Distributed (Correlation) Samplers: How to Remove a Trusted Dealer in One Round*

Damiano Abram, Peter Scholl, and Sophia Yakoubov

Aarhus University, Aarhus, Denmark

Abstract. Structured random strings (SRSs) and correlated randomness are important for many cryptographic protocols. In settings where interaction is expensive, it is desirable to obtain such randomness in as few rounds of communication as possible; ideally, simply by exchanging one reusable round of messages which can be considered public keys.
In this paper, we describe how to generate any SRS or correlated randomness in such a single round of communication, using, among other things, indistinguishability obfuscation. We introduce what we call a *distributed sampler*, which enables n parties to sample a single public value (SRS) from any distribution. We construct a semi-malicious distributed sampler in the plain model, and use it to build a semi-malicious *public-key PCF* (Boyle *et al.*, FOCS 2020) in the plain model. A public-key PCF can be thought of as a distributed *correlation* sampler; instead of producing a public SRS, it gives each party a private random value (where the values satisfy some correlation).
We introduce a general technique called an *anti-rusher* which compiles any one-round protocol with semi-malicious security without inputs to a similar one-round protocol with active security by making use of a programmable random oracle. This gets us actively secure distributed samplers and public-key PCFs in the random oracle model.
Finally, we explore some tradeoffs. Our first PCF construction is limited to *reverse-sampleable* correlations (where the random outputs of honest parties must be simulatable given the random outputs of corrupt parties); we additionally show a different construction without this limitation, but which does not allow parties to hold secret parameters of the correlation. We also describe how to avoid the use of a random oracle at the cost of relying on sub-exponentially secure indistinguishability obfuscation.

* Supported by the Independent Research Fund Denmark (DFF) under project number 0165-00107B (C3PO), the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No 803096 (SPEC), and a starting grant from Aarhus University Research Foundation.

Table of Contents

1 Introduction

Randomness is crucial for many cryptographic protocols. Participants can generate some randomness locally (e.g. by flipping coins), but the generation of other forms of randomness is more involved. For instance, a *uniform reference string* (URS) must be produced in such a way that a coalition of corrupt protocol participants — controlled by the adversary — cannot bias it too much. Even more complex is the generation of a *structured* reference string (SRS, such as an RSA modulus), which can depend on secrets that should not be known to anyone. For instance, constructions such as cryptographic accumulators [Bd94] use an RSA modulus whose factorization is known to nobody, while many succinct zero-knowledge proofs such as SNARKs [BCCT12] require a more complex form of SRS.

In contrast to common reference strings, which are public, some protocols demand *correlated randomness*, where each participant holds a secret random value, but because the values must satisfy some relationship, they cannot be generated locally by the participants. An example of correlated randomness is random oblivious transfer, where one participant has a list of random strings, and another has one of those strings as well as its index in the list. Such correlated randomness often allows cryptographic protocols to run with a more efficient online phase.

Typically, in order to set up an SRS or correlated randomness without making additional trust assumptions, the parties must run a secure multi-party computation protocol, which takes several rounds of interaction. This is the case, for instance, in "setup ceremonies" [BGG19, BGM17] that have been designed to generate trusted SNARK parameters for applications. In this paper, we explore techniques that let parties sample *any* common reference string or correlation in just *one round* of interaction.

1.1 Related Work

There are a number of lines of work that can be used to generate randomness in different ways.

Universal samplers. A universal sampler [HJK+16] is a kind of SRS which can be used to obliviously sample from any distribution that has an efficient sampling algorithm. That is, after a one-time trusted setup to generate the universal sampler, it can be used to generate arbitrary other SRSs. Hofheinz *et al.* [HJK+16] show how to build universal samplers from indistinguishability obfuscation and a random oracle, while allowing an unbounded number of adaptive queries. They also show how to build weaker forms of universal sampler in the standard model, from single-key functional encryption [LZ17]. A universal sampler is a very powerful tool, but in many cases impractical, due to the need for a trusted setup.

Non-interactive multiparty computation (NIMPC). Non-interactive multiparty computation (NIMPC, [BGI+14a]) is a kind of one-round protocol that allows

n parties to compute any function of their secret inputs in just one round of communication. However, NIMPC requires that the parties know one another's public keys before that one round, so there is another implicit round of communication.[1] NIMPC for general functions can be constructed based on subexponentially-secure indistinguishability obfuscation [HIJ+17].

Spooky encryption. Spooky encryption [DHRW16] is a kind of encryption which enables parties to learn joint functions of ciphertexts encrypted under independent public keys (given one of the corresponding secret keys). In order for semantic security to hold, what party i learns using her secret key should reveal nothing about the value encrypted to party j's public key; so, spooky encryption only supports the evaluation of *non-signaling* functions. An example of a non-signaling function is any function where the parties' outputs are an additive secret sharing. Dodis *et al.* [DHRW16] show how to build spooky encryption for any such additive function from the LWE assumption with a URS (this also implies multi-party homomorphic secret sharing for general functions). In the two-party setting, they also show how to build spooky encryption for a larger class of non-signaling functions from (among other things) sub-exponentially hard indistinguishability obfuscation.

Pseudorandom Correlation Generators and Functions (PCGs and PCFs). Pseudorandom correlation generators [BCG+19a, BCG+19b, BCG+20b] and functions [BCG+20a, OSY21] let parties take a small amount of specially correlated randomness (called the *seed* randomness) and expand it non-interactively, obtaining a large sample from a target correlation. Pseudorandom correlation generators (PCGs) support only a fixed, polynomial expansion; pseudorandom correlation functions (PCFs) allow the parties to produce exponentially many instances of the correlation (via evaluation of the function on any of exponentially many inputs).

PCGs and PCFs can be built for any *additively secret shared* correlation (where the parties obtain additive shares of a sample from some distribution) using LWE-based spooky encryption mentioned above. Similarly, with two parties, we can build PCGs and PCFs for more general *reverse-samplable* correlations by relying on spooky encryption from subexponentially secure iO. PCGs and PCFs with better concrete efficiency can be obtained under different flavours of the LPN assumption, for simpler correlations such as vector oblivious linear evaluation [BCGI18], oblivious transfer [BCG+19b] and others [BCG+20b, BCG+20a].

Of course, in order to use PCGs or PCFs, the parties must somehow get the correlated seed randomness. *Public-key* PCGs and PCFs allow the parties to instead derive outputs using their independently generated public keys, which can be published in a single round of communication. The above, spooky

[1] This requirement is inherent; otherwise, an adversary would be able to take the message an honest party sent, and recompute the function with that party's input while varying the other inputs. NIMPC does allow similar recomputation attacks, but only with *all* honest party inputs fixed, which a PKI can be used to enforce.

encryption-based PCGs and PCFs are public-key, while the LPN-based ones are not. Public-key PCFs for OT and vector-OLE were recently built based on DCR and QR [OSY21]; however, these require a structured reference string consisting of a public RSA modulus with hidden factorization.

1.2 Our Contributions

In this paper, we leverage indistinguishability obfuscation to build public-key PCFs for *any* correlation. On the way to realizing this, we define several other primitives, described in Fig. 1. One of these primitives is a *distributed sampler*, which is a weaker form of public-key PCF which only allows the sampling of public randomness. (A public-key PCF can be thought of as a distributed *correlation* sampler.) Our constructions, and the assumptions they use, are mapped out in Fig. 2. We pay particular attention to avoiding the use of sub-exponentially secure primitives where possible (which rules out strong tools such as probabilistic iO [CLTV15]).

Primitive	**Distribution**	**Output**
Distributed Sampler (DS, Def. 3.1)	fixed	public
Reusable Distributed Universal Sampler (Def. 7.6)	any	public
Public-key PCF (pk-PCF, [OSY21])	fixed, reverse-samplable	private
Ideal pk-PCF (Def. 7.2)	any	private

Fig. 1. In this table we describe one-round n-party primitives that can be used for sampling randomness. They differ in terms of whether a given execution enables sampling from *any* distribution (or just a fixed one), and in terms of whether they only output public randomness (in the form of a URS or SRS) or also return private correlated randomness to the parties.

We begin by exploring constructions secure against semi-malicious adversaries, where corrupt parties are assumed to follow the protocol other than in their choice of random coins. We build a semi-malicious distributed sampler, and use it to build a semi-malicious public-key PCF. We then compile those protocols to be secure against active adversaries. This leads to a public-key PCF that requires a random oracle, and supports the broad class of *reverse-sampleable* correlations (where, given only corrupt parties' values in a given sample, honest parties' values can be simulated in such a way that they are indistinguishable from the ones in the original sample).

We also show two other routes to public-key PCFs with active security. One of these avoids the use of a random oracle, but requires sub-exponentially secure building blocks. The other requires a random oracle, but can support general correlations, not just reverse-sampleable ones. (The downside is that it does not support correlations *with master secrets*, which allow parties to have secret input parameters to the correlation.)

These are valuable trade-offs, as described below.

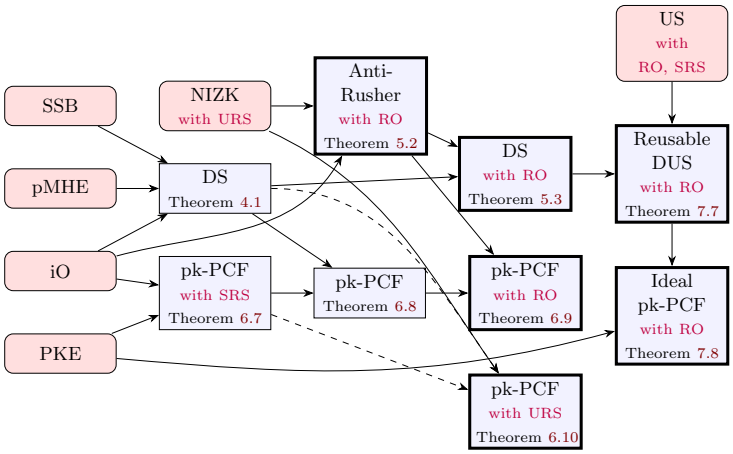

Fig. 2. In this table we describe the constructions in this paper. In pink are assumptions: they include somewhere statistically binding hash functions (SSB), multiparty homomorphic encryption with private evaluation (pMHE [AJJM20], a weaker form of multi-key FHE), indistinguishability obfuscation (iO), non-interactive zero knowledge proofs (NIZK), and universal samplers (US). In blue are constructions of distributed samplers (DS, Def. 3.1), reusable distributed universal samplers (reusable DUS, Def. 7.6) and public-key pseudorandom correlation functions (pk-PCFs, [OSY21]). Constructions with bold outlines are secure against active adversaries; the rest are secure against semi-malicious adversaries. In magenta are necessary setup assumptions. (Note that the availability of a random oracle (RO) immediately implies the additional availability of a URS.) Dashed lines denote the use of sub-exponentially secure tools.

On the importance of realizing general correlations: There are many valuable correlation that are *not* reverse-sampleable. An example of such a correlation gives a garbled circuit to one party, and all the wire labels to another. Since the labels cannot be reverse-sampled from the circuit (without violating the security properties of the garbling scheme), such a correlation is not reverse-sampleable; however, it is a valuable form of pre-processing for secure two-party computation.

On the importance of avoiding a random oracle: Random oracles are an idealized assumption with no known realization; there is a large gap between a random oracle and a hash function, which is often substituted for a random oracle in practice.

On the importance of avoiding sub-exponential assumptions: It may seem strange to want to avoid sub-exponentially secure primitives,[2] when many candidates for indistinguishability obfuscation itself are based on subexponential assumptions [JLS21]. However, despite informal arguments [LZ17], this is not known to be inherent: earlier iO candidates are based on polynomial hardness [GGH+13] (albeit for an exponential family of assumptions), and in future we may obtain iO from a single, polynomial hardness assumption. In general, it is always preferable to require a weaker form of security from a primitive, and this also leads to better parameters in practice. The problem of removing sub-exponential assumptions from iO, or applications of iO, has been studied previously in various settings [GPSZ17, LZ17].

1.3 Technical Overview

Distributed Samplers We start by introducing a new tool called a *distributed sampler* (DS, Section 3). A distributed sampler allows n parties to sample a single, public output from an efficiently sampleable distribution $\mathcal{D}$ with just one round of communication (which is modelled by the exchange of public keys).

Semi-malicious distributed samplers. We use multiparty homomorphic encryption with private evaluation (pMHE [AJJM20], a weaker, setup-free version of multi-key FHE) and indistinguishability obfuscation to build semi-malicious distributed samplers in the plain model (Section 4). In our distributed sampler construction, all parties can compute an encryption of the sample from everyones' public keys (using, among other things, the homomorphic properties of the encryption scheme), and then use an obfuscated program in party i's public key to get party i's partial decryption of the sample. The partial decryptions can then be combined to recover the sample itself. The tricky thing is that, in the proof, we must ensure that we can replace the real sample with an ideal sample. To do this, we must remove all information about the real sample from the public keys. However, pMHE secret keys are not *puncturable*; that is, there is no way to ensure that they do not reveal any information about the contents of one cipher-

[2] By sub-exponential security, we mean that no PPT adversary cannot break the security of that primitive with probability better than $2^{-\lambda^c}$ for a constant c.

text, while correctly decrypting all others. We could, in different hybrids, hardcode the correct partial decryption for each of the exponentially many possible ciphertexts, but this would blow up the size of the obfuscated program. Therefore, instead of directly including a pMHE ciphertext in each party's DS public key, we have each party obfuscate an additional program which produces a new pMHE ciphertext each time it is used. This way, when we need to remove all information about a given sample, we can remove the entire corresponding secret key (via the appropriate use of puncturable PRFs and hardcoded values). This technique may be useful for other primitives, such as NIMPC [BGI+14a] and probabilistic iO [CLTV15], to avoid the use of an exponential number of hybrids.

Achieving active security with a random oracle. Upgrading to active security is challenging because we need to protect against two types of attacks: malformed messages, and rushing adversaries, who wait for honest parties' messages before sending their own. We protect against the former using non-interactive zero knowledge proofs. (This requires a URS which, though it is a form of setup, is much weaker than an SRS.) We protect against the latter via a generic transformation that we call an *anti-rusher* (Section 5.1). To use our anti-rusher, each party includes in her public key an obfuscated program which takes as input a hash (i.e. a random oracle output) of all parties' public keys. It then samples *new* (DS) public keys, using this hash as a PRF nonce. This ensures that even an adversary who selects her public keys after seeing the honest party public keys cannot influence the selected sample other than by re-sampling polynomially many times.

Public-key PCFs We start by building a public-key PCF that requires an SRS (Section 6.3). The SRS consists of an obfuscated program that, given a nonce and n parties' public encryption keys, uses a PRF to generate correlated randomness, and encrypts each party's random output to its public key. We can then eliminate the need for a pre-distributed SRS by instead using a distributed sampler to sample it (Section 6.4).

Public-key PCFs without random oracles. The proofs of security for the constructions sketched above only require polynomially many hybrids, roughly speaking because the random oracle allows the simulator to predict and control the inputs to the obfuscated programs. We can avoid the use of the random oracle, at the cost of going through exponentially many hybrids in the proof of security, and thus requiring sub-exponentially secure primitives.

Public-key PCFs for any correlation with a random oracle. Boyle *et al.* [BCG+19b] prove that a public-key PCF in the plain model that can handle *any* correlation (not just reverse-sampleable ones) must have keys at least as large as all the correlated randomness it yields. We observe that we can use a random oracle to sidestep this lower bound by deriving additional randomness from the oracle.

As a stepping stone, we introduce a different flavour of the distributed sampler, which we call the *reusable distributed universal sampler* (reusable DUS).

It is *reusable* because it can be queried multiple times (without the need for additional communication), and it is *universal* because each query can produce a sample from a different distribution (specified by the querier). We build a reusable distributed universal sampler from a universal sampler, a random oracle and a distributed sampler (by using the distributed sampler to produce the universal sampler). Our last public-key PCF (Section 7) then uses the reusable distributed universal sampler to sample from a distribution that first picks the correlated randomness and then encrypts each party's share under her public key.

2 Preliminaries

Notation. We denote the security parameter by λ and the set $\{1, 2, \ldots, m\}$ by $[m]$. Our constructions are designed for an ordered group of n parties $P_1, P_2, \ldots, P_n$. We will denote the set of (indexes of) corrupted parties by C, whereas its complementary, the set of honest players, is H.

We indicate the probability of an event E by $\mathbb{P}[E]$. We use the term *noticeable* to refer to a non-negligible quantity. A probability p is instead *overwhelming* if $1 - p$ is negligible. We say that a cryptographic primitive is sub-exponentially secure, if the advantage of the adversary is bounded by $2^{-\lambda^c}$ for some constant $c > 0$. When the advantage is negligible, we say that it is polynomially secure.

