs honestly, we generate it using the simulator S_1 and also simulate the proof π^* using S_2. The two hybrids are indistinguishable by computational zero-knowledge property of NIZK used.

Next, using a sequence of hybrids, using the two-key switching technique we change both m_1^* and m_2^* to include a punctured key $K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})$ instead of original key $K_{\mathsf{id}^*,\mathsf{out}}$. In these hybrids we will be relying on $i\mathcal{O}$ as well as $\mathsf{IND}-\mathsf{CPA}$ security of public key encryption scheme pke. For details, refer to the full version.

1. Choose PRF key K at random for PRF_1.
2. Pick $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{PKGen}(1^\lambda)$, $(\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow \mathsf{PKGen}(1^\lambda)$.
3. Pick $(\mathsf{avk}, \mathsf{ask}) \leftarrow \mathsf{AuthGen}(1^\lambda)$ with respect to cover-free set system $\mathcal{T}$ and PRG.
4. On receiving a corrupt user's identity id, return the authentication as in real execution. That is, compute $t_{\mathsf{id}} \leftarrow \mathsf{AuthProve}(\mathsf{ask}, \mathsf{id})$ and send $(t_{\mathsf{id}}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs})$.
5. Set $t_{\mathsf{id}^*} = \mathsf{AuthGen}(\mathsf{ask}, \mathsf{id}^*)$.
6. Set $\mathsf{flag}^* = 0$.
7. Let r^* be chosen randomly.
8. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for PRF_2.
9. Let x^* be the input. Let $m_1^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})||\mathsf{flag}^*)$ and $m_2^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})||\mathsf{flag}^*)$.
10. Compute c_1^*, c_2^* as follows: $c_1^* = \mathsf{PKEnc}(\mathsf{pk}_1, m_1^*; r_1)$, $c_2^* = \mathsf{PKEnc}(\mathsf{pk}_2, m_2^*; r_2)$.
11. Pick $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$.
12. Compute $\pi^* = \mathsf{S}_2(\mathsf{crs}, \tau, \mathsf{stmt})$.
13. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
14. Let $y^* = (r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*) \oplus \mathsf{F}(\mathsf{id}^*, x^*))$.
15. Let $P_{comp} = i\mathcal{O}(\text{Compute})$.
16. Output $(P_{comp}, (\mathsf{id}^*, c^*))$ and the queries of other honest users.

Compute

Constants: $(\mathsf{id}^*, c^*, y^*)$, Secret key sk_1, puncturable PRF key $K(\{(\mathsf{id}^*, c^*)\})$, verification key avk and common reference string crs.
Input: Identity id and ciphertext $c = (c_1, c_2, \pi)$.

1. If $(\mathsf{id}, c) = (\mathsf{id}^*, c^*)$ output y^* and end.
2. If $\mathsf{NIZKVerify}(\mathsf{crs}, (c_1, c_2), \pi) = 0$, output $\perp$ and end.
3. Let $(\mathsf{id}'||x||t_{\mathsf{id}}||K_{\mathsf{id},\mathsf{out}}||\mathsf{flag}) = \mathsf{PKDec}(\mathsf{sk}_1, c_1)$. If $\mathsf{id} \neq \mathsf{id}'$, output $\perp$.
4. Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}, c))$.
5. If $\mathsf{flag} = 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r))$ and end.
6. Else if $\mathsf{flag} = 0$, and $\mathsf{Authverify}(\mathsf{avk}, \mathsf{id}, t_{\mathsf{id}}) = 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x))$ and end.
7. Output $y = \perp$ and end.

Fig. 7: Program Compute

$\mathsf{Hyb}_{i:4}$. Here, first we change the value of $y^* = (r^*, u^*)$ where u^* is a uniformly random string of appropriate length. By pseudorandomness property of punctured key $K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})$, $\mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*)$ is indistinguishable from random string.

Then, we change the value of y^* to $(r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*))$. Again indistinguishability follows from pseudorandomness property of punctured key.

