Responsible Delegation of Computation With User Privacy

James Bartusek and Orr Paradise

UC Berkeley

Abstract. Cloud computing allows a computationally weak user to delegate computation to a powerful server. The unprecedented accessibility of enormous computing power is not always used for good: malicious users may take advantage of powerful cloud servers to carry out ill deeds, as demonstrated by the 2011 Sony hack that was launched from the Amazon EC2 cloud platform. This highlights the importance of *responsible* delegation of computation services that actively attempt to prevent malicious use of their platform.
Meanwhile, recent cryptographic developments are enabling *private* delegation of computation, allowing users to delegate computation on sensitive data without disclosing anything about it to the server. *Responsibility* and *privacy* seem to be at odds: how can we expect cloud services to prevent misuse of their platform if they cannot even look at the delegated computation?
In this work, we devise a theoretical framework to model misuse-prevention by cloud services, offering a rigorous definition of *responsible* delegation of computation. We then give an answer to the above question by presenting a cryptographic compiler that takes any (publicly-verifiable) one-round delegation protocol and produces a responsible and private one-round delegation protocol, using *fully-homomorphic encryption* and *non-interactive zero-knowledge arguments.*

1 Introduction

In 2011, attackers breached Sony's Playstation Network, compromising the personal data of tens of millions of users. The stolen data included passwords, purchase histories, and possibly credit card information [21]. At the time, this hack was one of the largest data breaches ever discovered, but another aspect makes this breach stand out from others: the hackers launched their attack from Amazon EC2, a cloud computing service [14]. Amazon later claimed to be employing "a number of automated detection and mitigation techniques to prevent the misuse of [their] services" [37]; though the exact tools used by Amazon remain secret, other researchers are openly developing tools to detect misuse of cloud platforms [12, 13, 31].

Meanwhile, recent advances in cryptography are enabling *private* delegation of computation, allowing users to delegate computation on sensitive data without disclosing anything about the data to the server. This promising development brings the power of cloud computing to regulated domains previously out of reach, such as remote health monitoring [29, 28] and the aggregation of medical and financial data [35, 34, 11].

These two developments—malicious uses of the cloud, and private delegation of computation—stand as opposites in a state of tension. On the one hand, cloud computation can be used by malefactors to carry out their schemes. As cloud computing becomes cheaper and more accessible, responsible cloud service providers should take active measures to prevent such misuse. On the other hand, private delegation (desirably) prevents the cloud service provider from knowing anything about the delegated computation, so it cannot analyze the computation and attempt to determine if it is malicious. This tension can be phrased as the following key question:

How can we expect cloud services to prevent misuse of their platform
if they cannot "look" at the delegated computation?

This work proposes a rigorous definition of *responsible* delegation, in which the cloud server does not leak any information about computations deemed to be malicious. We then answer the above key question by showing how to generically compile any publicly-verifiable[1] delegation protocol into a *responsible and private*

[1] To stay focused on the key notions of *responsibility* and *privacy*, we leave the discussion of verifiability to later. Briefly put, verifiability (or soundness) means that the user can verify that the server executed the computation correctly. Public verifiability means that anyone can verify the computation, not just the user who delegated it.

delegation protocol, using strong but standard cryptographic tools. Our focus is on one-round verifiable delegation protocols for arbitrary polynomial-time computations.

1.1 Our contributions in a nutshell

Before diving in, we give an informal summary of our two main contributions: i) a definition of responsible delegation of computation, and ii) a compiler that transforms any publicly-verifiable delegation protocol into a responsible private delegation protocol.

In this work we consider a computationally weak user and a (relatively) powerful server. The user wants to learn the outcome of a computation outside of its resource limitations, so it delegates it to the server. The server responds with the (alleged) outcome of the computation, and the user then either *accepts* the server's response or *rejects* it. Since the user does not trust the server to execute the computation correctly, it engages in a *verifiable* delegation protocol, meaning that no malicious server can convince the user to accept a false statement. Notice that this informal description includes only one message from the user to the server, and one message from the server back to the user. We refer to such a protocol as one-round.

Remark 1. For the remainder of this paper, a *delegation protocol* refers to a one-round verifiable delegation protocol for arbitrary polynomial-time computation.

Of particular interest are *private* delegation protocols, in which the server learns nothing about the delegated computation while interacting with the user. On the other hand, a *responsible* delegation platform is one which actively attempts to prevent misuse of its services. In this work, *malicious* is defined by the service provider, which can (and should) incorporate regulations mandated by the state in its terms.[2] We restrict the discussion to malicious usage that can be identified by an automated *misuse-detection program*, as alluded to by the Amazon representative in the opening paragraph of this paper.

We model a misuse-detection program as a program D that takes as input the delegated program M and input x: if $D(M, x) = 1$ then computation $M(x)$ is considered malicious by the server, and if $D(M, x) = 0$ then the computation is not malicious. A misuse-detection program can be anything from a simple check against a database of known malicious programs to a more intelligent analysis of the code of the delegated program and its input. The point is that a responsible server may prevent misuse as defined by *any* given misuse-detection program.

Definition 1 (Responsible delegation protocol, informally). *A delegation protocol* ($\mathsf{Server}, \mathsf{User}$) *is* responsible *if, in addition to verifiability, it satisfies the following property:*

- Misuse-prevention: *For any misuse-detection program D given to* Server*,* User *learns* nothing *but the results of computations (M, x) such that $D(M, x) = 0$, after interacting with* Server*.*

Note that the user may learn the results of other computations so long as they are not flagged as malicious. A detailed discussion of this definition can be found in Section 2.1, and its formalization awaits in Section 3.1. For now, it would be helpful to point out the distinction between the *misuse-detection* program (that detects malicious computation), and the property of *misuse-prevention* (that guarantees that the server is able to prevent misuse, for a given misuse-detection tool).

In this work we also show that Definition 1 is attainable: we show how to transform any publicly-verifiable delegation protocol into a *responsible and private* delegation protocol. Public verifiability means that anyone can verify the computation, not just the user; if verification requires a secret known only to the user, then the protocol is privately verifiable. Indeed, any of the known publicly-verifiable delegation protocols (for example, [19, 9, 30, 4, 8, 25]) may be used by our compiler.

Theorem 1 (Main theorem, informally). *Assuming Fully Homomorphic Encryption and Non-interactive Zero Knowledge Arguments of Knowledge, there exists a compiler that takes any publicly-verifiable delegation protocol, and produces a* responsible and private *delegation protocol of comparable user and server efficiency.*

We remark that the resulting protocol is *privately verifiable* (by necessity, as discussed in Section 2.2) and in the Common Reference String (CRS) model.

An overview of the proof is in Section 2.2. A formal statement of the theorem is given in Section 3.

[2] The fascinating legal and regulatory discussion of what constitutes *malicious* computation is outside the scope of this work.

1.2 Example applications

Next are two motivating examples of how our definition and compiler may be of use. The first is a general example of how a server can carry any existing misuse-detection tool over to the private setting using our protocol. The second considers a specific application: prevention of non-consensual Deepfakes delegated to a specialized service. This specific application highlights a scenario in which *both* responsibility and privacy are needed.

General application: privacy-preserving misuse-prevention Let us give a more concrete idea of how the protocol described in Theorem 1 allows for responsible delegation without sacrificing privacy. This example shows that even when the server cannot directly examine the delegated computation, it can just as well prevent malicious use of its platform via existing misuse-detection tools. The exact detection tool is purposefully left unspecified – the point is that our protocol works regardless of the specific methods used for misuse-detection.

Consider a fictional cloud computing service *egCloud* that uses a certain tool, denoted egMis, to detect misuse of its platform: whenever a user delegates program M and input x, the *egCloud* server first checks that $\mathsf{egMis}(M, x)$ is not flagged as malicious, and only then proceeds with its delegation protocol for $M(x)$. An example of misuse may be using the *egCloud* platform for password cracking, as done in [36].

egCloud now wish to expand their service into the realm of *private* delegation and allow users to delegate computation that contains sensitive information not to be shared with *egCloud*. But, importantly, they also want to remain a *responsible* delegation service that takes active measures to prevent misuse of its platform.

To achieve these goals, *egCloud* take their current publicly-verifiable protocol and compile it using Theorem 1, with misuse-detection program D in the protocol chosen to be their existing tool egMis. *egCloud* can now offer a delegation service with the following features:

– *Verifiability: egCloud* can prove to the user that they correctly carried out the delegated computation, if it was not flagged for misuse.
– *Privacy: egCloud* never learns anything about the delegated computation. Note that this is true even if the delegated computation was flagged for misuse, which might be seen as a bug rather than a feature; it could be desirable to alert *egCloud* when a certain user attempted to misuse their platform. However, in this initial work we decided to guarantee absolute privacy.[3]
– *Responsibility:* Just as in the non-private setting, *egCloud* could use their misuse-detection tool egMis to prevent misuse of their platform. In fact, they are free to update the detection tool after the initial setup of the protocol (by updating the D argument to $\mathsf{Server}(\cdot)$). This is especially useful if we think of the misuse-detection tool egMis as checking against a database of malicious programs, which may be updated daily.

Specialized application: prevention of non-consensual Deepfakes Deepfake is a video forgery technique that uses deep learning to place the face of one person on the body of another. The remarkable quality of Deepfaked videos (aka Deepfakes) enables a "disturbing array of malicious use" [10], among them are non-consensual pornography, political misinformation, and fraud [1]. While Deepfaking does have beneficial uses (e.g. in art and education), the already-realized potential for societal harm calls for special care by those providing access to this tool [10].

This example considers services offering delegation of Deepfake computation such as FaceApp and Zao [32, 33]. For simplicity, we model Deepfake delegation as follows:

– The user sends a *source video* and a *target video*, containing a *source person* and *target person* (respectively).
– The server Deepfakes the face of the source person onto the body of the target person and sends the resulting Deepfaked video to the user.

[3] One concern is that if privacy is not absolute, and *egCloud* could identify malicious users, then *egCloud* could also use a misuse-detection circuit of their choice to learn any arbitrary predicate on the delegated computation.

We consider the fictional platform *egFake* that offers its users delegation of Deepfake computation. Given the great power that Deepfakes hold, *egFake* make the *responsible* choice of actively preventing the creation of harmful Deepfakes on its platform. Specifically, they wish to prevent non-consensual Deepfakes, in which either the source or the target person did not provide their consent to have their likeness manipulated.[4] Such a mechanism can be realized if *egFake* has access to a database relating images of people with their public key, and having *egFake* process only videos signed by the people appearing in them (using facial recognition to identify the person in each video).