We use the simple arrow $\leftarrow$ to assign the output of a deterministic algorithm $\mathsf{Alg}(x)$ or a specific value a to a variable y, i.e. $y \leftarrow \mathsf{Alg}(x)$ or $y \leftarrow a$. If Alg is instead probabilistic, we write $y \xleftarrow{\$} \mathsf{Alg}(x)$ and we assume that the random tape is sampled uniformly. If the latter is set to a particular value r, we write however $y \leftarrow \mathsf{Alg}(x; r)$. We use $\xleftarrow{\$}$ also if we sample the value of y uniformly over a set X, i.e. $y \xleftarrow{\$} X$. Finally, we refer to algorithms having no input as distributions. The latter are in most cases parametrised by λ. The terms *circuit* and *program* are used interchangeably.

2.1 Indistinguishability Obfuscation

We recall the formal definition of indistinguishability obfuscation (iO) [BGI+01, GGH+13]. Informally speaking an obfuscator is an efficient algorithm that "scrambles" any given circuit Cr until it is impossible to extract any information about Cr except its original input-output behaviour. Furthermore, the result of the operation is another program computing exactly the same function as Cr. We provide a formal definition.

Definition 2.1 (Indistinguishability Obfuscator). *Let $(\mathcal{L}_\lambda)_{\lambda\in\mathbb{N}}$ be a class of circuits such that every $\mathsf{Cr} \in \mathcal{L}_\lambda$ maps a $\mathsf{inp}(\lambda)$-bit input into a $\mathsf{out}(\lambda)$-bit output.*

An indistinguishability obfuscator for $(\mathcal{L}_\lambda)_{\lambda\in\mathbb{N}}$ is a PPT algorithm iO with the following properties.

- ***Correctness.*** *For every $\lambda \in \mathbb{N}$, circuit $\mathsf{Cr} \in \mathcal{L}_\lambda$ and input $x \in \{0,1\}^{\mathsf{inp}(\lambda)}$*

$$\mathbb{P}\left[\, \mathsf{Cr}'(x) = \mathsf{Cr}(x) \mid \mathsf{Cr}' \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathsf{Cr}) \,\right] = 1$$

– ***Security.*** *For every pair of circuits* $\mathsf{Cr}_0, \mathsf{Cr}_1 \in \mathcal{L}_\lambda$ *such that* $\mathsf{Cr}_0(x) = \mathsf{Cr}_1(x)$ *for each* $x \in \{0,1\}^{\mathsf{inp}(\lambda)}$*, the distributions given by* $\mathsf{iO}(1\!\!1^\lambda, \mathsf{Cr}_0)$ *and* $\mathsf{iO}(1\!\!1^\lambda, \mathsf{Cr}_1)$ *are computationally indistinguishable.*

Observe that the obfuscator is tailored to a specific class of circuits. The latter often affects the size of the obfuscated programs, increasing it as the class becomes larger.

The first candidate indistinguishability obfuscator was presented in 2013 by Garg *et al.* [GGH+13]. The construction relies on subexponentially secure primitives. Subsequent work has more and more weakened the requirements under which it is possible to build obfuscation. However, all known constructions still rely on sub-exponentially secure primitives or exponentially-many assumptions. Indistinguishability obfuscators not suffering from these problems are still purpose of research.

2.2 Puncturable PRFs

We recall the formal definition of puncturable PRF [KPTZ13, BW13, BGI14b]. A punturable PRF is a PRF construction F in which the keys K have an additional property: we can puncture them in any position x, obtaining a possibly longer key containing no information about $F_K(x)$ but still allowing computing $F_K(y)$ for every $y \neq x$.

Definition 2.2 (Puncturable PRF). *A puncturable PRF with output space* $(\mathcal{X}_\lambda)_{\lambda\in\mathbb{N}}$ *is a pair of PPT algorithms* (F, Punct) *with the following properties.*

– ***Correctness.*** *For every* $\lambda \in \mathbb{N}$*, key* $K \in \{0,1\}^\lambda$ *and values* $x, y \in \{0,1\}^*$ *with* $x \neq y$

$$\mathbb{P}\big[\, F_K(y) = F_{\hat{K}}(y) \mid \hat{K} \leftarrow \mathsf{Punct}(K, x) \,\big] = 1$$

– ***Security.*** *For every value* $x \in \{0,1\}^\lambda$*, the following two distributions are indistinguishable.*

$$\left\{ (\hat{K}, r) \;\middle|\; \begin{array}{l} K \xleftarrow{\$} \{0,1\}^\lambda \\ \hat{K} \leftarrow \mathsf{Punct}(K, x) \\ r \leftarrow F_K(x) \end{array} \right\} \qquad \left\{ (\hat{K}, r) \;\middle|\; \begin{array}{l} K \xleftarrow{\$} \{0,1\}^\lambda \\ \hat{K} \leftarrow \mathsf{Punct}(K, x) \\ r \xleftarrow{\$} \mathcal{X}_\lambda \end{array} \right\}$$

As noticed in [BW13], it easy to build puncturable PRFs by relying on the GGM construction [GGM86].

2.3 Simulation-Extractable NIZKs

We recall the formal definition of simulation-extractable NIZK [GO07]. Let $\mathcal{R}$ be an efficiently computable relation consisting of pairs (x, w), where x is called statement and w witness. As widely known, a NIZK for $\mathcal{R}$ is a construction that allows a party to prove the knowledge of a witness w for a statement x with

only one message and without revealing any information about w itself (zero-knowledge). The procedure relies on a CRS, which, depending on the construction, can be structured or unstructured. Zero-knowledge is formulated by requiring the existence of two PPT simulators: the first one generates a fake CRS embedding a trapdoor τ into it, the second one leverages the knowledge of τ to produce valid proofs for various statements without needing the corresponding witnesses.

A simulation-extractable NIZK has also an additional property: there exists a PPT algorithm that, in conjunction with the two simulators, is able to extract the witness from any valid proof generated by the adversary. Such algorithm is called extractor and exploits the knowledge of the trapdoor in the CRS. We now formalise the syntax and the security properties of what we have just described.

Definition 2.3 (Non-Interactive Proof). *A non-interactive proof for a relation $\mathcal{R}$ is a triple of PPT algorithms* $(\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$ *with the following syntax.*

- Gen *takes as input the security parameter* $\mathbb{1}^\lambda$ *and outputs a CRS.*
- Prove *takes as input the security parameter* $\mathbb{1}^\lambda$*, the CRS, a statement x and a witness w. The output is a proof π for x.*
- Verify *takes as input a CRS, a proof π and a statement x. The output is a bit indicating whether the proof has been accepted or not.*

Definition 2.4 (Simulation-Extractable NIZK). *A non-interactive proof* $(\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$ *for the relation $\mathcal{R}$ is a simulation extractable NIZK if it satisfies the following properties.*

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

$$\mathbb{P}\left[\mathsf{Verify}(\mathsf{crs}, \pi, x) = 1 \;\middle|\; \begin{array}{l} \mathsf{crs} \xleftarrow{\$} \mathsf{Gen}(\mathbb{1}^\lambda) \\ \pi \xleftarrow{\$} \mathsf{Prove}(\mathbb{1}^\lambda, \mathsf{crs}, x, w) \end{array}\right] = 1$$

- ***Multi-Theorem Zero Knowledge.*** *There exist PPT simulators* Sim_1 *and* Sim_2 *such that, for every set of pairs $\{(x_i, w_i)\}_{i\in[m]}$ in $\mathcal{R}$, no PPT adversary is able to distinguish between the following distributions.*

$$\left\{ \left(\mathsf{crs}, (x_i, \pi_i)_{i\in[m]}\right) \;\middle|\; \begin{array}{l} \mathsf{crs} \xleftarrow{\$} \mathsf{Gen}(\mathbb{1}^\lambda) \\ \forall i \in [m]: \quad \pi_i \xleftarrow{\$} \mathsf{Prove}(\mathbb{1}^\lambda, \mathsf{crs}, x_i, w_i) \end{array} \right\}$$

$$\left\{ \left(\mathsf{crs}, (x_i, \pi_i)_{i\in[m]}\right) \;\middle|\; \begin{array}{l} (\mathsf{crs}, \tau) \xleftarrow{\$} \mathsf{Sim}_1(\mathbb{1}^\lambda) \\ \forall i \in [m]: \quad \pi_i \xleftarrow{\$} \mathsf{Sim}_2(\mathsf{crs}, \tau, x_i) \end{array} \right\}$$

- ***Simulation-Extractability.*** *There exists a PPT extractor* $\mathsf{Extract}$ *such that, for every set of statements $\{x_i\}_{i\in[m]}$ and PPT adversary $\mathcal{A}$, we have*

that

$$\mathbb{P}\left[\begin{array}{c|l} \forall i \in [m] : (\pi', x') \neq (\pi_i, x_i) & (\mathsf{crs}, \tau) \xleftarrow{\$} \mathsf{Sim}_1(1\!\!1^\lambda) \\ \mathsf{Verify}(\mathsf{crs}, \pi', x') = 1 & \forall i \in [m] : \quad \pi_i \xleftarrow{\$} \mathsf{Sim}_2(\mathsf{crs}, \tau, x_i) \\ (x', w') \notin \mathcal{R} & (x', \pi') \xleftarrow{\$} \mathcal{A}\big(1\!\!1^\lambda, \mathsf{crs}, (x_i, \pi_i)_{i\in[m]}\big) \\ & w' \leftarrow \mathsf{Extract}(\mathsf{crs}, \tau, x', \pi') \end{array}\right] = \mathsf{negl}(\lambda)$$

Observe that simulation-extractability and zero-knowledge imply soundness.

NIZKs can be classified based on the type of CRS. When the latter is a uniform string of bits, we talk about NIZK with URS (uniform reference string). When the CRS is instead structured and possibly depends on secrets that must not be revealed to the parties, we talk about NIZKs with SRS (structured reference string).

2.4 MHE with Private Evaluation

We recall the definition of MHE with private evaluation [AJJM20]. This is a construction that permits any party P_i to encrypt a value x_i obtaining a ciphertext c_i and the corresponding partial decryption key sk_i. Once given the ciphertexts of the other parties, P_i can apply a circuit Cr on $(c_j)_{j\in[n]}$, obtaining the i-th partial decryption d_i by means of pk_i. By pooling the partial plaintexts $(d_j)_{j\in[n]}$, it is then possible to obtain $\mathsf{Cr}(x_1, \ldots, x_n)$ without learning any additional information on the inputs. The construction differs from multi-key FHE as there actually exists no public key but only a private one that changes from ciphertext to ciphertext. Furthermore, the encryption algorithm needs to be provided with the parameters (input size, output size and depth) of the circuits we are going to apply on the ciphertexts. The final decryption needs instead to know the exact circuit that was used to produce the partial plaintext it is provided with. It is possible to build MHE schemes with private evaluation from polynomially secure LWE [AJJM20].

Definition 2.5 (Multiparty Homomorphic Encryption with Private Evaluation). *A multiparty homomorphic encryption scheme with private evaluation (pMHE) is a triple of PPT algorithms* $(\mathsf{Enc}, \mathsf{PrivEval}, \mathsf{FinDec})$ *with the following syntax.*

- Enc *is a randomised algorithm that takes as input the security parameter* $1\!\!1^\lambda$*, the parameters of a circuit (input size, output size and depth), an index* i *and a message* x_i*. The output is a ciphertext* c_i *and a partial decryption key* sk_i*.*
- $\mathsf{PrivEval}$ *is a randomised algorithm. It takes as input a partial decryption key* sk_i*, the description of a circuit* Cr *mapping* n *inputs into a single output and* n *ciphertexts* $c_1, c_2, \ldots, c_n$*. The output is a partial plaintext* d_i*.*
- FinDec *is a deterministic algorithm taking as input a circuit* Cr *and the corresponding* n *partial decryptions* $d_1, d_2, \ldots, d_n$*. The output is a plaintext* d*.*

Definition 2.6 (Reusable Semi-Malicious Security). *We say that a pMHE scheme satisfies* reusable semi-malicious security *if it satisfies the following properties.*

- ***Correctness.*** *For every security parameter $\lambda \in \mathbb{N}$, circuit Cr mapping n inputs into one output and inputs $x_1, x_2, \ldots, x_n$, we have*

$$\mathbb{P}\left[d = \mathsf{res} \;\middle|\; \begin{array}{ll} \forall i \in [n]: & (c_i, \mathsf{sk}_i) \xleftarrow{\$} \mathsf{Enc}(1\!\!1^\lambda, \mathsf{Cr.params}, i, x_i) \\ \forall i \in [n]: & d_i \xleftarrow{\$} \mathsf{Eval}(\mathsf{sk}_i, \mathsf{Cr}, c_1, c_2, \ldots, c_n) \\ & d \leftarrow \mathsf{FinDec}(\mathsf{Cr}, d_1, d_2, \ldots, d_n) \\ & \mathsf{res} \leftarrow \mathsf{Cr}(x_1, x_2, \ldots, x_n) \end{array} \right] = 1$$

- ***Reusable Semi-Malicious Security.*** *There exist PPT simulators Sim_1 and Sim_2 such that, for every $n \in \mathbb{N}$, set of corrupted parties C, circuit Cr, inputs of the honest parties $(x_h)_{h \in H}$, no PPT adversary is able to win the game $\mathcal{G}_{pMHE}^{C,\mathsf{Cr},(x_h)_{h\in H}}(\lambda)$ (see Fig. 3) with noticeable advantage.*

$\mathcal{G}_{\mathrm{pMHE}}^{C,\mathsf{Cr},(x_h)_{h\in H}}(\lambda)$

Initialisation.

1. $b \xleftarrow{\$} \{0,1\}$
2. $\forall h \in H: \quad (c_h^0, \mathsf{sk}_h^0) \xleftarrow{\$} \mathsf{Enc}(1\!\!1^\lambda, \mathsf{Cr.params}, h, x_h)$
3. $\big(\tau, (c_h^1)_{h\in H}\big) \xleftarrow{\$} \mathsf{Sim}_1(1\!\!1^\lambda, H, \mathsf{Cr.params})$
4. Activate the adversary with input $\big(1\!\!1^\lambda, (c_h^b)_{h\in H}\big)$.
5. Receive the inputs $(x_i)_{i\in C}$ and the randomness $(r_i)_{i\in C}$ of the corrupted parties from the adversary.
6. $\forall i \in C: \quad (c_i^0, \mathsf{sk}_i^0) \leftarrow \mathsf{Enc}(1\!\!1^\lambda, \mathsf{Cr.params}, i, x_i; r_i)$

Decryption Queries. On input $(\mathsf{Decrypt}, \mathsf{Cr}')$ where $\mathsf{Cr}'.\mathsf{params} = \mathsf{Cr.params}$.

1. $\forall h \in H: \quad d_h^0 \xleftarrow{\$} \mathsf{PrivEval}(\mathsf{sk}_h, \mathsf{Cr}', c_1^0, c_2^0, \ldots, c_n^0)$
2. $\big(\tau, (d_h^1)_{h\in H}\big) \xleftarrow{\$} \mathsf{Sim}_2\big(\tau, \mathsf{Cr}', \mathsf{Cr}'(x_1, x_2, \ldots, x_n), (x_i, r_i)_{i\in C}\big)$
3. Reply with $(d_h^b)_{h\in H}$.

Output. The adversary wins if the final output is b.

Fig. 3. The pMHE Security Game

Observe that correctness states that if we evaluate a circuit Cr over encrypted values $x_1, x_2, \ldots, x_n$ and we pool the partial decryptions, we obtain $C(x_1, x_2, \ldots, x_n)$. Security instead declares that if we publish the encryption of a value x_i and we later on disclose the partial plaintext corresponding to a certain circuit Cr, we reveal nothing more than the output of Cr.

2.5 Somewhere Statistically Binding Hash Functions

We recall the formal definition of somewhere statistically binding hashing (SSB) [HW15], which can be regarded as a more powerful, obfuscation-friendly version

of hash functions. The notion was first introduced by Hubáček and Wichs in [HW15] and can be constructed from FHE [HW15].

The primitive is composed of two algorithms Gen and Hash. The first one is used to generate a key hiding a special index i. The second one instead produces a digest after receiving the key and a message as inputs. The interesting property of SSB hashing is that there exists no pair of messages with different i-th block but same digest. We provide now formal definitions.

Definition 2.7 (Somewhere Statistically Binding Hash Function). *A somewhere statistically binding hash function (SSB) with block alphabet Σ is a pair of PPT algorithms $(\mathsf{Gen}, \mathsf{Hash})$ with the following syntax.*

- Gen *is a randomised procedure taking as input the security parameter $1\!\!1^\lambda$, the maximum number of blocks B and an index $l \in [B]$. The output is a hash key hk.*
- Hash *is a deterministic procedure taking as input a hash key hk and a message $x \in \Sigma^B$ where B is the maximum number of blocks supported by hk. The output is a digest y.*

Definition 2.8 (Security of SSB Hashing). *A somewhere statistically binding hash function is secure if it satisfies the following properties.*

- ***Index Hiding.*** *For every $B \in \mathbb{N}$ and $i, j \in [B]$, no PPT adversary can distinguish between $\mathsf{Gen}(1\!\!1^\lambda, B, i)$ and $\mathsf{Gen}(1\!\!1^\lambda, B, j)$.*
- ***Somewhere Statistically Binding.*** *For every $B \in \mathbb{N}$ and $i \in [B]$,*

$$\mathbb{P}\left[\begin{array}{c} \exists\, x, x' \in \Sigma^B \\ x_i \neq x'_i \\ \mathsf{Hash}(\mathsf{hk}, x) = \mathsf{Hash}(\mathsf{hk}, x') \end{array} \middle| \mathsf{hk} \xleftarrow{\$} \mathsf{Gen}(1\!\!1^\lambda, B, i) \right] = \mathsf{negl}(\lambda).$$

Observe that the SSB property, in conjunction with index hiding, implies the collision resistance of the hash function.

