

# Adaptively Secure Two-party Computation from Indistinguishability Obfuscation\*

Ran Canetti<sup>†</sup>      Shafi Goldwasser<sup>‡</sup>      Oxana Poburinnaya<sup>§</sup>

## Abstract

We present the first two-round, two-party general function evaluation protocol that is secure against honest-but-curious adaptive corruption of both parties. In addition, the protocol is incoercible for one of the parties, and fully leakage tolerant. It requires a *global* (non-programmable) reference string and is based on one way functions and general-purpose indistinguishability obfuscation with sub-exponential security, as well as augmented non-committing encryption.

A Byzantine version of the protocol, obtained by applying the Canetti et al. [STOC 02] compiler, achieves UC security with comparable efficiency parameters, but is no longer incoercible.<sup>1</sup>

## 1 Introduction

Obtaining adaptive security, namely guaranteeing security against adversaries that decide who to corrupt in an adaptive way depending on their view of the computation so far, has been a major challenge in secure computation since its inception. Indeed, adaptive security provides a more realistic modeling of adversarial behavior and party infection in modern communication networks. Furthermore, when combined with an additional property called *corruption oblivious simulation*, adaptive security implies a strong variant of leakage tolerance [BCH12], namely resilience to side channel attacks on the participating computational devices.

Guaranteeing adaptive security turns out to be considerably more challenging than guaranteeing security in the static setting where the set of corrupted parties is fixed in advance. As in the static setting, the security guarantees become stronger when the adversary is allowed to corrupt more parties. Furthermore, while in the static case the situation where all the parties are corrupted is trivial, in the adaptive case protecting against adversaries that can eventually corrupt all parties is by far the hardest case. Note that withstanding corruption of all parties is crucial for guaranteeing meaningful security of a protocol within a larger system or context. Also, the transformation from adaptive security to leakage tolerance is most meaningful in this case (namely, leakage from all parties). In particular:

---

\*Research Supported by the NSF MACS Frontier project, and by the Defense Advanced Research Projects Agency (DARPA) and the U.S. Army Research Office under contract number W911NF-15-C-0226.

<sup>†</sup>Tel-Aviv University and Boston University. [canetti@bu.edu](mailto:canetti@bu.edu). Supported in addition by the Check Point Institute for Information Security, ISF grant 1523/14, and NSF Algorithmic Foundations grant 1218461.

<sup>‡</sup>Weizmann and MIT. [shafi@csail.mit.edu](mailto:shafi@csail.mit.edu)

<sup>§</sup>Boston University. [oxanapob@bu.edu](mailto:oxanapob@bu.edu). Supported in addition by the Check Point Institute for Information Security and NSF Algorithmic Foundations grant 1218461.

<sup>1</sup>©IACR 2015. This article is the final version submitted by the authors to the IACR and to Springer-Verlag on January 13. The version published by Springer-Verlag is available at ?.

- The best round complexity of a fully adaptively secure protocol (namely a protocol that does not rely on secure erasure of information and that withstands adaptive corruption of all parties) is  $\tilde{\Omega}(d)$ , where  $d$  is the depth of the circuit being evaluated [BGW88, CFGN96, CLOS02]. (The works of [IPS08], [GS12] obtain constant number of rounds; however they cannot support corruption of all parties.) Furthermore, this is the best known round complexity even in the case of two party computation, even for the honest but curious setting, and even in the common reference string model.
- No fully leakage-tolerant (hence also no non-erasing oblivious simulation adaptively secure) general function evaluation protocol is known, with any number of rounds. Again, this holds even for honest-but-curious corruptions and even for two party protocols. (The protocol of [BDL14] obtains leakage tolerance in a setting with an initial, leakage free interactive set-up state.)

**Our results.** We present a *two-message*, two party secure function evaluation protocol that is secure against adaptive honest-but-curious corruption of all parties — thereby resolving a long standing open problem in the theory of secure computation. Furthermore, the protocol has non-erasing oblivious simulation, implying leakage tolerance. Security is based on subexponentially secure indistinguishability obfuscation for all circuits and one way functions, as well as augmented non-committing encryption as in [DN00, CLOS02].

The protocol requires a global, non-programmable reference string. Specifically, the string contains an obfuscated program to be run by parties. We call this mild version of the reference string model the *factory model*, since it is reminiscent of a setting where the obfuscated program is generated by a “trusted factory”.

The protocol is also *incoercible* [CG96] for one of the parties. That is, it provides one of the parties with a mechanism to present “convincing evidence” that explains its outgoing messages as resulting from any arbitrary input value (that may be different than the input value actually used). This holds even when the “coercer” expects to see the full internal state of the party. That is, we show:

**Theorem 1.** *Assume existence of sub-exponentially secure indistinguishability obfuscators for all circuits and one way functions, as well as augmented non-committing encryption. Then there exists a two-message, two party protocol, in the factory model, for evaluating any function with UC security in the presence of adaptive, honest-but-curious corruption of both parties. Furthermore:*

- (a) *The protocol is leakage tolerant as in [BCH12].*
- (b) *The protocol is incoercible with respect to one of the parties.*

In fact we show that the protocol satisfies a stronger variant of the [CG96] definition, that avoids a weakness in the original definition and is also universally composable. Furthermore, new definition may be of interest independently of the present protocol; in particular, it applies also to multi-party protocols and general (Byzantine) corruptions.

Compiling this protocol via the [GMW87, CLOS02] compiler, we obtain a constant-round, adaptively secure UC protocol for Byzantine adversaries in the standard CRS model. While the protocol remains leakage resilient, it is no longer incoercible.

**The protocol and techniques.** Before presenting the protocol, let us recall the definition of adaptive security. Security requires existence of a simulator that has access only to the trusted party for the function, and still emulates for the adversary (or, rather, the environment) an execution with the actual protocol. Since we are in the honest but curious model, we can assume without loss of generality that the adversary first waits to see the entire communication of the protocol to the end, and then corrupts all parties. The simulator should first create a simulated public transcript of the computation; then, when a party is corrupted and the simulator learns the input and the output of that party, the simulator should present the adversary with the appropriate random choices of the party that are consistent with the party’s input and messages sent.

Our starting point is Yao’s garbled circuit two party protocol, together with a two-message oblivious transfer. Recall that the first message in the protocol is the first OT message from the evaluator to the garbler. The second message, from the garbler to the evaluator, consists of the second OT message together with the garbled circuit. The evaluator then outputs the result of the computation. (If both parties wish to learn the output then they run another copy of the protocol in parallel, with reverse roles; or the evaluator can send the result to the garbler, but this adds one more round.)

When the OT is adaptively secure (as in, say, [CLOS02]) and the garbler’s message is encrypted using non-committing encryption, the protocol becomes adaptively secure with respect to the corruption of the evaluator. That is, the simulator can indeed create the transcript of the communication ahead of time (this is just ciphertexts of non-committing encryption) and when the evaluator is corrupted, provide the receiver message for the adaptively secure OT protocol. Note however that here the simulator has to commit to the garbled circuit, without knowing the garbler’s input.

Now, simulating the corruption of the garbler gets stuck: Here the environment expects to see the internal randomness of the garbler, including the random choices used for the generation of the garbled circuit. This we do not know how to do efficiently. In fact, in some cases such valid opening simply does not exist.

Our approach to get around this apparently inherent difficulty is to provide the garbler with an *obfuscated* version of his program. That is, let the common reference string contain an obfuscated version of the garbler’s program. The garbler will then run the obfuscated program on its input and random input and send the resulting message. The hope is that this will hide the internal randomness of the garbling, even when the environment sees the random input of the party.

However, this naive attempt does not work by itself, since the randomness for the protocol may well be correlated with the internal randomness that’s not supposed to be leaked. We address this issue by applying a pseudorandom function to the random and real inputs, and using the result as randomness to the protocol. In addition, to make the simulation go through with only indistinguishability obfuscation we follow the lead of Sahai and Waters [SW14] and use puncturable PRFs and an “explain” algorithm that allows the simulator to generate randomness that “explains” any given outgoing message.

As simple as the protocol is, the proof of security is rather delicate. One subtle point that deserves highlighting is the treatment of adaptivity in the choice of inputs. We first prove security in a model where the inputs are “selective”: the environment determines the inputs to the computation before it sees the reference string (namely the obfuscated programs). This is a rather weak security model. We then extend the analysis to the setting where the environment chooses the inputs adaptively. Here is where we use the sub-exponential security of the indistinguishability obfuscator: the analysis here requires as many hybrids as the number of potential inputs to the computation.

This number can be exponential. We note, however, that since the parameters of the obfuscation can be chosen to be larger than the size of the inputs to the computation, this requires only *sub*-exponential security of the  $i\mathcal{O}$  in use.

Finally we remark that the trust requirements from the reference string are relatively mild. First, it is non-programmable, in the sense that the simulator need not know any secret information related to the string. This means that the same instance of the reference string (namely, the same obfuscated program) can be used by multiple protocols and instances thereof without compromising security [CDPW07]. Second, static security holds even if the secrets associated with the reference string, namely the secrets of the obfuscation and the secret keys, are exposed.

**Concurrent work.** Concurrently to and independently from this work, two other works develop fully adaptively secure protocols using indistinguishability obfuscation. Both of these works appear in these proceedings [GP14, DKR14]. We give account of these works. Like ours, both works describe protocols for evaluating general functions, not only adaptively well formed ones as in [CLOS02]. Furthermore, all works obtain resilience against adaptive corruption for *all* parties. Finally, all works use the CRS model, where the CRS contains indistinguishability-obfuscated programs.

Dachman-Soled, Katz and Rao [DKR14] describe a general mechanism to transform programs into deniable ones, and use this mechanism to construct a four-round, multiparty, adaptively secure protocol against honest-but-curious corruptions. They then compile their protocol using the [CLOS02] compiler to handle Byzantine corruptions. Their analysis assumes only indistinguishability obfuscator and one way functions that are secure against polysize adversaries. Garg and Polychroniadou [GP14] directly describe a multi-party, two round, adaptively secure protocol against Byzantine corruptions. Similarly to this work, their analysis assumes sub-exponentially secure indistinguishability obfuscation and one way functions. Both protocols need a *programmable* (i.e., non-global) CRS, and neither protocol is incoercible.

**Organization.** Section 2 sketches the models of computation and recalls the main results of this work. Section 3 provides an overview of the construction. Section 4 provides a detailed presentation and analysis of the main protocol.

## 2 The models of computation

We consider the standard UC model of computation (as in [Can01]) with adaptive, honest-but-curious party corruptions. The parties and the environment have access to a *global, public* common reference string functionality. That is, the functionality first draws the reference string from a predefined distribution; next, all parties, including the adversary and the environment, obtain that string.

In our protocol the reference string is a description of programs run by parties; these programs are obfuscated and contain secret keys which shouldn't be known to the parties. We refer to such a global reference string model as “the factory model”, since it is intended to represent a situation where all parties obtain the program from a trusted “factory”.

**Leakage tolerance.** We will show that our protocol is leakage-tolerant. The leakage tolerance model we consider is the one in [BCH12], which is aimed as capturing protocols that are tolerant

to arbitrary amount of leakage, and where the security loss grows gradually with the amount of leakage. More specifically, in that model a protocol  $\pi$  computes a function  $f$  *in a leakage tolerant way* if no adversarial environment can tell whether it is interacting with the parties running  $\pi$ , while obtaining some  $\ell$ -bit leakage function of the individual internal states of the participants, or alternatively with a simulator and an ideal process for evaluating  $f$ , in which the simulator obtains some arbitrary  $\ell$ -bit function of each of the inputs of the parties.

It is shown there that if a protocol is shown to be adaptively secure with a special type of simulator, called *corruption oblivious simulator* (defined below), then the protocol is leakage tolerant.