1. Choose PRF key K at random for PRF_1.
2. Pick $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{PKGen}(1^\lambda)$, $(\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow \mathsf{PKGen}(1^\lambda)$.
3. Pick $(\mathsf{avk}, \mathsf{ask}) \leftarrow \mathsf{AuthGen}(1^\lambda)$ with respect to cover-free set system $\mathcal{T}$ and PRG.
4. On receiving a corrupt user's identity id, return the authentication as in real execution. That is, compute $t_{\mathsf{id}} \leftarrow \mathsf{AuthProve}(\mathsf{ask}, \mathsf{id})$ and send $(t_{\mathsf{id}}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs})$.
5. Set $t_{\mathsf{id}^*} = \mathsf{AuthGen}(\mathsf{ask}, \mathsf{id}^*)$.
6. Set $\mathsf{flag}^* = 0$.
7. Let r^* be chosen randomly.
8. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for PRF_2.
9. Let x^* be the input. Let $m_1^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})||\mathsf{flag}^*)$ and $m_2^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})||\mathsf{flag}^*)$.
10. Compute c_1^*, c_2^* as follows: $c_1^* = \mathsf{PKEnc}(\mathsf{pk}_1, m_1^*; r_1)$, $c_2^* = \mathsf{PKEnc}(\mathsf{pk}_2, m_2^*; r_2)$.
11. Pick $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$.
12. Compute $\pi^* = \mathsf{S}_2(\mathsf{crs}, \tau, \mathsf{stmt})$.
13. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
14. Let $y^* = (r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*))$.
15. Let $P_{comp} = i\mathcal{O}(\text{Compute})$.
16. Output $(P_{comp}, (\mathsf{id}^*, c^*))$ and the queries of other honest users.

Compute

Constants: $(\mathsf{id}^*, c^*, y^*)$, Secret key sk_1, puncturable PRF key $K(\{(\mathsf{id}^*, c^*)\})$, verification key avk and common reference string crs.
Input: Identity id and ciphertext $c = (c_1, c_2, \pi)$.

1. If $(\mathsf{id}, c) = (\mathsf{id}^*, c^*)$ output y^* and end.
2. If $\mathsf{NIZKVerify}(\mathsf{crs}, (c_1, c_2), \pi) = 0$, output $\perp$ and end.
3. Let $(\mathsf{id}'||x||t_{\mathsf{id}}||K_{\mathsf{id},\mathsf{out}}||\mathsf{flag}) = \mathsf{PKDec}(\mathsf{sk}_1, c_1)$. If $\mathsf{id} \neq \mathsf{id}'$, output $\perp$.
4. Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}, c))$.
5. If $\mathsf{flag} = 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r))$ and end.
6. Else if $\mathsf{flag} = 0$, and $\mathsf{Authverify}(\mathsf{avk}, \mathsf{id}, t_{\mathsf{id}}) = 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x))$ and end.
7. Output $y = \perp$ and end.

Fig. 8: Program Compute

$\mathsf{Hyb}_{i,5}$. In this hybrid, we change the value of flag^* to 1 instead of 0 and t_{id^*} to be a random string of appropriate length. We also set $x^* = 0^{\ell_{\mathsf{in}}}$. We also change back the key $K_{\mathsf{id}^*,\mathsf{out}}$ used in m_1^* and m_2^* to the original unpunctured key.

The indistinguishability follows via a sequence of hybrids using the two-key switching techniques. Note that here we crucially use that the fact that the program in the previous hybrid does not use x^* in computing the output on input c^*. Moreover, because of the initial "if" condition, there is no check on flag^* or t_{id^*}. Hence, while switching keys for decryption, functional equivalence follows in a straight-forward manner.

1. Choose PRF key K at random for PRF_1.
2. Pick $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{PKGen}(1^\lambda)$, $(\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow \mathsf{PKGen}(1^\lambda)$.
3. Pick $(\mathsf{avk}, \mathsf{ask}) \leftarrow \mathsf{AuthGen}(1^\lambda)$ with respect to cover-free set system $\mathcal{T}$ and PRG.
4. On receiving a corrupt user's identity id, return the authentication as in real execution. That is, compute $t_{\mathsf{id}} \leftarrow \mathsf{AuthProve}(\mathsf{ask}, \mathsf{id})$ and send $(t_{\mathsf{id}}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs})$.
5. Pick t_{id^*} to be a uniformly random string of appropriate length.
6. Set $\mathsf{flag}^* = 1$.
7. Let r^* be chosen randomly.
8. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for PRF_2.
9. Set $x^* = 0^{\ell_{\mathsf{in}}}$. Let $m_1^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}||\mathsf{flag}^*)$ and $m_2^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}||\mathsf{flag}^*)$.
10. Compute c_1^*, c_2^* as follows: $c_1^* = \mathsf{PKEnc}(\mathsf{pk}_1, m_1^*; r_1)$, $c_2^* = \mathsf{PKEnc}(\mathsf{pk}_2, m_2^*; r_2)$.
11. Pick $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$.
12. Compute $\pi^* = \mathsf{S}_2(\mathsf{crs}, \tau, \mathsf{stmt})$.
13. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
14. Let $y^* = (r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*))$.
15. Let $P_{comp} = i\mathcal{O}(\text{Compute})$.
16. Output $(P_{comp}, (\mathsf{id}^*, c^*))$ and the queries of other honest users.

