

# Adaptively Secure, Universally Composable, Multiparty Computation in Constant Rounds

Dana Dachman-Soled<sup>1</sup>, Jonathan Katz<sup>1\*</sup>, and Vanishree Rao<sup>2\*\*</sup>

<sup>1</sup> University of Maryland, USA

danadach@ece.umd.edu, jkatz@cs.umd.edu

<sup>2</sup> University of California at Los Angeles, USA

vanishri@cs.ucla.edu

**Abstract.** Cryptographic protocols with *adaptive security* ensure that security holds against an adversary who can dynamically determine which parties to corrupt as the protocol progresses—or even after the protocol is finished. In the setting where all parties may potentially be corrupted, and secure erasure is not assumed, it has been a long-standing open question to design secure-computation protocols with adaptive security running in *constant* rounds.

Here, we show a constant-round, universally composable protocol for computing any functionality, tolerating a malicious, adaptive adversary corrupting any number of parties. Interestingly, our protocol can compute *all* functionalities, not just adaptively well-formed ones. The protocol relies on indistinguishability obfuscation, and assumes a common reference string.

## 1 Introduction

When designing and analyzing protocols for secure computation, there are several different adversarial models one can consider. The original definitions of security assume a *static* adversary who decides which parties to corrupt before execution of the protocol begins. Subsequently [3, 11], researchers began to consider the more challenging setting in which the adversary may *adaptively* decide which parties to corrupt as the protocol progresses—or even after the protocol ends. It is easy to come up with examples of protocols that are secure in a static-corruption model, but that are trivially insecure in the adaptive setting.

Even in a setting where adaptive corruptions are considered, there are different assumptions one can make. Initial work on adaptive security [3] made the assumption that honest parties can securely *erase* local data (e.g., randomness or other internal state) when no longer needed. Later work, led by Canetti et al. [11], sought to avoid this assumption, arguing that it is unwise to rely on other parties to erase data (since there is no way such erasure can be verified) and that it is generally difficult—even for an honest party who intends to erase data—to

---

\* Work supported in part by NSF awards #1111599 and #1223623.

\*\* Work done while visiting the University of Maryland.

ensure that all traces of data are gone. Whether or not erasure is assumed has a significant impact on the complexity of adaptively secure protocols; for example, adaptively secure public-key encryption is fairly simple and efficient [3] if erasure is assumed, but much more complicated (and much less efficient) [11, 2, 18, 16] without this assumption. Similarly, adaptively secure two-party computation is much easier with the assumption of secure erasure [30] than without [14].

Designing protocols without the assumption of secure erasure is difficult, in part, due to the need to deal with *post-execution corruption* (PEC), whereby an adversary can corrupt parties (and hence obtain the randomness they used) even after execution of the protocol has concluded. Handling PEC is inherent to the setting of universal composability (UC) [9], and is important for ensuring sequential composition even in the stand-alone setting [8]. If secure erasure is assumed, the definition of adaptive security does not change whether or not PEC is allowed [10], but without erasure the requirement of dealing with PEC adds significant additional complications.

**Prior work.** We are interested in adaptive security, with PEC, in a model where secure erasure is not assumed. Some prior protocols for secure computation in this setting (e.g., [11, 2]) assume a majority of the parties remain uncorrupted. Other work [28, 27, 22, 25], including concurrent work of [19], allows *all but one* of the parties to be corrupted. While it may seem strange to worry about corruption of *all* parties, consideration of this case is important when a protocol  $\Pi_{\text{outer}}$  invokes some protocol  $\Pi_{\text{inner}}$  (not involving all parties running  $\Pi_{\text{outer}}$ ) as a subroutine. In this case, all parties running  $\Pi_{\text{inner}}$  may eventually be corrupted, and security of  $\Pi_{\text{outer}}$  should still be guaranteed.

To the best of our knowledge, all prior work giving adaptively secure protocols for general functionalities (without erasure), and tolerating an arbitrary number of corruptions, are based on the Goldreich-Micali-Wigderson [23] paradigm for semi-honest computation, and thus have round complexity linear in the depth of the circuit being computed. These include protocols in the common reference string model [14], the “sunspots” model [15], the key-registration model [1], and, more generally, based on adaptively secure UC puzzles [17]. In addition, all prior work in this setting handles only “adaptively well-formed functionalities” (see [14] for a definition).

## 1.1 Our Result

We show a constant-round, universally composable protocol for multiparty computation of arbitrary functionalities, with security against a malicious, adaptive adversary corrupting any number of parties. We highlight that our protocol can be used to securely compute *all* functionalities, not just adaptively well-formed ones. Our protocol relies on indistinguishability obfuscation, and assumes a common reference string.

**Overview of our techniques.** The main difficulty in our setting is to construct a constant-round protocol with security against a *semi-honest*, adaptive adversary corrupting any number of parties. Given any such protocol, we can compile

it as in [14] to obtain a universally composable protocol with security against a *malicious*, adaptive adversary, and still running in constant rounds. We may also assume secure channels, which can be implemented using adaptively secure encryption.

Our protocol in the semi-honest setting relies on a common reference string (CRS). While it would be more elegant to avoid this assumption, a CRS—or some other form of setup—is anyway needed [12] in order to obtain universally composable computation in the presence of *malicious* adversaries corrupting half or more of the parties, even in a static-corruption model. Thus, as far as our final result (i.e., our protocol with security in the malicious setting) is concerned, some form of setup is unavoidable. Moreover, results of Garg and Sahai [22] indicate that a CRS (or some other form of setup) is needed to obtain constant-round, universally composable, multiparty protocols with adaptive security even in the semi-honest case; see further discussion below.

At its core, our protocol relies on the ability to make arbitrary algorithms *explainable*, an idea we explain in more detail now. Fix some randomized algorithm  $\text{Alg}$ . Informally, an *explainable* version of  $\text{Alg}$  is an algorithm  $\widetilde{\text{Alg}}$  along with an associated *explain* algorithm  $\text{Explain}$  such that, for any input, (1) the distributions over the outputs of  $\text{Alg}(\text{input})$  and  $\widetilde{\text{Alg}}(\text{input})$  are statistically close, and (2) choosing random coins  $r$ , computing  $\text{output} := \text{Alg}(\text{input}; r)$ , and outputting  $(\text{output}, r)$  is computationally indistinguishable from choosing random coins  $r$ , computing  $\text{output} := \widetilde{\text{Alg}}(\text{input}; r)$ , and then outputting  $(\text{output}, \text{Explain}(\text{input}, \text{output}))$ . That is, the  $\text{Explain}$  algorithm provides the ability to sample random coins for  $\widetilde{\text{Alg}}$  that “explain” any given input/output pair. (A related notion was considered by Ishai et al. [26], though without any construction being given.)

Sahai and Waters [31] introduced the notion of explainability for the specific case of public-key encryption schemes, in the context of constructing a deniable encryption scheme. We observe that their techniques can be suitably generalized to give an explainable version of *arbitrary* algorithms based on indistinguishability obfuscation for general circuits (and one-way functions). We refer the reader to Section 3 for a formal statement of this result.

Let  $C$  be a circuit taking  $n$ -bit inputs.<sup>1</sup> Consider the following functionality  $\text{NextMsg}$  that (essentially) computes the next-message function for a two-round secure-computation protocol for  $C$  based on garbled circuits:  $\text{NextMsg}$  takes as input a sequence of first-round messages  $\text{OT}_{1,1}, \dots, \text{OT}_{1,n}$  for a two-round, adaptively secure, oblivious-transfer (OT) protocol (e.g., the protocol of [14]); it then (1) computes a garbled circuit  $\text{GC}$  corresponding to  $C$ , along with input-wire labels  $\{(y_{i,0}, y_{i,1})\}_{i=1}^n$ , and (2) computes a sequence of OT responses  $\text{OT}_{2,1}, \dots, \text{OT}_{2,n}$ . (These responses allow the party that generated  $\text{OT}_{1,i}$  using input bit  $b$  to recover  $y_{i,b}$  while learning nothing about  $y_{i,1-b}$ .) The output of  $\text{NextMsg}$  is  $(\text{GC}, \text{OT}_{2,1}, \dots, \text{OT}_{2,n})$ . The CRS for our protocol will be  $\widetilde{\text{NextMsg}}$ ,

<sup>1</sup> We assume for simplicity here that  $C$  is deterministic. Randomized functionalities are handled in Section 4.

an explainable version of  $\text{NextMsg}$ .<sup>2</sup> We note that, in contrast to [31], in the real-world execution no parties have access to the  $\widetilde{\text{Explain}}$  algorithm corresponding to  $\widetilde{\text{NextMsg}}$ .

Our multiparty protocol computing  $C$  can now be described quite simply. The protocol proceeds in four rounds. Say we have  $n$  parties  $P_1, \dots, P_n$  holding inputs  $x_1, \dots, x_n$ , respectively. These parties generate first-round OT messages  $\text{OT}_{1,1}, \dots, \text{OT}_{1,n}$  (with the party who is supposed to provide the  $i$ th input generating  $\text{OT}_{1,i}$ ), and send these to  $P_n$ . Party  $P_n$  then runs  $\widetilde{\text{NextMsg}}(\text{OT}_{1,1}, \dots, \text{OT}_{1,n})$  to obtain  $\text{GC}, \text{OT}_{2,1}, \dots, \text{OT}_{2,n}$ , and sends  $\text{OT}_{2,i}$  to the corresponding party (which might be itself). Each party  $P_i$  then locally recovers  $y_i$ , the label for the  $i$ th input wire of the garbled circuit, and sends  $y_i$  to  $P_n$ . Finally,  $P_n$  evaluates the garbled circuit  $\text{GC}$  using the provided input-wire labels to obtain the output  $z$ , and sends  $z$  to all the other parties.<sup>3</sup> Only the third- and fourth-round messages need to be sent via a secure channel.

We now describe the simulator informally. Our simulator begins by generating  $\widetilde{\text{NextMsg}}$  along with its associated  $\widetilde{\text{Explain}}$  algorithm, and letting  $\widetilde{\text{NextMsg}}$  be the CRS. It simulates  $\text{OT}_{1,1}, \dots, \text{OT}_{1,n}$  and  $\text{OT}_{2,1}, \dots, \text{OT}_{2,n}$  using the simulator for the OT protocol (recall the OT protocol is adaptively secure), and uses these for the first two rounds of the protocol. Upon corruption of party  $P_i$ , the simulator corrupts that party in the ideal world and learns its input  $x_i$  and the output  $z$ . Then:

- If this is the first corruption, the simulator generates a simulated garbled circuit  $\text{GC}$  consistent with output  $z$ , along with  $n$  input-wire labels  $y_1, \dots, y_n$ . It also uses  $\widetilde{\text{Explain}}$  to generate random coins  $r^*$  consistent with running  $\widetilde{\text{NextMsg}}$  on input  $\text{OT}_{1,1}, \dots, \text{OT}_{1,n}$  and obtaining output  $\text{GC}, \text{OT}_{2,1}, \dots, \text{OT}_{2,n}$ .
- The simulator uses the simulator for the OT protocol to generate internal state for  $P_i$  consistent with input  $x_i$  and output  $y_i$ , and returns this to the adversary. In addition, if  $P = P_n$  then it returns  $r^*$  to the adversary.

Notably, our simulator is “corruption oblivious” [4]. Roughly, this means the behavior of the simulator upon corruption of a party is independent of the ideal state learned previously. By [4, Theorem 1.2], this means our protocol is also the first *leakage-tolerant* protocol with arbitrary leakage for general functionalities under semi-honest corruption. Moreover, by [5, Theorem 1], we also obtain the first construction of a *2-component OCL compiler* (see [5] for a definition).

**Impossibility results?** We briefly mention two impossibility results regarding (constant-round) adaptively secure computation, and explain why they do not apply in our setting.

<sup>2</sup> As described, the CRS depends on the circuit  $C$ . However, by taking  $C$  to be a universal circuit, the CRS can be fixed independently of the “actual” function  $f$  the parties wish to compute (other than the size of a circuit for  $f$ ).

<sup>3</sup> As described, all parties learn the output of the computation. Standard techniques can be used to handle the general case in which each party learns a possibly different function of the inputs.

First, our protocol can compute *arbitrary* randomized functionalities, not just *adaptively well-formed* ones. (We refer to [14] for a definition of this term.) This may seem somewhat surprising in light of an impossibility result of Ishai et al. [26] showing that adaptively secure computation of functionalities that are not adaptively well-formed is impossible. A closer examination of their result, however, reveals that it does *not* hold in the CRS model.

Second, Garg and Sahai [22] show that no constant-round, adaptively secure, multiparty protocol can be proven secure using black-box techniques; although they only claim this result for protocols with security against malicious adversaries, their proof appears to extend to the case of semi-honest adversaries as well. Again, however, their impossibility result only applies to the “plain” model with no setup, whereas we assume a CRS.

**Concurrent work.** Independent of our work, two other groups of researchers have also studied the problem of constant-round adaptively secure computation. Canetti et al. [13] give a protocol that is similar in spirit to ours, but works only for the *two-party* case and requires sub-exponentially hard indistinguishability obfuscation. Garg and Polychroniadou [21], though also relying on indistinguishability obfuscation, follow a different approach. They give a *round-optimal*, adaptively secure protocol for the multiparty setting. We remark that both these other works only consider adaptively well-formed functionalities.