A simulator is *corruption oblivious* if the information it gathers upon corruption of a party, namely the secret input (and potentially also the secret output) of that party, is used *only* to generate a simulated view of the local state of that party. This information is not used anywhere else in the simulation. Formally, the simulator creates a special subroutine for simulating the internal state of that party. The newly learned input of the corrupted party does not leave the confines of this subroutine. It is shown in [BCH12] that if a protocol is adaptively secure with a corruption oblivious simulator then it is also leakage tolerant.

**Incoercibility.** Incoercibility aims to protect the protocol participants from external authoritative (or otherwise coercive) entities that try to entice a party to reveal its state voluntarily. The idea is to provide parties with a “faking” algorithm that takes any desired fake value of the secret input, and exhibits “fake randomness” that is consistent with both the newly decided fake value and all the past messages sent by the party so far. Incoercible computation was defined in [CG96], where a generic construction from any deniable encryption scheme [CDNO97, SW14] is given. However, the construction there has a large number of rounds and works only when strictly less than half of the parties are either coerced or corrupted.

We revisit the definition of coercion-free computation, providing a new definition that is significantly stronger than the one in [CG96]. Specifically, the security guarantees provided by the new definition are preserved under universal composition. The definition also overcomes another weakness in the [CG96], as explained below. The definition here fleshes out ideas from [Can01, P. 59].

Informally, the definition captures incoercibility by asking that the protocol in question emulates an ideal functionality that employs the following “ideally incoercible” corruption process. Whenever the ideal functionality is asked by the ideal-model adversary to provide the internal information of some participant  $P$  in the protocol, the ideal functionality first asks the environment (representing the entity that invoked party  $P$  to participate in the protocol) whether to reveal the real input that  $P$  contributed to the computation, or alternatively whether to report some fake input. If the environment instructs to reveal the real input, then the functionality returns the real input of  $P$  to the adversary. If the environment provides a fake value  $x$ , then the functionality returns  $x$  to the adversary. Crucially, the adversary does not learn whether the value provided was fake or real.<sup>2</sup> We say that a functionality with such behavior is *incoercible*.

---

<sup>2</sup>We remark that the definition in [CG96] reveals to the ideal-model adversary whether the value provided by the functionality is real or fake. This renders that definition weak. For instance, consider a protocol with a faking algorithm that outputs the empty string as “fake randomness”. While this protocol should clearly not be considered as “incoercible”, it could be accepted by a simulation based definition — as long as the simulator knows which parties are coerced and which ones are corrupted, since there is no problem for the simulator to output an empty string upon coercion request.

Now, consider a protocol  $\pi$  that realizes such an ideal functionality  $\mathcal{F}$ , and consider a party  $P$  that runs  $\pi$ . Now, upon receiving a corruption message from the adversary,  $\pi$  must instruct  $P$  to first ask the environment (which, again, represents the entity that invoked party  $P$  to participate in the protocol) whether to report the real internal state or to provide a fake one. If the response is to reveal the real input, then we require that  $P$  reveals its true internal state. If the response is to fake input value  $x$ , then  $P$  runs a special procedure **Fake**, which should be specified as part of a protocol description, and reveals the output of **Fake** instead of its true internal state.

We say that  $\pi$  is incoercible if it UC-emulates an incoercible ideal functionality  $\mathcal{F}$ . Such definition captures our intuitive notion of incoercibility: Indeed,  $\mathcal{F}$  provides “ideal incoercibility” in the sense that there the ideal adversary learns nothing about whether a party revealed the real or the fake input - beyond what is revealed by the legitimate outputs of the corrupted parties. Thus, the same must hold also with respect to the real adversary that interacts with  $\pi$  - or else the environment could tell the difference between the two interactions. Note however that this argument hinges on two facts: (a) in the real world the corrupted party must reveal its real input upon corruption, when instructed so by the environment, and (b) that the ideal adversary is not being told whether the input value it received upon corruption is real or fake.

Now let us give a more formal definition. First, we slightly change the model and define a notion of *corruption compliant* protocols, similar in spirit to the notion of compliant protocols in the plain UC. Intuitively, a protocol is corruption compliant if it reveals its true internal state when asked by the adversary. We only consider corruption-compliant protocols in our framework. Below we give a more precise definition.

**Corruption-compliant protocols.** A protocol is corruption-compliant if it consists of two processes, a body and a shell. When a message is written to an input, subroutine output, or communication tape of a party, it is first processed by the shell, which may decide to forward it to the body, possibly modified. Similarly, any message the body decides to send is first processed by the shell. The body doesn’t see the internal state of the shell.

In addition, the shell should behave in a certain way upon receiving a “corrupt P” message. Namely, it notifies the calling entity  $C_P$  of the party about the corruption, and expects to receive further instructions from  $C_P$ .  $C_P$  can reply either with “corrupt” or with “fake to  $\vec{x}^*$ ” message (where  $\vec{x}^*$  is a set of fake inputs and outputs). In the first case the shell should send the whole internal state  $I$  of  $P$  to the adversary. In case it was told to fake to  $\vec{x}^*$ , it runs **Fake**( $I, \vec{x}, \vec{x}^*$ ) on its internal state, true and fake inputs to obtain fake internal state  $I^*$ , and reports  $I^*$  to the adversary.

The second requirement describes what the shell should do upon receiving a notification of corruption from its subprocess. If the shell previously received a “corrupt” message from  $C_P$ , then it sends “corrupt” to its subprocess as well. If the message which the shell received from  $C_P$  was “fake to  $\vec{x}^*$ ”, then the shell sends to its subprocess “fake to  $\vec{y}^*$ ”, where  $\vec{y}^* = \text{Inputs}(\vec{x}^*, I^*)$ , and **Inputs** is a function which determines an input of a subprocess, given an input and randomness of the main party.

Note that if  $\rho$  and  $\pi$  are both corruption-compliant, then so is  $\rho^\pi$ .

**Incoercible functionalities.** We consider ideal functionalities  $\mathcal{F}$  where each input to  $\mathcal{F}$  is associated with two party identities: the first,  $P$ , represents the identity of the protocol participant that holds this input, and the second,  $C_P$ , is the identity of the “calling party”, namely the party that provided the input value(s) to  $P$ , and will obtain the output value(s) from the computation.

Such an ideal functionality  $\mathcal{F}$  is incoercible if it behaves as follows upon receiving a *corrupt*  $P$  message from the adversary.  $\mathcal{F}$  first outputs to  $C_P$  a **corrupted** value. Next, if  $C_P$  responds with **do not fake** then  $\mathcal{F}$  returns to the adversary all the inputs received from  $C_P$  and all the output passed to  $C_P$  in this interaction so far. If  $C_P$  responds with **fake to**  $\vec{x}^*$  then  $\mathcal{F}$  interprets  $\vec{x}^*$  as a list of inputs and outputs and hands this list to the adversary instead of the real one.

**Incoercible protocols.** We now give a definition of incoercible protocols:

**Definition 1.** *A protocol  $\pi$  is incoercible if it is corruption-compliant and it UC-emulates an incoercible ideal functionality  $\mathcal{F}$ .*

We also provide a definition of incoercible distributed function evaluation. Let  $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$  be an  $n$ -party function, and let  $\mathcal{F}_f$  be the incoercible ideal functionality that computes  $f$ , say with respect to some fixed set of party identities  $P_1, \dots, P_n$ . That is, upon receiving inputs from the calling parties of  $P_1, \dots, P_n$ ,  $\mathcal{F}_f$  evaluates  $f$  on these inputs and provides the caller of each  $P_i$  with its corresponding output value  $f(x_1, \dots, x_n)_i$ . Party corruptions are handled in an incoercible way as described above.

**Definition 2.** *Protocol  $\pi$  realizes an  $n$ -party function  $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$  if it UC-emulates  $\mathcal{F}_f$ . In addition, if  $\mathcal{F}_f$  is an incoercible ideal functionality and  $\pi$  is corruption-compliant, we say that  $\pi$  incoercibly realizes  $f$ .*

Note that the above definition of incoercibility did not specify whether the corruptions are honest-but-curious or Byzantine. Indeed, this definition is meaningful in both cases.

It follows from the universal composition theorem [Can01] that incoercible protocols can be composed together:

**Theorem 2.** *Let  $\pi$  be an incoercible protocol realizing  $\mathcal{F}$  given access to  $\mathcal{G}$ , and  $\rho$  be an incoercible protocol realizing  $\mathcal{G}$ . Then  $\rho^{\pi/\mathcal{G}}$  incoercibly realizes  $\mathcal{F}$ .*

This theorem directly follows from composition theorem of [Can01], after noticing that composition of corruption-compliant protocols is corruption-compliant as well.

**Concurrent work.** In a concurrent and independent work of Alwen, Ostrovsky, Zhou and Zikas [AOZZ15] another UC definition of incoercibility is presented. Their framework is called IUC and is built on top of the UC framework (IUC security implies basic UC security). However, unlike us they change the framework (in particular, they introduce a notion of a deception strategy, and the definition of IUC-secure protocol has two additional quantifiers over real and ideal deception strategies.)

They also give a composition theorem for IUC security, which says that two composed IUC-secure protocols result in a IUC-secure protocol.

Although our definition is syntactically different from their, the crux of both definitions is the same: the protocol is incoercible if in the ideal world the simulator can convincely simulate an internal state of the party without knowing whether it is simulating an internal state for real input or for fake input; this means that the real-world adversary cannot distinguish between real and fake inputs as well.

The rigorous comparison of the two definitions is yet to be determined.

### 3 Protocol overview

Let's first recall how the original Yao protocol looks like. Let's say parties  $P_0$  and  $P_1$  have inputs  $x_0$  and  $x_1$  and they want to evaluate  $y = C(x_0, x_1)$  for some circuit  $C$ .  $P_0$  generates a garbled circuit: that is, for every wire of  $C$   $P_0$  creates two random labels  $l_0, l_1$ , and a garbled circuit consists of 4 encryptions of output label under input labels as keys, and the result table, which lists 0 and 1 labels for output gates.

$P_0$  sends to  $P_1$  the garbled circuit together with the labels corresponding to  $P_0$ 's input. Then for every  $P_0$ 's input bit  $P_0$  and  $P_1$  run OT protocol, after which  $P_1$  learns the keys corresponding to his input. At this point  $P_1$  has all information he needs to evaluate the circuit: it has all input labels, and it keeps evaluating the circuit gate by gate, until finally it learns output labels. Then it uses result table to learn the output.

As shown in [LP09], the original Yao protocol is statically secure, given augmented non-committing encryption [DN00, CLOS02]. In particular, when  $P_1$  is corrupted, Simulator learns  $x_1$  and  $y$  and shows a fake garbled circuit which always evaluates to  $y$  and is indistinguishable from the real garbled circuit. (It cannot show the real garbled circuit since it doesn't know  $x_0$ .) Also the simulator shows labels corresponding to  $P_0$ 's and  $P_1$ 's inputs. Here it is crucial that an adversary sees only one label per each input bit and therefore cannot distinguish between a fake circuit and a real one.

The same simulation works in adaptive case with erasures:  $P_0$  should erase its internal state before sending the second message. However, in the adaptive case without erasures this simulation fails: an adversary could corrupt  $P_0$  after corrupting  $P_1$  and learning a fake garbled circuit. For every  $P_1$ 's input bit, a simulator has to show *both* labels since these labels were  $P_0$ 's input in OT protocol. Now the adversary sees one label for each one of  $P_0$ 's input bits and both labels for  $P_1$ 's input bit. This allows the adversary to detect that the garbled circuit is not valid.