Compute

Constants: $(\mathsf{id}^*, c^*, y^*)$, Secret key sk_1, puncturable PRF key $K(\{(\mathsf{id}^*, c^*)\})$, verification key avk and common reference string crs.
Input: Identity id and ciphertext $c = (c_1, c_2, \pi)$.

1. If $(\mathsf{id}, c) = (\mathsf{id}^*, c^*)$ output y^* and end.
2. If $\mathsf{NIZKVerify}(\mathsf{crs}, (c_1, c_2), \pi) = 0$, output $\perp$ and end.
3. Let $(\mathsf{id}'||x||t_{\mathsf{id}}||K_{\mathsf{id},\mathsf{out}}||\mathsf{flag}) = \mathsf{PKDec}(\mathsf{sk}_1, c_1)$ for appropriate length strings. If $\mathsf{id} \neq \mathsf{id}'$, output $\perp$ and end.
4. Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}, c))$.
5. If $\mathsf{flag} = 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r))$ and end.
6. Else if $\mathsf{flag} = 0$, and $\mathsf{Authverify}(\mathsf{avk}, \mathsf{id}, t_{\mathsf{id}}) = 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x))$ and end.
7. Output $y = \perp$ and end.

Fig. 9: Program Compute

$\mathsf{Hyb}_{i:6}$. In this sequence of hybrids, we revert back some of the changes we made. First, we again start generating the crs and the proof π^* honestly. The indistinguishability follows from the computational zero-knowledge property of the NIZK used.

Next, we set $r^* = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*))$ instead of random. The indistinguishability follows from the pseudorandomness property of the punctured PRF PRF_1.

Finally, we remove the initial "if" condition and the constants $(\mathsf{id}^*, c^*, y^*)$, and unpuncture the key K. The indistinguishability follows from the security of $i\mathcal{O}$.

1. Choose PRF key K at random for PRF_1.
2. Pick $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{PKGen}(1^\lambda)$, $(\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow \mathsf{PKGen}(1^\lambda)$.
3. Pick $(\mathsf{avk}, \mathsf{ask}) \leftarrow \mathsf{AuthGen}(1^\lambda)$ with respect to cover-free set system $\mathcal{T}$ and PRG.
4. On receiving a corrupt user's identity id, return the authentication as in real execution. That is, compute $t_{\mathsf{id}} \leftarrow \mathsf{AuthProve}(\mathsf{ask}, \mathsf{id})$ and send $(t_{\mathsf{id}}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs})$.
5. Pick t_{id^*} to be a uniformly random string of appropriate length.
6. Set $\mathsf{flag}^* = 1$.
7. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for PRF_2.
8. Set $x^* = 0^{\ell_{\mathsf{in}}}$. Let $m_1^* = (\mathsf{id}^* || x^* || t_{\mathsf{id}^*} || K_{\mathsf{id}^*,\mathsf{out}} || \mathsf{flag}^*)$ and $m_2^* = (\mathsf{id}^* || x^* || t_{\mathsf{id}^*} || K_{\mathsf{id}^*,\mathsf{out}} || \mathsf{flag}^*)$.
9. Compute c_1^*, c_2^* as follows: $c_1^* = \mathsf{PKEnc}(\mathsf{pk}_1, m_1^*; r_1)$, $c_2^* = \mathsf{PKEnc}(\mathsf{pk}_2, m_2^*; r_2)$.
10. Pick $\mathsf{crs} \leftarrow \mathsf{NIZKSetup}(1^\lambda)$.
11. Compute $\pi^* = \mathsf{NIZKProve}(\mathsf{crs}, \mathsf{stmt}, (m_1^*, r_1, r_2))$.
12. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
13. Let $r^* = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*))$.
14. Let $y^* = (r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*))$.
15. Let $P_{comp} = i\mathcal{O}(\text{Compute})$.
16. Output $(P_{comp}, (\mathsf{id}^*, c^*))$ and the queries of other honest users.