## 1.2 Organization of the Paper

We review some standard cryptographic background and primitives in Section 2. In Section 3, we introduce the notion of an *explainable* algorithm, and show how the Sahai-Waters compiler [31] can be used to make any algorithm explainable. Finally, in Section 4 we present a constant-round multiparty computation protocol tolerating a semi-honest, adaptive adversary corrupting any number of parties. Applying the compiler of Canetti et al. [14] yields a constant-round protocol tolerating a *malicious*, adaptive adversary corrupting any number of parties.

## 2 Preliminaries

We let  $\lambda$  denote the security parameter. We refer to previous work [8, 10, 30] for definitions of secure computation in the adaptive-corruption setting (with PEC).

### 2.1 Garbled Circuits

We rely on the standard notion of garbled circuits [32]. However, we use slightly non-standard notation that we introduce here. Let  $C$  be a randomized circuit taking  $n$ -bit inputs and using  $\lambda$  bits of randomness. We abstract the construction/evaluation of a garbled circuit for  $C$  via algorithms  $\text{GenGC}, \text{EvalGC}$  with the following properties.  $\text{GenGC}$  is a randomized algorithm that takes as input  $1^\lambda$  and  $C$ , and outputs a garbled circuit  $\text{GC}$  along with  $2n$  input-wire labels

$y_{1,0}, y_{1,1}, \dots, y_{n,0}, y_{n,1} \in \{0, 1\}^\lambda$  and  $2\lambda$  random-wire labels  $w_{1,0}, w_{1,1}, \dots, w_{\lambda,0}, w_{\lambda,1} \in \{0, 1\}^\lambda$ . Deterministic algorithm **EvalGC** takes as input **GC** and  $n + \lambda$  labels  $y_1, \dots, y_n, w_1, \dots, w_\lambda$ , and outputs a value  $z$ .

Correctness requires that for any **GC**,  $(\{y_{i,0}, y_{i,1}\}_{i=1}^n, \{w_{i,0}, w_{i,1}\}_{i=1}^\lambda)$  output by **GenGC** $(1^\lambda, C)$ , any  $x \in \{0, 1\}^n$  and any  $r \in \{0, 1\}^\lambda$ , we have

$$\text{EvalGC}(\text{GC}, \{y_{i,x_i}\}_{i=1}^n, \{w_{i,r_i}\}_{i=1}^\lambda) = C(x; r).$$

Security requires an efficient simulator **SimGC** such that for all  $x, r$ , the distribution

$$\left\{ (\text{GC}, \{(y_{i,0}, y_{i,1})\}_{i=1}^n, \{(w_{i,0}, w_{i,1})\}_{i=1}^\lambda) \leftarrow \text{GenGC}(1^\lambda, C) : \right. \\ \left. (\text{GC}, \{y_{i,x_i}\}_{i=1}^n, \{w_{i,r_i}\}_{i=1}^\lambda) \right\}$$

is computationally indistinguishable from the output of **SimGC** $(1^\lambda, C, C(x; r))$ .

## 2.2 Adaptively Secure Oblivious Transfer

Our protocol uses a two-round, semi-honest, adaptively secure OT protocol as a building block. A suitable construction can be found in [14].

A two-round OT protocol  $\Pi_{\text{OT}}$  comprises three algorithms: a receiver algorithm  $R_{\text{OT}}$ , a sender algorithm  $S_{\text{OT}}$ , and an evaluation algorithm  $E_{\text{OT}}$ . Algorithm  $R_{\text{OT}}$  takes as input a bit  $b$  and random coins  $r_R$ , and outputs initial message  $\text{OT}_1$ . Algorithm  $S_{\text{OT}}$  takes as input an initial message  $\text{OT}_1$ , a pair of  $\lambda$ -bit strings  $(y_0, y_1)$ , and randomness  $r_S$ , and outputs message  $\text{OT}_2$ . The evaluation algorithm  $E_{\text{OT}}$  takes as input  $b, r_R$ , and  $\text{OT}_2$  and outputs the  $\lambda$ -bit string  $y_b$ .

For our purposes we require the following property that is implied by semi-honest, adaptive security of  $\Pi_{\text{OT}}$ . There exist an efficient simulator  $\text{SimOT} = (\text{SimOT}_1, \text{SimOT}_2)$ , where  $\text{SimOT}_2$  is deterministic, such that (1)  $\text{SimOT}_1$  outputs a transcript  $(\text{OT}_1, \text{OT}_2)$  along with state  $\text{st}$  and (2)  $\text{SimOT}_2$ , given as input  $b, y$ , and  $\text{st}$ , outputs coins  $r_R$  for the receiver consistent with  $(\text{OT}_1, \text{OT}_2)$  and the receiver holding input  $b$  and obtaining output  $y$ ; for any  $b, y_0, y_1$ , the distribution

$$\left\{ r_R, r_S \leftarrow \{0, 1\}^*; \text{OT}_1 := R_{\text{OT}}(b; r_R) : (r_R, \text{OT}_1, S_{\text{OT}}(\text{OT}_1, y_0, y_1; r_S)) \right\}$$

is computationally indistinguishable from

$$\left\{ (\text{OT}_1, \text{OT}_2, \text{st}) \leftarrow \text{SimOT}_1(1^\lambda); : (r_R, \text{OT}_1, \text{OT}_2) \right\}.$$

That is, we only require “one-sided security” [25] for adaptive corruption of the receiver.

If we define algorithm  $\text{SimOT}'_1(1^\lambda)$  to run  $\text{SimOT}_1(1^\lambda)$  and output only  $(\text{OT}_1, \text{st})$ , and define the algorithm  $\text{SimOT}'_2(1^\lambda, b, \text{st})$  to simply run  $\text{SimOT}_2(1^\lambda, b, \text{st})$ , then for any  $b$  the distribution  $\left\{ r_R \leftarrow \{0, 1\}^* : (r_R, R_{\text{OT}}(b; r_R)) \right\}$  is computationally indistinguishable from

$$\left\{ (\text{OT}_1, \text{st}) \leftarrow \text{SimOT}'_1(1^\lambda); : (r_R, \text{OT}_1) \right\}.$$

### 2.3 Indistinguishability Obfuscation

We use an indistinguishability obfuscator as a building block. A PPT machine  $i\mathcal{O}$  is an *indistinguishability obfuscator* for a circuit class  $\{\mathcal{C}_\lambda\}$  if the following conditions are satisfied:

**Correctness.** For all  $\lambda$ , and all  $C \in \mathcal{C}_\lambda$ , it holds that  $C$  and  $i\mathcal{O}(1^\lambda, C)$  compute the same function.

**Polynomial slowdown.** There is a polynomial  $p(\cdot)$  such that  $|i\mathcal{O}(1^\lambda, C)| \leq p(\lambda) \cdot |C|$  for all  $C \in \mathcal{C}_\lambda$ .

**Indistinguishability.** For any sequence  $\{(C_{\lambda,0}, C_{\lambda,1}, \text{aux}_\lambda)\}_\lambda$  where  $C_{\lambda,0}, C_{\lambda,1} \in \mathcal{C}_\lambda$ ,  $C_{\lambda,0} \equiv C_{\lambda,1}$ , and  $|C_{\lambda,0}| = |C_{\lambda,1}|$ , and any PPT distinguisher  $D$ , there is a negligible function  $\text{negl}$  such that:

$$|\Pr[D(i\mathcal{O}(1^\lambda, C_{\lambda,0}), \text{aux}_\lambda) = 1] - \Pr[D(i\mathcal{O}(1^\lambda, C_{\lambda,1}), \text{aux}_\lambda) = 1]| \leq \text{negl}(\lambda).$$

When clear from the context, we will often omit the security parameter  $1^\lambda$  as an input to  $i\mathcal{O}$  and as a subscript for  $C$ .

$i\mathcal{O}$  is an *indistinguishability obfuscator* for P/poly if there is a polynomial  $p$  such that  $i\mathcal{O}$  is an indistinguishability obfuscator for  $\{\mathcal{C}_\lambda\}$ , where  $\mathcal{C}_\lambda$  contains all circuits of size at most  $p(\lambda)$ . Garg et al. [20] have shown the first candidate construction of indistinguishability obfuscators for P/poly.

## 3 Explainability Compilers

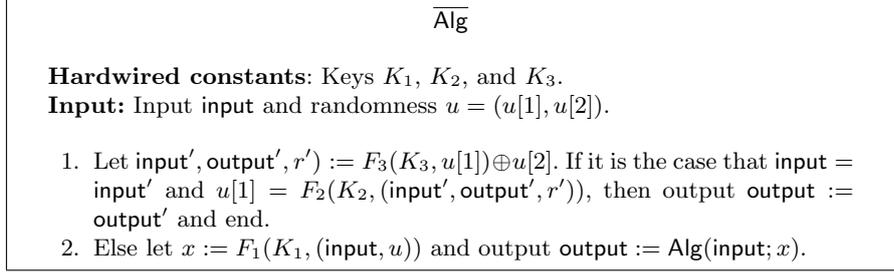
Sahai and Waters [31] define a notion of *explainability* for public-key encryption, and show a compiler that transforms any public-key encryption scheme into an explainable version. Here, we generalize the notion of explainability for an *arbitrary* algorithm  $\text{Alg}$ , and show that the Sahai-Waters compiler can be used to transform any algorithm  $\text{Alg}$  into an explainable version  $\widetilde{\text{Alg}}$ .

At a high level, an explainability compiler takes as input (a description of) a randomized algorithm  $\text{Alg}$ , and outputs two algorithms  $\widetilde{\text{Alg}}, \text{Explain}$ . The first of these is a randomized algorithm computing the same functionality as  $\text{Alg}$ . The second algorithm, roughly speaking, takes an input/output pair  $\text{input}, \text{output}$  and produces random coins  $r$  consistent with running  $\widetilde{\text{Alg}}(\text{input})$  and obtaining the result  $\text{output}$ . That is, the algorithm “explains” the input/output pair  $\text{input}, \text{output}$ . We now give a formal definition.

**Definition 1.** A PPT algorithm  $\text{Comp}$  is an explainability compiler if for every efficient, randomized circuit  $\text{Alg}$ , the following hold:

**Polynomial slowdown.** There is a polynomial  $p(\cdot)$  such that, for any  $(\widetilde{\text{Alg}}, \text{Explain})$  output by  $\text{Comp}(1^\lambda, \text{Alg})$  it holds that  $|\widetilde{\text{Alg}}| \leq p(\lambda) \cdot |\text{Alg}|$ .

**Statistical functional equivalence.** With overwhelming probability over choice of  $(\widetilde{\text{Alg}}, \star)$  as output by  $\text{Comp}(1^\lambda, \text{Alg})$ , the distribution of  $\widetilde{\text{Alg}}(\text{input})$  is statistically close to the distribution of  $\text{Alg}(\text{input})$  for all input.

Fig. 1. Program  $\overline{\text{Alg}}$ 

**Explainability.** *The success probability of every non-uniform, polynomial-time adversary  $\mathcal{A}$  in the following experiment is negligibly close to  $1/2$ :*

1.  $\mathcal{A}(1^\lambda)$  outputs  $\text{input}^*$  of its choice.
2.  $\text{Comp}(1^\lambda, \overline{\text{Alg}})$  is run to obtain  $(\widetilde{\text{Alg}}, \widetilde{\text{Explain}})$ .
3. Choose uniform coins  $r_0 \in \{0, 1\}^*$  and compute  $\text{output}^* := \widetilde{\text{Alg}}(\text{input}^*; r_0)$ .
4. Compute  $r_1 \leftarrow \widetilde{\text{Explain}}(\text{input}^*, \text{output}^*)$ .
5. Choose a uniform bit  $b$  and give  $\widetilde{\text{Alg}}, \text{output}^*, r_b$  to  $\mathcal{A}$ .
6.  $\mathcal{A}$  outputs a bit  $b'$ , and succeeds if  $b' = b$ .

We highlight one key difference between our definition and the corresponding one from [31]: in our case  $\text{input}^*$  is an arbitrary length value (depending on the domain of  $\overline{\text{Alg}}$ ) chosen by the adversary, whereas in [31] the input to the explainable algorithm is a *single bit* chosen uniformly (and given to the adversary). Because of this, and due to the way the explainability compiler is constructed, we require the adversary to choose  $\text{input}^*$  “non-adaptively,” i.e., before being given  $\overline{\text{Alg}}$ . This definition of explainability suffices for our eventual protocol.

### 3.1 Constructing an Explainability Compiler

Following [31], we now show how to construct an explainability compiler. As in [31], we rely on an indistinguishability obfuscator,  $i\mathcal{O}$ , for  $\text{P/poly}$  and three different pseudorandom function (PRF) variants (cf. Appendix A):

- A *puncturable, extracting* PRF  $F_1(K_1, \cdot)$  that accepts inputs of length  $\ell_1 + \ell_2 + \ell_{\text{in}}$ , and outputs strings of length  $\ell_r$ . It is extracting when the input min-entropy is greater than  $\ell_r + 2\lambda + 4$ , with statistical closeness less than  $2^{-(\lambda+1)}$ . Observe that  $\ell_{\text{in}} + \ell_1 + \ell_2 \geq \ell_r + 2\lambda + 4$ , and thus if one-way functions exist then such a PRF exists by Theorem 4.
- A *puncturable, statistically injective* PRF  $F_2(K_2, \cdot)$  that accepts inputs of length  $2\lambda + \ell_{\text{in}} + \ell_{\text{out}}$ , and outputs strings of length  $\ell_1$ . Observe that  $\ell_1 \geq 2 \cdot (2\lambda + \ell_{\text{in}} + \ell_{\text{out}}) + \lambda$ , and thus if one-way functions exist then such a PRF exists by Theorem 3.
- A *puncturable* PRF  $F_3(K_3, \cdot)$  that accepts inputs of length  $\ell_1$  and outputs strings of length  $\ell_2$ . If one-way functions exist, then such a PRF exists by Theorem 2.