The definition described above is actually weaker than the original one. In [HW15], an SSB hash function featured two additional algorithms Prove and Verify. The first one was used to generate short openings of the j-th preimage block of a digest, for any $j \in [B]$. The second one was used to verify them. These two algorithms are not used in this work.

2.6 Universal Samplers

We recall the definitions of universal sampler (US) [HJK+16], describing the syntax and the various security notions. Informally speaking, a US is a trusted-dealer-based construction that permits to deterministically derive samples from any distribution, learning no information about the corresponding randomness. This primitive is fundamental for the construction of our distributed universal samplers (see Section 7.1 and Section 7.1).

Definition 2.9 (Universal Sampler). *Let $\ell(\lambda)$, $r(\lambda)$ and $t(\lambda)$ be polynomials. A universal sampler for (ℓ, r, t)-distributions is a pair of PPT algorithms* (Setup, Sample) *with the following syntax.*

1. Setup *is a randomised algorithm taking as input the security parameter $1\!\!1^\lambda$ and outputting a sampler U.*
2. Sample *is a deterministic procedure possibly interacting with a random oracle $\mathcal{H}$. It takes as input a sampler U and the description $\mathcal{D}$ of an (ℓ, r, t)-distribution, outputting a sample R.*

Observe that in the above definition, we ask that Setup does not interact with any random oracle. If that was not the case, it would be impossible to generate a universal sampler inside an obfuscated program. The original definition of US [HJK+16] was not as restrictive. However, all the constructions presented in [HJK+16] satisfy the property described above, including those on which we rely in this work.

One-time universal samplers. We now formalise the weaker security notion of the primitive. In particular, we require that the samples look random only for one specific distribution selected ahead of time. In [HJK+16], the authors present a construction satisfying this definition without random oracle.

Definition 2.10 (One-Time Universal Sampler). *A universal sampler* (Setup, Sample) *for (ℓ, r, t)-distributions satisfies one-time security if there exists a PPT simulator* Sim *such that, for every (ℓ, r, t)-distribution $\mathcal{D}$, the following two distributions are computationally indistinguishable*

$$\left\{ U, R \;\middle|\; \begin{array}{l} U \xleftarrow{\$} \mathsf{Setup}(1\!\!1^\lambda) \\ R \leftarrow \mathsf{Sample}(U, \mathcal{D}) \end{array} \right\} \qquad \left\{ U, R \;\middle|\; \begin{array}{l} R \xleftarrow{\$} \mathcal{D} \\ U \xleftarrow{\$} \mathsf{Sim}(1\!\!1^\lambda, \mathcal{D}, R) \end{array} \right\}$$

Notice that security states that one-time USs can be programmed to output ideal samples from the distribution $\mathcal{D}$, without the adversary noticing it.

Adaptively secure universal samplers. We present the stronger security notion of universal samplers. We now ask that the samples look random for every distribution adaptively chosen by the adversary. In particular, we do not care only about one distribution, but multiple ones. This definition can only be satisfied in the random oracle model. Hofheinz *et al.* presented an example of such construction in [HJK+16].

Definition 2.11 (Adaptively Secure Universal Sampler). *A universal sampler* (Setup, Sample) *for (ℓ, r, t)-distributions satisfies adaptive security if there exist PPT simulators* SimGen *and* SimRO *such that no PPT adversary can win the game $\mathcal{G}_{US}(\lambda)$ (see Fig. 4) with noticeable advantage.*

As for the one-time case, security states that the sampler U and the random oracle $\mathcal{H}$ could be programmed to output ideal samples, without the adversary noticing it. The only difference is that now the adversary can adaptively choose the distribution multiple times.

$\mathcal{G}_{\mathrm{US}}(\lambda)$

Initialisation. The challenger samples a random bit $b \xleftarrow{\$} \{0,1\}$ and computes $U_0 \xleftarrow{\$} \mathsf{Setup}(\mathbb{1}^\lambda)$. Then, it instantiates a random oracle $\mathcal{H}$ and a sampling oracle $\mathcal{O}$. Upon receiving an (ℓ, r, t)-distribution $\mathcal{D}$, the latter replies with $\mathcal{D}(F(\mathcal{D}))$ where F is a truly random function outputting $r(\lambda)$ bits. Finally, the challenger computes $(\tau, U_1) \xleftarrow{\$} \mathsf{SimGen}^{\mathcal{O}}(\mathbb{1}^\lambda)$ and provides the adversary with U_b.
Oracle query. Upon receiving any query (RO, x) from the adversary, the challenger computes $r_0 \leftarrow \mathcal{H}(x)$ and $(\tau, r_1) \xleftarrow{\$} \mathsf{SimRO}^{\mathcal{O}}(\mathbb{1}^\lambda, x, \tau)$. It replies with r_b.
Sample. Upon receiving any query $(\mathsf{Sample}, \mathcal{D})$, the challenger computes $R_0 \leftarrow \mathsf{Sample}^{\mathcal{H}}(U_0, \mathcal{D})$ and $R_1 \leftarrow \mathcal{O}(\mathcal{D})$. Then, it replies with R_b.
Output. The adversary wins if its output is equal to b.

Fig. 4. The Universal Sampler Game

3 Defining Distributed Samplers

Informally speaking, a distributed sampler (DS) for the distribution $\mathcal{D}$ is a construction that allows n parties to obtain a random sample R from $\mathcal{D}$ with just one round of communication and without revealing any additional information about the randomness used for the generation of R. The output of the procedure can be derived given only the public transcript, so we do not aim to protect the privacy of the result against passive adversaries eavesdropping the communications between the parties.

If we assume an arbitrary trusted setup, building a DS becomes straightforward; we can consider the trivial setup that directly provides the parties with a random sample from $\mathcal{D}$. Obtaining solutions with a weaker (or no) trusted setup is much more challenging.

The structure and syntax of distributed samplers is formalised as follows. We then analyse different flavours of security definitions.

Definition 3.1 (n-party Distributed Sampler for the Distribution $\mathcal{D}$). *An n-party distributed sampler for the distribution $\mathcal{D}$ is a pair of PPT algorithms* $(\mathsf{Gen}, \mathsf{Sample})$ *with the following syntax:*

1. Gen *is a probabilistic algorithm taking as input the security parameter $\mathbb{1}^\lambda$ and a party index $i \in [n]$ and outputting a sampler share U_i for party i. Let $\{0,1\}^{L(\lambda)}$ be the space from which the randomness of the algorithm is sampled.*
2. Sample *is a deterministic algorithm taking as input n shares of the sampler $U_1, U_2, \ldots, U_n$ and outputting a sample R.*

In some of our security definitions, we will refer to the *one-round protocol* Π_{DS} that is induced by the distributed sampler $\mathsf{DS} = (\mathsf{Gen}, \mathsf{Sample})$. This is the natural protocol obtained from DS, where each party first broadcasts a message output by Gen, and then runs Sample on input all the parties' messages.

3.1 Security

In this section we formalise the definition of distributed samplers with relation to different security flavours, namely, semi-malicious and active. We always assume that we deal with a static adversary who can corrupt up to $n-1$ out of the n parties. We recall that a protocol has semi-malicious security if it remains secure even if the corrupt parties behave semi-honestly, but the adversary can select their random tapes.

Definition 3.2 (Distributed Sampler with Semi-Malicious Security). *A distributed sampler* $(\mathsf{Gen}, \mathsf{Sample})$ *has* semi-malicious security *if there exists a PPT simulator* Sim *such that, for every set of corrupt parties* $C \subsetneq [n]$ *and corresponding randomness* $(\rho_i)_{i\in C}$*, the following two distributions are computationally indistinguishable:*

$$\left\{ \begin{array}{c|ll} (U_i)_{i\in[n]} & \rho_i \xleftarrow{\$} \{0,1\}^{L(\lambda)} & \forall i \in H \\ (\rho_i)_{i\in C}, R & U_i \leftarrow \mathsf{Gen}(1\!\!1^\lambda, i;\, \rho_i) & \forall i \in [n] \\ & R \leftarrow \mathsf{Sample}(U_1, U_2, \ldots, U_n) & \end{array} \right\} \quad \textit{and}$$

$$\left\{ \begin{array}{c|l} (U_i)_{i\in[n]} & R \xleftarrow{\$} \mathcal{D}(1\!\!1^\lambda) \\ (\rho_i)_{i\in C}, R & (U_i)_{i\in H} \xleftarrow{\$} \mathsf{Sim}\big(1\!\!1^\lambda, C, R, (\rho_i)_{i\in C}\big) \end{array} \right\}$$

Observe that this definition implies that, even in the simulation, the relation

$$R = \mathsf{Sample}(U_1, U_2, \ldots, U_n)$$

holds with overwhelming probability. In other words, security requires that $(\mathsf{Gen}, \mathsf{Sample})$ securely implements the functionality that samples from $\mathcal{D}$ and outputs the result to all of the parties.

Observe that the previous definition can be adapted to passive security by simply sampling the randomness of the corrupted parties inside the game in the real world and generating it using the simulator in the ideal world.

We now define actively secure distributed samplers. Here, to handle the challenges introduced by a rushing adversary, we model security by defining an ideal functionality in the universal composability (UC) framework [Can01], and require that the protocol Π_{DS} securely implements this functionality.

Definition 3.3 (Distributed Sampler with Active Security). *Let* $\mathsf{DS} = (\mathsf{Gen}, \mathsf{Sample})$ *be a distributed sampler for the distribution* $\mathcal{D}$*. We say that* DS *has* active security *if the one-round protocol* Π_{DS} *securely implements the functionality* $\mathcal{F}_{\mathcal{D}}$ *(see Fig. 5) against a static and active adversary corrupting up to* $n-1$ *parties.*

Remark 3.4 (Distributed Samplers with a CRS or Random Oracle). Our constructions with active security rely on a setup assumption in the form of a common reference string (CRS) and random oracle. For a CRS, we assume the algorithms $\mathsf{Gen}, \mathsf{Sample}$ are implicitly given the CRS as input, which is modelled

$\mathcal{F}_{\mathcal{D}}$

Initialisation. On input Init from every honest party and the adversary, the functionality activates and sets $Q := \emptyset$. (Q will be used to keep track of queries.) If all the parties are honest, the functionality outputs $R \xleftarrow{\$} \mathcal{D}(\mathbb{1}^\lambda)$ to every honest party and sends R to the adversary, then it halts.

Query. On input Query from the adversary, the functionality samples $R \xleftarrow{\$} \mathcal{D}(\mathbb{1}^\lambda)$ and creates a fresh label id. It sends (id, R) to the adversary and adds the pair to Q.

Output. On input $(\mathsf{Output}, \widehat{\mathsf{id}})$ from the adversary, the functionality retrieves the only pair $(\mathsf{id}, R) \in Q$ with $\mathsf{id} = \widehat{\mathsf{id}}$. If such pair does not exist, the functionality does nothing. Otherwise, it outputs R to every honest party and terminates.

Abort. On input Abort from the adversary, the functionality outputs $\perp$ to every honest party and terminates.

Fig. 5. Distributed Sampler Functionality

as being sampled by an ideal setup functionality. As usual, the random oracle is modelled as an external oracle that may be queried by any algorithm or party, and programmed by the simulator in the security proof.

Observe that this definition allows the adversary to request several samples R from the functionality, and then select the one it likes the most. Our definition must allow this in order to deal with a rushing adversary who might wait for the messages $(U_i)_{i\in H}$ of all the honest parties and then locally re-generate the corrupt parties' messages $(U_i)_{i\in C}$, obtaining a wide range of possible outputs. Finally, it can broadcast the corrupt parties' messages that lead to the output it likes the most. This makes distributed samplers with active security rather useless when the distribution $\mathcal{D}$ has low entropy, i.e. when there exists a polynomial-size set S such that $\mathcal{D}(\mathbb{1}^\lambda) \in S$ with overwhelming probability. Indeed, in such cases, the adversary is able to select its favourite element in the image of $\mathcal{D}$.

On the usefulness of distributed samplers with a CRS. Our distributed samplers with active security require a CRS for NIZK proofs. Since one of the main goals of the construction is avoid trusted setup in multiparty protocols, assuming the existence of a CRS, which itself is some form of setup, may seem wrong.

We highlight, however, that some types of CRS are much easier to generate than others. A CRS that depends on values which must remain secret (e.g. an RSA modulus with unknown factorization, or an obfuscated program which contains a secret key) is difficult to generate. However, assuming the security of trapdoor permutations [FLS90], bilinear maps [GOS06], learning with errors [PS19] or indistinguishability obfuscation [BP15], we can construct NIZK proofs where the CRS is just a random string of bits, i.e. a URS. In the random oracle model, such a CRS can even be generated without any interaction. So, the CRS required by our constructions is the simplest, weakest kind of CRS setup.

4 A Construction with Semi-Malicious Security

We now present the main construction of this paper: a distributed sampler with semi-malicious security based on polynomially secure MHE with private evaluation and indistinguishability obfuscation. In Section 5, we explain how to upgrade this construction to achieve active security.

The basic idea. Our goal is to generate a random sample R from the distribution $\mathcal{D}$. The natural way to do it is to produce a random bit string s and feed it into $\mathcal{D}$. We want to perform the operation in an encrypted way as we need to preserve the privacy of s. A DS implements the functionality that provides samples from the underlying distribution, but not the randomness used to obtain them, so no information about s can be leaked.

We guarantee that any adversary corrupting up to $n-1$ parties is not able to influence the choice of s by XORing n bit strings of the same length, the i-th one of which is independently sampled by the i-th party P_i. Observe that we are dealing with a semi-malicious adversary, so we do not need to worry about corrupted parties adaptively choosing their shares after seeing those of the honest players.

Preserving the privacy of the random string. To preserve the privacy of s, we rely on an MHE scheme with private evaluation $\mathsf{pMHE} = (\mathsf{Enc}, \mathsf{PrivEval}, \mathsf{FinDec})$. Each party P_i encrypts s_i, publishing the corresponding ciphertext c_i and keeping the private key sk_i secret. As long as the honest players do not reveal their partial decryption keys, the privacy of the random string s is preserved. Using the homomorphic properties of the MHE scheme, the parties are also able to obtain partial plaintexts of R without any interactions. However, we run into an issue: in order to finalise the decryption, the construction would require an additional round of communication where the partial plaintexts are broadcast.

Reverting to a one-round construction. We need to find a way to perform the final decryption without additional interaction, while at the same time preserving the privacy of the random string s. That means revealing a very limited amount of information about the private keys $\mathsf{sk}_1, \mathsf{sk}_2, \ldots, \mathsf{sk}_n$, so that it is only possible to retrieve R, revealing nothing more.

Inspired by [HIJ+17], we build such a precise tool by relying on indistinguishability obfuscation: in the only round of interaction, each party P_i additionally publishes an obfuscated *evaluation program* EvProg_i containing the private key sk_i. When given the ciphertexts of the other parties, EvProg_i evaluates the circuit producing the final result R and outputs the partial decryption with relation to sk_i. Using the evaluation programs, the players are thus able to retrieve R by feeding the partial plaintexts into $\mathsf{pMHE.FinDec}$.

Dealing with the leakage about the secret keys. At first glance, the solution outlined in the previous paragraph seems to be secure. However, there are some sneaky issues we need to deal with.

In this warm-up construction, we aim to protect the privacy of the random string s by means of the reusable semi-malicious security of the MHE scheme with private evaluation. To rely on this assumption, no information on the secret keys must be leaked. However, this is not the case here, as the private keys are part of the evaluation programs.

In the security proof, we are therefore forced to proceed in two steps: first, we must remove the secret keys from the programs using obfuscation, and then we can apply reusable semi-malicious security. The first task is actually trickier than it may seem. iO states we cannot distinguish between the obfuscation of two equivalent programs. Finding a program with the same input-output behaviour as EvProg_i without it containing any information about sk_i is actually impossible, as any output of the program depends on the private key. We cannot even hard-code the partial decryptions under sk_i for all possible inputs into the obfuscated program as that would require storing an exponential amount of information, blowing up the size of EvProg_i.

In [HIJ+17], while constructing an NI-MPC protocol based on multi-key FHE and iO, the authors deal with an analogous issue by progressively changing the behaviour of the program input by input, first hard-coding the output corresponding to a specific input and then using the simulatability of partial decryptions to remove any dependency on the multi-key FHE secret key. Unfortunately, in our context, this approach raises additional problems. First of all, in contrast with some multi-key FHE definitions, MHE does not support simulatability of partial decryptions. Additionally, since the procedure of [HIJ+17] is applied input by input, the security proof would require exponentially many hybrids. In that case, security can be argued only if transitions between subsequent hybrids cause a subexponentially small increase in the adversary's advantage. In other words, we would need to rely on subexponentially secure primitives even if future research shows that iO does not. Finally, we would still allow the adversary to compute several outputs without changing the random strings $(s_h)_{h\in H}$ selected by the honest parties. Each of the obtained values leaks some additional information about the final output of the distributed sampler. In [HIJ+17], this fact did not constitute an issue as this type of leakage is intrinsically connected to the notion of NI-MPC.

