UNIVERSITY OF CALIFORNIA
Los Angeles

# Program Obfuscation: Applications and Optimizations

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Divya Gupta

2016

ABSTRACT OF THE DISSERTATION

# Program Obfuscation: Applications and Optimizations

by

Divya Gupta

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2016

Professor Amit Sahai, Chair

Recently, candidate constructions were given for indistinguishability obfuscation by Garg, Gentry, Halevi, Raykova, Sahai, and Waters [FOCS'13]. The goal of general-purpose program obfuscation is to make an arbitrary computer program "unintelligible" while preserving its functionality. Since then, this new tool has been used to obtain many new applications. In this dissertation, we study the problem of protecting software against new and powerful adversary structures as well as optimize the efficiency of secure general-purpose obfuscation schemes.

In the first part of this dissertation, we initiate the study of hosting services on an untrusted cloud. Specifically, 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. 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. We provide the first formalizations of security for this setting, and construction relying on indistinguishability obfuscation.

The great interest in the utility of obfuscation leads to a natural and pressing goal: to improve the efficiency of general-purpose obfuscation. In the second part of this dissertation, we initiate the work to optimize the efficiency of secure general-purpose obfuscation schemes. We focus on the problem of optimizing the obfuscation of Boolean formulas and branching programs - this corre-

sponds to optimizing the "core obfuscator" from the work of Garg et al., and all subsequent works constructing general-purpose obfuscators. Our efficiency improvement is obtained by generalizing the class of branching programs that can be directly obfuscated. This generalization allows us to achieve a simple simulation of formulas by branching programs while avoiding the use of Barrington's theorem, on which all previous constructions relied. Furthermore, the ability to directly obfuscate general branching programs (without bootstrapping) allows us to efficiently apply our construction to natural function classes that are not known to have polynomial-size formulas.

The dissertation of Divya Gupta is approved.

Suhas Diggavi

Rafail Ostrovsky

Alexander Sherstov

Amit Sahai, Committee Chair

University of California, Los Angeles

2016

*To my parents . . .*

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# ACKNOWLEDGMENTS

This dissertation would not have been possible without the support and encouragement of many people, and here is an attempt to express my gratitude.

I am extremely grateful to my advisor Amit Sahai. He is the one who introduced me to the field of cryptography and helped me develop expertise in it. I would like to thank him for spending a countless number of hours discussing intriguing questions in cryptography as well as the philosophies of life. Discussions with him have always been enlightening, have helped me get clarity in my thoughts, and shaped my personality as a researcher.

I would like to thank Rafail Ostrovsky, Suhas Diggavi and Alexander Sherstov for being on my dissertation committee and providing encouragement.

I thank Manoj Prabhakaran and Vipul Goyal for hosting me for summer internships and helping me diversify my research interests. Being at UCLA has provided me with the opportunity to have many intellectual discussions with Yuval Ishai that have also resulted in many fruitful collaborations. In all of these discussions, I have been stunned by the breadth of his knowledge.

I would like to thank Hemanta K. Maji for being an excellent collaborator, the go-to person for professional advice, as well as a constant source of entertainment. I thank my collaborators Dan Boneh, Ilya Mironov, Sanjam Garg, Abhishek Jain, Omkant Pandey, Prabhanjan Ananth, Shashank Agrawal and Saikrishna Badrinarayan. I am grateful to my official mentor Sanjam Garg and seniors in the lab Ran Gelles, Abishek Kumarasubhramanian, Abhishek Jain, Akshay Wadia and Vanishree Rao for patiently answering all my questions about cryptography and UCLA in general.

Finally, I would like to thank Rahul Sharma who has been a constant source of constructive criticism, my friends Anuraag Gupta, Esha Aggarwal, Bharath Hariharan and Saurabh Gupta, and my parents who helped me retain some sanity during the roller coaster journey of grad school.

# VITA

| | |
|---|---|
| 2006–2011 | B.Tech. and M.Tech. in Computer Science and Engineering, Indian Institute of Technology, Delhi. |
| 2011 | Silver Medal for Department Rank 1 in Computer Science Dual Degree, Indian Institute of Technology, Delhi. |
| 2011–2012 | Computer Science Department Fellowship, UCLA. |
| 2011–2016 | Ph.D. Student, Computer Science Department, UCLA. |
| 2015–2016 | Dissertation Year Fellowship, UCLA. |

# PUBLICATIONS

Divesh Aggarwal, Shashank Agrawal, Divya Gupta, Hemanta K. Maji, Omkant Pandey, and Manoj Prabhakaran, "Optimal Computational Split-state Non-malleable Codes", In *TCC 2016-A*

Saikrishna Badrinarayanan, Divya Gupta, Abhishek Jain, and Amit Sahai, "Multi-input Functional Encryption for Unbounded Arity Functions", In *ASIACRYPT 2015*

Vipul Goyal, Divya Gupta, and Amit Sahai, "Concurrent Secure Computation via Non-Black Box Simulation", In *CRYPTO 2015*

Shashank Agrawal, Divya Gupta, Hemanta K. Maji, Omkant Pandey, and Manoj Prabhakaran, "Explicit Non-malleable Codes Against Bit-Wise Tampering and Permutations", In *CRYPTO 2015*

Divya Gupta, Yuval Ishai, Hemanta K. Maji, and Amit Sahai, "Secure Computation from Leaky Correlated Randomness", In *CRYPTO 2015*

Dan Boneh, Divya Gupta, Ilya Mironov, and Amit Sahai, "Hosting Services on an Untrusted Cloud", In *EUROCRYPT 2015*

Shashank Agrawal, Divya Gupta, Hemanta K. Maji, Omkant Pandey, and Manoj Prabhakaran, "A Rate-Optimizing Compiler for Non-malleable Codes Against Bit-Wise Tampering and Permutations", In *TCC 2015*

Prabhanjan Vijendra Ananth, Divya Gupta, Yuval Ishai, and Amit Sahai, "Optimizing Obfuscation: Avoiding Barrington's Theorem", In *CCS 2014*

Sanjam Garg and Divya Gupta, "Efficient Round Optimal Blind Signatures", In *EUROCRYPT 2014*

Divya Gupta and Amit Sahai, "On Constant-Round Concurrent Zero-Knowledge from a Knowledge Assumption", In *INDOCRYPT 2014*

Vipul Goyal, Divya Gupta, and Abhishek Jain, "What Information Is Leaked under Concurrent Composition?", In *CRYPTO 2013*

Khaled M. Elbassioni, Naveen Garg, Divya Gupta, Amit Kumar, Vishal Narula, and Arindam Pal, "Approximation Algorithms for the Unsplittable Flow Problem on Paths and Trees", In *FSTTCS 2012*

# CHAPTER 1

# Introduction

The goal of general-purpose program obfuscation is to make an arbitrary computer program "unintelligible" while preserving its functionality. Obfuscation allows us to achieve a powerful capability: software that can keep a secret. That is, software that makes use of secrets to perform its computations, but with the additional property that these secrets remain secure even if the code of the software is captured in its entirety by an adversary. At least as far back as the work of Diffie and Hellman in 1976 [DH76][1], researchers have contemplated applications of general-purpose obfuscation. Indeed, if secure general-purpose obfuscation could be cryptographically achieved *efficiently*, the implications to computer security would be profound [BGI01].

To understand why obfuscation can be so useful, it is instructive to contemplate what kinds of secrets we might want to hide within our software code. An important instance of such secrets is hiding the existence and nature of *rare input/output behavior* that our software may exhibit. This leads to several interesting motivating scenarios:

- Our software may be a control algorithm that is programmed to enter a failsafe mode on certain rare and hard-to-predict inputs. We would not want an adversary that gains access to the code of the control software to be able to learn these rare inputs. By securely obfuscating the control software, the existence of the failsafe mode itself would be hidden from the adversary.

- We may *modify* software to introduce such rare input/output behavior to suit our goals. Consider the problem of software watermarking, where we want to add an undetectable imprint to our software that we can later identify. We may do so by modifying the behavior of

[1] Diffie and Hellman suggested the use of general-purpose obfuscation to convert private-key cryptosystems to public-key cryptosystems.

our software, so that on several rare and hard-to-predict inputs, it outputs a watermark code instead of performing its usual computation. An obfuscated version of this modified software would hide the existence of these imprints, and thereby also prevent an adversary from removing them unless the adversary rewrites from scratch almost all of the software.

- So far our examples have dealt with hiding known rare input/output behavior. But obfuscation could also be used to hide the existence of *unknown* and unintentional rare input/output behavior: Consider software bugs that are particularly resistant to good-faith software testing, because the input/output behavior that is affected by these bugs only arises from inputs that are rare and hard to predict given only the functionality of the software. Then, obfuscation can be used to hide the existence of such software bugs (and the vulnerabilities they introduce), even from an attacker that has the code of the software.

- Finally, turning the previous example around, obfuscation can also be used to hide which of these software bugs are being fixed by a software patch, thereby preventing adversaries from learning vulnerabilities from software patches and using this knowledge to attack unpatched software.

As these motivating scenarios illustrate, secure obfuscation would greatly expand the scope of security problems addressable through cryptographic means. However, efficient and secure obfuscation would also have powerful applications to data security, specifically to protecting against data breaches by low-level insiders. Low-level insiders can cause data breaches if they go rogue, or if their computing systems are compromised through theft or malware attack. As a result, a critical problem arises when such insiders hold decryption keys – indeed even low-level insiders may need such keys to perform basic functions. For example, an employee tasked with generating summaries of financial statistics may need decryption keys in order to decrypt sensitive financial spreadsheets. If this decryption key is captured by an adversary, however, it can be used to steal vast quantities of sensitive information, even though the decryption key was only meant to allow the insider to generate low-value statistical summaries. Obfuscation, however, provides a powerful solution to this problem: The decryption keys can be safely hidden within the statistical summary generation software that is entrusted to the low-level insider. Then, even if the insider turns rogue,

the only power he can derive from his software is the ability to generate statistics[2]; he cannot abuse his position to directly decrypt the underlying financial files.

The above examples provide only a fractional view of the applicability that efficient secure obfuscation would have to computer security. However, until 2013, even heuristic constructions for secure general-purpose obfuscation were not known.

This changed with the work of Garg, Gentry, Halevi, Raykova, Sahai, and Waters [GGH13b], which gave the first candidate cryptographic construction for a general-purpose obfuscator. Formal exploration of the applications of general-purpose obfuscation began shortly thereafter [GGH13b, SW14]. Since then, the floodgates have opened, and many new applications of general-purpose obfuscation have been explored [BCC14, BP15, MR13, BCP14, ABG13, MO14, BH15, GJK15, BST14, PPS15, GGH14b, GGG14, BBC14, BFM14, KNY14, GHR14, HSW14, BR14a, BR14b, GGH14a].

In this dissertation, we study the problem of protecting software against new and powerful adversary structures as well as optimize the efficiency of secure general-purpose obfuscation schemes.

In the first part of this dissertation, we initiate the study of hosting services on an untrusted cloud. 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. 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 construction are novel techniques to al-

[2]Of course, the statistical software itself must be carefully written to avoid vulnerabilities that allow a user to extract specific sensitive information by making unexpected statistical queries.

low 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.

In the second part of the dissertation, we seek to optimize the efficiency of secure general-purpose obfuscation schemes. We focus on the problem of optimizing the obfuscation of Boolean formulas and branching programs – this corresponds to optimizing the "core obfuscator" from the work of [GGH13b], and all subsequent works constructing general-purpose obfuscators. This core obfuscator builds upon approximate multilinear maps, where efficiency in proposed instantiations is closely tied to the maximum number of "levels" of multilinearity required. The most efficient previous construction of a core obfuscator, due to [BGK14], required the maximum number of levels of multilinearity to be $O(\ell s^{3.64})$, where $s$ is the size of the Boolean formula to be obfuscated, and $\ell$ is the number of input bits to the formula. In contrast, our construction only requires the maximum number of levels of multilinearity to be roughly $\ell s$. This results in significant improvements in both the total size of the obfuscation and the running time of evaluating an obfuscated formula. Our efficiency improvement is obtained by generalizing the class of branching programs that can be directly obfuscated. This generalization allows us to achieve a simple simulation of formulas by branching programs while avoiding the use of Barrington's theorem, on which all previous constructions relied. Furthermore, the ability to directly obfuscate general branching programs (without bootstrapping) allows us to efficiently apply our construction to natural function classes that are not known to have polynomial-size formulas.

## 1.1 Hosting Services on an Untrusted Cloud

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 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 $(\mathsf{id}, x)$ directly from a user with identity $\mathsf{id}$, 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 $\mathsf{id}$ performs some simple one-time set-up interaction with the service provider to obtain a key $K_{\mathsf{id}}$. This key $K_{\mathsf{id}}$ 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 construction 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, 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 [GHR14] 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 [GGG14] 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 [GGG14], 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 that would be 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 [GGG14], 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 [GGG14], which largely mirrors the definition of indistinguishability obfuscation security. We elaborate on these challenges below, when we discuss our techniques in greater detail.

**Our Results.** In this work, we formalize the notion of secure cloud service scheme (Section 2.2) 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 [GGH13b] and multi-input functional encryption [GGG14] 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 $2^\ell$ hybrids, where $\ell$ 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.*

**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 $K_{\mathsf{id}}$, 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 $\mathsf{id}$ and a ciphertext $c$. It then decrypts $c$ using $SK$ to yield $(x, K_{\mathsf{id}})$. 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 [BGI01]. 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 $K_{\mathsf{id}}$, 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 $\mathsf{id}$ 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 $\mathsf{id}^*$, 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 $\mathsf{id}$. 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 $\mathsf{id}^*$. 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 $\mathsf{id}^*$. 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 [EFF85, KRS99] 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 $K_{\mathsf{id}}$ 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 [SW14] 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 2.3 and proof in Section 2.3.1 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 [SW14] 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. The details of this approach are presented in Section 2.4.1.

## 1.2 Optimizing Obfuscation

This great interest in the utility of obfuscation leads to a natural and pressing goal: to improve the efficiency of general-purpose obfuscation. Up to this point, the simplest and most efficient proposed general-purpose obfuscator was given by [BGK14], building upon [GGH13b, BR14b]. However, the general-purpose obfuscator presented in [BGK14] (see below for more details) remains extremely inefficient.

Our work aims to initiate a systematic research program into improving the efficiency of general-purpose obfuscation. Tackling this important problem will no doubt be the subject of many works to come. We begin by recalling the two-stage approach to general-purpose obfuscation outlined in [GGH13b] and present in all subsequent work on constructing general-purpose

obfuscators:

1. At the heart of their construction is the "core obfuscator" for Boolean formulas (equivalently, $\mathbf{NC}^1$ circuits), building upon a simplified subset of the Approximate Multilinear Maps framework of Garg, Gentry, and Halevi [GGH13a] that they call Multilinear Jigsaw Puzzles. (We will defer discussion of security to later.)

2. Next, a way to bootstrap from the core obfuscator for Boolean formulas to general circuits is used. The works of [GGH13b, BR14b, BGK14] all adopt a method for bootstrapping using Fully Homomorphic Encryption. This bootstrapping method works provably with the security definition of indistinguishability obfuscation, and can rely on well-studied cryptographic assumptions such as the LWE assumption. Alternatively, the earlier work of Goyal et al. [GIS10] constructed a universal stateless hardware token for obfuscation that can be implemented by polynomial-size boolean formulas using a pseudorandom function in $\mathbf{NC}^1$. Applebaum [App14] gives a simpler alternative construction that has the disadvantage of requiring the size of the Boolean formulas to be polynomial in the input size and the security parameter (rather than only in the security parameter in [GIS10]). Using either of these alternative approaches [GIS10, App14], however, requires an ad-hoc (but arguably plausible) assumption to bootstrap from obfuscation for Boolean formulas to obfuscation for general circuits.

Our work focuses on improving the efficiency of the first of these steps: namely, the core obfuscator for Boolean formulas. We give one set of results for the setting of boolean formulas over the $\{\text{AND}, \text{NOT}, \text{OR}\}$-basis, and another set of results for general basis.

Previous constructions of a core obfuscator [GGH13b, BR14b, BGK14] first apply Barrington's theorem [Bar86] to convert the Boolean formula into an equivalent "matrix branching program," which is then obfuscated. Roughly speaking, a matrix branching program computes an iterated product of $n$ full-rank matrices, where each matrix in the product is determined by one of the input bits, and the result of the product should be either the identity matrix (corresponding to an output of 1) or some other fixed full-rank matrix (corresponding to an output of 0). The length of the program is $n$ and its width is the matrix dimension.

For any circuit or formula of depth $d$, Barrington's theorem gives a constant-width matrix branching program of length $4^d$. Since the length is exponential in the formula depth, it is crucial to balance the depth of the formula in order to avoid the exponential blowup. Hence, the first step would be to balance the formula to get a depth which is logarithmic in the size and then apply Barrington's theorem. For general formulas of size $s$, the best known depth obtained by balancing them is $1.73 \log s + d_0$ by Khrapchenko [Khr78, Juk12] where $d_0$ is a constant. However, the constant $d_0$ is quite large, which can have an adverse effect on concrete efficiency.[3] Instead, one can balance the formula using a method by Preparata and Muller [PM76]. The depth of the balanced formula obtained by this method is $1.82 \log s$. There have been other works which try to optimize the *size* of balanced formulas [BB94], but the depth of the formula obtained by these works is worse.

The matrix branching program obtained by applying Barrington's theorem to a formula of depth $1.82 \log s$ has length $s^{3.64}$. This is a major source of inefficiency. In particular, the bound of $s^{3.64}$ on the length of the branching program not only affects the number of elements given out as the final obfuscation, but also the number of levels of multilinearity required by the scheme. Since the size of each multilinear encoding grows with the number of levels of multilinearity required in known realizations of approximate multilinear maps [GGH13a, CLT13], this greatly affects the size of the final obfuscated program and also the evaluation time. Hence, in order to optimize the size of obfuscation it is critical to find an alternative approach.

**Our Contributions.** In our work, we posit an alternative strategy for obfuscation that avoids Barrington's theorem, as well as the need to balance Boolean formulas at all. In fact, this strategy can be efficiently applied to general (deterministic or even non-deterministic) *branching programs*, which are not known to be simulated by polynomial-size formulas. Our strategy employs variants of randomization techniques that were used in the context of secure multiparty computation [FKN94, CFI03], adapting them to the setting of obfuscation.

A crucial first step is to formulate a notion of a "relaxed matrix branching program" (RMBP) which relaxes some of the requirements of matrix branching programs needed in [GGH13b, BR14b,

[3] Note that once we apply Barrington's theorem, $d_0$ goes into the exponent and hence the size of the resulting obfuscation scheme will incur a factor of $4^{d_0}$.

BGK14]. The relaxation replaces permutation matrices by general full-rank matrices over a finite field and, more importantly, determines the output by testing whether some fixed entry in the matrix product is nonzero. (See Section 3.1.4 for a formal definition.) We show how to adapt the construction and security proofs of [BGK14] to work with RMBPs. The efficiency of this obfuscation will be discussed in more detail below. Roughly speaking, given the efficiency of current candidate multilinear encodings, the complexity of obfuscating RMBPs grows quadratically with the width and cubicly with the length. For now, we will measure efficiency in terms of the length and width of the RMBP.

Armed with the ability to obfuscate RMBPs, we look for simple and efficient ways to convert Boolean formulas and traditional types of branching programs into RMBPs without invoking Barrington's theorem. For this, we can use a previous transformation implicit in [FKN94] towards converting any ordinary graph-based non-deterministic branching program[4] of size $s$ into an RMBP of length $s$ and width $2(s+1)$. We also provide more efficient variants of this transformation that apply to classes of layered branching programs that satisfy certain technical conditions and arise in natural applications.

The above is already enough for efficiently obfuscating functions that are represented by small branching programs. However, in many cases functions are more naturally represented by Boolean formulas. In order to efficiently obfuscate formulas, we turn to the (abundant) literature on simulating formulas by branching programs. In the case of formulas consisting of only AND, OR, and NOT gates, we can use a simple transformation of any such formula of size $s$ into a branching program of the same size (cf. Theorem 6 in [Mas76] and Section 3.2.1.1).

The above simple transformation is limited in that it does not directly apply to formulas with XOR gates, and even without such gates its efficiency leaves much to be desired. Concretely, a formula of size $s$ is transformed into an RMBP whose length and width are roughly $s$ and $2s$, respectively, leading to a total of $O(s^3)$ matrix elements. To get around both limitations we rely on the work of Giel [Gie01], which builds on previous results of [BB94, Cle90] to efficiently transform a formula over the full basis to a layered branching program of constant width. The layered

[4] A non-deterministic branching program is a standard computational model that corresponds to non-deterministic logarithmic space. Such branching programs are believed to be strictly stronger than deterministic branching programs and formulas (see below) and strictly weaker than general circuits. See Section 3.1.3 for a formal definition.

branching program described in [Gie01] satisfies our conditions and can be used to obfuscate formulas over the full set of binary gates. Concretely, a formula of size $s$ can be transformed into an RMBP of length $O(s^{1+\epsilon})$, for an arbitrarily small constant $\epsilon > 0$, and constant width (depending only on $\epsilon$).

As in previous obfuscation techniques [GGH13b, BR14b, BGK14], a direct application of the above methods reveals the order in which input variables are read. Thus, to obfuscate a class of branching programs or formulas which may read the inputs in a varying order, we (as well as previous works) need to apply an additional step to make the RMBP family *input-oblivious*. This incurs an additional multiplicative overhead of $\ell$ to the length. However, this step and the resulting overhead can be avoided when the RMBP family is already input-oblivious. This is guaranteed in the useful case of obfuscating a class of *keyed* functions, namely a class of functions of the form $f_z(x) = \phi(z, x)$ where $\phi$ is a publicly known branching program or formula of size $s$. In other words, the goal is to obfuscate the class $\phi(z, \cdot)$ to hide the key $z$. In this case, an RMBP for $\phi$ can be easily turned into an input-oblivious family of RMBPs for the class $f_z$ with no additional overhead.

**Efficiency comparison.** We now quantify the efficiency improvements we obtain over previous work; we will do so both asymptotically and with explicit numbers through examples. The efficiency of our obfuscation scheme can be compared to previous ones by considering (1) the level $\kappa$ of the multilinear encoding being employed, and (2) the number $S$ of encoded field elements. The parameter $\kappa$ is of special importance, as the bit-length of each encoded element in current multilinear encoding candidates [GGH13a, CLT13] grows quadratically with $\kappa$. Thus, a good estimate for the total size (in bits) of an obfuscated program is $\tilde{O}(\kappa^2 \cdot S)$, where $\tilde{O}$ hides a multiplicative factor which depends polynomially on the security parameter. Moreover, our constructions (as well as previous ones) can be implemented so that the running time required for evaluating the obfuscated program is quasi-linear in the obfuscation size. Thus, from here on we will not explicitly refer to the asymptotic running time.

The concrete cost of implementing optimized multilinear encoding candidates is a subject of much ongoing research [CLT13, LSS14, LS14], and as of the time of this writing, explicit running

time and size estimates for multilinear candidates optimized for obfuscation[5] are not available for the $\kappa$ values that we need. However, as research in this direction is still in its infancy, it is reasonable to expect major improvements in the near future. For this reason, we do not attempt to provide real-life running time estimates, but rather compare our constructions with previous ones by considering the parameters $\kappa$ and $S$ described above.

The obfuscation methods from [GGH13b, BR14b, BGK14], when applied to a (strict) matrix branching program of length $n$ and width $w$ (one whose evaluation involves the product of $n$ matrices of size $w \times w$) requires $\kappa = n$ levels of multilinearity and $S = w^2 n$ encoded elements. The same holds for our method when applied to an RMBP of length $n$ and width $w$. Our simple and direct transformation for a (keyed) formula of size $s$ over the standard basis yields an RMBP of length $n \approx s$ and width $w \approx 2s$. This should be compared with the previous Barrington-based solution combined with the best known formula balancing results, leading to a matrix branching program with parameters $n = O(s^{3.64})$ and $w = O(1)$. Thus, under the quadratic cost assumption mentioned above, the obfuscation size is improved from $\tilde{O}(\kappa^2 \cdot S) = \tilde{O}(s^{10.92})$ to $\tilde{O}(s^5)$. (For the case of a completely balanced formula, the obfuscation size of the previous method is reduced to $\tilde{O}(s^6)$.) By further applying the result from [Gie01], we can obfuscate formulas over a full basis while reducing the total size to $\tilde{O}(s^{3(1+\epsilon)})$. See Table 1.1 for a detailed summary of old and new results for obfuscating formulas.

We note that even if future implementations of multilinear maps achieve an encoding size that only grows linearly with $\kappa$, our results would still yield significant improvements. (An encoding size that grows sublinearly with $\kappa$ seems out of reach with current lattice-based methods, due to error growth.)

Finally, to the best of our knowledge, it is not known how to simulate general branching programs (even deterministic ones) by strict (i.e., non-relaxed) matrix branching programs with a polynomial overhead. Thus, for the purpose of obfuscating branching programs without the use of bootstrapping, our method provides a *super-polynomial* efficiency improvement over previous core obfuscators. See Table 1.2 for a detailed summary of old and new results for obfuscating

[5] We note that obfuscation only requires Multilinear Jigsaw Puzzles [GGH13b], a strict subset of the full multilinear map functionality, which allows for substantial efficiency improvements in implementations. However, as of this writing, no experimental study of Multilinear Jigsaw Puzzle implementations has been completed.

branching programs.

**Examples.** We illustrate our concrete efficiency gains by two examples. The first example is motivated by the goal of obfuscating a psudorandom function (PRF) and deals with a conjectural PRF. As discussed above, PRFs can be used to bootstrap general obfuscation [GIS10, App14]. While practical PRF candidates such as AES are not known to have small formulas or branching programs, it seems plausible that there are good PRF candidates with relatively small formulas or layered non-deterministic branching programs. Suppose that a PRF family $f_z : \{0,1\}^{100} \to \{0,1\}$ can be implemented by a layered, invertible non-deterministic branching program of length 300 and width 30 (see Section 3.1.3 for definition). Obfuscating such a PRF family using our methods would require roughly 270,000 encoded field elements, with multilinearity $\kappa \approx 300$. In contrast, obfuscating such a PRF with previous approaches would require one to decide reachability in a layered graph of length 300 and width 30. The latter can be done using at least $\lceil \log_2 300 \rceil = 9$ levels of recursion, each implemented by a circuit of depth 6, leading to a circuit of at least depth 54. Thus, a direct use of the Barrington-based approach would require using $\kappa > 2^{100}$ levels, which is infeasible. We pose the design and analysis of such an "obfuscation-friendly" PRF as a major open question that is motivated by our work.

As another example, consider the task of obfuscating a "fuzzy match" functionality, defined by a Hamming ball of radius $r$ around a secret point $z \in \{0,1\}^n$. That is, the obfuscated function $f_z(x)$ evaluates to 1 if the Hamming distance between $x$ and $z$ is at most $r$, and evaluates to 0 otherwise. Functions from this class can be implemented by (input-oblivious, special) layered branching programs of width $r+1$ and length $n$, leading to an obfuscation that contains roughly $4r^2n$ encoded elements with multilinearity $\kappa \approx n$. For the case $n = 100$ and $r = 20$, we get an obfuscation that consists of roughly 160,000 encoded elements, with multilinearity $\kappa \approx 100$. In contrast, representing such functions by formulas or low-depth circuits, which is essentially equivalent to computing the $(n, r)$-threshold function, leads to a best known formula size $s > n^{4.4}$ and circuit depth $d > 4.9 \log_2 n$ [PZ93, Ser14], which in turn require $\kappa > 10^{19}$ levels of multilinearity using previous obfuscation methods. Thus in this concrete example, our improvement just to the level of multilinearity $\kappa$ is over $10^{17}$; the improvement in the overall running time and size would be even greater.