On the other hand, existing Deepfake platforms are provoking concerns for privacy of its users [7, 33]: what prevents the delegation platform from reusing or selling the source and target videos? Despite its fictitiousness, *egFake* is not exempt from these concerns, and wants to offer *private* delegation so that it learns nothing about the source or target videos.

Using the delegation protocol from Theorem 1, *egFake* ensure the privacy of their users without settling on their societal responsibilities. Despite not being able to examine their users' videos, their service will never create non-consensual Deepfakes.

2 Technical Overview

Next is a discussion motivating our definition of responsible delegation, and an overview of how any publicly-verifiable delegation scheme can be transformed into a responsible and private delegation scheme using standard cryptographic tools.

2.1 Defining responsible delegation

Delegation of computation is a protocol between a weak user and a stronger server. The user would like to execute the computation of program M on input x where $M(x)$ runs in time T, which is beyond the user's computational resources but within the server's. We formalize this by having the user run in time polynomial in $(|M|, |x|, \log(T))$, while allowing the server to run in time polynomial in $(|M|, |x|, T)$. Crucially, the user's run time only depends logarithmically on T. This imbalance will play a crucial role in the definition of misuse-prevention.

A responsible delegation protocol is one in which misuse, as defined by a *misuse-detection program* D, is prevented by the server. The most straightforward requirement would be to have a responsible delegation server provide no information about a delegated computation that is flagged by D: when a user delegates a computation (M, x) such that $D(M, x) = 1$, it learns nothing from the response of the server. However, a responsible server should also ensure that the user learns nothing about *other* computations except for their output: even if $D(M, x) = 0$, the user learns nothing except $M(x)$.[5]

"Learning nothing" is modelled by the simulation paradigm of Goldwasser *et al.* [20]: a user *learns nothing* if they cannot distinguish between interaction with the real server, and interaction with a simulator. Our setting may be modelled as a two-party computation between a user and a server, where the user has input (M, x), the server has input D, and after interaction, the user learns $M(x)$ if $D(x) = 0$, and nothing otherwise.

Note that a standard application of simulation-based security for multi-party computation does not quite make sense here, since the server does not have any secret input that must be hidden from the user. In particular, all computations submitted by the user are polynomial time, so a polynomial time simulator could simply run the honest server.

We model the simulator as a machine that runs in some *fixed* polynomial time $p(\cdot)$ in its input (received from the user), but has oracle access to an ideal functionality $\mathcal{F}_D$ that on input (M, x) returns $M(x)$ if $D(x) = 0$ and $\perp$ otherwise. In particular, this allows the simulator to simulate proofs for computations with *arbitrary* polynomial run time, so long as they are not flagged for misuse. At the same time, indistinguishability between

[4] Non-consensual pornography accounted for 96% of Deepfaked videos on the internet in 2019 [1].

[5] At first glance, it may appear that this condition is too stringent. After all, if the computation is not flagged by D, shouldn't we allow the server to use any delegation protocol it likes to respond to the user? But this condition is vital: consider the case that the server's response on input (M, x) also leaks the outcome of $M'(x')$ for a related *but malicious* computation (M', x').

the real world and simulated world ensures that the user cannot gain any additional information from the interaction that cannot be computed in a some *fixed* polynomial time.

This sums up the main points addressed by our definition of responsible delegation. Before moving on, we highlight four additional points about our definition.

- Our definition takes into account *malicious users* that do not necessarily follow the honest protocol when generating messages to the server. In particular, the user may send messages that do not correspond to any valid computation, and the simulator must be able to account for this possibility.
- Our definition models a user that interacts arbitrarily many times with the server. In particular, it ensures that a user cannot learn any extra information about a particular computation via interaction with the server on another computation in the future.
- Moreover, we allow the server to arbitrarily update its misuse-detection circuit D over time. That is, we actually define the ideal functionality with respect to a sequence of circuits $\mathbb{D} = (D_1, \ldots, D_m)$ where the i'th circuit is used by the server to respond to the i'th message sent by the user. In the security definition, we quantify over all such sequences.
- The user does not need to know the misuse-detection program D,[6] nor does its runtime depend on the runtime of misuse-detection. This is desirable so as not to burden the (weak) user with computing a potentially complicated misuse-detection procedure D.

Interestingly, the server *prevents* malicious computation without *detecting* when a user submitted such a computation. Detection could be seen as desirable to allow the server to ban repeat offenders from using the platform, or avoid wasting resources to perform a (malicious) computation whose results will not be seen by the user. But detection is fundamentally at odds with privacy, since the server could then pick any D of its choice in order to learn an arbitrary predicate on the user's computation. Since the prime directive of this work is responsible computing *with user privacy*, we chose absolute privacy over detection.

2.2 Adding misuse-prevention and privacy

Next is an overview of our compiler, which takes any publicly-verifiable (see below) delegation protocol Φ for Turing machines,[7] and produces a responsible private delegation protocol Φ'.

Public verifiability stipulates that any user that delegates a computation (M, x) to the server may also produce a public verification key vk, allowing anybody to obtain the result of $M(x)$ from the server's response, and verify that the computation was executed correctly. While this notion is not necessarily at odds with privacy in the standard setting (i.e., with no misuse-prevention),[8] public-verifiability contradicts privacy for responsible delegation schemes. This is because being able to publicly verify the server's message implies the ability to detect whether the user's input satisfied D or not, and as mentioned earlier, this allows the the server to learn any predicate it wants on the user's computation. Thus, our compiler *must* output a delegation scheme that is *privately* verifiable.

Now we describe our compiler, which is based on a non-interactive zero-knowledge argument of knowledge NIZK and a fully homomorphic encryption scheme *FHE*. The publicly-verifiable delegation scheme Φ (to be compiled into a responsible and private delegation scheme) is specified by four algorithms ($\Phi.\mathsf{Setup}$, $\Phi.\mathsf{User.Del}$, $\Phi.\mathsf{Server}$, and $\Phi.\mathsf{User.Ver}$). A delegation of the computation (M, x) using this scheme goes as follows:

1. The algorithm $\Phi.\mathsf{Setup}$ produces a public proving key pk and a public delegation key dk.
2. The user computes a delegation message and verification key $(\mathsf{msg}, \mathsf{vk}) \leftarrow \Phi.\mathsf{User.Del}(\mathsf{dk}, M, x)$.
3. The server executes the computation sent by the user, and computes a proof that it has done so $\pi \leftarrow \Phi.\mathsf{Server}(\mathsf{pk}, \mathsf{msg})$.
4. Finally, the user verifies the proof sent by the server $\Phi.\mathsf{User.Ver}(\mathsf{vk}, \pi)$ and either accepts or rejects.

[6] That said, the misuse-detection procedure is *not* necessarily kept secret from the user: the user obviously learns information about it from the outcome of the delegation protocol.

[7] Although we present the compiler for Turing machine delegation, it would work just as well for RAM program delegation or circuit delegation in the pre-processing setting, see Appendix B.1.

[8] Indeed, there are works that achieve publicly-verifiable delegation with user privacy for various computational models [19, 30, 8, 2].

So, how is Φ compiled into a responsible private scheme? First, we need a mechanism for the server to i) compute π based on the user's msg, and ii) compute any given misuse-detection program D on the user's input (M, x), without revealing anything about (M, x) to the server. Furthermore, we need the result of this computation to determine whether the server responds with π, or a string containing no information. This is straightforward to implement with *FHE*.

1. The user encrypts the the message $\mathsf{ct}_{\mathsf{msg}} \leftarrow \mathit{FHE}.\mathsf{Enc}(\mathsf{msg})$ and computation $\mathsf{ct}_{M,x} \leftarrow \mathit{FHE}.\mathsf{Enc}(M, x)$.
2. The server constructs the circuit $\mathsf{C}[\Phi.\mathsf{pk}, D]$ that takes as input a user message msg and a description of a computation (M, x) and outputs $\Phi.\mathsf{Server}(\mathsf{msg})$ if $D(M, x) = 0$, and $\perp$ if $D(M, x) = 1$ (signaling that the computation is flagged for misuse).
   The server then "responsibly executes" the delegated computation under the FHE by computing $\mathsf{ct}_\pi \coloneqq \mathit{FHE}.\mathsf{Eval}(\mathsf{C}[\Phi.\mathsf{pk}, D], (\mathsf{ct}_{\mathsf{msg}}, \mathsf{ct}_{M,x}))$ and sending it to the user.
3. The user decrypts $\pi' \coloneqq \mathit{FHE}.\mathsf{Dec}(\mathsf{ct}_\pi)$ and verifies $\Phi.\mathsf{User}.\mathsf{Ver}(\pi')$.

Note that if the computation was flagged for misuse then $\pi' = \perp$, but otherwise if the computation was not flagged (and the protocol was executed correctly by both the user and the server), the user should decrypt a valid proof π and therefore accept.

However, this basic scheme would not quite satisfy our notion of responsibility since, as discussed earlier, the server's proof π may itself leak arbitrary information about (M, x) or related computations.[9] To address this, we have the server, instead of sending π, compute and send a zero-knowledge proof that it knows some π that would make $\Phi.\mathsf{User}$ accept. This is where we use the assumption that that the original protocol Φ is publicly-verifiable.

Lastly, we need to ensure that even a malicious user who does not follow the specifications of the protocol cannot gain any additional information. This is done in a standard way by requiring the user to provide a zero-knowledge proof that it generated its message honestly. In particular, it is crucial that $\Phi.\mathsf{msg}$ and ct are generated with respect to the same computation (M, x). To argue that the resulting scheme satisfies our simulation-based definition, we actually make use of a non-interactive zero-knowledge *argument of knowledge*, enabling the simulator to extract the user's effective input (M, x), query its ideal functionality, and return either an encryption of a simulated proof, or an encryption of $\perp$.

3 Responsible Private Delegation of Computation

In this section, we present our definition of responsible private delegation of computation (Section 3.1), and a compiler that adds misuse-prevention and privacy to any publicly-verifiable delegation scheme.

3.1 Definitions

Without further ado, we present the formal definition of responsible private delegation of computation. It consists of the standard notions of completeness, soundness, and privacy, plus our new definition of *misuse-prevention.*

The reader is referred to Appendix A.1 for preliminary cryptographic notions used in our definitions, namely negligible functions, probabilistic polynomial time (PPT) machines, and oracle machines.

The definition is given for Turing machines, modeled with a tuple $z = (M, x, T)$, where M is the description of a Turing machine, x is an input, and T is a time bound. Let $\mathcal{U}(\cdot)$ be a *universal* Turing machine, so that $\mathcal{U}((M, x, T))$ outputs 1 if $M(x)$ accepts within T steps, and outputs 0 otherwise.