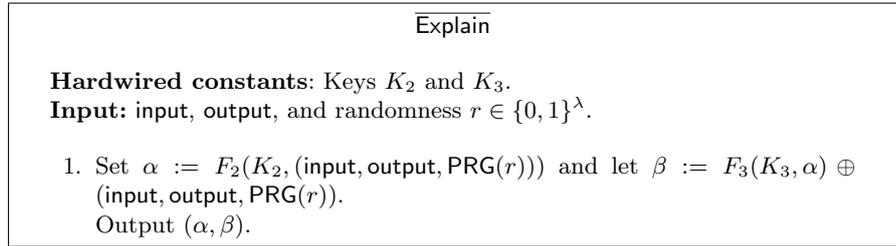
We define  $\text{Comp}(1^\lambda, \text{Alg})$  as follows. Let  $\text{Alg} : \{0, 1\}^{\ell_{\text{in}}} \times \{0, 1\}^{\ell_r} \rightarrow \{0, 1\}^{\ell_{\text{out}}}$  be an algorithm with domain  $\{0, 1\}^{\ell_{\text{in}}}$ , range  $\{0, 1\}^{\ell_{\text{out}}}$ , and randomness length  $\ell_r$ . Our compiled program  $\widetilde{\text{Alg}}$  will take input  $\text{input} \in \{0, 1\}^{\ell_{\text{in}}}$  and randomness  $u = (u[1], u[2])$  of length  $\ell_1 + \ell_2$ , where  $|u[1]| = \ell_1 = 5\lambda + 2(\ell_{\text{in}} + \ell_{\text{out}}) + \ell_r$  and  $|u[2]| = \ell_2 = 2\lambda + \ell_{\text{in}} + \ell_{\text{out}}$ . Our compiler first samples keys  $K_1, K_2$ , and  $K_3$  for PRFs  $F_1, F_2$ , and  $F_3$ , respectively. It then defines algorithms  $\widetilde{\text{Alg}}$  and  $\widetilde{\text{Explain}}$  as in Figures 1 and 2, respectively. Finally, it computes  $\widetilde{\text{Alg}} \leftarrow \text{iO}(\widetilde{\text{Alg}})$  and  $\widetilde{\text{Explain}} \leftarrow \text{iO}(\widetilde{\text{Explain}})$ , and outputs  $(\widetilde{\text{Alg}}, \widetilde{\text{Explain}})$ .

The proofs of security for our compiler, given for completeness in Appendix B, follow closely along the lines of the analogous proofs in [31]. Specifically, the proof of statistical functional equivalence closely follows the proof used by Sahai and Waters to establish IND-CPA security of their deniable encryption scheme, and the proof of explainability follows the Sahai-Waters proof establishing explainability of their deniable encryption scheme. We highlight, however, that in our proof of explainability a difference arises because in our case the input  $\text{input}^*$  is an arbitrary length value (depending on the domain of  $\text{Alg}$ ), whereas in [31] the input is just a single bit. We are able to adapt the proof to this case because we do not allow  $\text{input}^*$  to depend on  $\widetilde{\text{Alg}}$ .

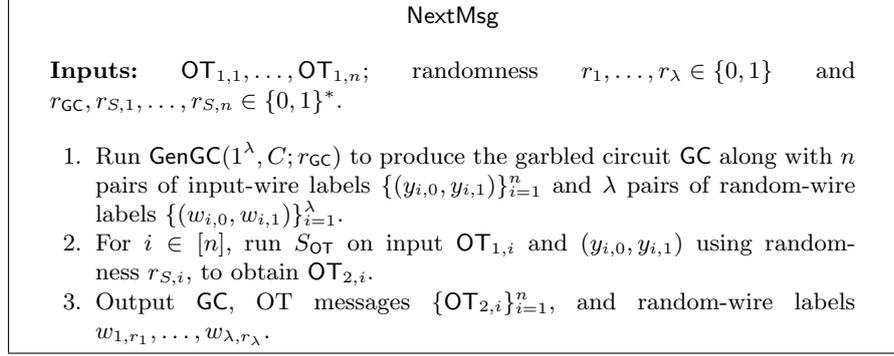
#### 4 A Semi-Honest, Adaptively Secure Protocol

We describe here a protocol for secure computation of a randomized circuit  $C$  by a set of parties  $P_1, \dots, P_n$ . We assume for simplicity that all parties learn the output of  $C$ ; using standard techniques, we can handle the general case in which each party learns a possibly different function of the inputs. For ease of notation, we assume that the domain of  $C$  is  $\{0, 1\}^n$  with party  $P_i$  providing the  $i$ th input  $x_i \in \{0, 1\}$ . (One can easily verify that our protocol and proof generalize to the case of arbitrary-length inputs.) We also assume without loss of generality that  $C$  uses  $\lambda$  random bits.

The CRS of our protocol is an “explainable” version  $\widetilde{\text{NextMsg}}$  of the algorithm  $\text{NextMsg}$  defined in Figure 3. That is, the CRS is generated by computing  $(\widetilde{\text{NextMsg}}, \widetilde{\text{Explain}}) \leftarrow \text{Comp}(1^\lambda, \text{NextMsg})$  and letting the CRS be  $\widetilde{\text{NextMsg}}$ . As described, the CRS depends on  $C$  (since  $\text{NextMsg}$  does); however, by letting  $C$



**Fig. 2.** Program  $\widetilde{\text{Explain}}$



**Fig. 3.** Algorithm NextMsg. The security parameter  $1^\lambda$  and circuit  $C$  are hardwired.

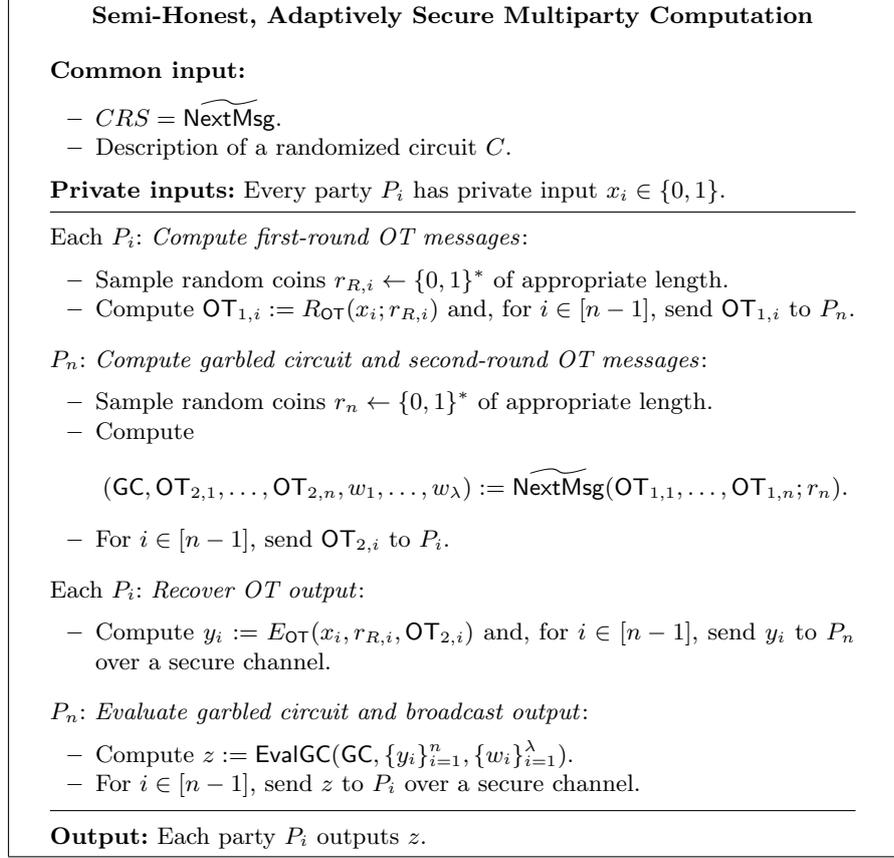
be a universal circuit the CRS can be fixed independently of the “actual” function the parties wish to compute. We note that we allow the environment  $\mathcal{Z}$  to choose the parties’ inputs depending on the CRS.

Let  $\Pi_{\text{OT}} = (R_{\text{OT}}, S_{\text{OT}}, E_{\text{OT}})$  be a two-round, semi-honest, adaptively secure OT protocol (cf. Section 2.2). Our secure-computation protocol  $\Pi$  is defined in Figure 4. We describe the protocol assuming the existence of secure channels; these can be instantiated using any adaptively secure public-key encryption scheme.

**Theorem 1.** *Assume  $\text{Comp}$  is an explainability compiler, and  $\text{GenGC}$  and  $\Pi_{\text{OT}}$  satisfy the definitions from Sections 2.1 and 2.2, respectively. Then protocol  $\Pi$  in Figure 4 UC-realizes functionality  $C$  in the presence of a semi-honest, adaptive adversary corrupting any number of parties.*

*Proof.* Let  $\text{SimGC}$ ,  $\text{SimOT}$  denote appropriate simulators as defined in Section 2. Fix an environment  $\mathcal{Z}$  and a dummy adversary  $\mathcal{A}$  attacking protocol  $\Pi$ . Recall that we allow the environment  $\mathcal{Z}$  to adaptively choose the inputs of all parties *after* seeing the common reference string. Without loss of generality, we assume  $\mathcal{Z}$  first observes the entire protocol transcript (which, since we use secure channels in rounds 3 and 4, consists only of the messages sent in the first two rounds) before corrupting any parties. Our simulator  $\text{Sim}$  for this adversary proceeds as follows:

1. Compute  $(\widetilde{\text{NextMsg}}, \text{Explain}) \leftarrow \text{Comp}(1^\lambda, \text{NextMsg})$ , and give  $\widetilde{\text{NextMsg}}$  to  $\mathcal{Z}$  as the CRS.
2. Run  $\text{SimOT}_1(1^\lambda)$  a total of  $n$  times to obtain  $\{(\text{OT}_{1,i}, \text{OT}_{2,i}, \text{st}_i)\}_{i=1}^n$ . Give  $\text{OT}_{1,1}, \dots, \text{OT}_{1,n-1}$  to  $\mathcal{Z}$  as the first-round message, and  $\text{OT}_{2,1}, \dots, \text{OT}_{2,n-1}$  to  $\mathcal{Z}$  as the second-round message.
3. When  $\mathcal{Z}$  requests to corrupt party  $P_i$ , corrupt  $P_i$  in the ideal world to learn its input  $x_i$  and the output  $z$ . Then:
  - If this is the first party to be corrupted, compute  $(\text{GC}, \{y_i\}_{i=1}^n, \{w_i\}_{i=1}^\lambda) \leftarrow \text{SimGC}(1^\lambda, C, z)$  and  $r_n^* \leftarrow \text{Explain}((\text{OT}_{1,1}, \dots, \text{OT}_{1,n}), (\text{GC}, \text{OT}_{2,1}, \dots,$



**Fig. 4.** Protocol II for computing randomized circuit  $C$ .

- $\text{OT}_{2,n}, w_1, \dots, w_n$ ). Store these values to be used, as needed, in the rest of the simulation.
- In any case, compute  $r_{R,i} := \text{SimOT}_2(1^\lambda, x_i, y_i, \text{st}_i)$  and give  $x_i, z, y_i$ , and  $r_{R,i}$  to  $\mathcal{Z}$ . In addition, if  $i = n$  give  $\{y_i\}_{i=1}^{n-1}$  and  $r_n^*$  to  $\mathcal{Z}$ .
4. Output whatever  $\mathcal{Z}$  outputs.

We prove that the output of  $\mathcal{Z}$  when interacting with  $\mathcal{A}$  and parties in a real-world execution of protocol II is indistinguishable from the output of  $\mathcal{Z}$  when interacting with  $\text{Sim}$  and the functionality  $C$  in an ideal-world execution of the protocol. We do so by considering a sequence of hybrid experiments, beginning with the real-world execution and ending with the ideal-world execution, and showing that each experiment is computationally indistinguishable from the preceding one.

**Hybrid 0.** This corresponds to the real-world execution of the protocol. We write the experiment in a format convenient for the proof. This experiment proceeds via the following steps:

1. Compute  $(\widetilde{\text{NextMsg}}, \text{Explain}) \leftarrow \text{Comp}(1^\lambda, \text{NextMsg})$ , and give  $\widetilde{\text{NextMsg}}$  to  $\mathcal{Z}$  as the CRS.  $\mathcal{Z}$  chooses inputs  $x_1, \dots, x_n$ .
2. For  $i \in [n]$ , sample coins  $r_{R,i}$  and compute  $\text{OT}_{1,i} := R_{\text{OT}}(x_i; r_{R,i})$ . Give the sequence of values  $\text{OT}_{1,1}, \dots, \text{OT}_{1,n-1}$  to  $\mathcal{Z}$  as the first-round message.
3. Sample coins  $r_n$  and compute

$$(\text{GC}, \text{OT}_{2,1}, \dots, \text{OT}_{2,n}, w_1, \dots, w_\lambda) := \widetilde{\text{NextMsg}}(\text{OT}_{1,1}, \dots, \text{OT}_{1,n}; r_n).$$

Give  $\text{OT}_{2,1}, \dots, \text{OT}_{2,n-1}$  to  $\mathcal{Z}$  as the second-round message.