Indeed, consider a circuit that consists of just one AND gate. The simulator corrupts  $P_1$  and learns its input  $x_1 = 0$  and  $y = 0$ . At this point the simulator still doesn't know  $P_0$ 's input, but it has to show the garbled circuit, therefore it shows fake circuit where all four ciphertexts encrypt the same key  $l_0$ , and it shows the result table where  $l_0$  is decrypted to 0. Now the simulator corrupts  $P_0$  and learns  $x_0 = 1$ . It has to show keys corresponding to both  $x_1 = 0$  and  $x_1 = 1$ . This means that the adversary knows the keys for  $x_0 = 1$ ,  $x_1 = 0$  and  $x_1 = 1$  and can evaluate the circuit on inputs  $(1, 1)$  and  $(1, 0)$ . Since the circuit is just an AND gate, the result should be different. However, since our garbled circuit contained the same key in all four encryptions, an adversary trying to evaluate the circuit will get 0 in both cases and will detect cheating.

The problem is that an adversary learns too much at the moment of corruption: learning both keys for  $P_1$ 's input allows him to evaluate the circuit on many inputs and to check that the circuit is a fake. To avoid this problem, we change the protocol such that  $P_0$  himself doesn't know the keys for  $P_1$ 's input. In order to achieve this, we "glue together" the garbled circuit generation, the input labels generation and the OT into one program  $P$  which outputs the next message function for the Yao protocol. This program will be obfuscated by the factory. Now,  $P_0$  will run this program on his input and local randomness and send its output to  $P_1$ .

Naively one may hope that, since the program is obfuscated,  $P_0$  himself doesn't know more than just inputs he used and output it sent to  $P_1$  (in particular, it doesn't know the keys for  $P_1$ 's input). However, this is not enough: it might be the case that the input itself reveals the keys (say, if the keys are just set to be some substring of the random input). To deal with this problem, we don't use the random input directly in the protocol. Instead, we first apply a pseudorandom function to



the input and random input, and then use the output of the pseudorandom function as the random input to the protocol.

The next set of challenges deals with making the above plan to work with an obfuscation mechanism that only guarantees indistinguishability obfuscation. Here we follow the lead of Sahai and Waters [SW14] and use similar constructs and techniques as there. Specifically, we use the technique of embedding “hidden triggers” in the random input to the program  $P$ . If the program recognizes a hidden trigger then it just outputs the value encrypted in that trigger. Else, the program used the randomness as in the Yao protocol. We publish  $P$  together with a “faking” algorithm Explain that allows anyone to generate hidden triggers of one’s choice. This addition has a twofold effect: For one it provides for incoercibility for the garbler. In addition it also simplifies the proof of security.

Throughout, and following [SW14], we employ *constrained*, or *puncturable* pseudorandom functions [GGM86, BGI13, BW13], which enables applying indistinguishability obfuscation to pseudorandom function in a meaningful way.

We describe and analyze the scheme in a simple setting where the parties have secure communication channels, and with only honest but curious corruptions. Once we have such a protocol, we can implement secure channels using non-committing encryption. We can also deal with Byzantine corruptions by forcing semi-honest behavior.

We also assume without loss of generality that only the evaluator learns the output. If both parties need to obtain outputs from the computation then they can run the same protocol twice, on the same inputs but with reverse roles. (Alternatively, at the cost of adding a message to the protocol, the evaluator can send the function value to the garbler.)

*Implementing secure channels.* As we will see later, only the second message in our protocol should be sent over a secure channel. This means that  $P_1$  can send  $EK_{NCE}$  in the first message, and the protocol still remains two-round after implementing secure channels.

*Corruption obliviousness and leakage resilience.* The naive protocol, described above, does not naturally lend to corruption-oblivious simulation. Indeed, to simulate the corruption of the garbler, the simulator needs to come up with a second message, namely a garbled circuit, that outputs the correct output of the computation. This needs to be done without knowing the input or output of the evaluator, and only using the input of the garbler. Furthermore, when the evaluator is corrupted, the simulator needs to come up with the *same garbled circuit*, without knowing the input of the garbler. This is not known to be possible in general. We get around this issue by making a simple modification to the protocol: Instead of evaluating  $f(x_0, x_1)$ , the parties will use the above protocol to evaluate  $f'(x_0, (x_1, z)) = f(x_0, x_1) \oplus z$ . The evaluator,  $P_1$ , will choose  $z$  at random, and after obtaining the output value  $y$ , it will set its output to be  $y \oplus z$ .

With this modification in place, the simulator can set the output of the garbled circuit to be a random value fixed in advance and then deal with the corruption of the parties in an oblivious way.

*Incoercibility.* We provide incoercibility for the garbler. This is done in a straightforward way: Since the explain procedure is public, a coerced garbler can demonstrate random input that explains any input value of its choice, in the same way as in [SW14].

*Handling Byzantine corruptions.* Here we use the generic transformation of [CLOS02] (based on [GMW87]) that transforms a protocol that is secure against adaptive honest but curious corruptions into a protocol that is secure against adaptive Byzantine corruptions.

## 4 Detailed description and analysis

**Preliminaries.** In our construction we use the following primitives. The reader is referred to the papers cited for detailed definitions.

1. Indistinguishability obfuscation  $i\mathcal{O}$  for polynomial-size circuits, as defined, constructed and used in [BGI<sup>+</sup>01, GR14, GGH<sup>+</sup>13, SW14].
2. Augmented non-committing encryption scheme  $Enc$  ([DN00, CLOS02]). We denote its generation, oblivious generation and inverting algorithms as  $Enc.Gen$ ,  $Enc.oGen$  and  $Enc.Inv$ .
3. Puncturable PRFs which are additionally extracting or injective [BGI13, BW13, SW14].
4. The garbled circuit generation algorithm  $Gen$  together with an algorithm  $SimGen$  for generating fake garbled circuit from [LP09]. These programs use a special encryption scheme which they call a public key encryption with elusive efficiently verifiable range.

**Deterministic single-party-output functionalities.** First, we recall that it suffices to be able to compute deterministic functionalities: indeed, there exists a standard reduction of any randomized functionality to a deterministic one, given by  $f_{det}((x_0, r_0), (x_1, r_1)) = f_{rand}(x_0, x_1; r_0 \oplus r_1)$ . Second, it is enough to compute functionalities where only one party gets the output (and the other party gets nothing): parties can run in parallel two instances of the protocol with the same input, where in the first execution only the first party generates output and in the second execution only the second party generates output.

In our protocol  $P_0$  is the garbler and  $P_1$  is the evaluator for the Yao protocol. The natural thing to do would be to create a garbled circuit for the functionality they want to compute  $(-; f(x_0, x_1))$ . However, in this case the simulation is not corruption-oblivious.<sup>3</sup> We therefore slightly modify a protocol:  $P_1$  first generates random  $z$ , and  $P_0$  generates a garbled circuit for the function  $f'(x_0, (x_1, z)) = f(x_0, x_1) \oplus z$ . As we'll see, this will suffice for making the simulation corruption-oblivious.

**Oblivious transfer.** We use the following one out of two OT protocol, based on [EGL85]: assume  $P_0$  has  $k_0, k_1$  and  $P_1$  has a bit  $b$ ; we want  $P_1$  to learn  $k_b$ . First,  $P_1$  generates keys  $(EK_b, DK_b)$  and  $EK_{1-b}$  without corresponding decryption key (this encryption scheme, in addition to normal key generation, should have oblivious key generation algorithm which outputs encryption keys without corresponding decryption keys, in such a way that this encryption keys are indistinguishable from normal encryption keys. For this we use augmented non-committing encryption).  $P_1$  sends  $EK_0, EK_1$  to  $P_0$ .  $P_0$  sends back encryptions  $c_0 = Enc(EK_0; k_0)$  and  $c_1 = Enc(EK_1; k_1)$ . Since  $P_1$  has  $DK_b$ , he can decrypt  $k_b = Dec(DK_b; c_b)$ . However, since there is no  $DK_{1-b}$  generated, the second value  $k_{1-b}$  remains unknown to  $P_1$ . Following [CLOS02], we make the OT adaptively secure by using non-committing encryption for the encryption scheme.

With this implementation of OT, the Yao protocol consists of the following two messages:

---

<sup>3</sup>Indeed, for the simulation to be corruption-oblivious, the subroutine for generating  $P_1$ 's internal state should be able to create a fake garbled circuit without knowing  $x_0$ . At the same time, the subroutine for creating  $P_0$  internal state should be able to create (the same) fake garbled circuit without knowing the output  $y$ . It is not clear how to do that for the above "natural" garbling method.

1. First,  $P_1$  generates two sets of encryption keys  $\overline{PK}_0, \overline{PK}_1$  and one set of decryption keys  $\overline{SK}_{x_1}$  (such that for every input bit  $x_1^i$   $P_1$  only knows  $DK_{x_1^i}$ ).  $P_1$  sends  $\overline{PK}_0, \overline{PK}_1$  to  $P_0$ .
2.  $P_0$  generates a garbled circuit GC and sends to  $P_1$  GC, keys for  $P_0$ 's input bits, and keys for all possible  $P_1$ 's input bits encrypted under  $\overline{PK}_0, \overline{PK}_1$  (we will call this a *Yao message*).  $P_1$  decrypts the keys corresponding to its input, and, since it has GC and all input labels, it evaluates the circuit gate by gate.

**Protocol description.** We have parties  $P_0, P_1$  with inputs  $x_0, x_1$  respectively. The protocol for allowing  $P_1$  to learn the value  $f(x_0, x_1)$  for some function  $f$  is described in Figure 1. The referee string consists of programs  $P$  and Explain, described in Figures 2 and 3. The circuit  $C$  that program  $P$  evaluates will be the circuit that computes the function  $f'(x_0, (x_1, z)) = f(x_0, x_1) \oplus z$ . (The value  $z$  will be chosen by  $P_1$  at random as part of the protocol.)

The protocol consists of two rounds. In round one,  $P_1$  (the evaluator) chooses randomness  $s$  and  $z$  and sets  $x'_1 = (x_1, z)$  to be its new input. It samples secret and public keys for oblivious transfer using  $s$  (public keys which do not correspond to  $P_1$ 's input are sampled obliviously).  $P_1$  sends all public keys to  $P_0$ . In the second round  $P_0$  chooses its randomness  $r$  and runs a program  $P$  on its input  $x_0$ , randomness  $r$  and a set of public keys from  $P_1$ . The program  $P$  internally generates new randomness  $u$  and runs the underlying subroutine Gen to generate a Yao message, which becomes the program output.  $P_0$  sends this message to  $P_1$ .  $P_1$  gets the labels for  $x_0$ , decrypts the labels for  $x_1$  and evaluates the circuit, obtaining  $f(x_0, x_1) \oplus z$ . Then  $P_1$  xor's the result with  $z$  and gets the output  $f(x_0, x_1)$ .

The program Explain is not used in the protocol directly. However, it is used in the case when parties want to deny their inputs, as well as in the proof.

**The Protocol:**

1.  $P_1$  chooses random  $z$  and sets  $x'_1 \leftarrow (x_1, z)$ . Then it chooses random  $s$  and generates  $\overline{PK}_{x'_1}, \overline{SK}_{x'_1} \leftarrow Enc.Gen(s[0])$  and  $PK_{1-x'_1} \leftarrow Enc.oGen(s[1])$ . It sets  $\alpha^* \leftarrow \overline{PK}_0, \overline{PK}_1$  and sends  $\alpha^*$  to  $P_0$ .
2.  $P_0$  chooses random  $r^*$ , runs  $\beta^* \leftarrow P(x_0, \alpha^*; r^*)$  and sends  $\beta^*$
3.  $P_1$  evaluates the garbled circuit taken from  $\beta^*$ , using the labels and output table from  $\beta^*$ , and outputs the result xor'ed with  $z$ .