Table 1.1: Comparing the efficiency of obfuscation schemes for keyed formulas over different bases. We use $\tilde{O}$ to suppress the multiplicative polynomial dependence on the security parameter and other poly-logarithmic terms and $O_\epsilon$ to suppress multiplicative constants which depend on $\epsilon$. Here $s$ is the formula size, $\epsilon > 0$ is an arbitrarily small constant, and $\phi$ is a constant such that for $\kappa$-level multilinear encodings, the size of each encoding is $\tilde{O}(\kappa^\phi)$. The current best known constructions have $\phi = 2$. Evaluation time is given in the form $a \cdot b$, where $a$ denotes the number of multilinear operations (up to lower order additive terms) and $b$ denotes the time for carrying out one multilinear operation.

| Work | Levels of Multilinearity | Size of Obfuscation/ Evaluation Time |
|---|---|---|
| [BGK14] + [PM76] (previous work) $\{\text{AND}, \text{OR}, \text{NOT}\}$-basis | $O(s^{3.64})$ | $O(s^{3.64}) \cdot \tilde{O}((s^{3.64})^\phi)$ |
| This work (direct) $\{\text{AND}, \text{OR}, \text{NOT}\}$-basis | $s$ | $4s^3 \cdot \tilde{O}(s^\phi)$ |
| This work + [Gie01] any complete basis | $O(s^{1+\epsilon})$ | $O_\epsilon(s^{(1+\epsilon)}) \cdot \tilde{O}((s^{(1+\epsilon)})^\phi)$ |

**Security.** While improving security of obfuscation is not the focus of this work, our work on improving efficiency of obfuscation would be meaningless if it sacrificed security. We give evidence for the security of our constructions in the same way that the work of [BGK14] does: by showing that our constructions unconditionally achieve a strong virtual black-box notion of security [BGI01], against adversaries that are limited to algebraic attacks allowed in a generic multilinear model. In fact, our obfuscators are information-theoretically secure against query-bounded adversaries in this generic model. We note that our work actually provides a new *feasibility* result in the generic multilinear model, namely an information-theoretic (and unconditional) obfuscation for non-deterministic branching programs which capture the complexity class NL. This should be compared to previous results in the same model, which only efficiently apply to formulas (or the complexity class $\text{NC}^1$).

Table 1.2: Comparing the efficiency of obfuscation schemes for keyed non-deterministic branching programs and special layered branching programs, as defined in Section 3.1.3. For a general branching program, $s$ denotes the size of the branching program. For a special layered branching program, $n$ is the length and $w$ is the width. Other notation is as in Table 1.1.

| Work | Levels of Multilinearity | Size of Obfuscation/ Evaluation Time |
|---|---|---|
| Previous work (general) | $s^{O(\log s)}$ | $s^{O(\log s)} \cdot \tilde{O}(s^{O(\log s)})$ |
| This work (general) | $s$ | $4s^3 \cdot \tilde{O}(s^{\phi})$ |
| Previous work (special layered) | $n^{O(\log w)}$ | $n^{O(\log w)} \cdot \tilde{O}(n^{O(\log w)})$ |
| This work (special layered) | $n$ | $4nw^2 \cdot \tilde{O}(n^{\phi})$ |

As in the case of [BGK14], our security proof in the generic model can be interpreted in two natural ways: (1) Our proof can be viewed as evidence of virtual black-box security for practical applications, in a similar spirit to proofs of security in the random oracle model [BR93]. It is important to note that analogous to known attacks on contrived schemes in the random oracle model (e.g. [CGH04]), there are known attacks to virtual black-box security for obfuscating quite contrived functionalities [BGI01]. However, no attacks are known for virtual black-box obfuscation for obfuscating practical functionalities. (2) Our proof can also be viewed as evidence that our obfuscator achieves the notion of indistinguishability obfuscation [BGI01], which is a definition of security of obfuscation that does not suffer from any known attacks even for contrived functionalities, but which nevertheless has proven to be quite useful.

# CHAPTER 2

# Hosting Services on an Untrusted Cloud

## 2.1 Prelims

Let $\lambda$ be the security parameter. Below, we describe the primitives used in our scheme.

### 2.1.1 Public Key Encryption Scheme

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

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

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

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

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

### 2.1.2 Indistinguishability Obfuscation

The definition below is from [GGH13b]; there it is called a "family-indistinguishable obfuscator", however they show that this notion follows immediately from their standard definition of indistin-

guishability 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 $\alpha$ 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 chapter, 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 $\lambda$. 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 [GGH13b].

### 2.1.3 Puncturable Pseudorandom Functions

Puncturable Pseudorandom Functions (PRFs) are a simple types of constrained PRFs [BW13, KPT13, BGI14]. 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 [SW14].

**Definition 3.** *A puncturable family of PRFs* $\mathsf{F}$ *is given by a triple of turing machines* $\mathsf{PRFKey}_\mathsf{F}$, $\mathsf{Puncture}_\mathsf{F}, \mathsf{Eval}_\mathsf{F}$*, 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:*

$$\Pr\left[\mathsf{Eval}_\mathsf{F}(K,x) = \mathsf{Eval}_\mathsf{F}(K_S,x) : K \leftarrow \mathsf{PRFKey}_\mathsf{F}(1^\lambda), K_S = \mathsf{Puncture}_\mathsf{F}(K,S)\right] = 1$$

- **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 $\mathsf{st}$, 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 $U_\ell$ denotes the uniform distribution over $\ell$ 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 [GGM84] from one-way functions are easily seen to yield puncturable PRFs, as recently observed by [BW13, KPT13, BGI14]. Thus we have:

**Theorem 3.** *[GGM84, BW13, KPT13, BGI14] 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.*

### 2.1.4 Statistical Simulation-Sound Non-Interactive Zero-Knowledge

This primitive was introduced in [GGH13b] 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 $\mathcal{A}$, following holds.

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

**Computational zero-knowledge [FLS99].** 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 $\mathsf{S}_1$ outputs a simulated CRS and a trapdoor $\tau$ for proving $x$ and $\mathsf{S}_2$ 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.

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

**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.

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

### 2.1.5 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 [EFF85, KRS99]. Our definitions and constructions are inspired by those in [KRS99].

**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: $T \setminus \cup_{i\in[k]} T_i \neq \emptyset$.*

[KRS99] 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 $\lambda$ 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 1.** *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 $\mathsf{ask}$ and announces the public verification key $\mathsf{avk}$. 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 $t_{\mathsf{id}}$ 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 $\mathsf{id}$. Interpret $\mathsf{id}$ as a polynomial in $F_{q,d}$ for $d = \frac{q-1}{k}$, i.e., $\mathsf{id} \in \mathbb{F}_q^{d+1}$. Let $T_{\mathsf{id}}$ 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 $\mathsf{id}$ as a polynomial in $F_{q,d}$ for $d = \frac{q-1}{k}$, i.e., $\mathsf{id} \in \mathbb{F}_q^{d+1}$. Let $T_{\mathsf{id}}$ 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 $t_{\mathsf{id}}$ 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 $\mathsf{id}$, 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 $t_{\mathsf{id}^*}$ 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.

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

Our scheme satisfies unforgeability as follows: Since $\mathcal{T}$ is a $k$-cover-free set system, there exists an element in $T_{\mathsf{id}^*}$ 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 $\mathsf{PRG}$.

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 $\mathsf{avk}$ 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.

**Fake Setup:** The algorithm $\mathsf{FakeAuthGen}(1^\lambda, \mathsf{id}_1, \ldots, \mathsf{id}_k)$ works follows: For each $i \in [k]$, interpret $\mathsf{id}_i$ 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 $T_{\mathsf{id}_i}$ 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 $\mathsf{ask}$. 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 $\mathsf{FakeAuthGen}$ are as follows:

**Correct authentication for all** $\mathsf{id} \in \mathcal{I}$**.** It is easy to see that for any corrupt user $\mathsf{id} \in \mathcal{I}$, $\mathsf{AuthProve}$ will produce a $t_{\mathsf{id}}$ which will verify under $\mathsf{avk}$.

**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 $t_{\mathsf{id}}$ can verify under avk using algorithm Authverify.

**Indistinguishability.** This implies that any PPT adversary given avk and $t_{\mathsf{id}}$ 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 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.

## 2.2 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* $\lambda$*, program* $P$ *and a bound* $k$ *on the number of corrupt users and returns encoded program* $\tilde{P}$ *and a secret* $\sigma$ *to be useful in authentication.*
- $\mathsf{auth}_{\mathsf{id}} \leftarrow \mathsf{SCSS.auth}(\mathsf{id}, \sigma)$*: Takes the identity of a client and the secret* $\sigma$ *and produces an authentication* $\mathsf{auth}_{\mathsf{id}}$ *for the client.*
- $(\tilde{x}, \alpha) \leftarrow \mathsf{SCSS.inp}(1^\lambda, \mathsf{auth}_{\mathsf{id}}, x)$*: Takes as input the security parameter, authentication for the identity and the input* $x$ *to produce encoded input* $\tilde{x}$*. It also outputs* $\alpha$ *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* $\alpha$ *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 $\sigma$. It will then send $\tilde{P}$ to the cloud. Later, it will authenticate users using $\sigma$. 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 $\alpha$. 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 $\alpha$.

**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* $\mathsf{SCSS}$ *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 $\mathsf{Real}'$ 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* 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* $\mathsf{id}$.
- *If* $\mathsf{id} \notin \mathcal{I}'$ *set* $\tilde{y} = \perp$. *Otherwise, run* $\mathsf{decode}(\sigma, \tilde{x}_i)$ *which either outputs* $(x_i, \tau_i)$ *or* $\perp$. *If it outputs* $\perp$, *set* $\tilde{y} = \perp$. *Else, the simulator sends* $(\mathsf{id}, x_i)$ *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 2.3 and prove its security in Section 2.3.2.

### 2.2.1 Additional Properties.

Our scheme can also be modified to achieve some additional properties. As providing these properties is not the focus of this work, however, we omit the details of these extensions in this submission. But 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 $\tilde{y}$ are indeed the correct output of the computation. We stress that verifiability is not the focus of this work. The scheme we present in Section 2.3 can be augmented with known techniques to get verifiability in a straightforward manner. One such method is to use one-time MACs as suggested in [GHR14]. To do so, we can encode an augmented program $\mathsf{P}_{\mathsf{Auth}}$ which gets as input $(k, x)$, evaluates $y = P(\mathsf{id}, x)$ and outputs $(y, \phi)$ where $\phi$ is an authentication-tag $\phi = MAC_k(y)$ for some message-authentication code MAC. Later, the client verifies the MAC. For details, see [GHR14].

**Persistent Memory.** Using techniques similar to those in the recent work of [GHR14], we can also extend our scheme for the setting where the cloud also holds a user-specific persistent memory that maintains state across different invocations of the service by the user. Then we can 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. This result would require the assumptions present in Theorem 1, and would not require any complexity leveraging.

### 2.2.2 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 $(\mathsf{id}, x_i)$, 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 2.4 and prove its security in Section 2.4.1.

## 2.3 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 $\lambda$ be the security parameter. Note that the number of users can be any (unbounded) polynomial in $\lambda$. 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 2.1.5. 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 $2^\lambda$. 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 $\ell_{\mathsf{auth}}$. 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 $\ell_c$. Here $\ell_{\mathsf{in}}$ is the length of the input of the user and $\ell_{\mathsf{kout}}$ is the length of the key for $\mathsf{PRF}_2$ 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 [NY90, Sah99, GGH13b].

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 $\ell_r$. b) $\mathsf{PRF}_2(K_{\mathsf{id}}, \cdot)$ that accepts inputs of length $\ell_r$ and returns strings of length $\ell_{\mathsf{out}}$, where $\ell_{\mathsf{out}}$ 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 $\mathsf{F}$ 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 $\mathsf{PRF}_1$.
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 $\mathsf{PRG} : \{0,1\}^\lambda \rightarrow \{0,1\}^{2\lambda}$.
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 2.1.

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.

**Compute**

**Constants**: Secret key $\mathsf{sk}_1$, puncturable PRF key $K$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ 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.

Figure 2.1: Encoded program Compute given to the cloud

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 $\mathsf{id}$ 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 2.2.

Finally, we describe the procedures $\mathsf{SCSS.inp}$ and $\mathsf{SCSS.eval}$. This interaction between the user and the cloud is also described in Figure 2.3.

**Provider and User**

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

1. The user sends his identity $\mathsf{id}$ 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.

Figure 2.2: Authentication phase between the provider and the user.

**User and Cloud**

**Inputs:** Let the user's identity be $\mathsf{id}$. 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 $K_{\mathsf{id},\mathsf{out}}$ for $\mathsf{PRF}_2$. Set $\mathsf{flag} = 0$.
2. Let $m = (\mathsf{id}||x||t_{\mathsf{id}}||K_{\mathsf{id},\mathsf{out}}||\mathsf{flag})$. 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 $\mathsf{stmt}$ is the following statement

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

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

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 $y = \mathsf{PRF}_2(\alpha, \tilde{y}_1) \oplus \tilde{y}_2$.

Figure 2.3: Encoding of input by an authenticated user and evaluation by the cloud

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 $K_{\mathsf{id},\mathsf{out}}$ for $\mathsf{PRF}_2$. Let $m = (\mathsf{id}||x||t_{\mathsf{id}}||K_{\mathsf{id},\mathsf{out}}||0)$. 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 $\pi$ 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 $\mathsf{SCSS.eval}(\tilde{F} = P_{comp}, \tilde{x})$: Run $\tilde{F}$ on $\tilde{x}$ to obtain $\tilde{y}$. The user parses $\tilde{y}$ as $\tilde{y}_1, \tilde{y}_2$ and computes $y = \mathsf{PRF}_2(\alpha, \tilde{y}_1) \oplus \tilde{y}_2$.

### 2.3.1 Security Proof Overview

In this section, we give a proof overview for Theorem 1 for the scheme described above. Among all the security properties, only the proof of untrusted cloud security is most challenging. Hence, in this section, we will prove untrusted cloud security (Definition 7) of our scheme via a sequence of hybrids. Before that, since it is easy to see why our scheme satisfies untrusted client security (Definition 8), we only give a proof overview below.

**Proof Overview for untrusted client security.** We first prove the untrusted client security. In our scheme, the secret information $\sigma$ 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 $\mathsf{decode}$ procedure can work identically to the program Compute to extract an input $x$, authentication $t_{\mathsf{id}}$, a key $K_{\mathsf{id},\mathsf{out}}$, and $\mathsf{flag}$ from $\tilde{x}$. If $\mathsf{flag} = 1$, it gives $y = 0$ to $\mathsf{response}$ procedure. Else, if authentication $t_{\mathsf{id}}$ verifies using $\mathsf{avk}$, it sends the $(\mathsf{id}, x)$ 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 $t_{\mathsf{id}}$ 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 $\mathsf{id}$ who has not done the set up phase to obtain $\mathsf{auth}_{\mathsf{id}}$ should not be able to learn the output of $\mathsf{F}$ on any input using interaction with the honest cloud. Note that in our scheme $\mathsf{F}$ is invoked only if the authentication extracted by the program verifies under $\mathsf{avk}$. Hence, the security will follow from the $k$-unforgeability property of our scheme (see Section 2.1.5).

**Proof Overview of Untrusted Cloud Security** Consider a PPT adversary $\mathcal{A}$ who controls the cloud and a collection of at most $k$ users. Let $\mathsf{F}$ and $\mathsf{G}$ be two functions such that $\mathsf{F}$ and $\mathsf{G}$ 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 $\mathsf{F}$ or $\mathsf{G}$. 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 $\mathsf{F}$ 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 $\mathsf{F}$ on any of the honest ids. We also know that the functions $\mathsf{F}$ and $\mathsf{G}$ are functionally equivalent for all the corrupt ids. At this point, we can rely on the indistinguishability of obfuscations of program Compute using $\mathsf{F}$ and program Compute using $\mathsf{G}$.

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

Formal description of hybrids is provided in Section 2.3.2.

### 2.3.2 Formal Security Proof

Now, we describe the hybrids formally. We denote changes between subsequent hybrids using underlined font. In this sequence of hybrids, we will be changing the program being obfuscated multiple times and rely on the security of indistinguishability obfuscation. Since these changes are quite subtle, we follow the presentation style of [SW14], where we write one hybrid per page spelling out the whole experiment in each hybrid.

Each hybrid below is an experiment that takes as input $1^\lambda$. The final output of each hybrid experiment is the output produced by the adversary when it terminates. Moreover, in each of these

hybrids, the adversary also receives authentication identities for all the corrupt users. We omit writing this in each hybrid because these are not changed explicitly.

The first hybrid $\mathsf{Hyb}_0$ is the real execution with $\mathsf{F}$.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{PRG}$.
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 $\mathsf{id}$, 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 2.3 for details.
8. Output $P_{comp}$ and queries of all honest users $\mathsf{id}_1, \ldots, \mathsf{id}_m$.

**Compute**

**Constants**: Secret key $\mathsf{sk}_1$, puncturable PRF key $K$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ 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.

Figure 2.4: Program Compute

Next we describe the first sequence of hybrids $\mathsf{Hyb}_{1:1}, \ldots, \mathsf{Hyb}_{1:14}, \ldots, \mathsf{Hyb}_{m:1}, \ldots, \mathsf{Hyb}_{m:14}$. In the sub-sequence of hybrids $\mathsf{Hyb}_{i:1}, \ldots, \mathsf{Hyb}_{i:14}$, we only change the behavior of the honest user $\mathsf{id}_i$. All the other honest users $\mathsf{id}_j$ such that $j \neq i$ behave identically as in $\mathsf{Hyb}_{i-1:14}$. Hence, we omit their behavior from the description of the hybrids for ease of notation.

Also, we denote $\mathsf{id}_i$ by $\mathsf{id}^*$.

Let $\mathsf{Hyb}_{0:14} = \mathsf{Hyb}_0$.

$\mathsf{Hyb}_{i:1}$. This is same as $\mathsf{Hyb}_{i-1:14}$. We use this hybrid as a way to write how the user $\mathsf{id}^*$ 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 $\mathsf{Hyb}_{i-1:14}$ and $\mathsf{Hyb}_{i:1}$ is identical.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{PRF}_2$.
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 $\pi^* = \mathsf{NIZKProve}(\mathsf{crs}, \mathsf{stmt}, (m_1^*, r_1, r_2))$, where stmt is defined in Figure 2.3.
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 $\mathsf{sk}_1$, puncturable PRF key $K$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ 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.

Figure 2.5: Program Compute

$\mathsf{Hyb}_{i:2}$**.** In this hybrid, we modify the Compute program as follows. First, we add the constants $\mathsf{id}^*, c^*, y^*$ to the program. Then, we add an if statement at the start that outputs $y^*$ if the input is $(\mathsf{id}^*, c^*)$, as this is exactly what the original Compute program would do. Now, because the "if" statement is in place, we know that $\mathsf{PRF}_1(K, \cdot)$ cannot be evaluated at $(\mathsf{id}^*, c^*)$ 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 security of $i\mathcal{O}$.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{id}$, 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 $\mathsf{PRF}_2$.

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 $\pi^* = \mathsf{NIZKProve}(\mathsf{crs}, \mathsf{stmt}, (m_1^*, r_1, r_2))$, where stmt is defined in Figure 2.3.
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**: $(\mathsf{id}^*, c^*, y^*)$, Secret key $\mathsf{sk}_1$, 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.

Figure 2.6: Program Compute

$\mathsf{Hyb}_{i:3}$. In this hybrid, the value $r^*$ is chosen randomly, instead of as the output of $\mathsf{PRF}_1(K, (\mathsf{id}^*, c^*))$. Moreover, since $r^*$ is a uniform random string, it can be picked anytime. So for convenience in later hybrids, we move it up before picking $K_{\mathsf{id}^*,\mathsf{out}}$.

The indistinguishability of two hybrids follows by pseudorandomness property of the punctured PRF $\mathsf{PRF}_1$.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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. 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.
6. Set $t_{\mathsf{id}^*} = \mathsf{AuthGen}(\mathsf{ask}, \mathsf{id}^*)$.
7. Set $\mathsf{flag}^* = 0$.
8. Let $r^*$ be chosen randomly.
9. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for $\mathsf{PRF}_2$.
10. Let $x^*$ be the input. Let $m_1^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}||\mathsf{flag}^*)$ and $m_2^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}||\mathsf{flag}^*)$.
11. Compute $c_1^*, c_2^*$ as follows: $c_1^* = \mathsf{PKEnc}(\mathsf{pk}_1, m_1^*; r_1)$, $c_2^* = \mathsf{PKEnc}(\mathsf{pk}_2, m_2^*; r_2)$.
12. Compute $\pi^* = \mathsf{NIZKProve}(\mathsf{crs}, \mathsf{stmt}, (m_1^*, r_1, r_2))$, where stmt is defined in Figure 2.3.
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.

$\mathsf{Hyb}_{i:4}$**.** In this hybrid, instead of generating crs honestly, we generate it using the simulator $\mathsf{S}_1$ and also simulate the proof $\pi^*$ using $\mathsf{S}_2$.

The two hybrids are indistinguishable by computational zero-knowledge property of NIZK used.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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.

**Compute**

**Constants**: $(\mathsf{id}^*, c^*, y^*)$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K(\{(\mathsf{id}^*, c^*)\})$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ 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.

Figure 2.7: Program Compute

4. On receiving a corrupt user's identity $\mathsf{id}$, 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. Let $r^*$ be chosen randomly.
8. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for $\mathsf{PRF}_2$.
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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.3.
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 $\mathsf{sk}_1$, puncturable PRF key $K(\{(\mathsf{id}^*, c^*)\})$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ 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.

Figure 2.8: Program Compute

$\mathsf{Hyb}_{i,5}$. In this hybrid, we change $m_2^*$ to include a punctured key $K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})$ instead of original key $K_{\mathsf{id}^*,\mathsf{out}}$.

The two hybrids are indistinguishable by $\mathsf{IND} - \mathsf{CPA}$ security of public key encryption scheme $\mathsf{pke}$. This is because the hybrids do not use the secret $\mathsf{sk}_2$.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{id}$, 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. Let $r^*$ be chosen randomly.
8. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for $\mathsf{PRF}_2$.
9. Let $x^*$ be the input. Let $m_1^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}||\mathsf{flag}^*)$ and $m_2^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})||\mathsf{flag}^*)$.
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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.3.
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 $\mathsf{sk}_1$, 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.

Figure 2.9: Program Compute

$\mathsf{Hyb}_{i:6}$. In this hybrid, we start using $\mathsf{sk}_2$ in the program instead of $\mathsf{sk}_1$.

We claim that the programs Compute in the two hybrids are functionally equivalent in the two hybrids and hence the indistinguishability follows by the security of $i\mathcal{O}$. This is because for all $c' = (c'_1, c'_2, \pi')$ such that $c' \neq c^*$, if $\pi'$ is accepted then $\mathsf{PKDec}(\mathsf{sk}_1, c'_1) = \mathsf{PKDec}(\mathsf{sk}_2, c'_2)$. Also, on input $(\mathsf{id}^*, c^*)$, the output of the program is identical in the two hybrids. Finally, on input $(\mathsf{id}, c^*)$ for some $\mathsf{id} \neq \mathsf{id}^*$, both programs output $\perp$ due to condition in Step 3. This proves that the two programs are functionally equivalent.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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. Let $r^*$ be chosen randomly.
8. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for $\mathsf{PRF}_2$.
9. Let $x^*$ be the input. Let $m_1^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}||\mathsf{flag}^*)$ and $m_2^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})||\mathsf{flag}^*)$.
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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.3.
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.

$\mathsf{Hyb}_{i:7}$. In this hybrid, we change $m_1^*$ to include a punctured key $K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})$ instead of original key $K_{\mathsf{id}^*,\mathsf{out}}$.

**Compute**

**Constants**: $(\mathsf{id}^*, c^*, y^*)$, Secret key $\mathsf{sk}_2$, puncturable PRF key $K(\{(\mathsf{id}^*, c^*)\})$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ 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 $\underline{(\mathsf{id}'||x||t_{\mathsf{id}}||K_{\mathsf{id},\mathsf{out}}||\mathsf{flag}) = \mathsf{PKDec}(\mathsf{sk}_2, c_2)}$ 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.

Figure 2.10: Program Compute

The two hybrids are indistinguishable by $\mathsf{IND} - \mathsf{CPA}$ security of public key encryption scheme $\mathsf{pke}$. This is because the hybrids do not use the secret $\mathsf{sk}_1$.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{PRG}$.
4. On receiving a corrupt user's identity $\mathsf{id}$, 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. Let $r^*$ be chosen randomly.
8. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for $\mathsf{PRF}_2$.
9. Let $x^*$ be the input. Let $\underline{m_1^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})||\mathsf{flag}^*)}$ 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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.3.
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 $\mathsf{sk}_2$, 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 $(\mathsf{id}'||x||t_{\mathsf{id}}||K_{\mathsf{id},\mathsf{out}}||\mathsf{flag}) = \mathsf{PKDec}(\mathsf{sk}_2, c_2)$ 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.

Figure 2.11: Program Compute

$\mathsf{Hyb}_{i:8}$. In this hybrid, we switch back to using $\mathsf{sk}_1$ in the program instead of $\mathsf{sk}_2$.

We claim that the programs compute in the two hybrids are functionally equivalent in the two hybrids and hence the indistinguishability follows by the security of $i\mathcal{O}$. this is because for all $c' = (c_1', c_2', \pi')$ such that $c' \neq c^*$, if $\pi'$ is accepted then $\mathsf{PKDec}(\mathsf{sk}_1, c_1') = \mathsf{PKDec}(\mathsf{sk}_2, c_2')$. Also, on input $(\mathsf{id}^*, c^*)$, the output of the program is identical in the two hybrids. Finally, on input $(\mathsf{id}, c^*)$ for some $\mathsf{id} \neq \mathsf{id}^*$, both programs output $\perp$ due to condition in Step 3. This proves that the two programs are functionally equivalent.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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. Let $r^*$ be chosen randomly.
8. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for $\mathsf{PRF}_2$.
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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.3.
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.

$\mathsf{Hyb}_{i:9}$**.** In this hybrid, we change the value of $y^* = (r^*, u^*)$ where $u^*$ is a uniformly random string of appropriate length.

The indistinguishability of the two hybrids follows by pseudorandomness property of punctured key $K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})$. Because of this $\mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*)$ is indistinguishable from random string.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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.

**Compute**

**Constants**: $(\mathsf{id}^*, c^*, y^*)$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K(\{(\mathsf{id}^*, c^*)\})$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ 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.

Figure 2.12: Program Compute

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 $\mathsf{PRF}_2$.
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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where $\mathsf{stmt}$ is defined in Figure 2.3.
12. Compute $\pi^* = \mathsf{S}_2(\mathsf{crs}, \tau, \mathsf{stmt})$.
13. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
14. Let $y^* = (r^*, u^*)$, where $u^*$ is a uniformly random string of appropriate length.
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 $\mathsf{sk}_1$, puncturable PRF key $K(\{(\mathsf{id}^*, c^*)\})$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ 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.

Figure 2.13: Program Compute

$\mathsf{Hyb}_{i:10}$. In this hybrid, we change the value of $y^*$ to $(r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*))$.

Similar to above, the indistinguishability of the two hybrids follows by pseudorandomness property of punctured key $K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})$. In particular, $\mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*)$ is indistinguishable from random string.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{id}$, 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. Let $r^*$ be chosen randomly.
8. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for $\mathsf{PRF}_2$.
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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.3.
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 $\mathsf{sk}_1$, 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.

Figure 2.14: Program Compute

$\mathsf{Hyb}_{i:11}$**.** In this hybrid, we change the value of $\mathsf{flag}^*$ to 1 instead of 0 and $t_{\mathsf{id}^*}$ 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 similar to hybrids $\mathsf{Hyb}_{i:5}$ to $\mathsf{Hyb}_{i:8}$ 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 when the input is $c^*$. Moreover, because of the initial "if" condition, there is no check on $\mathsf{flag}^*$ or $t_{\mathsf{id}^*}$. Hence, while switching keys for decryption, functional equivalence follows in a straight-forward manner.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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. Pick $t_{\mathsf{id}^*}$ 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 $\mathsf{PRF}_2$.
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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.3.
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.

$\mathsf{Hyb}_{i:12}$**.** In this hybrid, we again start generating the crs and the proof $\pi^*$ honestly.

The indistinguishability follows from the computational zero-knowledge property of the NIZK used.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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.

**Compute**

**Constants**: $(\mathsf{id}^*, c^*, y^*)$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K(\{(\mathsf{id}^*, c^*)\})$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ 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.

Figure 2.15: Program Compute

4. On receiving a corrupt user's identity $\mathsf{id}$, 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. Pick $t_{\mathsf{id}^*}$ 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 $\mathsf{PRF}_2$.
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 $\mathsf{crs} \leftarrow \mathsf{NIZKSetup}(1^\lambda)$.
12. Compute $\pi^* = \mathsf{NIZKProve}(\mathsf{crs}, \mathsf{stmt}, (m_1^*, r_1, r_2))$, where $\mathsf{stmt}$ is defined in Figure 2.3.
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 $\mathsf{sk}_1$, puncturable PRF key $K(\{(\mathsf{id}^*, c^*)\})$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ 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.