4. When  $\mathcal{Z}$  requests to corrupt party  $P_i$ , compute  $y_i := E_{\text{OT}}(x_i, r_{R,i}, \text{OT}_{2,i})$  and give  $x_i, z, y_i$ , and  $r_{R,i}$  to  $\mathcal{Z}$ . In addition, if  $i = n$  then compute  $y_i := E_{\text{OT}}(x_i, r_{R,i}, \text{OT}_{2,i})$  for  $i \in [n-1]$  and give  $\{y_i\}_{i=1}^{n-1}$  and  $r_n$  to  $\mathcal{Z}$ .

**Hybrid 1.** This experiment is similar to the previous one, except that the  $\text{OT}_1$  messages and the random coins  $\{r_{R,i}\}$  are generated by the simulator for the OT protocol (cf. Section 2.2). That is, the experiment proceeds via the following steps:

1. Compute  $(\widetilde{\text{NextMsg}}, \text{Explain}) \leftarrow \text{Comp}(1^\lambda, \text{NextMsg})$ , and give  $\widetilde{\text{NextMsg}}$  to  $\mathcal{Z}$  as the CRS.  $\mathcal{Z}$  chooses inputs  $x_1, \dots, x_n$ .
2. Run  $\text{SimOT}'_1(1^\lambda)$  a total of  $n$  times to obtain  $\{(\text{OT}_{1,i}, \text{st}_i)\}_{i=1}^n$ . Give the sequence of values  $\text{OT}_{1,1}, \dots, \text{OT}_{1,n-1}$  to  $\mathcal{Z}$  as the first-round message.
3. Sample coins  $r_n$  and compute

$$(\text{GC}, \text{OT}_{2,1}, \dots, \text{OT}_{2,n}, w_1, \dots, w_\lambda) := \widetilde{\text{NextMsg}}(\text{OT}_{1,1}, \dots, \text{OT}_{1,n}; r_n).$$

Give  $\text{OT}_{2,1}, \dots, \text{OT}_{2,n-1}$  to  $\mathcal{Z}$  as the second-round message.

4. When  $\mathcal{Z}$  corrupts party  $P_i$ , compute  $r_{R,i} := \text{SimOT}'_2(1^\lambda, x_i, \text{st}_i)$  and  $y_i := E_{\text{OT}}(x_i, r_{R,i}, \text{OT}_{2,i})$ , and give  $x_i, z, y_i$ , and  $r_{R,i}$  to  $\mathcal{Z}$ . In addition, if  $i = n$  then for  $i \in [n-1]$  compute  $r_{R,i} := \text{SimOT}'_2(1^\lambda, x_i, \text{st}_i)$  and  $y_i := E_{\text{OT}}(x_i, r_{R,i}, \text{OT}_{2,i})$ , and give  $\{y_i\}_{i=1}^{n-1}$  and  $r_n$  to  $\mathcal{Z}$ .

It follows immediately by security of the OT protocol (and a straightforward hybrid argument) that this experiment is computationally indistinguishable from the previous one.

**Hybrid 2.** This experiment is similar to the previous one, except that we now use the Explain algorithm to generate the random coins  $r_n$ . That is, the experiment proceeds as follow:

1. Compute  $(\widetilde{\text{NextMsg}}, \text{Explain}) \leftarrow \text{Comp}(1^\lambda, \text{NextMsg})$ , and give  $\widetilde{\text{NextMsg}}$  to  $\mathcal{Z}$  as the CRS.  $\mathcal{Z}$  chooses inputs  $x_1, \dots, x_n$ .
2. Run  $\text{SimOT}'_1(1^\lambda)$  a total of  $n$  times to obtain  $\{(\text{OT}_{1,i}, \text{st}_i)\}_{i=1}^n$ . Give the sequence of values  $\text{OT}_{1,1}, \dots, \text{OT}_{1,n-1}$  to  $\mathcal{Z}$  as the first-round message.
3. Sample coins  $r_n$  and compute

$$(\text{GC}, \text{OT}_{2,1}, \dots, \text{OT}_{2,n}, w_1, \dots, w_\lambda) := \widetilde{\text{NextMsg}}(\text{OT}_{1,1}, \dots, \text{OT}_{1,n}; r_n).$$

In addition, let  $\text{input}^* = (\text{OT}_{1,1}, \dots, \text{OT}_{1,n})$  and  $\text{output}^* = (\text{GC}, \text{OT}_{2,1}, \dots, \text{OT}_{2,n}, w_1, \dots, w_\lambda)$ , and compute  $r_n^* \leftarrow \text{Explain}(\text{input}^*, \text{output}^*)$ .

Give  $\text{OT}_{2,1}, \dots, \text{OT}_{2,n-1}$  to  $\mathcal{Z}$  as the second-round message.

4. When  $\mathcal{Z}$  corrupts party  $P_i$ , compute  $r_{R,i} := \text{SimOT}'_2(1^\lambda, x_i, \text{st}_i)$  and  $y_i := E_{\text{OT}}(x_i, r_{R,i}, \text{OT}_{2,i})$ , and give  $x_i, z, y_i$ , and  $r_{R,i}$  to  $\mathcal{Z}$ . In addition, if  $i = n$  then for  $i \in [n-1]$  compute  $r_{R,i} := \text{SimOT}'_2(1^\lambda, x_i, \text{st}_i)$  and  $y_i := E_{\text{OT}}(x_i, r_{R,i}, \text{OT}_{2,i})$ , and give  $\{y_i\}_{i=1}^{n-1}$  and  $r_n^*$  to  $\mathcal{Z}$ .

Computationally indistinguishability of this experiment from the previous one follows from the definition of explainability (cf. Definition 1), and the fact that  $\text{Comp}$  is an explainability compiler. To see this, say there is an efficient adversary  $\mathcal{Z}$  and a non-uniform, polynomial-time distinguisher  $D$  that distinguishes the outcome of Hybrid 1 from that of Hybrid 2. We show how to use this to construct an attacker  $\mathcal{A}'$  violating explainability.  $\mathcal{A}'$  works as follows: it runs  $\text{SimOT}'_1(1^\lambda)$  a total of  $n$  times to obtain  $\{(\text{OT}_{1,i}, \text{st}_i)\}_{i=1}^n$ , and outputs the value  $\text{input}^* = (\text{OT}_{1,1}, \dots, \text{OT}_{1,n})$ . Given  $\widetilde{\text{NextMsg}}, \text{output}^*, r$  in response, where  $\text{output}^* = (\text{GC}, \text{OT}_{2,1}, \dots, \text{OT}_{2,n}, w_1, \dots, w_\lambda)$ , it then does:

1. Give  $\widetilde{\text{NextMsg}}$  to  $\mathcal{Z}$  as the CRS.  $\mathcal{Z}$  chooses inputs  $x_1, \dots, x_n$ .
2. Give  $\text{OT}_{1,1}, \dots, \text{OT}_{1,n-1}$  to  $\mathcal{Z}$  as the first-round message, and  $\text{OT}_{2,1}, \dots, \text{OT}_{2,n-1}$  to  $\mathcal{Z}$  as the second-round message.
3. When  $\mathcal{Z}$  corrupts party  $P_i$ , compute  $r_{R,i} := \text{SimOT}'_2(1^\lambda, x_i, \text{st}_i)$  and  $y_i := E_{\text{OT}}(x_i, r_{R,i}, \text{OT}_{2,i})$ , and give  $x_i, z, y_i$ , and  $r_{R,i}$  to  $\mathcal{Z}$ . In addition, if  $i = n$  then for  $i \in [n-1]$  compute  $r_{R,i} := \text{SimOT}'_2(1^\lambda, x_i, \text{st}_i)$  and  $y_i := E_{\text{OT}}(x_i, r_{R,i}, \text{OT}_{2,i})$ , and give  $\{y_i\}_{i=1}^{n-1}$  and  $r$  to  $\mathcal{Z}$ .

Finally, run  $D$  on the output of  $\mathcal{Z}$  and output the result. It is easy to see that if the coins  $r$  are those used to run  $\widetilde{\text{NextMsg}}$ , then the view of  $\mathcal{Z}$  when run as a subroutine by  $\mathcal{A}'$  corresponds to Hybrid 1, whereas if the coins  $r$  are those output by  $\text{Explain}$ , then the view of  $\mathcal{Z}$  when run as a subroutine by  $\mathcal{A}'$  corresponds to Hybrid 2. Indistinguishability of the two experiments follows.

**Hybrid 3.** This is similar to the previous experiment, except that  $\widetilde{\text{NextMsg}}$  and  $\text{Explain}$  are used in place of  $\text{NextMsg}$ . That is, the experiment proceeds as follows:

1. Compute  $(\widetilde{\text{NextMsg}}, \text{Explain}) \leftarrow \text{Comp}(1^\lambda, \text{NextMsg})$ , and give  $\widetilde{\text{NextMsg}}$  to  $\mathcal{Z}$  as the CRS.  $\mathcal{Z}$  chooses inputs  $x_1, \dots, x_n$ .
2. Run  $\text{SimOT}'_1(1^\lambda)$  a total of  $n$  times to obtain  $\{(\text{OT}_{1,i}, \text{st}_i)\}_{i=1}^n$ . Give the sequence of values  $\text{OT}_{1,1}, \dots, \text{OT}_{1,n-1}$  to  $\mathcal{Z}$  as the first-round message.
3. Compute

$$(\text{GC}, \text{OT}_{2,1}, \dots, \text{OT}_{2,n}, w_1, \dots, w_\lambda) \leftarrow \text{NextMsg}(\text{OT}_{1,1}, \dots, \text{OT}_{1,n}).$$

In addition, let  $\text{input}^* = (\text{OT}_{1,1}, \dots, \text{OT}_{1,n})$  and  $\text{output}^* = (\text{GC}, \text{OT}_{2,1}, \dots, \text{OT}_{2,n}, w_1, \dots, w_\lambda)$ , and compute  $r_n^* \leftarrow \text{Explain}(\text{input}^*, \text{output}^*)$ .

Give  $\text{OT}_{2,1}, \dots, \text{OT}_{2,n-1}$  to  $\mathcal{Z}$  as the second-round message.

4. When  $\mathcal{Z}$  corrupts party  $P_i$ , compute  $r_{R,i} := \text{SimOT}'_2(1^\lambda, x_i, \text{st}_i)$  and  $y_i := E_{\text{OT}}(x_i, r_{R,i}, \text{OT}_{2,i})$ , and give  $x_i, z, y_i$ , and  $r_{R,i}$  to  $\mathcal{Z}$ . In addition, if  $i = n$  then for  $i \in [n-1]$  compute  $r_{R,i} := \text{SimOT}'_2(1^\lambda, x_i, \text{st}_i)$  and  $y_i := E_{\text{OT}}(x_i, r_{R,i}, \text{OT}_{2,i})$ , and give  $\{y_i\}_{i=1}^{n-1}$  and  $r_n^*$  to  $\mathcal{Z}$ .

Indistinguishability of this experiment from the previous one follows by statistical equivalence of  $\widetilde{\text{NextMsg}}$  and  $\widetilde{\text{NextMsg}}$ .

**Hybrid 4.** In this experiment, we first make explicit the steps of  $\widetilde{\text{NextMsg}}$ . (This is just a syntactic rewriting, and does not affect the experiment.) In addition, we now set  $y_i = y_{i,x_i}$  instead of computing  $y_i$  using the OT-evaluation algorithm  $E_{\text{OT}}$ . This experiment proceeds as follows:

1. Compute  $(\widetilde{\text{NextMsg}}, \text{Explain}) \leftarrow \text{Comp}(1^\lambda, \text{NextMsg})$ , and give  $\widetilde{\text{NextMsg}}$  to  $\mathcal{Z}$  as the CRS.  $\mathcal{Z}$  chooses inputs  $x_1, \dots, x_n$ .
2. Run  $\text{SimOT}'_1(1^\lambda)$  a total of  $n$  times to obtain  $\{(\text{OT}_{1,i}, \text{st}_i)\}_{i=1}^n$ . Give the sequence of values  $\text{OT}_{1,1}, \dots, \text{OT}_{1,n-1}$  to  $\mathcal{Z}$  as the first-round message.
3. Compute  $(\text{GC}, \{(y_{i,0}, y_{i,1})\}_{i=1}^n, \{(w_{i,0}, w_{i,1})\}_{i=1}^\lambda) \leftarrow \text{GenGC}(1^\lambda, C)$  and set  $y_i = y_{i,x_i}$  for all  $i$ . For  $i \in [n]$ , run  $\text{OT}_{2,i} \leftarrow S_{\text{OT}}(\text{OT}_1, y_{i,0}, y_{i,1})$ . Choose uniform  $r_1, \dots, r_\lambda \in \{0, 1\}$ , and let  $\text{input}^* = (\text{OT}_{1,1}, \dots, \text{OT}_{1,n})$  and  $\text{output}^* = (\text{GC}, \text{OT}_{2,1}, \dots, \text{OT}_{2,n}, w_{r_1}, \dots, w_{r_\lambda})$ . Compute  $r_n^* \leftarrow \text{Explain}(\text{input}^*, \text{output}^*)$ . Give  $\text{OT}_{2,1}, \dots, \text{OT}_{2,n-1}$  to  $\mathcal{Z}$  as the second-round message.
4. When  $\mathcal{Z}$  corrupts party  $P_i$ , compute  $r_{R,i} := \text{SimOT}'_2(1^\lambda, x_i, \text{st}_i)$ . Give  $x_i, z, y_i$ , and  $r_{R,i}$  to  $\mathcal{Z}$ . In addition, if  $i = n$  then give  $\{y_i\}_{i=1}^{n-1}$  and  $r_n^*$  to  $\mathcal{Z}$ .