First, a definition of a *private delegation scheme* which satisfies the standard notions of completeness, soundness and privacy from the literature (see, for example, [4]).

Definition 2 (Private delegation scheme). *A private delegation scheme describes a protocol run between a user and a server. It consists of four algorithms* $(\mathsf{Setup}, \mathsf{User}.\mathsf{Del}, \mathsf{Server}, \mathsf{User}.\mathsf{Ver})$ *that satisfy four requirements:* efficiency*,* completeness*,* soundness *and* input-privacy*. The algorithms and requirements are as follows:*

[9] Certain delegation schemes in the literature [19, 30] actually already have the property that the proof is independent of the computation itself. However, we aim for full generality, and do not want to assume anything extra of the underlying delegation protocol.

– $\mathsf{Setup}(1^\lambda, 1^n) \to (\mathsf{pk}, \mathsf{dk})$: *The setup algorithm takes as input a security parameter λ and a maximum computation description length n, and outputs a proving key* pk *to be used by the server, and a delegation key* dk *to be used by the user.*
– $\mathsf{User.Del}(\mathsf{dk}, z) \to (\mathsf{msg}, \mathsf{vk})$: *The user takes as input a tuple $z = (M, x, T)$ and uses the delegation key to produce a message* msg *and a verification key* vk*.*
– $\mathsf{Server}(\mathsf{pk}, \mathsf{msg}, D) \to \pi$: *The server takes as input a message* msg *and a misuse-detection circuit D, and uses the proving key* pk *to produce a proof π.*
– $\mathsf{User.Ver}(\mathsf{vk}, \pi) \to y$: *The user takes a proof π as input and uses the verification key* vk *to output a string $y \in \{0, 1\}$.*

Efficiency *There exists a fixed polynomial $t(\cdot)$ such that for every $z = (M, x, T)$ with $|z| \le n$,* $\mathsf{Setup}, \mathsf{User.Del},$ *and* $\mathsf{User.Ver}$ *run in time $t(\lambda, n)$, and* Server *runs in time $t(\lambda, |M|, |x|, T, |D|)$.*

Completeness *For any tuple $z = (M, x, T)$ with $|z| \le n$ such that $\mathcal{U}(z) = 1$, and circuit* D *such that $\mathsf{D}(z) = 0$,*[10]

$$\Pr\left[1 \leftarrow \mathsf{User.Ver}(\mathsf{vk}, \pi) : \begin{array}{r} (\mathsf{pk}, \mathsf{dk}) \leftarrow \mathsf{Setup}(1^\lambda, 1^n) \\ (\mathsf{msg}, \mathsf{vk}) \leftarrow \mathsf{User.Del}(\mathsf{dk}, z) \\ \pi \leftarrow \mathsf{Server}(\mathsf{pk}, \mathsf{msg}, D) \end{array}\right] = 1.$$

Soundness *For any tuple $z = (\mathsf{M}, x, T)$ with $|z| \le n$ such that $\mathcal{U}(z) = 0$, and PPT adversary $\mathcal{A}$,*

$$\Pr\left[1 \leftarrow \mathsf{User.Ver}(\mathsf{vk}, \pi^*) : \begin{array}{r} (\mathsf{pk}, \mathsf{dk}) \leftarrow \mathsf{Setup}(1^\lambda, 1^n) \\ (\mathsf{msg}, \mathsf{vk}) \leftarrow \mathsf{User.Del}(\mathsf{dk}, z) \\ \pi^* \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{msg}) \end{array}\right] = \mathsf{negl}(\lambda).$$

Privacy *For any PPT adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ and time bound T,*[11]

$$\Pr\left[\mathcal{A}_1^{\mathsf{User.Del}(\mathsf{dk}, \cdot)}(\mathsf{pk}, \mathsf{dk}, \mathsf{st}, \mathsf{msg}) = b : \begin{array}{r} (\mathsf{pk}, \mathsf{dk}) \leftarrow \mathsf{Setup}(1^\lambda, 1^n) \\ ((M_0, x_0), (M_1, x_1), \mathsf{st}) \leftarrow \mathcal{A}_0(\mathsf{pk}, \mathsf{dk}) \\ b \leftarrow \{0, 1\} \\ (\mathsf{msg}, \mathsf{vk}) \leftarrow \mathsf{User.Del}(\mathsf{dk}, (M_b, x_b, T)) \end{array}\right] \le \frac{1}{2} + \mathsf{negl}(\lambda).$$

We consider *publicly delegatable* schemes, so we assume that the delegation key dk is public (see Remark 3 for more discussion). If soundness of the scheme requires the verification key vk be kept secret by the user then the scheme is *privately verifiable.* If the scheme is sound even if vk is public (and in particular, known by the untrusted server) it is *publicly verifiable.*

Finally, the formal definition of a *responsible private delegation scheme* follows.

Definition 3 (Responsible private delegation scheme). *A responsible private delegation scheme is one that, in addition to satisfying Definition 2, satisfies the following additional requirement:*

Misuse-Prevention: There exists a fixed polynomial $p(\cdot)$, a simulator $\mathcal{S} := (\mathcal{S}_0, \mathcal{S}_1)$ such that $\mathcal{S}_0, \mathcal{S}_1$ both run in time $p(\cdot)$ in their input, and a negligible function $\mathsf{negl}(\cdot)$*, such that for any description length n, repetitions m, PPT adversary $\mathcal{A}$, and any sequence of circuits $\mathbb{D} := (D_1, \ldots, D_m)$ with inputs in $\{0, 1\}^n$,*

$$\left|\Pr\left[\mathsf{Exp}^{\mathsf{REAL}}_{\mathcal{A}, \mathsf{Setup}, \mathsf{Server}}(1^\lambda, \mathbb{D}) = 1\right] - \Pr\left[\mathsf{Exp}^{\mathsf{IDEAL}}_{\mathcal{A}, \mathcal{S}_0, \mathcal{S}_1}(1^\lambda, \mathbb{D}) = 1\right]\right| = \mathsf{negl}(\lambda).$$

[10] Observe that we only require the server to convince the user of claims (M, x, T) such that $\mathsf{M}(x) = 1$ within T steps. In particular, we don't require the server to convince the user of the fact that M outputs 0 on x within T steps. However, the user can always use the server to learn the result of $\mathsf{M}(x)$ by delegating both M and $1 - \mathsf{M}$.

[11] Other works, for example [4], take privacy to mean that the string x on which M is being run is hidden. This is equivalent to our definition, which requires the computation (M, x) to be hidden, since one could always delegate the universal Turing machine, which takes (M, x, T) as input.

```
$\mathsf{Exp}^{\mathsf{REAL}}_{\mathcal{A},\mathsf{Setup},\mathsf{Server}}(1^\lambda, \mathbb{D})$

$(\mathsf{pk}, \mathsf{dk}) \leftarrow \mathsf{Setup}(1^\lambda, 1^n)$
$i := 0$
return $\mathcal{A}^{\mathcal{O}_{\mathsf{Server},\mathbb{D}}(\mathsf{pk},\cdot)}(1^\lambda, \mathsf{pk}, \mathsf{dk})$

$\underline{\mathcal{O}_{\mathsf{Server},\mathbb{D}}(\mathsf{pk}, \mathsf{msg})}$
$i := i + 1$
parse $\mathbb{D} = (D_1, \ldots, D_m)$
if $i \leq m$, return $\mathsf{Server}(\mathsf{pk}, \mathsf{msg}, D_i)$
else return $\bot$
```

```
$\mathsf{Exp}^{\mathsf{IDEAL}}_{\mathcal{A},\mathcal{S}_0,\mathcal{S}_1}(1^\lambda, \mathbb{D})$

$(\mathsf{pk}, \mathsf{dk}, \mathsf{st}) \leftarrow \mathcal{S}_0(1^\lambda, 1^n)$
$i := 0$
return $\mathcal{A}^{\mathcal{O}_{\mathcal{S}_1,\mathbb{D}}(\mathsf{pk},\cdot)}(1^\lambda, \mathsf{pk}, \mathsf{dk})$

$\underline{\mathcal{O}_{\mathcal{S}_1,\mathbb{D}}(\mathsf{pk}, \mathsf{msg})}$
return $\mathcal{S}_1^{\mathcal{F}_{\mathbb{D}}(\cdot)}(\mathsf{st}, \mathsf{msg})$

$\underline{\mathcal{F}_{\mathbb{D}}(z)}$
$i := i + 1$
parse $\mathbb{D} = (D_1, \ldots, D_m)$
if $i \leq m$:
  if $D_i(z) = 0$, return $\mathcal{U}(z)$
  else return $0$
else return $\bot$
```

Note that the definition includes the existence of a simulator with a *bounded* polynomial run-time. Interest in bounding the run-time of simulation is reminiscent of the notion of knowledge tightness ([18]), though in our particular setting (in contrast with traditional knowledge-tight zero-knowledge) we have the additional requirement of succinct communication. Note also that considering *any* fixed polynomial $p(\cdot)$ below gives a meaningful definition, since the user may be delegating computations of *unbounded* polynomial size.

3.2 Compiler

In this section, we show how to generically add misuse-prevention and privacy to any delegation scheme that is complete, sound, and publicly-verifiable, using standard cryptographic tools.

Theorem 2. *Assuming a NIZK argument of knowledge for NP, and a rerandomizable FHE scheme, any publicly-verifiable delegation scheme Φ can be compiled into a responsible and private delegation scheme Π, with a fixed polynomial blowup in the runtime of the server and the user.*

See Appendix B.1 for some context regarding the assumptions made in Theorem 2.

Next, we present the compiler asserted in Theorem 2. Proof of its soundness, privacy, and misuse-prevention are deferred to Appendix B.2.

3.3 The compiler of Theorem 2

We now give a detailed description of the compiler asserted in Theorem 2 (recall its overview in Section 2.2). The reader is referred to Appendices A.2 and A.3 for definitions of fully homomorphic encryption and non-interctive zero knowledge.