Figure 2.16: Program Compute

$\mathsf{Hyb}_{i:13}$. In this hybrid, we set $r^* = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*))$ instead of random.

The indistinguishability follows from the pseudorandomness property of the punctured PRF $\mathsf{PRF}_1$.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{id}$, 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. Pick $t_{\mathsf{id}^*}$ 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 $\mathsf{PRF}_2$.
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 $\pi^* = \mathsf{NIZKProve}(\mathsf{crs}, \mathsf{stmt}, (m_1^*, r_1, r_2))$, where stmt is defined in Figure 2.3.
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**: $(\mathsf{id}^*, c^*, y^*)$, Secret key $\mathsf{sk}_1$, 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.

Figure 2.17: Program Compute

$\mathsf{Hyb}_{i:14}$**.** In this hybrid, we remove the initial "if" condition and the constants $(\mathsf{id}^*, c^*, y^*)$, and un-puncture the key $K$.

The indistinguishability follows from the security of $i\mathcal{O}$.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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. Pick $t_{\mathsf{id}^*}$ 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 $\mathsf{PRF}_2$.
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 $\pi^* = \mathsf{NIZKProve}(\mathsf{crs}, \mathsf{stmt}, (m_1^*, r_1, r_2))$, where stmt is defined in Figure 2.3.
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 $\mathsf{sk}_1$, 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.

Figure 2.18: 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 encodes his actual input $x$ with $\mathsf{flag} = 0$ and a valid authentication $t_{\mathsf{id}}$, and obtains output according to the function $\mathsf{F}$ on $(\mathsf{id}, x)$. 2) The honest user encodes $0^{\ell_{\mathsf{in}}}$ with $\mathsf{flag} = 1$ and uniformly random $t_{\mathsf{id}}$, and receives encoding of $0$ as output (without invoking the function $\mathsf{F}$.)

Below we write the final hybrid obtained above as $\mathsf{Hyb}_1$ as follows:

$\mathsf{Hyb}_1$**:** This hybrid is same as $\mathsf{Hyb}_{m:14}$. In the hybrid $\mathsf{Hyb}_{m:14}$, all the user queries will have $\mathsf{flag} = 1$, $t_{\mathsf{id}}$ will be a random string, and input will be $0^{\ell}_{\mathsf{in}}$. Hence, the program Compute will not invoke the function $\mathsf{F}$ for any of the honest users.

The underlined statement summarizes the main difference between $\mathsf{Hyb}_0$ and $\mathsf{Hyb}_1$.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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, $t_{\mathsf{id}}$ 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 $\pi$ 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 $\mathsf{sk}_1$, puncturable PRF key $K$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ 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.

Figure 2.19: Program Compute

$\mathsf{Hyb}_2$**:** 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 2.1.5). Note that both hybrids do not depend on $\mathsf{ask}$ and need only the valid authentications for corrupt users.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\underline{(\mathsf{avk}, \mathsf{ask}) \leftarrow \mathsf{FakeAuthGen}(1^\lambda, \mathcal{I})}$ 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 $\mathsf{id}$, 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. As noted before, $\mathsf{AuthProve}$ still returns valid authentication for all users in $\mathcal{I}$.
7. For each of the honest users, $t_{\mathsf{id}}$ is set to a random string of appropriate length and $\mathsf{flag}$ is set

to 1. Ciphertexts $(c_1, c_2)$ and proof $\pi$ 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 $\mathsf{sk}_1$, puncturable PRF key $K$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ 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.

Figure 2.20: Program Compute

$\mathsf{Hyb}_3$**:** This is the most important hybrid, where we change the program to use $\mathsf{G}$ instead of $\mathsf{F}$.

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 $\mathsf{avk}$. 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 $\mathsf{F}$ and $\mathsf{G}$ are equivalent for corrupt users.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $(\mathsf{avk}, \mathsf{ask}) \leftarrow \mathsf{FakeAuthGen}(1^\lambda, \mathcal{I})$ with respect to cover-free set system $\mathcal{T}$ and $\mathsf{PRG}$.
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 $\mathsf{id}$, 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, $t_{\mathsf{id}}$ is set to a random string of appropriate length and flag is set to 1. Ciphertexts $(c_1, c_2)$ and proof $\pi$ 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 $\mathsf{sk}_1$, puncturable PRF key $K$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ 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.

Figure 2.21: Program Compute

Finally, using a similar sequence of hybrids we can move from $\mathsf{Hyb}_3$ to a hybrid which corresponds to real execution using $\mathsf{G}$ and honest party inputs $x'_1, \ldots, x'_m$.

## 2.4 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 $\lambda$ be the security parameter. Note that the number of users can be any (unbounded) polynomial in $\lambda$. 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 $k$.

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 2.1.5. 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 $2^\lambda$. We will interpret the user's identity $\mathsf{id}$ 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 $\ell_{\mathsf{auth}}$. 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 $\ell_c$. Here $\ell_{\mathsf{in}}$ is the length of the input of the user and $\ell_{\mathsf{kout}}$ is the length of the key for $\mathsf{PRF}_2$ 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 [NY90, Sah99, GGH13b].

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 $\ell_z$ is the length of the cloud's input $z$. b) $\mathsf{PRF}_2(K_{\mathsf{id}}, \cdot)$ that accepts inputs of length $\ell_r$ and returns strings of length $\ell_{\mathsf{out}}$, where $\ell_{\mathsf{out}}$ 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 $\mathsf{PRF}_1$ because in the proof we would require that a random string of length $\ell_r$ does not lie in the image of $\mathsf{PRF}_1(K, \cdot)$.

Now we describe our scheme.

Consider an honest provider $\mathcal{H}$ who holds a function $\mathsf{F}$ 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 $\mathsf{PRF}_1$.
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 $\mathsf{PRG}$.
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 2.22.

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

**Compute**

**Constants**: Secret key $\mathsf{sk}_1$, puncturable PRF key $K$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id,out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
6. Output $y = \perp$ and end.

Figure 2.22: Encoded program Compute given to the cloud (scheme with cloud input)

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 2.23.

**Provider and User**

**Inputs:** Let the user's identity be id. The provider has input two public keys $\mathsf{pk}_1, \mathsf{pk}_2$, common reference string crs and the 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 $(t_{\mathsf{id}}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs})$ to the user.

Figure 2.23: Authentication phase between the provider and the user (scheme with cloud input)

Finally, we describe the procedures SCSS.inp and SCSS.eval. This interaction between the user and the cloud is also described in Figure 2.24.

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 $\mathsf{PRF}_2$. 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 $\pi$ 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 $\mathsf{SCSS.eval}(\tilde{F} = P_{comp}, \tilde{x}, z)$: 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.** We prove that our scheme satisfies Theorem 2 i.e. it is a secure cloud service scheme with cloud inputs in Section 2.4.1.

### 2.4.1 Security Proofs for Secure Cloud Service Scheme with Cloud Inputs

In this section, we prove Theorem 2 for the scheme described in Section 2.4.

It is easy to see that our scheme satisfies the untrusted client security defined in Definition 8 and security against unauthenticated clients in the same way as the previous scheme. In this section, we will prove untrusted cloud security (Definition 7) of our scheme described in Section 2.4.

**User and Cloud**

**Inputs:** Let the user's identity be $\mathsf{id}$. Let the user's input to the function be $x$. In addition, An authenticated user also has the authentication $(t_{\mathsf{id}}, \mathsf{pk}_1, \mathsf{pk}_2, \mathsf{crs})$ obtained from the provider.
The cloud has obfuscated program $P_{comp}$ and input $z$. 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 $K_{\mathsf{id},\mathsf{out}}$ for $\mathsf{PRF}_2$. Set $\mathsf{flag} = 0$.
2. Let $m = (\mathsf{id}||x||t_{\mathsf{id}}||K_{\mathsf{id},\mathsf{out}}||\mathsf{flag})$. 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 $\mathsf{stmt}$ is the following statement

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

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

The cloud runs the program $P_{comp}$ on the input $(\tilde{x}, z)$ 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 $y = \mathsf{PRF}_2(\alpha, \tilde{y}_1) \oplus \tilde{y}_2$.

Figure 2.24: Encoding of input by an authenticated user and evaluation by the cloud (scheme with cloud input)

Consider a PPT adversary $\mathcal{A}$ who controls the cloud and a collection of at most $k$ users. Let $\mathsf{F}$ and $\mathsf{G}$ be two functions such that $\mathsf{F}$ and $\mathsf{G}$ are functionally equivalent for corrupt users. That is, for all the inputs of the corrupt users and all the inputs of the cloud, the functions $\mathsf{F}$ and $\mathsf{G}$ produce identical outputs. Then, we will prove that $\mathcal{A}$ can not distinguish between the cases when the provider uses the function $\mathsf{F}$ or $\mathsf{G}$. 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 $\mathsf{F}$ is not being invoked for any of the honest users. In this step, we will have $O(m \cdot 2^{\ell_z})$ hybrids, where $\ell_z$ is the length of

the cloud's input to the program Compute.

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. In this step, we will have at most $q^2$ hybrids. Now, we can be absolutely certain that the program does not invoke the function $\mathsf{F}$ on any of the honest ids. We also know that the functions $\mathsf{F}$ and $\mathsf{G}$ are functionally equivalent for all the corrupt ids. At this point, we can rely on the indistinguishability of obfuscations of program Compute using $\mathsf{F}$ and program Compute using $\mathsf{G}$.

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

In this sequence of hybrids, we will be changing the program being obfuscated multiple times and rely on the security of indistinguishability obfuscation. Since these changes are quite subtle, we follow the presentation style of [SW14], where we write one hybrid per page spelling out the whole experiment in each hybrid.

Each hybrid below is an experiment that takes as input $1^\lambda$. The final output of each hybrid experiment is the output produced by the adversary when it terminates. Moreover, in each of these hybrids, the adversary also receives authentication identities for all the corrupt users. We omit writing this in each hybrid because these are not changed explicitly.

The first hybrid $\widetilde{\mathsf{Hyb}_0}$ is the real execution with $\mathsf{F}$.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{PRG}$.
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 2.24 for details.

8. Output $P_{comp}$ and queries of all honest users $\mathsf{id}_1, \ldots, \mathsf{id}_m$.

**Compute**

**Constants**: Secret key $\mathsf{sk}_1$, puncturable PRF key $K$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
6. Output $y = \perp$ and end.

Figure 2.25: Program Compute

Next we describe the first sequence of hybrids. We have the following sub-sequence of hybrids for each honest user $\mathsf{id}_i$:

$\mathsf{Hyb}_i^{(1)}, \mathsf{Hyb}_i^{(2)}, \mathsf{Hyb}_i^{(3)}, \mathsf{Hyb}_{i:1:1}, \ldots, \mathsf{Hyb}_{i:1:12}, \ldots \mathsf{Hyb}_{i:2^{\ell_z}:1} \ldots \mathsf{Hyb}_{i:2^{\ell_z}:12}, \mathsf{Hyb}_i^{(4)}, \mathsf{Hyb}_i^{(5)}, \mathsf{Hyb}_i^{(6)}, \mathsf{Hyb}_i^{(7)}$

In this sub-sequence of hybrids we only change the behavior of the honest user $\mathsf{id}_i$ and the program for all the inputs of the cloud $\{1, \ldots, 2^{\ell_z}\}$. All the other honest users $\mathsf{id}_j$ such that $j \neq i$ behave identically as in $\mathsf{Hyb}_{i-1}^{(7)}$. Hence, we omit their behavior from the description of the hybrids for ease of notation.

Also, we denote $\mathsf{id}_i$ by $\mathsf{id}^*$.

**$\mathsf{Hyb}_i^{(1)}$.** This is same as $\mathsf{Hyb}_{i-1}^{(7)}$. We use this hybrid as a way to write how the user $\mathsf{id}^*$ 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 $\mathsf{Hyb}_{i-1}^{(7)}$ and $\mathsf{Hyb}_i^{(1)}$ is identical.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.

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 $\mathsf{PRF}_2$.
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 $\pi^* = \mathsf{NIZKProve}(\mathsf{crs}, \mathsf{stmt}, (m_1^*, r_1, r_2))$, where stmt is defined in Figure 2.24.
12. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
13. Let $r^* = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, 1))$.
14. Let $y^* = (r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*) \oplus \mathsf{F}(\mathsf{id}^*, x^*, 1))$.
15. Let $P_{comp} = i\mathcal{O}(\text{Compute})$.
16. Output $(P_{comp}, (\mathsf{id}^*, c^*))$ and the queries of other honest users.

The computations of $r^*, y^*$ are not being utilized yet in this hybrid. But will still write these steps to be consistent with future hybrids. Also, note that $y^*$ is computed using cloud's input $z = 1$.

$\mathsf{Hyb}_i^{(2)}$**.** In this hybrid, we modify the Compute program as follows. First, we add the constants $(\mathsf{id}^*, c^*, x^*, K_{\mathsf{id}^*,\mathsf{out}})$ to the program. Then, we add an "if" statement at the start such that the output in this case is exactly what the original Compute program would do.

The output of the adversary in $\mathsf{Hyb}_i^{(1)}$ and $\mathsf{Hyb}_i^{(2)}$ is indistinguishable by the security of $i\mathcal{O}$. This is because the constants $(x^*, K_{\mathsf{id}^*,\mathsf{out}})$ are the same as what is being encryped in $c^*$. Also, in $c^*$, $t_{\mathsf{id}^*}$ is valid and $\mathsf{flag}^* = 0$. Finally, recall that cloud's inputs are $\{1, \ldots, 2^{\ell_z}\}$. Hence, the "if" condition $(z < 1)$ is never satisfied.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.

**Compute**

**Constants**: Secret key $\mathsf{sk}_1$, puncturable PRF key $K$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
6. Output $y = \perp$ and end.

Figure 2.26: Program Compute

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 $\mathsf{id}$, 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 $\mathsf{PRF}_2$.
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 $\pi^* = \mathsf{NIZKProve}(\mathsf{crs}, \mathsf{stmt}, (m_1^*, r_1, r_2))$, where $\mathsf{stmt}$ is defined in Figure 2.24.
12. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
13. Let $r^* = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, 1))$.
14. Let $y^* = (r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*) \oplus \mathsf{F}(\mathsf{id}^*, x^*, 1))$.

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**: $(\mathsf{id}^*, c^*, x^*, K_{\mathsf{id}^*,\mathsf{out}})$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

1. If $(\mathsf{id}, c) = (\mathsf{id}^*, c^*)$, do the following:
   - Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, z))$.
   - If $z < 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r))$ and end.
   - Else, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x^*, z))$ 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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
7. Output $y = \perp$ and end.

Figure 2.27: Program Compute

$\mathsf{Hyb}_{i,}^{(3)}$**.** In this hybrid, instead of generating $\mathsf{crs}$ honestly, we generate it using the simulator $\mathsf{S}_1$ and also simulate the proof $\pi^*$ using the simulator $\mathsf{S}_2$.

The two hybrids are indistinguishable by computational zero-knowledge property of NIZK used.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{PRF}_2$.
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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.24.
11. Compute $\pi^* = \mathsf{S}_2(\mathsf{crs}, \tau, \mathsf{stmt})$.
12. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
13. Let $r^* = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, 1))$.
14. Let $y^* = (r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*) \oplus \mathsf{F}(\mathsf{id}^*, x^*, 1))$.
15. Let $P_{comp} = i\mathcal{O}(\text{Compute})$.
16. Output $(P_{comp}, (\mathsf{id}^*, c^*))$ and the queries of other honest users.

Below, we describe the sequence of hybrids, $\mathsf{Hyb}_{i:j:1}, \ldots, \mathsf{Hyb}_{i:j:12}$ for any $j \in [2^{\ell_z}]$.

$\mathsf{Hyb}_{i:j:1}$. This hybrid is equivalent to the hybrid $\mathsf{Hyb}_{i:j-1:12}$. We consider this hybrid to explicitly write the changes achieved so far. These have been shown in **bold** font.

The output of the adversary in $\mathsf{Hyb}_{i:j:1}$ and $\mathsf{Hyb}_{i:j-1:12}$ is identical.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{flag}^* = 0$.

**Compute**

**Constants**: $(\mathsf{id}^*, c^*, x^*, K_{\mathsf{id}^*,\mathsf{out}})$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

1. If $(\mathsf{id}, c) = (\mathsf{id}^*, c^*)$, do the following:
   - Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, z))$.
   - If $z < 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r))$ and end.
   - Else, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x^*, z))$ 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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
7. Output $y = \perp$ and end.

Figure 2.28: Program Compute

7. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for $\mathsf{PRF}_2$.
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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.24.
11. Compute $\pi^* = \mathsf{S}_2(\mathsf{crs}, \tau, \mathsf{stmt})$.
12. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
13. Let $r^* = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, j))$.
14. Let $y^* = (r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*) \oplus \mathsf{F}(\mathsf{id}^*, x^*, j))$.
15. Let $P_{comp} = i\mathcal{O}(\text{Compute})$.

16. Output $(P_{comp}, (\mathsf{id}^*, c^*))$ and the queries of other honest users.

Note that $r^*, y^*$ are computed using $z = 1$. These have not been used by the program yet.

**Compute**

**Constants**: $(\mathsf{id}^*, c^*, x^*, K_{\mathsf{id}^*,\mathsf{out}})$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

1. If $(\mathsf{id}, c) = (\mathsf{id}^*, c^*)$, do the following:
   - Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, z))$.
   - **If $z < j$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r))$ and end.**
   - **Else, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x^*, z))$ 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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
7. Output $y = \perp$ and end.

Figure 2.29: Program Compute

$\mathsf{Hyb}_{i:j:2}$. In this hybrid, we modify the Compute program as follows. First, we add the constants $j, y^*$ to the program. Then, we add an if statement at the start that outputs $y^*$ if the user's input is $(\mathsf{id}^*, c^*)$, and cloud's input is $j$ as this is exactly what the original Compute program would do. Now, because the "if" statement is in place, we know that $\mathsf{PRF}_1(K, \cdot)$ cannot be evaluated at $(\mathsf{id}^*, c^*, j)$ 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 security of $i\mathcal{O}$.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{PRF}_2$.
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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.24.
11. Compute $\pi^* = \mathsf{S}_2(\mathsf{crs}, \tau, \mathsf{stmt})$.
12. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
13. Let $r^* = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, j))$.
14. Let $y^* = (r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*) \oplus \mathsf{F}(\mathsf{id}^*, x^*, j))$.
15. Let $P_{comp} = i\mathcal{O}(\text{Compute})$.
16. Output $(P_{comp}, (\mathsf{id}^*, c^*))$ and the queries of other honest users.

$\mathsf{Hyb}_{i:j:3}$. In this hybrid, the value $r^*$ is chosen randomly, instead of as the output of $\mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, j))$. Moreover, since $r^*$ is a uniform random string, it can be picked anytime. So for convenience in later hybrids, we move it up before picking $K_{\mathsf{id}^*,\mathsf{out}}$.

The indistinguishability of two hybrids follows by pseudorandomness property of the punctured PRF $\mathsf{PRF}_1$.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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.

**Compute**

**Constants**: $(\mathsf{id}^*, c^*, x^*, K_{\mathsf{id}^*,\mathsf{out}})$, $(j, y^*)$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K(\{(\mathsf{id}^*, c^*, j)\})$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

1. If $(\mathsf{id}, c) = (\mathsf{id}^*, c^*)$, do the following:
   - If $z = j$, output $y^*$.
   - Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, z))$.
   - If $z < j$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r))$ and end.
   - Else, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x^*, z))$ 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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
7. Output $y = \perp$ and end.

Figure 2.30: Program Compute

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 $\mathsf{PRF}_2$.
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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.24.
12. Compute $\pi^* = \mathsf{S}_2(\mathsf{crs}, \tau, \mathsf{stmt})$.

13. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
14. Let $y^* = (r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*) \oplus \mathsf{F}(\mathsf{id}^*, x^*, j))$.
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**: $(\mathsf{id}^*, c^*, x^*, K_{\mathsf{id}^*,\mathsf{out}})$, $(j, y^*)$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K(\{(\mathsf{id}^*, c^*, j)\})$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

1. If $(\mathsf{id}, c) = (\mathsf{id}^*, c^*)$, do the following:
   - If $z = j$, output $y^*$.
   - Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, z))$.
   - If $z < j$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r))$ and end.
   - Else, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x^*, z))$ 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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
7. Output $y = \perp$ and end.

Figure 2.31: Program Compute

$\mathsf{Hyb}_{i:j:4}$. In this hybrid, we change $m_2^*$ to include a punctured key $K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})$ instead of original key $K_{\mathsf{id}^*,\mathsf{out}}$.

The two hybrids are indistinguishable by $\mathsf{IND} - \mathsf{CPA}$ security of public key encryption scheme $\mathsf{pke}$. This is because the hybrids do not use the secret $\mathsf{sk}_2$.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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. Let $r^*$ be chosen randomly.
8. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for $\mathsf{PRF}_2$.
9. Let $x^*$ be the input. Let $m_1^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}||\mathsf{flag}^*)$ and $\underline{m_2^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})||\mathsf{flag}^*).}$
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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.24.
12. Compute $\pi^* = \mathsf{S}_2(\mathsf{crs}, \tau, \mathsf{stmt})$.
13. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
14. Let $y^* = (r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*) \oplus \mathsf{F}(\mathsf{id}^*, x^*, j))$.
15. Let $P_{comp} = i\mathcal{O}(\text{Compute})$.
16. Output $(P_{comp}, (\mathsf{id}^*, c^*))$ and the queries of other honest users.

$\mathsf{Hyb}_{i:j:5}$. In this hybrid, we start using $\mathsf{sk}_2$ in the program instead of $\mathsf{sk}_1$.

We claim that the programs Compute in the two hybrids are functionally equivalent in the two hybrids and hence the indistinguishability follows by the security of $i\mathcal{O}$. This is because for all $c' = (c_1', c_2', \pi')$ such that $c' \neq c^*$, if $\pi'$ is accepted then $\mathsf{PKDec}(\mathsf{sk}_1, c_1') = \mathsf{PKDec}(\mathsf{sk}_2, c_2')$. Also, on input $(\mathsf{id}^*, c^*)$, the output of the program is identical in the two hybrids for all cloud inputs. Finally, on input $(\mathsf{id}, c^*)$ for some $\mathsf{id} \neq \mathsf{id}^*$, both programs output $\perp$ due to condition in Step 3. This proves that the two programs are functionally equivalent.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
2. Pick $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{PKGen}(1^\lambda)$, $(\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow \mathsf{PKGen}(1^\lambda)$.

**Compute**

**Constants**: $(\mathsf{id}^*, c^*, x^*, K_{\mathsf{id}^*,\mathsf{out}})$, $(j, y^*)$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K(\{(\mathsf{id}^*, c^*, j)\})$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

1. If $(\mathsf{id}, c) = (\mathsf{id}^*, c^*)$, do the following:
   - If $z = j$, output $y^*$.
   - Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, z))$.
   - If $z < j$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r))$ and end.
   - Else, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x^*, z))$ 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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
7. Output $y = \perp$ and end.

Figure 2.32: Program Compute

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 $\mathsf{id}$, 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. Let $r^*$ be chosen randomly.
8. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for $\mathsf{PRF}_2$.
9. Let $x^*$ be the input. Let $m_1^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}||\mathsf{flag}^*)$ and $m_2^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})||\mathsf{flag}^*)$.

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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.24.
12. Compute $\pi^* = \mathsf{S}_2(\mathsf{crs}, \tau, \mathsf{stmt})$.
13. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
14. Let $y^* = (r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*) \oplus \mathsf{F}(\mathsf{id}^*, x^*, j))$.
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**: $(\mathsf{id}^*, c^*, x^*, K_{\mathsf{id}^*,\mathsf{out}})$, $(j, y^*)$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K(\{(\mathsf{id}^*, c^*, j)\})$, verification key avk and common reference string crs.

**Input:** Identity id and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

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

Figure 2.33: Program Compute

$\mathsf{Hyb}_{i:j:6}$. In this hybrid, we change $m_1^*$ to include a punctured key $K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})$ instead of original key $K_{\mathsf{id}^*,\mathsf{out}}$. The two hybrids are indistinguishable by $\mathsf{IND}-\mathsf{CPA}$ security of public key encryption scheme $\mathsf{pke}$. This is because the hybrids do not use the secret $\mathsf{sk}_1$.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{PRG}$.
4. On receiving a corrupt user's identity $\mathsf{id}$, 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. Let $r^*$ be chosen randomly.
8. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for $\mathsf{PRF}_2$.
9. Let $x^*$ be the input. Let $\underline{m_1^* = (\mathsf{id}^*||x^*||t_{\mathsf{id}^*}||K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})||\mathsf{flag}^*)}$ 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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where $\mathsf{stmt}$ is defined in Figure 2.24.
12. Compute $\pi^* = \mathsf{S}_2(\mathsf{crs}, \tau, \mathsf{stmt})$.
13. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
14. Let $y^* = (r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*) \oplus \mathsf{F}(\mathsf{id}^*, x^*, j))$.
15. Let $P_{comp} = i\mathcal{O}(\text{Compute})$.
16. Output $(P_{comp}, (\mathsf{id}^*, c^*))$ and the queries of other honest users.

$\mathsf{Hyb}_{i:j:7}$. In this hybrid, we switch back to using $\mathsf{sk}_1$ in the program instead of $\mathsf{sk}_2$.

We claim that the programs compute in the two hybrids are functionally equivalent in the two hybrids and hence the indistinguishability follows by the security of $i\mathcal{O}$. this is because for all $c' = (c_1', c_2', \pi')$ such that $c' \neq c^*$, if $\pi'$ is accepted then $\mathsf{PKDec}(\mathsf{sk}_1, c_1') = \mathsf{PKDec}(\mathsf{sk}_2, c_2')$. Also, on input $(\mathsf{id}^*, c^*)$, the output of the program is identical in the two hybrids for all cloud inputs. Finally, on input $(\mathsf{id}, c^*)$ for some $\mathsf{id} \neq \mathsf{id}^*$, both programs output $\perp$ due to condition in Step 3. This proves that the two programs are functionally equivalent.

**Compute**

**Constants**: $(\mathsf{id}^*, c^*, x^*, K_{\mathsf{id}^*,\mathsf{out}})$, $(j, y^*)$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K(\{(\mathsf{id}^*, c^*, j)\})$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

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

Figure 2.34: Program Compute

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{id}$, 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. Let $r^*$ be chosen randomly.
8. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for $\mathsf{PRF}_2$.

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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.24.
12. Compute $\pi^* = \mathsf{S}_2(\mathsf{crs}, \tau, \mathsf{stmt})$.
13. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
14. Let $y^* = (r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*) \oplus \mathsf{F}(\mathsf{id}^*, x^*, j))$.
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**: $(\mathsf{id}^*, c^*, x^*, K_{\mathsf{id}^*,\mathsf{out}})$, $(j, y^*)$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K(\{(\mathsf{id}^*, c^*, j)\})$, verification key avk and common reference string crs.

**Input:** Identity id and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

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

Figure 2.35: Program Compute

$\mathsf{Hyb}_{i:j:8}$. In this hybrid, we change the constant in the program from $K_{\mathsf{id}^*,\mathsf{out}}$ to $K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})$.

We claim that the programs compute in the two hybrids are functionally equivalent with all but negligible probability and hence the indistinguishability follows by the security of $i\mathcal{O}$. This is because with high probability over the choice of $r^*$, it is true that $r^*$ is not in the image of the $\mathsf{PRF}_1(K,\cdot)$, and therefore this puncturing also does not change the functionality of the Compute program. Recall that $\mathsf{PRF}_1(K,\cdot)$ is a puncturable PRF such that its image is a negligible fraction of its co-domain (see Section 2.4).

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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. Let $r^*$ be chosen randomly.
8. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for $\mathsf{PRF}_2$.
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 $(\mathsf{crs},\tau) \leftarrow \mathsf{S}_1(1^\lambda,\mathsf{stmt})$, where stmt is defined in Figure 2.24.
12. Compute $\pi^* = \mathsf{S}_2(\mathsf{crs},\tau,\mathsf{stmt})$.
13. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
14. Let $y^* = (r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*) \oplus \mathsf{F}(\mathsf{id}^*, x^*, j))$.
15. Let $P_{comp} = i\mathcal{O}(\text{Compute})$.
16. Output $(P_{comp}, (\mathsf{id}^*, c^*))$ and the queries of other honest users.