Computational indistinguishability of this experiment from the previous one follows from security of the OT protocol.

**Hybrid 5.** In the previous experiment the  $\text{OT}_2$  messages were generated honestly as part of  $\widetilde{\text{NextMsg}}$ . Here, we have the OT simulator output them instead. That is, we now do:

1. Compute  $(\widetilde{\text{NextMsg}}, \text{Explain}) \leftarrow \text{Comp}(1^\lambda, \text{NextMsg})$ , and give  $\widetilde{\text{NextMsg}}$  to  $\mathcal{Z}$  as the CRS.  $\mathcal{Z}$  chooses inputs  $x_1, \dots, x_n$ .
2. Run  $\text{SimOT}_1(1^\lambda)$  a total of  $n$  times to obtain  $\{(\text{OT}_{1,i}, \text{OT}_{2,i}, \text{st}_i)\}_{i=1}^n$ . Give the sequence of values  $\text{OT}_{1,1}, \dots, \text{OT}_{1,n-1}$  to  $\mathcal{Z}$  as the first-round message, and give  $\text{OT}_{2,1}, \dots, \text{OT}_{2,n-1}$  to  $\mathcal{Z}$  as the second-round message.
3. Compute  $(\text{GC}, \{(y_{i,0}, y_{i,1})\}_{i=1}^n, \{(w_{i,0}, w_{i,1})\}_{i=1}^\lambda) \leftarrow \text{GenGC}(1^\lambda, C)$  and set  $y_i = y_{i,x_i}$  for all  $i$ . Choose uniform values  $r_1, \dots, r_\lambda \in \{0, 1\}$ , and let  $\text{input}^* = (\text{OT}_{1,1}, \dots, \text{OT}_{1,n})$  and  $\text{output}^* = (\text{GC}, \text{OT}_{2,1}, \dots, \text{OT}_{2,n}, w_{r_1}, \dots, w_{r_\lambda})$ . Compute  $r_n^* \leftarrow \text{Explain}(\text{input}^*, \text{output}^*)$ .
4. When  $\mathcal{Z}$  corrupts party  $P_i$ , compute  $r_{R,i} := \text{SimOT}_2(1^\lambda, x_i, y_i, \text{st}_i)$ . Give  $x_i, z, y_i$ , and  $r_{R,i}$  to  $\mathcal{Z}$ . In addition, if  $i = n$  then give  $\{y_i\}_{i=1}^{n-1}$  and  $r_n^*$  to  $\mathcal{Z}$ .

Again, computational indistinguishability between this experiment and the previous one follows by security of the OT protocol.

**Hybrid 6.** Here we use the garbled-circuit simulator (cf. Section 2.1) instead of the garbled-circuit generation algorithm. Thus, the experiment now proceeds as follows:

1. Compute  $(\widetilde{\text{NextMsg}}, \text{Explain}) \leftarrow \text{Comp}(1^\lambda, \text{NextMsg})$ , and give  $\widetilde{\text{NextMsg}}$  to  $\mathcal{Z}$  as the CRS.  $\mathcal{Z}$  chooses inputs  $x_1, \dots, x_n$ .

2. Run  $\text{SimOT}_1(1^\lambda)$  a total of  $n$  times to obtain  $\{(\text{OT}_{1,i}, \text{OT}_{2,i}, \text{st}_i)\}_{i=1}^n$ . Give  $\text{OT}_{1,1}, \dots, \text{OT}_{1,n-1}$  to  $\mathcal{Z}$  as the first-round message, and  $\text{OT}_{2,1}, \dots, \text{OT}_{2,n-1}$  to  $\mathcal{Z}$  as the second-round message.
3. Compute  $(\text{GC}, \{y_i\}_{i=1}^n, \{w_i\}_{i=1}^\lambda) \leftarrow \text{SimGC}(1^\lambda, C, z)$ . Let  $\text{input}^* = (\text{OT}_{1,1}, \dots, \text{OT}_{1,n})$  and  $\text{output}^* = (\text{GC}, \text{OT}_{2,1}, \dots, \text{OT}_{2,n}, w_{r_1}, \dots, w_{r_\lambda})$ . Compute  $r_n^* \leftarrow \text{Explain}(\text{input}^*, \text{output}^*)$ .
4. When  $\mathcal{Z}$  corrupts party  $P_i$ , compute  $r_{R,i} := \text{SimOT}_2(1^\lambda, x_i, y_i, \text{st}_i)$ . Give  $x_i, z, y_i$ , and  $r_{R,i}$  to  $\mathcal{Z}$ . In addition, if  $i = n$  then for  $i \in [n-1]$  give  $\{y_i\}_{i=1}^{n-1}$  and  $r_n^*$  to  $\mathcal{Z}$ .

Computational indistinguishability between this experiment and the previous one follows from security of garbled circuits.

We conclude the proof by noting that Hybrid 6 is simply a syntactic rewriting of the ideal-world execution involving the simulator originally defined.

## 5 Conclusions and Open Questions

In this work we have shown the first constant-round, universally composable protocol tolerating a malicious, adaptive adversary that can corrupt any number of parties, in a setting where secure erasure is not assumed. In addition, we have shown the first adaptively secure protocol, regardless of round complexity, that can compute arbitrary functionalities (and not only adaptively well-formed ones) in the presence of any number of corruptions and without erasures.

Several interesting open questions remain. Although a CRS (or some other form of setup) is necessary if we wish to obtain a universally composable protocol with security against malicious adversaries corrupting an arbitrary number of parties, it is still possible that the CRS can be avoided in the semi-honest case, or in the stand-alone setting. Moreover, our protocol assumes that the CRS depends on the circuit  $C$  being computed or, if we let  $C$  be a universal circuit (cf. footnote 2), an a priori bound on the size of the circuit being computed. It would be interesting to see if this can be avoided.

## References

1. Boaz Barak, Ran Canetti, Jesper Buus Nielsen, and Rafael Pass. Universally composable protocols with relaxed set-up assumptions. In *45th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 186–195. IEEE, 2004.
2. Donald Beaver. Plug and play encryption. In *Advances in Cryptology—Crypto ’97*, volume 1294 of *LNCS*, pages 75–89. Springer, 1997.
3. Donald Beaver and Stuart Haber. Cryptographic protocols provably secure against dynamic adversaries. In *Advances in Cryptology—Eurocrypt ’92*, volume 658 of *LNCS*, pages 307–323. Springer, 1992.
4. Nir Bitansky, Ran Canetti, and Shai Halevi. Leakage-tolerant interactive protocols. In *9th Theory of Cryptography Conference—TCC 2012*, volume 7194 of *LNCS*, pages 266–284. Springer, 2012.

5. Nir Bitansky, Dana Dachman-Soled, and Huijia Lin. Leakage-tolerant computation with input-independent preprocessing. In *Advances in Cryptology—Crypto 2014, Part II*, volume 8617 of *LNCS*, pages 146–163. Springer, 2014.
6. Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *Advances in Cryptology—Asiacrypt 2013, Part II*, volume 8270 of *LNCS*, pages 280–300. Springer, 2013.
7. Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In *17th Intl. Conference on Theory and Practice of Public Key Cryptography—PKC 2014*, volume 8383 of *LNCS*, pages 501–519. Springer, 2014.
8. Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.
9. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 136–145. IEEE, 2001. Full version at <http://eprint.iacr.org/2000/067/>.
10. Ran Canetti, Ivan Damgård, Stefan Dziembowski, Yuval Ishai, and Tal Malkin. Adaptive versus non-adaptive security of multi-party protocols. *J. Crypto*, 17(3):153–207, 2004.
11. Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. Adaptively secure multiparty computation. In *28th Annual ACM Symposium on Theory of Computing (STOC)*, pages 639–648. ACM Press, 1996.
12. Ran Canetti and Marc Fischlin. Universally composable commitments. In *Advances in Cryptology—Crypto 2001*, volume 2139 of *LNCS*, pages 19–40. Springer, 2001.
13. Ran Canetti, Shafi Goldwasser, and Oxana Poburinnaya. Adaptively secure two-party computation from indistinguishability obfuscation, 2014. Cryptology ePrint Archive, Report 2014/845.
14. Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th Annual ACM Symposium on Theory of Computing (STOC)*, pages 494–503. ACM Press, 2002. Full version available at <http://eprint.iacr.org/2002/140>.
15. Ran Canetti, Rafael Pass, and Abhi Shelat. Cryptography from sunspots: How to use an imperfect reference string. In *48th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 249–259. IEEE, 2007.
16. Seung Geol Choi, Dana Dachman-Soled, Tal Malkin, and Hoeteck Wee. Improved non-committing encryption with applications to adaptively secure protocols. In *Advances in Cryptology—Asiacrypt 2009*, volume 5912 of *LNCS*, pages 287–302. Springer, 2009.
17. Dana Dachman-Soled, Tal Malkin, Mariana Raykova, and Muthuramakrishnan Venkitasubramaniam. Adaptive and concurrent secure computation from new adaptive, non-malleable commitments. In *Advances in Cryptology—Asiacrypt 2013, Part I*, volume 8269 of *LNCS*, pages 316–336. Springer, 2013.
18. Ivan Damgård and Jesper Buus Nielsen. Improved non-committing encryption schemes based on a general complexity assumption. In *Advances in Cryptology—Crypto 2000*, volume 1880 of *LNCS*, pages 432–450. Springer, 2000.
19. Ivan Damgård, Antigoni Polychroniadou, and Vanishree Rao. Secure UC constant round multi-party computation, 2014. Cryptology ePrint Archive, Report 2014/830.
20. 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 Symposium on Foundations of Computer Science (FOCS)*, pages 40–49. IEEE, 2013.

21. Sanjam Garg and Antigoni Polychroniadou. Two-round adaptively secure MPC from indistinguishability obfuscation, 2014. Cryptology ePrint Archive, Report 2014/844.
22. Sanjam Garg and Amit Sahai. Adaptively secure multi-party computation with dishonest majority. In *Advances in Cryptology—Crypto 2012*, volume 7417 of *LNCS*, pages 105–123. Springer, 2012.
23. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. In *19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229. ACM Press, 1987.
24. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. On the cryptographic applications of random functions. In *Advances in Cryptology—Crypto '84*, volume 196 of *LNCS*, pages 276–288. Springer, 1985.
25. Carmit Hazay and Arpita Patra. One-sided adaptively secure two-party computation. In *9th Theory of Cryptography Conference—TCC 2014*, volume 8349 of *LNCS*, pages 368–393. Springer, 2014.
26. Yuval Ishai, Abishek Kumarasubramanian, Claudio Orlandi, and Amit Sahai. On invertible sampling and adaptive security. In *Advances in Cryptology—Asiacrypt 2010*, volume 6477 of *LNCS*, pages 466–482. Springer, 2010.
27. Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer—efficiently. In *Advances in Cryptology—Crypto 2008*, volume 5157 of *LNCS*, pages 572–591. Springer, 2008.
28. Jonathan Katz and Rafail Ostrovsky. Round-optimal secure two-party computation. In *Advances in Cryptology—Crypto 2004*, volume 3152 of *LNCS*, pages 335–354. Springer, 2004.
29. Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In *20th ACM Conf. on Computer and Communications Security (CCS)*, pages 669–684. ACM Press, 2013.
30. Andrew Lindell. Adaptively secure two-party computation with erasures. In *Cryptographers' Track—RSA 2009*, LNCS, pages 117–132. Springer, 2009.
31. Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: Deniable encryption, and more. In *46th Annual ACM Symposium on Theory of Computing (STOC)*, pages 475–484. ACM Press, 2014.
32. Andrew C.-C. Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 162–167. IEEE, 1986.

## A Puncturable PRFs

Puncturable PRFs are a type of constrained PRF [6, 7, 29] whereby it is possible to generate a key that defines the function everywhere except on some polynomial-size set of inputs:

**Definition 2.** A puncturable family of PRFs is defined by polynomials  $n(\cdot)$  and  $m(\cdot)$  and a triple of Turing machines  $\text{Key}_F$ ,  $\text{Puncture}_F$ , and  $\text{Eval}_F$  satisfying the following conditions:

**Functionality preserved under puncturing.** For all polynomial-size sets  $S \subseteq \{0, 1\}^{n(\lambda)}$  and all  $x \in \{0, 1\}^{n(\lambda)} \setminus S$ , we have:

$$\Pr [K \leftarrow \text{Key}_F(1^\lambda); K_S = \text{Puncture}_F(K, S) : \text{Eval}_F(K, x) = \text{Eval}_F(K_S, x)] = 1.$$

**Pseudorandom at punctured points.** For every PPT adversary  $(A_1, A_2)$  such that  $A_1(1^\lambda)$  outputs a set  $S \subseteq \{0, 1\}^{n(\lambda)}$  and state  $\sigma$ , consider an experiment where  $K \leftarrow \text{Key}_F(1^\lambda)$  and  $K_S = \text{Puncture}_F(K, S)$ . Then we have

$$\left| \Pr [A_2(\sigma, K_S, S, \text{Eval}_F(K, S)) = 1] - \Pr [A_2(\sigma, K_S, S, U_{m(\lambda) \cdot |S|}) = 1] \right| \leq \text{negl}(\lambda)$$

where  $\text{Eval}_F(K, S)$  is the concatenation of  $\text{Eval}_F(K, x_1), \dots, \text{Eval}_F(K, x_k)$ , and  $S = \{x_1, \dots, x_k\}$  denoted the elements of  $S$  in lexicographic order.