Our compiler takes a publicly-verifiable delegation scheme

$$\Phi := (\Phi.\mathsf{Setup}, \Phi.\mathsf{User.Del}, \Phi.\mathsf{Server}, \Phi.\mathsf{User.Ver}),$$

and outputs a responsible private delegation scheme

$$\Pi := (\Pi.\mathsf{Setup}, \Pi.\mathsf{User.Del}, \Pi.\mathsf{Server}, \Pi.\mathsf{User.Ver}).$$

Notation

- λ denotes the security parameter, and n denotes a bound on the description length of computation to be delegated.
- p_{User} is the fixed polynomial such that $|\mathsf{CheckUser}| = p_{\mathsf{User}}(\lambda, n)$ (see Fig. 1).
- p_{Server} is the fixed polynomial such that $|\Phi.\mathsf{User.Ver}| = p_{\mathsf{Server}}(\lambda, n)$.
- $\mathcal{U}$ is the universal Turing machine: $\mathcal{U}(M, x, T)$ outputs 1 if $M(x)$ accepts within T steps. In the construction, $\mathcal{U}$ will be computed on data encrypted under the FHE. This requires the server to evaluate $\mathcal{U}$ as a circuit, and thus it will have to know an upper bound on T. Without being explicit in the construction below, we assume that the user provides T to the server as part of its message.

The construction

- $\mathit{\Pi}.\mathsf{Setup}(1^\lambda, 1^n)$:
  1. Compute the proving and delegation keys of the original scheme,
  $$(\mathit{\Phi}.\mathsf{pk}, \mathit{\Phi}.\mathsf{dk}) \leftarrow \mathit{\Phi}.\mathsf{Setup}(1^\lambda, 1^n).$$
  2. Let $n_{\mathsf{User}} \coloneqq p_{\mathsf{User}}(\lambda, n)$ and $n_{\mathsf{Server}} \coloneqq p_{\mathsf{Server}}(\lambda, n)$.
  3. Compute the common reference strings
  $$crs_{\mathsf{User}} \leftarrow \mathsf{NIZK.Setup}(1^\lambda, 1^{n_{\mathsf{User}}}) \text{ and } crs_{\mathsf{Server}} \leftarrow \mathsf{NIZK.Setup}(1^\lambda, 1^{n_{\mathsf{Server}}}).$$
  4. Output the proving and delegation keys
  $$\mathit{\Pi}.\mathsf{pk} \coloneqq (\mathit{\Phi}.\mathsf{pk}, crs_{\mathsf{User}}, crs_{\mathsf{Server}}) \text{ and } \mathit{\Pi}.\mathsf{dk} \coloneqq (\mathit{\Phi}.\mathsf{dk}, crs_{\mathsf{User}}, crs_{\mathsf{Server}}).$$
- $\mathit{\Pi}.\mathsf{User.Del}(\mathit{\Pi}.\mathsf{dk}, z)$:
  1. Parse $\mathit{\Pi}.\mathsf{dk}$ as $(\mathit{\Phi}.\mathsf{dk}, crs_{\mathsf{User}}, crs_{\mathsf{Server}})$.
  2. Generate $(\mathit{FHE}.\mathsf{pk}, \mathit{FHE}.\mathsf{sk}) \leftarrow \mathit{FHE}.\mathsf{KeyGen}(1^\lambda)$.
  3. Compute the user's message in the original protocol $\mathit{\Phi}.\mathsf{msg} \leftarrow \mathit{\Phi}.\mathsf{User.Del}(\mathit{\Phi}.\mathsf{dk}, z)$.
  4. Encrypt the user's message $\mathsf{ct}_{\mathsf{msg}} \leftarrow \mathit{FHE}.\mathsf{Enc}(\mathit{FHE}.\mathsf{pk}, \mathit{\Phi}.\mathsf{msg})$.
  5. Encrypt the computation to be delegated $\mathsf{ct}_z \leftarrow \mathit{FHE}.\mathsf{Enc}(\mathit{FHE}.\mathsf{pk}, z)$.
  6. Compute a proof that the above steps were executed correctly, where correct execution is described by $\mathsf{CheckUser}$ in Figure 1:
  $$\pi_{\mathsf{User}} \leftarrow \mathsf{NIZK.Prove}\left(\begin{array}{l} crs_{\mathsf{User}}, \mathsf{CheckUser}[\mathit{\Phi}.\mathsf{dk}, \mathit{FHE}.\mathsf{pk}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{ct}_z], \\ (\mathit{\Phi}.\mathsf{msg}, z, r_1, r_2, r_3, r_4) \end{array}\right),$$
  and where r_i were the random coins used in step $i+1$.
  7. Output message $\mathit{\Pi}.\mathsf{msg} \coloneqq (\mathit{FHE}.\mathsf{pk}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{ct}_z, \pi_{\mathsf{User}})$ and verification key $\mathit{\Pi}.\mathsf{vk} \coloneqq (crs_{\mathsf{Server}}, \mathit{FHE}.\mathsf{sk})$.
- $\mathit{\Pi}.\mathsf{Server}(\mathit{\Pi}.\mathsf{pk}, \mathit{\Pi}.\mathsf{msg}, D)$:
  1. Parse $\mathit{\Pi}.\mathsf{pk}$ as $(\mathit{\Phi}.\mathsf{pk}, crs_{\mathsf{User}}, crs_{\mathsf{Server}})$ and $\mathit{\Pi}.\mathsf{msg}$ as $(\mathit{FHE}.\mathsf{pk}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{ct}_z, \pi_{\mathsf{User}})$.
  2. Assert that the user generated their message correctly: if
  $$\mathsf{NIZK.Ver}(crs_{\mathsf{User}}, \mathsf{CheckUser}[\mathit{\Phi}.\mathsf{dk}, \mathit{FHE}.\mathsf{pk}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{ct}_z], \pi_{\mathsf{User}}) = 0,$$
  halt and output $\perp$.
  3. "Under the FHE", check that the delegated computation is not malicious, and if so execute it and prove that the execution is correct (this procedure is captured by $\mathsf{ServerEval}$ Figure 2):
  $$\mathsf{ct}_\pi \coloneqq \mathit{FHE}.\mathsf{Eval}(\mathit{FHE}.\mathsf{pk}, \mathsf{ServerEval}[\mathit{\Phi}.\mathsf{pk}, D, crs_{\mathsf{Server}}], (\mathsf{ct}_{\mathsf{msg}}, \mathsf{ct}_z)).$$
  4. Rerandomize the ciphertext of the above proof $\mathit{\Pi}.\pi \coloneqq \mathit{FHE}.\mathsf{Rerand}(\mathsf{ct}_\pi)$. Output $\mathit{\Pi}.\pi$.
- $\mathit{\Pi}.\mathsf{User.Ver}(\mathit{\Pi}.\mathsf{vk}, \mathit{\Pi}.\pi)$:
  1. Parse $\mathit{\Pi}.\mathsf{vk}$ as $(crs_{\mathsf{Server}}, \mathit{FHE}.\mathsf{sk})$.
  2. If $\mathit{\Pi}.\pi = \perp$, output 0.
  3. Decrypt the server's proof $\pi_{\mathsf{Server}} \leftarrow \mathit{FHE}.\mathsf{Dec}(\mathit{FHE}.\mathsf{sk}, \pi)$.
  4. If $\pi_{\mathsf{Server}} = \perp$, output 0.
  5. Verify the proof: if $\mathsf{NIZK.Ver}(crs_{\mathsf{Server}}, \mathit{\Phi}.\mathsf{User.Ver}, \pi_{\mathsf{Server}}) = 1$, output 1, otherwise, output 0.

```
CheckUser

Input. $\Phi.\mathsf{msg}, z, r_1, r_2, r_3, r_4$.
Hardcoded. $\Phi.\mathsf{dk}, \mathit{FHE}.\mathsf{pk}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{ct}_z$.

If the following checks pass, output 1, otherwise output 0.

 1. Check $(\mathit{FHE}.\mathsf{pk}, \mathit{FHE}.\mathsf{sk}) = \mathit{FHE}.\mathsf{KeyGen}(1^\lambda; r_1)$.
 2. Compute $\Phi.\mathsf{msg} = \Phi.\mathsf{User}.\mathsf{Del}(\Phi.\mathsf{dk}, z; r_2)$ and check that
        $\mathsf{ct}_{\mathsf{msg}} = \mathit{FHE}.\mathsf{Enc}(\mathit{FHE}.\mathsf{pk}, \Phi.\mathsf{msg}; r_3)$.
 3. Check $\mathsf{ct}_z = \mathit{FHE}.\mathsf{Enc}(\mathit{FHE}.\mathsf{pk}, z; r_4)$.
```

Fig. 1: The program CheckUser.

```
ServerEval

Input. $\Phi.\mathsf{msg}, z$.
Hardcoded. $\Phi.\mathsf{pk}, D, crs_{\mathsf{Server}}$.

 1. Compute $y := \mathcal{U}(z) \wedge D(z)$. (Recall that $\mathcal{U}(\cdot)$ is defined to output a bit 0 or 1.)
 2. If $y = 0$, output $\perp$ (appropriately padded).
 3. Otherwise,
    (a) Compute $\Phi.\pi \leftarrow \Phi.\mathsf{Server}(\Phi.\mathsf{pk}, \Phi.\mathsf{msg})$.
    (b) Output $\pi_{\mathsf{Server}} \leftarrow \mathsf{NIZK}.\mathsf{Prove}(crs_{\mathsf{Server}}, \Phi.\mathsf{User}.\mathsf{Ver}, \Phi.\pi)$.
```

Fig. 2: The program ServerEval.

4 Conclusion and Future Work

In this work we presented a rigorous framework modelling an important real-world concern in delegation of computation: misuse-prevention and its tension with user privacy. Our primary contribution is in identifying this issue and providing initial definitions and constructive results, but there is much more to be explored. Here are a few possible directions for future research:

– Providing a definition for responsible delegation of computation for *interactive* delegation protocols; the definitions in Section 3.1 consider only one-round protocols.
– Compiling *any* delegation scheme into a responsible and private delegation scheme; Theorem 2 requires the initial scheme to be publicly verifiable.
– Constructing responsible private delegation schemes from weaker cryptographic assumptions. These can be non-generic, as in a white-box modification of an existing delegation scheme rather than a black-box compiler. For example, a long line of work constructs privately-verifiable delegation protocols from standard assumptions such as computational private information retrieval [26, 27, 24, 5, 3]. While our compiler does not apply to these schemes due to private verifiability, one could hope to tweak the schemes themselves in order to achieve responsibility. In particular, these schemes make use of a no-signaling PCP to prove the correctness of computations performed by the server. Perhaps requiring the PCP to also be *zero-knowledge* (in addition to having the server run the misuse-detection circuit on the encrypted computation) would suffice to satisfy our definition of responsibility.