**Figure 1:** Protocol description

We show:

**Theorem 3.** *Let:*

- $SEnc$  be CPA-secure symmetric key encryption scheme with an elusive efficiently verifiable range ([LP09])
- $Enc$  be an augmented non-committing encryption scheme
- $E = \{E_{k_E}\}$  be an extracting puncturable PRF family

### Program P

**inputs:**  $P_0$ 's input  $x$ ,  $P_1$ 's 1-round message  $\alpha$ , randomness  $r = r[1]r[2]$   
 $P(x, \alpha; r)$  :

1. check if  $r$  has encoded value inside:
  - (a)  $M' \leftarrow F_k(r[2]) \oplus r[1]$ ; if  $I_{k_I}(M') \neq r[2]$  then goto 2;
  - (b) parse  $M'$  as  $\beta', x', \alpha', \rho'$ . If  $(x', \alpha') \neq (x, \alpha)$  then goto 2;
  - (c) output  $\beta'$
2. else run Gen:
  - (a)  $u \leftarrow E_{k_E}(x, \alpha, r)$
  - (b) output  $Gen(x, \alpha; u)$

### Program Gen.

**Constants:** circuit  $C$  with  $m$  wires and  $s$  output wires; let's assume that first  $2n$  wires are input wires and last  $s$  wires are output wires

**Input:**  $P_0$ 's input  $x_0$ ;  $P_1$ 's two sets of public keys  $\overline{PK}_0, \overline{PK}_1$ ;  
 randomness  $u = u_1u_2u_3u_4$

$Gen(x_0, \overline{PK}; u)$ :

1. Create labels for wires:  $(k_1^0, k_1^1), \dots, (k_m^0, k_m^1) \leftarrow u_1$
2. Create encryptions of labels:
  - (a) Partition  $u_2$  into  $u_{21}, \dots, u_{2m}$ , and each  $u_{2t}$  into  $u_{2t1}, \dots, u_{2t4}$
  - (b) Partition  $u_3$  into  $u_{31}, \dots, u_{3m}$ , and each  $u_{3t}$  into  $u_{3t1}, \dots, u_{3t4}$
  - (c) For every gate  $t$  in  $C$  create 4 encryptions:
    - if  $t$  is an AND gate:
 
$$GC_t[0, 0] \leftarrow SEnc_{k_i^0}(SEnc_{k_j^0}(k_l^0; u_{2t1}); u_{3t1})$$

$$GC_t[0, 1] \leftarrow SEnc_{k_i^0}(SEnc_{k_j^1}(k_l^0; u_{2t2}); u_{3t2})$$

$$GC_t[1, 0] \leftarrow SEnc_{k_i^1}(SEnc_{k_j^0}(k_l^0; u_{2t3}); u_{3t3})$$

$$GC_t[1, 1] \leftarrow SEnc_{k_i^1}(SEnc_{k_j^1}(k_l^1; u_{2t4}); u_{3t4})$$
    - if  $t$  is an OR gate:
 
$$GC_t[0, 0] \leftarrow SEnc_{k_i^0}(SEnc_{k_j^0}(k_l^0; u_{2t1}); u_{3t1})$$

$$GC_t[0, 1] \leftarrow SEnc_{k_i^0}(SEnc_{k_j^1}(k_l^1; u_{2t2}); u_{3t2})$$

$$GC_t[1, 0] \leftarrow SEnc_{k_i^1}(SEnc_{k_j^0}(k_l^1; u_{2t3}); u_{3t3})$$

$$GC_t[1, 1] \leftarrow SEnc_{k_i^1}(SEnc_{k_j^1}(k_l^1; u_{2t4}); u_{3t4})$$
  - (d) shuffle  $GC_t[0, 0], GC_t[1, 0], GC_t[0, 1], GC_t[1, 1]$
3. Create encryptions of labels for  $P_1$ 's input:
  - (a) Partition  $u_4$  into  $u_{401}, \dots, u_{40n}, u_{411}, \dots, u_{41n}$
  - (b) For all  $i = 1, \dots, n$   $c_i^0 \leftarrow Enc_{PK_0^i}(k_{n+i}^0; u_{40i}), c_i^1 \leftarrow Enc_{PK_1^i}(k_{n+i}^1; u_{41i})$
4. output:
  - (a)  $GC_i[0, 0], GC_i[0, 1], GC_i[1, 0], GC_i[1, 1]$  for  $i = 1..m$  (garbled circuit)

Program Explain

**inputs:** message  $m$  which should be encoded; randomness  $\rho$

$P(m; \rho)$  :

1.  $M \leftarrow m, \text{prg}(\rho)$
2.  $r[2] \leftarrow I_{k_I}(M), \quad r[1] \leftarrow F_k(r[2]) \oplus M$
3. output  $r = r[1]r[2]$

**Figure 3:** Program Explain.

- $I = \{I_{k_I}\}$  be an injective puncturable PRF family
- $F = \{F_k\}$  be a puncturable PRF family
- PRG be an input-doubling PRG
- $i\mathcal{O}$  be indistinguishability obfuscator

then the protocol is adaptively secure with oblivious simulation in the factory model in the presence of semi-honest adversaries given secure channels.

**The choice of parameters.** Since we use different types of PRFs (in particular, extracting PRFs and injective PRFs) in the construction, we must ensure that the lengths of all values fit the requirements for these PRFs. Indeed, as shown in [SW14], there exist:

- injective puncturable PRFs which map  $n(\lambda)$  bits to  $m(\lambda)$  bits where injectivity holds with probability  $1 - 2^{-e(\lambda)}$  (over the choice of a key), as long as  $m(\lambda) \geq 2n(\lambda) + e(\lambda)$ ;
- extracting puncturable PRFs which map  $n(\lambda)$  bits to  $m(\lambda)$  bits for distribution  $X$  with min-entropy  $k(\lambda)$  with statistical distance between  $(k, F_k(X))$  and  $(k, U_m)$  at most  $2^{-e(\lambda)}$ , as long as  $n(\lambda) \geq k(\lambda) \geq m(\lambda) + 2e(\lambda) + 2$ .

Let's recall how we use these PRFs in the computation. Let's denote the lengths of a Yao message  $\beta$  and randomness used to create it  $u$  as  $|\beta|$  and  $|u|$ ; also we denote the length of  $M$  (the hidden value prepared by a simulator and encoded inside randomness) as  $|M|$ . All these lengths are polynomial in security parameter as well as a circuit size and inputs length. We have to choose randomness length to guarantee that both injective and extracting PRFs exist. Recall that randomness  $r$  (denoted as  $er$  in simulated case) consists of two parts  $r[1]$  and  $r[2]$ . Note that the way  $er[1]$ , the first part of randomness, is generated ( $er[1] \leftarrow F_k(er_2) \oplus M$ ) implies that its length is exactly  $|M|$ .

1.  $I_{k_I}$  should be an injective PRF with negligible failure. It takes as input  $M$  and outputs  $er[2]$ . Thus, it should be the case that  $|er[2]| \geq 2|M| + \lambda$ .
2.  $E_{k_E}$  should be an extracting PRF with negligible distance. It takes as input  $(x_0, \overline{PK}, r[1]r[2])$  and outputs  $u$ . We are going to use extracting property when  $r = r[1]r[2]$  is chosen at random, and min-entropy of input is at least  $|r| = |r[1]| + |r[2]|$ . Thus, it should be the case that  $|x_0| + |\overline{PK}| + |r[1]| + |r[2]| \geq |r[1]| + |r[2]| \geq |u| + 2\lambda + 2$ .

Once a security parameter and a circuit are fixed, all values above are also fixed except  $|r[2]|$ . Note that by choosing  $|r[2]|$  large enough (but still polynomial in the security parameter), we can satisfy both inequalities.

*Proof.* The outline of the proof is the following. First, we give a description of our simulator. Then we prove that no environment can distinguish between a real execution and a simulation. We do this in two steps. In step one we deal with the case of non-adaptively chosen inputs; that is, the environment first chooses parties' inputs and only then sees a CRS. In order to show indistinguishability in non-adaptive case, we consider an intermediate middle hybrid where all protocol messages are generated as in a real execution, but the randomness is explained. In two lemmas we prove that this middle hybrid is indistinguishable from both real execution and simulation. In step two we consider the case of adaptive inputs choice, thus proving the theorem statement.

**Simplifying assumptions.** In our honest-but-curious setting we can assume that corruptions happen after the protocol execution and that both parties are corrupted. Since our simulator, as we see later, is corruption-oblivious (information learned in one party corruption is not used in the other party corruption), we don't need to think about different order of corrupting parties. Also we assume secure channels, therefore our simulator has to show the protocol transcript only after one of the parties is corrupted.

In our proofs of lemmas instead of having an interactive game with the adversary we just run an experiment and show to the adversary the resulting distribution, asking it to guess which hybrid it sees. Indeed, by itself the security definition is interactive: an environment first sees a CRS and then outputs inputs; after this, it sees protocol messages. Then it can send corruption requests and get back parties' internal states. Given this information, the adversary chooses which hybrid it sees. However, in the case of non-adaptively chosen inputs, we can use a non-interactive security definition: the inputs are fixed in advance, therefore we can send a CRS later with other values the adversary should see. Next, we assumed that all parties are corrupted, and therefore the adversary doesn't need to send corruption requests; the simulator will send it all parties' internal states itself. Therefore, instead of playing an interactive game with the adversary, in our security definitions the simulator generates all protocol information (protocol messages, parties' internal states) and sends it to the adversary, who should distinguish between hybrids.

**Description of the simulator.** Our simulator is described in Figure 4. It gets a CRS, generates randomness needed ( $s_{PKE}$  to create  $P_1$ 's keys for encryption scheme,  $s_{GC}$  to create a fake garbled circuit, and  $s_y$ , a random value which is the result of  $z \oplus y$  in a real execution), and sets its state to be  $s = (CRS, s_{PKE}, s_{GC}, s_y)$ .

Since we assume secure channels, the simulator doesn't need to show a transcript before corruptions. Upon corruption of a party  $P_i$ , the simulator calls its subroutine  $Sim_{P_i}(CRS, s_{PKE}, s_{GC}, s_y)$  to simulate  $P_i$ 's internal state. Each subroutine has to show randomness used by a party and the communication it sees.  $Sim_{P_i}$  first generates secret and public keys for  $P_1$  and sets  $\alpha^*$  to be  $P_1$ 's public keys (note that since all three programs ( $Sim$ ,  $Sim_{P_0}$  and  $Sim_{P_1}$ ) use the same state to generate values, they get the same result - public keys and garbled circuit). Then it generates a fake garbled circuit and encryptions for OT  $\beta^* \leftarrow SimGen(s_y, \alpha^*, s_{GC})$ . The next step depends on the party. A simulator for  $P_0$  computes explained randomness  $er^* \leftarrow Explain((\beta^*; x_0, \overline{PK}; \rho^*)$  for randomly chosen  $\rho^*$  and shows  $er^*$  (internal state) and  $\alpha^*$  (communication). A simulator for  $P_1$  sets its randomness  $z$  to be consistent with the garbled circuit output and the protocol output (that

is,  $z = y \oplus s_y$ ) and then, using an inversion algorithm, creates randomness  $es^*$ , which produces obliviously sampled keys  $\overline{PK}_{1-x_1}$ . The simulator shows  $es^*$  and  $z$  as  $P_1$ 's internal state and  $\beta^*$  as the communication seen.

Note that to simulate a party during corruption, the simulator doesn't use internal information of the other party; only this party's input/output is used, together with randomness  $s$  which acts as a state of the simulator. Therefore this simulator is corruption oblivious.

**Step one - non-adaptive inputs case.** In the following two lemmas, we prove that real and simulated experiments are indistinguishable. To achieve this we consider a middle hybrid where all protocol messages are generated honestly like in a real execution, but the randomness shown to the adversary is obtained using Explain algorithm. In the first lemma we show that this middle hybrid is indistinguishable from the simulation; indistinguishability between the middle hybrid and a real execution is shown in lemma 2. In both proofs we first give an overview of hybrids, and then present a detailed description with reductions.