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

As observed by [6, 7, 29], the GGM construction [24] of PRFs from one-way functions yields puncturable PRFs. Thus:

**Theorem 2.** [6, 7, 29] Assuming one-way functions exist, for all polynomials  $n(\lambda), m(\lambda)$  there is a puncturable PRF family that maps  $n(\lambda)$  bits to  $m(\lambda)$  bits.

We follow [31] for the following definitions of puncturable PRFs with enhanced properties:

**Definition 3.** A statistically injective (puncturable) PRF family with failure probability  $\epsilon(\cdot)$  is a family of (puncturable) PRFs  $F$  such that with probability  $1 - \epsilon(\lambda)$  over the random choice of key  $K \leftarrow \text{Key}_F(1^\lambda)$ , we have that  $F(K, \cdot)$  is injective.

**Definition 4.** An extracting (puncturable) PRF family with error  $\epsilon(\cdot)$  for min-entropy  $k(\cdot)$  is a family of (puncturable) PRFs  $F$  mapping  $n(\lambda)$  bits to  $m(\lambda)$  bits such that for all  $\lambda$ , if  $X$  is any distribution over  $n(\lambda)$  bits with min-entropy greater than  $k(\lambda)$ , then the statistical distance between  $(K \leftarrow \text{Key}_F(1^\lambda), F(K, X))$  and  $(K \leftarrow \text{Key}_F(1^\lambda), U_{m(\lambda)})$  is at most  $\epsilon(\lambda)$ .

The following results were proved in [31]:

**Theorem 3 ([31]).** If one-way functions exist, then for all efficiently computable functions  $n(\lambda)$ ,  $m(\lambda)$ , and  $e(\lambda)$  such that  $m(\lambda) \geq 2n(\lambda) + e(\lambda)$ , there exists a puncturable statistically injective PRF family with failure probability  $2^{-e(\lambda)}$  that maps  $n(\lambda)$  bits to  $m(\lambda)$  bits.

**Theorem 4.** If one-way functions exist, then for all efficiently computable functions  $n(\lambda)$ ,  $m(\lambda)$ ,  $k(\lambda)$ , and  $e(\lambda)$  such that  $n(\lambda) \geq k(\lambda) \geq m(\lambda) + 2e(\lambda) + 2$ , there exists an extracting puncturable PRF family that maps  $n(\lambda)$  bits to  $m(\lambda)$  bits with error  $2^{-e(\lambda)}$  for min-entropy  $k(\lambda)$ .

## B Proof of Security for Our Explainability Compiler

In this section we prove security of our explainability compiler. We must show two properties: statistical functional equivalence and explainability. (Polynomial slowdown is obvious.) The proof of statistical functional equivalence is largely identical to the analogous proof in [31], and is omitted. Instead, we focus on explainability.

We first state the following lemma, whose proof is the same as in [31].

**Lemma 1.** *Except with negligible probability over the choice of key  $K_2$ , the following hold:*

1. *For any fixed  $u[1] = \alpha$ , there exists at most one pair  $(\text{input}, \beta)$  such that the input  $\text{input}$  with randomness  $u = (\alpha, \beta)$  will cause the Step 1 check of  $\widetilde{\text{Alg}}$  to be satisfied.*
2. *There are at most  $2^{2\lambda + \ell_{\text{in}} + \ell_{\text{out}}}$  values for the randomness  $u$  that can cause the Step 1 check of  $\widetilde{\text{Alg}}$  to be satisfied.*

Given the above, we prove:

**Theorem 5.** *If  $F_1, F_2, F_3$  are PRFs that satisfy the properties specified in Section 3.1, and  $\text{iO}$  is an indistinguishability obfuscator for  $\text{P/poly}$ , then our construction  $\text{Comp}(\cdot, \cdot)$  satisfies explainability.*

*Proof.* Recall the explainability experiment from Definition 1:

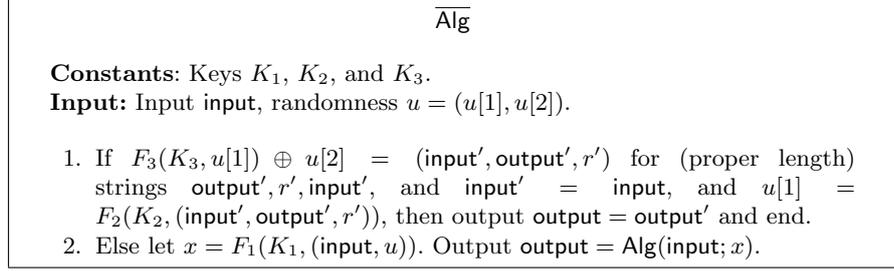
1.  $\mathcal{A}(1^\lambda)$  outputs  $\text{input}^*$  of its choice.
2.  $\text{Comp}(1^\lambda, \text{Alg})$  is run to obtain  $(\widetilde{\text{Alg}}, \text{Explain})$ .
3. Choose random coins  $r_0 \leftarrow \{0, 1\}^*$ , and compute  $\text{output}^* \leftarrow \widetilde{\text{Alg}}(\text{input}^*; r_0)$ .
4. Compute  $r_1 \leftarrow \text{Explain}(\text{input}^*, \text{output}^*)$ .
5. Choose a uniform bit  $b$  and give  $\widetilde{\text{Alg}}, \text{output}^*, r_b$  to  $\mathcal{A}$ .
6.  $\mathcal{A}$  outputs a bit  $b'$ , and succeeds if  $b' = b$ .

Let  $\text{Expl}_{\text{Alg}, \mathcal{A}}$  be a random variable set to 1 if  $\mathcal{A}$  succeeds in outputting  $b' = b$  in the above experiment. Security of  $\text{Comp}(1^\lambda, \text{Alg})$  requires that for every PPT  $\mathcal{A}$  and for every efficient algorithm  $\text{Alg}$ , we have  $\Pr[\text{Expl}_{\text{Alg}, \mathcal{A}} = 1] \leq 1/2 + \text{negl}(\lambda)$ .

Assume towards a contradiction that there is some PPT adversary  $\mathcal{A}$  and some efficient algorithm  $\text{Alg}$  such that  $\Pr[\text{Expl}_{\text{Alg}, \mathcal{A}} = 1] \geq 1/2 + \varepsilon(\lambda)$ , for non-negligible  $\varepsilon(\cdot)$ . We derive a contradiction via a sequence of hybrid experiments. The change between each experiment and the previous one will be denoted by underlined text.

**Original experiment.** We consider the probability that  $b' = b$  in the following experiment:

1.  $b \leftarrow \{0, 1\}$ .
2.  $\text{input}^* \leftarrow \mathcal{A}(1^\lambda)$ .
3. Choose  $K_1, K_2, K_3$  at random.

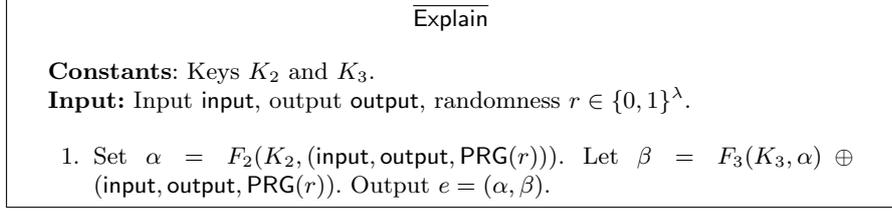
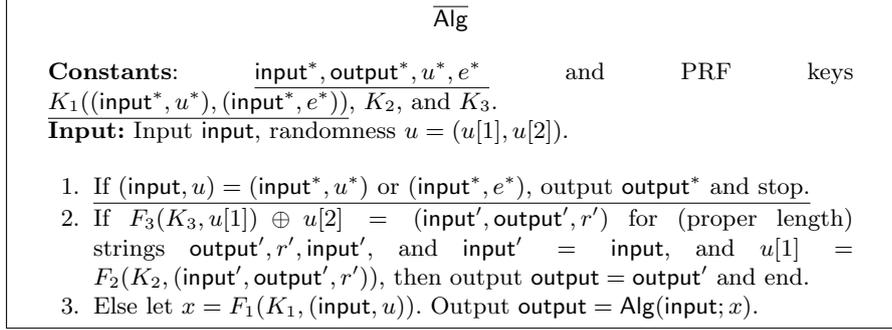
Fig. 5. Program  $\overline{\text{Alg}}$ 

4. Select  $u^*$  and  $r^*$  at random.
5. If  $F_3(K_3, u[1]) \oplus u[2] = (\text{input}', \text{output}', r')$  for (proper length) strings  $\text{output}', r', \text{input}'$ , and  $\text{input}' = \text{input}^*$ , and  $u[1] = F_2(K_2, (\text{input}', \text{output}', r'))$ , then let  $\text{output}^* = \text{output}'$  and jump to the next step. Otherwise, let  $x^* = F_1(K_1, (\text{input}^*, u^*))$  and  $\text{output}^* = \text{Alg}(\text{input}^*; x^*)$ .
6. Set  $\alpha^* = F_2(K_2, (\text{input}^*, \text{output}^*, \text{PRG}(r^*)))$ . Let  $\beta^* = F_3(K_3, \alpha^*) \oplus (\text{input}^*, \text{output}^*, \text{PRG}(r^*))$ , and set  $e^* = (\alpha^*, \beta^*)$ .
7. Let  $\widetilde{\text{Alg}} \leftarrow \text{iO}(\overline{\text{Alg}})$  for  $\overline{\text{Alg}}$  as in Figure 1. Let  $\widetilde{\text{Explain}} \leftarrow \text{iO}(\overline{\text{Explain}})$  for  $\overline{\text{Explain}}$  as in Figure 2.
8. If  $b = 0$ , set  $b' \leftarrow \mathcal{A}(\widetilde{\text{Alg}}, \text{output}^*, u^*)$ . If  $b = 1$ , set  $b' \leftarrow \mathcal{A}(\widetilde{\text{Alg}}, \text{output}^*, e^*)$ .

**Hybrid 0.** Next, we eliminate the check in Step 1 from the  $\overline{\text{Alg}}$  program when preparing the outputs of the fixed challenge  $\text{input}^*$ . Hybrid 0 is statistically close to the original experiment by Lemma 1. Consider the probability that  $b' = b$  in the following experiment:

1.  $b \leftarrow \{0, 1\}$ .
2.  $\text{input}^* \leftarrow \mathcal{A}(1^\lambda)$ .
3. Choose  $K_1, K_2, K_3$  at random.
4. Select  $u^*$  and  $r^*$  at random.
5. If  $F_3(K_3, u[1]) \oplus u[2] = (\text{input}', \text{output}', r')$  for (proper length) strings  $\text{output}', r', \text{input}'$ , and  $\text{input}' = \text{input}^*$ , and  $u[1] = F_2(K_2, (\text{input}', \text{output}', r'))$ , then let  $\text{output}^* = \text{output}'$  and jump to the next step. Otherwise, let  $x^* = F_1(K_1, (\text{input}^*, u^*))$  and  $\text{output}^* = \text{Alg}(\text{input}^*; x^*)$ .
6. Set  $\alpha^* = F_2(K_2, (\text{input}^*, \text{output}^*, \text{PRG}(r^*)))$ . Let  $\beta^* = F_3(K_3, \alpha^*) \oplus (\text{input}^*, \text{output}^*, \text{PRG}(r^*))$ , and set  $e^* = (\alpha^*, \beta^*)$ .
7. Let  $\widetilde{\text{Alg}} \leftarrow \text{iO}(\overline{\text{Alg}})$  for  $\overline{\text{Alg}}$  as in Figure 5. Let  $\widetilde{\text{Explain}} \leftarrow \text{iO}(\overline{\text{Explain}})$  for  $\overline{\text{Explain}}$  as in Figure 6.
8. If  $b = 0$ , set  $b' \leftarrow \mathcal{A}(\widetilde{\text{Alg}}, \text{output}^*, u^*)$ . If  $b = 1$ , set  $b' \leftarrow \mathcal{A}(\widetilde{\text{Alg}}, \text{output}^*, e^*)$ .

**Hybrid 1.** Here, we modify the  $\overline{\text{Alg}}$  program as follows: First, we add constants  $\text{input}^*, \text{output}^*, u^*, e^*$  to the program. Then, we add an “if” statement at the start that outputs  $\text{output}^*$  if the input is either  $(\text{input}^*, u^*)$  or  $(\text{input}^*, e^*)$ , as

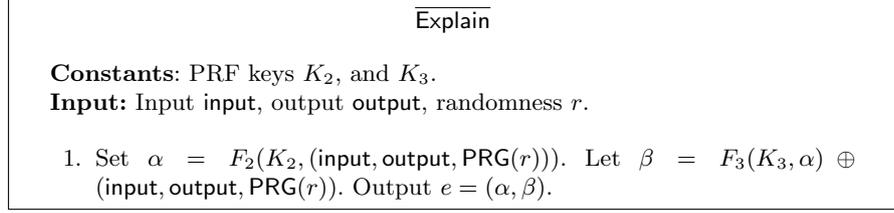
**Fig. 6.** Program  $\overline{\text{Explain}}$ **Fig. 7.** Program  $\overline{\text{Alg}}$ 

this is exactly what the original  $\overline{\text{Alg}}$  program would do by our choice of  $u^*, e^*$ . Because this “if” statement is in place, we know that  $F_1$  cannot be evaluated at either  $(\text{input}^*, u^*)$  or  $(\text{input}^*, e^*)$  within the program, and therefore we can safely puncture  $K_1$  at those two positions.