References

1. Ajder, H., Patrini, G., Cavalli, F., Cullen, L.: The state of Deepfakes: Landscape, threats and impact. Tech. rep., Deeptrace Labs (Sep 2019)
2. Ananth, P., Lombardi, A.: Succinct garbling schemes from functional encryption through a local simulation paradigm. In: Beimel, A., Dziembowski, S. (eds.) TCC 2018, Part II. LNCS, vol. 11240, pp. 455–472. Springer, Heidelberg (Nov 2018). https://doi.org/10.1007/978-3-030-03810-6_17
3. Badrinarayanan, S., Kalai, Y.T., Khurana, D., Sahai, A., Wichs, D.: Succinct delegation for low-space non-deterministic computation. In: Diakonikolas, I., Kempe, D., Henzinger, M. (eds.) 50th ACM STOC. pp. 709–721. ACM Press (Jun 2018). https://doi.org/10.1145/3188745.3188924
4. Bitansky, N., Garg, S., Lin, H., Pass, R., Telang, S.: Succinct randomized encodings and their applications. In: Servedio, R.A., Rubinfeld, R. (eds.) 47th ACM STOC. pp. 439–448. ACM Press (Jun 2015). https://doi.org/10.1145/2746539.2746574
5. Brakerski, Z., Holmgren, J., Kalai, Y.T.: Non-interactive delegation and batch NP verification from standard computational assumptions. In: Hatami, H., McKenzie, P., King, V. (eds.) 49th ACM STOC. pp. 474–482. ACM Press (Jun 2017). https://doi.org/10.1145/3055399.3055497
6. Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) LWE. In: Ostrovsky, R. (ed.) 52nd FOCS. pp. 97–106. IEEE Computer Society Press (Oct 2011). https://doi.org/10.1109/FOCS.2011.12
7. Brewster, T.: FaceApp: Is the Russian face-aging app a danger to your privacy? Forbes (Jul 2019), `https://www.forbes.com/sites/thomasbrewster/2019/07/17/faceapp-is-the-russian-face-aging-app-a-danger-to-your-privacy/`
8. Canetti, R., Holmgren, J.: Fully succinct garbled RAM. In: Sudan, M. (ed.) ITCS 2016. pp. 169–178 (Jan 2016). https://doi.org/10.1145/2840728.2840765
9. Canetti, R., Holmgren, J., Jain, A., Vaikuntanathan, V.: Succinct garbling and indistinguishability obfuscation for RAM programs. In: Servedio, R.A., Rubinfeld, R. (eds.) 47th ACM STOC. pp. 429–437. ACM Press (Jun 2015). https://doi.org/10.1145/2746539.2746621
10. Chesney, R., Citron, D.: Deep fakes: A looming challenge for privacy, democracy, and national security. California Law Review **107**(6), 1753–1820 (Dec 2019), `http://www.californialawreview.org/print/deep-fakes-a-looming-challenge-for-privacy-democracy-and-national-security/`
11. Corena, J.C., Ohtsuki, T.: Secure and fast aggregation of financial data in cloud-based expense tracking applications. J. Network Syst. Manage. **20**(4), 534–560 (2012). https://doi.org/10.1007/s10922-012-9248-y, `https://doi.org/10.1007/s10922-012-9248-y`
12. Doelitzscher, F., Reich, C., Knahl, M., Paßfall, A., Clarke, N.L.: An agent based business aware incident detection system for cloud environments. J. Cloud Computing **1**, 9 (2012). https://doi.org/10.1186/2192-113X-1-9, `https://doi.org/10.1186/2192-113X-1-9`

13. Dölitzscher, F., Knahl, M., Reich, C., Clarke, N.L.: Anomaly detection in iaas clouds. In: IEEE 5th International Conference on Cloud Computing Technology and Science, CloudCom 2013, Bristol, United Kingdom, December 2-5, 2013, Volume 1. pp. 387–394. IEEE Computer Society (2013). https://doi.org/10.1109/CloudCom.2013.57, `https://doi.org/10.1109/CloudCom.2013.57`
14. Galante, J., Kharif, O., Alpeyev, P.: Sony network breach shows Amazon cloud's appeal for hackers. Bloomberg (May 2011), `https://www.bloomberg.com/news/articles/2011-05-15/sony-attack-shows-amazon-s-cloud-service-lures-hackers-at-pennies-an-hour`
15. Garg, S., Srinivasan, A.: A simple construction of iO for turing machines. In: Beimel, A., Dziembowski, S. (eds.) TCC 2018, Part II. LNCS, vol. 11240, pp. 425–454. Springer, Heidelberg (Nov 2018). https://doi.org/10.1007/978-3-030-03810-6_16
16. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) 41st ACM STOC. pp. 169–178. ACM Press (May / Jun 2009). https://doi.org/10.1145/1536414.1536440
17. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. LNCS, vol. 8042, pp. 75–92. Springer, Heidelberg (Aug 2013). https://doi.org/10.1007/978-3-642-40041-4_5
18. Goldreich, O.: Foundations of Cryptography: Volume 1. Cambridge University Press, USA (2006)
19. Goldwasser, S., Kalai, Y.T., Popa, R.A., Vaikuntanathan, V., Zeldovich, N.: Reusable garbled circuits and succinct functional encryption. In: Boneh, D., Roughgarden, T., Feigenbaum, J. (eds.) 45th ACM STOC. pp. 555–564. ACM Press (Jun 2013). https://doi.org/10.1145/2488608.2488678
20. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. SIAM Journal on Computing **18**(1), 186–208 (1989)
21. Greenberg, A.: Sony hacker may have accessed 77 million users' data, possibly including credit cards. Forbes (Apr 2011), `https://www.forbes.com/sites/andygreenberg/2011/04/26/sony-hacker-may-have-accessed-77-million-users-data-possibly-including-credit-cards/`
22. Groth, J., Ostrovsky, R., Sahai, A.: Perfect non-interactive zero knowledge for NP. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 339–358. Springer, Heidelberg (May / Jun 2006). https://doi.org/10.1007/11761679_21
23. Hubacek, P., Wichs, D.: On the communication complexity of secure function evaluation with long output. In: Roughgarden, T. (ed.) ITCS 2015. pp. 163–172 (Jan 2015). https://doi.org/10.1145/2688073.2688105
24. Kalai, Y.T., Paneth, O.: Delegating RAM computations. In: Hirt, M., Smith, A.D. (eds.) TCC 2016-B, Part II. LNCS, vol. 9986, pp. 91–118. Springer, Heidelberg (Oct / Nov 2016). https://doi.org/10.1007/978-3-662-53644-5_4
25. Kalai, Y.T., Paneth, O., Yang, L.: How to delegate computations publicly. In: 51st ACM STOC. pp. 1115–1124. ACM Press (2019). https://doi.org/10.1145/3313276.3316411
26. Kalai, Y.T., Raz, R., Rothblum, R.D.: Delegation for bounded space. In: Boneh, D., Roughgarden, T., Feigenbaum, J. (eds.) 45th ACM STOC. pp. 565–574. ACM Press (Jun 2013). https://doi.org/10.1145/2488608.2488679
27. Kalai, Y.T., Raz, R., Rothblum, R.D.: How to delegate computations: the power of no-signaling proofs. In: Shmoys, D.B. (ed.) 46th ACM STOC. pp. 485–494. ACM Press (May / Jun 2014). https://doi.org/10.1145/2591796.2591809
28. Kocabas, Ö., Soyata, T.: Utilizing homomorphic encryption to implement secure and private medical cloud computing. In: Pu, C., Mohindra, A. (eds.) 8th IEEE International Conference on Cloud Computing, CLOUD 2015, New York City, NY, USA, June 27 - July 2, 2015. pp. 540–547. IEEE Computer Society (2015). https://doi.org/10.1109/CLOUD.2015.78, `https://doi.org/10.1109/CLOUD.2015.78`
29. Kocabas, Ö., Soyata, T., Couderc, J., Aktas, M., Xia, J., Huang, M.C.: Assessment of cloud-based health monitoring using homomorphic encryption. In: 2013 IEEE 31st International Conference on Computer Design, ICCD 2013, Asheville, NC, USA, October 6-9, 2013. pp. 443–446. IEEE Computer Society (2013). https://doi.org/10.1109/ICCD.2013.6657078, `https://doi.org/10.1109/ICCD.2013.6657078`
30. Koppula, V., Lewko, A.B., Waters, B.: Indistinguishability obfuscation for turing machines with unbounded memory. In: Servedio, R.A., Rubinfeld, R. (eds.) 47th ACM STOC. pp. 419–428. ACM Press (Jun 2015). https://doi.org/10.1145/2746539.2746614
31. Lindemann, J.: Towards abuse detection and prevention in iaas cloud computing. In: 10th International Conference on Availability, Reliability and Security, ARES 2015, Toulouse, France, August 24-27, 2015. pp. 211–217. IEEE Computer Society (2015). https://doi.org/10.1109/ARES.2015.72, `https://doi.org/10.1109/ARES.2015.72`
32. Lomas, N.: FaceApp uses neural networks for photorealistic selfie tweaks (Feb 2017), `http://social.techcrunch.com/2017/02/08/faceapp-uses-neural-networks-for-photorealistic-selfie-tweaks/`
33. Murphy, C., Huang, Z.: A popular Chinese face-swap app lets users make realistic Deepfakes. Time (Sep 2019), `https://time.com/5668482/chinese-face-swap-app-zao-deep-fakes/`

34. Peng, H., Hsu, W.W.Y., Ho, J., Yu, M.: Homomorphic encryption application on financialcloud framework. In: 2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016, Athens, Greece, December 6-9, 2016. pp. 1–5. IEEE (2016). https://doi.org/10.1109/SSCI.2016.7850013, `https://doi.org/10.1109/SSCI.2016.7850013`
35. Raisaro, J.L., Klann, J.G., Wagholikar, K.B., Estiri, H., Hubaux, J.P., Murphy, S.N.: Feasibility of Homomorphic Encryption for Sharing I2B2 Aggregate-Level Data in the Cloud. AMIA Joint Summits on Translational Science proceedings. AMIA Joint Summits on Translational Science **2017**, 176–185 (May 2018), `https://www.ncbi.nlm.nih.gov/pubmed/29888067`
36. Roth, T.: Breaking encryption in the cloud: Cheap, GPU assisted supercomputing for everyone (Aug 2011), `https://www.blackhat.com/html/bh-us-11/bh-us-11-briefings.html#Roth2`, black Hat Briefings
37. Tsukayama, H.: Is Sony hacking back? If so, Amazon says it isn't happening on its systems. Washngton Post (Dec 2014), `https://www.washingtonpost.com/news/the-switch/wp/2014/12/11/is-sony-hacking-back-if-so-amazon-says-it-isnt-happening-on-its-systems/`

A Preliminaries

A.1 Basic preliminaries

We recall some standard cryptographic definitions. Let λ denote the security parameter. A function $\mu(\cdot) : \mathbb{N} \to \mathbb{R}^+$ is said to be negligible if for any polynomial $\mathsf{poly}(\cdot)$ there exists $\lambda_0 \in \mathbb{N}$ such that for all $\lambda > \lambda_0$ we have $\mu(\lambda) < \frac{1}{\mathsf{poly}(\lambda)}$. We will use $\mathsf{negl}(\cdot)$ to denote an unspecified negligible function and $\mathsf{poly}(\cdot)$ to denote an unspecified polynomial function.