Bounding the leakage: key generation programs. To avoid the problems described above, we introduce the idea of *key generation programs*. Each party P_i publishes an obfuscated program KGProg_i which encrypts a freshly chosen string s_i, keeping the corresponding partial decryption key secret.

The randomness used by KGProg_i is produced via a puncturable PRF F taking as a nonce the key generation programs of the other parties. In this way, any slight change in the programs of the other parties leads to a completely unrelated string s_i, ciphertext c_i and key sk_i. It is therefore possible to protect the privacy of s_i using a polynomial number of hybrids, as we need only worry about a single combination of inputs. Specifically, we can remove any information about sk_i from EvProg_i and hard-code the partial plaintext d_i corresponding to $(c_j)_{j\in[n]}$. At that point, we can rely on the reusable semi-malicious security of the

MHE scheme with private evaluation, removing any information about s_i from c_i and d_i and programming the final output to be a random sample R from $\mathcal{D}$.

The introduction of the key generation programs requires minimal modifications to the evaluation programs. In order to retrieve the MHE private key, EvProg_i needs to know the same PRF key K_i used by KGProg_i. Moreover, it now takes as input the key generation programs of the other parties, from which it will derive the MHE ciphertexts needed for the computation of R. Observe that EvProg_i will also contain KGProg_i, which will be fed into the other key generation programs in a nested execution of obfuscated circuits.

Compressing the inputs. The only problem with the construction above, is that we now have a circularity issue: we cannot actually feed one key generation program as input to another key generation program, since the programs are of the same size. This holds even if we relied on obfuscation for Turing machines, since to prove security, we would need to puncture the PRF keys in the nonces, i.e. the key generation programs of the other parties. The point at which the i-th key is punctured, which is at least as big as the program itself, must be hard-coded into KGProg_i, which is clearly too small.

Instead of feeding entire key generation programs into KGProg_i, we can input their hash, which is much smaller. This of course means that there now exist different combinations of key generation programs leading to the same MHE ciphertext-key pair (c_i, sk_i), and the adversary could try to extract information about sk_i by looking for collisions. The security of the hash function should, however, prevent this attack. The only issue is that iO does not really get along with this kind of argument based on collision-resistant hashing. We instead rely on the more iO-friendly notion of a *somewhere statistically binding* hash function $\mathsf{SSB} = (\mathsf{Gen}, \mathsf{Hash})$ [HW15].

Final construction. We now present the formal description of our semi-maliciously secure DS. The algorithms Gen and Sample, as well as the unobfuscated key generation program $\mathcal{P}_{\mathsf{KG}}$ and evaluation program $\mathcal{P}_{\mathsf{Eval}}$, can be found in Fig. 6. In the description, we assume that the puncturable PRF F outputs pseudorandom strings (r_1, r_2, r_3) where each of r_1, r_2 and r_3 is as long as the randomness needed by $\mathcal{D}$, $\mathsf{pMHE.Enc}$, and $\mathsf{HE.PrivEval}$ respectively. Moreover, we denote by B the maximum number of blocks in the messages fed into $\mathsf{SSB.Hash}$.

Theorem 4.1. *If* $\mathsf{SSB} = (\mathsf{Gen}, \mathsf{Hash})$ *is a somewhere statistically binding hash function,* $\mathsf{pMHE} = (\mathsf{Enc}, \mathsf{PrivEval}, \mathsf{FinDec})$ *is a MHE scheme with private evaluation,* iO *is an indistinguishability obfuscator and* (F, Punct) *is a puncturable PRF, the construction in Fig. 6 is an n-party distributed sampler with semi-malicious security for the distribution $\mathcal{D}$.*

Proof. We prove the security of the construction in Fig. 6 in a sequence of hybrids. In the initial hybrid (hybrid 0), we start with the real game; in the last hybrid (hybrid 6), we produce a simulated sampler share on behalf of one of the honest parties, which leads the parties to output an R sampled at random from

Distributed Sampler with Semi-Malicious Security

$\mathsf{Gen}(1\!\!1^\lambda, i)$:

1. $K \xleftarrow{\$} \{0,1\}^\lambda$
2. $\mathsf{hk} \xleftarrow{\$} \mathsf{SSB.Gen}(1\!\!1^\lambda, B, 0)$
3. $\mathsf{KGProg} \xleftarrow{\$} \mathsf{iO}\big(1\!\!1^\lambda, \mathcal{P}_{\mathsf{KG}}[K, i]\big)$
4. $\mathsf{EvProg} \xleftarrow{\$} \mathsf{iO}\big(1\!\!1^\lambda, \mathcal{P}_{\mathsf{Eval}}[K, i, \mathsf{hk}, \mathsf{KGProg}]\big)$
5. Output $U := (\mathsf{hk}, \mathsf{KGProg}, \mathsf{EvProg})$.

$\mathsf{Sample}\big(\big(U_i = (\mathsf{hk}_i, \mathsf{KGProg}_i, \mathsf{EvProg}_i)\big)_{i\in[n]}\big)$:

1. $\forall i \in [n]: \quad d_i \leftarrow \mathsf{EvProg}_i\big((\mathsf{hk}_j, \mathsf{KGProg}_j)_{j\neq i}\big)$
2. Output $R \leftarrow \mathsf{pMHE.FinDec}\big(\tilde{\mathcal{D}}, (d_i)_{i\in[n]}\big)$

The algorithm $\tilde{\mathcal{D}}$.
Given a set of n random strings $s_1, s_2, \ldots, s_n$, perform the following operations.

1. $s \leftarrow s_1 \oplus s_2 \oplus \cdots \oplus s_n$
2. Output $R \leftarrow \mathcal{D}(1\!\!1^\lambda; s)$

$\mathcal{P}_{\mathsf{KG}}[K, i]$: the key generation program

Hard-coded. The private key K and the index i of the party.
Input. A hash y.

1. $(r_1, r_2, r_3) \leftarrow F_K(y)$
2. $s \leftarrow r_1$
3. $(c, \mathsf{sk}) \leftarrow \mathsf{pMHE.Enc}(1\!\!1^\lambda, \tilde{\mathcal{D}}.\mathsf{params}, i, s;\, r_2)$
4. Output c.

$\mathcal{P}_{\mathsf{Eval}}[K, i, \mathsf{hk}_i, \mathsf{KGProg}_i]$: the evaluation program

Hard-coded. The private key K, the index i of the party, the hash key hk_i, and the obfuscated key generation program KGProg_i.
Input. A set of $n-1$ pairs $(\mathsf{hk}_j, \mathsf{KGProg}_j)_{j\neq i}$ where the first element is a hash key and the second is an obfuscated key generation program.

1. $\forall j \in [n]: \quad y_j \leftarrow \mathsf{SSB.Hash}\big(\mathsf{hk}_j, (\mathsf{hk}_l, \mathsf{KGProg}_l)_{l\neq j}\big)$
2. $\forall j \neq i: \quad c_j \leftarrow \mathsf{KGProg}_j(y_j)$
3. $(r_1, r_2, r_3) \leftarrow F_K(y_i)$
4. $s_i \leftarrow r_1$
5. $(c_i, \mathsf{sk}_i) \leftarrow \mathsf{pMHE.Enc}(1\!\!1^\lambda, \tilde{\mathcal{D}}.\mathsf{params}, i, s_i;\, r_2)$
6. $d_i \leftarrow \mathsf{pMHE.PrivEval}(\mathsf{sk}_i, \tilde{\mathcal{D}}, c_1, c_2, \ldots, c_n;\, r_3)$
7. Output d_i.

Fig. 6. A Distributed Sampler with Semi-Malicious Security

$\mathcal{D}$. We choose one honest party h, and throughout hybrids $1-6$, we modify only how the sampler share U_h is produced. The rest of the honest parties continue to produce their sampler shares as per the Gen protocol in Fig. 6.

Because the simulator has access to all parties' random tapes, it of course knows all their secrets. In the following, we refer to $U_i = (\mathsf{hk}_i, \mathsf{KGProg}_i, \mathsf{EvProg}_i)$ as the sampler share produced by party i for $i \neq h$. We also refer to the secret keys K_i contained in those programs (also known to the simulator). For whatever values $U_h = (\mathsf{hk}_h, \mathsf{KGProg}_h, \mathsf{EvProg}_h)$ the simulator produces on behalf of party h in a given hybrid, we let $\hat{s}_i$ denote the share of the randomness generated by KGProg_i (on the appropriate nonce y), and $(\hat{c}_i, \hat{\mathsf{sk}}_i)$ denote the encryption of that randomness and the corresponding partial decryption key produced by KGProg_i. Finally, we let $\hat{r}_2^i$ denote the random string input in $\mathsf{pMHE.Enc}$ by KGProg_i for the generation of $\hat{c}_i$.

Hybrid 0: This is the initial hybrid, where the simulator, on behalf of every honest party i, generates a sampler share U_i as per the Gen algorithm in Fig. 6.

Hybrid 1: In this hybrid, the simulator, on behalf of honest party h, punctures the key K_h at the relevant point (the hash of $(\mathsf{hk}_j, \mathsf{KGProg}_j)_{j\neq h}$ produced by the other parties j), but programs the appropriate output at that point into the programs. By the correctness of F puncturing, the input-output behaviour of both programs is the same as it was in the previous hybrid. Therefore, by the security of iO, this hybrid is indistinguishable from the previous one.

More specifically, let $(\mathsf{hk}_j, \mathsf{KGProg}_j)_{j\neq h}$ be the hash key and obfuscated key generation program of every other party. (The simulator knows these — even for corrupt parties — since she gets to see the randomness tape of corrupt parties in the definition of semi-malicious security.) The simulator does the following during Gen on behalf of party h, where the text in red indicates what changed since the previous hybrid:

1. $K_h \xleftarrow{\$} \{0,1\}^\lambda$
2. $\mathsf{hk}_h \xleftarrow{\$} \mathsf{SSB.Gen}(1\!\!1^\lambda, B, 0)$
3. $\hat{y}_h \leftarrow \mathsf{SSB.Hash}\big(\mathsf{hk}_h, (\mathsf{hk}_j, \mathsf{KGProg}_j)_{j\neq h}\big)$
4. $\hat{K}_h \leftarrow \mathsf{Punct}(K_h, \hat{y}_h)$
5. $(\hat{s}_h, \hat{r}_2, \hat{r}_3) \leftarrow F_{K_h}(\hat{y}_h)$
6. $(\hat{c}_h, \hat{\mathsf{sk}}_h) \leftarrow \mathsf{pMHE.Enc}(1\!\!1^\lambda, \tilde{\mathcal{D}}.\mathsf{params}, h, \hat{s}_h; \hat{r}_2)$
7. $\mathsf{KGProg}_h \xleftarrow{\$} \mathsf{iO}\big(1\!\!1^\lambda, \mathcal{P}^1_{\mathsf{KG}}[\hat{K}_h, h, \hat{y}_h, \hat{c}_h]\big)$ (see Fig. 7)
8. $\mathsf{EvProg}_h \xleftarrow{\$} \mathsf{iO}\big(1\!\!1^\lambda, \mathcal{P}^1_{\mathsf{Eval}}[\hat{K}_h, h, \mathsf{hk}_h, \mathsf{KGProg}_h, \hat{y}_h, \hat{c}_h, \hat{\mathsf{sk}}_h, \hat{r}_3]\big)$ (see Fig. 8)
9. Output $U_h := (\mathsf{hk}_h, \mathsf{KGProg}_h, \mathsf{EvProg}_h)$.

Next, for every l from 0 to the length of the input to $\mathsf{SSB.Hash}$, we proceed first to Hybrid $2.l.1$, then to Hybrid $2.l.2$.

Hybrid $2.l.1$: In this hybrid, the simulator, on behalf of honest party h, makes the hash key hk_h statistically binding at index l (whereas before it was statistically binding at index $l-1$). This hybrid is indistinguishable from the previous one by the index hiding property of SSB.

Hybrid $2.l.2$: In this hybrid, the simulator, on behalf of honest party h, changes the evaluation program EvProg_h to only use the hardcoded key and

$\mathcal{P}^1_{\mathsf{KG}}[K, i, \hat{y}, \hat{c}]$

Hard-coded. The private key K, the index i of the party, as well as a nonce $\hat{y}$, and a ciphertext $\hat{c}$.
Input. A nonce y.

1. If $y = \hat{y}$, output $\hat{c}$.
2. Otherwise, continue executing $\mathcal{P}_{\mathsf{KG}}[K]$:
3. $(r_1, r_2, r_3) \leftarrow F_K(y)$
4. $s \leftarrow r_1$
5. $(c, \mathsf{sk}) \leftarrow \mathsf{pMHE.Enc}(1\!\!1^\lambda, \tilde{\mathcal{D}}.\mathsf{params}, i, s; r_2)$
6. Output c.

Fig. 7. The Key Generation Program

$\mathcal{P}^1_{\mathsf{Eval}}[K, i, \mathsf{hk}_i, \mathsf{KGProg}_i, \hat{y}, \hat{c}, \hat{\mathsf{sk}}, \hat{r}]$

Hard-coded. The private key K, the index i of the party, the hash key hk_i, the obfuscated key generation program KGProg_i, as well as a nonce $\hat{y}$, a ciphertext $\hat{c}$, a secret key $\hat{\mathsf{sk}}$, and randomness $\hat{r}$.
Input. A set of $n-1$ pairs $(\mathsf{hk}_j, \mathsf{KGProg}_j)_{j \neq i}$ where the first element is a hash key and the second is an obfuscated key generation program.

1. $\forall j \in [n]: \quad y_j \leftarrow \mathsf{SSB.Hash}\big(\mathsf{hk}_j, (\mathsf{hk}_l, \mathsf{KGProg}_l)_{l \neq j}\big)$
2. $\forall j \neq i: \quad c_j \leftarrow \mathsf{KGProg}_j(y_j)$
3. If $y_i = \hat{y}$, set $\mathsf{sk}_i \leftarrow \hat{\mathsf{sk}}$, $c_i \leftarrow \hat{c}$ and $r_3 \leftarrow \hat{r}$.
4. Otherwise,
   (a) $(r_1, r_2, r_3) \leftarrow F_K(y_i)$
   (b) $s_i \leftarrow r_1$
   (c) $(c_i, \mathsf{sk}_i) \leftarrow \mathsf{pMHE.Enc}(1\!\!1^\lambda, \tilde{\mathcal{D}}.\mathsf{params}, i, s_i; r_2)$
5. $d_i \leftarrow \mathsf{pMHE.PrivEval}(\mathsf{sk}_i, \tilde{\mathcal{D}}, c_1, c_2, \ldots, c_n; r_3)$
6. Output d_i.

Fig. 8. The Evaluation Program

ciphertext if the first l blocks of the input coincide with a hardcoded reference input. (The simulated party h now obfuscates $\mathcal{P}^2_{\mathsf{Eval}}$ (Fig. 9) instead of $\mathcal{P}^1_{\mathsf{Eval}}$ (Fig. 8).) By the fact that $\mathsf{SSB.Hash}$ is statistically binding at l, the input-output behaviour of both programs is the same as it was in the previous hybrid. Therefore, by the security of iO, this hybrid is indistinguishable from the previous one.

More specifically, the simulator does the following during Gen on behalf of party h, where the text in red indicates what changed since the previous hybrid:

1. $K_h \xleftarrow{\$} \{0,1\}^\lambda$
2. $\mathsf{hk}_h \xleftarrow{\$} \mathsf{SSB.Gen}(1\!\!1^\lambda, B, l)$
3. $\hat{y}_h \leftarrow \mathsf{SSB.Hash}\big(\mathsf{hk}_h, (\mathsf{hk}_j, \mathsf{KGProg}_j)_{j\neq h}\big)$
4. $\hat{K}_h \leftarrow \mathsf{Punct}(K_h, \hat{y}_h)$
5. $(\hat{s}_h, \hat{r}_2, \hat{r}_3) \leftarrow F_{K_h}(\hat{y}_h)$
6. $(\hat{c}_h, \hat{\mathsf{sk}}_h) \leftarrow \mathsf{pMHE.Enc}(1\!\!1^\lambda, \tilde{\mathcal{D}}.\mathsf{params}, h, \hat{s}_h; \hat{r}_2)$
7. $\mathsf{KGProg}_h \xleftarrow{\$} \mathsf{iO}\big(1\!\!1^\lambda, \mathcal{P}^1_{\mathsf{KG}}[\hat{K}_h, h, \hat{y}_h, \hat{c}_h]\big)$ (see Fig. 7)
8. $\hat{w} \leftarrow (\mathsf{hk}_j, \mathsf{KGProg}_j)_{j\neq h}$
9. $\mathsf{EvProg}_h \xleftarrow{\$} \mathsf{iO}\big(1\!\!1^\lambda, \mathcal{P}^2_{\mathsf{Eval}}[\hat{K}_h, h, \mathsf{hk}_h, \mathsf{KGProg}_h, \hat{y}_h, \hat{c}_h, \hat{\mathsf{sk}}_h, \hat{r}_3, \hat{w}]\big)$ (see Fig. 9)
10. Output $U_h := (\mathsf{hk}_h, \mathsf{KGProg}_h, \mathsf{EvProg}_h)$.