*Our notation.* To denote the first and the second part of randomness, we write  $r[1]$  and  $r[2]$ . By  $\overline{PK}$  we denote a set of public keys for each possible input bit of  $P_1$ 's input;  $\overline{PK}_0$  and  $\overline{PK}_1$  mean sets of public keys for input bits 0 and input bits 1. By  $PK_{x_1}$  we mean the set of public keys corresponding to  $P_1$ 's input, that is,  $PK_{x_1} = (PK_{x_1}^1, \dots, PK_{x_1}^n)$ . By  $PK_{1-x_1}$  we mean the opposite set of public keys.

We mark the values obtained in the experiment with a star to distinguish these values from variables in programs. We denote the first round message ( $P_1$ 's public keys) as  $\alpha^*$  and the second round message (a garbled circuit, an output table, labels for  $P_0$ 's input, encrypted labels for all possible  $P_1$ 's inputs) as  $\beta^*$ .

**Lemma 1.** *The results of the following two experiments are indistinguishable:*

**Experiment Simulation:**

1. choose randomness  $s_{PKE}, s_{CRS}, s_{GC}, s_y$ . Set  $z = y \oplus s_y$ . Set  $x'_1 \leftarrow (x_1, z)$
2. generate a CRS: prf keys  $k_E, k_I, k$ , internal keys for Gen, and choose randomness for obfuscation  $x_P, x_{Expl}$  using  $s_{CRS}$ . Create obfuscated programs  $P \leftarrow O(P_{k_E, k_I, k}; Gen; x_P), Explain \leftarrow O(Explain_{k_I, k}; x_{Expl})$ .
3. sample  $P_0$ 's keys  $\overline{PK}_0, \overline{PK}_1, \overline{SK}_0, \overline{SK}_1 \leftarrow PKE.Gen(s_{PKE})$ .  
Set  $\alpha^* \leftarrow \overline{PK}_0, \overline{PK}_1$
4. run  $\beta^* \leftarrow SimGen(s_y, \alpha^*; s_{GC})$
5. choose  $\rho^*$  at random
6.  $er^* \leftarrow Explain(\beta^*; x_0, \alpha^*; \rho^*), es^* \leftarrow Enc.Inv(s_{PKE}, x'_1)$

An adversary sees protocol transcript  $(\alpha^*, \beta^*)$ , internal states  $er^*$  and  $(es^*, z)$ , programs  $(P, Explain)$ .  
and

**Experiment Middle:**

1. choose randomness  $s_{PKE}, s_{CRS}, s_{GC}, s_y$ . Choose random  $z$ . Set  $x'_1 \leftarrow (x_1, z)$

**The simulation:**

1. Obtain the public programs  $CRS = P, Explain$
2. Choose randomness for simulation  $(s_{PKE}, s_{GC}, s_y)$ . Set the state to be  $s = (CRS, s_{PKE}, s_{GC}, s_y)$
3. upon corruption of  $P_0$ : output  $Sim_{P_0}(s)$
4. upon corruption of  $P_1$ : output  $Sim_{P_1}(s)$

$Sim_{P_0}(CRS, s_{PKE}, s_{GC}, s_y)$

1. learn  $x_0$
2. generate  $\overline{PK}_0, \overline{SK}_0, \overline{PK}_1, \overline{SK}_1 \leftarrow Enc.Gen(s_{PKE})$ ; set  $\alpha^* \leftarrow \overline{PK}_0, \overline{PK}_1$
3. set  $\beta^* \leftarrow SimGen(s_y, \overline{PK}; s_{GC})$
4. choose random  $\rho^*$  and set  $er^* \leftarrow Explain(\beta^*; x_0, \overline{PK}; \rho^*)$
5. output  $(er^*, \alpha^*)$

$Sim_{P_1}(CRS, s_{PKE}, s_{GC}, s_y)$

1. learn  $x_1, y$
2. generate  $\overline{PK}_0, \overline{SK}_0, \overline{PK}_1, \overline{SK}_1 \leftarrow Enc.Gen(s_{PKE})$
3. set  $\beta^* \leftarrow SimGen(s_y, \overline{PK}; s_{GC})$
4. set  $z \leftarrow s_y \oplus y, x'_1 \leftarrow (x_1, z)$
5. set  $es^* \leftarrow Enc.Inv(s, x'_1)$
6. output  $(es^*, z; \beta^*)$

**Figure 4:** Simulation



2. generate a CRS: prf keys  $k_E, k_I, k$ , Gen internal keys and choose randomness for obfuscation  $x_P, x_{Expl}$  using  $s_{CRS}$ . Create obfuscated programs  $P \leftarrow O(P_{k_E, k_I, k}; Gen; x_P), Explain \leftarrow O(Explain_{k_I, k}; x_{Expl})$ .
3. sample  $P_0$ 's keys  $\overline{PK}_{x'_1}, \overline{SK}_{x'_1} \leftarrow PKE.Gen(s_{PKE}[0]), \overline{PK}_{1-x'_1} \leftarrow PKE.oGen(s_{PKE}[1])$ . Set  $\alpha^* \leftarrow \overline{PK}_0, \overline{PK}_1$
4. choose random  $r^*$
5. run  $\beta^* \leftarrow P(x_0, \alpha^*; r^*)$
6. choose  $\rho^*$  at random
7.  $er^* \leftarrow Explain(\beta^*; x_0, \alpha^*; \rho^*)$

An adversary sees protocol transcript  $(\alpha^*, \beta^*)$ , internal states  $er^*$  and  $(s_{PKE}, z)$ , programs  $(P, Explain)$ .

*Proof.* We show indistinguishability using several hybrids as described below:

1. H0 = Simulation
2. H1: like a simulation, but OT public keys  $\overline{PK}_{1-x_1}$  (which do not correspond to  $P_1$ 's input) are sampled obliviously
3. H2: like H1, but  $\beta^*$  is chosen as a result of  $Gen(x_0; \alpha^*; u^*)$  for some random  $u^*$ ; previously  $\beta^*$  was the result of  $SimGen$ . Based on indistinguishability between a fake and a real garbled circuit.
4. H3: Like H2, but  $u^*$  is chosen as  $E_{k_E}(x_0, \alpha^*, r^*)$  for random  $r^*$ ; previously it was chosen at random. Based on extracting property of  $E_{k_E}$
5. H4 = Middle: Like H3, but  $\beta^* \leftarrow P(x_0, \alpha^*; r^*)$  (which means that now first check 1 is performed on randomness  $r^*$  before generating the output). Based on the fact that  $r^*$  is random and for a random value this check passes with negligible probability.

### H1.

1. choose randomness  $s_{PKE}, s_{CRS}, s_{GC}, s_y$ . Set  $z = y \oplus s_y$ . Set  $x'_1 \leftarrow (x_1, z)$
2. generate a CRS: prf keys  $k_E, k_I, k$ , Gen internal keys and choose randomness for obfuscation  $x_P, x_{Expl}$  using  $s_{CRS}$ . Create obfuscated programs  $P \leftarrow O(P_{k_E, k_I, k}; Gen; x), Explain \leftarrow O(Explain_{k_I, k}; x_{Expl})$ .
3. sample  $P_0$ 's keys  $\overline{PK}_{x'_1}, \overline{SK}_{x'_1} \leftarrow PKE.Gen(s_{PKE}[0]), \overline{PK}_{1-x'_1} \leftarrow PKE.oGen(s_{PKE}[1])$ . Set  $\alpha^* \leftarrow \overline{PK}_0, \overline{PK}_1$
4. run  $\beta^* \leftarrow SimGen(s_y, \alpha^*; s_{GC})$
5. choose  $\rho^*$  at random
6.  $er^* \leftarrow Explain(\beta^*; x_0, \alpha^*; \rho^*)$

An adversary sees protocol transcript  $(\alpha^*, \beta^*)$ , internal states  $er^*$  and  $(s_{PKE}, z)$ , programs  $(P, Explain)$ .

In this hybrid we generate public keys for OT which do not correspond to  $P_1$ ' input obliviously and show to the adversary the real randomness  $s_{PKE}$  which was used to generate these keys. Indistinguishability holds because of the property of augmented non-committing encryption: no adversary can distinguish between a real randomness used for oblivious key generation and a randomness obtained as a result of inverting algorithm.

**H2.**

1. choose randomness  $s_{PKE}, s_{CRS}, s_{GC}, s_y$ . Choose random  $z$ . Set  $x'_1 \leftarrow (x_1, z)$
2. generate a CRS: prf keys  $k_E, k_I, k$ , Gen internal keys and choose randomness for obfuscation  $x_P, x_{Expl}$  using  $s_{CRS}$ . Create obfuscated programs  $P \leftarrow O(P_{k_E, k_I, k}; Gen; x_P), Explain \leftarrow O(Explain_{k_I, k}; x_{Expl})$ .
3. sample  $P_0$ 's keys  $\overline{PK}_{x'_1}, \overline{SK}_{x'_1} \leftarrow PKE.Gen(s_{PKE}[0]), \overline{PK}_{1-x'_1} \leftarrow PKE.oGen(s_{PKE}[1])$ . Set  $\alpha^* \leftarrow \overline{PK}_0, \overline{PK}_1$
4. choose random  $u^*$
5. run  $\beta^* \leftarrow Gen(x_0, \alpha^*; u^*)$
6. choose  $\rho^*$  at random
7.  $er^* \leftarrow Explain(\beta^*; x_0, \alpha^*; \rho^*)$

An adversary sees protocol transcript  $(\alpha^*, \beta^*)$ , internal states  $er^*$  and  $(s_{PKE}, z)$ , programs  $(P, Explain)$ .

In this hybrid we changed the way  $\beta^*$  is generated. Previously it contained a fake garbled circuit which always evaluates to  $s_y$ , now it contains a real garbled circuit. Indistinguishability is based on indistinguishability between a fake garbled circuit and a real one, as shown in [LP09].

**H3.**

1. choose randomness  $s_{PKE}, s_{CRS}, s_{GC}, s_y$ . Choose random  $z$ . Set  $x'_1 \leftarrow (x_1, z)$
2. generate a CRS: prf keys  $k_E, k_I, k$ , Gen internal keys and choose randomness for obfuscation  $x_P, x_{Expl}$  using  $s_{CRS}$ . Create obfuscated programs  $P \leftarrow O(P_{k_E, k_I, k}; Gen; x_P), Explain \leftarrow O(Explain_{k_I, k}; x_{Expl})$ .
3. sample  $P_0$ 's keys  $\overline{PK}_{x'_1}, \overline{SK}_{x'_1} \leftarrow PKE.Gen(s_{PKE}[0]), \overline{PK}_{1-x'_1} \leftarrow PKE.oGen(s_{PKE}[1])$ . Set  $\alpha^* \leftarrow \overline{PK}_0, \overline{PK}_1$
4. choose random  $r^*$ . Set  $u^* \leftarrow E_{k_E}(x_0, \alpha^*, r^*)$
5. run  $\beta^* \leftarrow Gen(x_0, \alpha^*; u^*)$
6. choose  $\rho^*$  at random
7.  $er^* \leftarrow Explain(\beta^*; x_0, \alpha^*; \rho^*)$

An adversary sees protocol transcript  $(\alpha^*, \beta^*)$ , internal states  $er^*$  and  $(s_{PKE}, z)$ , programs  $(P, Explain)$ .