For a probabilistic algorithm $\mathcal{A}$, we let $\mathcal{A}(x; r)$ be the output of $\mathcal{A}$ on input x with random tape r. When r is omitted, $\mathcal{A}(x)$ denotes a distribution on the outputs of $\mathcal{A}(x; r)$ for a uniformly random r, and we denote by $y \leftarrow \mathcal{A}(x)$ the process of sampling from this distribution. The notation $y \coloneqq \mathcal{A}(x)$ is used when $\mathcal{A}$ is deterministic. The abbreviation PPT refers to a Probabilistic Polynomial Time algorithm. Finally, we use the notation $\mathcal{A}^{\mathcal{F}}$ to denote an algorithm $\mathcal{A}$ with *oracle access* to a function $\mathcal{F}$.

A.2 Fully Homomorphic Encryption

Definition 4 (Fully Homomorphic Encryption (*FHE*)). *A (public-key) fully homomorphic encryption scheme FHE is a tuple of four algorithms*

$$FHE = (FHE.\mathsf{KeyGen}, FHE.\mathsf{Enc}, FHE.\mathsf{Dec}, FHE.\mathsf{Eval})$$

with the following syntax.

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow FHE.\mathsf{KeyGen}(1^\lambda)$ *generates a public key secret key pair* $(\mathsf{pk}, \mathsf{sk})$.
- $\mathsf{ct} \leftarrow FHE.\mathsf{Enc}(\mathsf{pk}, x)$ *takes as input a public key* pk *and a message* $x \in \{0, 1\}$*, and outputs a ciphertext* ct.
- $x \leftarrow FHE.\mathsf{Dec}(\mathsf{sk}, \mathsf{ct})$ *takes as input a secret key* sk *and a ciphertext* ct*, and outputs a message* $x \in \{0, 1\}$.
- $c^* \leftarrow FHE.\mathsf{Eval}(\mathsf{pk}, C, \mathsf{ct}_1, \dots, \mathsf{ct}_n)$ *takes as input a public key* pk*, a circuit* $C : \{0, 1\}^n \to \{0, 1\}$*, and* n *ciphertexts* $c_1, \dots, c_n$*, and outputs a ciphertext* c^*.

The algorithms must satisfy the following properties.

- ***Encryption Correctness:*** *For any choice of* $(\mathsf{pk}, \mathsf{sk}) \leftarrow FHE.\mathsf{KeyGen}(1^\lambda)$*, any* $x \in \{0, 1\}$*, and any* $\mathsf{ct} \leftarrow FHE.\mathsf{Enc}(\mathsf{pk}, x)$*, it holds that* $FHE.\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}) = x$.
- ***Evaluation Correctness:*** *For any choice of* $(\mathsf{pk}, \mathsf{sk}) \leftarrow FHE.\mathsf{KeyGen}(1^\lambda)$*, any inputs* $x_1, \dots, x_n \in \{0, 1\}$ *with ciphertexts* $\mathsf{ct}_i \leftarrow FHE.\mathsf{Enc}(\mathsf{pk}, x_i)$*, and any polynomial-size circuit* $C : \{0, 1\}^n \to \{0, 1\}$*, it holds that*
$$FHE.\mathsf{Dec}(\mathsf{sk}, FHE.\mathsf{Eval}(\mathsf{pk}, C, \mathsf{ct}_1, \dots, \mathsf{ct}_n)) = C(x_1, \dots, x_n).$$
Moreover, there exists some fixed *polynomial* $s = s(\lambda)$ *such the output of* $FHE.\mathsf{Eval}$ *is at most* s *bits, regardless of the size of* f *or the number of inputs* n.

– ***Semantic Security:*** *For any PPT $\mathcal{A}$,*

$$\left| \Pr_{(\mathsf{pk},\mathsf{sk})\leftarrow FHE.\mathsf{KeyGen}(1^\lambda)}[\mathcal{A}(\mathsf{pk}, FHE.\mathsf{Enc}(\mathsf{pk},0))=1] - \Pr_{(\mathsf{pk},\mathsf{sk})\leftarrow FHE.\mathsf{KeyGen}(1^\lambda)}[\mathcal{A}(\mathsf{pk}, FHE.\mathsf{Enc}(\mathsf{pk},1))=1] \right| = \mathsf{negl}(\lambda).$$

We will actually make use of a variant of circuit-private *FHE, formalized in [23] as the notion of "FHE with rerandomization". A rerandomizable FHE scheme provides an additional algorithm:*

– $\mathsf{ct}_{\mathsf{new}} \leftarrow FHE.\mathsf{Rerand}(\mathsf{ct}_{\mathsf{old}})$ *takes as input a ciphertext* $\mathsf{ct}_{\mathsf{old}}$ *and produces a ciphertext* $\mathsf{ct}_{\mathsf{new}}$ *such that* $FHE.\mathsf{Dec}(\mathsf{sk},\mathsf{ct}_{\mathsf{old}}) = FHE.\mathsf{Dec}(\mathsf{sk},\mathsf{ct}_{\mathsf{new}})$.

Rerandomization guarantees that for every *choice of* $(\mathsf{pk},\mathsf{sk}) \leftarrow FHE.\mathsf{KeyGen}(1^\lambda)$*, and* every *pair of ciphertexts* ct,ct' *with* $FHE.\mathsf{Dec}(\mathsf{sk},\mathsf{ct}) = FHE.\mathsf{Dec}(\mathsf{sk},\mathsf{ct}')$*, the distribution* $FHE.\mathsf{Rerand}(c)$ *is statistically close to the distribution* $FHE.\mathsf{Rerand}(c')$*. Standard FHE schemes in the literature support rerandomization [6, 17].*

A.3 Non-interactive Zero Knowledge

Definition 5 (Non-interactive Zero Knowledge (NIZK)). *A non-interactive zero-knowledge argument of knowledge for an* NP *relation* $\mathcal{R} = \{(x,w)\}$ *is a tuple of four algorithms* ($\mathsf{NIZK.Setup}$, $\mathsf{NIZK.Prove}$, $\mathsf{NIZK.Ver}$) *with the following syntax.*

– $crs \leftarrow \mathsf{NIZK.Setup}(1^\lambda, 1^n)$ *takes a security parameter* λ *and a statement length* n*, and generates a common reference string* crs.
– $\pi \leftarrow \mathsf{NIZK.Prove}(crs, x, w)$ *takes a crs, and a statement, witness pair* $(x,w)\in\mathcal{R}$*, and outputs a proof* π.
– $0/1 \leftarrow \mathsf{NIZK.Ver}(crs, x, \pi)$ *takes a crs, a statement* x*, and proof* π *and either accepts (outputs 1) or rejects.*

The algorithms must satisfy the following properties. Let $\mathcal{L} := \{x : \exists\ w \text{ s.t. } (x,w)\in\mathcal{R}\}$.

Completeness *For every* $(x,w)\in\mathcal{R}$,

$$\Pr\left[\mathsf{NIZK.Ver}(crs,x,\pi)=1 : \begin{array}{r} crs \leftarrow \mathsf{NIZK.Setup}(1^\lambda,1^{|x|}) \\ \pi \leftarrow \mathsf{NIZK.Prove}(crs,x,w) \end{array}\right] = 1.$$

(Adaptive) Soundness *For all PPT* $\mathcal{A}$,

$$\Pr\left[\mathsf{NIZK.Ver}(crs,x,\pi)=1 \wedge x\notin\mathcal{L} : \begin{array}{r} crs \leftarrow \mathsf{NIZK.Setup}(1^\lambda,1^n) \\ (x,\pi) \leftarrow \mathcal{A}(crs) \end{array}\right] = \mathsf{negl}(\lambda).$$

(Adaptive) Zero Knowledge *There exists a PPT simulator* $\mathsf{NIZK}.Sim := (\mathsf{NIZK}.Sim_1, \mathsf{NIZK}.Sim_2)$ *such that for all PPT* $\mathcal{A}$,

$$\left| \Pr\left[\mathcal{A}^{\mathsf{NIZK.Prove}(crs,\cdot,\cdot)}(crs)=1 : crs \leftarrow \mathsf{NIZK.Setup}(1^\lambda,1^n)\right] - \Pr\left[\mathcal{A}^{\mathsf{NIZK}.Sim_2'(crs,\tau,\cdot,\cdot)}(crs)=1 : (crs,\tau) \leftarrow \mathsf{NIZK}.Sim_1(1^\lambda,1^n)\right] \right| = \mathsf{negl}(\lambda),$$

where $\mathsf{NIZK}.Sim_2'(crs,\tau,x,w) = \mathsf{NIZK}.Sim_2(crs,\tau,x)$ *if* $(x,w)\in\mathcal{R}$ *and outputs* $\perp$ *otherwise.*

Knowledge Extraction *There exists a PPT extractor* $\mathsf{NIZK.Ext} := (\mathsf{NIZK.Ext}_1, \mathsf{NIZK.Ext}_2)$ *such that for all PPT* $\mathcal{A}$*, the following two properties hold.*

$$\left| \Pr\left[\mathcal{A}(crs)=1 : crs \leftarrow \mathsf{NIZK.Setup}(1^\lambda,1^n)\right] - \Pr\left[\mathcal{A}(crs)=1 : (crs,\tau) \leftarrow \mathsf{NIZK.Ext}_1(1^\lambda,1^n)\right] \right| = \mathsf{negl}(\lambda),$$

$$\Pr\left[\mathsf{NIZK.Ver}(crs,x,\pi)=1 \wedge (x,w)\notin\mathcal{R} : \begin{array}{r} (crs,\tau) \leftarrow \mathsf{NIZK.Ext}_1(1^\lambda,1^n) \\ (x,\pi) \leftarrow \mathcal{A}(crs) \\ w \leftarrow \mathsf{NIZK.Ext}_2(crs,\tau,x,\pi) \end{array}\right] = \mathsf{negl}(\lambda).$$

Remark 2. We let "NIZK for NP" denote a NIZK for an NP-complete relation such as $\mathsf{CEval} = \{(C, w) : C(w) = 1\}$ where C is a polynomial-size circuit and w is an input to C.