$\mathcal{P}^2_{\mathsf{Eval}}[K, i, \mathsf{hk}_i, \mathsf{KGProg}_i, \hat{y}, \hat{\mathsf{pk}}, \hat{c}, \hat{\mathsf{sk}}, \hat{r}, \hat{w}]$

Hard-coded. The private key K, the index i of the party, the hash key hk_i, the obfuscated key generation program KGProg_i, as well as a nonce $\hat{y}$, a ciphertext $\hat{c}$, a secret key $\hat{\mathsf{sk}}$, randomness $\hat{r}$, and a hardcoded input $\hat{w}$.
Input. A set of $n-1$ pairs $(\mathsf{hk}_j, \mathsf{KGProg}_j)_{j\neq i}$ where the first element is a hash key and the second is an obfuscated key generation program.

1. $\forall j \in [n]: \quad y_j \leftarrow \mathsf{SSB.Hash}\big(\mathsf{hk}_j, (\mathsf{hk}_l, \mathsf{KGProg}_l)_{l\neq j}\big)$
2. $\forall j \neq i: \quad c_j \leftarrow \mathsf{KGProg}_j(y_j)$
3. If $y_i = \hat{y}$ and the first l blocks of $\hat{w}$ and $(\mathsf{hk}_j, \mathsf{KGProg}_j)_{j\neq h}$ coincide, set $\mathsf{sk}_i \leftarrow \hat{\mathsf{sk}}$, $c_i \leftarrow \hat{c}$ and $r_3 \leftarrow \hat{r}$.
4. Otherwise,
   (a) $(r_1, r_2, r_3) \leftarrow F_K(y_i)$
   (b) $s_i \leftarrow r_1$
   (c) $(c_i, \mathsf{sk}_i) \leftarrow \mathsf{pMHE.Enc}(1\!\!1^\lambda, \tilde{\mathcal{D}}.\mathsf{params}, i, s_i; r_2)$
5. $d_i \leftarrow \mathsf{pMHE.PrivEval}(\mathsf{sk}_i, \tilde{\mathcal{D}}, c_1, c_2, \ldots, c_n; r_3)$
6. Output d_i.

Fig. 9. The Evaluation Program

Hybrid 3: At this point, EvProg_h only uses the hardcoded key and ciphertext if the *entire* input matches the hardcoded reference input. Since that is the case,

EvProg_h will only ever use the hardcoded decryption key on one ciphertext; so, we can remove the hardcoded decryption key entirely, and instead hardcode a partial decryption value. In this hybrid, the simulator, on behalf of party h, replaces $\mathcal{P}^2_{\mathsf{Eval}}$ (Fig. 9) with $\mathcal{P}^3_{\mathsf{Eval}}$ (Fig. 10) which does exactly this. This hybrid is indistinguishable from the previous one by the security of iO.

More specifically, the simulator does the following during Gen on behalf of party h, where the text in red indicates what changed since the previous hybrid:

1. $K_h \xleftarrow{\$} \{0,1\}^\lambda$
2. $\mathsf{hk}_h \xleftarrow{\$} \mathsf{SSB.Gen}(1\!\!1^\lambda, B, B)$
3. $\hat{y}_h \leftarrow \mathsf{SSB.Hash}\big(\mathsf{hk}_h, (\mathsf{hk}_j, \mathsf{KGProg}_j)_{j\neq h}\big)$
4. $\hat{K}_h \leftarrow \mathsf{Punct}(K_h, \hat{y}_h)$
5. $(\hat{s}_h, \hat{r}_2, \hat{r}_3) \leftarrow F_{K_h}(\hat{y}_h)$
6. $(\hat{c}_h, \hat{\mathsf{sk}}_h) \leftarrow \mathsf{pMHE.Enc}(1\!\!1^\lambda, \tilde{\mathcal{D}}.\mathsf{params}, h, \hat{s}_h; \hat{r}_2)$
7. $\mathsf{KGProg}_h \xleftarrow{\$} \mathsf{iO}\big(1\!\!1^\lambda, \mathcal{P}^1_{\mathsf{KG}}[\hat{K}_h, h, \hat{y}_h, \hat{c}_h]\big)$ (see Fig. 7)
8. $\hat{d}_h \leftarrow \mathsf{pMHE.PrivEval}(\hat{\mathsf{sk}}_h, \tilde{\mathcal{D}}, \hat{c}_1, \hat{c}_2, \ldots, \hat{c}_n; \hat{r}_3)$
9. $\hat{w} \leftarrow (\mathsf{hk}_j, \mathsf{KGProg}_j)_{j\neq h}$
10. $\mathsf{EvProg}_h \xleftarrow{\$} \mathsf{iO}\big(1\!\!1^\lambda, \mathcal{P}^3_{\mathsf{Eval}}[\hat{K}_h, h, \mathsf{hk}_h, \mathsf{KGProg}_h, \hat{w}, \hat{d}_h]\big)$ (see Fig. 10)
11. Output $U_h := (\mathsf{hk}_h, \mathsf{KGProg}_h, \mathsf{EvProg}_h)$.

$\mathcal{P}^3_{\mathsf{Eval}}[K, i, \mathsf{hk}_i, \mathsf{KGProg}_i, \hat{w}, \hat{d}]$

Hard-coded. The private key K, the index i of the party, the hash key hk_i, the obfuscated key generation program KGProg_i, as well as a hardcoded input $\hat{w}$ and a partial decryption $\hat{d}$.
Input. A set of $n-1$ pairs $(\mathsf{hk}_j, \mathsf{KGProg}_j)_{j\neq i}$ where the first element is a hash key and the second is an obfuscated key generation program.

1. If $(\mathsf{hk}_j, \mathsf{KGProg}_j)_{j\neq i} = \hat{w}$, output $\hat{d}$.
2. Otherwise, $\forall j \in [n]: \quad y_j \leftarrow \mathsf{SSB.Hash}\big(\mathsf{hk}_j, (\mathsf{hk}_l, \mathsf{KGProg}_l)_{l\neq j}\big)$
3. $\forall j \neq i: \quad c_j \leftarrow \mathsf{KGProg}_j(y_j)$
4. $(r_1, r_2, r_3) \leftarrow F_K(y_i)$
5. $s_i \leftarrow r_1$
6. $(c_i, \mathsf{sk}_i) \leftarrow \mathsf{pMHE.Enc}(1\!\!1^\lambda, \tilde{\mathcal{D}}.\mathsf{params}, i, s_i; r_2)$
7. $d_i \leftarrow \mathsf{pMHE.PrivEval}(\mathsf{sk}_i, \tilde{\mathcal{D}}, c_1, c_2, \ldots, c_n; r_3)$
8. Output d_i.

Fig. 10. The Evaluation Program

Hybrid 4. In this hybrid, the simulator computes the final output R directly as $R \leftarrow \mathcal{D}(1\!\!1^\lambda; s)$ where $s = \hat{s}_1 \oplus \hat{s}_2 \oplus \cdots \oplus \hat{s}_n$. (The simulator of course has all these values, as it has access to all parties' random tapes.) This hybrid is indistinguishable from the previous one by the correctness of obfuscation and MHE with private evaluation.

Hybrid 5. In this hybrid, the simulator, on behalf of party h, replaces the hardcoded output of F at $\hat{y}_h$ with a truly random one. This hybrid is indistinguishable from the previous one by the security of F.

More specifically, the simulator does the following during Gen on behalf of party h, where the text in red indicates what changed since the previous hybrid:

1. $K_h \xleftarrow{\$} \{0,1\}^\lambda$
2. $\mathsf{hk}_h \xleftarrow{\$} \mathsf{SSB.Gen}(1\!\!1^\lambda, B, B)$
3. $\hat{y}_h \leftarrow \mathsf{SSB.Hash}\big(\mathsf{hk}_h, (\mathsf{hk}_j, \mathsf{KGProg}_j)_{j\neq h}\big)$
4. $\hat{K}_h \leftarrow \mathsf{Punct}(K_h, \hat{y}_h)$
5. Sample $(\hat{s}_h, \hat{r}_2, \hat{r}_3)$ at random from the appropriate space
6. $(\hat{c}_h, \hat{\mathsf{sk}}_h) \leftarrow \mathsf{pMHE.Enc}(1\!\!1^\lambda, \tilde{\mathcal{D}}.\mathsf{params}, h, \hat{s}_h; \hat{r}_2)$
7. $\mathsf{KGProg}_h \xleftarrow{\$} \mathsf{iO}\big(1\!\!1^\lambda, \mathcal{P}^1_{\mathsf{KG}}[\hat{K}_h, h, \hat{y}_h, \hat{c}_h]\big)$ (see Fig. 7)
8. $\hat{d}_h \leftarrow \mathsf{pMHE.PrivEval}(\hat{\mathsf{sk}}_h, \tilde{\mathcal{D}}, \hat{c}_1, \hat{c}_2, \ldots, \hat{c}_n; \hat{r}_3)$
9. $\hat{w} \leftarrow (\mathsf{hk}_j, \mathsf{KGProg}_j)_{j\neq h}$
10. $\mathsf{EvProg}_h \xleftarrow{\$} \mathsf{iO}\big(1\!\!1^\lambda, \mathcal{P}^3_{\mathsf{Eval}}[\hat{K}_h, h, \mathsf{hk}_h, \mathsf{KGProg}_h, \hat{w}, \hat{d}_h]\big)$ (see Fig. 10)
11. Output $U_h := (\mathsf{hk}_h, \mathsf{KGProg}_h, \mathsf{EvProg}_h)$.

Hybrid 6. In this hybrid, the simulator replaces the real ciphertext $\hat{c}_h$ and partial decryption $\hat{d}_h$ with simulated ones. The production of this simulated values does not require party h's secret decryption key nor the plaintext $\hat{s}_h$; it forces the final decryption to output the value R. Since the view of the adversary contains now no information about $\hat{s}_h$, the simulator can sample R at random from $\mathcal{D}$. This hybrid is indistinguishable from the previous one by the reusable semi-malicious security of the MHE scheme with private evaluation.

More specifically, the simulator does the following during Gen on behalf of party h, where the text in red indicates what changed since the previous hybrid:

1. $K_h \xleftarrow{\$} \{0,1\}^\lambda$
2. $\mathsf{hk}_h \xleftarrow{\$} \mathsf{SSB.Gen}(1\!\!1^\lambda, B, B)$
3. $\hat{y}_h \leftarrow \mathsf{SSB.Hash}\big(\mathsf{hk}_h, (\mathsf{hk}_j, \mathsf{KGProg}_j)_{j\neq h}\big)$
4. $\hat{K}_h \leftarrow \mathsf{Punct}(K_h, \hat{y}_h)$
5. $(\tau, \hat{c}_h) \xleftarrow{\$} \mathsf{pMHE.Sim}_1(1\!\!1^\lambda, \{h\}, \tilde{\mathcal{D}}.\mathsf{params})$
6. $\mathsf{KGProg}_h \xleftarrow{\$} \mathsf{iO}\big(1\!\!1^\lambda, \mathcal{P}^1_{\mathsf{KG}}[\hat{K}_h, h, \hat{y}_h, \hat{c}_h]\big)$ (see Fig. 7)
7. $R \xleftarrow{\$} \mathcal{D}(1\!\!1^\lambda)$
8. Retrieve the values $(\hat{s}_j)_{j\neq h}$ and the randomness $(\hat{r}^j_2)_{j\neq h}$ used by the other parties for the generation of $(\hat{c}_j)_{j\neq h}$
9. $(\tau, \hat{d}_h) \xleftarrow{\$} \mathsf{pMHE.Sim}_2\big(\tau, \tilde{\mathcal{D}}, R, (\hat{s}_j, \hat{r}^j_2)_{j\neq h}\big)$
10. $\hat{w} \leftarrow (\mathsf{hk}_j, \mathsf{KGProg}_j)_{j\neq h}$
11. $\mathsf{EvProg}_h \xleftarrow{\$} \mathsf{iO}\big(1\!\!1^\lambda, \mathcal{P}^3_{\mathsf{Eval}}[\hat{K}_h, h, \mathsf{hk}_h, \mathsf{KGProg}_h, \hat{w}, \hat{d}_h]\big)$ (see Fig. 10)
12. Output $U_h := (\mathsf{hk}_h, \mathsf{KGProg}_h, \mathsf{EvProg}_h)$.

Since this hybrid gives us exactly the distribution we want our original Hybrid 0 to be indistinguishable from, this completes the proof.

□

Observe that a distributed sampler with semi-malicious security also has passive security.

5 Upgrading to Active Security

When moving from semi-malicious to active security, there are two main issues we need to tackle: corrupt parties publishing malformed shares of the sampler, and rushing adversaries. The former can be easily dealt with by adding NIZK proofs of well-formedness to the sampler shares (for this reason, our solution relies on a URS). Rushing adversaries are a more challenging problem, and to deal with this, we rely on a random oracle.

The problem of rushing. In the semi-maliciously secure construction described in Section 4, the randomness used to generate an honest party's MHE ciphertexts and private keys is output by a PRF, which takes as input a nonce that depends on the key generation programs of all parties (including the corrupt ones). To prove security, we need to puncture the PRF key at that nonce, erasing any correlation between the MHE ciphertext and the PRF key. This can be done in the semi-malicious case, as the simulator knows the programs of the corrupted parties before it must produce those of the honest parties. In the actively secure case, we run into an issue. The adversary is able to adaptively choose the programs of the corrupted parties after seeing those of the other players, in what is called *rushing behaviour*. In the security proof, we would therefore need to puncture a PRF key without knowing the actual position where puncturing is needed.

Although the issue we described above is very specific, dealing with rushing behaviour is a general problem. In a secure distributed sampler, we can program the shares of the honest parties to output an ideal sample when used in conjunction with the shares of the corrupted players. Since the latter are unknown upon generation of the honest players' shares, the immediate approach would be to program the outputs for every possible choice of the adversary. We run however into an incompressibility problem as we would need to store exponentially many ideal outputs in the polynomial-sized sampler shares.

5.1 Defeating Rushing

In this section, we present a compiler that allows us to deal with rushing behaviour without adding any additional rounds of interaction. This tool handles rushing behaviour not only for distributed samplers, but for a wide range of applications (including our public-key PCF in Section 6). Consider any single-round protocol with no private inputs, where $\mathsf{SendMsg}$ is the algorithm which party i runs to choose a message to send, and Output is an algorithm that determines each party's output (from party i's state and all the messages sent). More concretely, we can describe any such one-round protocol using the following syntax:

$\mathsf{SendMsg}(\mathbb{1}^\lambda, i; r_i) \to \mathsf{g}_i$ generates party i's message g_i, and
$\mathsf{Output}(i, r_i, (\mathsf{g}_j)_{j\in[n]}) \to \mathsf{res}_i$ produces party i's output res_i.

(In the case of distributed samplers, $\mathsf{SendMsg}$ corresponds to Gen, and Output corresponds to Sample.)

We define modified algorithms $(\mathsf{ARMsg}, \mathsf{AROutput})$ such that the associated one-round protocol realizes an ideal functionality that first waits for the corrupted parties' randomness, and then generates the randomness and messages of the honest parties.

This functionality clearly denies the adversary the full power of rushing: the ability to choose corrupt parties' messages based on honest parties' messages. For this reason, we call it the *no-rush* functionality $\mathcal{F}_{\mathsf{NoRush}}$. However, we do allow the adversary a weaker form of rushing behaviour: *selective sampling*. The functionality allows the adversary to re-submit corrupt parties' messages as many times as it wants, and gives the adversary the honest parties' messages in response (while hiding the honest parties' randomness). At the end, the adversary can select which execution she likes the most.

Definition 5.1 (Anti-Rusher). *Let* $(\mathsf{SendMsg}, \mathsf{Output})$ *be a one-round* n*-party protocol where* $\mathsf{SendMsg}$ *needs* $L(\lambda)$ *bits of randomness to generate a message. An anti-rusher for* $\mathsf{SendMsg}$ *is a one-round protocol* $(\mathsf{ARMsg}, \mathsf{AROutput})$ *implementing the functionality* $\mathcal{F}_{\mathsf{NoRush}}$ *(see Fig. 11) for* $\mathsf{SendMsg}$ *against an active adversary.*

$\mathcal{F}_{\mathsf{NoRush}}$

Initialisation. Upon receiving Init from every party and the adversary, the functionality activates and enters the querying phase.

Querying phase. Upon receiving the id-th Query from the adversary, the functionality waits for r_i from every corrupted party P_i. Then, for every $h \in H$, it samples $r_h \xleftarrow{\$} \{0,1\}^{L(\lambda)}$ and computes $\mathsf{g}_i \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, i; r_i)$ for every $i \in [n]$. Finally, it stores $(r_i, \mathsf{g}_i)_{i\in[n]}$ along with the identity id and sends $(\mathsf{g}_h)_{h\in H}$ back to the adversary.