$\mathsf{Hyb}_{i:j:9}$. In this hybrid, we change the value of $y^*$ to $(r^*, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*))$.

The indistinguishability of the two hybrids follows by pseudorandomness property of punctured key $K_{\mathsf{id}^*,\mathsf{out}}(\{r^*\})$. Because of this $\mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*)$ is indistinguishable from random

**Compute**

**Constants**: $(\mathsf{id}^*, c^*, x^*, \underline{K_{\mathsf{id},\mathsf{out}}(\{r^*\})})$, $(j, y^*)$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K(\{(\mathsf{id}^*, c^*, j)\})$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

1. If $(\mathsf{id}, c) = (\mathsf{id}^*, c^*)$, do the following:
   - If $z = j$, output $y^*$.
   - Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, z))$.
   - If $z < j$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r))$ and end.
   - Else, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x^*, z))$ 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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
7. Output $y = \perp$ and end.

Figure 2.36: Program Compute

string. This implies that $\mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*)$ is indistinguishable from $\mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r^*) \oplus \mathsf{F}(\mathsf{id}^*, x^*, j)$.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{id}$, 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. Let $r^*$ be chosen randomly.

8. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for $\mathsf{PRF}_2$.
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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.24.
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.

$\mathsf{Hyb}_{i:j:10}$. In this hybrid, we change back the key $K_{\mathsf{id}^*,\mathsf{out}}$ used in $m_1^*$, $m_2^*$ and the program to the original unpunctured key.

The indistinguishability follows via a sequence of hybrids similar to hybrids $\mathsf{Hyb}_{i:j:4}$ to $\mathsf{Hyb}_{i:j:8}$ using the two-key switching techniques.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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. Let $r^*$ be chosen randomly.
8. Choose a random PRF key $K_{\mathsf{id}^*,\mathsf{out}}$ for $\mathsf{PRF}_2$.
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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.24.

**Compute**

**Constants**: $(\mathsf{id}^*, c^*, x^*, K_{\mathsf{id},\mathsf{out}}(\{r^*\}))$, $(j, y^*)$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K(\{(\mathsf{id}^*, c^*, j)\})$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

1. If $(\mathsf{id}, c) = (\mathsf{id}^*, c^*)$, do the following:
   - If $z = j$, output $y^*$.
   - Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, z))$.
   - If $z < j$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r))$ and end.
   - Else, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x^*, z))$ 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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
7. Output $y = \perp$ and end.

Figure 2.37: Program Compute

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.

$\mathsf{Hyb}_{i:j:11}$. In this hybrid, we set $r^* = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, j))$ instead of random.

The indistinguishability follows from the pseudorandomness property of the punctured PRF $\mathsf{PRF}_1$.

**Compute**

**Constants**: $(\mathsf{id}^*, c^*, x^*, \underline{K_{\mathsf{id},\mathsf{out}}})$, $(j, y^*)$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K(\{(\mathsf{id}^*, c^*, j)\})$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

1. If $(\mathsf{id}, c) = (\mathsf{id}^*, c^*)$, do the following:
   - If $z = j$, output $y^*$.
   - Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, z))$.
   - If $z < j$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r))$ and end.
   - Else, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x^*, z))$ 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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
7. Output $y = \perp$ and end.

Figure 2.38: Program Compute

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{id}$, 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 $\mathsf{PRF}_2$.
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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.24.
11. Compute $\pi^* = \mathsf{S}_2(\mathsf{crs}, \tau, \mathsf{stmt})$.
12. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
13. Let $r^* = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, j))$.
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**: $(\mathsf{id}^*, c^*, x^*, K_{\mathsf{id},\mathsf{out}})$, $(j, y^*)$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K(\{(\mathsf{id}^*, c^*, j)\})$, verification key avk and common reference string crs.

**Input:** Identity id and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

1. If $(\mathsf{id}, c) = (\mathsf{id}^*, c^*)$, do the following:
   - If $z = j$, output $y^*$.
   - Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, z))$.
   - If $z < j$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r))$ and end.
   - Else, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x^*, z))$ 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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
7. Output $y = \perp$ and end.

Figure 2.39: Program Compute

$\mathsf{Hyb}_{i:j:12}$. In this hybrid, we remove the secondary "if" condition from the first step and the constants $(y^*, j)$, and un-puncture the key $K$. We move the value of $z = j$ to the second case.

The indistinguishability follows from the security of $i\mathcal{O}$.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{PRF}_2$.
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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.24.
11. Compute $\pi^* = \mathsf{S}_2(\mathsf{crs}, \tau, \mathsf{stmt})$.
12. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
13. Let $r^* = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, j))$.
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.

Now we describe the hybrid $\mathsf{Hyb}_i^{(4)}$ which is same as $\mathsf{Hyb}_{i:2^{\ell_z}:12}$. We underline the changes we did w.r.t. $\mathsf{Hyb}_i^{(3)}$.

Note that the Else statement in the first step is never invoked. Hence, we remove this condition in subsequent hybrids.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
2. Pick $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{PKGen}(1^\lambda)$, $(\mathsf{pk}_2, \mathsf{sk}_2) \leftarrow \mathsf{PKGen}(1^\lambda)$.

**Compute**

**Constants**: $(\mathsf{id}^*, c^*, x^*, K_{\mathsf{id},\mathsf{out}})$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

1. If $(\mathsf{id}, c) = (\mathsf{id}^*, c^*)$, do the following:
   - Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, z))$.
   - If $z < j + 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r))$ and end.
   - Else, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x^*, z))$ 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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
7. Output $y = \perp$ and end.

Figure 2.40: Program Compute

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 $\mathsf{id}$, 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 $\mathsf{PRF}_2$.
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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where stmt is defined in Figure 2.24.

11. Compute $\pi^* = \mathsf{S}_2(\mathsf{crs}, \tau, \mathsf{stmt})$.
12. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
13. Let $r^* = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, 2^{\ell_z}))$.
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**: $(\mathsf{id}^*, c^*, x^*, K_{\mathsf{id},\mathsf{out}})$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

1. If $(\mathsf{id}, c) = (\mathsf{id}^*, c^*)$, do the following:
   - Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, z))$.
   - If $z < 2^{\ell_z} + 1$, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r))$ and end.
   - Else, output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x^*, z))$ 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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
7. Output $y = \perp$ and end.

Figure 2.41: Program Compute

$\mathsf{Hyb}_i^{(5)}$**.** In this hybrid, we change the value of $\mathsf{flag}^*$ to 1 instead of 0 and $t_{\mathsf{id}^*}$ to be a random string of appropriate length. We also set $x^* = 0^{\ell_{\mathsf{in}}}$.

The indistinguishability follows via a sequence of hybrids similar to hybrids $\mathsf{Hyb}_{i:j:5}$ to $\mathsf{Hyb}_{i:j:8}$

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 when the input is $c^*$. Moreover, because of the initial "if" condition, there is no check on $\mathsf{flag}^*$ or $t_{\mathsf{id}^*}$. Hence, while switching keys for decryption, functional equivalence follows in a straight-forward manner. Moreover, we will use the fact that in all these hybrids we are simulating the $\mathsf{crs}$ as well as the proof $\pi^*$.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{PRG}$.
4. On receiving a corrupt user's identity $\mathsf{id}$, 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. Pick $t_{\mathsf{id}^*}$ 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 $\mathsf{PRF}_2$.
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 $(\mathsf{crs}, \tau) \leftarrow \mathsf{S}_1(1^\lambda, \mathsf{stmt})$, where $\mathsf{stmt}$ is defined in Figure 2.24.
11. Compute $\pi^* = \mathsf{S}_2(\mathsf{crs}, \tau, \mathsf{stmt})$.
12. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
13. Let $r^* = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, 2^{\ell_z}))$.
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.

$\mathsf{Hyb}_i^{(6)}$**.** In this hybrid, we start generating the $\mathsf{crs}$ and the proof $\pi^*$ honestly.

The indistinguishability of the hybrids follows by computational zero-knowledge property of the NIZK used.

**Compute**

**Constants**: $(\mathsf{id}^*, c^*, x^*, K_{\mathsf{id},\mathsf{out}})$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

1. If $(\mathsf{id}, c) = (\mathsf{id}^*, c^*)$, do the following:
   - Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, z))$.
   - Output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r))$ 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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
7. Output $y = \perp$ and end.

Figure 2.42: Program Compute

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{id}$, 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. Pick $t_{\mathsf{id}^*}$ 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 $\mathsf{PRF}_2$.
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 $\mathsf{crs} \leftarrow \mathsf{NIZKSetup}(1^\lambda)$, where stmt is defined in Figure 2.24.

11. Compute $\pi^* = \mathsf{NIZKProve}(\mathsf{crs}, \mathsf{stmt}, (m_1^*, r_1, r_2))$, where stmt is defined in Figure 2.24.
12. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
13. Let $r^* = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, 2^{\ell_z}))$.
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**: $(\mathsf{id}^*, c^*, x^*, K_{\mathsf{id},\mathsf{out}})$, Secret key $\mathsf{sk}_1$, puncturable PRF key $K$, verification key avk and common reference string crs.

**Input:** Identity id and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

1. If $(\mathsf{id}, c) = (\mathsf{id}^*, c^*)$, do the following:
   - Compute $r = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, z))$.
   - Output $y = (r, \mathsf{PRF}_2(K_{\mathsf{id}^*,\mathsf{out}}, r))$ 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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
7. Output $y = \perp$ and end.

Figure 2.43: Program Compute

$\mathsf{Hyb}_i^{(7)}$**.** In this hybrid, we remove the initial "if" condition and the constants $(\mathsf{id}^*, c^*, x^*, K_{\mathsf{id},\mathsf{out}})$. The indistinguishability follows from the security of $i\mathcal{O}$.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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. Pick $t_{\mathsf{id}^*}$ 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 $\mathsf{PRF}_2$.
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 $\pi^* = \mathsf{NIZKProve}(\mathsf{crs}, \mathsf{stmt}, (m_1^*, r_1, r_2))$, where stmt is defined in Figure 2.24.
12. Set $c^* = (c_1^*, c_2^*, \pi^*)$.
13. Let $r^* = \mathsf{PRF}_1(K, (\mathsf{id}^*, c^*, 2^{\ell_z}))$.
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 $\mathsf{sk}_1$, puncturable PRF key $K$, verification key avk and common reference string crs.

**Input:** Identity id and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
6. Output $y = \perp$ and end.

Figure 2.44: Program Compute

After we go through the sequence of hybrids for each honest user, we obtain the hybrid $\mathsf{Hyb}_m^{(7)}$. 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 encodes his actual input $x$ with $\mathsf{flag} = 0$ and a valid authentication $t_{\mathsf{id}}$, and obtains output according to the function $\mathsf{F}$ on $(\mathsf{id}, x, z)$. 2) The honest user encodes $0^{\ell_{\mathsf{in}}}$ with $\mathsf{flag} = 1$ and uniformly random $t_{\mathsf{id}}$, and receives encoding of $0$ as output (without invoking the function $\mathsf{F}$.)

Below, we describe a hybrid $\widetilde{\mathsf{Hyb}_1}$ which is equivalent to $\mathsf{Hyb}_m^{(7)}$.

$\widetilde{\mathsf{Hyb}_1}$**:** This hybrid is same as $\mathsf{Hyb}_m^{(7)}$. In the hybrid $\mathsf{Hyb}_m^{(7)}$, all the user queries will have $\mathsf{flag} = 1$, $t_{\mathsf{id}}$ will be a random string, and input will be $0^{\ell}_{\mathsf{in}}$. Hence, the program Compute will not invoke the function $\mathsf{F}$ for any of the honest users.

The underlined statement summarizes the main difference between $\widetilde{\mathsf{Hyb}_0}$ and $\widetilde{\mathsf{Hyb}_1}$.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $\mathsf{PRG}$.
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 $\mathsf{id}$, 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, $t_{\mathsf{id}}$ is set to a random string of appropriate length, $\mathsf{flag}$ and is set to $1$ input is set to $0^{\ell_{\mathsf{in}}}$. Ciphertexts $(c_1, c_2)$ and proof $\pi$ is generated as in real execution.
8. Output $P_{comp}$ and queries of all honest users $\mathsf{id}_1, \ldots, \mathsf{id}_m$.

$\widetilde{\mathsf{Hyb}_2}$**:** 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.

**Compute**

**Constants**: Secret key $\mathsf{sk}_1$, puncturable PRF key $K$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
6. Output $y = \perp$ and end.

Figure 2.45: Program Compute

The two hybrids are indistinguishable by security properties of $\mathsf{FakeAuthGen}$ (see Section 2.1.5). Note that both hybrids do not depend on $\mathsf{ask}$ and need only the valid authentications for corrupt users.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $(\mathsf{avk}, \mathsf{ask}) \leftarrow \mathsf{FakeAuthGen}(1^\lambda, \mathcal{I})$ 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 $\mathsf{id}$, 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. As noted before, $\mathsf{AuthProve}$ still returns valid authentication for all users in $\mathcal{I}$.
7. For each of the honest users, $t_{\mathsf{id}}$ is set to a random string of appropriate length, $\mathsf{flag}$ is set to 1 and input is set to $0^{\ell_{\mathsf{in}}}$. Ciphertexts $(c_1, c_2)$ and proof $\pi$ 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 $\mathsf{sk}_1$, puncturable PRF key $K$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{F}(\mathsf{id}, x, z))$ and end.
6. Output $y = \perp$ and end.

Figure 2.46: Program Compute

$\widetilde{\mathsf{Hyb}_3}$**:** This is the most important hybrid, where we change the program to use $\mathsf{G}$ instead of $\mathsf{F}$.

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 $\mathsf{avk}$. 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 $\mathsf{F}$ and $\mathsf{G}$ are equivalent for corrupt users.

1. Choose PRF key $K$ at random for $\mathsf{PRF}_1$.
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 $(\mathsf{avk}, \mathsf{ask}) \leftarrow \mathsf{FakeAuthGen}(1^\lambda, \mathcal{I})$ with respect to cover-free set system $\mathcal{T}$ and $\mathsf{PRG}$.
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 $\mathsf{id}$, 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. As noted before, $\mathsf{AuthProve}$ still returns valid authentication for all users in $\mathcal{I}$.
7. For each of the honest users, $t_{\mathsf{id}}$ is set to a random string of appropriate length, $\mathsf{flag}$ is set to

1 and input is set to $0^{\ell_{\mathsf{in}}}$. Ciphertexts $(c_1, c_2)$ and proof $\pi$ 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 $\mathsf{sk}_1$, puncturable PRF key $K$, verification key $\mathsf{avk}$ and common reference string $\mathsf{crs}$.

**Input:** Identity $\mathsf{id}$ and ciphertext $c = (c_1, c_2, \pi)$ from the user, and input $z$ from the cloud.

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 $r = \mathsf{PRF}_1(K, (\mathsf{id}, c, z))$.
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 $y = (r, \mathsf{PRF}_2(K_{\mathsf{id},\mathsf{out}}, r) \oplus \mathsf{G}(\mathsf{id}, x, z))$ and end.
6. Output $y = \perp$ and end.

Figure 2.47: Program Compute

Finally, using a similar sequence of hybrids (in the reverse order) we can move from $\widetilde{\mathsf{Hyb}_3}$ to a hybrid which corresponds to real execution using $\mathsf{G}$ and honest party inputs $x'_1, \ldots, x'_m$.

# CHAPTER 3

# Optimizing Obfuscation: Avoiding Barrington's Theorem

## 3.1 Preliminaries

We denote the security parameter by $\lambda$. We use $[n]$ to denote the set $\{1, \ldots, n\}$.

### 3.1.1 "Virtual Black-Box" Obfuscation in an Idealized Model

Let $\mathcal{M}$ be some oracle. Below we define "Virtual Black-Box" obfuscation in the $\mathcal{M}$-idealized model taken verbatim from [BGK14]. In this model, both the obfuscator and the evaluator have access to the oracle $\mathcal{M}$. However, the function family that is being obfuscated does not have access to $\mathcal{M}$.

**Definition 9.** *For a (possibly randomized) oracle $\mathcal{M}$, and a circuit class $\left\{\mathcal{C}_\ell\right\}_{\ell\in\mathbb{N}}$, we say that a uniform PPT oracle machine $\mathcal{O}$ is a "Virtual Black-Box" Obfuscator for $\left\{\mathcal{C}_\ell\right\}_{\ell\in\mathbb{N}}$ in the $\mathcal{M}$-idealized model, if the following conditions are satisfied:*

- *Functionality: For every $\ell \in \mathbb{N}$, every $C \in \mathcal{C}_\ell$, every input $x$ to $C$, and for every possible coins for $\mathcal{M}$:*

$$\Pr[(\mathcal{O}^{\mathcal{M}}(C))(x) \neq C(x)] \leqslant \mathrm{negl}(|C|) \ ,$$

  *where the probability is over the coins of $\mathcal{C}$.*

- *Polynomial Slowdown: There exist a polynomial $p$ such that for every $\ell \in \mathbb{N}$ and every $C \in \mathcal{C}_\ell$, we have that $|\mathcal{O}^{\mathcal{M}}(C)| \leqslant p(|C|)$.*

- *Virtual Black-Box: For every PPT adversary $\mathcal{A}$ there exist a PPT simulator $\mathsf{Sim}$, and a negligible function $\mu$ such that for all PPT distinguishers $D$, for every $\ell \in \mathbb{N}$ and every $C \in \mathcal{C}_\ell$:*

$$\Big|\Pr[D(\mathcal{A}^{\mathcal{M}}(\mathcal{O}^{\mathcal{M}}(C))) = 1] -$$

$$\Pr[D(\mathsf{Sim}^C(1^{|C|})) = 1]\big| \leqslant \mu(|C|) \ ,$$

*where the probabilities are over the coins of $D$,$\mathcal{A}$,$\mathsf{Sim}$,$\mathcal{O}$ and $\mathcal{M}$.*

Note that in this model, both the obfuscator and the evaluator have access to the oracle $\mathcal{M}$ but the function family that is being obfuscated does not have access to $\mathcal{M}$.

### 3.1.2 Boolean Formulae

A boolean circuit for a function $f : \{0,1\}^\ell \to \{0,1\}$ is a directed acyclic graph (DAG). The vertices in this graph are either input variables or gates. We assume that all the gates in the circuit have fan-in at most 2. The outdegree of an output gate is 0 and it is at least 1 for all other vertices. The *fan-out* of a gate is the out-degree of that gate. In this work, we consider a special type of circuits called formulae. A boolean formula is a boolean circuit where the fan-out of each gate is 1. A formula can be viewed as binary tree where the root is the output gate. We define the *size* of a formula to be the number of leaves in this binary tree.

### 3.1.3 Branching Programs

In this section we define a non-deterministic branching program, and several types of layered branching programs that are useful for our purpose.

A *non-deterministic branching program* (BP) is a finite directed acyclic graph with two special nodes, a source node and a sink node, also referred to as an "accept" node. Each non-sink node is labeled with a variable $x_i$ and can have arbitrary out-degree. [1] Each of the out-edges is either labeled with $x_i = 0$ or $x_i = 1$. The sink node has out-degree 0. In the following, we denote a branching program by $\mathsf{BP}$ and denote the restriction of the branching program consistent with input $x$ by $\mathsf{BP}|_x$. An input $x \in \{0,1\}^\ell$ is accepted if and only if there is a path from the source node to the accept node in $\mathsf{BP}|_x$. Note that an input can have multiple computation paths in $\mathsf{BP}|_x$. The *length* of the BP is the maximum length of any such path in the graph. The *size* $s$ of the branching program is the total number of non-sink nodes in the graph, i.e., total number of nodes minus 1.

[1] We assume for simplicity that the out-degree is bound by some fixed constant (say 4), so that the total number of paths is bounded by $2^{O(s)}$ as opposed to $s^s$.

A *layered* branching program is a branching program such that nodes can be partitioned into a sequence of layers where all the nodes in one layer are labeled with the same variable and edges go only from nodes of one layer to the next. We can assume without loss of generality that the first layer contains the source node and the last layer contains the sink node. The *length* $n$ of a layered branching program is the number of layers minus 1 and its *width* $w$ is the maximum number of nodes in any layer. It will be convenient to assume that a layered BP has exactly $w$ nodes in each layer. We denote the $k^{th}$ node in layer $i$ by $v_{i,k}$ for $0 \leqslant i \leq n$ and $k \in [w]$.

The following nonstandard types of branching programs will be useful for our purposes. A *special* layered branching program is a layered branching program with the following additional property. For each layer $i$, $0 \leqslant i < n$, and each $k \in [w]$, there is an edge from $v_{i,k}$ to $v_{i+1,k}$ labeled by both $0$ and $1$ (namely, this edge is consistent with all inputs).

Finally, we define an *invertible* layered branching program as follows. An invertible layered branching program is a type of a layered branching program. Corresponding to each $i \in [n]$, we define two $w \times w$ matrices $B_{i,0}$ and $B_{i,1}$ as follows: $B_{i,b}[x, y] = 1$ if and only if there is an edge from node $v_{i-1,x}$ to node $v_{i,y}$ labeled with $b$. Otherwise, $B_{i,b}[x, y] = 0$. We say that the layered branching program is *invertible* if $B_{i,b}$ is full rank for all $i \in [n]$ and $b \in \{0, 1\}$.

### 3.1.4 Relaxed Matrix Branching Programs

In this section we define the original notion of matrix branching programs used in [GGH13b] followed by our notion of relaxed matrix branching programs.

**Definition 10** (Matrix Branching Program (MBP))**.** *[BGK14] A matrix branching program of width $w$ and length $n$ for $\ell$-bit inputs is given by a $w \times w$ permutation matrix $P_{\mathsf{reject}}$ such that $P_{\mathsf{reject}} \neq I_{w\times w}$ and by a sequence:*

$$\mathsf{BP} = (\mathsf{inp}, B_{i,0}, B_{i,1})_{i\in[n]},$$

*where $B_{i,b}$, for $i \in [n], b \in \{0, 1\}$, are $w \times w$ permutation matrices and $\mathsf{inp} : [n] \to [\ell]$ is the evaluation function of $\mathsf{BP}$. The output of $\mathsf{BP}$ on input $x \in \{0, 1\}^\ell$, denoted by $\mathsf{BP}(x)$, is determined*

*as follows:*

$$\mathsf{BP}(x) = \begin{cases} 1 & \text{if } \prod_{i=1}^{n} B_{i,x_{\mathsf{inp}(i)}} = I_{w\times w} \\ 0 & \text{if } \prod_{i=1}^{n} B_{i,x_{\mathsf{inp}(i)}} = P_{\mathsf{reject}} \\ \perp & \text{otherwise} \end{cases}$$

*We say that a family of MBPs are* input-oblivious *if all programs in the family share the same parameters $w, n, \ell$ and the evaluation function* $\mathsf{inp}$.

Barrington [Bar86] showed that every circuit with depth $d$ and fan-in 2 can be represented by a MBP of length at most $4^d$ and width 5. Previous works [GGH13b, BR14b, BGK14] used MBPs obtained by applying Barrington's theorem to obfuscate circuits. Since the MBP obtained has length exponential in the depth of the circuit, this turns out to be a bottleneck for efficiency. In this work, we will use relaxed MBPs towards obfuscation.

In MBP after evaluation we either get $I_{w\times w}$ or $P_{\mathsf{reject}}$ which decides the output. We relax this requirement as follows. We only require that a single designated entry in the final product is either 0 or non-zero depending on the output and place no restriction on other entries. Note that this is a further relaxation of the notion considered in [PST14]. More formally, we define the notion of relaxed matrix branching programs as follows.

**Definition 11** (Relaxed MBP (RMBP))**.** *Let $R$ be any finite ring. A relaxed matrix branching program (over R) of size $w$ and length $n$ for $\ell$-bit inputs is given by a sequence:*

$$\mathsf{BP} = (\mathsf{inp}, B_{i,0}, B_{i,1})_{i\in[n]},$$

*where each $B_{i,b}$ is a $w \times w$ full-rank, i.e. invertible, matrix and* $\mathsf{inp} : [n] \to [\ell]$ *is the evaluation function of* BP*. The output of* BP *on input $x \in \{0,1\}^\ell$, denoted by* $\mathsf{BP}(x)$*, is determined as follows:*

$$\mathsf{BP}(x) = 1 \text{ if and only if } \Big(\prod_{i=1}^{n} B_{i,x_{\mathsf{inp}(i)}}\Big)[1, w] \neq 0$$

**Dual-input Relaxed Matrix Branching Programs.** Similar to [BGK14], for the proof of obfuscation we would need to consider dual input matrix branching programs. We define dual input RMBP as follows.

**Definition 12** (Dual Input RMBP)**.** *Let $R$ be a finite ring. A dual-input relaxed matrix branching program (over R) of size $w$ and length $n$ for $\ell$-bit inputs is given by a sequence:*

$$\mathsf{BP} = (\mathsf{inp}_1, \mathsf{inp}_2, B_{i,b_1,b_2})_{i\in[n],b_1,b_2\in\{0,1\}},$$

*where each $B_{i,b_1,b_2}$ is a $w \times w$ full-rank matrix and $\mathsf{inp}_1, \mathsf{inp}_2 : [n] \to [\ell]$ are the evaluation functions of $\mathsf{BP}$. The output of $\mathsf{BP}$ on input $x \in \{0,1\}^\ell$, denoted by $\mathsf{BP}(x)$, is determined as follows:*

$$\mathsf{BP}(x) = 1 \text{ if and only if } \Big(\prod_{i=1}^{n} B_{i,x_{\mathsf{inp}_1(i)},x_{\mathsf{inp}_2(i)}}\Big)[1, w] \neq 0$$

*We say that a family of matrix branching programs is* input-oblivious *if all programs in the family share the same parameters $w, n, \ell$ and the evaluation functions $\mathsf{inp}_1, \mathsf{inp}_2$.*

For the purpose of obfuscation we would consider *dual input oblivious relaxed matrix branching programs*.

## 3.2 From Branching Programs to Relaxed Matrix Branching Programs

In this section we describe a sequence of transformations which allow us to transform a non-deterministic branching program of size $s$ to a relaxed matrix branching program of width $2(s+1)$ and length $s$. These transformations are close variants of similar transformations from [FKN94]. The main steps are to convert a non-deterministic branching program to a special layered branching program, then to an invertible layered branching program, and finally to an RMBP. These intermediate steps can be independently useful, as they allow for more efficient transformations of special or invertible layered branching programs into RMBPs.

**Branching program to special layered branching program.**

**Lemma 1.** *Any non-deterministic branching program $\mathsf{BP}$ of size $s$ can be efficiently converted to an equivalent special layered branching program $\mathsf{SLBP}$ of length $s$ and width $s + 1$.*

*Proof Sketch.* Recall that since we do not include the sink node in the size of a BP, a BP of size $s$ has $s + 1$ nodes. Given a branching program with $s + 1$ nodes, first do a topological sort of the

nodes, say $\{v_1, \ldots, v_{s+1}\}$. Without loss of generality, assume that $v_1$ is the source node and $v_{s+1}$ is the sink node. We construct a special layered branching program with $s+1$ layers where each layer has $s+1$ nodes as follows. Let the nodes in layer $i$ be $\{v_{i,1}, \ldots, v_{i,s+1}\}$. That is, we denote $k^{th}$ node in layer $i$ by $v_{i,k}$. For each $0 \leqslant i < s$ we do the following: Let node $v_{i+1}$ be labeled with $x_j$ in original BP. Then, we label layer $i$ with $x_j$. We draw the outgoing edges from node $v_{i,i+1}$ to node $v_{i+1,k}$ if there was an edge from $v_{i+1}$ to $v_k$ in the original BP. Labels on the edges are retained. Now, we also add edges between $v_{i,k}$ to $v_{i+1,k}$ for all $k \in [s+1]$ for both $x_j = 0$ and $x_j = 1$.

It is easy to see that there is a path between source, i.e., $v_1$ to the accept node, i.e., $v_s$ in the original branching program if and only if there is a path from $v_{0,1}$ to $v_{s,s+1}$.

**Special layered branching program to an invertible layered branching program.**

**Lemma 2.** *Any special layered branching program* SLBP *of length* $n$ *and width* $w$ *can be efficiently converted to an equivalent invertible layered branching program* ILBP *of length* $n$ *and width* $2w$.