Compute

Constants: Secret key sk_1, puncturable PRF key K, verification key avk and common reference string crs.
Input: Identity id and ciphertext $c = (c_1, c_2, \pi)$.

1. If $\mathsf{NIZKVerify}(\mathsf{crs}, (c_1, c_2), \pi) = 0$, output $\perp$ and end.
2. Let $(\mathsf{id}' || x || t_{\mathsf{id}} || K_{\mathsf{id},\mathsf{out}} || \mathsf{flag}) = \mathsf{PKDec}(\mathsf{sk}_1, c_1)$. If $\mathsf{id} \neq \mathsf{id}'$, output $\perp$.
3. Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}, c))$.
4. If $\mathsf{flag} = 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r))$ and end.
5. Else if $\mathsf{flag} = 0$, and $\mathsf{Authverify}(\mathsf{avk}, \mathsf{id}, t_{\mathsf{id}}) = 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x))$ and end.
6. Output $y = \perp$ and end.

Fig. 10: Program Compute

In this sequence of hybrids described above, we have shown that the view of the adversary is indistinguishable in the following two scenarios: 1) The honest user en-

codes his actual input x with $\mathsf{flag} = 0$ and a valid authentication t_{id}, and obtains output according to the function F on (id, x). 2) The honest user encodes $0^{\ell_{\mathsf{in}}}$ with $\mathsf{flag} = 1$ and uniformly random t_{id}, and receives encoding of 0 as output (without invoking the function F.)

Below we write the final hybrid obtained above as Hyb_1 as follows:

Hyb_1*:* This hybrid is same as $\mathsf{Hyb}_{m:6}$. In the hybrid $\mathsf{Hyb}_{m:6}$, all the user queries will have $\mathsf{flag} = 1$, t_{id} will be a random string, and input will be 0^{ℓ}_{in}. Hence, the program Compute will not invoke the function F for any of the honest users.

The underlined statement summarizes the main difference between Hyb_0 and Hyb_1. Since the program being obfuscated does not change between Hyb_0 and Hyb_1, we omit its description from here.

1. Choose PRF key K at random for PRF_1.
2. Pick $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{PKGen}(1^\lambda)$, $(\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow \mathsf{PKGen}(1^\lambda)$.
3. Pick $(\mathsf{avk}, \mathsf{ask}) \leftarrow \mathsf{AuthGen}(1^\lambda)$ with respect to cover-free set system $\mathcal{T}$ and PRG.
4. Pick $\mathsf{crs} \leftarrow \mathsf{NIZKSetup}(1^\lambda)$.
5. Let $P_{comp} = i\mathcal{O}(\text{Compute})$.
6. On receiving a corrupt user's identity id, return the authentication as in real execution. That is, compute $t_{\mathsf{id}} \leftarrow \mathsf{AuthProve}(\mathsf{ask}, \mathsf{id})$ and send $(t_{\mathsf{id}}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs})$ to the user.
7. For each of the honest users, t_{id} is set to a random string of appropriate length, flag is set to 1 and input is set to $0^{\ell_{\mathsf{in}}}$. Ciphertexts (c_1, c_2) and proof π are generated honestly.
8. Output P_{comp} and queries of all honest users $\mathsf{id}_1, \ldots, \mathsf{id}_m$.

Hyb_2*:* We change the setup phase of authentication scheme to use $\mathsf{FakeAuthGen}$ instead of $\mathsf{AuthGen}$. Let $\mathcal{I}$ denote the set of corrupt user identities. Note that $|\mathcal{I}| \leqslant k$ and set system $\mathcal{T}$ used in our scheme is a k-cover-free set system.

The two hybrids are indistinguishable by security properties of $\mathsf{FakeAuthGen}$ (see Section 2.5). Note that both hybrids do not depend on ask and need only the valid authentications for corrupt users.