Output. Upon receiving Output from the adversary, the functionality waits for a value $\widehat{\mathsf{id}}$ from the adversary, and retrieves the corresponding tuple $(r_i, \mathsf{g}_i)_{i\in[n]}$ (or outputs $\perp$ if there is no such tuple). It then outputs r_h and $(\mathsf{g}_i)_{i\in[n]}$ to P_h for every $h \in H$.

Fig. 11. The Anti-Rushing Functionality $\mathcal{F}_{\mathsf{NoRush}}$

If $(\mathsf{SendMsg}, \mathsf{Output}) = (\mathsf{Gen}, \mathsf{Sample})$ is a distributed sampler with semi-malicious security, applying this transformation gives a distributed sampler with active security.

Intuition Behind our Anti-Rushing Compiler. We define $(\mathsf{ARMsg}, \mathsf{AROutput})$ as follows. When called by party i, ARMsg outputs an obfuscated program S_i; this program takes as input a response of the random oracle, and uses it as a nonce for a PRF F_{K_i}. The program then feeds the resulting pseudorandom string r into $\mathsf{SendMsg}$, and outputs whatever message $\mathsf{SendMsg}$ generates.

Our techniques are inspired by the *delayed backdoor programming* technique of Hofheinz *et al.* [HJK+16], used for adaptively secure universal samplers.

The trapdoor. In order to prove that our compiler realizes $\mathcal{F}_{\mathsf{NoRush}}$ for $\mathsf{SendMsg}$, a simulator must be able to force the compiled protocol to return given outputs of $\mathsf{SendMsg}$, even *after* sending messages (outputs of ARMsg) on behalf of the honest parties.

Behind its usual innocent behaviour, the program S_i hides a trapdoor that allows it to secretly communicate with the random oracle. S_i owns a key k_i for a special authenticated encryption scheme based on puncturable PRFs. Every time it receives a random oracle response as input, S_i parses it as a ciphertext-nonce pair and tries to decrypt it. If decryption succeeds, S_i outputs the corresponding plaintext; otherwise, it resumes the usual innocent behaviour, and runs $\mathsf{SendMsg}$. (The encryption scheme guarantees that the decryption of random strings fails with overwhelming probability; this trapdoor is never used accidentally, but it will play a crucial role in the proof.) Obfuscation conceals how the result has been computed as long as it is indistinguishable from a random $\mathsf{SendMsg}$ output.

The inputs fed into $(\mathsf{S}_i)_{i\in[n]}$ are generated by querying the random oracle with the programs themselves and NIZKs proving their well-formedness. The random oracle response consists of a random nonce v and additional n blocks $(u_i)_{i\in[n]}$, the i-th one of which is addressed to S_i. The input to S_i will be the pair (u_i, v). When the oracle tries to secretly communicate a message to S_i, u_i will be a ciphertext, whereas v will be the corresponding nonce.

Given a random oracle query, using the simulation-extractability of the NIZKs, the simulator can retrieve the secrets (in particular, the PRF keys) of the corrupted parties. It can then use this information to learn the randomness used to generate the corrupted parties' messages (i.e. their outputs of $\mathsf{SendMsg}$). The simulator then needs only to encrypt these messages received from $\mathcal{F}_{\mathsf{NoRush}}$ using $(k_i)_{i\in H}$, and include these ciphertexts in the oracle response.

Formal Description of our Anti-Rushing Compiler. We now formalise the ideas we presented in the previous paragraphs. Our anti-rushing compiler is described in Fig. 13. The unobfuscated program $\mathcal{P}_{\mathsf{AR}}$ is available in Fig. 12. We assume that its obfuscation needs $M(\lambda)$ bits of randomness. Observe that $\mathcal{P}_{\mathsf{AR}}$ is based on two puncturable PRFs F and F', the first one of which is used to generate the randomness fed into $\mathsf{SendMsg}$.

The second puncturable PRF is part of the authenticated encryption scheme used in the trapdoor. We assume that its outputs are naturally split into $2m$ λ-bit blocks, where $m(\lambda)$ is the size of an output of $\mathsf{SendMsg}$ (after padding). To encrypt a plaintext $(x^1, \ldots, x^m) \in \{0,1\}^m$ using the key k and nonce $v \in \{0,1\}^\lambda$, we first expand v using F'_k. The ciphertext consists of m λ-bit blocks, the j-th one of which coincides with the $(2j + x^j)$-th block output by F'. Decryption is done by reversing these operations. For this reason, we assume that the values $(u_i)_{i\in[n]}$ in the oracle responses are naturally split into m λ-bit chunks. Observe that if the j-th block of the ciphertext is different from both the $2j$-th and the $(2j + 1)$-th block output by the PRF, decryption fails.

Finally, let $\mathsf{NIZK} = (\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Sim}_1, \mathsf{Sim}_2, \mathsf{Extract})$ be a simulation-extractable NIZK for the relation $\mathcal{R}$ describing the well-formedness of the obfuscated programs $(\mathsf{S}_i)_{i\in[n]}$. Formally, a statement consists of the pair (S_i, i), whereas the corresponding witness is the triple containing the PRF keys k_i and K_i hard-coded in S_i and the randomness used for the obfuscation of the latter.

$\mathcal{P}_{\mathsf{AR}}[\mathsf{SendMsg}, k, K, i]$

Hard-coded. The algorithm $\mathsf{SendMsg}$, PRF keys k and K and the index i of the party.
Input. Oracle responses $(u, v) \in \{0,1\}^{\lambda \cdot m(\lambda)} \times \{0,1\}^{\lambda}$.

1. $(y_1^0, y_1^1, y_2^0, y_2^1, \ldots, y_m^0, y_m^1) \leftarrow F'_k(v)$
2. For every $j \in [m]$ set
$$x^j \leftarrow \begin{cases} 0 & \text{if } y_j^0 = u^j, \\ 1 & \text{if } y_j^1 = u^j, \\ \bot & \text{otherwise.} \end{cases}$$
3. If $x^j \neq \bot$ for every $j \in [m]$, output $(x^1, x^2, \ldots, x^m)$.
4. Set $r \leftarrow F_K(u, v)$.
5. Output $\mathsf{g}_i \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, i; r)$.

Fig. 12. The Anti-Rushing Program

Theorem 5.2. *If* $(\mathsf{SendMsg}, \mathsf{Output})$ *is a one-round* n*-party protocol,* $\mathsf{NIZK} = (\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Sim}_1, \mathsf{Sim}_2, \mathsf{Extract})$ *is a simulation-extractable NIZK with URS for the relation* $\mathcal{R}$*,* iO *is an indistinguishability obfuscator and* (F, Punct) *and* (F', Punct') *are two puncturable PRFs satisfying the properties described above, the protocol* $\Pi_{\mathsf{NoRush}} = (\mathsf{ARMsg}, \mathsf{AROutput})$ *described in Fig. 13 realizes* $\mathcal{F}_{\mathsf{NoRush}}$ *for* $\mathsf{SendMsg}$ *in the random oracle model with a URS.*

We prove Theorem 5.2 in Appendix A.

Theorem 5.3. *Suppose that* $\mathsf{DS} = (\mathsf{Gen}, \mathsf{Sample})$ *is a semi-maliciously secure distributed sampler for the distribution* $\mathcal{D}$*. Assume that there exists an anti-rusher for* $\mathsf{DS.Gen}$*. Then, there exists an actively secure distributed sampler for* $\mathcal{D}$*.*

On the novelty of this compiler. Observe that the idea of a compiler converting passive protocols into actively secure ones is not new. The most famous example is GMW [GMW87], which achieves this by adding ZK proofs proving the well-formedness of all the messages in the protocol. The novelty of our construction consists of doing this without increasing the number of rounds. GMW deals with rushing by requiring all the parties to commit to their randomness at the beginning of the protocol and then prove that all the messages in the interaction are

Anti-Rushing Compiler Π_{NoRush}

URS. The protocol needs a URS $\mathsf{urs} \xleftarrow{\$} \mathsf{NIZK.Gen}(1\!\!1^\lambda)$ for the NIZK proofs.

$\mathsf{ARMsg}(1\!\!1^\lambda, i, \mathsf{urs})$:

1. $k_i \xleftarrow{\$} \{0,1\}^\lambda$
2. $K_i \xleftarrow{\$} \{0,1\}^\lambda$
3. $w_i \xleftarrow{\$} \{0,1\}^{M(\lambda)}$
4. $\mathsf{S}_i \leftarrow \mathsf{iO}\big(1\!\!1^\lambda, \mathcal{P}_{\mathsf{AR}}[\mathsf{SendMsg}, k_i, K_i, i]; w_i\big)$ (see Fig. 12)
5. $\pi_i \xleftarrow{\$} \mathsf{Prove}\big(1\!\!1^\lambda, \mathsf{urs}, (\mathsf{S}_i, i), (k_i, K_i, w_i)\big)$
6. Output $\mathsf{armsg}_i := (\mathsf{S}_i, \pi_i)$.

$\mathsf{AROutput}\big(i, \big(\mathsf{armsg}_j = (\mathsf{S}_j, \pi_j)\big)_{j\in[n]}, \mathsf{urs}\big)$:

1. If there exists $j \in [n]$ such that $\mathsf{Verify}\big(\mathsf{urs}, \pi_j, (\mathsf{S}_j, j)\big) = 0$, output $\perp$.
2. Query $(\mathsf{S}_j, \pi_j)_{j\in[n]}$ to the random oracle $\mathcal{H}$ to get $\big(v, (u_j)_{j\in[n]}\big)$.
3. $\forall j \in [n]:\quad \mathsf{g}_j \leftarrow \mathsf{S}_j(u_j, v)$.
4. Output $(\mathsf{g}_j)_{j\in[n]}$ and $F_{K_i}(u_i, v)$.

Fig. 13. Anti-Rushing Compiler

consistent with the initial commitments. A passively secure one-round protocol would therefore be compiled, in the best case, into a 2-round one.

Although the techniques were inspired by [HJK+16], this work employs the ideas in a new context, generalising them to multiple players and applying them in multiparty protocols. Observe indeed that [HJK+16] devised the techniques to construct adaptively secure universal samplers. To some extent, we still use them to prevent the adversary from making adaptive choices.

6 Public-Key PCFs for Reverse-Samplable Correlations

We now consider the concept of a *distributed correlation sampler*, where the distribution $\mathcal{D}$ produces *private, correlated* outputs $R_1, R_2, \ldots, R_n$, where R_i is given only to the i-th party. This can also model the case where the distribution $\mathcal{D}$ has only one output $R = R_1 = \cdots = R_n$, which must be accessible only to the parties that took part in the computation (but not to outsiders; unlike with a distributed sampler).

PCGs and PCFs. The concept of distributed correlation samplers has been previously studied in the form of pseudorandom correlation generators (PCGs) [BCGI18, BCG+19a, BCG+19b, BCG+20b] and pseudorandom correlation functions (PCFs)[BCG+20a, OSY21]. These are tailored to distributions with n outputs, each one addressed to a different player. Specifically, they consist of two algorithms $(\mathsf{Gen}, \mathsf{Eval})$: Gen is used to generate n short correlated seeds or

keys, one for each party. Eval is then used to locally expand the keys and non-interactively produce a large amount of correlated randomness, analogously to the non-correlated setting of a PRG (for PCG) or PRF (for PCF).

Both PCGs and PCFs implicitly rely on a trusted dealer for the generation and distribution of the output of Gen, which in practice can be realized using a secure multiparty protocol. The communication overhead of this computation should be small, compared with the amount of correlated randomness obtained from Eval.

If we consider a one-round protocol to distribute the output of Gen, the message of the i-th party and the corresponding randomness r_i act now as a kind of public/private key pair (r_i is necessary to retrieve the i-th output.) Such a primitive is called a *public-key PCF* [OSY21]. Orlandi *et al.* [OSY21] built public-key PCFs for the random OT and vector-OLE correlations based on Paillier encryption with a common reference string (a trusted RSA modulus). In this section, we will build public-key PCFs for general correlations, while avoiding trusted setups.

6.1 Correlation Functions and their Properties

Instead of considering singe-output distributions $\mathcal{D}$, we now consider n-output correlations $\mathcal{C}$. We also allow different samples from $\mathcal{C}$ to themselves be correlated by some secret parameters, which allows handling correlations such as vector-OLE and authenticated multiplication triples (where each sample depends on some fixed MAC keys). This is modelled by allowing each party i to input a *master secret* mk_i into $\mathcal{C}$. These additional inputs are independently sampled by each party using an algorithm Secret.

Some example correlations. Previous works have focussed on a simple class of *additive correlations*, where the outputs $R_1, \ldots, R_n$ form an additive secret sharing of values sampled from a distribution. This captures, for instance, oblivious transfer, (vector) oblivious linear evaluation and (authenticated) multiplication triples, which are all useful correlations for secure computation tasks. Vector OLE and authenticated triples are also examples requiring a master secret, which is used to fix a secret scalar or secret MAC keys used to produce samples. Assuming LWE, we can construct public-key PCFs for any additive correlation [BCG+20a], using homomorphic secret-sharing based on multi-key FHE [DHRW16]. However, we do not know how to build PCFs for broader classes of correlations, except for in the two-party setting and relying on subexponentially secure iO [DHRW16].

As motivation, consider the following important types of non-additive correlations:

- *Pseudorandom secret sharing.* This can be seen as a correlation that samples sharings of uniformly random values under some linear secret sharing scheme. Even for simple t-out-of-n threshold schemes such as Shamir, the best previous construction requires $\binom{n}{t}$ complexity [CDI05].

– *Garbled circuits.* In the two-party setting, one can consider a natural garbled circuit correlation, which for some circuit C, gives a garbling of C to one party, and all pairs of input wire labels to the other party. Having such a correlation allows preprocessing for secure 2-PC, where in the online phase, the parties just use oblivious transfer to transmit the appropriate input wire labels.[3] Similarly, this can be extended to the multi-party setting, by for instance, giving n parties the garbled circuit together with a secret-sharing of the input wire labels.

For garbled circuits, it may also be useful to consider a variant that uses a master secret, if e.g. we want each garbled circuit to be sampled with a fixed offset used in the free-XOR technique [KS08].

Reverse-Samplable Correlations. The natural way to define a public-key PCF would be a one-round protocol implementing the functionality that samples from the correlation function $\mathcal{C}$ and distributes the outputs. However, Boyle *et al.* [BCG+19b] prove that for PCGs, any construction satisfying this definition in the plain model would require that the messages be at least as long as the randomness generated, which negates one of the main advantages of using a PCF. Following the approach of Boyle *et al.*, in this section we adopt a weaker definition. We require that no adversary can distinguish the real samples of the honest parties from simulated ones which are *reverse sampled* based on the outputs of the corrupted players. This choice restricts the set of correlation functions to those whose outputs are efficiently reverse-samplable[4]. We formalise this property below.

Definition 6.1 (Reverse Samplable Correlation Function with Master Secrets). *An n-party correlation function with master secrets is a pair of PPT algorithms $(\mathsf{Secret}, \mathcal{C})$ with the following syntax:*

– Secret *takes as input the security parameter $1\!\!1^\lambda$ and the index of a party $i \in [n]$. It outputs the i-th party's master correlation secret mk_i.*
– *$\mathcal{C}$ takes as input the security parameter $1\!\!1^\lambda$ and the master secrets $\mathsf{mk}_1, \ldots, \mathsf{mk}_n$. It outputs n correlated values $R_1, R_2, \ldots, R_n$, one for each party.*

We say that $(\mathsf{Secret}, \mathcal{C})$ is reverse samplable if there exists a PPT algorithm $\mathsf{RSample}$ such that, for every set of corrupted parties $C \subsetneq [n]$ and master secrets $(\mathsf{mk}_i)_{i\in[n]}$ and $(\mathsf{mk}'_h)_{h\in H}$ in the image of Secret, no PPT adversary is able to

[3] Note that formally, in the presence of malicious adversaries, preprocessing garbled circuits in this way requires the garbling scheme to be adaptively secure [BHR12].

[4] In the examples above, reverse-samplability is possible for pseudorandom secret-sharing, but not for garbled circuits, since we should not be able to find valid input wire labels when given only a garbled circuit.

distinguish between $\mathcal{C}(1\!\!1^\lambda, \mathsf{mk}_1, \mathsf{mk}_2, \ldots, \mathsf{mk}_n)$ *and*

$$\left\{ (R_1, R_2, \ldots, R_n) \;\middle|\; \begin{array}{l} \forall i \in C: \quad \mathsf{mk}'_i \leftarrow \mathsf{mk}_i \\ (R'_1, R'_2, \ldots, R'_n) \xleftarrow{\$} \mathcal{C}(1\!\!1^\lambda, \mathsf{mk}'_1, \mathsf{mk}'_2, \ldots, \mathsf{mk}'_n) \\ \forall i \in C: \quad R_i \leftarrow R'_i \\ (R_h)_{h \in H} \xleftarrow{\$} \mathsf{RSample}\big(1\!\!1^\lambda, C, (R_i)_{i \in C}, (\mathsf{mk}_i)_{i \in C}, (\mathsf{mk}_h)_{h \in H}\big) \end{array} \right\}$$

Notice that indistinguishability cannot rely on the secrecy of the master secrets $(\mathsf{mk}_i)_{i \in [n]}$ and $(\mathsf{mk}'_h)_{h \in H}$, since the adversary could know their values. Furthermore, RSample does not take as input the same master secrets that were used for the generation of the outputs of the corrupted parties. The fact that indistinguishability holds in spite of this implies that the elements $(R_i)_{i \in C}$ leak no information about the master secrets of the honest players.