In this hybrid we choose  $u^*$  as  $u^* \leftarrow E_{k_E}(x_0, \alpha^*, r^*)$ , instead of choosing it at random. Indistinguishability holds because of extracting property of  $E_{k_E}$ . Indeed, since min-entropy of the PRF input is at least  $|r^*|$ , then by our choice of parameters the output of this PRF is indistinguishable from random. We can reduce these hybrids to an extracting prf game as follows: given  $k_E$  and random  $w$  or  $w = E_{k_E}(x_0, \alpha^*, r^*)$  for random  $r^*$ , we choose other keys and obfuscate programs, and then compute other variables using  $u^* = w$ . Depending on whether  $w$  is random or not, we are either in H2 or in H3.

**H4 (Middle).**

1. choose randomness  $s_{PKE}, s_{CRS}, s_{GC}, s_y$ . Choose random  $z$ . Set  $x'_1 \leftarrow (x_1, z)$
2. generate a CRS: prf keys  $k_E, k_I, k$ , Gen internal keys and choose randomness for obfuscation  $x_P, x_{Expl}$  using  $s_{CRS}$ . Create obfuscated programs  $P \leftarrow O(P_{k_E, k_I, k}; Gen; x_P), Explain \leftarrow O(Explain_{k_I, k}; x_{Expl})$ .
3. sample  $P_0$ 's keys  $\overline{PK}_{x'_1}, \overline{SK}_{x'_1} \leftarrow PKE.Gen(s_{PKE}[0]), \overline{PK}_{1-x'_1} \leftarrow PKE.oGen(s_{PKE}[1])$ . Set  $\alpha^* \leftarrow \overline{PK}_0, \overline{PK}_1$
4. choose random  $r^*$ .
5. run  $\beta^* \leftarrow P(x_0, \alpha^*; r^*)$
6. choose  $\rho^*$  at random
7.  $er^* \leftarrow Explain(\beta^*; x_0, \alpha^*; \rho^*)$

An adversary sees protocol transcript  $(\alpha^*, \beta^*)$ , internal states  $er^*$  and  $(s_{PKE}, z)$ , programs  $(P, Explain)$ .

In this hybrid we generate  $\beta^*$  as a result of a program  $P$ . In other words, before computing  $u^*$  we perform check 1 in  $P$ . Since for randomly chosen  $r^*$  this check passes with negligible probability, hybrids are statistically close to each other.

Thus lemma 1 is proved. □

**Lemma 2.** *No PPT adversary can distinguish between the following two distributions:*

**Experiment Middle:**

1. choose randomness  $s_{PKE}, s_{CRS}, s_y$ . Choose random  $z$ . Set  $x'_1 \leftarrow (x_1, z)$
2. generate a CRS: prf keys  $k_E, k_I, k$ , Gen internal keys and choose randomness for obfuscation  $x_P, x_{Expl}$  using  $s_{CRS}$ . Create obfuscated programs  $P \leftarrow O(P_{k_E, k_I, k}; Gen; x_P), Explain \leftarrow O(Explain_{k_I, k}; x_{Expl})$ .
3. sample  $P_0$ 's keys  $\overline{PK}_{x'_1}, \overline{SK}_{x'_1} \leftarrow PKE.Gen(s_{PKE}[0]), \overline{PK}_{1-x'_1} \leftarrow PKE.oGen(s_{PKE}[1])$ . Set  $\alpha^* \leftarrow \overline{PK}_0, \overline{PK}_1$
4. choose random  $r^*$
5. run  $\beta^* \leftarrow P(x_0, \alpha^*; r^*)$

6. choose  $\rho^*$  at random
7.  $er^* \leftarrow Explain(\beta^*; x_0, \alpha^*; \rho^*)$

An adversary sees  $(\alpha^*, \beta^*, er^*, s_{PKE}, z)$ , programs  $(P, Explain)$ .

### Experiment Real:

1. choose randomness  $s_{PKE}, s_{CRS}, s_y$ . Choose random  $z$ . Set  $x'_1 \leftarrow (x_1, z)$
2. generate a CRS: prf keys  $k_E, k_I, k$ , Gen internal keys and choose randomness for obfuscation  $x_P, x_{Expl}$  using  $s_{CRS}$ . Create obfuscated programs  $P \leftarrow O(P_{k_E, k_I, k}; Gen; x_P), Explain \leftarrow O(Explain_{k_I, k}; x_{Expl})$ .
3. sample  $P_0$ 's keys  $\overline{PK}_{x'_1}, \overline{SK}_{x'_1} \leftarrow PKE.Gen(s_{PKE}[0]), \overline{PK}_{1-x'_1} \leftarrow PKE.oGen(s_{PKE}[1])$ . Set  $\alpha^* \leftarrow \overline{PK}_0, \overline{PK}_1$
4. choose random  $r^*$
5. run  $\beta^* \leftarrow P(x_0, \alpha^*; r^*)$

An adversary sees  $(\alpha^*, \beta^*, r^*, s_{PKE}, z)$ , programs  $(P, Explain)$ .

*Proof.* The lemma states that the view of an adversary in the real execution is indistinguishable from its view in the experiment when instead of real randomness, explained randomness is shown (which we called a middle experiment). To prove the lemma statement, we consider a sequence of hybrids  $Real = H_0^0 \sim \dots \sim H_6^0 \sim H_6^1 \sim \dots \sim H_0^1 = Middle$ . For  $b = 0, 1$  we will show that  $H_0^b$  is indistinguishable from  $H_6^b$ . After this, we show that  $H_6^0$  and  $H_6^1$  are indistinguishable as well. This proves that a middle hybrid and a real execution are indistinguishable.

Hybrids overview:

1. In  $H1^b$  we skip check 1 in the program  $P$  and directly compute  $u^* \leftarrow E_{k_E}(x_0, \alpha^*; r^*), \beta^* \leftarrow Gen(x_0, \alpha^*; u^*)$ . Since  $r^*$  is random, the check passes with negligible probability.
2. In  $H2^b$ , instead of computing  $\hat{\rho}^* \leftarrow prg(\rho^*)$  (and then evaluating  $er^*$  using this  $\hat{\rho}^*$ ), we choose  $\hat{\rho}^*$  at random. Indistinguishability is based on security of a PRG.
3. In  $H3^b$  we show punctured programs  $P : 1$  and  $Explain : 1$  instead of original ones. We prove that new programs have the same functionality and rely the indistinguishability on the security of  $i\mathcal{O}$ .
4. In  $H4^b$  we choose  $u^*$  at random instead of  $E_{k_E}(x_0^*, \alpha^*; r^*)$ . Based on punctured PRF  $E_{k_E}$ .
5. In  $H5^b$  we choose  $er^*[2]$  at random instead of  $I_{k_I}(\beta^*; x_0, \alpha^*; \hat{\rho}^*)$ . Based on punctured PRF  $I_{k_I}$ .
6. In  $H6^b$  we choose  $er^*[1]$  at random instead of  $F_k(er^*[2]) \oplus (\beta^*; x_0, \alpha^*; \hat{\rho}^*)$ . Based on punctured PRF  $F_k$ .

$H0^b$

1. choose randomness  $s_{PKE}, s_{CRS}, s_y$ . Choose random  $z$ . Set  $x'_1 \leftarrow (x_1, z)$

2. generate a CRS: prf keys  $k_E, k_I, k$ , Gen internal keys and choose randomness for obfuscation  $x_P, x_{Expl}$  using  $s_{CRS}$ . Create obfuscated programs  $P \leftarrow O(P_{k_E, k_I, k}; Gen; x_P), Explain \leftarrow O(Explain_{k_I, k}; x_{Expl})$ .
3. sample  $P_0$ 's keys  $\overline{PK}_{x'_1}, \overline{SK}_{x'_1} \leftarrow PKE.Gen(s_{PKE}[0]), \overline{PK}_{1-x'_1} \leftarrow PKE.oGen(s_{PKE}[1])$ . Set  $\alpha^* \leftarrow \overline{PK}_0, \overline{PK}_1$
4. choose random  $r^*$
5. run  $\beta^* \leftarrow P(x_0, \alpha^*; r^*)$
6. choose  $\rho^*$  at random
7.  $er^* \leftarrow Explain(\beta^*; x_0, \alpha^*; \rho^*)$

If  $b = 0$ , an adversary sees  $(\alpha^*, \beta^*, r^*, s_{PKE}, z)$ , programs  $(P, Explain)$ .

If  $b = 1$ , an adversary sees  $(\alpha^*, \beta^*, er^*, s_{PKE}, z)$ , programs  $(P, Explain)$ .

$H1^b$

1. choose randomness  $s_{PKE}, s_{CRS}, s_y$ . Choose random  $z$ . Set  $x'_1 \leftarrow (x_1, z)$
2. generate a CRS: prf keys  $k_E, k_I, k$ , Gen internal keys and choose randomness for obfuscation  $x_P, x_{Expl}$  using  $s_{CRS}$ . Create obfuscated programs  $P \leftarrow O(P_{k_E, k_I, k}; Gen; x_P), Explain \leftarrow O(Explain_{k_I, k}; x_{Expl})$ .
3. sample  $P_0$ 's keys  $\overline{PK}_{x'_1}, \overline{SK}_{x'_1} \leftarrow PKE.Gen(s_{PKE}[0]), \overline{PK}_{1-x'_1} \leftarrow PKE.oGen(s_{PKE}[1])$ . Set  $\alpha^* \leftarrow \overline{PK}_0, \overline{PK}_1$
4. choose random  $r^*, u^* \leftarrow E_{k_E}(x_0, \alpha^*; r^*)$ ,
5.  $\beta^* \leftarrow Gen(x_0, \alpha^*; u^*)$ .
6. choose  $\rho^*$  at random
7.  $er^* \leftarrow Explain(\beta^*; x_0, \alpha^*; \rho^*)$

If  $b = 0$ , an adversary sees  $(\alpha^*, \beta^*, r^*, s_{PKE}, z)$ , programs  $(P, Explain)$ .

If  $b = 1$ , an adversary sees  $(\alpha^*, \beta^*, er^*, s_{PKE}, z)$ , programs  $(P, Explain)$ .

In this hybrid we omit check 1 in the program P while computing  $\beta^*$ . Since for randomly chosen  $r^*$  the check passes with negligible probability, hybrids are statistically close.

$H2^b$

1. choose randomness  $s_{PKE}, s_{CRS}, s_y$ . Choose random  $z$ . Set  $x'_1 \leftarrow (x_1, z)$
2. generate a CRS: prf keys  $k_E, k_I, k$ , Gen internal keys and choose randomness for obfuscation  $x_P, x_{Expl}$  using  $s_{CRS}$ . Create obfuscated programs  $P \leftarrow O(P_{k_E, k_I, k}; Gen; x_P), Explain \leftarrow O(Explain_{k_I, k}; x_{Expl})$ .
3. sample  $P_0$ 's keys  $\overline{PK}_{x'_1}, \overline{SK}_{x'_1} \leftarrow PKE.Gen(s_{PKE}[0]), \overline{PK}_{1-x'_1} \leftarrow PKE.oGen(s_{PKE}[1])$ . Set  $\alpha^* \leftarrow \overline{PK}_0, \overline{PK}_1$
4. choose random  $r^*, u^* \leftarrow E_{k_E}(x_0, \alpha^*; r^*)$ ,

5.  $\beta^* \leftarrow \text{Gen}(x_0, \alpha^*; u^*)$ .
6. choose  $\hat{\rho}^*$  at random
7. set  $M^* \leftarrow \beta^*; x_0, \alpha^*; \hat{\rho}^*$
8.  $er^*[2] \leftarrow I_{k_I}(M^*)$
9.  $er^*[1] \leftarrow F_k(er^*[2]) \oplus M^*$

If  $b = 0$ , an adversary sees  $(\alpha^*, \beta^*, r^*, s_{PKE}, z)$ , programs  $(P, \text{Explain})$ .