B Proof of Theorem 2

B.1 Context for Theorem 2

Let us present Theorem 2 in the context of current literature. As stated in the theorem, our compiler may be applied to any publicly-verifiable delegation scheme for Turing machines. However, we remark here that our definition of responsible delegation and the accompanying compiler are actually agnostic to the computational model. For example, our compiler could be applied to a delegation scheme for RAM machines to produce a responsible private delegation scheme for RAM machines. Moreover, one could operate in the *pre-processing* model. In this case, the Setup algorithm will additionally take as input the description of a (potentially very large) circuit C, and the computation z delegated by the user will simply be a particular input to C.

Again, regardless of the computational model, our compiler only works given a publicly-verifiable scheme. In light of this, we briefly review the literature on publicly-verifiable delegation schemes. A long line of work has constructed schemes for Turing machines [30, 2, 15] and RAM machines [4, 9, 8] that rely on one-way functions and *indistinguishability obfuscation*, a very strong and non-standard cryptographic assumption. Recently, a publicly-verifiable delegation scheme for these models of computation [25] was constructed from a new assumption on bilinear maps. Any of these schemes may be plugged in to our compiler, and any progress on publicly-verifiable delegation now immediately implies progress on responsible and private delegation.

Furthermore, publicly-verifiable delegation schemes in the pre-processing setting are known from standard assumptions. In particular, [19] present a scheme from the Learning with Errors (LWE) assumption (though with the caveat that the user's runtime depends on the *depth* of the circuit C). Our compiler may be applied to their scheme to produce a responsible private delegation scheme in the pre-processing setting. This is particularly relevant when considering the Deepfake application described in Section 1.2, as it would be natural to define the circuit C to be a universal Deepfake program.

And what about the cryptographic tools used in our compiler? Fully homomorphic encryption (FHE) schemes [16, 6, 17] are constructed by making a circular security assumption on an appropriate LWE-based encryption scheme. As far as we know, there is no reason to believe that these LWE-based schemes would not achieve circular security, yet it remains an open problem to construct FHE based *only* on LWE (or any other standard assumption). We do note that one can build *levelled* FHE based only on LWE, where homomorphic evaluation is only guaranteed to be correct up to a pre-specified circuit depth. Our compiler can be instantiated with such a scheme, as long as we fix in advance the maximum depth of both the computation being delegated and the misuse-detection circuit D being used. Finally, non-interactive zero knowledge arguments of knowledge are a standard cryptographic primitive, and follow for example from bilinear maps [22].

Remark 3 (Public delegatability). We briefly discuss the concept of public delegatability. Note that (in the setting of Turing machine delegation) without the additional property of misuse-prevention, the Setup algorithm could be included as part of User.Del without affecting efficiency, and indeed some works do not define a separate Setup algorithm [4]. In this setting, the notion of public delegatability holds by default. However, observe that including a separate Setup algorithm is crucial when considering the additional property of misuse-prevention, since this notion is defined relative to a malicious user. Thus, we stress that our compiler produces a responsible private delegation protocol with *public delegatability*, so that the output of Setup may be used and re-used by any user (and also any server).

Furthermore, public delegatability becomes crucial in the pre-processing setting, as it allows the costly Setup to run once (its complexity depends on the circuit size), and then allows any user to use the resulting (small) public parameters for delegation. Thus, we stress that in the pre-preprocessing setting, our compiler would take any *publicly-delegatable* delegation protocol and produce a *publicly-delegatable* responsible private delegation protocol.

B.2 Proof of Theorem 2

In this section, we prove that the scheme $\mathit{\Pi}$ (as presented in Section 3.3) is a responsible private delegation scheme. As is often the case, completeness is straightforward. Soundness, privacy and misuse-prevention are shown in Lemmas 1 to 3, respectively.

Lemma 1. *Let $\mathit{\Pi}$ be the scheme presented in Section 3.3. $\mathit{\Pi}$ is sound.*

Proof. Assume that there exists a PPT cheating server $\mathcal{A}$ that convinces $\mathit{\Pi}.\mathsf{User}$ to accept a computation z such that $\mathcal{U}(z) = 0$. Concretely, $\mathcal{A}$ takes as input the tuple

$$(\mathit{\Pi}.\mathsf{pk}, \mathit{\Pi}.\mathsf{msg}) := ((\mathit{\Phi}.\mathsf{pk}, crs_{\mathsf{User}}, crs_{\mathsf{Server}}), (\mathit{FHE}.\mathsf{pk}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{ct}_z, \pi_{\mathsf{User}}))$$

that results from running $\mathit{\Pi}.\mathsf{Setup}(1^\lambda, 1^n)$ followed by $\mathit{\Pi}.\mathsf{User}.\mathsf{Del}$ with input z. With probability at least $1/\mathsf{poly}(\lambda)$, it outputs a proof π^* such that $1 \leftarrow \mathit{\Pi}.\mathsf{User}.\mathsf{Ver}(\mathit{\Pi}.\mathsf{vk}, \pi^*)$.

We change $\mathcal{A}$'s input distribution in three steps, arguing that each only affects $\mathcal{A}$'s probability of success by a negligible amount.

1. Rather than generating crs_{Server} with $\mathsf{NIZK}.\mathsf{Setup}$, generate $(crs_{\mathsf{Server}}, \tau_{\mathsf{Server}}) \leftarrow \mathsf{NIZK}.\mathsf{Ext}_1(1^\lambda, 1^{n_{\mathsf{Server}}})$. By the knowledge extraction of NIZK, $\mathcal{A}$ will notice this switch with only negligible probability.
2. Rather than generating crs_{User} with $\mathsf{NIZK}.\mathsf{Setup}$, generate

$$(crs_{\mathsf{User}}, \tau_{\mathsf{User}}) \leftarrow \mathsf{NIZK}.\mathit{Sim}_1(1^\lambda, 1^{n_{\mathsf{User}}})$$
$$\pi_{\mathsf{User}} \leftarrow \mathsf{NIZK}.\mathit{Sim}_2(crs_{\mathsf{User}}, \tau_{\mathsf{User}}, \mathsf{CheckUser}[\mathit{\Phi}.\mathsf{dk}, \mathit{FHE}.\mathsf{pk}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{ct}_z]).$$

   Since the $\mathsf{CheckUser}$ circuit has an accepting input, zero knowledge of NIZK implies that $\mathcal{A}$ will only notice this switch with negligible probability.
3. Set $\mathsf{ct}_z \leftarrow \mathit{FHE}.\mathsf{Enc}(\mathit{FHE}.\mathsf{pk}, 0)$. By semantic security of FHE, $\mathcal{A}$ will notice this switch with only negligible probability.

Now we use $\mathcal{A}$ to construct an adversary $\mathcal{A}'$ that breaks the soundness of $\mathit{\Phi}$. $\mathcal{A}'$ operates as follows.

1. Take as input $(\mathit{\Phi}.\mathsf{pk}, \mathit{\Phi}.\mathsf{msg})$ that results from running $\mathit{\Phi}.\mathsf{Setup}(1^\lambda, 1^n)$ followed by $\mathit{\Phi}.\mathsf{User}.\mathsf{Del}$ with input z.
2. Generate $(crs_{\mathsf{Server}}, \tau_{\mathsf{Server}}) \leftarrow \mathsf{NIZK}.\mathsf{Ext}_1(1^\lambda, 1^{n_{\mathsf{Server}}})$.
3. Generate $(crs_{\mathsf{User}}, \tau_{\mathsf{User}}) \leftarrow \mathsf{NIZK}.\mathit{Sim}_1(1^\lambda, 1^{n_{\mathsf{User}}})$.
4. Generate $(\mathit{FHE}.\mathsf{pk}, \mathit{FHE}.\mathsf{sk}) \leftarrow \mathit{FHE}.\mathsf{KeyGen}(1^\lambda)$.
5. Encrypt $\mathsf{ct}_z \leftarrow \mathit{FHE}.\mathsf{Enc}(\mathit{FHE}.\mathsf{pk}, 0)$.
6. Generate $\pi_{\mathsf{User}} \leftarrow \mathsf{NIZK}.\mathit{Sim}_2(crs_{\mathsf{User}}, \tau_{\mathsf{User}}, \mathsf{CheckUser}[\mathit{\Phi}.\mathsf{dk}, \mathit{FHE}.\mathsf{pk}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{ct}_z])$.
7. Forward $(\mathit{\Pi}.\mathsf{pk}, \mathit{\Pi}.\mathsf{msg}) := ((\mathit{\Phi}.\mathsf{pk}, crs_{\mathsf{User}}, crs_{\mathsf{Server}}), (\mathit{FHE}.\mathsf{pk}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{ct}_z, \pi_{\mathsf{User}}))$ to $\mathcal{A}$. Note that, as argued above, this distribution is indistiguishable from an honestly generated input distribution for $\mathcal{A}$.
8. Receive a proof π^* from $\mathcal{A}$ and decrypt $\pi_{\mathsf{Server}} := \mathit{FHE}.\mathsf{Dec}(\mathit{FHE}.\mathsf{sk}, \pi^*)$.
9. Compute and output $\mathit{\Phi}.\pi^* \leftarrow \mathsf{NIZK}.\mathsf{Ext}_2(crs_{\mathsf{Server}}, \tau_{\mathsf{Server}}, \mathit{\Phi}.\mathsf{User}.\mathsf{Ver}, \pi_{\mathsf{Server}})$.

By assumption that $\mathcal{A}$ succeeds with $1/\mathsf{poly}(\lambda)$ probability, and by knowledge extraction of NIZK, $\mathit{\Phi}.\pi^*$ will be a valid witness for $\mathit{\Phi}.\mathsf{User}.\mathsf{Ver}$ with $1/\mathsf{poly}(\lambda)$ probability. Thus, $\mathcal{A}'$ breaks the soundness of $\mathit{\Phi}$.

□

Lemma 2. *Let $\mathit{\Pi}$ be the scheme presented in Section 3.3. $\mathit{\Pi}$ is private.*

Proof. As this proof has a similar flavor as the proof of soundness, we give a slightly less formal sketch. Assume that there exists a PPT cheating server $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ that breaks the privacy of $\mathit{\Pi}$ with $1/\mathsf{poly}(\lambda)$ probability. We change the distribution seen by $\mathcal{A}_0$ and $\mathcal{A}_1$ in a sequence of steps, such that each switch is noticed with negligible probability.