6.2 Defining Public Key PCFs

We now formalise the definition of public key PCF as it was sketched at the beginning of the section. We start by specifying the syntax, we will then focus our attention on security, in particular against semi-malicious and active adversaries.

Definition 6.2 (Public-Key PCF with Master Secrets). *A public-key PCF for the n-party correlation function with master secrets* $(\mathsf{Secret}, \mathcal{C})$ *is a pair of PPT algorithms* $(\mathsf{Gen}, \mathsf{Eval})$ *with the following syntax:*

- Gen *takes as input the security parameter* $1\!\!1^\lambda$ *and the index of a party* $i \in [n]$*, and outputs the PCF key pair* $(\mathsf{sk}_i, \mathsf{pk}_i)$ *of the* i*-th party.* Gen *needs* $L(\lambda)$ *bits of randomness.*
- Eval *takes as input an index* $i \in [n]$*,* n *PCF public keys, the* i*-th PCF private key* sk_i *and a nonce* $x \in \{0,1\}^{l(\lambda)}$*. It outputs a value* R_i *corresponding to the* i*-th output of* $\mathcal{C}$*.*

Every public-key PCF $(\mathsf{Gen}, \mathsf{Eval})$ for $\mathcal{C}$ induces a one-round protocol $\Pi_{\mathcal{C}}$. This is the natural construction in which every party broadcasts pk_i output by Gen, and then runs Eval on all the parties' messages, its own private key and various nonces.

Definition 6.3 (Semi-Maliciously Secure Public-Key PCF for Reverse Samplable Correlation). *Let* $(\mathsf{Secret}, \mathcal{C})$ *be an* n*-party, reverse samplable correlation function with master secrets. A public-key PCF* $(\mathsf{Gen}, \mathsf{Eval})$ *for* $(\mathsf{Secret}, \mathcal{C})$ *is semi-maliciously secure if the following properties are satisfied.*

- ***Correctness.*** *No PPT adversary can win the game* $\mathcal{G}_{PCF\text{-}Corr}(\lambda)$ *(see Fig. 14) with noticeable advantage.*
- ***Security.*** *There exists a PPT extractor* Extract *such that for every set of corrupted parties* $C \subsetneq [n]$ *and corresponding randomness* $(\rho_i)_{i \in C}$*, no PPT adversary can win the game* $\mathcal{G}^{C,(\rho_i)_{i \in C}}_{PCF\text{-}Sec}(\lambda)$ *(see Fig. 15) with noticeable advantage.*

$\mathcal{G}_{\text{PCF-Corr}}(\lambda)$

Initialisation.

1. $b \xleftarrow{\$} \{0,1\}$
2. $\forall i \in [n]: \quad (\mathsf{sk}_i, \mathsf{pk}_i) \xleftarrow{\$} \mathsf{Gen}(1\!\!1^\lambda, i)$
3. $\forall i \in [n]: \quad \mathsf{mk}'_i \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, i)$
4. Activate the adversary with input $(1\!\!1^\lambda, (\mathsf{pk}_i)_{i\in[n]})$.

Repeated querying. On input $(\mathsf{Correlation}, x)$ from the adversary where $x \in \{0,1\}^{l(\lambda)}$, compute

1. $\forall i \in [n]: \quad R_i^0 \leftarrow \mathsf{Eval}(i, \mathsf{pk}_1, \ldots, \mathsf{pk}_n, \mathsf{sk}_i, x)$
2. $(R_i^1)_{i\in[n]} \xleftarrow{\$} \mathcal{C}(1\!\!1^\lambda, \mathsf{mk}'_1, \ldots, \mathsf{mk}'_n)$
3. Give $(R_1^b, R_2^b, \ldots, R_n^b)$ to the adversary.

Output. The adversary wins if its final output is b.

Fig. 14. Correctness Game for the Public-Key PCF

Correctness requires that the samples output by the PCF are indistinguishable from those produced by $\mathcal{C}$ even if the adversary receives all the public keys. Security instead states that a semi-malicious adversary learns no information about the samples and the master secrets of the honest players except what can be deduced from the outputs of the corrupted parties themselves.

Like for distributed samplers, the above definition can be adapted to passive security by modifying the security game. Specifically, it would be sufficient to sample the randomness of the corrupted parties inside the game, perhaps relying on a simulator when $b = 1$.

In our definition, nonces are adaptively chosen by the adversary; however, in a *weak* PCF [BCG+20a], the nonces are sampled randomly or selected by the adversary ahead of time. We can define a weak public-key PCF similarly, and use the same techniques as Boyle *et al.* [BCG+20a] to convert a weak public-key PCF into a public-key PCF by means of a random oracle.

Active security. We define actively secure public-key PCFs using an ideal functionality, similarly to how we defined actively secure distributed samplers.

Definition 6.4 (Actively Secure Public-Key PCF for Reverse Samplable Correlation). *Let* $(\mathsf{Secret}, \mathcal{C})$ *be an* n*-party reverse samplable correlation function with master secrets. A public-key PCF* $(\mathsf{Gen}, \mathsf{Eval})$ *for* $(\mathsf{Secret}, \mathcal{C})$ *is* actively secure *if the corresponding one-round protocol* $\Pi_{\mathcal{C}}$ *implements the functionality* $\mathcal{F}_{\mathcal{C}}^{\mathsf{RSample}}$ *(see Fig. 16) against a static and active adversary corrupting up to* $n-1$ *parties.*

Any protocol that implements $\mathcal{F}_{\mathcal{C}}^{\mathsf{RSample}}$ will require either a CRS or a random oracle; this is inherent for meaningful correlation functions, since the simulator needs to retrieve the values $(R_i)_{i\in C}$ in order to forward them to $\mathcal{F}_{\mathcal{C}}^{\mathsf{RSample}}$. Therefore, some kind of trapdoor is needed.

$\mathcal{G}^{C,(\rho_i)_{i\in C}}_{\text{PCF-Sec}}(\lambda)$

Initialisation.

1. $b \xleftarrow{\$} \{0,1\}$
2. $\forall h \in H: \quad \rho_h \xleftarrow{\$} \{0,1\}^{L(\lambda)}$
3. $\forall i \in [n]: \quad (\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow \mathsf{Gen}(1\!\!1^\lambda, i;\, \rho_i)$
4. $(\mathsf{mk}_i)_{i\in C} \leftarrow \mathsf{Extract}(C, \rho_1, \rho_2, \ldots, \rho_n)$.
5. $\forall h \in H: \quad \mathsf{mk}'_h \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, h)$
6. Activate the adversary with $1\!\!1^\lambda$ and provide it with $(\mathsf{pk}_i)_{i\in[n]}$ and $(\rho_i)_{i\in C}$.

Repeated querying. On input $(\mathsf{Correlation}, x)$ from the adversary where $x \in \{0,1\}^{l(\lambda)}$, compute

1. $\forall i \in [n]: \quad R^0_i \leftarrow \mathsf{Eval}(i, \mathsf{pk}_1, \ldots, \mathsf{pk}_n, \mathsf{sk}_i, x)$
2. $\forall i \in C: \quad R^1_i \leftarrow R^0_i$
3. $(R^1_h)_{h\in H} \xleftarrow{\$} \mathsf{RSample}\big(1\!\!1^\lambda, C, (R^1_i)_{i\in C}, (\mathsf{mk}_i)_{i\in C}, (\mathsf{mk}'_h)_{h\in H}\big)$
4. Give $(R^b_1, R^b_2, \ldots, R^b_n)$ to the adversary.

Output. The adversary wins if its final output is b.

Fig. 15. Security Game for the Public-Key PCF

Notice also that the algorithm $\mathsf{RSample}$ takes as input the master secrets of the corrupted parties. We can therefore assume that whenever the values $(R_i)_{i\in C}$ chosen by the adversary are inconsistent with $(\mathsf{mk}_i)_{i\in C}$ or with $\mathcal{C}$ itself, the output of the reverse sampler is $\perp$. As a consequence, an actively secure public-key PCF must not allow the corrupted parties to select these irregular outputs; otherwise distinguishing between real world and ideal world would be trivial.

6.3 Public-Key PCF with Trusted Setup

We will build our semi-maliciously secure public-key PCF by first relying on a trusted setup and then removing it by means of a distributed sampler. A public-key PCF with trusted setup is defined by Def. 6.2 to include an algorithm Setup that takes as input the security parameter $1\!\!1^\lambda$ and outputs a CRS. The CRS is then provided as an additional input to the evaluation algorithm Eval, but not to the generation algorithm Gen. (If Gen required the CRS, then substituting Setup with a distributed sampler would give us a two-round protocol, not a one-round protocol.)

We say that a public-key PCF with trusted setup is semi-maliciously secure if it satisfies Def. 6.3, after minor tweaks to the games $\mathcal{G}_{\text{PCF-Corr}}(\lambda)$ and $\mathcal{G}^{C,(\rho_i)_{i\in C}}_{\text{PCF-Sec}}(\lambda)$ to account for the modified syntax. Notice that in the latter, the extractor needs to be provided with the CRS but not with the randomness used to produce it. If that was not the case, we would not be able to use a distributed sampler to remove the CRS.

Definition of public key PCF with trusted setup. We formalise the concept of public key PCF with trusted setup describing its syntax and security proper-

$\mathcal{F}_{\mathcal{C}}^{\mathsf{RSample}}$

Initialisation. On input Init from every honest party and the adversary, the functionality samples $\mathsf{mk}_h \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, h)$ for every $h \in H$ and waits for $(\mathsf{mk}_i)_{i\in C}$ from the adversary.

Correlation. On input a fresh nonce $x \in \{0,1\}^{l(\lambda)}$ from a party P_j, the functionality waits for $(R_i)_{i\in C}$ from the adversary. Then, it computes

$$(R_h)_{h\in H} \xleftarrow{\$} \mathsf{RSample}\big(1\!\!1^\lambda, C, (R_i)_{i\in C}, (\mathsf{mk}_i)_{i\in C}, (\mathsf{mk}_h)_{h\in H}\big),$$

sends R_j to P_j and stores $\big(x, (R_i)_{i\in[n]}\big)$. If x has already been queried, the functionality retrieves the stored tuple $\big(x, (R_i)_{i\in[n]}\big)$ and outputs R_j to P_j.

Fig. 16. The Actively Secure Public-Key PCF Functionality for Reverse Samplable Correlation

ties. The definitions closely resemble those of public key PCF (see Def. 6.2 and Def. 6.3). The only difference consists in the CRS generated by the setup algorithm, which is now provided as an additional input to Eval and $\mathsf{Extract}$ and whose value is always disclosed to the adversary in the security games.

Definition 6.5 (Public Key PCF with Trusted Setup). *A public key PCF with trusted setup for the n-party correlation function with master secret $(\mathsf{Secret}, \mathcal{C})$ is a triple of PPT algorithms $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval})$ with the following syntax:*

- Setup *takes as input the security parameter $1\!\!1^\lambda$ and outputs a CRS S.*
- Gen *takes as input the security parameter $1\!\!1^\lambda$ and the index of a party $i \in [n]$, outputting the PCF key pair $(\mathsf{sk}_i, \mathsf{pk}_i)$ of the i-th party. The algorithm needs $L(\lambda)$ bits of randomness.*
- Eval *takes as input an index $i \in [n]$, a CRS S, n PCF public keys, one for each party, the PCF private key sk_i of the i-th party and a nonce $x \in \{0,1\}^{l(\lambda)}$. The output is a value R_i corresponding to the i-th output of $\mathcal{C}$.*

Definition 6.6 (Semi-Maliciously Secure Public Key PCF with Trusted Setup for Reverse Samplable Correlation). *Let $(\mathsf{Secret}, \mathcal{C})$ be an n-party, reverse samplable correlation function with master secret. A public key PCF with trusted setup $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval})$ for $(\mathsf{Secret}, \mathcal{C})$ is semi-maliciously secure if the following properties are satisfied.*

- ***Correctness.*** *No PPT adversary can win the game $\mathcal{G}_{SetupCorr}(\lambda)$ (see Fig. 17) with noticeable advantage.*
- ***Security.*** *There exists a PPT extractor $\mathsf{Extract}$ such that for every set of corrupted parties $C \subsetneq [n]$ and corresponding randomness $(\rho_i)_{i\in C}$, no PPT adversary can win the game $\mathcal{G}_{SetupSec}^{C,(\rho_i)_{i\in C}}(\lambda)$ (see Fig. 18) with noticeable advantage.*

$\mathcal{G}_{\text{SetupCorr}}(\lambda)$

Initialisation.

1. $b \xleftarrow{\$} \{0,1\}$
2. $S \xleftarrow{\$} \mathsf{Setup}(1\!\!1^\lambda)$
3. $\forall i \in [n]: \quad (\mathsf{sk}_i, \mathsf{pk}_i) \xleftarrow{\$} \mathsf{Gen}(1\!\!1^\lambda, i)$
4. $\forall i \in [n]: \quad \mathsf{mk}'_i \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, i)$
5. Activate the adversary with $1\!\!1^\lambda$ and provide it with $(\mathsf{pk}_i)_{i\in[n]}$ and S.

Repeated querying. On input $(\mathsf{Correlation}, x)$ from the adversary where $x \in \{0,1\}^{l(\lambda)}$, compute

1. $\forall i \in [n]: \quad R_i^0 \leftarrow \mathsf{Eval}(i, S, \mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_n, \mathsf{sk}_i, x)$
2. $(R_i^1)_{i\in[n]} \xleftarrow{\$} \mathcal{C}(1\!\!1^\lambda, \mathsf{mk}'_1, \ldots, \mathsf{mk}'_n)$
3. Give $(R_1^b, R_2^b, \ldots, R_n^b)$ to the adversary.

Output. The adversary wins if its final output is b.

Fig. 17. Correctness Game for Public Key PCFs with Setup

$\mathcal{G}^{C,(\rho_i)_{i\in C}}_{\text{SetupSec}}(\lambda)$

Initialisation.

1. $b \xleftarrow{\$} \{0,1\}$
2. $S \xleftarrow{\$} \mathsf{Setup}(1\!\!1^\lambda)$
3. $\forall h \in H: \quad \rho_i \xleftarrow{\$} \{0,1\}^{L(\lambda)}$
4. $\forall i \in [n]: \quad (\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow \mathsf{Gen}(1\!\!1^\lambda, i;\, \rho_i)$
5. $(\mathsf{mk}_i)_{i\in C} \leftarrow \mathsf{Extract}(C, S, \rho_1, \rho_2, \ldots, \rho_n)$.
6. $\forall h \in H: \quad \mathsf{mk}'_h \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, h)$
7. Activate the adversary with $1\!\!1^\lambda$ and provide it with $(\mathsf{pk}_i)_{i\in[n]}$, S and $(\rho_i)_{i\in C}$.

Repeated querying. On input $(\mathsf{Correlation}, x)$ from the adversary where $x \in \{0,1\}^{l(\lambda)}$, compute

1. $\forall i \in [n]: \quad R_i^0 \leftarrow \mathsf{Eval}(i, S, \mathsf{pk}_1, \ldots, \mathsf{pk}_n, \mathsf{sk}_i, x)$
2. $\forall i \in C: \quad R_i^1 \leftarrow R_i^0$
3. $(R_h^1)_{h\in H} \xleftarrow{\$} \mathsf{RSample}\big(1\!\!1^\lambda, C, (R_i^1)_{i\in C}, (\mathsf{mk}_i)_{i\in C}, (\mathsf{mk}'_h)_{h\in H}\big)$
4. Give $(R_1^b, R_2^b, \ldots, R_n^b)$ to the adversary.

Output. The adversary wins if its final output is b.

Fig. 18. Security Game for Public Key PCFs with Setup

Our public-key PCF with trusted setup. Our construction is based once again on iO. The key of every party i is a simple PKE pair $(\mathsf{sk}_i, \mathsf{pk}_i)$. The generation of the correlated samples and their distribution is handled by the CRS, which is an obfuscated program. Specifically, the latter takes as input the public keys of the parties and a nonce $x \in \{0,1\}^{l(\lambda)}$. After generating the master secrets $\mathsf{mk}_1, \mathsf{mk}_2, \ldots, \mathsf{mk}_n$ using Secret and the correlated samples $R_1, R_2, \ldots, R_n$ using $\mathcal{C}$, the program protects their privacy by encrypting them under the provided public keys. Specifically, R_i and mk_i are encrypted using pk_i, making the i-th party the only one able to retrieve the underlying plaintext.