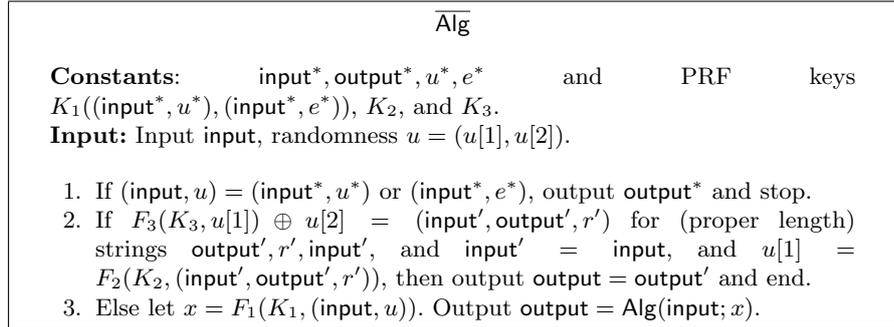
By construction, the new  $\overline{\text{Alg}}$  program is functionally equivalent to the original  $\overline{\text{Alg}}$  program. Therefore, indistinguishability of Hybrid 0 and Hybrid 1 follows by the security of  $i\mathcal{O}$ . Thus, the difference in the probabilities that  $\mathcal{A}$  outputs  $b' = b$  in Hybrid 0 and Hybrid 1 is negligible.

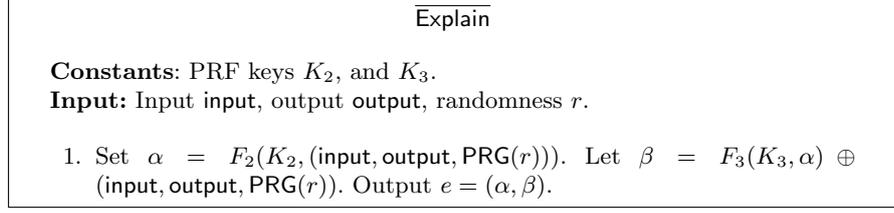
1.  $b \leftarrow \{0, 1\}$ .
2.  $\text{input}^* \leftarrow \mathcal{A}(1^\lambda)$ .
3. Choose  $K_1, K_2, K_3$  at random.
4. Select  $u^*$  and  $r^*$  at random.
5. Let  $x^* = F_1(K_1, (\text{input}^*, u^*))$  and let  $\text{output}^* = \text{Alg}(\text{input}^*; x^*)$ .
6. Set  $\alpha^* = F_2(K_2, (\text{input}^*, \text{output}^*, \text{PRG}(r^*)))$ . Let  $\beta^* = F_3(K_3, \alpha^*) \oplus (\text{input}^*, \text{output}^*, \text{PRG}(r^*))$ , and set  $e^* = (\alpha^*, \beta^*)$ .
7. Let  $\widetilde{\text{Alg}} \leftarrow i\mathcal{O}(\overline{\text{Alg}})$  for  $\overline{\text{Alg}}$  as in Figure 7. Let  $\widetilde{\text{Explain}} \leftarrow i\mathcal{O}(\overline{\text{Explain}})$  for  $\overline{\text{Explain}}$  as in Figure 8.
8. If  $b = 0$ , set  $b' \leftarrow \mathcal{A}(\widetilde{\text{Alg}}, \text{output}^*, u^*)$ . If  $b = 1$ , set  $b' \leftarrow \mathcal{A}(\widetilde{\text{Alg}}, \text{output}^*, e^*)$ .

**Hybrid 2.** Here, the value  $x^*$  is chosen uniformly instead of as the output of  $F_1(K_1, (\text{input}^*, u^*))$ . Pseudorandomness of  $F_1$  thus implies that the difference in the probabilities that  $\mathcal{A}$  outputs  $b' = b$  in Hybrid 1 and Hybrid 2 is negligible.

**Fig. 8.** Program  $\overline{\text{Explain}}$ 

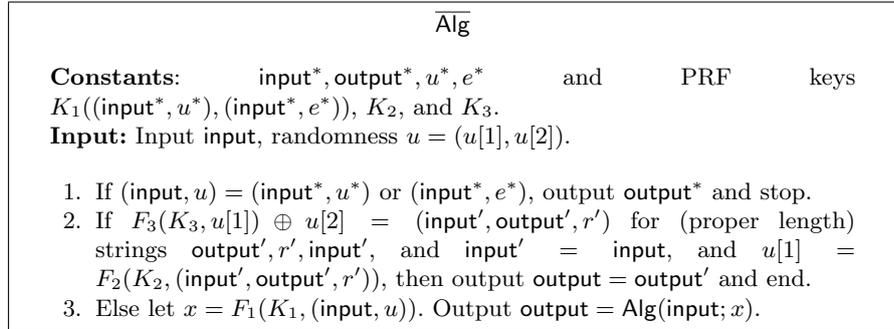
1.  $b \leftarrow \{0, 1\}$ .
2.  $\text{input}^* \leftarrow \mathcal{A}(1^\lambda)$ .
3. Choose  $K_1, K_2, K_3$  at random.
4. Select  $u^*$  and  $r^*$  at random.
5. Choose uniform  $x^*$  and let  $\text{output}^* = \text{Alg}(\text{input}^*; x^*)$ .
6. Set  $\alpha^* = F_2(K_2, (\text{input}^*, \text{output}^*, \text{PRG}(r^*)))$ . Let  $\beta^* = F_3(K_3, \alpha^*) \oplus (\text{input}^*, \text{output}^*, \text{PRG}(r^*))$ , and set  $e^* = (\alpha^*, \beta^*)$ .
7. Let  $\widetilde{\text{Alg}} \leftarrow \text{iO}(\widetilde{\text{Alg}})$  for  $\widetilde{\text{Alg}}$  as in Figure 9. Let  $\overline{\text{Explain}} \leftarrow \text{iO}(\overline{\text{Explain}})$  for  $\overline{\text{Explain}}$  as in Figure 10.
8. If  $b = 0$ , set  $b' \leftarrow \mathcal{A}(\widetilde{\text{Alg}}, \text{output}^*, u^*)$ . If  $b = 1$ , set  $b' \leftarrow \mathcal{A}(\widetilde{\text{Alg}}, \text{output}^*, e^*)$ .

**Fig. 9.** Program  $\overline{\text{Alg}}$

Fig. 10. Program  $\overline{\text{Explain}}$ 

**Hybrid 3.** Here, instead of choosing uniform  $r^*$  and applying a PRG to it, a value  $\tilde{r}$  is chosen uniformly from the range of the PRG. Security of the PRG implies that the difference in the probabilities that  $\mathcal{A}$  outputs  $b' = b$  in Hybrid 2 and Hybrid 3 is negligible.

1.  $b \leftarrow \{0, 1\}$ .
2.  $\text{input}^* \leftarrow \mathcal{A}(1^\lambda)$ .
3. Choose  $K_1, K_2, K_3$  at random.
4. Select  $u^*$  and  $\tilde{r}$  at random.
5. Choose uniform  $x^*$  and let  $\text{output}^* = \text{Alg}(\text{input}^*; x^*)$ .
6. Set  $\alpha^* = F_2(K_2, (\text{input}^*, \text{output}^*, \tilde{r}))$ . Let  $\beta^* = F_3(K_3, \alpha^*) \oplus (\text{input}^*, \text{output}^*, \tilde{r})$ , and set  $e^* = (\alpha^*, \beta^*)$ .
7. Let  $\widetilde{\text{Alg}} \leftarrow \text{iO}(\overline{\text{Alg}})$  for  $\overline{\text{Alg}}$  as in Figure 11. Let  $\widetilde{\text{Explain}} \leftarrow \text{iO}(\overline{\text{Explain}})$  for  $\overline{\text{Explain}}$  as in Figure 12.
8. If  $b = 0$ , set  $b' \leftarrow \mathcal{A}(\widetilde{\text{Alg}}, \text{output}^*, u^*)$ . If  $b = 1$ , set  $b' \leftarrow \mathcal{A}(\widetilde{\text{Alg}}, \text{output}^*, e^*)$ .

Fig. 11. Program  $\overline{\text{Alg}}$ 

**Hybrid 4.** Here, the  $\overline{\text{Alg}}$  and  $\overline{\text{Explain}}$  programs are modified as shown below. In Lemma 2, (proven below), we argue that except with negligible probability over choice of constants, these modifications do not alter the functionality of either program. Thus, the  $\text{iO}$  security property implies that the difference in the probabilities that  $\mathcal{A}$  outputs  $b' = b$  in Hybrid 3 and Hybrid 4 is negligible.

$\overline{\text{Explain}}$
<p><b>Constants:</b> PRF keys <math>K_2</math>, and <math>K_3</math>.</p> <p><b>Input:</b> Input <math>\text{input}</math>, output <math>\text{output}</math>, randomness <math>r</math>.</p> <p>1. Set <math>\alpha = F_2(K_2, (\text{input}, \text{output}, \text{PRG}(r)))</math>. Let <math>\beta = F_3(K_3, \alpha) \oplus (\text{input}, \text{output}, \text{PRG}(r))</math>. Output <math>e = (\alpha, \beta)</math>.</p>

Fig. 12. Program  $\overline{\text{Explain}}$ 

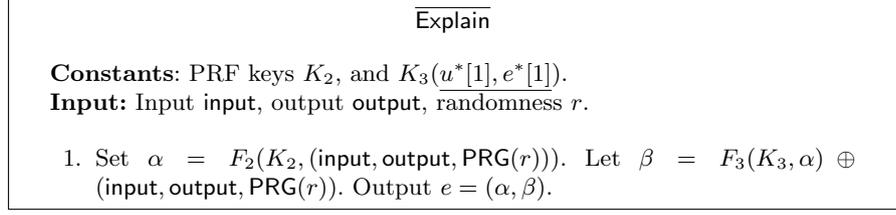
1.  $b \leftarrow \{0, 1\}$ .
2.  $\text{input}^* \leftarrow \mathcal{A}(1^\lambda)$ .
3. Choose  $K_1, K_2, K_3$  at random.
4. Select  $u^*$  and  $\tilde{r}$  at random.
5. Choose uniform  $x^*$  and let  $\text{output}^* = \text{Alg}(\text{input}^*; x^*)$ .
6. Set  $\alpha^* = F_2(K_2, (\text{input}^*, \text{output}^*, \tilde{r}))$ . Let  $\beta^* = F_3(K_3, \alpha^*) \oplus (\text{input}^*, \text{output}^*, \tilde{r})$ , and set  $e^* = (\alpha^*, \beta^*)$ .
7. Let  $\widetilde{\text{Alg}} \leftarrow \text{iO}(\overline{\text{Alg}})$  for  $\overline{\text{Alg}}$  as in Figure 13. Let  $\overline{\text{Explain}} \leftarrow \text{iO}(\overline{\text{Explain}})$  for  $\overline{\text{Explain}}$  as in Figure 14.
8. If  $b = 0$ , set  $b' \leftarrow \mathcal{A}(\widetilde{\text{Alg}}, \text{output}^*, u^*)$ . If  $b = 1$ , set  $b' \leftarrow \mathcal{A}(\widetilde{\text{Alg}}, \text{output}^*, e^*)$ .

$\overline{\text{Alg}}$
<p><b>Constants:</b> <math>\text{input}^*, \text{output}^*, u^*, e^*</math> and PRF keys <math>K_1((\text{input}^*, u^*), (\text{input}^*, e^*))</math>, <math>K_2</math>, and <math>K_3(u^*[1], e^*[1])</math>.</p> <p><b>Input:</b> Input <math>\text{input}</math>, randomness <math>u = (u[1], u[2])</math>.</p> <p>1. If <math>(\text{input}, u) = (\text{input}^*, u^*)</math> or <math>(\text{input}^*, e^*)</math>, output <math>\text{output}^*</math> and stop.</p> <p>2. If <math>u[1] = e^*[1]</math> or <math>u[1] = u^*[1]</math>, then skip this step. If <math>F_3(K_3, u[1]) \oplus u[2] = (\text{input}', \text{output}', r')</math> for (proper length) strings <math>\text{output}', r', \text{input}'</math>, and <math>\text{input}' = \text{input}</math>, and <math>u[1] = F_2(K_2, (\text{input}', \text{output}', r'))</math>, then output <math>\text{output} = \text{output}'</math> and end.</p> <p>3. Else let <math>x = F_1(K_1, (\text{input}, u))</math>. Output <math>\text{output} = \text{Alg}(\text{input}; x)</math>.</p>

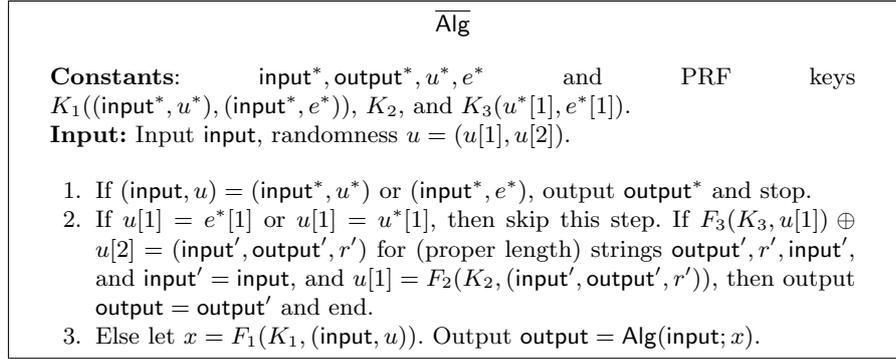
Fig. 13. Program  $\overline{\text{Alg}}$ 

**Hybrid 5.** Here, the value  $e^*[2]$ , denoted  $\beta^*$ , is chosen at random instead of being chosen as  $\beta^* = F_3(K_3, \alpha^*) \oplus (\text{input}^*, \text{output}^*, \tilde{r})$ . Pseudorandomness of  $F_3$  thus implies that the difference in the probabilities that  $\mathcal{A}$  outputs  $b' = b$  in Hybrid 4 and Hybrid 5 is negligible.