1. Generate $(crs_{\mathsf{User}}, \tau_{\mathsf{User}}) \leftarrow \mathsf{NIZK}.\mathit{Sim}_1(1^\lambda, 1^{n_{\mathsf{User}}})$.
2. For the challenge message input to $\mathcal{A}_1$, and additionally whenever $\mathcal{A}_1$ queries its $\mathit{\Pi}.\mathsf{User}.\mathsf{Del}$ oracle, simulate π_{User} with $\mathsf{NIZK}.\mathit{Sim}_2$.

3. For the challenge message input to $\mathcal{A}_1$, and additionally whenever $\mathcal{A}_1$ queries its $\Pi.\mathsf{User.Del}$ oracle, set ct_z to be an encryption of 0.

At this point, each message given to $\mathcal{A}_1$ can be generated given only $\mathit{FHE}.\mathsf{pk}$ and $\mathsf{ct}_{\mathsf{msg}}$. Thus, $\mathcal{A}$ can be used directly to break semantic security of FHE. □

Lemma 3. *Let Π be the scheme presented in Section 3.3. Π is misuse-preventing.*

Proof. We define the following simulator $\Pi.\mathcal{S} = (\Pi.\mathcal{S}_0, \Pi.\mathcal{S}_1)$.

- $\Pi.\mathcal{S}_0(1^\lambda)$:
  1. Compute $(\Phi.\mathsf{pk}, \Phi.\mathsf{dk}) \leftarrow \Phi.\mathsf{Setup}(1^\lambda)$.
  2. Let $n_{\mathsf{User}} := p_{\mathsf{User}}(\lambda, n)$ and $n_{\mathsf{Server}} := p_{\mathsf{Server}}(\lambda, n)$.
  3. Compute $(crs_{\mathsf{User}}, \tau_{\mathsf{User}}) \leftarrow \mathsf{NIZK.Ext}_1(1^\lambda, 1^{n_{\mathsf{User}}})$ and $(crs_{\mathsf{Server}}, \tau_{\mathsf{Server}}) \leftarrow \mathsf{NIZK}.\mathit{Sim}_1(1^\lambda, 1^{n_{\mathsf{Server}}})$.
  4. Output $\Pi.\mathsf{pk} := (\Phi.\mathsf{pk}, crs_{\mathsf{User}}, crs_{\mathsf{Server}})$, $\mathsf{dk} := (\Phi.\mathsf{dk}, crs_{\mathsf{User}}, crs_{\mathsf{Server}})$, and the simulation state $\mathsf{st} := (\Pi.\mathsf{pk}, \tau_{\mathsf{User}}, \tau_{\mathsf{Server}})$.
- $\Pi.\mathcal{S}_1^{\mathcal{F}_D}(\mathsf{st}, \Pi.\mathsf{msg})$:
  1. Parse st as $((\Phi.\mathsf{pk}, crs_{\mathsf{User}}, crs_{\mathsf{Server}}), \tau_{\mathsf{User}}, \tau_{\mathsf{Server}})$ and $\Pi.\mathsf{msg}$ as $(\mathit{FHE}.\mathsf{pk}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{ct}_z, \pi_{\mathsf{User}})$.
  2. Check that $\mathsf{NIZK.Ver}(crs_{\mathsf{User}}, \mathsf{CheckUser}[\Phi.\mathsf{dk}, \mathit{FHE}.\mathsf{pk}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{ct}_z], \pi_{\mathsf{User}}) = 1$, if not return $\perp$. Otherwise, compute
$$(\Phi.\mathsf{msg}, z, r_1, r_2, r_3, r_4) \leftarrow \mathsf{NIZK.Ext}_2\left(\begin{matrix} crs_{\mathsf{User}}, \tau_{\mathsf{User}}, \\ \mathsf{CheckUser}[\Phi.\mathsf{dk}, \mathit{FHE}.\mathsf{pk}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{ct}_z], \pi_{\mathsf{User}} \end{matrix}\right).$$
  3. Compute π_{Server} as follows.
     (a) Set $y := \mathcal{F}_D(z)$.
     (b) If $y = 0$, set $\pi_{\mathsf{Server}} := \perp$ (appropriately padded).
     (c) Otherwise, compute $\pi_{\mathsf{Server}} \leftarrow \mathsf{NIZK}.\mathit{Sim}_2(crs_{\mathsf{Server}}, \tau_{\mathsf{Server}}, \Phi.\mathsf{User.Ver})$.
  4. Return $\Pi.\pi := \mathit{FHE}.\mathsf{Rerand}(\mathit{FHE}.\mathsf{Enc}(\mathit{FHE}.\mathsf{pk}, \pi_{\mathsf{Server}}))$.

First, note that the running time of $\Pi.\mathcal{S}$ does not depend on the computation z being delegated. Each of $\Pi.\mathcal{S}_0$ and $\Pi.\mathcal{S}_1$ runs in some fixed polynomial time in its input length, where the exact polynomial may be derived from the NIZK and FHE schemes that Π is instantiated with.

Now, we show that the above simulation is indistinguishable from interaction with the actual server via a sequence of hybrids. We highlight (in red) the steps the differ from hybrid to hybrid.

Hybrid_0: The real world interaction.
Hybrid_1: During $\Pi.\mathsf{Setup}$,
  3. Compute $(crs_{\mathsf{User}}, \tau_{\mathsf{User}}) \leftarrow \mathsf{NIZK.Ext}_1(1^\lambda, 1^{n_{\mathsf{User}}})$ and $(crs_{\mathsf{Server}}, \tau_{\mathsf{Server}}) \leftarrow \mathsf{NIZK}.\mathit{Sim}_1(1^\lambda, 1^{n_{\mathsf{Server}}})$.
  Indistinguishability follows from the zero knowledge and knowledge extraction properties of NIZK.
Hybrid_2: During $\Pi.\mathsf{Server}$,
  2. Check that $\mathsf{NIZK.Ver}(crs_{\mathsf{User}}, \mathsf{CheckUser}[\Phi.\mathsf{dk}, \mathit{FHE}.\mathsf{pk}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{ct}_z], \pi_{\mathsf{User}}) = 1$, if not return $\perp$. Otherwise, compute
$$(\Phi.\mathsf{msg}, z, r_1, r_2, r_3, r_4) \leftarrow \mathsf{NIZK.Ext}_2\left(\begin{matrix} crs_{\mathsf{User}}, \tau_{\mathsf{User}}, \\ \mathsf{CheckUser}[\Phi.\mathsf{dk}, \mathit{FHE}.\mathsf{pk}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{ct}_z], \pi_{\mathsf{User}} \end{matrix}\right).$$
  3. Compute π_{Server} as follows.
     (a) Compute $y := \mathcal{U}(z) \wedge D(z)$.
     (b) If $y = 0$, output $\perp$ (appropriately padded).
     (c) Otherwise,
        i. Compute $\Phi.\pi \leftarrow \Phi.\mathsf{Server}(\Phi.\mathsf{pk}, \Phi.\mathsf{msg})$.
        ii. Output $\pi_{\mathsf{Server}} \leftarrow \mathsf{NIZK.Prove}(crs_{\mathsf{Server}}, \Phi.\mathsf{User.Ver}, \Phi.\pi)$.

4. Return $\mathit{\Pi}.\pi := \mathit{FHE}.\mathsf{Rerand}(\mathit{FHE}.\mathsf{Enc}(\mathit{FHE}.\mathsf{pk}, \pi_{\mathsf{Server}}))$.

First, note that by the knowledge extraction property of NIZK, in Hybrid_2, $\mathsf{NIZK}.\mathsf{Ext}_2$ will extract a valid witness for $\mathsf{CheckUser}[\mathit{\Phi}.\mathsf{dk}, \mathit{FHE}.\mathsf{pk}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{ct}_z]$ with overwhelming probability. In this case, we are guaranteed that $\mathsf{ct}_{\mathsf{msg}}, \mathsf{ct}_z$ are valid encryptions of $\mathit{\Phi}.\mathsf{msg}, z$ under $\mathit{FHE}.\mathsf{pk}$. Now observe that in step 3, we are computing ServerEval on $(\mathit{\Phi}.\mathsf{msg}, z)$ in the clear, rather than under the FHE. Thus by perfect correctness of FHE, we know that the ct_π computed in step 3 of Hybrid_1 is an encryption of the π_{Server} computed in Hybrid_2 under $\mathit{FHE}.\mathsf{pk}$. Finally, the statistical rerandomizable property of FHE ensures that the $\mathit{\Pi}.\pi$ computed in step 4 of each hybrid is indistinguishable.

Hybrid_3: During $\mathit{\Pi}.\mathsf{Server}$,

3. Compute π_{Server} as follows.
   (a) Compute $y := \mathcal{U}(z) \wedge D(z)$.
   (b) If $y = 0$, output $\perp$ (appropriately padded).
   (c) Otherwise, compute $\pi_{\mathsf{Server}} \leftarrow \mathsf{NIZK}.\mathit{Sim}_1(\mathit{crs}_{\mathsf{Server}}, \tau_{\mathsf{Server}}, \mathit{\Phi}.\mathsf{User}.\mathsf{Ver})$.

First we argue that in Hybrid_2, $\mathit{\Phi}.\pi$ is a valid witness for $\mathit{\Phi}.\mathsf{User}.\mathsf{Ver}$ with overwhelming probability. To see this, observe that knowledge extraction of NIZK implies that $\mathit{\Phi}_{\mathsf{msg}}$ is in the support of $\mathit{\Phi}.\mathsf{User}.\mathsf{Del}$ on input z. If step 3.(c) is reached, we are guaranteed that $\mathcal{U}(z) = 1$. Thus, the $\mathit{\Phi}.\pi$ computed in step 3.(c).(i) must be an accepting input to $\mathit{\Phi}.\mathsf{User}.\mathsf{Ver}$, by the perfect completeness of $\mathit{\Phi}$. Given this, indistinguishability follows by the zero knowledge property of NIZK.

Hybrid_4: During $\mathit{\Pi}.\mathsf{Server}$,

3. Compute π_{Server} as follows.
   (a) Set $y := \mathcal{F}_D(z)$.
   (b) If $y = 0$, output $\perp$ (appropriately padded).
   (c) Otherwise, compute $\pi_{\mathsf{Server}} \leftarrow \mathsf{NIZK}.\mathit{Sim}_2(\mathit{crs}_{\mathsf{Server}}, \tau_{\mathsf{Server}}, \mathit{\Phi}.\mathsf{User}.\mathsf{Ver})$.

This is merely syntactic switch, since $\mathcal{F}_D$ computes exactly $\mathcal{U}(z) \wedge D(z)$.

□