*Proof Sketch.* Let edges in layer $i$ be $\{v_1, \ldots, v_w\}$. For each layer $i$, where $0 \leqslant i < n$, we add $w$ dummy nodes, say $\{v_{i,w+1}, \ldots, v_{i,2w}\}$ and add the following edges. For each layer $i$ and $j \in [w]$ add edges from $v_{i,j}$ to $v_{i+1,w+j}$ and from $v_{i,w+j}$ to $v_{i+1,j}$.

It is easy to see that the new layered branching program is invertible. More precisely, the columns can be re-arranged so that the matrices obtained are upper-triangular with all the main diagonal entries set to 1. It also easy to observe that if $\mathsf{SLBP}(x) = 1$ then $\mathsf{ILBP}(x) = 1$. For the other direction, note that adding these extra nodes and edges does not create a path between any two original nodes if there was no path before.

**Invertible layered branching program to relaxed matrix branching program.**

**Lemma 3.** *Any invertible layered branching program* ILBP *of length* $n$ *and width* $w$ *can be efficiently converted to an equivalent relaxed matrix branching program* RMBP *of length* $n$ *and width* $w$.

*Proof.* Consider a large enough [2] prime $p$. Corresponding to each layer in the layered branching

[2] In the following construction, we use the fact that the prime $p$ is large enough so that there are no wrap-arounds

program, we will have two $(w \times w)$ matrices $B_{i,0}$ and $B_{i,1}$ over $\mathbb{Z}_p$. Let the label of this layer be $x_j$ for some $j \in [\ell]$. For $i \in [n]$ and $b \in \{0,1\}$, define $B_{i,b}$ as follows:

For any $x, y \in [w]$, set $B_{i,b}[x,y] = 1$ if there is an edge between $v_{i-1,x}$ to $v_{i,y}$ labeled $x_j = b$. Set the rest of the entries in $B_{i,b}$ to be 0. We define $\mathsf{inp}$ to be a function from $[n]$ to $\ell$, where $\ell$ is the input length of $\mathsf{ILBP}$. We set $\mathsf{inp}(i) = j$ if all the nodes in the $i^{th}$ layer depend on the $j^{th}$ input bit.

Since we are given an invertible layered branching program, it is easy to see that all the matrices are full-rank. Without loss of generality, let the source node be the node $v_{0,1}$ and accept node be the node $v_{n,w}$ of the invertible layered branching program. Then,

**Claim 2.** *Consider an input $x \in \{0,1\}^\ell$. Denote the product $\prod_{i=1}^{n} B_{i,x_{\mathsf{inp}(i)}}$ by $P$. Then, $P[1,w] \geqslant 1$ if and only if $\mathsf{ILBP}(x) = 1$.*

*Proof Sketch:* We prove this via induction on the number of layers in the branching program. Intuitively, we will prove that the following invariant is maintained. Let $P_j = \prod_{i=1}^{j} B_{i,x_{\mathsf{inp}(i)}}$. Then, $P_j[x,y]$ will denote the number of paths from $v_{0,x}$ to $v_{j,y}$. In particular, $P[1,w]$ captures the number of paths from the source node to the sink node. And hence, $P[1,w]$ is non-zero iff $\mathsf{ILBP}(x) = 1$.

We argue this by induction. We define graph $G_j$, for $j \in [n]$, to be a subgraph of $\mathsf{ILBP}|_x$ as follows. It consists of all the vertices in the layers $\mathcal{L}_1, \ldots, \mathcal{L}_{j+1}$ and any two vertices in $G_j$ have an edge if and only if the correpsonding two vertices in $\mathsf{ILBP}|_x$ have an edge. We will denote the vertex set associated to $G_j$ as $V_j$. Without loss of generality we will assume that $V_j = \{1, \ldots, 2(j+1)\}$, since each layer has two vertices.

At each point in the induction we maintain the invariant that $P_j[u,v] = c_{u,v}$, where $P_j = \prod_{i=1}^{j} B_{i,x_{\mathsf{inp}(i)}}$ and $u, v \in V_j$ and $c_{u,v}$ is the number of possible paths from $u$ to $v$ in $G_j$.

The base case in the induction step is for the case of $G_1$ and the invariant follows from the definition of $G_1$. We now proceed to the induction hypothesis. Assume that the matrices $(B_{i,x_{\mathsf{inp}(i)}})_{i \in [j]}$, for $j < n$ is such that their product, which is $P_j$, satisfies the condition that $P_j[u,v]$ is the number of paths from $u$ to $v$ in graph $G_j$. Consider the product $P_{j+1} = \prod_{i=1}^{j+1}(B_{i,x_{\mathsf{inp}(i)}})$ which is essentially the product $P_j \cdot B_{j+1,x_{\mathsf{inp}(j+1)}}$. Now, consider $P_{j+1}[u,v] = \sum_{i=1}^{w} P_j[u,i] B_{j+1,x_{\mathsf{inp}(j+1)}}[i,v]$ for $u, v \in V_{j+1}$. Each term indicates the total number of paths from $u$ to $v$ with $i$ as an intermediate

while multiplying the matrices. In particular, assume that $p = 2^{\Omega(n)}$.

vertex in the graph $G_{j+1}$. Note that an intermediate vertex of any path of length at least 2 in $G_{j+1}$ should be in $V_j$. And the summation of all these terms indicates the total number of paths from $u$ to $v$ in $G_{j+1}$.

We have established that $P_n[u, v]$ represents the number of paths from the $u$ to $v$ in graph $G_n$. But $G_n$ is nothing but the graph $\mathsf{ILBP}|_x$ and $P_n$ is nothing but the matrix $P$. This shows that $P[u, v]$ denotes the number of paths from $u$ to $v$ in graph $\mathsf{ILBP}|_x$ and more specifically, $P[1, w]$ is non-zero iff $\mathsf{ILBP}(x) = 1$. This proves the lemma.

□

**Theorem 4.** *Any non-deterministic branching program* $\mathsf{BP}$ *of size* $s$ *can be efficiently converted to an equivalent relaxed matrix branching program* $\mathsf{RMBP}$ *of length* $s$ *and width* $2(s + 1)$.

*Proof.* It follows directly from Lemmas 1, 2 and 3. □

**Converting relaxed matrix branching program to dual input oblivious relaxed matrix branching program.** First note that if the family of invertible layered matrix branching programs is input oblivious, then the relaxed matrix branching program obtained from the above transformation would also be input oblivious. If that is not the case, we can convert it to a dual-input relaxed matrix branching program by incurring a multiplicative cost of $\ell$ in the length of the branching program. More formally,

**Lemma 4.** *Any relaxed matrix branching program* $\mathsf{RMBP} = (\mathsf{inp}, B_{i,0}, B_{i,1})_{i\in[n]}$ *of length* $n$ *and width* $w$ *can be efficiently converted to a dual-input oblivious relaxed matrix branching program of length at most* $n\ell$ *and width* $w$.

*Proof.* We first make our relaxed matrix branching program oblivious, i.e. make the evaluation function $\mathsf{inp}$ independent of the formula $\mathsf{F}$. Wlog, assume that the length of the relaxed matrix branching program, $n$, is a multiple of $(\ell - 1)$, i.e. $n = k \cdot (\ell - 1)$ for some $k \in \mathbb{N}$. If this not the case, add at most $(\ell - 2)$ pairs of identity matrices of dimension $w \times w$ to the relaxed matrix branching program. We will use this assumption while making the branching program dual input. Now we will describe a new (relaxed) matrix branching program of length $n' = n \cdot \ell$ and width $w$ and evaluation function $\mathsf{inp}_1$ as follows:

- Define $\mathsf{inp}_1(i) = i \bmod \ell$ for all $i \in [n]$.

- For each $j \in [n]$, $M_{(j-1)\cdot\ell+\mathsf{inp}(j),b} = B_{j,b}$ for $b \in \{0,1\}$. Rest all matrices are set to $I_{w\times w}$.

Informally the above transformation can be described as follows: In $j^{th}$ block of $\ell$ matrices, all the matrices are identity matrices apart from the matrices at index $\mathsf{inp}(j)$. At this index, we place the two non-trivial matrices $B_{j,0}$ and $B_{j,1}$ which help in actual computation.

**Claim 3.** *For any input* $x \in \{0,1\}^\ell$, $\prod_{i=1}^{n} B_{i,x_{\mathsf{inp}(i)}} = \prod_{i=1}^{n'} M_{i,x_{\mathsf{inp}_1(i)}}$.

Now we make the above relaxed matrix branching program dual-input, by pairing the input position used at each index with a dummy input position in an oblivious manner which is independent of the formula. For convenience of notation, we will also ensure that each pair of input bits is used as the selector same number of times. We will ensure that at any index of the RMBP, the two input positions used are distinct, i.e. $\mathsf{inp}_1(i) \neq \mathsf{inp}(i)$ for any $i \in [\ell]$. We define the evaluation function $\mathsf{inp}_2$ as follows: Consider $i$ of the form $k_1\ell(\ell-1) + k_2\ell + k_3$ then

$$\mathsf{inp}_2(i) = ((k_2 + k_3) \bmod \ell) + 1$$

□

### 3.2.1 From Formula to Relaxed Matrix Branching Program

**Direct construction.** Transforming formulas to branching programs is a well studied problem [Mas76, Bar86, Cle90, SWW99]. In particular, it is well known [Mas76] that formula of size $s$ over AND, OR, and NOT gates can be converted to a branching program of essentially the same size; for self containment, we describe such a transformation next which satisfies the following lemma.

**Lemma 5.** *Any formula of size $s$ can be converted to a branching program of size $s$.*

#### 3.2.1.1 From Formula to Branching Programs

In this section, we give a transformation of boolean formulas over AND and NOT gates to a branching program. Note that any formula over AND, OR and NOT gates can be converted to a formula over AND and NOT gates of the same size.

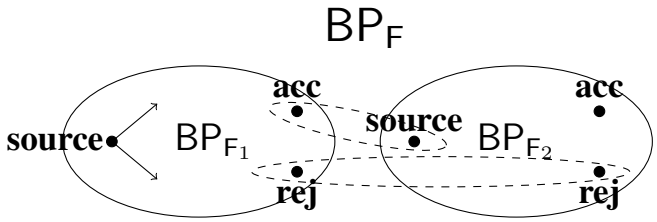


Figure 3.1: The branching program for an AND gate.

Consider a formula, denoted by $\mathsf{F}$. We inductively transform $\mathsf{F}$ to a branching program $\mathsf{BP}$. Our construction will maintain a stronger induction hypothesis. There will be two sink nodes, "accept" and "reject." Also, there will be a path from the source to the accept iff the output is 1 and there will be a path from the source to the reject iff the output is 0.

The base case corresponds to an input wire $w$. Let input variable be $x_i$. We construct a branching program for $w$, denoted by $\mathsf{BP}_w$ consists of three nodes denoted by source, acc and rej. We add an edge labeled 0 from source to rej and an edge labeled 1 from source to acc. We label the source with $x_i$.

We proceed to the induction hypothesis. Consider a gate $G$.

**Case (1) AND gate**:- Let $\mathsf{F}_1$ and $\mathsf{F}_2$ be two sub-formulae such that their output wires are fed to $\mathsf{F}$. Let $\mathsf{BP}_{\mathsf{F}_1}$ and $\mathsf{BP}_{\mathsf{F}_2}$ be the branching programs for $\mathsf{F}_1$ and $\mathsf{F}_2$, respectively. We construct a branching program for $\mathsf{F}$ as follows (see Figure 3.1). We merge the accept node of $\mathsf{BP}_{\mathsf{F}_1}$ with the source node of $\mathsf{BP}_{\mathsf{F}_2}$. Similarly, merge the reject node of $\mathsf{BP}_{\mathsf{F}_1}$ with the reject node of $\mathsf{BP}_{\mathsf{F}_2}$.

**Case (2) NOT gate**:- Let $\mathsf{F}'$ be the sub-formula such that the output wire of $\mathsf{F}'$ is fed into the gate $G$. Let $\mathsf{BP}_{\mathsf{F}'}$ be a branching program for $\mathsf{F}'$. To construct the branching program for $\mathsf{F}$ we simply rename accept node of $\mathsf{BP}_{\mathsf{F}'}$ as reject node for $\mathsf{BP}_{\mathsf{F}}$. We also rename reject node of $\mathsf{BP}_{\mathsf{F}'}$ as accept node for $\mathsf{BP}_{\mathsf{F}}$.

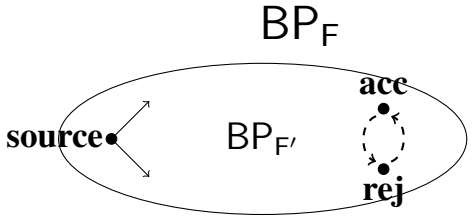


Figure 3.2: The branching program for a NOT gate.

Note that once the transformation is complete for the formula, the final reject node can be deleted. So our final construction will only have one sink node, the "accept" node.

It is easy to see that the above described layered branching program correctly evaluates the formula $\mathsf{F}$. More formally,

**Lemma 6.** *For every input $x \in \{0,1\}^l$, we have $\mathsf{F}(x) = 1$ if and only if $\mathsf{BP}_{\mathsf{F}}(x) = 1$. Moreover, for a formula of size $s$, the size of the branching program $\mathsf{BP}_{\mathsf{F}}$ is at most $s$.*

*Proof Sketch.* It follows by an induction on the structure of the formula by noting that the number of leaves in a formula is the sum of the leaves of the left sub-tree and the right sub-tree.

**Completing the transformation.** Using the transformations described in the previous section, we obtain the following.

**Theorem 5.** *Any formula of size $s$ can be efficiently converted to an equivalent relaxed matrix branching program of width $2(s+1)$ and length $s$. Moreover, it can be converted to a dual input oblivious matrix branching program of width $2(s+1)$ and length $s\ell$.*

**Keyed formulas.** Consider the class of keyed formulas, namely a class of formulas of the form $f_z(x) = \phi(z, x)$ such that $\phi$ is a formula of size $s$. While obfuscating this class of formulas, we only need to hide the key $z$ since $\phi$ is public. Since we do not require the matrix branching program to be input oblivious, we do not incur the additional factor of $\ell$ in the length of the matrix branching program. So the length of the branching program for this class of functions is at most $s$.

**Alternate approach.** We note that there is an asymptotically more efficient transformation to obtain relaxed matrix branching program using the work of Giel [Gie01]. The transformation consists of the following steps – first the formula is balanced and then the resulting balanced formula is converted to a linear bijection straightline-program (LBSP) which is then converted to an RMBP. More formally, we have the following result due to Giel [Gie01].

**Theorem 6.** *[Gie01] Given a boolean formula of size $s$ over any complete basis, there exists a relaxed matrix branching program of size $O(s^{1+\epsilon})$ with the width of each matrix is a constant depending only on $\epsilon$,where $\epsilon > 0$ is any constant.*

## 3.3 Randomization of Random Matrix Branching Programs

In this section, we describe how to randomize the matrices in the (dual-input and oblivious) relaxed matrix branching program obtained from the construction in Section 3.2. The result of the randomization process is another relaxed matrix branching program such that the restriction of the relaxed matrix branching program[3] on input $x$ can be simulated by just knowing the output of the branching program on input $x$. Looking ahead, this property will come in handy when proving the security of the obfuscation scheme in the ideal graded encoding model. The randomization technique we employ closely follows a similar randomization technique that was used in [CFI03] in the context of secure multiparty computation.

The non-triviality of the randomization process here compared to [BGK14] is the following: in [BGK14] the product matrix corresponding to an input $x$ depends only on the output of the function on input $x$. More specifically it is either an identity matrix or a fixed matrix $P_{\mathsf{reject}}$ (Definition 10). Thus, the product matrix does not reveal any information about the branching program. However, in our case the entries in the product matrix might contain useful information about the branching program – specifically the product matrix in our RMBP captures the number of paths between every pair of vertices. Hence, we have to randomize the matrices in such a way that the product of the matrices only reveals information about the output of the function. We do this in two steps. In the first step we design a randomization procedure, denoted by $\mathsf{randBP}$, which reveals just the $(1, w)^{th}$ entry of the product matrix. Note that this itself will not be enough for us because the $(1, w)^{th}$ entry essentially contains the number of paths between the source and the accept vertex and hence has more information than just the output of the function. And so in the second step, we describe how to randomize the RMBP using the procedure $\mathsf{randBP}'$, such that the resulting (equivalent) RMBP when restricted to any particular input $x$ can be simulated by just knowing the output of the RMBP on $x$.

We first describe the $\mathsf{randBP}$ procedure. Though the procedure $\mathsf{randBP}$ to randomize our matrices is similar in spirit to Kilian's randomization (also used in [GGH13b, BGK14]), the way we will simulate these matrices will deviate from that of Kilian.

[3]Recall that the restriction of a relaxed matrix branching program $\mathsf{BP} = (\mathsf{inp}_1, \mathsf{inp}_2, \{B_{i,b_1,b_2}\}_{i\in[n],b_1,b_2\in\{0,1\}})$ is defined to be $\{B_{i,x_{\mathsf{inp}_1(i)},x_{\mathsf{inp}_2(i)}}\}$.

**Notation.** We will denote the relaxed matrix branching program as $\mathsf{BP} = (\mathsf{inp}_1, \mathsf{inp}_2, \{B_{i,b_1,b_2}\}_{i\in[n],b_1,b_2\in\{0,1\}})$ with length $n$, width $w$ and input of $\ell$ bits. For any $x \in \{0,1\}^\ell$, define $P_x := \prod_{i=1}^{n} B_{i,x_{\mathsf{inp}_1(i)},x_{\mathsf{inp}_2(i)}}$ and,

$$\mathsf{BP}\Big|_x := (B_{i,x_{\mathsf{inp}_1(i)},x_{\mathsf{inp}_2(i)}})_{i\in[n]}$$

Let $e_1, e_w \in \{0,1\}^w$, be such that $e_1 = (1,0,0,\ldots,0)$ and $e_w = (0,0,\ldots,0,1)$. For notational convenience, let $e_1$ be a row vector and $e_w$ we a column vector.

**Procedure** $\mathsf{randBP}$**.** The input to the randomization procedure is an oblivious dual-input RMBP $\mathsf{BP} = (\mathsf{inp}_1, \mathsf{inp}_2, \{B_{i,b_1,b_2}\}_{i\in[n],b_1,b_2\in\{0,1\}})$ of length $n$, width $w$ and input of $\ell$ bits.

Procedure $\mathsf{randBP}(\mathsf{BP})$:

- Pick $n+1$ random full-rank matrices $R_0, \ldots, R_n \in \mathbb{Z}_p^{w\times w}$.
- Compute the matrices $\tilde{B}_{i,b_1,b_2} = R_{i-1} \cdot B_{i,b_1,b_2} \cdot R_i^{-1}$ for all $i \in [n]$ and $b_1, b_2 \in \{0,1\}$.
- Finally, compute $\tilde{s} = e_1 \cdot R_0^{-1}$ and $\tilde{t} = R_n \cdot e_w$.
- Output $\widetilde{\mathsf{BP}} = \left(\mathsf{inp}_1, \mathsf{inp}_2, \tilde{s}, \left\{\tilde{B}_{i,b_1,b_2}\right\}_{i\in[n],b_1,b_2\in\{0,1\}}, \tilde{t}\right)$.

It follows that the branching program output by the above procedure on input $\mathsf{BP}$ is functionally equivalent to $\mathsf{BP}$.

We can construct a simulator $\mathsf{Sim}_{\mathsf{BP}}$ such that the following theorem holds. At a high level, the theorem states that the matrices in $\widetilde{\mathsf{BP}}$ (which is the output of $\mathsf{randBP}$ on $\mathsf{BP}$) when restricted to a particular input $x$ can be simulated by just knowing the $(1,w)^{th}$ entry in the product matrix obtained by evaluating $\mathsf{BP}$ on input $x$.

**Theorem 7.** *Consider an oblivious dual-input RMBP* $\mathsf{BP} = (\mathsf{inp}_1, \mathsf{inp}_2, \{B_{i,b_1,b_2}\}_{i\in[n],b_1,b_2\in\{0,1\}})$ *of length* $n$*, of width* $w$ *and input of* $\ell$ *bits. Then for every* $x \in \{0,1\}^\ell$*,*

$$\left\{ \mathsf{randBP}(\mathsf{BP})\Big|_x \right\} \equiv \left\{ \mathsf{Sim}_{\mathsf{BP}}(1^n, 1^w, 1^\ell, P_x[1,w]) \right\}.$$

*Proof.* We first describe the simulator $\mathsf{Sim}_{\mathsf{BP}}$ which simulates the output of $\mathsf{randBP}$ for any input $x$. More formally, let $\mathsf{randBP}(\mathsf{BP})\Big|_x$ be defined as $(\tilde{s}, \{\tilde{B}_{i,x_{\mathsf{inp}_1(i)},x_{\mathsf{inp}_2(i)}}\}_{i\in[n]}, \tilde{t})$. We describe a

simulator $\mathsf{Sim}_{\mathsf{BP}}$ which takes as input
$(1^n, 1^w, 1^\ell, P_x[1,w])$ and outputs a tuple which is identically distributed to $\mathsf{randBP}(\mathsf{BP})\Big|_x$. Recall that $s$ is the size of the formula.

Before we describe $\mathsf{Sim}_{\mathsf{BP}}$ we will first recall the following theorem.

**Theorem 8.** *([Kil88]) Consider a dual-input branching program* $\mathsf{BP} = \{\mathsf{inp}_1, \mathsf{inp}_2, \{B_{i,b_1,b_2}\}_{i\in[n],b_1,b_2\in\{0,1\}}\}$. *There exists a PPT simulator* $\mathsf{Sim}_{\mathsf{K}}$ *such that for every* $x \in \{0,1\}^l$,

$$\{R_0, \{R_{i-1}B_{i,x_{\mathsf{inp}_1(i)},x_{\mathsf{inp}_2(i)}}R_i^{-1}\}_{i\in[n]}, R_n\}$$
$$\equiv \mathsf{Sim}_{\mathsf{K}}(1^n, 1^w, 1^\ell, \mathsf{BP}(x))$$

We are now ready to describe $\mathsf{Sim}_{\mathsf{BP}}$.

$\mathsf{Sim}_{\mathsf{BP}}(1^n, 1^w, 1^\ell, P_x[1,w])$:

- If $P_x[1,w] \neq 0$, define the matrix $A$ as $A := P_x[1,w] \cdot I_{w\times w}$. Else, $A :=$ "mirror-image" of $I_{w\times w}$.
- Run $\mathsf{Sim}_{\mathsf{K}}(1^n, A)$ to obtain full-rank matrices $R_0, R_1, \ldots, R_{n+1} \in \mathbb{Z}_p^{w\times w}$ such that $\prod_{i\geqslant 0} R_i = A$. Note that $\mathsf{Sim}_{\mathsf{K}}$ is the simulator as defined in Theorem 8.
- Let $\hat{R}_0 = e_1 \cdot R_0^{-1}$ and $\hat{R}_{n+1} = R_{n+1} \cdot e_w$.
- Output $(\hat{R}_0, R_1, \ldots, R_n, \hat{R}_{n+1})$.

We now show that:

$$\left\{ \mathsf{randBP}(\mathsf{BP})\Big|_x \right\} \equiv \left\{ \mathsf{Sim}_{\mathsf{BP}}(1^s, P_x[1,w]) \right\}.$$

As a first step, we state the following lemma from Cramer et al. [CFI03] that will be useful to prove the theorem.

**Lemma 7.** *For any* $x, y \in \mathbb{Z}_p^w \backslash \{0\}$ *and a full rank matrix* $M \in \mathbb{Z}_p^{w\times w}$ *there exist full rank matrices* $X, Y \in (\mathbb{Z}_p)^{n\times n}$ *such that the first row of* $X$ *is* $x^T$, *the first column of* $Y$ *is* $y$, *and* $XMY$ *depends only on* $x^T My$. *In particular, there is a procedure* Extend, *running in time polynomial in* $n$ *and*

*$w$, that takes as input $(x^T My, x, y, M)$, where $x, y$ and $M$ are as defined in the above lemma and outputs $X$ and $Y$ such that $XMY$ is $(x^T My) \cdot I_{w \times w}$ if $x^T My \neq 0$ else it is "mirror-image" of $I$.* [4]

We now proceed to proving that the output distributions of $\mathsf{randBP}$ and $\mathsf{Sim}_{\mathsf{BP}}$ are identical. We first define a sequence of hybrids such that the first hybrid is the real experiment (which is $\mathsf{randBP}$) while the last hybrid is the simulated experiment (which is $\mathsf{Sim}_{\mathsf{BP}}$). Then, we show that the output distribution of each hybrid is identical to the output distribution of the previous hybrid which will prove the theorem.

$\underline{\mathsf{Hybrid}_0}$: This is the same as the real experiment. That is, on input $\mathsf{BP}$ and $x$ it first executes $\mathsf{randBP}(\mathsf{BP})$ to obtain $\widetilde{\mathsf{BP}}$. It then outputs $\widetilde{\mathsf{BP}}\Big|_x = (\tilde{s}, \{\tilde{B}_{i,x_{\mathsf{inp}_1(i)},x_{\mathsf{inp}_2(i)}}\}_{i\in[n]}, \tilde{t})$
$\underline{\mathsf{Hybrid}_1}$: We describe $\mathsf{Hybrid}_1$ as follows. The input to $\mathsf{Hybrid}_1$ is $\widehat{\mathsf{BP}} = \mathsf{BP}\big|_x = (B_{i,x_{\mathsf{inp}_1(i)},x_{\mathsf{inp}_2(i)}})_{i\in[n]}$. Let $M_i = B_{i,x_{\mathsf{inp}_1(i)},x_{\mathsf{inp}_2(i)}}$.

$\mathsf{Hybrid}_1\Big(\widehat{\mathsf{BP}} = (M_1, \ldots, M_n)\Big)$ :

- Pick $n+1$ random full-rank matrices $R_0, \ldots, R_n \in \mathbb{Z}_p^{w\times w}$.
- Compute the matrices $\tilde{M}_i = R_{i-1} \cdot M_i \cdot R_i^{-1}$ for $i \in [n]$.
- Finally, compute $\tilde{s} = e_1 \cdot R_0^{-1}$ and $\tilde{t} = R_n \cdot e_w$.
- Output $\Big(\tilde{s}, \{\tilde{M}_i\}_{i\in[n]}, \tilde{t}\Big)$.

It can be seen that the output distribution of this hybrid is identical to the output distribution of the previous hybrid $\mathsf{Hybrid}_0$.

$\underline{\mathsf{Hybrid}_2}$: $\mathsf{Hybrid}_2$ is same as $\mathsf{Hybrid}_1$ except the way we compute $\tilde{s}$ and $\tilde{t}$. The input to $\mathsf{Hybrid}_2$, like the previous hybrid, is $\widehat{\mathsf{BP}} = \mathsf{BP}\big|_x$.

$\mathsf{Hybrid}_2\Big(\widehat{\mathsf{BP}} = (M_1, \ldots, M_n)\Big)$ :

[4] *The "mirror-image" of a $w \times w$ identity matrix is also a $w \times w$ matrix such that the $(i, w-i+1)^{th}$ entry in the matrix is 1 and the rest of the entries in the matrix are 0.*

- Pick $n+1$ random full-rank matrices $R_0, \ldots, R_n \in \mathbb{Z}_p^{w \times w}$.
- Compute the matrices $\tilde{M}_i = R_{i-1} \cdot M_i \cdot R_i^{-1}$ for $i \in [n]$.
- Define $P := \prod_{i=1}^{n} M_i$ and $c := e_1 \cdot P \cdot e_w$.
- Execute $\mathsf{Extend}$ on input $(c, e_1, e_w, P)$ to obtain $w \times w$ matrices $S$ and $T$ as described in Lemma 7. Compute $\hat{S} = SR_0^{-1}$ and $\hat{T} = R_n T$. Finally, compute $\tilde{s} = e_1 \hat{S}$ and $\tilde{t} = \hat{T} e_w$.
- Output $(\tilde{s}, \{\tilde{M}_i\}_{i \in [n]}, \tilde{t})$.