If  $b = 1$ , an adversary sees  $(\alpha^*, \beta^*, er^*, s_{PKE}, z)$ , programs  $(P, \text{Explain})$ .

In this hybrid we use randomly chosen  $\hat{\rho}^*$  instead of the result of applying a PRG to  $\rho^*$  while generating  $er^*$ . Indistinguishability of hybrids immediately follows from the security of a PRG.

$H\mathfrak{Z}^b$

1. choose randomness  $s_{PKE}, s_{CRS}, s_y$ . Choose random  $z$ . Set  $x'_1 \leftarrow (x_1, z)$
2. generate a CRS: prf keys  $k_E, k_I, k$ , Gen internal keys and choose randomness for obfuscation  $x_P, x_{Expl}$  using  $s_{CRS}$ . Create obfuscated programs  $P \leftarrow O(P_{k_E, k_I, k}; \text{Gen}; x_P)$ ,  $\text{Explain} \leftarrow O(\text{Explain}_{k_I, k}; x_{Expl})$ .
3. sample  $P_0$ 's keys  $\overline{PK}_{x'_1}, \overline{SK}_{x'_1} \leftarrow PKE.Gen(s_{PKE}[0])$ ,  $\overline{PK}_{1-x'_1} \leftarrow PKE.oGen(s_{PKE}[1])$ . Set  $\alpha^* \leftarrow \overline{PK}_0, \overline{PK}_1$
4. choose random  $r^*, u^* \leftarrow E_{k_E}(x_0, \alpha^*; r^*)$ ,
5.  $\beta^* \leftarrow \text{Gen}(x_0, \alpha^*; u^*)$ .
6. choose  $\hat{\rho}^*$  at random
7. set  $M^* \leftarrow \beta^*; x_0, \alpha^*; \hat{\rho}^*$
8.  $er^*[2] \leftarrow I_{k_I}(M^*)$
9.  $er^*[1] \leftarrow F_k(er^*[2]) \oplus M^*$

If  $b = 0$ , an adversary sees  $(\alpha^*, \beta^*, r^*, s_{PKE}, z)$ , programs  $(P : 1, \text{Explain} : 1)$ .

If  $b = 1$ , an adversary sees  $(\alpha^*, \beta^*, er^*, s_{PKE}, z)$ , programs  $(P : 1, \text{Explain} : 1)$ .

In this hybrid we show punctured programs  $P : 1$  (Fig. 6) and  $\text{Explain} : 1$  (Fig. 7) instead of their normal versions. We rely the indistinguishability on  $i\mathcal{O}$  security: modified programs have the same functionality as original ones, as proven in [SW14] in their proof for deniable encryption scheme (with a natural modification of the input from their input  $m, r$  to our input  $(x_0, \overline{PK}, r)$ ). However, for the sake of self-containment we briefly sketch it here:

Program P:

1. we add a line “if  $(x, \alpha, r) = (x_0, \alpha^*, r^*)$  or  $(x, \alpha, r) = (x_0, \alpha^*, er^*)$  then output  $\beta^*$ ”, this is exactly what the original program outputs on these inputs.

2. add “f  $r[2] = r^*[2]$  or  $r[2] = er^*[2]$  then goto 2”. If  $r[2] = r^*[2]$ , then the check in step one will not pass since a random  $r^*[2]$  with high probability is outside the image of  $I_{k_I}$ , so we can go to step 2. If  $r[2] = er^*[2]$ , then either the check doesn’t pass and we can go to step 2, or, if it passes, then the encoded message  $M' = M^*$  (due to injectivity of  $I_{k_I}$ ), and therefore  $r[1] = er^*[1]$ ,  $(x', \alpha') = (x_0, \alpha^*)$ , which would be detected in the first added line in P:1.
3. now  $F_k$  is never called on  $r^*[2]$  or  $er^*[2]$ , therefore we can safely puncture at these points.
4. add “if  $M' = M^*$  then goto 2”. If  $M' = M^*$  and the check passes, then  $r[2] = er^*[2]$ ,  $r[1] = er^*[1]$ , and this would be detected in the first line in P:1.
5. now  $I_{k_I}$  will not be called on  $M^*$ , and we can puncture at this point.
6. we can puncture  $F_{k_1\{(x_0, \alpha^*, r^*), (x_0, \alpha^*, er^*)\}}$ , since these inputs are treated in the first line of P:1.

Program Explain:

1. we puncture  $k_I$  at  $M^*$ , since  $\hat{\rho}^*$  (which is a part of  $M^*$ ) is generated at random (instead of  $prg(\rho^*)$ ) and with high probability is outside the image of a PRG; therefore no input results in  $M = M^*$  in Explain.
2. we puncture  $k$  at both points  $r^*[2]$  and  $er^*[2]$ . Since  $r^*[2]$  is randomly chosen, with high probability it is outside the image of a PRF  $I_{k_I}$ , therefore no input for Explain results in  $r[2] = r^*[2]$  and therefore  $F_k$  is never called on  $r^*[2]$ . Furthermore, as we said no input for Explain results in  $M = M^*$ , and due to  $I_{k_I}$  injectivity no input for Explain results in  $er^*[2] = I_{k_I}(M^*)$ , which means that  $F_k$  is not called on  $er^*[2]$  as well.

$H4^b$

1. choose randomness  $s_{PKE}, s_{CRS}, s_y$ . Choose random  $z$ . Set  $x'_1 \leftarrow (x_1, z)$
2. generate a CRS: prf keys  $k_E, k_I, k$ , Gen internal keys and choose randomness for obfuscation  $x_P, x_{Expl}$  using  $s_{CRS}$ . Create obfuscated programs  $P : 1 \leftarrow O(P : 1_{k_E, k_I, k}; Gen; x_P), Explain : 1 \leftarrow O(Explain : 1_{k_I, k}; x_{Expl})$ .
3. sample  $P_0$ 's keys  $\overline{PK}_{x'_1}, \overline{SK}_{x'_1} \leftarrow PKE.Gen(s_{PKE}[0]), \overline{PK}_{1-x'_1} \leftarrow PKE.oGen(s_{PKE}[1])$ . Set  $\alpha^* \leftarrow \overline{PK}_0, \overline{PK}_1$
4. choose random  $u^*, r^*$
5.  $\beta^* \leftarrow Gen(x_0, \alpha^*; u^*)$ .
6. choose  $\hat{\rho}^*$  at random
7. set  $M^* \leftarrow \beta^*; x_0, \alpha^*; \hat{\rho}^*$
8.  $er^*[2] \leftarrow I_{k_I}(M^*)$
9.  $er^*[1] \leftarrow F_k(er^*[2]) \oplus M^*$

If  $b = 0$ , an adversary sees  $(\alpha^*, \beta^*, r^*, s_{PKE}, z)$ , programs  $(P : 1, Explain : 1)$ .

If  $b = 1$ , an adversary sees  $(\alpha^*, \beta^*, er^*, s_{PKE}, z)$ , programs  $(P : 1, Explain : 1)$ .

In this hybrid we choose  $u^*$  at random instead of choosing it as  $E_{k_E}(x_k, \alpha_{1-k}^*, r^*)$ . Security follows from pseudorandomness of a puncturable PRF. Indeed, given a punctured key  $k_E\{(x_k, \alpha_{1-k}^*, r^*)\}$  and  $w$ , which is random or  $E_{k_E}(x_k, \alpha_{1-k}^*, r^*)$ , we choose other keys ourselves and create programs. Then we evaluate variables in the experiment setting  $u^* = w$  and showing the resulting distribution to the adversary. If  $w$  was random, then the adversary sees  $H_4^b$ , otherwise  $H_3^b$ .

$H5^b$

1. choose randomness  $s_{PKE}, s_{CRS}, s_y$ . Choose random  $z$ . Set  $x'_1 \leftarrow (x_1, z)$
2. generate a CRS: prf keys  $k_E, k_I, k$ , Gen internal keys and choose randomness for obfuscation  $x_P, x_{Expl}$  using  $s_{CRS}$ . Create obfuscated programs  $P : 1 \leftarrow O(P : 1_{k_E, k_I, k}; Gen; x_P), Explain : 1 \leftarrow O(Explain : 1_{k_I, k}; x_{Expl})$ .
3. sample  $P_0$ 's keys  $\overline{PK}_{x'_1}, \overline{SK}_{x'_1} \leftarrow PKE.Gen(s_{PKE}[0]), \overline{PK}_{1-x'_1} \leftarrow PKE.oGen(s_{PKE}[1])$ . Set  $\alpha^* \leftarrow \overline{PK}_0, \overline{PK}_1$
4. choose random  $u^*, r^*$
5.  $\beta^* \leftarrow Gen(x_0, \alpha^*; u^*)$ .
6. choose  $\hat{\rho}^*$  at random
7. set  $M^* \leftarrow \beta^*; x_0, \alpha^*; \hat{\rho}^*$
8. choose random  $er^*[2]$
9.  $er^*[1] \leftarrow F_k(er^*[2]) \oplus M^*$

If  $b = 0$ , an adversary sees  $(\alpha^*, \beta^*, r^*, s_{PKE}, z)$ , programs  $(P : 1, Explain : 1)$ .

If  $b = 1$ , an adversary sees  $(\alpha^*, \beta^*, er^*, s_{PKE}, z)$ , programs  $(P : 1, Explain : 1)$ .

In this hybrid we choose  $er^*[2]$  at random instead of choosing it as  $I_{k_I}(M^*)$ . Security follows from pseudorandomness of a puncturable PRF. Indeed, given a punctured key  $k_I\{M^*\}$  and  $w$ , which is random or  $I_{k_I}(M^*)$ , we choose other keys ourselves and create programs. Then we evaluate variables in the experiment setting  $er^*[2] = w$  and showing the resulting distribution to the adversary. If  $w$  was random, then the adversary sees  $H_5^b$ , otherwise  $H_4^b$ .

$H6^b$

1. choose randomness  $s_{PKE}, s_{CRS}, s_y$ . Choose random  $z$ . Set  $x'_1 \leftarrow (x_1, z)$
2. generate a CRS: prf keys  $k_E, k_I, k$ , Gen internal keys and choose randomness for obfuscation  $x_P, x_{Expl}$  using  $s_{CRS}$ . Create obfuscated programs  $P \leftarrow O(P_{k_E, k_I, k}; Gen; x_P), Explain \leftarrow O(Explain_{k_I, k}; x_{Expl})$ .
3. sample  $P_0$ 's keys  $\overline{PK}_{x'_1}, \overline{SK}_{x'_1} \leftarrow PKE.Gen(s_{PKE}[0]), \overline{PK}_{1-x'_1} \leftarrow PKE.oGen(s_{PKE}[1])$ . Set  $\alpha^* \leftarrow \overline{PK}_0, \overline{PK}_1$
4. choose random  $u^*, r^*$
5.  $\beta^* \leftarrow Gen(x_0, \alpha^*; u^*)$ .



6. choose  $\hat{\rho}^*$  at random
7. set  $M^* \leftarrow \beta^*; x_0, \alpha^*; \hat{\rho}^*$
8. choose random  $er^*[2]$
9. choose random  $er^*[1]$

If  $b = 0$ , an adversary sees  $(\alpha^*, \beta^*, r^*, s_{PKE}, z)$ , programs  $(P : 1, Explain : 1)$ .

If  $b = 1$ , an adversary sees  $(\alpha^*, \beta^*, er^*, s_{PKE}, z)$ , programs  $(P : 1, Explain : 1)$ .

In this hybrid we choose  $er^*[1]$  at random instead of choosing it as  $F_k(er^*[2]) \oplus M$ . Security follows from pseudorandomness of a puncturable PRF. Indeed, given a punctured key  $k\{er^*[2]\}$  and  $w$ , which is random or  $F_k(M^*)$ , we choose other keys ourselves and create programs. Then we evaluate variables in the experiment setting  $er^*[2] = w$  and showing the resulting distribution to the adversary. If  $w$  was random, then the adversary sees  $H_6^b$ , otherwise  $H_5^b$ .