1.  $b \leftarrow \{0, 1\}$ .
2.  $\text{input}^* \leftarrow \mathcal{A}(1^\lambda)$ .
3. Choose  $K_1, K_2, K_3$  at random.

**Fig. 14.** Program  $\overline{\text{Explain}}$ 

4. Select  $u^*$  and  $\tilde{r}$  at random.
5. Choose uniform  $x^*$  and let  $\text{output}^* = \text{Alg}(\text{input}^*; x^*)$ .
6. Set  $\alpha^* = F_2(K_2, (\text{input}^*, \text{output}^*, \tilde{r}))$ . Choose uniform  $\beta^*$ , and set  $e^* = (\alpha^*, \beta^*)$ .
7. Let  $\widetilde{\text{Alg}} \leftarrow \text{iO}(\overline{\text{Alg}})$  for  $\overline{\text{Alg}}$  as in Figure 15. Let  $\widetilde{\text{Explain}} \leftarrow \text{iO}(\overline{\text{Explain}})$  for  $\overline{\text{Explain}}$  as in Figure 16.
8. If  $b = 0$ , set  $b' \leftarrow \mathcal{A}(\widetilde{\text{Alg}}, \text{output}^*, u^*)$ . If  $b = 1$ , set  $b' \leftarrow \mathcal{A}(\widetilde{\text{Alg}}, \text{output}^*, e^*)$ .

**Fig. 15.** Program  $\overline{\text{Alg}}$ 

**Hybrid 6.** First we modify the  $\overline{\text{Alg}}$  program to add a condition to the check in Step 2 to determine if  $(\text{input}', \text{output}', r') = (\text{input}^*, \text{output}^*, \tilde{r})$  and, if so, to skip this check. This does not change the functionality of the program, because  $e^*[1] = F_2(K_2, (\text{input}^*, \text{output}^*, \tilde{r}))$ , and therefore the check cannot be satisfied if  $(\text{input}', \text{output}', r') = (\text{input}^*, \text{output}^*, \tilde{r})$ , since Step 2 is skipped entirely if  $u[1] = e^*[1]$ . Furthermore, both the  $\overline{\text{Alg}}$  and  $\overline{\text{Explain}}$  programs are modified to have  $K_2$  punctured at the points  $(\text{input}^*, \text{output}^*, \tilde{r})$ . This puncturing does not change the functionality of the  $\overline{\text{Alg}}$  program because of the new “if” condition just added. With overwhelming probability,  $\tilde{r}$  is not in the image of the PRG and therefore this puncturing also does not change the functionality of the  $\overline{\text{Explain}}$  program. Thus, the difference in the probabilities that  $\mathcal{A}$  outputs  $b' = b$  in Hybrid 5 and Hybrid 6 is negligible.

$\overline{\text{Explain}}$

**Constants:** PRF keys  $K_2$ , and  $K_3(u^*[1], e^*[1])$ .  
**Input:** Input  $\text{input}$ , output  $\text{output}$ , randomness  $r$ .

1. Set  $\alpha = F_2(K_2, (\text{input}, \text{output}, \text{PRG}(r)))$ . Let  $\beta = F_3(K_3, \alpha) \oplus (\text{input}, \text{output}, \text{PRG}(r))$ . Output  $e = (\alpha, \beta)$ .

Fig. 16. Program  $\overline{\text{Explain}}$ 

1.  $b \leftarrow \{0, 1\}$ .
2.  $\text{input}^* \leftarrow \mathcal{A}(1^\lambda)$ .
3. Choose  $K_1, K_2, K_3$  at random.
4. Select  $u^*$  and  $\tilde{r}$  at random.
5. Choose uniform  $x^*$  and let  $\text{output}^* = \text{Alg}(\text{input}^*; x^*)$ .
6. Set  $\alpha^* = F_2(K_2, (\text{input}^*, \text{output}^*, \tilde{r}))$ . Choose uniform  $\beta^*$ , and set  $e^* = (\alpha^*, \beta^*)$ .
7. Let  $\overline{\text{Alg}} \leftarrow \text{iO}(\overline{\text{Alg}})$  for  $\overline{\text{Alg}}$  as in Figure 17. Let  $\overline{\text{Explain}} \leftarrow \text{iO}(\overline{\text{Explain}})$  for  $\overline{\text{Explain}}$  as in Figure 18.
8. If  $b = 0$ , set  $b' \leftarrow \mathcal{A}(\overline{\text{Alg}}, \text{output}^*, u^*)$ . If  $b = 1$ , set  $b' \leftarrow \mathcal{A}(\overline{\text{Alg}}, \text{output}^*, e^*)$ .

$\overline{\text{Alg}}$

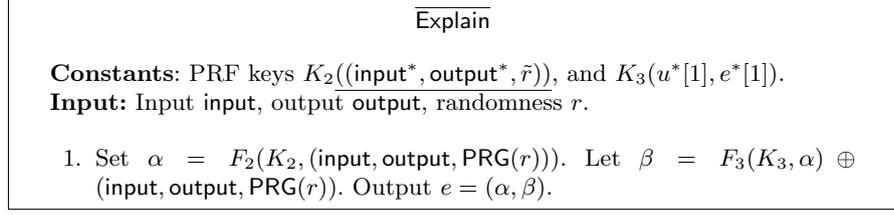
**Constants:**  $\text{input}^*, \text{output}^*, u^*, e^*, \tilde{r}$  and PRF keys  $K_1((\text{input}^*, u^*), (\text{input}^*, e^*))$ ,  $K_2((\text{input}^*, \text{output}^*, \tilde{r}))$ , and  $K_3(u^*[1], e^*[1])$ .  
**Input:** Input  $\text{input}$ , randomness  $u = (u[1], u[2])$ .

1. If  $(\text{input}, u) = (\text{input}^*, u^*)$  or  $(\text{input}^*, e^*)$ , output  $\text{output}^*$  and stop.
2. If  $u[1] = e^*[1]$  or  $u[1] = u^*[1]$ , then skip this step. If  $F_3(K_3, u[1]) \oplus u[2] = (\text{input}', \text{output}', r')$  for (proper length) strings  $\text{output}', r', \text{input}'$ , and  $\text{input}' = \text{input}$ , and  $(\text{input}', \text{output}', r') \neq (\text{input}^*, \text{output}^*, \tilde{r})$ , then also check if  $u[1] = F_2(K_2, (\text{input}', \text{output}', r'))$ , then output  $\text{output} = \text{output}'$  and end.
3. Else let  $x = F_1(K_1, (\text{input}, u))$ . Output  $\text{output} = \text{Alg}(\text{input}; x)$ .

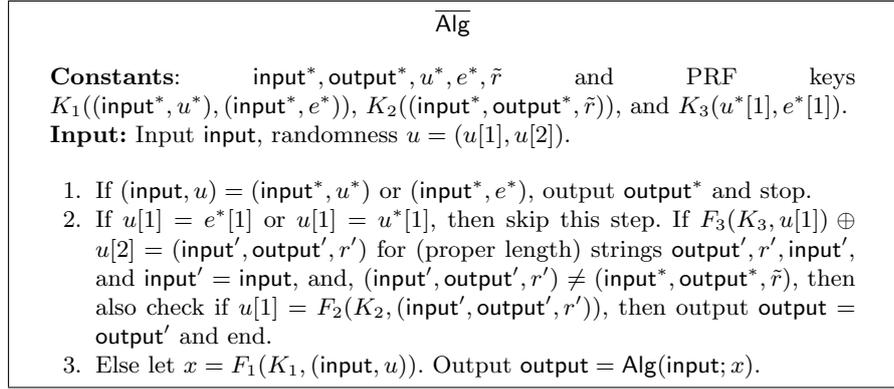
Fig. 17. Program  $\overline{\text{Alg}}$ 

**Hybrid 7.** Finally, we modify  $e^*[1]$ , denoted  $\alpha^*$ , to be uniform, instead of being computed as  $\alpha^* = F_2(K_2, (\text{input}^*, \text{output}^*, \tilde{r}))$ . Pseudorandomness of  $F_2$  implies that the difference in the probabilities that  $\mathcal{A}$  outputs  $b' = b$  in Hybrid 6 and Hybrid 7 is negligible.

1.  $b \leftarrow \{0, 1\}$ .
2.  $\text{input}^* \leftarrow \mathcal{A}(1^\lambda)$ .
3. Choose  $K_1, K_2, K_3$  at random.

**Fig. 18.** Program  $\overline{\text{Explain}}$ 

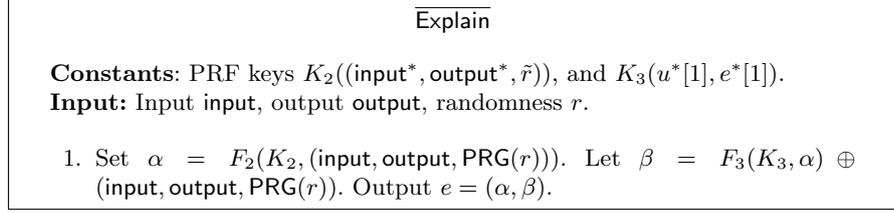
4. Select  $u^*$  and  $\tilde{r}$  at random.
5. Choose uniform  $x^*$  and let  $\text{output}^* = \text{Alg}(\text{input}^*; x^*)$ .
6. Choose uniform  $\alpha^*$  and  $\beta^*$ , and set  $e^* = (\alpha^*, \beta^*)$ .
7. Let  $\widetilde{\text{Alg}} \leftarrow i\mathcal{O}(\overline{\text{Alg}})$  for  $\overline{\text{Alg}}$  as in Figure 19. Let  $\widetilde{\text{Explain}} \leftarrow i\mathcal{O}(\overline{\text{Explain}})$  for  $\overline{\text{Explain}}$  as in Figure 20.
8. If  $b = 0$ , set  $b' \leftarrow \mathcal{A}(\widetilde{\text{Alg}}, \text{output}^*, u^*)$ . If  $b = 1$ , set  $b' \leftarrow \mathcal{A}(\widetilde{\text{Alg}}, \text{output}^*, e^*)$ .

**Fig. 19.** Program  $\overline{\text{Alg}}$ 

In Hybrid 7 we observe that the variables  $e^*, u^*$  are now uniform and independent. Thus, the inputs to  $\mathcal{A}$  are distributed identically regardless of whether  $b = 0$  or  $b = 1$  are identical, and so  $b = b'$  with probability exactly 1/2. The lemma below thus concludes the proof.

The proof above relies on the following lemma showing that the programs obfuscated in Hybrid 3 are equivalent to the corresponding programs in Hybrid 4.

**Lemma 2.** *Except with negligible probability over the choice of  $u^*[1]$  and  $e^*[1]$ , the  $\overline{\text{Alg}}$  and  $\overline{\text{Explain}}$  programs in Hybrid 4 are equivalent to the  $\widetilde{\text{Alg}}$  and  $\widetilde{\text{Explain}}$  programs in Hybrid 3.*

**Fig. 20.** Program  $\overline{\text{Explain}}$ 

*Proof.* We consider below each change to the programs.

First, an “if” statement is added to Step 2 of the  $\overline{\text{Alg}}$  program, to skip the check in Step 2 if either  $u[1] = e^*[1]$  or  $u[1] = u^*[1]$ . To see why this change does not affect the functionality of the program, let us consider each case in turn. By Lemma 1, if  $u[1] = e^*[1]$ , then the only way the Step 2 check can be satisfied is if  $\text{input} = \text{input}^*$  and  $u[2] = e^*[2]$ . But this case is already handled in Step 1; therefore, skipping Step 2 if  $u[1] = e^*[1]$  does not affect the functionality of the program. On the other hand, recall that  $u^*[1]$  is chosen at random, and therefore the probability that  $u^*[1]$  is in the image of  $F_2(K_2, \cdot)$  is negligible. Thus, with overwhelming probability over the choice of constants  $u^*[1]$ , the check in Step 2 cannot be satisfied if  $u[1] = u^*[1]$ . Therefore, the addition of this “if” statement does not alter the functionality of the  $\overline{\text{Alg}}$  program.

Also, the key  $K_3$  is punctured at  $u^*[1], e^*[1]$  in both the  $\overline{\text{Alg}}$  and  $\overline{\text{Explain}}$  programs. The new “if” statement above ensures that  $F_3(K_3, \cdot)$  is never called at these values in the  $\overline{\text{Alg}}$  program. Recall that the  $\overline{\text{Explain}}$  program only calls  $F_3(K_3, \cdot)$  on values computed as  $F_2(K_2, (\text{input}, \text{output}, \text{PRG}(r)))$  for some bit  $\text{input}$  and strings  $\text{output}$  and  $r$ . Furthermore,  $F_2$  is statistically injective with a very sparse image set, by our choice of parameters. Since every  $u^*[1]$  is randomly chosen, it is very unlikely to be in the image of  $F_2(K_2, \cdot)$ . Since every  $e^*[1]$  is chosen based on a random  $\tilde{r}$  value instead of a PRG output, it is very unlikely to correspond to  $F_2(K_2, (\text{input}, \text{output}, \text{PRG}(r)))$  for any  $(\text{input}, \text{output}, r)$ . Thus, these values are not called by the  $\overline{\text{Explain}}$  program, except with negligible probability over the choice of these constants  $u^*[1]$  and  $e^*[1]$ .