$\mathsf{Hybrid}_1$ and $\mathsf{Hybrid}_2$ differ only in the way $\tilde{s}$ and $\tilde{t}$ are computed. In $\mathsf{Hybrid}_2$, $\tilde{s} = e_1 \hat{S} = e_1 \cdot (SR_0^{-1}) = (e_1 \cdot S) \cdot R_0^{-1} = x^T \cdot R_0^{-1}$, where $x$ is the first row of $S$. But the first row of $S$ is $e_1$ and hence, $\tilde{s} = e_1 \cdot R_0^{-1}$, which is same as the value in $\mathsf{Hybrid}_1$. Similarly, we can show this for $\tilde{t}$.

$\underline{\mathsf{Hybrid}_3}$: This is same as the simulated experiment. That is, it takes as input $(1^n, 1^w, 1^\ell)$ and $P_x[1, w]$ and then executes $\mathsf{Sim}_{\mathsf{BP}}(1^n, 1^w, 1^\ell, P_x[1, w])$. The output of $\mathsf{Hybrid}_3$ is the output of $\mathsf{Sim}_{\mathsf{BP}}$.

We now argue that $\mathsf{Hybrid}_2$ and $\mathsf{Hybrid}_3$ are identically distributed. First note that in $\mathsf{Hybrid}_2$, $c = P[1, w]$. Then it follows from Lemma 7 that if $c \neq 0$, $S \cdot P \cdot T = c \cdot I$, else $S \cdot P \cdot T = J$, where $J$ is the "mirror-image" of $I$. Theorem 8 can be used to show that hybrids $\mathsf{Hybrid}_2$ and $\mathsf{Hybrid}_3$ are identically distributed. This shows that the output distribution of $\mathsf{Hybrid}_0$ is identically distributed to $\mathsf{Hybrid}_3$. This completes the proof. □

We now move to the second step where we show how to randomize the branching program using the procedure $\mathsf{randBP}'$ in such a way that the product of the matrices (which will be a $1 \times 1$ matrix) corresponding to an input only reveals the output of the function and nothing else. To achieve this, we need to ensure that the product of the matrices corresponding to one input is not correlated to the product of matrices corresponding to a different input, where both the inputs are such that they evaluate to 1. We solve this by multiplying the matrix $B_{i,b_1,b_2}$ by $\alpha_{i,b_1,b_2}$ (which is picked at random). This ensures that multiplying the matrices corresponding to two different inputs result in two different products of $\alpha$'s which are mutually independent which in turn makes it feasible to

achieve simulation of these matrices by just knowing the value of the function. We now describe the procedure $\mathsf{randBP}'$. Note that $\mathsf{randBP}'$ takes as input $\widetilde{\mathsf{BP}}$ which is the output of $\mathsf{randBP}$ on the relaxed matrix branching program $\mathsf{BP}$.

**Proceduce** $\mathsf{randBP}'$**.** In this procedure, we describe how to further randomize the output of $\mathsf{randBP}$ and then show how to simulate this by having just the output of $\mathsf{BP}$. The input to $\mathsf{randBP}'$ is a randomized relaxed matrix branching program $\widetilde{\mathsf{BP}} = (\tilde{s}, \{\tilde{B}_{i,b_1,b_2}\}_{i\in[n],b_1,b_2\in\{0,1\}}, \tilde{t})$.

**Procedure** $\mathsf{randBP}'(\widetilde{\mathsf{BP}})$:

- It picks random and independent non-zero scalars $\{\alpha_{i,b_1,b_2} \in \mathbb{Z}_p\}_{i\in[n],b_1,b_2\in\{0,1\}}$ and computes $C_{i,b_1,b_2} = \alpha_{i,b_1,b_2} \cdot \tilde{B}_{i,b_1,b_2}$. It outputs $(\tilde{s}, \{C_{i,b_1,b_2}\}_{i\in[n],b_1,b_2\in\{0,1\}}, \tilde{t})$.

Before we describe how to simulate the output of $\mathsf{randBP}'$, we will prove a claim about this procedure. Let $M_1, M_2, \ldots, M_n$ be a given set of matrices. Let $(N_1, \ldots, N_n)$ be the output of $\mathsf{randBP}'(M_1, M_2, \ldots, M_n)$. We have that $N_1 = \alpha_1 M_1, N_2 = \alpha_2 M_2, \ldots, N_n = \alpha_n M_n$, where $\alpha_1, \alpha_2, \ldots, \alpha_n$ are non-zero scalars chosen uniformly at random from $\mathbb{Z}_p$. Define $c = (\prod_i N_i)[1, w]$.

**Claim 4.** *If* $(\prod_i M_i)[1, w] \neq 0$*, then* $c$ *is distributed uniformly in* $\mathbb{Z}_p^*$*.*

*Proof.* Since $c = (\prod_i N_i)[1, w] = (\prod_i \alpha_i M_i)[1, w] = (\prod_i \alpha_i)(\prod_i M_i)[1, w]$. Since each $\alpha_i$ is chosen uniformly at random from $\mathbb{Z}_p^*$, $\prod_i \alpha_i$ is distributed uniformly in $\mathbb{Z}_p^*$. Hence, when $(\prod_i M_i)[1, w] \neq 0$, $c$ is distributed uniformly in $\mathbb{Z}_p^*$. □

**Simulator** $\mathsf{Sim}'_{\mathsf{BP}}$**.** Next, we describe the simulator $\mathsf{Sim}'_{\mathsf{BP}}$ which takes as input $(1^s, \mathsf{BP}(x))$, where $s$ is the size of the formula and $x \in \{0,1\}^\ell$.

$\mathsf{Sim}'_{\mathsf{BP}}(1^s, \mathsf{BP}(x))$:

- If $\mathsf{BP}(x) = 0$, output whatever $\mathsf{Sim}_{\mathsf{BP}}(1^s, 0)$ outputs. Else, pick a $\alpha$ uniformly at random from $\mathbb{Z}_p^*$ and output whatever $\mathsf{Sim}_{\mathsf{BP}}(1^s, \alpha)$ outputs.

Now, we prove the following.

**Theorem 9.** *Consider an oblivious dual-input RMBP* $\mathsf{BP} = (\mathsf{inp}_1, \mathsf{inp}_2, \{B_{i,b_1,b_2}\}_{i\in[n],b_1,b_2\in\{0,1\}})$ *of length* $n$*, width* $w$ *and input of* $\ell$ *bits. Then there exists a PPT simulator* $\mathsf{Sim}'_{\mathsf{BP}}$ *such that for every* $x \in \{0,1\}^\ell$*,*

$$\left\{ \mathsf{randBP}'(\mathsf{randBP}(\mathsf{BP}))\Big|_x \right\} \equiv \left\{ \mathsf{Sim}'_{\mathsf{BP}}(1^s, \mathsf{BP}(x)) \right\}.$$

*Proof.* Let us denote $\mathsf{BP}\Big|_x$ by $(M_1, M_2, \ldots, M_n)$. Observe that

$$\left\{ \mathsf{randBP}'(\mathsf{randBP}(\mathsf{BP}))\Big|_x \right\} \equiv \left\{ \mathsf{randBP}(\mathsf{randBP}'(M_1, M_2, \ldots, M_n)) \right\}.$$

This holds by just observing that applying $\mathsf{randBP}'(\mathsf{randBP}(\cdot))$ operation on the relaxed matrix branching program and then evaluating the result on an input $x$ is equivalent to first evaluating the relaxed matrix branching program on an input $x$ and then applying the $\mathsf{randBP}'(\mathsf{randBP}(\cdot))$ operation. Now, we need to show that

$$\left\{ \mathsf{randBP}(\mathsf{randBP}'(M_1, M_2, \ldots, M_n)) \right\} \equiv \left\{ \mathsf{Sim}'_{\mathsf{BP}}(1^s, \mathsf{BP}(x)) \right\}.$$

We will show that for any tuple $V$, the probability of output being $V$ is identical in the real and simulated experiments above. We begin by calculating the probability of $V$ in the real experiment, where probability is taken over the random coins of both $\mathsf{randBP}$ and $\mathsf{randBP}'$. Let $V_2 = M_1, M_2, \ldots, M_n$.

$$\begin{aligned}\Pr[\mathsf{randBP}(\mathsf{randBP}'(V_2)) = V] &= \sum_{V_1} \Pr[\mathsf{randBP}(V_1) = V \wedge \mathsf{randBP}'(V_2) = V_1] \\ &= \sum_{V_1} \Pr[\mathsf{randBP}(V_1) = V] \cdot \Pr[\mathsf{randBP}'(V_2) = V_1]\end{aligned}$$

Now let $V_1 = (N_1, N_2, \ldots, N_n)$ and $\beta_{V_1}$ denote $(\prod_i N_i)[1, w]$. Then by Theorem 7, $\Pr[\mathsf{randBP}(V_1) = V] = \Pr[\mathsf{Sim}_{\mathsf{BP}}(1^s, \beta_{V_1}) = V]$. Substituting in above, we get

$$\begin{aligned}\Pr[\mathsf{randBP}(\mathsf{randBP}'(V_2)) = V] &= \sum_{V_1} \Pr[\mathsf{Sim}_{\mathsf{BP}}(1^s, \beta_{V_1}) = V] \cdot \Pr[\mathsf{randBP}'(V_2) = V_1] \\ &= \sum_{\alpha} \sum_{V_1 s.t. \beta_{V_1} = \alpha} \Pr[\mathsf{Sim}_{\mathsf{BP}}(1^s, \alpha) = V] \cdot \Pr[\mathsf{randBP}'(V_2) = V_1] \\ &= \sum_{\alpha} \Pr[\mathsf{Sim}_{\mathsf{BP}}(1^s, \alpha) = V] \cdot \sum_{V_1 s.t. \beta_{V_1} = \alpha} \Pr[\mathsf{randBP}'(V_2) = V_1]\end{aligned}$$

We have two cases based on whether $\mathsf{BP}(x) = 1$ or $\mathsf{BP}(x) = 0$.

- $\mathsf{BP}(x) = 0$**:** This case is easy to handle. Note that in this case, $\prod_i M_i[1, w] = 0 = \beta_{V_1}$. Hence, in the above expression, $\sum_{V_1 s.t. \beta_{V_1} = \alpha} \Pr[\mathsf{randBP}'(V_2) = V_1] = 1$ for $\beta_{V_1} = 0$ and 0 otherwise. Substituting in the above expression we get,

$$\begin{aligned}\Pr[\mathsf{randBP}(\mathsf{randBP}'(V_2)) = V] &= \Pr[\mathsf{Sim}_{\mathsf{BP}}(1^s, 0) = V] \\ &= \Pr[\mathsf{Sim}'_{\mathsf{BP}}(1^s, \mathsf{BP}(x)) = V]\end{aligned}$$

- $\mathsf{BP}(x) = 1$**:** In this case, $\prod_i M_i[1, w] \neq 0$. By Claim 4, $\sum_{V_1 s.t. \beta_{V_1} = \alpha} \Pr[\mathsf{randBP}'(V_2) = V_1] = \frac{1}{p-1}$. Substituting in above equation we get,

$$\begin{aligned}\Pr[\mathsf{randBP}(\mathsf{randBP}'(V_2)) = V] &= \frac{1}{p-1} \cdot \sum_{\alpha} \Pr[\mathsf{Sim}_{\mathsf{BP}}(1^s, \alpha) = V] \\ &= \Pr[\mathsf{Sim}'_{\mathsf{BP}}(1^s, \mathsf{BP}(x)) = V]\end{aligned}$$

□

## 3.4 Ideal Graded Encoding Model

In this section, we describe the ideal graded encoding model. This section has been taken almost verbatim from [BGK14]. All parties have access to an oracle $\mathcal{M}$, implementing an ideal graded encoding. The oracle $\mathcal{M}$ implements an idealized and simplified version of the graded encoding schemes from [GGH13a]. The parties are provided with encodings of various elements at different

levels. They are allowed to perform arithmetic operations of addition/multiplication and testing equality to zero as long as they respect the constraints of the multilinear setting. We start by defining an algebra over the elements.

**Definition 13.** *Given a ring $R$ and a universe set $\mathbb{U}$, an element is a pair $(\alpha, S)$ where $\alpha \in R$ is the* value *of the element and $S \subseteq \mathbb{U}$ is the* index *of the element. Given an element $e$ we denote by $\alpha(e)$ the value of the element, and we denote by $S(e)$ the index of the element. We also define the following binary operations over elements:*

- *For two elements $e_1, e_2$ such that $S(e_1) = S(e_2)$, we define $e_1 + e_2$ to be the element $(\alpha(e_1) + \alpha(e_2), S(e_1))$, and $e_1 - e_2$ to be the element $(\alpha(e_1) - \alpha(e_2), S(e_1))$.*
- *For two elements $e_1, e_2$ such that $S(e_1) \cap S(e_2) = \emptyset$, we define $e_1 \cdot e_2$ to be the element $(\alpha(e_1) \cdot \alpha(e_2), S(e_1) \cup S(e_2))$.*

Next, we describe the oracle $\mathcal{M}$. $\mathcal{M}$ is a stateful oracle mapping elements to "generic" representations called *handles*. Given handles to elements, $\mathcal{M}$ allows the user to perform operations on the elements. $\mathcal{M}$ will implement the following interfaces:

**Initialization.** $\mathcal{M}$ will be initialized with a ring $R$, a universe set $\mathbb{U}$, and a list $L$ of initial elements. For every element $e \in L$, $\mathcal{M}$ generates a handle. We do not specify how the handles are generated, but only require that the value of the handles are independent of the elements being encoded, and that the handles are distinct (even if $L$ contains the same element twice). $\mathcal{M}$ maintains a handle table where it saves the mapping from elements to handles. $\mathcal{M}$ outputs the handles generated for all the elements in $L$. After $\mathcal{M}$ has been initialized, all subsequent calls to the initialization interface fail.

**Algebraic operations.** Given two input handles $h_1, h_2$ and an operation $\circ \in \{+, -, \cdot\}$, $\mathcal{M}$ first locates the relevant elements $e_1, e_2$ in the handle table. If any of the input handles do not appear in the handle table (that is, if the handle was not previously generated by $\mathcal{M}$) the call to $\mathcal{M}$ fails. If the expression $e_1 \circ e_2$ is undefined (i.e., $S(e_1) \neq S(e_2)$ for $\circ \in \{+, -\}$, or $S(e_1) \cap S(e_2) \neq \emptyset$ for

$\circ \in \{\cdot\}$) the call fails. Otherwise, $\mathcal{M}$ generates a new handle for $e_1 \circ e_2$, saves this element and the new handle in the handle table, and returns the new handle.

**Zero testing.** Given an input handle $h$, $\mathcal{M}$ first locates relevant element $e$ in the handle table. If $h$ does not appear in the handle table (that is, if $h$ was not previously generated by $\mathcal{M}$) the call to $\mathcal{M}$ fails. If $S(e) \neq \mathbb{U}$, the call fails. Otherwise, $\mathcal{M}$ returns 1 if $\alpha(e) = 0$, and returns 0 if $\alpha(e) \neq 0$.

## 3.5 Straddling Set System

In this section, we describe a straddling set system which is same as the one considered in [BGK14]. Then we will prove two combinatorial properties of this set system, which will be very useful in proving the VBB security of our scheme.

**Definition 14.** *A straddling set system* $\mathbb{S}_n = \{S_{i,b} : i \in [n], b \in \{0,1\}\}$ *with* $n$ *entries over the universe* $\mathbb{U} = \{1, 2, \ldots, 2n-1\}$ *is as follows:*
$S_{1,0} = \{1\}, S_{2,0} = \{2,3\}, \ldots, S_{i,0} = \{2i-2, 2i-1\}, \ldots, S_{n-1,0} = \{2n-4, 2n-3\}$, $S_{n,0} = \{2n-2, 2n-1\}$
$S_{1,1} = \{1,2\}, S_{2,1} = \{3,4\}, \ldots, S_{i,1} = \{2i-1, 2i\}, \ldots, S_{n-1,1} = \{2n-3, 2n-2\}$, $S_{n,1} = \{2n-1\}$

**Claim 5** (Two unique covers of universe)**.** *The only exact covers of* $\mathbb{U}$ *are* $\{S_{i,0}\}_{i\in[n]}$ *and* $\{S_{i,1}\}_{i\in[n]}$.

*Proof.* Since any exact cover of $\mathbb{U}$ needs to pick a set with element 1, it either contains the set $S_{1,0}$ or $S_{1,1}$. Let $\mathcal{C}$ be a cover of $\mathbb{U}$ containing $S_{1,0}$. Then, we prove that $S_{i,0} \in \mathcal{C}, \forall i \in [n]$. We will prove this via induction on $i$. It is trivially true for $i = 1$. Let us assume that the statement is true for $i$, and prove the statement for $i+1$. There are only two sets, namely $S_{i+1,0}$ and $S_{i,1}$ which contain the element $2i \in \mathbb{U}$. Since, by induction hypothesis, $S_{i,0} \in \mathcal{C}$ and $S_{i,0} \cap S_{i,1} \neq \emptyset$, $S_{i+1,0} \in \mathcal{C}$ in order to cover all the elements in $\mathbb{U}$. This shows that there is a unique cover of $\mathbb{U}$ containing $S_{1,0}$.

Similarly, we can show that there is a unique cover of $\mathbb{U}$ containing the set $S_{1,1}$ which is $\{S_{i,1}\}_{i\in[n]}$. As mentioned before, any exact cover of $\mathbb{U}$ contains either $S_{1,0}$ or $S_{1,1}$ in order to cover the element $1 \in \mathbb{U}$. This proves the claim. □

**Claim 6** (Collision at universe)**.** *Let $\mathcal{C}$ and $\mathcal{D}$ be non-empty collections of sets such that $\mathcal{C} \subseteq \{S_{i,0}\}_{i\in[n]}$, $\mathcal{D} \subseteq \{S_{i,1}\}_{i\in[n]}$, and $\bigcup_{S\in\mathcal{C}} S = \bigcup_{S\in\mathcal{D}} S$, then following must hold:*

$$\mathcal{C} = \{S_{i,0}\}_{i\in[n]}\ ,\ \mathcal{D} = \{S_{i,1}\}_{i\in[n]}.$$

*Proof.* We will prove this claim by contradiction. Let us assume that $\mathcal{C} \subset \{S_{i,0}\}_{i\in[n]}$. Then there exists a maximal sub-interval $[i, j] \subset [n]$ such that $S_{k,0} \in \mathcal{C}$ for all $i \leqslant k \leqslant j$ but *either* (1) $i > 1$ and $S_{i-1,0} \notin \mathcal{C}$ *or* (2) $j < n$ and $S_{j+1,0} \notin \mathcal{C}$.

(1) Since $(2i - 2) \in S_{i,0} \in \mathcal{C}$ and $\bigcup_{S\in\mathcal{C}} S = \bigcup_{S\in\mathcal{D}} S$, it should be the case that $S_{i-1,1} \in \mathcal{D}$. Now by a similar argument, since $(2i - 3) \in S_{i-1,1} \in \mathcal{D}$ and $\bigcup_{S\in\mathcal{C}} S = \bigcup_{S\in\mathcal{D}} S$, it should be the case that $S_{i-1,0} \in \mathcal{C}$. This contradicts the assumption that $i > 1$ and $S_{i-1,0} \notin \mathcal{C}$.

(2) Since $(2j - 1) \in S_{j,0} \in \mathcal{C}$ and $\bigcup_{S\in\mathcal{C}} S = \bigcup_{S\in\mathcal{D}} S$, it should be the case that $S_{j,1} \in \mathcal{D}$. Now by a similar argument, since $(2j) \in S_{j,1} \in \mathcal{D}$ and $\bigcup_{S\in\mathcal{C}} S = \bigcup_{S\in\mathcal{D}} S$, it should be the case that $S_{j+1,0} \in \mathcal{C}$. This contradicts the assumption that $j < n$ and $S_{j+1,0} \notin \mathcal{C}$.

Since $\mathcal{C} = \{S_{i,0}\}_{[}n]$, it has to be the case that $\mathcal{D} = \{S_{i,1}\}_{[}n]$. □

## 3.6 Obfuscation in the Idealized Graded Encoding Model

In this section, we describe our VBB obfuscator $\mathcal{O}$ for polynomial sized formulae in the ideal graded encoding model.

**Input.** The input to our obfuscator $\mathcal{O}$ is a dual-input oblivious relaxed matrix branching program $\mathsf{BP}$ of length $n$, width $w$, input length $\ell$:

$$\mathsf{BP} = (\mathsf{inp}_1, \mathsf{inp}_2, \{B_{i,b_1,b_2}\}_{i\in[n],b_1,b_2\in\{0,1\}})$$

such that $\mathsf{inp}_1$ and $\mathsf{inp}_2$ are evaluation functions mapping $[n] \to [\ell]$, and each $B_{i,b_1,b_2} \in \{0, 1\}^{w\times w}$ is a full rank matrix.

We make a simplifying assumption that every input bit is inspected by $\mathsf{BP}$ exactly $\ell'$ number of times. We denote the set of indices that inspect the input bit $j$ by $\mathsf{ind}(j)$.

$$\mathsf{ind}(j) = \{i \in [n] : \mathsf{inp}_1(i) = j\} \cup \{i \in [n] : \mathsf{inp}_2(i) = j\}\ .$$

**Step 1: Randomizing the relaxed matrix branching program** $\mathsf{BP}$**.** The obfuscator $\mathcal{O}$ randomizes the branching program in two steps using procedures $\mathsf{randBP}$ and $\mathsf{randBP}'$ described in Section 3.3. It begins by sampling a large enough prime $p$ of $\Omega(n)$ bits.

1. It invokes the procedure $\mathsf{randBP}$ on the relaxed matrix branching program $\mathsf{BP}$ obtained above to get $\widetilde{\mathsf{BP}} = \left(\mathsf{inp}_1, \mathsf{inp}_2, \tilde{s}, \left\{\tilde{B}_{i,b_1,b_2}\right\}_{i\in[n],b_1,b_2\in\{0,1\}}, \tilde{t}\right)$. Recall that $\tilde{s}, \tilde{t} \in \mathbb{Z}_p^w$ and $\tilde{B}_{i,b_1,b_2} \in \mathbb{Z}_p^{w\times w}$ for all $i \in [n], b_1, b_2 \in \{0,1\}$.

2. It then executes the procedure $\mathsf{randBP}'$ on input $\widetilde{\mathsf{BP}}$ to obtain $(\tilde{s}, \{C_{i,b_1,b_2}\}_{i\in[n],b_1,b_2\in\{0,1\}}, \tilde{t})$. The matrices $C_{i,b_1,b_2}$ are such that $C_{i,b_1,b_2} = \alpha_{i,b_1,b_2} \cdot \tilde{B}_{i,b_1,b_2}$, where $\alpha_{i,b_1,b_2} \in \mathbb{Z}_p$ with $i \in [n], b_1, b_2 \in \{0,1\}$ are picked uniformly at random.

The output of this phase is $(\mathsf{inp}_1, \mathsf{inp}_2, \tilde{s}, \{C_{i,b_1,b_2}\}_{\substack{i\in[n],\\ b_1,b_2\in\{0,1\}}}, \tilde{t})$.

Looking ahead, the final obfuscation of $\mathsf{BP}$ will consist of ideal encodings of these elements with respect to a carefully chosen set system. Next, we describe how these sets are chosen.

**Step 2: Initialization of the set systems.** Consider a universe set $\mathbb{U}$. Let $\mathbb{U}_s$, $\mathbb{U}_t$, $\mathbb{U}_1$, $\mathbb{U}_2, \ldots, \mathbb{U}_\ell$ be partitions of $\mathbb{U}$ such that for all $j \in [\ell]$, $|\mathbb{U}_j| = (2\ell' - 1)$. That is, $\mathbb{U}_s$, $\mathbb{U}_t$, $\mathbb{U}_1$, $\mathbb{U}_2, \ldots, \mathbb{U}_\ell$ are disjoint sets and $\mathbb{U} = \mathbb{U}_s \cup \mathbb{U}_t \cup \bigcup_{j=1}^{\ell} \mathbb{U}_j$.

Now let $\mathbb{S}^j$ be the straddling set system (defined in Section 3.5) over the elements in $\mathbb{U}_j$. Note that $\mathbb{S}^j$ will have $|\mathsf{ind}(j)| = \ell'$ sets in the system for each $j \in [\ell]$. We now associate the entries in the straddling set system $\mathbb{S}^j$ with the indices of $\mathsf{BP}$ which depend on $x_j$, i.e. the set $\mathsf{ind}(j)$. More precisely, let

$$\mathbb{S}^j = \{S^j_{k,b} : k \in \mathsf{ind}(j), b \in \{0,1\}\}.$$

**Step 3: Associating elements of randomized RMBP with sets.** Next, we associate a set to each element output by the randomization step. Recall that in a dual-input relaxed matrix branching program, each step depends on two fixed bits in the input defined by the evaluation functions $\mathsf{inp}_1$ and $\mathsf{inp}_2$. For each step $i \in [n]$, $b_1, b_2 \in \{0,1\}$, we define the set $S(i, b_1, b_2)$ using the straddling sets for input bits $\mathsf{inp}_1(i)$ and $\mathsf{inp}_2(i)$ as follows:

$$S(i, b_1, b_2) := S^{\mathsf{inp}_1(i)}_{i,b_1} \cup S^{\mathsf{inp}_2(i)}_{i,b_2}.$$

**Step 4: Encoding of elements in randomized RMBP.** We use the set $S(i, b_1, b_2)$ to encode the elements of $C_{i,b_1,b_2}$. We will use the sets $\mathbb{U}_s$ and $\mathbb{U}_t$ to encode the elements in $\tilde{s}$ and $\tilde{t}$ respectively. More formally, $\mathcal{O}$ does the following:

$\mathcal{O}$ initializes the oracle $\mathcal{M}$ with the ring $\mathbb{Z}_p$ and universe set $\mathbb{U}$. Then it asks for the encodings of the following elements:

$$\left\{(\tilde{s}[k], \mathbb{U}_s),\ (\tilde{t}[k], \mathbb{U}_t)\right\}_{k\in[w]}$$

$$\{(C_{i,b_1,b_2}[j,k], S(i,b_1,b_2)\}_{i\in[n],b_1,b_2\in\{0,1\},j,k\in[w]}$$

$\mathcal{O}$ receives a list of handles for these elements from $\mathcal{M}$. Let $[\beta]_S$ denote the handle to $(\beta, S)$. For a matrix $M$, let $[M]_S$ denote a matrix of handles such that $[M]_S[j,k]$ is a handle for $(M[j,k], S)$. Thus, $\mathcal{O}$ receives the following handles, which is then output by $\mathcal{O}$.

$$[\tilde{s}]_{\mathbb{U}_s},\ \ [\tilde{t}]_{\mathbb{U}_t},\ \ \left\{[C_{i,b_1,b_2}]_{S(i,b_1,b_2)}\right\}_{i\in[n],b_1,b_2\in\{0,1\}}$$

**Evaluation of $\mathcal{O}(\mathsf{BP})$ on input $x$.** Recall that two handles corresponding to the same set $S$ can be added. If $[\beta]_S$ and $[\gamma]_S$ are two handles, we denote the handle for $(\beta+\gamma, S)$ obtained from $\mathcal{M}$ on addition query by $[\beta]_S + [\gamma]_S$. Similarly, two handles corresponding to $S_1$ and $S_2$ can be multiplied if $S_1 \cap S_2 = \emptyset$. We denote the handle for $(\beta \cdot \gamma, S_1 \cup S_2)$ obtained from $\mathcal{M}$ on valid multiplication query on $[\beta]_{S_1}$ and $[\gamma]_{S_2}$ by $[\beta]_{S_1} \cdot [\gamma]_{S_2}$. Similarly, we denote the handle for $(M_1 \cdot M_2, S_1 \cup S_2)$ by $[M_1]_{S_1} \cdot [M_2]_{S_2}$.

Given $x \in \{0,1\}^\ell$, to compute $\mathsf{BP}(x)$, $\mathcal{O}(\mathsf{BP})$ computes the handle for the following expression:

$$h = [\tilde{s}]_{\mathbb{U}_s} \cdot \prod_{i=1}^{n} \left[C_{i,x_{\mathsf{inp}_1(i)},x_{\mathsf{inp}_2(i)}}\right]_{S(i,x_{\mathsf{inp}_1(i)},x_{\mathsf{inp}_2(i)})} \cdot [\tilde{t}]_{\mathbb{U}_t}$$

Next, $\mathcal{O}(\mathsf{BP})$ uses the oracle $\mathcal{M}$ to do a zero-test on $h$. If the zero-test returns a 1, then $\mathcal{O}(\mathsf{BP})$ outputs 0 else it outputs 1.