1. Choose PRF key K at random for PRF_1.
2. Pick $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{PKGen}(1^\lambda)$, $(\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow \mathsf{PKGen}(1^\lambda)$.
3. Pick $(\mathsf{avk}, \mathsf{ask}) \leftarrow \mathsf{FakeAuthGen}(1^\lambda, \mathcal{I})$ w.r.t. cover-free set system $\mathcal{T}$ and PRG.
4. Pick $\mathsf{crs} \leftarrow \mathsf{NIZKSetup}(1^\lambda)$.
5. Let $P_{comp} = i\mathcal{O}(\text{Compute})$.
6. On receiving a corrupt user's identity id, return the authentication as in real execution. That is, compute $t_{\mathsf{id}} \leftarrow \mathsf{AuthProve}(\mathsf{ask}, \mathsf{id})$ and send $(t_{\mathsf{id}}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs})$. As noted before, $\mathsf{AuthProve}$ still returns valid authentication for all users in $\mathcal{I}$.
7. For each of the honest users, t_{id} is set to a random string of appropriate length and flag is set to 1. Ciphertexts (c_1, c_2) and proof π is generated as in real execution.
8. Output P_{comp} and queries of all honest users $\mathsf{id}_1, \ldots, \mathsf{id}_m$.

Hyb_3: This is the most important hybrid, where we change the program from F to F. The two hybrids are indistinguishable by security of $i\mathcal{O}$. Note that in both hybrids the function is invoked iff the authentication of the user verifies under avk. In the both hybrids, this can happen only for corrupt users as there is no valid authentication for honest users. Finally, recall that the functions F and G are equivalent for corrupt users.

1. Choose PRF key K at random for PRF_1.
2. Pick $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{PKGen}(1^\lambda)$, $(\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow \mathsf{PKGen}(1^\lambda)$.
3. Pick $(\mathsf{avk}, \mathsf{ask}) \leftarrow \mathsf{FakeAuthGen}(1^\lambda, \mathcal{I})$ w.r.t. cover-free set system $\mathcal{T}$ and PRG.
4. Pick $\mathsf{crs} \leftarrow \mathsf{NIZKSetup}(1^\lambda)$.
5. Let $P_{comp} = i\mathcal{O}(\text{Compute})$.
6. On receiving a corrupt user's identity id, return the authentication as in real execution. That is, compute $t_{\mathsf{id}} \leftarrow \mathsf{AuthProve}(\mathsf{ask}, \mathsf{id})$ and send $(t_{\mathsf{id}}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs})$.
7. For each of the honest users, t_{id} is set to a random string of appropriate length and flag is set to 1. Ciphertexts (c_1, c_2) and proof π is generated as in real execution.
8. Output P_{comp} and queries of all honest users $\mathsf{id}_1, \ldots, \mathsf{id}_m$.

Compute

Constants: Secret key sk_1, puncturable PRF key K, verification key avk and common reference string crs.
Input: Identity id and ciphertext $c = (c_1, c_2, \pi)$.

1. If $\mathsf{NIZKVerify}(\mathsf{crs}, (c_1, c_2), \pi) = 0$, output $\perp$ and end.
2. Let $(\mathsf{id}'||x||t_{\mathsf{id}}||K_{\mathsf{id},\mathsf{out}}||\mathsf{flag}) = \mathsf{PKDec}(\mathsf{sk}_1, c_1)$ for appropriate length strings. If $\mathsf{id} \neq \mathsf{id}'$, output $\perp$ and end.
3. Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}, c))$.
4. If $\mathsf{flag} = 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r))$ and end.
5. Else if $\mathsf{flag} = 0$, and $\mathsf{Authverify}(\mathsf{avk}, \mathsf{id}, t_{\mathsf{id}}) = 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{G}(\mathsf{id}, x))$ and end.
6. Output $y = \perp$ and end.

Fig. 11: Program Compute

Finally, using a similar sequence of hybrids we can move from Hyb_3 to a hybrid which corresponds to real execution using G and honest party inputs $x'_1, \ldots, x'_m$.

5 Our Secure Cloud Service Scheme with Cloud Inputs

In this section, we describe our modified scheme for service hosting on the cloud with cloud inputs. As before, we have three different parties: The provider who owns the service, the cloud where the service is hosted, and the users. Recall that we assume that the provider of the service is honest.

As before, let λ be the security parameter. Let k be the bound on the number of corrupt users.