Finally we notice that distributions  $H_6^0$  and  $H_6^1$  are the same, since both programs and the experiment treat  $r^*$  and  $er^*$  in the same manner (i.e. both  $r^*$  and  $er^*$  are chosen at random and are not connected to other variables in the protocol). Therefore no adversary can distinguish between these two hybrids, and lemma statement is proved.  $\square$

**Step two - dealing with adaptive inputs.** In order to deal with adaptively chosen inputs, we guess which inputs will be chosen by the adversary. Let us denote by  $l$  the internal state of the parties. Define the variables of the real experiment as  $CRS^r \leftarrow \text{GenCRS}$ ,  $l^r \leftarrow U_n$ ,  $\text{transcript}^r \leftarrow \pi(x_0, x_1, l)$ ; also define the variables of the ideal experiment as  $(\text{transcript}^s, CRS^s) \leftarrow \text{Sim}$ ,  $l^s \leftarrow \text{Sim}(x_0, x_1)$ . Here  $(x_0, x_1) \leftarrow \text{Adv}_1(\text{CRS})$  is the result of running an adversary on a given  $\text{CRS}$ .<sup>4</sup> Then the advantage of the adaptive adversary is

$$\begin{aligned}
& \Pr[\text{Adv}_2(\text{CRS}^s, \text{transcript}^s, l^s, x_0, x_1) = 1] - \Pr[\text{Adv}_2(\text{CRS}^r, \text{transcript}^r, l^r, x_0, x_1) = 1] \leq \\
& \leq \sum_{x_0^*, x_1^* \in \{0,1\}^n} (\Pr[\text{Adv}_2(\text{CRS}^s, \text{transcript}^s, l^s, x_0^*, x_1^*) = 1] - \Pr[\text{Adv}_2(\text{CRS}^r, \text{transcript}^r, l^r, x_0^*, x_1^*) = 1]) \\
& \leq 2^{2n} \cdot \max_{x_0^*, x_1^*} (\Pr[\text{Adv}_2(\text{CRS}^s, \text{transcript}^s, l^s, x_0^*, x_1^*) = 1] - \Pr[\text{Adv}_2(\text{CRS}^r, \text{transcript}^r, l^r, x_0^*, x_1^*) = 1]).
\end{aligned}$$

If  $\text{iO}$  and PRFs are subexponentially secure, then the advantage of the adversary is negligible in the security parameter.  $\square$

## 4.1 Obtaining Incoercibility

Recall that, to be incoercible, the protocol should be augmented by *faking algorithms* for the two parties. The faking algorithm for a party takes as input a value  $x'$ , representing a fake input value for the party, as well as the party's local state and the messages sent by that party so far, and outputs a "fake random input"  $r'$  for the party, such that running the party's program on input

<sup>4</sup>We can assume that both  $\text{Adv}_1, \text{Adv}_2$  are deterministic, and thus there is no need to pass the state from  $\text{Adv}_1$  to  $\text{Adv}_2$ , since  $\text{Adv}_2$  can always run  $\text{Adv}_1$ .

$x'$  and random input  $r'$  results in the messages sent by the party so far, and furthermore  $r'$  “looks random” given the rest of the view of the adversary. More precisely, the protocol together with the faking algorithm should be simulatable as in the definition of incoercible computation presented in Section 2.

To show incoercibility for the garbler, we demonstrate a faking algorithm: Having received message  $\alpha$ , sent message  $\beta$ , and given the fake input value  $x'$ , simply run the Explain algorithm with input message  $m = \beta, x', \alpha$  and some fresh randomness. Then output the output of Explain.

It is straightforward to see that the same simulation actually demonstrates incoercibility for the garbler. Indeed, the simulator exhibits the same information for coercion and corruption attacks.

## References

- [AOZZ15] Joël Alwen, Rafail Ostrovsky, Hong-Sheng Zhou, and Vassilis Zikas. Incoercible multi-party computation and universally composable receipt-free voting. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, pages 763–780, 2015.
- [BCH12] Nir Bitansky, Ran Canetti, and Shai Halevi. Leakage-tolerant interactive protocols. In *Theory of Cryptography - 9th Theory of Cryptography Conference, TCC 2012, Taormina, Sicily, Italy, March 19-21, 2012. Proceedings*, pages 266–284, 2012.
- [BDL14] Nir Bitansky, Dana Dachman-Soled, and Huijia Lin. Leakage-tolerant computation with input-independent preprocessing. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, pages 146–163, 2014.
- [BGI<sup>+</sup>01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *Electronic Colloquium on Computational Complexity (ECCC)*, 8(057), 2001.
- [BGI13] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. *IACR Cryptology ePrint Archive*, 2013:401, 2013.
- [BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10, 1988.
- [BW13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *ASIACRYPT*, pages 280–300, 2013.
- [Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. Full version in IACR Eprint Archive, record 2000/067 (2013 revision), 2001.
- [CDNO97] Ran Canetti, Cynthia Dwork, Moni Naor, and Rafail Ostrovsky. Deniable encryption. In *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, pages 90–104, 1997.

- [CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, pages 61–85, 2007.
- [CFGN96] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. Adaptively secure multiparty computation. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 639–648, 1996.
- [CG96] Ran Canetti and Rosario Gennaro. Incoercible multiparty computation (extended abstract). In *37th Annual Symposium on Foundations of Computer Science, FOCS '96, Burlington, Vermont, USA, 14-16 October, 1996*, pages 504–513, 1996.
- [CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 494–503, 2002.
- [DKR14] Dana Dachman-Soled, Jonathan Katz, and Vanishree Rao. Adaptively secure, universally composable, multi-party computation in constant rounds. *IACR Cryptology ePrint Archive*, 2014:858, 2014.
- [DN00] Ivan Damgård and Jesper Buus Nielsen. Improved non-committing encryption schemes based on a general complexity assumption. In *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*, pages 432–450, 2000.
- [EGL85] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, 1985.
- [GGH<sup>+</sup>13] 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*, pages 40–49, 2013.
- [GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.
- [GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229, 1987.
- [GP14] Sanjam Garg and Antigoni Polychroniadou. Two-round adaptively secure MPC from indistinguishability obfuscation. *IACR Cryptology ePrint Archive*, 2014:844, 2014.
- [GR14] Shafi Goldwasser and Guy N. Rothblum. On best-possible obfuscation. *J. Cryptology*, 27(3):480–505, 2014.

- [GS12] Sanjam Garg and Amit Sahai. Adaptively secure multi-party computation with dishonest majority. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 105–123, 2012.
- [IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, pages 572–591, 2008.
- [LP09] Yehuda Lindell and Benny Pinkas. A proof of security of yao’s protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.
- [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*, pages 475–484, 2014.

### Program SimGen

**Constants:** circuit  $C$  with  $m$  wires and  $s$  output wires; let's assume that first  $2n$  wires are input wires and last  $s$  wires are output wires

**Input:** the result of the computation  $y$ ;  $P_1$ 's two sets of public keys  $\overline{PK}_0, \overline{PK}_1$ ; randomness  $u = u_1 u_2 u_3 u_4$

$Gen(y, \overline{PK}; u)$ :

1. Create labels for wires:  $(k_1^0, k_1^1), \dots, (k_m^0, k_m^1) \leftarrow u_1$
2. Create encryptions of labels:
  - (a) Partition  $u_2$  into  $u_{21}, \dots, u_{2m}$ , and each  $u_{2t}$  into  $u_{2t1}, \dots, u_{2t4}$
  - (b) Partition  $u_3$  into  $u_{31}, \dots, u_{3m}$ , and each  $u_{3t}$  into  $u_{3t1}, \dots, u_{3t4}$
  - (c) For every gate  $t$  in  $C$  create 4 encryptions (all 4 encryptions encrypt the same label):
$$GC_t[0, 0] \leftarrow SEnc_{k_i^0}(SEnc_{k_j^0}(k_l^0; u_{2t1}); u_{3t1})$$

$$GC_t[0, 1] \leftarrow SEnc_{k_i^0}(SEnc_{k_j^1}(k_l^0; u_{2t2}); u_{3t2})$$

$$GC_t[1, 0] \leftarrow SEnc_{k_i^1}(SEnc_{k_j^0}(k_l^0; u_{2t3}); u_{3t3})$$

$$GC_t[1, 1] \leftarrow SEnc_{k_i^1}(SEnc_{k_j^1}(k_l^0; u_{2t4}); u_{3t4})$$
  - (d) shuffle  $GC_t[0, 0], GC_t[1, 0], GC_t[0, 1], GC_t[1, 1]$
3. Create encryptions of labels for  $P_1$ 's input:
  - (a) Partition  $u_4$  into  $u_{401}, \dots, u_{40n}, u_{411}, \dots, u_{41n}$
  - (b) For all  $i = 1, \dots, n$   $c_i^0 \leftarrow Enc_{PK_0^i}(k_{n+i}^0; u_{40i}), c_i^1 \leftarrow Enc_{PK_1^i}(k_{n+i}^1; u_{41i})$
4. output:
  - (a)  $GC_i[0, 0], GC_i[0, 1], GC_i[1, 0], GC_i[1, 1]$  for  $i = 1..m$  (garbled circuit)
  - (b)  $(y_1 : k_{m-s+1}^0; 1 - y_1 : k_{m-s+1}^1), \dots, (y_s : k_m^0; 1 - y_s : k_m^1)$  (the result table)
  - (c)  $k_1^0, \dots, k_n^0$  (labels for  $P_0$ 's input)
  - (d)  $(c_1^0, c_1^1) \dots, (c_n^0, c_n^1)$  (encrypted labels for  $P_1$ 's input)

**Figure 5:** Program SimGen, used by a simulator to create a fake garbled circuit.

Program P:1

**constants:**  $\alpha^*, r^*, er^*, \beta^*, M^*, x_0$ .

**inputs:** protocol input  $x$ , 1-round message  $\alpha$ , randomness  $r = r[1]r[2]$

$P(x, \alpha; r)$  :

1. check if  $r$  has encoded value inside:
  - (a) if  $(x, \alpha, r) = (x_0, \alpha^*, r^*)$  or  $(x, \alpha, r) = (x_0, \alpha^*, er^*)$  then output  $\beta^*$
  - (b) if  $r[2] = r^*[2]$  or  $r[2] = er^*[2]$  then goto 2
  - (c)  $M' \leftarrow F_{k\{r^*[2], er^*[2]\}}(r[2]) \oplus r[1]$ ;
  - (d) if  $M' = M^*$  then goto 2;
  - (e) if  $I_{k_I\{M^*\}}(M') \neq r[2]$  then goto 2;
  - (f) parse  $M'$  as  $\beta', x', \alpha', \rho'$ . If  $(x', \alpha') \neq (x, \alpha)$  then goto 2;
  - (g) output  $\beta'$
2. else run Gen:
  - (a)  $u \leftarrow E_{k_E\{(x_0, \alpha^*, r^*), (x_0, \alpha^*, er^*)\}}(x, \alpha, r)$
  - (b) output  $Gen(x, \alpha; u)$

**Figure 6:** Program P:1.

Program Explain:1

**constants:**  $M^*, r^*, er^*$

**inputs:** message  $m$  which should be encoded; randomness  $\rho$

$P(m; \rho)$  :

1.  $M \leftarrow m, prg(\rho)$
2.  $r[2] \leftarrow I_{k_I\{M^*\}}(M)$ ,  $r[1] \leftarrow F_{k\{r^*[2], er^*[2]\}}(r[2]) \oplus M$
3. output  $r = r[1]r[2]$

**Figure 7:** Program Explain:1.