**Correctness of Evaluation.** We first assume that none of the calls to $\mathcal{M}$ fail and show that $\mathcal{O}(\mathsf{BP})$ on $x$ outputs 1 iff $\mathsf{BP}(x) = 1$. We denote $b_1^i = x_{\mathsf{inp}_1(i)}$ and $b_2^i = x_{\mathsf{inp}_2(i)}$ in the following

equation. From the description of the evaluation above, $\mathcal{O}(\mathsf{BP})$ outputs $0$ on $x$ if and only if

$$\begin{aligned}
0 = \tilde{s} \cdot \prod_{i=1}^{n} C_{i,b_1^i,b_2^i} \cdot \tilde{t} &= \tilde{s} \cdot \prod_{i=1}^{n} \alpha_{i,b_1^i,b_2^i} \cdot \tilde{B}_{i,b_1^i,b_2^i} \cdot \tilde{t} \\
&= \left( (e_1 R_0^{-1}) \cdot \prod_{i=1}^{n} R_{(i-1)} \cdot B_{i,b_1^i,b_2^i} \cdot R_i^{-1} \cdot (R_n e_w) \right) \prod_{i=1}^{n} \alpha_{i,b_1^i,b_2^i} \\
&= \left( e_1 \cdot \prod_{i=1}^{n} B_{i,b_1^i,b_2^i} \cdot e_w \right) \cdot \prod_{i=1}^{n} \alpha_{i,b_1^i,b_2^i} = P_x[1,w] \cdot \prod_{i=1}^{n} \alpha_{i,b_1^i,b_2^i}
\end{aligned}$$

We conclude with the following theorem and corollary which summarize our results.

**Theorem 10.** *There is a virtual black box obfuscator $\mathcal{O}$ in the idealized model for all poly-sized RMBPs. For a family of input-oblivious RMBPs of length $n$ and width $w$, the obfuscation requires $n$ levels of multilinearity over a field of size $p = 2^{\Omega(n)}$, the obfuscated program consists of $nw^2$ encodings of field elements, and its evaluation involves $O(nw^2)$ multilinear operations.*

The proof of the above theorem follows along the lines of Barak et al. [BGK14]. We provide the formal proof in the next section.

The following corollary follows from Theorem 4, Theorem 5 and the above theorem.

**Corollary 11.** *There is a virtual black box obfuscator $\mathcal{O}$ in the idealized model for non-deterministic branching programs. For a family of keyed branching programs (or formulas) of size $s$, the obfuscation requires $s$ levels of multilinearity over a field of size $p = 2^{\Omega(s)}$, the obfuscated program consists of $O(s^3)$ encodings of field elements, and its evaluation involves $O(s^3)$ multilinear operations. For a family of input-oblivious, special layered branching programs of length $n$ and width $w$, the obfuscation requires $n$ levels of multilinearity over a field of size $p = 2^{\Omega(n)}$, the obfuscated program consists of $O(nw^2)$ encodings of field elements, and its evaluation involves $O(nw^2)$ multilinear operations.*

In the above theorem and its corollary, the obliviousness requirement can be relaxed by incurring an additional multiplicative overhead of $\ell$ to the levels of multilinearity and the number of multilinear operations, where $\ell$ is the number of input variables.

## 3.7 Proof of Virtual Black Box Obfuscation in the Idealised Graded Encoding Model

In this section, we prove that the obfuscator $\mathcal{O}$ described in Section 3.6 is a good VBB obfuscator for polynomial sized formulas in the ideal graded encoding model.

Let $\mathcal{F} = \{\mathcal{F}_\ell\}_{\ell \in \mathbb{N}}$ be a formula class such that every formula in $\mathcal{F}_\ell$ is of size $O(\ell)$. We assume WLOG that all formulas in $\mathcal{F}_\ell$ are of the same size (otherwise the formula can be padded). It follows from Theorem 5 that for any formula $\mathsf{F}$ there exists a RMBP represented in the form of $O(|\mathsf{F}|)$ matrices each of width $O(|\mathsf{F}|)$. Hence, there exists linear functions $n(\cdot)$ and $w(\cdot)$ such that $\mathcal{O}$ in Section 3.6 outputs a dual-input oblivious RMBP of size $n(|\mathsf{F}|)$ and width $w(|\mathsf{F}|)$ computing on $\ell(|\mathsf{F}|)$ inputs. Hence, $\mathcal{O}$ satisfies the polynomial slowdown requirement. We also showed that $\mathcal{O}$ satisfies the functionality requirement and always computes the correct output (see Section 3.6). We are now left to show that $\mathcal{O}$ satisfies the virtual black box property.

**The Simulator** $\mathsf{Sim}$ Here we construct a simulator $\mathsf{Sim}$ that takes as input $1^{|\mathsf{F}|}$ and description of the adversary $\mathcal{A}$, and is given oracle access to the formula $\mathsf{F}$. This simulator is required to simulate the view of the adversary.

The simulator begins by emulating the obfuscator $\mathcal{O}$ on $\mathsf{F}$. First, the simulator needs to compute the RMBP $\mathsf{BP}_\mathsf{F}$ and the matrices $B_{i,b_1,b_2}$ corresponding to the branching program. Note that the simulator is only given oracle access to the formula $\mathsf{F}$ and has no way to compute these matrices. Thus, $\mathsf{Sim}$ initializes the oracle $\mathcal{M}$ with formal variables. Also note that the simulator can compute the evaluation functions $\mathsf{inp}_1$ and $\mathsf{inp}_2$ and also the system used for encodings since the RMBPs are oblivious. This would be important when $\mathsf{Sim}$ simulates the oracle queries of $\mathcal{A}$.

More formally, we extend the definition of an element to allow for values that are formal variables and also expressions over formal variables, instead of just being ring elements. When we perform an operation $\circ$ on two elements $e_1$ and $e_2$, that contain formal variables, the resultant element $e_1 \circ e_2$ is a corresponding arithmetic expression over formal variables. This way we represent formal expressions as arithmetic circuits. We denote by $\alpha(e)$ the arithmetic expression over formal variables for element $e$. An element is called *basic* element if the corresponding

arithmetic circuit has no gates, i.e. either it is a constant or a formal variable. We say that $e'$ is a *sub-element* of $e$ if the circuit corresponding to $e'$ is a sub-circuit of the circuit for $e$.

Next, $\mathsf{Sim}$ will emulate the oracle $\mathcal{M}$ that $\mathcal{O}$ accesses as follows: $\mathsf{Sim}$ will maintain a table of handles and corresponding level of encodings that have been initialized so far. As mentioned before, $\mathsf{Sim}$ will initialize the oracle $\mathcal{M}$ with formal variables. Note that $\mathsf{Sim}$ can emulate all the interfaces of $\mathcal{M}$ apart from the zero-testing. Note that $\mathcal{O}$ does not make any zero-test queries. Hence, the simulation of the obfuscator $\mathcal{O}$ is perfect.

When $\mathsf{Sim}$ completes the emulation of $\mathcal{O}$ it obtains a simulated obfuscation $\tilde{\mathcal{O}}(\mathsf{F})$. Now $\mathsf{Sim}$ has to simulate the view of the adversary on input $\tilde{\mathcal{O}}(\mathsf{F})$. Our $\mathsf{Sim}$ will use the same handles table for emulating the oracle calls of both $\mathcal{O}$ and $\mathcal{A}$. Hence, $\mathsf{Sim}$ can perfectly emulate all the oracle calls made by $\mathcal{A}$ apart from zero-testing. The problem with answering zero-test queries is that $\mathsf{Sim}$ cannot zero-test the expressions involving formal variables. Zero-testing is the main challenge for simulation, which we describe in the next section. Since the distribution of handles generated during the simulation and during the real execution are identical, and since the obfuscation consists only of handles (as opposed to elements), we have that the simulation of the obfuscation $\tilde{\mathcal{O}}$ and the simulation of $\mathcal{M}$'s answers to all the queries, except for zero-test queries, is perfect.

**Simulating zero testing queries** In this part we describe how our simulator handles the zero-test queries made by $\mathcal{A}$. This part is the non-trivial part of the analysis for the following reason. The handle being zero-tested is an arithmetic circuit whose value depends on the formal variables which are unknown to the simulator. The real value for these formal variables would depend on the formula $\mathsf{F}$. At a very high level, we show that these values can be simulated given oracle access to $\mathsf{F}$.

There are two steps to zero-testing an element. Note that the adversary may have combined the handles provided in very convoluted manner. More precisely, $\mathcal{A}$ may have computed sub-expressions involving multiple inputs and hence, the value of the element being zero-tested may depend on formal variables which correspond to using multiple inputs. Hence, the first step is to decompose this elements into "simpler" elements that we call *single-input elements*. As the name suggests, any single input element's circuit consists of formal variables corresponding to a

distinct input $x \in \{0,1\}^\ell$. Namely, it only depends on formal variables in matrices $C_{i,b_1,b_2}$ such that $b_1 = x_{\mathsf{inp}_1(i)}$ and $b_2 = x_{\mathsf{inp}_2(i)}$. In the first step we show that any element $e$, such that $S(e) = \mathbb{U}$ which is zero-tested can be decomposed into polynomial number of single input elements.

In the second step, $\mathsf{Sim}$ simulates the value of each of the single input elements obtained via decomposition independently. More formally, we use Theorem 9 to show that value of each single-input element can be simulated perfectly. But we run into the following problem. We cannot simulate the value of all the single input elements together as these have correlated randomness of the obfuscator. Instead we show that it suffices to zero-test each single-input element individually. For this we use the fact that each of the matrix $\tilde{B}_{i,b_1,b_2}$ was multiplied by $\alpha_{i,b_1,b_2}$. Using this we prove that value of each single input element depends on product of different $\alpha$'s which is determined by the input being used. Now, we use the fact that the probability that $\mathcal{A}$ creates an element such that non-zero value of two single input elements cancel each other is negligible. Therefore, it holds that element is zero iff each of the single input elements are zero independently.

### 3.7.1 Decomposition to Single-Input Elements

Next we show how every element can be decomposed into polynomial number of single-input elements. We start by introducing some notation.

For every element $e$, we will assign an *input-profile* $\mathsf{Prof}(e) \in \{0,1,*\}^\ell \cup \{\perp\}$. Intuitevely, if $e$ is a sub-expression in the evaluation of the obfuscated program on some input $x \in \{0,1\}^\ell$, then $\mathsf{Prof}(e)$ is used to represent the partial information about $x$ which can be learnt from formal variables which occur in $e$. For example, we say that $\mathsf{Prof}(e)_j$ is *consistent* with the bit $b$ if there exists a basic sub-element $e'$ of $e$ such that $S(e') = S(i, b_1, b_2)$ such that $\mathsf{inp}_1(i) = j$ and $b_1 = b$ or $\mathsf{inp}_2(i) = j$ and $b_2 = b$. Next, for every $j \in [\ell]$ we set $\mathsf{Prof}(e)_j = b$ iff $\mathsf{Prof}(e)_j$ is consistent with $b$ and is not consistent with $(1-b)$. If $\mathsf{Prof}(e)_j$ is neither consistent with $b$ nor $(1-b)$, we set $\mathsf{Prof}(e)_j = *$. Finally, we set $\mathsf{Prof}(e) = \perp$ iff there exists a $j \in [\ell]$ such that $\mathsf{Prof}(e)$ is consistent with both $b$ and $(1-b)$. We call $e$ a *single-input* element iff $\mathsf{Prof}(e) \neq \perp$. Finally, if $\mathsf{Prof}(e) \in \{0,1\}^\ell$, we say that input-profile of $e$ is *complete*. Otherwise, we say that input-profile of $e$ is *partial*.

We also define the partial symmetric operation $\odot : \{0,1,*,\perp\} \times \{0,1,*,\perp\} \to \{0,1,\perp\}$ as

follows: $b \odot * = b$ for $b \in \{0, 1, *, \perp\}$, $b \odot b = b$, and $b \odot (1 - b) = \perp$ for $b \in \{0, 1\}$, and $\perp \odot \perp = \perp$. If $\odot$ is applied to two vectors, it is performed separately for each position.

Next we describe an algorithm $D$ used by $\mathsf{Sim}$ to decompose elements into single-input elements. Parts of this description have been taken verbatim from [BGK14]. Given an element $e$, $D$ outputs a set of single-input elements with distinct input-profiles such that $e = \sum_{s \in D(e)} s$, where the equality between the elements means that their values compute the same function (it does not mean that the arithmetic circuits that represent these values are identical). Note that the above requirement implies that for every $s \in D(e)$, $S(s) = S(e)$. Moreover, for each $s \in D(e)$, $D$ also computes the input-profile of $s$ recursively.

The decomposition algorithm $D$ outputs a set of elements and their associated input profile and is defined recursively, as follows:

- Element $e$ is basic: $D$ outputs the singleton set $\{e\}$. Let $S(e) = S(i, b_1, b_2)$. Then $\mathsf{Prof}(e)$ is as follows: $\mathsf{Prof}(e)_{\mathsf{inp}_1(i)} = b_1$, $\mathsf{Prof}(e)_{\mathsf{inp}_2(i)} = b_2$, and $\mathsf{Prof}(e)_j = *$ for all $j \in [\ell], j \neq \mathsf{inp}_1(i), j \neq \mathsf{inp}_2(i)$.

- Element $e$ is of the form $e_1 + e_2$: $D$ computes recursively $L_1 = D(e_1), L_2 = D(e_2)$ and outputs $L = L_1 \cup L_2$. If there exist elements $s_1, s_2 \in L$ with the same input-profile, $D$ replaces the two elements with a single element $s = s_1 + s_2$ and $\mathsf{Prof}(s) = \mathsf{Prof}(s_1)$. It repeats this process until all the input-profiles in $L$ are distinct and outputs $L$.

- Element $e$ is of the form $e_1 \cdot e_2$: $D$ computes recursively $L_1 = D(e_1), L_2 = D(e_2)$. For every $s_1 \in L_1$ and $s_2 \in L_2$, $D$ adds the expression $s_1 \cdot s_2$ to the output set $L$ and sets $\mathsf{Prof}(s) = \mathsf{Prof}(s_1) \odot \mathsf{Prof}(s_2)$. $D$ then eliminates repeating input-profiles from $L$ as described above, and outputs $L$.

**Remark 1.** *Note that if $s = s_1 \cdot s_2$ such that $\mathsf{Prof}(s_1)_j = 0$ and $\mathsf{Prof}(s_2)_j = 1$, then $\mathsf{Prof}(s)_j = \perp$. Hence, multiplication gates can lead to an element with* invalid *input-profile. This observation will be used often in the later proofs.*

The fact that in the above decomposition algorithm indeed $e = \sum_{s \in D(e)} s$, and that the input profiles are distinct follows from a straightforward induction. Now, we prove a set of claims and

conclude that $D(e)$ runs in polynomial time (see Claim 9). We begin by proving a claim about the relation between the level of encoding of $e$ and a sub-element $e'$ of $e$.

**Claim 7.** *If $e'$ is a sub-element of $e$, then there exists a collection of disjoint sets $\mathcal{C}$ from our set systems $\{\mathbb{S}^j\}_{j\in[\ell]}$, $\mathbb{U}_s$ and $\mathbb{U}_t$ such that the sets in $\mathcal{C}$ are disjoint with $S(e')$ and $S(e) = S(e') \cup \bigcup_{S\in\mathcal{C}} S$.*

The above claim says that if $e'$ is a sub-element of $e$, the set corresponding to the encoding of $e$ can be seen as being derived from the set used for encoding of $e'$. Intuitively, this is true because in obtaining $e$ from $e'$, the set of encoding never shrinks. It remains same with each addition and increases as union of two disjoint sets with each multiplication. Thus, there would exist a collection of sets such that $S(e)$ can be written as the union of this collection of disjoint sets along with the set of $e'$. In other words, there exists a cover for $S(e)$ which involves the set $S(e')$ and some other disjoint sets from our set system.

*Proof.* (of Claim 7) We will prove this claim by induction on the size of $e$. If $e = 1$, i.e. $e$ is a basic element, then the claim trivially holds. If $e = e_1 + e_2$, then either (1) $e' = e$ or (2) $e'$ is a sub-element of either $e_1$ or $e_2$. In the first case, the claim is trivially true. In the second case, let wlog $e'$ be sub-element of $e_1$. Then by induction hypothesis, there exists a collection of disjoint sets $\mathcal{C}$ from our set systems such that the sets in $\mathcal{C}$ are disjoint with $S(e')$ and $S(e_1) = S(e') \cup \bigcup_{S\in\mathcal{C}} S$. The claim follows by noting that $S(e) = S(e_1)$.

Finally, if $e = e_1 \cdot e_2$, either (1) $e' = e$ or (2) $e'$ is a sub-element of either $e_1$ or $e_2$. In the first case, the claim is trivially true. In the second case, let wlog $e'$ be sub-element of $e_1$. Then by induction hypothesis, there exists a collection of disjoint sets $\mathcal{C}_1$ from our set systems such that the sets in $\mathcal{C}_1$ are disjoint with $S(e')$ and $S(e_1) = S(e') \cup \bigcup_{S\in\mathcal{C}_1} S$. Now, for $e_2$ either (1) $e_2$ is a basic element or (2) there exists a basic sub-element $e''$ of $e_2$. In the first case, $\mathcal{C} = \mathcal{C}_1 \cup \{S(e_2)\}$ since for valid multiplication $S(e_1) \cap S(e_2) = \emptyset$. In the second case, we apply the induction hypothesis on $e_2, e''$ and get a collection of sets $\mathcal{C}_2$ and $\mathcal{C} = \mathcal{C}_1 \cup (S(e'') \cup \mathcal{C}_2)$. Note that $S(e'')$ is a union of two disjoint sets from our set system. □

Next, we prove that for elements which can be zero-tested, i.e. elements at the highest level of encoding, all the elements output by the procedure $D$ are single input elements. In this direction,

we first observe that adding two elements does not create new input-profiles. That is, only way to create new profiles is to multiply two elements. As noted in Remark 1, multiplication of two elements can lead to invalid profiles. Here we use the observation that if $e = e_1 \cdot e_2$ has invalid input profile then computations involving $e$ cannot lead to an element at the universe set and cannot be zero-tested. Here we crucially use the properties of straddling sets and Claim 7. More formally,

**Claim 8.** *If $\mathbb{U} = S(e)$ then all the elements in $D(e)$ are single-input elements. Namely, for every $s \in D(e)$ we have that $\mathsf{Prof}(s) \neq \perp$.*

*Proof.* We will prove this claim by contradiction. Let us assume that the claim is false. Then there exists a sub-element $e^{\mathsf{bad}}$ of $e$ such that $D(e^{\mathsf{bad}})$ contains an invalid input-profile but decomposition of all sub-elements of $e^{\mathsf{bad}}$ have valid input-profiles. We now do a case analysis on the structure of $e^{\mathsf{bad}}$.

$e^{\mathsf{bad}}$ cannot be a basic sub-element since input-profile of all basic sub-elements is valid. Also, $e^{\mathsf{bad}}$ cannot be of the form $e_1 + e_2$ because input-profiles in $D(e^{\mathsf{bad}})$ is a union of input-profiles in $D(e_1)$ and $D(e_2)$. Hence, $e^{\mathsf{bad}}$ is of the form $e_1 \cdot e_2$.

The only way $D(e^{\mathsf{bad}})$ contains an invalid input-profile when all input profiles in $D(e_1)$ and $D(e_2)$ are valid is the following: There exists a $s_1 \in D(e_1)$ and $s_2 \in D(e_2)$ such that $\mathsf{Prof}(s_1) \neq \perp$ and $\mathsf{Prof}(s_2) \neq \perp$ but $\mathsf{Prof}(s_1 \cdot s_2) = \perp$. Then, wlog there exists $j \in [\ell]$ such that $\mathsf{Prof}(s_1) = 0$ and $\mathsf{Prof}(s_2) = 1$. From the description of input profiles, there exists a basic sub-element $\hat{e_1}$ of $s_1$ such that $S(\hat{e_1}) \cap \mathbb{U}_j = S^j_{k,0} \in \mathbb{S}^j$ for some $k \in \mathsf{ind}(j)$. Similarly, there exists a basic sub-element $\hat{e_2}$ of $s_2$ such that $S(\hat{e_2}) \cap \mathbb{U}_j = S^j_{k',1} \in \mathbb{S}^j$ for some $k \in \mathsf{ind}(j)$.

Intuitively, using Claim 5, we show that there is no way of combining $\hat{e_1}$ and $\hat{e_2}$ to form a valid element $e$ such that $S(e) \supseteq \mathbb{U}_j$. For this, we critically use the properties of the straddling set system and the fact that the set used for encoding only grows as union of two disjoint sets (as we do more multiplications). Hence, to obtain $e$ using $\hat{e_1}$ and $\hat{e_2}$, we need to find a collection of disjoint sets whose union along with $S(\hat{e_1})$ and $S(\hat{e_2})$ gives $\mathbb{U}$. This is not possible by properties of straddling sets. More formally, we have the following:

Since, $\hat{e_1}$ is a basic sub-element of $s_1$, by Claim 7, there exists a collection $\mathcal{C}_1$ such that $S(s_1) = S(\hat{e_1}) \cup \bigcup_{S \in \mathcal{C}_1} S$. Similarly, there exists a collection $\mathcal{C}_2$ such that $S(s_2) = S(\hat{e_2}) \cup \bigcup_{S \in \mathcal{C}_2} S$.

Since $(s_1 \cdot s_2)$ is a valid multiplication, $\left(S(\hat{e_1}) \cup \bigcup_{S \in \mathcal{C}_1} S\right) \bigcup \left(S(\hat{e_2}) \cup \bigcup_{S \in \mathcal{C}_2} S\right) = S(s_1 \cdot s_2) = S(e_1 \cdot e_2) = S(e^{\mathsf{bad}})$.

Again, since $e^{\mathsf{bad}}$ is a sub-element of $e$, using Claim 7, there exists a collection $\mathcal{C}$ such that $S(e^{\mathsf{bad}})$ and $\mathcal{C}$ form a cover for $S(e)$. This implies that there is an exact cover of $\mathbb{U}$ using both $S^j_{k,0}$ and $S^j_{k',1}$ for some $k, k' \in \mathsf{ind}(j), j \in [\ell]$. This is a contradiction to Claim 5 for straddling set system $\mathbb{S}^j$ for $\mathbb{U}_j$. □

Finally, we prove the main claim of this section that $D$ runs in polynomial time. First observe that only multiplication can create new input profiles. We show that if $e$ is an element of the form $e_1 \cdot e_2$ and $D(e)$ contains a new input-profile then $e$ must itself be a single-input element (that is, $D(e)$ will be the singleton set $\{e\}$). This means that the number of elements in the decomposition of $e$ is bounded by the number of sub-elements of $e$, and therefore is polynomial. To prove the above we first observe that if $D(e)$ is not a singleton, then either $D(e_1)$ or $D(e_2)$ are also not singletons. Then we show that if $D(e_1)$ contains more than one input-profile then all input-profiles in $D(e_1)$ must be complete. Here again we use the structure of the straddling set system and therefore the multiplication $e_1 \cdot e_2$ cannot contain any new profiles.

**Claim 9.** *$D(e)$ runs in polynomial time, i.e. number of elements in $D(e)$ is polynomial.*

*Proof.* Observe that the running time of $D$ on $e$ is polynomial in the number of the single-input elements in $D(e)$. Hence, to show that $D$ runs in polynomial time, we will show that the size of the set $D(e)$ is bounded by the number of sub-elements of $e$. More precisely, for each $s \in D(e)$, we show a single-input sub-element $e'$ of $e$ such that $\mathsf{Prof}(e') = \mathsf{Prof}(s)$. Since $D(e)$ has single input elements with distinct profiles, we get that $|D(e)|$ is polynomial since $e$ has a polynomial number of sub-elements.

For each $s \in D(e)$, let $e'$ be the first sub-element of $e$ such that $D(e')$ contains a single input element with input-profile $\mathsf{Prof}(s)$ and decomposition of no sub-element of $e'$ contains a single-input element with input-profile $\mathsf{Prof}(s)$. Then we claim that $e'$ is a single input element, i.e. $D(e') = \{e'\}$. We have the following cases.

$e'$ is a basic sub-element of $e$, then by definition, $D(e') = \{e'\}$. Next, if $e' = e_1 + e_2$, then all the input-profiles in $D(e')$ are either in $e_1$ or $e_2$. That is, $e'$ cannot be the first sub-element of $e$ which

contains the input profile $\mathsf{Prof}(s)$. Finally, let $e' = e_1 \cdot e_2$. We need to show that $D(e') = \{e'\}$. Suppose not, that is $|D(e')| > 1$. In this case, we will show that $D(e')$ cannot contain any new input profiles. Let $s' \in D(e')$ such that $\mathsf{Prof}(s) = \mathsf{Prof}(s')$.

By the definition of $D$, either $|D(e_1)| > 1$ or $D(e_2) > 1$. Wlog, let us assume that $D(e_1) > 1$, that is there exists $s_{11}, s_{12} \in D(e_1)$ and $s_2 \in D(e_2)$ such that $s' = s_{11} \cdot s_2$. By the definition of $D$, it holds that $S(s_{11}) = S(s_{12})$ and since the all the input-profiles in the decompisition are distinct $\mathsf{Prof}(s_{11}) \neq \mathsf{Prof}(s_{12})$. Wlog, there exists a $j \in [\ell]$ such that $\mathsf{Prof}(s_{11})_j = 0$ and $\mathsf{Prof}(s_{12})_j \in \{1, *\}$.

First, we claim that if $S(s_{11}) = S(s_{12})$ and $\mathsf{Prof}(s_{11})_j = 0$ then $\mathsf{Prof}(s_{12})_j \neq *$. By the definition of input-profiles, $S(x) \cap \mathbb{U}_j = \emptyset$ if and only if $\mathsf{Prof}(x)_j = *$. Hence, if $\mathsf{Prof}(s_{11})_j = 0$ and $\mathsf{Prof}(s_{12})_j = *$ then $S(s_{11}) \cap \mathbb{U}_j \neq \emptyset$ and $S(s_{12}) \cap \mathbb{U}_j = \emptyset$. Then, $S(s_{11}) \neq S(s_{12})$, which is a contradiction.

The remaining case is $\mathsf{Prof}(s_{11})_j = 0$ and $\mathsf{Prof}(s_{12})_j = 1$. We claim that there is no basic sub-element $s'_{11}$ of $s_{11}$ such that $S(s'_{11}) \cap \mathbb{U}_j = S^j_{k,1}$. If this not true, then $\mathsf{Prof}(s_{11}) = \perp$. Similarly, for $s_{12}$, there is no basic sub-element $s'_{12}$ such that $S(s'_{12}) \cap \mathbb{U}_j = S^j_{k,0}$. This means that $s_{11}$ and $s_{12}$ have consistently used $x_j = 0$ and $x_j = 1$ in their evaluation. Now, by Claim 6, for $S(s_{11}) = S(s_{12})$ it has to be the case that $\mathbb{U}_j \subseteq S(s_{11}) = S(s_{12})$. By Claim 10, $\mathsf{Prof}(s_{11})$ is complete. But, multiplying an element with complete profile to another element cannot lead to any new *valid* profile. Hence, we get a contradiction to the assumption on $e'$.

**Claim 10.** *If $s$ is a single-input element such that $\mathbb{U}_j \subseteq S(s)$ for some $j \in [\ell]$, then $\mathsf{Prof}(s)$ is complete.*

*Proof.* Since $s$ is a single input element, $\mathsf{Prof}(s)_j \neq \perp$. Also, $\mathsf{Prof}(s)_j \neq *$ because $S(s) \cap \mathbb{U}_j \neq \emptyset$. Let $\mathsf{Prof}(s) = b$ for some $b \in \{0, 1\}$. Also, since $\mathbb{U}_j \subseteq S(s)$, for every $i \in \mathsf{ind}(j)$ there exists a basic sub-element $s_i$ of $s$ such that $S(s_i) \cap \mathbb{U}_j = S^j_{i,b}$. Moreover, $S(s_i) = S(i, b_1, b_2)$ such that $\mathsf{Prof}(s)_{\mathsf{inp}_1(i)} = b_1$ and $\mathsf{Prof}(s)_{\mathsf{inp}_2(i)} = b_2$.