Let $\mathcal{T}$ be a k-cover-free set system using a finite field $\mathbb{F}_q$ and polynomials of degee $d = (q-1)/k$ and $(\mathsf{AuthGen}, \mathsf{AuthProve}, \mathsf{Authverify})$ be the authentication scheme w.r.t. $\mathcal{T}$ as described in Section 2.5. As mentioned before, we use $q = k\lambda$, so that the number of sets/users is at least 2^λ. We interpret the user's identity id as the coefficients of a polynomial over $\mathbb{F}_q$ of degree at most d. Let the length of the identity be $\ell_{\mathsf{id}} := (d+1)\lg q$ and length of the authentication be ℓ_{auth}. Note that in our scheme $\ell_{\mathsf{auth}} = 2\lambda q$.

Let $\mathsf{pke} = (\mathsf{PKGen}, \mathsf{PKEnc}, \mathsf{PKDec})$ be public key encryption scheme which accepts messages of length $\ell_e = (\ell_{\mathsf{id}} + \ell_{\mathsf{in}} + \ell_{\mathsf{auth}} + \ell_{\mathsf{kout}} + 1)$ and returns ciphertexts of length ℓ_c. Here ℓ_{in} is the length of the input of the user and ℓ_{kout} is the length of the key for PRF_2 described below.

Let $(\mathsf{NIZKSetup}, \mathsf{NIZKProve}, \mathsf{NIZKVerify})$ be the statistical simulation-sound non-interactive zero-knowledge proof system with simulator $(\mathsf{S}_1, \mathsf{S}_2)$. In our scheme we use the two-key paradigm along with statistically simulation-sound non-interactive zero-knowledge for non-malleability inspired from [15,17,6].

We will make use of two different family of puncturable PRFs. a) $\mathsf{PRF}_1(K, \cdot)$ that accepts inputs of length $(\ell_{\mathsf{id}} + \ell_c + \ell_z)$ and returns strings of length $\ell_r \geqslant (\ell_{\mathsf{id}} + \ell_c + \ell_z) + \lambda$. Here ℓ_z is the length of the cloud's input z. b) $\mathsf{PRF}_2(K_{\mathsf{id}}, \cdot)$ that accepts inputs of length ℓ_r and returns strings of length ℓ_{out}, where ℓ_{out} is the length of the output of program. Such PRFs exist by Theorem 3.

We put a lower bound on the length of output of PRF_1 because in the proof we would require that a random string of length ℓ_r does not lie in the image of $\mathsf{PRF}_1(K, \cdot)$.

Scheme Description. Consider an honest provider $\mathcal{H}$ who holds a function F which he wants to hosts on the cloud $\mathcal{C}$. Also, there will be a collection of users who will interact with the provider to obtain authentication which will enable them to run the program stored on the cloud. The provider does the following:

1. Chooses PRF key K at random for PRF_1.
2. Picks $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{PKGen}(1^\lambda)$, $(\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow \mathsf{PKGen}(1^\lambda)$.
3. Picks $(\mathsf{avk}, \mathsf{ask}) \leftarrow \mathsf{AuthGen}(1^\lambda)$ with respect to k-cover-free set system $\mathcal{T}$ and pseudorandom generator PRG.
4. Picks $\mathsf{crs} \leftarrow \mathsf{NIZKSetup}(1^\lambda)$.
5. Creates an indistinguishability obfuscation $P_{comp} = i\mathcal{O}(\text{Compute})$, where Compute is the same program as in Figure 1 with the following change: It takes an additional input z from the cloud along with identity id and ciphertext $c = (c_1, c_2, \pi)$ from the user. And while computing the output in Step 5 when $\mathsf{flag} = 0$ and user is authenticated as $y = (r, \mathsf{PRF}_2(K_{\mathsf{id,out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ (using the cloud input z).

   Here $\tilde{F} = P_{comp}$ and $\sigma = (\mathsf{ask}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs})$.

Next, we describe the procedures $\mathsf{SCSS.auth}$, $\mathsf{SCSS.inp}$ and $\mathsf{SCSS.eval}$.

Procedure $\mathsf{SCSS.auth}$: It takes as input user's id and secret state of the service provider $\sigma = (\mathsf{ask}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs}))$. The provider computes $t_{\mathsf{id}} = \mathsf{AuthProve}(\mathsf{ask}, \mathsf{id})$ and sends back $\mathsf{auth}_{\mathsf{id}} = (t_{\mathsf{id}}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs})$.

Procedure $\mathsf{SCSS.inp}(1^\lambda, \mathsf{auth}_{\mathsf{id}} = (t_{\mathsf{id}}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs}), x)$: The user chooses a key $K_{\mathsf{id,out}}$ for PRF_2. Let $m = (\mathsf{id}||x||t_{\mathsf{id}}||K_{\mathsf{id,out}}||0)$. It then computes $c_1 = \mathsf{PKEnc}(\mathsf{pk}_1, m; r_1)$,

$c_2 = \mathsf{PKEnc}(\mathsf{pk}_2, m; r_2)$ and a SSS-NIZK proof π for

$$\exists m, t_1, t_2 \text{ s.t. } (c_1 = \mathsf{PKEnc}(\mathsf{pk}_1, m; t_1) \wedge c_2 = \mathsf{PKEnc}(\mathsf{pk}_2, m; t_2))$$

It outputs $\tilde{x} = (\mathsf{id}, c = (c_1, c_2, \pi))$ and $\alpha = K_{\mathsf{id},\mathsf{out}}$.

Procedure $\mathsf{SCSS.eval}(\tilde{F} = P_{comp}, \tilde{x}, z)$: Let the cloud's input be z. Run $\tilde{F}$ on $(\tilde{x}, z)$ to obtain $\tilde{y}$. The user parses $\tilde{y}$ as $\tilde{y}_1, \tilde{y}_2$ and computes $y = \mathsf{PRF}_2(\alpha, \tilde{y}_1) \oplus \tilde{y}_2$.

Security Proof. For a formal proof that our scheme satisfies Theorem 2 i.e. it is a secure cloud service scheme with cloud inputs please refer to our full version.

References

1. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S.P., Yang, K.: On the (im)possibility of obfuscating programs. In: CRYPTO. pp. 1–18 (2001)
2. Boneh, D., Waters, B.: Constrained pseudorandom functions and their applications. In: ASIACRYPT. pp. 280–300 (2013)
3. Boyle, E., Goldwasser, S., Ivan, I.: Functional signatures and pseudorandom functions. In: PKC. pp. 501–519 (2014)
4. Erdös, P., Frankl, P., Füredi, Z.: Families of finite sets in which no set is covered by the union of r others. Israel Journal of Mathematics 51(1-2), 79–89 (1985)
5. Feige, U., Lapidot, D., Shamir, A.: Multiple noninteractive zero knowledge proofs under general assumptions. SIAM J. Comput. 29(1), 1–28 (1999)
6. Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B.: Candidate indistinguishability obfuscation and functional encryption for all circuits. In: FOCS. pp. 40–49 (2013)
7. Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In: CRYPTO. pp. 465–482 (2010)
8. Gentry, C., Halevi, S., Raykova, M., Wichs, D.: Outsourcing private ram computation. Cryptology ePrint Archive, Report 2014/148 (2014), http://eprint.iacr.org/
9. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions (extended abstract). In: FOCS. pp. 464–479 (1984)
10. Goldwasser, S., Gordon, S.D., Goyal, V., Jain, A., Katz, J., Liu, F., Sahai, A., Shi, E., Zhou, H.: Multi-input functional encryption. In: EUROCRYPT. pp. 578–602 (2014)
11. Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: Delegating computation: interactive proofs for muggles. In: STOC. pp. 113–122 (2008)
12. Kiayias, A., Papadopoulos, S., Triandopoulos, N., Zacharias, T.: Delegatable pseudorandom functions and applications. In: CCS. pp. 669–684 (2013)
13. Kumar, R., Rajagopalan, S., Sahai, A.: Coding constructions for blacklisting problems without computational assumptions. In: CRYPTO. pp. 609–623 (1999)
14. Micali, S.: CS proofs (extended abstracts). In: 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994. pp. 436–453 (1994)
15. Naor, M., Yung, M.: Public-key cryptosystems provably secure against chosen ciphertext attacks. In: STOC. pp. 427–437 (1990)
16. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: Nearly practical verifiable computation. In: SP. pp. 238–252 (2013)
17. Sahai, A.: Non-malleable non-interactive zero knowledge and adaptive chosen-ciphertext security. In: FOCS. pp. 543–553 (1999)
18. Sahai, A., Waters, B.: How to use indistinguishability obfuscation: deniable encryption, and more. In: STOC. pp. 475–484 (2014)