We will show that for any $k \in [\ell]$, $\mathsf{Prof}(s)_k \neq *$. By the property of dual input relaxed matrix branching program, there exists $i^* \in [n]$ such that wlog, $(\mathsf{inp}_1(i^*), \mathsf{inp}_2(i^*)) = (j, k)$. Since $\mathbb{U}_j \subseteq S(s)$, there exists a basic sub-element $s_{i^*}$ of $s$ such that $S(s_{i^*}) = S(i^*, b_1, b_2)$. Since

$\mathsf{inp}_2(i) = k$, $\mathsf{Prof}(s)_k \neq *$. □

□

### 3.7.2 Simulation of Zero-testing

We first describe the simulation of the zero-testing at a high level and then will formally describe the simulation. The simulator uses the decomposition algorithm defined in the previous section to decompose the element $e$, that is to be zero tested, into single-input elements. Zero-testing of $e$ essentially involves zero-testing every element in its decomposition. Then we establish that if $e$ corresponds to a zero polynomial then indeed every element in the decomposition of $e$ should correspond to a zero polynomial. The intuition is that every element in its decomposition has product of $\alpha$'s which is different for every in its decomposition. And hence, with negligible probability it happens that the $\alpha$'s cancel out and yield a zero-polynomial. The only part left is to show that indeed we can perform zero-testing on every element in decomposition individually. To perform this we use the simulation algorithm defined in Section 3.3. We evaluate the polynomial corresponding to the single-input element on the output of the simulation algorithm. We then argue that the probability that if the single-input element was indeed a non-zero polynomial then with negligible probability the polynomial evaluates to 0. This establishes that if the polynomial is a non-zero polynomial then we can indeed detect some single-input element in its decomposition to be non-zero with overwhelming probability.

We now describe zero testing performed by the simulator $\mathsf{Sim}$. Denote the element to be zero tested to be $e$ and denote the polynomial computed by the circuit $\alpha(e)$ by $p_e$.

1. $\mathsf{Sim}$ first executes the decomposition algorithm $D$ described before on $e$. Denote the set of resulting single-input elements by $D(e)$. The output of $\mathsf{Sim}$ is either "Zero" or "Non-zero" depending on whether the element is zero or not.

2. For every $s \in D(e)$ execute the following steps:

   (a) Find the input $x$ that corresponds to the element $s$. More formally, denote $x$ by $\mathsf{Prof}(s)$.

It then queries the $\mathsf{F}$ oracle on $x$ to obtain $\mathsf{F}(x)$.

(b) Execute $\mathsf{Sim}_{\mathsf{BP}}$ on input $(1^s, \mathsf{F}(x))$, where $s$ is the size of the formula $\mathsf{F}$ to obtain the following distribution represented by the random variable $\mathcal{V}_s^{\mathsf{Sim}}$.

$$\left\{\tilde{s}, \tilde{B}_{i,b_1^i,b_2^i}, \tilde{t} : i \in [n], b_1^i = x_{\mathsf{inp}_1(i)}, b_2^i = x_{\mathsf{inp}_2(i)}\right\}$$

(c) We evaluate the polynomial $p_s$, which is the polynomial computed by the circuit $\alpha(s)$, on $\mathcal{V}_s^{\mathsf{Sim}}$. If the evaluation yields a non-zero result then $\mathsf{Sim}$ outputs "Non-zero".

3. For all $s \in D(e)$, if $p_s(\mathcal{V}_s^{\mathsf{Sim}}) = 0$ then $\mathsf{Sim}$ outputs "Zero".

This completes the description of the zero-testing as performed by the simulator. We now argue that the simulator runs in polynomial time.

*Running time*. From Claim 9 it follows that the first step, which is the execution of the decomposition algorithm, takes polynomial time. We now analyse the running time of the steps $(a), (b)$ and $(c)$. Step $(a)$ takes linear time. The running time of Step $(b)$ is essentially the running time of $\mathsf{Sim}_{\mathsf{BP}}$ which is again polynomial. Finally, Step $(c)$ is executed in time which is proportional to the number of queries made by the adversary to the oracle $\mathcal{O}(\mathcal{M})$ which are simulated by the simulator. Since the number of queries is polynomial, even Step $(c)$ is executed in polynomial time. Finally we argue that the Steps $(a), (b)$ and $(c)$ are executed polynomially many times. This follows from Claim 9 which shows that the number of elements in the decomposition is polynomial and hence the number of iterations is polynomial. Hence, our simulator runs in polynomial time.

We prove the following two claims about the structure of the polynomial representing the element to be zero tested that establishes the correctness of simulation. This will be useful when we will show later that element is zero iff all the elements obtained by its decomposition are zero.

**Claim 11.** *Consider an element $e$ such that $U \subseteq S(e)$. The polynomial computed by the circuit $\alpha(e)$, denoted by $p_e$, can be written as follows.*

$$p_e = \sum_{s \in D(e)} p_s = \sum_{s \in D(e)} q_{\mathsf{Prof}(s)} \cdot \tilde{\alpha}_{\mathsf{Prof}(s)}$$

*where for every $s \in D(e)$ the following holds.*

1. *The value $\tilde{\alpha}_{\mathsf{Prof}(s)}$ denotes the product $\prod_{i\in[n]} \alpha_{i,b_1^i,b_2^i}$ where $(b_1^i, b_2^i) = (\mathsf{Prof}(s)_{\mathsf{inp}_1(i)}, \mathsf{Prof}(s)_{\mathsf{inp}_2(i)})$.*
2. *$q_{\mathsf{Prof}(s)}$ is a polynomial in $\tilde{s}, \tilde{t}$ and in the entries of $\tilde{B}_{i,b_1^i,b_2^i}$. Further the degree of every variable in $q_{\mathsf{Prof}(s)}$ is 1.*

*Proof.* Consider an element $s \in D(e)$. As before denote the circuit representing $s$ by $\alpha(s)$. Alternately, we view $\alpha(s)$ as a polynomial with the $k^{th}$ monomial being represented by $s_k$. Moreover, the value $s_k$ satisfies the following three properties.

- For every $s_k$ we have that $S(s_k) = S(s)$ and therefore $U_j \subseteq S(s_k)$ for every $j \in [l]$.
- The circuit $\alpha(s_k)$ contains only multiplication gates.
- The basic sub-elements of each $s_k$ are a subset of the basic sub-elements of some $s$

From the first property and Claim 10, we have that $\mathsf{Prof}(s_k)$ is complete. Since every basic sub-element of $s_k$ is a also a sub-element of $s$ and also because $s$ is a sinlge-input element we have that $\mathsf{Prof}(s_k) = \mathsf{Prof}(s)$. Further for every $i \in [l]$, there exists a basic sub-element $e'$ of $s_k$ such that $S(e') = S(i, b_1^i, b_2^i)$ for $b_1^i = \mathsf{Prof}(s_k)_{\mathsf{inp}_1(i)}$ and $b_2^i = \mathsf{Prof}(s_k)_{\mathsf{inp}_2(i)}$. There can be many such basic sub-elements but the second property ensures that there is a unique such element. The only basic elements given to the adversary as part of the obfuscation with index set $S(i, b_1^i, b_2^i)$ are the elements $\alpha_{i,b_1^i,b_2^i} \cdot \tilde{B}_{i,b_1^i,b_2^i}$. From this it follows that we can write the polynomial $p_s$ as $q_{\mathsf{Prof}(s)} \cdot \tilde{\alpha}_{\mathsf{Prof}(s)}$ where $q_{\mathsf{Prof}(s)}$ and $\tilde{\alpha}_{\mathsf{Prof}(s)}$ are described in the claim statement. □

Before we describe the next claim we will introduce some notation. Consider a random variable $X$. Let $g$ be a polynomial. We say that $g(X) \equiv 0$ if $g$ is 0 on all the support of $X$. We define $\mathcal{V}_C^{\mathsf{real}}$ to be the distribution of the assignment of the values to $p_e$.

**Claim 12.** *Consider an element $e$. Let $p_e$ be a polynomial of degree $\mathsf{poly}(n)$ represented by $\alpha(C)$. If $p_e \not\equiv 0$ then the following holds.*

$$\mathsf{Pr}_{\mathcal{V}_C^{\mathsf{real}}}[p_e(\mathcal{V}_C^{\mathsf{real}}) = 0] = \mathsf{negl}(n)$$

*Proof.* The claim would directly follow from Schwartz-Zippel lemma if the distribution corresponding to the random variable $\mathcal{V}_C^{\mathsf{real}}$ is a uniform distribution or even if the distribution could be computed by a low degree polynomial over values uniformly distributed over $\mathbb{Z}_p$. But this is not true since the entries in $R^{-1}$ cannot be expressed as a polynomial in the entries of $R$. To this end, we do the following. We transform $p_e$ into another polynomial $p'_e$ and further transform $\mathcal{V}_C^{\mathsf{real}}$ into another distribution $\tilde{\mathcal{V}}_C^{\mathsf{real}}$ such that the following holds:

- $\underset{\mathcal{V}_C^{\mathsf{real}}}{\mathsf{Pr}}[p_e(\mathcal{V}_C^{\mathsf{real}}) = 0] = \underset{\tilde{\mathcal{V}}_C^{\mathsf{real}}}{\mathsf{Pr}}[p'_e(\tilde{\mathcal{V}}_C^{\mathsf{real}}) = 0]$
- The degree of $p'_e = \mathsf{poly}(n)$.
- The distribution corresponding to $\mathcal{V}_C^{\mathsf{real}}$ can be computed by a polynomial over values that are uniform over $\mathbb{Z}_p$.

In order to obtain $p'_e$ from $p_e$ we essentially replace the matrices $R_i^{-1}$ in $p_e$ with adjugate matrices $\mathsf{adj}(R_i) \prod_{j \neq i} \mathsf{det}(R_j)$ where $\mathsf{adj}(R_i) = R_i^{-1} \cdot \mathsf{det}(R_i)$. In a similar way we obtain $\tilde{\mathcal{V}}_C^{\mathsf{real}}$ from $\mathcal{V}_C^{\mathsf{real}}$ by replacing all the assignment values corresponding to $R_i^{-1}$ by assignment values corresponding to $\mathsf{adj}(R_i) \prod_{j \neq i} \mathsf{det}(R_j)$.

We now argue $p'_e$ satisfies all the three properties stated above. The following shows that the first property is satisfied.

$$\begin{aligned} \underset{\mathcal{V}_C^{\mathsf{real}}}{\mathsf{Pr}}[p_e(\mathcal{V}_C^{\mathsf{real}}) = 0] &= \underset{\mathcal{V}_C^{\mathsf{real}}}{\mathsf{Pr}}[p_e(\mathcal{V}_C^{\mathsf{real}}) \prod_{i \in [n]} \mathsf{det}(R_j) = 0] \\ &= \underset{\tilde{\mathcal{V}}_C^{\mathsf{real}}}{\mathsf{Pr}}[p'_e(\tilde{\mathcal{V}}_C^{\mathsf{real}}) = 0] \end{aligned}$$

We now show that the second property is satisfied. The degree of $\prod_{i \in [n]} \mathsf{det}(R_i)$ is at most $n \cdot w$ and hence the degree of $p'_e$ is at most $n \cdot w$ times the degree of $p_e$, which is still a polynomial in $n$. Finally, we show that that the third property is satisfied. To see this note that $\mathsf{adj}(R_i)$ can be expressed as polynomial with degree at most $w$ in the entries of $R_i$. Using this, we have that the distribution corresponding to $\tilde{\mathcal{V}}_C^{\mathsf{real}}$ can be computed by a polynomial (of degree at most $w$) over values that are uniform over $\mathbb{Z}_p$.

Now that we have constructed the polynomial $p'_e$, we will invoke the Schwartz-Zippel lemma

on $p'_e$ to obtain the desired result as follows:

$$\underset{\mathcal{V}_C^{\mathsf{real}}}{\mathsf{Pr}}[p_e(\mathcal{V}_C^{\mathsf{real}}) = 0] = \underset{\tilde{\mathcal{V}}_C^{\mathsf{real}}}{\mathsf{Pr}}[p'_e(\tilde{\mathcal{V}}_C^{\mathsf{real}}) = 0] = \mathsf{negl}(n)$$

We now show that in order to zero-test an element it suffices to individually zero-test all the elements in its decomposition. This will complete the proof that our simulator satisfies the correctness property.

**Theorem 12.** *Consider an element $e$ such that $U \subseteq S(e)$ and let $p_e$ be the polynomial computed by the circuit $\alpha(e)$. We have the following:*

- *If $p_e$ is a non-zero polynomial then $p_s(\mathcal{V}_C^{\mathsf{real}}) = 0$ with negligible (in $n$) probability, for some $s \in D(e)$.*
- *If $p_e$ is a zero polynomial then $p_s(\mathcal{V}_C^{\mathsf{real}}) \equiv 0$*

*Proof.* We first consider the case when $p_e$ is a non-zero polynomial. From Claim 12, we have that $\mathsf{Pr}_{\mathcal{V}_C^{\mathsf{real}}}[p_e(\mathcal{V}_C^{\mathsf{real}}) = 0] = 0$ with negligible probability. Further since $p_e = \sum_{s \in D(e)} p_s$, we have the following.

$$\begin{aligned} \mathsf{Pr}_{\mathcal{V}_C^{\mathsf{real}}}[p_e(\mathcal{V}_C^{\mathsf{real}}) = 0] &= \mathsf{Pr}_{\mathcal{V}_C^{\mathsf{real}}}[\exists s \in D(e) \ : \ p_s(\mathcal{V}_C^{\mathsf{real}}) = 0] \\ &= \mathsf{negl}(n) \end{aligned}$$

Further We now move to the case when $p_e$ is a zero polynomial. We claim that $p_s$ is a zero polynomial for every $s \in D(e)$. From Claim 12 we know that $p_s$ can be expressed as $q_{\mathsf{Prof}(s)} \cdot \tilde{\alpha}_{i,b_1^i,b_2^i}$, where $(b_1^i, b_2^i) = (\mathsf{Prof}(s)_{\mathsf{inp}_1(i)}, \mathsf{Prof}(s)_{\mathsf{inp}_2(i)})$. Observe that the marginal distribution of $\tilde{\alpha}_{\mathsf{Prof}(s)}$ is uniform for every $s \in D(e)$. Hence, $q_{\mathsf{Prof}(s)}$ should be zero on all points of its support. In other words, $q_{\mathsf{Prof}(s)} \equiv 0$ and hence, $p_s \equiv 0$ thus proving the theorem □

As a consequence of the above theorem, we prove the following corollary.

**Corollary 13.** *Consider an element $e$ such that $U \subseteq S(e)$ and let $p_e$ be the polynomial computed by the circuit $\alpha(e)$. We have the following.*

- *If $p_e$ is a non-zero polynomial then $p_s(\mathcal{V}_s^{\mathsf{Sim}}) = 0$ with negligible (in $n$) probability, for some $s \in D(e)$.*

- *If $p_e$ is a zero polynomial then $p_s(\mathcal{V}_s^{\mathsf{Sim}}) \equiv 0$.*

The proof of the above corollary follows from the above theorem and the following claim. This completes the proof of correctness of the simulation of zero-testing.

**Claim 13.** *For every single-input element $s$ such that $U \subseteq S$ we have that the assignment $\mathcal{V}_s^{\mathsf{Sim}}$, which is the distribution output by $\mathsf{Sim}_{\mathsf{BP}}$, and the assignment to the same subset of variables in $\mathcal{V}_C^{\mathsf{real}}$ are identically distributed.*

*Proof.* The distributions of the following variables generated by $\mathsf{Sim}$ and $\mathcal{O}(\mathsf{F})$ are identical from Theorem 9:

$$R_0, \left\{ B_{i,b_1^i,b_2^i} \;\; i \in [n], b_1^i = \mathsf{Prof}(s)_{\mathsf{inp}_1(i)}, b_2^i = \mathsf{Prof}(s)_{\mathsf{inp}_2(i)} \right\}, R_n$$

Further, the following variables are sampled uniformly at random both by $\mathsf{Sim}$ and by $\mathcal{O}(\mathsf{F})$:

$$\left\{ \alpha_{i,b_1^i,b_2^i} \;:\; i \in [n], b_1^i = \mathsf{Prof}(s)_{\mathsf{inp}_1(i)}, b_2^i = \mathsf{Prof}(s)_{\mathsf{inp}_2(i)} \right\}$$

The claim follows from the fact that the assignment $\mathcal{V}_s^{\mathsf{Sim}}$ generated by $\mathsf{Sim}$ and the assignment to the same subset of variables in $\mathcal{V}_C^{\mathsf{real}}$ are both computed from the above values in the same way. □

# REFERENCES


[ABG13] Prabhanjan Ananth, Dan Boneh, Sanjam Garg, Amit Sahai, and Mark Zhandry. "Differing-Inputs Obfuscation and Applications." *IACR Cryptology ePrint Archive*, **2013**:689, 2013. 3

[App14] Benny Applebaum. "Bootstrapping Obfuscators via Fast Pseudorandom Functions." In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, pp. 162–172, 2014. 13, 18

[Bar86] David A. Mix Barrington. "Bounded-Width Polynomial-Size Branching Programs Recognize Exactly Those Languages in $NC^1$." In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pp. 1–5, 1986. 13, 104, 109

[BB94] Maria Luisa Bonet and Samuel R. Buss. "Size-Depth Tradeoffs for Boolean Fomulae." *Inf. Process. Lett.*, **49**(3):151–155, 1994. 14, 15

[BBC14] Boaz Barak, Nir Bitansky, Ran Canetti, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. "Obfuscation for Evasive Functions." In *Theory of Cryptography - 11th Theory of Cryptography Conference, TCC 2014, San Diego, CA, USA, February 24-26, 2014. Proceedings*, pp. 26–51, 2014. 3

[BCC14] Nir Bitansky, Ran Canetti, Henry Cohn, Shafi Goldwasser, Yael Tauman Kalai, Omer Paneth, and Alon Rosen. "The Impossibility of Obfuscation with Auxiliary Input or a Universal Simulator." In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, pp. 71–89, 2014. 3

[BCP14] Elette Boyle, Kai-Min Chung, and Rafael Pass. "On Extractability Obfuscation." In *Theory of Cryptography - 11th Theory of Cryptography Conference, TCC 2014, San Diego, CA, USA, February 24-26, 2014. Proceedings*, pp. 52–73, 2014. 3

[BFM14] Christina Brzuska, Pooya Farshim, and Arno Mittelbach. "Indistinguishability Obfuscation and UCEs: The Case of Computationally Unpredictable Sources." In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, pp. 188–205, 2014. 3

[BGI01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. "On the (Im)possibility of Obfuscating Programs." In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, pp. 1–18, 2001. 1, 10, 19, 20

[BGI14] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. "Functional Signatures and Pseudorandom Functions." In *Public-Key Cryptography - PKC 2014 - 17th International*

*Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings*, pp. 501–519, 2014. 22, 23

[BGK14] Boaz Barak, Sanjam Garg, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. "Protecting Obfuscation against Algebraic Attacks." In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, pp. 221–238, 2014. 4, 12, 13, 15, 16, 17, 19, 20, 101, 103, 104, 112, 119, 121, 125, 129

[BH15] Mihir Bellare and Viet Tung Hoang. "Adaptive Witness Encryption and Asymmetric Password-Based Cryptography." In *Public-Key Cryptography - PKC 2015 - 18th IACR International Conference on Practice and Theory in Public-Key Cryptography, Gaithersburg, MD, USA, March 30 - April 1, 2015, Proceedings*, pp. 308–331, 2015. 3

[BP15] Elette Boyle and Rafael Pass. "Limits of Extractability Assumptions with Distributional Auxiliary Input." In *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*, pp. 236–261, 2015. 3

[BR93] Mihir Bellare and Phillip Rogaway. "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols." In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993.*, pp. 62–73, 1993. 20

[BR14a] Zvika Brakerski and Guy N. Rothblum. "Black-box obfuscation for d-CNFs." In *Innovations in Theoretical Computer Science, ITCS'14, Princeton, NJ, USA, January 12-14, 2014*, pp. 235–250, 2014. 3

[BR14b] Zvika Brakerski and Guy N. Rothblum. "Virtual Black-Box Obfuscation for All Circuits via Generic Graded Encoding." In *Theory of Cryptography - 11th Theory of Cryptography Conference, TCC 2014, San Diego, CA, USA, February 24-26, 2014. Proceedings*, pp. 1–25, 2014. 3, 12, 13, 15, 16, 17, 104

[BST14] Mihir Bellare, Igors Stepanovs, and Stefano Tessaro. "Poly-Many Hardcore Bits for Any One-Way Function and a Framework for Differing-Inputs Obfuscation." In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, pp. 102–121, 2014. 3

[BW13] Dan Boneh and Brent Waters. "Constrained Pseudorandom Functions and Their Applications." In *Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II*, pp. 280–300, 2013. 22, 23

[CFI03] Ronald Cramer, Serge Fehr, Yuval Ishai, and Eyal Kushilevitz. "Efficient Multi-party Computation over Rings." In *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, pp. 596–613, 2003. 14, 112, 114

[CGH04] Ran Canetti, Oded Goldreich, and Shai Halevi. "The random oracle methodology, revisited." *J. ACM*, **51**(4):557–594, 2004. 20

[Cle90] Richard Cleve. "Towards Optimal Simulations of Formulas by Bounded-Width Programs." In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pp. 271–277, 1990. 15, 109

[CLT13] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. "Practical Multilinear Maps over the Integers." In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pp. 476–493, 2013. 14, 16

[DH76] Whitfield Diffie and Martin E. Hellman. "Multiuser cryptographic techniques." In *American Federation of Information Processing Societies: 1976 National Computer Conference, 7-10 June 1976, New York, NY, USA*, pp. 109–112, 1976. 1

[EFF85] P Erdös, P. Frankl, and Z. Füredi. "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. 11, 25

[FKN94] Uriel Feige, Joe Kilian, and Moni Naor. "A minimal model for secure computation (extended abstract)." In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23-25 May 1994, Montréal, Québec, Canada*, pp. 554–563, 1994. 14, 15, 105

[FLS99] Uriel Feige, Dror Lapidot, and Adi Shamir. "Multiple NonInteractive Zero Knowledge Proofs Under General Assumptions." *SIAM J. Comput.*, **29**(1):1–28, 1999. 24

[GGG14] Shafi Goldwasser, S. Dov Gordon, Vipul Goyal, Abhishek Jain, Jonathan Katz, Feng-Hao Liu, Amit Sahai, Elaine Shi, and Hong-Sheng Zhou. "Multi-input Functional Encryption." In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, pp. 578–602, 2014. 3, 7, 8, 9

[GGH13a] Sanjam Garg, Craig Gentry, and Shai Halevi. "Candidate Multilinear Maps from Ideal Lattices." In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, pp. 1–17, 2013. 13, 14, 16, 119

[GGH13b] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. "Candidate Indistinguishability Obfuscation and Functional Encryption for all Circuits." In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pp. 40–49, 2013. 3, 4, 9, 12, 13, 15, 16, 17, 21, 22, 23, 34, 63, 103, 104, 112

[GGH14a] Sanjam Garg, Craig Gentry, Shai Halevi, and Mariana Raykova. "Two-Round Secure MPC from Indistinguishability Obfuscation." In *Theory of Cryptography - 11th Theory of Cryptography Conference, TCC 2014, San Diego, CA, USA, February 24-26, 2014. Proceedings*, pp. 74–94, 2014. 3

[GGH14b] Sanjam Garg, Craig Gentry, Shai Halevi, and Daniel Wichs. "On the Implausibility of Differing-Inputs Obfuscation and Extractable Witness Encryption with Auxiliary Input." In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, pp. 518–535, 2014. 3

[GGM84] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. "How to Construct Random Functions (Extended Abstract)." In *25th Annual Symposium on Foundations of Computer Science, West Palm Beach, Florida, USA, 24-26 October 1984*, pp. 464–479, 1984. 23

[GHR14] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. "Outsourcing Private RAM Computation." In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pp. 404–413, 2014. 3, 7, 32, 33

[Gie01] Oliver Giel. "Branching Program Size Is Almost Linear in Formula Size." *J. Comput. Syst. Sci.*, **63**(2):222–235, 2001. 15, 16, 17, 19, 111

[GIS10] Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. "Founding Cryptography on Tamper-Proof Hardware Tokens." In *Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings*, pp. 308–326, 2010. 13, 18

[GJK15] Vipul Goyal, Abhishek Jain, Venkata Koppula, and Amit Sahai. "Functional Encryption for Randomized Functionalities." In *Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II*, pp. 325–351, 2015. 3

[HSW14] Susan Hohenberger, Amit Sahai, and Brent Waters. "Replacing a Random Oracle: Full Domain Hash from Indistinguishability Obfuscation." In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, pp. 201–220, 2014. 3

[Juk12] Stasys Jukna. *Boolean Function Complexity - Advances and Frontiers*, volume 27 of *Algorithms and combinatorics*. Springer, 2012. 14

[Khr78] V.M. Khrapchenko. "On a relation between the complexity and the depth." *Metody Diskretnogo Analiza Synthesis of Control Systems*, **32**:76–94, 1978. 14

[Kil88] Joe Kilian. "Founding Cryptography on Oblivious Transfer." In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pp. 20–31, 1988. 114

[KNY14] Ilan Komargodski, Moni Naor, and Eylon Yogev. "Secret-Sharing for NP." In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, pp. 254–273, 2014. 3

[KPT13] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. "Delegatable pseudorandom functions and applications." In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pp. 669–684, 2013. 22, 23

[KRS99] Ravi Kumar, Sridhar Rajagopalan, and Amit Sahai. "Coding Constructions for Blacklisting Problems without Computational Assumptions." In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pp. 609–623, 1999. 11, 25

[LS14] Hyung Tae Lee and Jae Hong Seo. "Security Analysis of Multilinear Maps over the Integers." In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, pp. 224–240, 2014. 16

[LSS14] Adeline Langlois, Damien Stehlé, and Ron Steinfeld. "GGHLite: More Efficient Multilinear Maps from Ideal Lattices." In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, pp. 239–256, 2014. 16

[Mas76] William J. Masek. "A fast algorithm for the string editing problem and decision graph complexity.", 1976. 15, 109

[MO14] Antonio Marcedone and Claudio Orlandi. "Obfuscation $\Rightarrow$ (IND-CPA Security !$\Rightarrow$ Circular Security)." In *Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings*, pp. 77–90, 2014. 3

[MR13] Tal Moran and Alon Rosen. "There is no Indistinguishability Obfuscation in Pessiland." *IACR Cryptology ePrint Archive*, **2013**:643, 2013. 3

[NY90] Moni Naor and Moti Yung. "Public-key Cryptosystems Provably Secure against Chosen Ciphertext Attacks." In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pp. 427–437, 1990. 34, 63

[PM76] Franco P. Preparata and David E. Muller. "Efficient Parallel Evaluation of Boolean Expression." *IEEE Trans. Computers*, **25**(5):548–549, 1976. 14, 19

[PPS15] Omkant Pandey, Manoj Prabhakaran, and Amit Sahai. "Obfuscation-Based Non-black-box Simulation and Four Message Concurrent Zero Knowledge for NP." In *Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II*, pp. 638–667, 2015. 3

[PST14] Rafael Pass, Karn Seth, and Sidharth Telang. "Indistinguishability Obfuscation from Semantically-Secure Multilinear Encodings." In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, pp. 500–517, 2014. 104

[PZ93] Mike Paterson and Uri Zwick. "Shallow Circuits and Concise Formulae for Multiple Addition and Multiplication." *Computational Complexity*, **3**:262–291, 1993. 18

[Sah99] Amit Sahai. "Non-Malleable Non-Interactive Zero Knowledge and Adaptive Chosen-Ciphertext Security." In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pp. 543–553, 1999. 34, 63

[Ser14] I. S. Sergeev. "Upper bounds for the formula size of symmetric Boolean functions." *Russian Mathematics*, **58 (5)**:30–42, 2014. 18

[SW14] Amit Sahai and Brent Waters. "How to use indistinguishability obfuscation: deniable encryption, and more." In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pp. 475–484, 2014. 3, 11, 12, 22, 38, 67

[SWW99] Martin Sauerhoff, Ingo Wegener, and Ralph Werchner. "Relating Branching Program Size and Formula Size over the Full Binary Basis." In *STACS 99, 16th Annual Symposium on Theoretical Aspects of Computer Science, Trier, Germany, March 4-6, 1999, Proceedings*, pp. 57–67, 1999. 109