The randomness used for the generation of the samples, the master secrets and the encryption is produced by means of two puncturable PRF keys k and K, known to the CRS program. The CRS program is equipped with two keys: k and K. The first one is used to generate the master secrets; the input to the PRF is the sequence of all public keys $(\mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_n))$. The master secrets remain the same if the nonce x varies. The second PRF key is used to generate the randomness fed into $\mathcal{C}$ and the encryption algorithm; here, the PRF input consists of all the program inputs. As a result, any slight change in the inputs leads to completely unrelated ciphertexts and samples.

On the size of the nonce space. Unfortunately, in order to obtain semi-maliciously security, we need to assume that the nonce space is of polynomial size. In the security proof, we need to change the behaviour of the CRS program for all nonces. This is due to the fact that we cannot rely on the reverse samplability of the correlation function as long as the program contains information about the real samples of the honest players. If the number of nonces is exponential, our security proof would rely on a non-polynomial number of hybrids and therefore we would need to assume the existence of sub-exponentially secure primitives.

The formal description of our solution. Our public-key PCF with trusted setup for $(\mathsf{Secret}, \mathcal{C})$ is described in Fig. 19 together with the program $\mathcal{P}_{\mathsf{CG}}$ used as a CRS.

Our solution relies on an IND-CPA PKE scheme $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and two puncturable PRFs F and F'. We assume that the output of the first one is naturally split into $n+1$ blocks, the initial one as big as the randomness needed by $\mathcal{C}$, the remaining ones the same size as the random tape of $\mathsf{PKE.Enc}$. We also assume that the output of F' is split into n blocks as big as the randomness used by Secret.

Theorem 6.7 (Public Key PCFs with Trusted Setup). *Let* $(\mathsf{Secret}, \mathcal{C})$ *be an* n*-party, reverse samplable correlation function with master secrets. If* $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is an IND-CPA PKE scheme,* iO *is an indistinguishability obfuscator,* (F, Punct) *and* (F', Punct') *are puncturable PRFs with the properties described above and* $l(\lambda)$ *is* $\mathsf{polylog}(\lambda)$*, the construction presented in Fig. 19 is a semi-maliciously secure public-key PCF with trusted setup for* $(\mathsf{Secret}, \mathcal{C})$*.*

Furthermore, if PKE*,* iO*,* (F, Punct) *and* (F', Punct') *are sub-exponentially secure, the public-key PCF with trusted setup is semi-maliciously secure even if* $l(\lambda)$ *is* $\mathsf{poly}(\lambda)$*.*

Public-Key PCF with Trusted Setup

$\mathsf{Setup}(1\!\!1^\lambda)$

1. $k \xleftarrow{\$} \{0,1\}^\lambda$
2. $K \xleftarrow{\$} \{0,1\}^\lambda$
3. Output $\mathsf{CGP} \xleftarrow{\$} \mathsf{iO}\big(1\!\!1^\lambda, \mathcal{P}_{\mathsf{CG}}[k,K]\big)$

$\mathsf{Gen}(1\!\!1^\lambda, i)$

1. Output $(\mathsf{sk}_i, \mathsf{pk}_i) \xleftarrow{\$} \mathsf{PKE.Gen}(1\!\!1^\lambda)$

$\mathsf{Eval}(i, \mathsf{CGP}, \mathsf{pk}_1, \ldots, \mathsf{pk}_n, \mathsf{sk}_i, x)$

1. $(c_1, c_2, \ldots, c_n) \leftarrow \mathsf{CGP}(\mathsf{pk}_1, \ldots, \mathsf{pk}_n, x)$
2. $(R_i, \mathsf{mk}_i) \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}_i, c_i)$
3. Output R_i.

$\mathcal{P}_{\mathsf{CG}}[k,K]$

Hard-coded. Two puncturable PRF keys k and K.
Input. n public keys $\mathsf{pk}_1, \ldots, \mathsf{pk}_n$ and a nonce $x \in \{0,1\}^{l(\lambda)}$.

1. $(r, r_1, r_2, \ldots, r_n) \leftarrow F_K(\mathsf{pk}_1, \ldots, \mathsf{pk}_n, x)$.
2. $(s_1, s_2, \ldots, s_n) \leftarrow F'_k(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$
3. $\forall i \in [n]: \quad \mathsf{mk}_i \leftarrow \mathsf{Secret}(1\!\!1^\lambda, i; s_i)$
4. $(R_1, R_2, \ldots, R_n) \leftarrow \mathcal{C}(1\!\!1^\lambda, \mathsf{mk}_1, \ldots, \mathsf{mk}_n; r)$
5. $\forall i \in [n]: \quad c_i \leftarrow \mathsf{PKE.Enc}\big(\mathsf{pk}_i, (R_i, \mathsf{mk}_i); r_i\big)$
6. Output $c_1, c_2, \ldots, c_n$.

Fig. 19. A Public-Key PCF with Trusted Setup

In both cases, the size of the CRS and the PCF keys is $\mathsf{poly}(l)$.

We prove Theorem 6.7 in Appendix B.

6.4 Our Public-Key PCFs

As mentioned in the previous section, once we obtain a semi-maliciously secure public-key PCF with trusted setup, we can easily remove the CRS using a distributed sampler. We therefore obtain a public-key PCF with security against semi-malicious adversaries. If the size of the CRS and the keys of the initial construction is logarithmic in the size of the nonce space, the key length after removing the setup is still polynomial in $l(\lambda)$.

Theorem 6.8 (Semi-Maliciously Secure Public Key PCFs). *Let* $(\mathsf{Secret}, \mathcal{C})$ *be an* n*-party, reverse samplable correlation function with master secrets. Suppose that* $\mathsf{pkPCFS} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval})$ *is a semi-maliciously secure public-key*

PCF with trusted setup for $(\mathsf{Secret}, \mathcal{C})$*. Moreover, assume that there exists a semi-maliciously secure* n*-party distributed sampler for* $\mathsf{pkPCFS.Setup}$*. Then, public-key PCFs for* $(\mathsf{Secret}, \mathcal{C})$ *with semi-malicious security exist.*

We will not prove Theorem 6.8 formally. Security follows from the fact that distributed samplers implement the functionality that samples directly from the underlying distribution. From this point of view, it is fundamental that the randomness input into Setup is not given as input to the extractor of the public-key PCF pkPCFS.

Active security in the random oracle model. If we rely on a random oracle, it is easy to upgrade a semi-maliciously secure public-key PCF to active security. We can use an anti-rusher (see Section 5.1) to deal with rushing and malformed messages. If the key size of the semi-malicious construction is polynomial in $l(\lambda)$, after compiling with the anti-rusher, the key length is still $\mathsf{poly}(l)$. The technique described above allows us to deduce the security of our solution from the semi-malicious security of the initial public-key PCF. The result is formalised by the following theorem. Again, we will not provide a formal proof.

Theorem 6.9 (Actively Secure Public Key PCFs in the Random Oracle Model). *Let* $(\mathsf{Secret}, \mathcal{C})$ *be an* n*-party, reverse samplable correlation function with master secret. Assume that* $\mathsf{pkPCF} = (\mathsf{Gen}, \mathsf{Eval})$ *is a semi-maliciously secure public-key PCFs for* $(\mathsf{Secret}, \mathcal{C})$ *and suppose there exists an anti-rusher for the associated protocol. Then, actively secure public-key PCFs for* $(\mathsf{Secret}, \mathcal{C})$ *exist.*

Active security from sub-exponentially secure primitives. So far, all our constructions rely on polynomially secure primitives. However, we often work in the random oracle model. We now show that it is possible to build actively secure public-key PCFs in the URS model assuming the existence of sub-exponentially secure primitives. Furthermore, these constructions come with no restrictions on the size of the nonce space.

Our solution is obtained by assembling a sub-exponentially and semimaliciously secure public-key PCF with trusted setup with a sub-exponentially and semi-maliciously secure distributed sampler. We add witness-extractable NIZKs proving the well-formedness of the messages. Like for our semi-malicious construction, if the size of the CRS and the keys of the public-key PCF with trusted setup is polynomial in the nonce length $l(\lambda)$, after composing with the DS, the key size remains $\mathsf{poly}(l)$.

Theorem 6.10 (Actively Secure Public Key PCFs from Subexponentially Secure Primitives). *Let* $(\mathsf{Secret}, \mathcal{C})$ *be an* n*-party, reverse samplable correlation function with master secret. Suppose that* $\mathsf{pkPCFS} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval})$ *is a sub-exponentially and semi-maliciously secure public-key PCF with trusted setup for* $(\mathsf{Secret}, \mathcal{C})$*. Assume that there exists a sub-exponentially and semi-maliciously secure* n*-party distributed sampler for* $\mathsf{pkPCFS.Setup}$*. If there exist*

simulation-extractable NIZKs with URS proving the well-formedness of the sampler shares and the PCF public keys, there exists an actively secure public-key PCF for $(\mathsf{Secret}, \mathcal{C})$ *in the URS model.*

We prove Theorem 6.10 in Appendix C.

7 Ideal Public Key PCFs and Distributed Universal Samplers

We now inspect the feasibility of ideal public key PCFs. The term is used to denote one-round constructions implementing the functionality that directly samples the outputs from adaptively chosen correlations and distributes them to the parties. In contrast with the other PCFs we described, ideal public key PCFs are not tailored to any specific correlation function; instead, the correlation function can be chosen on the fly. However, they can exist only in the random oracle model.

Defining ideal public key PCFs. We deal with generic correlations $\mathcal{C}$ without master secrets. $\mathcal{C}$ takes no inputs, and generates n correlated outputs. It does not need to satisfy any specific properties (in particular it is not required to be reverse samplable). Since the correlations supported by an ideal public key PCF are restricted to those whose description is polynomially bounded, we define the class of (n, ℓ, r, t)-correlations as the set of functions mapping r bits of randomness into n t-bit outputs and having an ℓ-bit description as a circuit.

The syntax of ideal public key PCFs is derived from that of their non-ideal counterparts (see Def. 6.2). The only difference is that the evaluation algorithm Eval takes as input the description of an (n, ℓ, r, t)-correlation instead of a nonce.

Definition 7.1 (Ideal Public-Key PCF). *Let $\ell(\lambda)$, $r(\lambda)$ and $t(\lambda)$ be polynomials. An ideal public-key PCF for (n, ℓ, r, t)-correlations is a pair of PPT algorithms* $(\mathsf{Gen}, \mathsf{Eval})$ *with the following syntax:*

- Gen *takes as input the security parameter $\mathbb{1}^\lambda$ and the index of a party $i \in [n]$, and outputs the PCF key pair $(\mathsf{sk}_i, \mathsf{pk}_i)$ of the i-th party.*
- Eval *takes as input an index $i \in [n]$, n PCF public keys, the i-th PCF private key sk_i and the description of an (n, ℓ, r, t)-correlation $\mathcal{C}$. It outputs a value R_i corresponding to the i-th output of $\mathcal{C}$.*

Definition 7.2 (Ideal Public Key PCF with Active Security). *An ideal public key PCF* $(\mathsf{Gen}, \mathsf{Eval})$ *for (n, ℓ, r, t)-correlations satisfies* active security *if the corresponding one-round protocol $\Pi_{\mathcal{C}}$ implements the functionality $\mathcal{F}_{\mathcal{C}}^{\mathsf{Ideal}}$ (see Fig. 20) against a static and active adversary corrupting up to $n - 1$ parties.*

Like in the definition of the actively secure distributed sampler (see Def. 3.3), the adversary is allowed to request different samples stored under different labels. Afterwards, the adversary can specify a label of its choice, forcing the functionality to output the associated values to the honest players. We must allow

$\mathcal{F}_{\mathcal{C}}^{\mathsf{Ideal}}$

Initialisation. On input Init from every honest party and the adversary, the functionality activates and enters the querying phase.

Querying phase.

- On input $(\mathsf{Query}, \mathsf{id}, \mathcal{C})$ where $\mathcal{C}$ is a (n, ℓ, r, t)-correlation that has not been queried previously, the functionality samples $(R_i)_{i\in[n]} \xleftarrow{\$} \mathcal{C}(1\!\!1^\lambda)$, sends $(R_i)_{i\in C}$ to the adversary and stores the tuple $\left(\mathsf{id}, \mathcal{C}, (r_i)_{i\in[n]}\right)$.
- On input $(\mathsf{Choice}, \hat{\mathsf{id}})$ from the adversary, the functionality stores $\hat{\mathsf{id}}$, ends the querying phase and begins the correlation phase. If $\hat{\mathsf{id}} = \mathsf{Abort}$, the functionality outputs $\perp$ to every honest party and halts.

Correlation phase. On input a (n, ℓ, r, t)-correlation $\mathcal{C}$ from party P_i, where a tuple $\left(\hat{\mathsf{id}}, \mathcal{C}, (R_i)_{i\in[n]}\right)$ has not been stored previously, the functionality samples $(R_1, R_2, \ldots, R_n) \xleftarrow{\$} \mathcal{C}(1\!\!1^\lambda)$, stores $\left(\hat{\mathsf{id}}, \mathcal{C}, (R_i)_{i\in[n]}\right)$ and outputs R_i to P_i. If $\left(\hat{\mathsf{id}}, \mathcal{C}, (R_i)_{i\in[n]}\right)$ has been previously stored (in either the querying or correlation phase), the functionality outputs R_i to P_i.

Fig. 20. The Functionality for Ideal Public Key PCFs with Active Security

this kind of influence in order to model rushing; an active adversary can always wait for the messages of the honest parties and adaptively choose the reply of the corrupted players. She is allowed to rerun the procedure as many time as it desire, repeatedly re-generating the messages of the corrupted parties and obtaining different collections of samples. The adversary can then use the messages that led to the most favourable results.

We will build our ideal public key PCFs upon a new primitive called a *distributed universal sampler*. We will present and analyse it in the following subsection.

7.1 Distributed Universal Samplers

A distributed universal sampler (DUS) generalises the concept of distributed sampler. Recall that a distributed sampler is tailored to output a single sample from some fixed distribution $\mathcal{D}$. In some applications, for instance when we need to sample from multiple distributions, chosen on-the-fly, this may be too restrictive.

What we want instead is analogous to a universal sampler (US) [HJK+16], where a trusted dealer first generates and publishes a sampler U. Later, the parties can use U to sample from arbitrary distributions, learning no additional information about the randomness used to generate the output. With a *distributed* universal sampler, we aim to remove the trusted dealer, sampling from generic distributions in a distributed way and with only one round of interaction.

Formally, (distributed) universal samplers do not support completely generic distributions, but are instead restricted to those whose descriptions are polynomially-

bounded. We therefore define the class of (ℓ, r, t)-distributions as the set of all distributions converting r bits of randomness into a t-bit output and having an ℓ-bit description as a circuit.

The syntax of a DUS is obtained by augmenting the Sample algorithm of a DS with an additional input, namely the description of the distribution from which to sample the output.

Definition 7.3 (Distributed Universal Sampler). *Let $\ell(\lambda)$, $r(\lambda)$ and $t(\lambda)$ be polynomials. An n-party distributed universal sampler for (ℓ, r, t)-distributions consists of a pair of PPT algorithms $(\mathsf{Gen}, \mathsf{Sample})$ with the following syntax.*

1. Gen *is a probabilistic algorithm taking as input the security parameter $1\!\!1^\lambda$ and a party index $i \in [n]$ and outputting a sampler share U_i for party i. Suppose that the procedure needs $L(\lambda)$ bits of randomness.*
2. Sample *is a deterministic algorithm taking as input n shares of the sampler $U_1, U_2, \ldots, U_n$ and the description of an (ℓ, r, t)-distribution $\mathcal{D}$, outputting a sample R.*

Similarly to a DS, any distributed universal sampler $\mathsf{DUS} = (\mathsf{Gen}, \mathsf{Sample})$ naturally corresponds to a one-round protocol Π_{DUS}, where each party first broadcasts a message output by Gen, and then, for every required sample, runs Sample on input all the parties' messages and the desired distribution $\mathcal{D}$.

As in the setting of universal samplers [HJK+16], we can classify a DUS in two main ways: security for distributions chosen *selectively* by the adversary ahead of time, and security for adversaries who can *adaptively* choose distributions on-the-fly. We refer to the first class as *one-time security*, and the latter as *reusable security*.

One-Time Distributed Universal Samplers While reusable distributed universal samplers need a random oracle independently of the power of the adversary, it is possible to build one-time DUSs with semi-malicious security in the plain model. Indeed, we now consider a one-time, selective security definition, where the sampler may only be queried once, and on a distribution $\mathcal{D}$ that is fixed ahead of time. We formalise the idea.

Definition 7.4 (Distributed Universal Sampler with One-Time Semi-Malicious Security). *A distributed universal sampler $(\mathsf{Gen}, \mathsf{Sample})$ satisfies* one-time semi-malicious security *if there exists a PPT simulator Sim such that, for every set of corrupted parties $C \subsetneq [n]$, corresponding randomness $(\rho_i)_{i \in C}$ and (ℓ, r, t)-distribution $\mathcal{D}$, the following two distributions are computationally indistinguishable.*

$$\left\{ \begin{pmatrix} (U_i)_{i\in[n]}, (\rho_i)_{i\in C} \\ R \end{pmatrix} \middle| \begin{array}{l} \forall i \in H: \quad \rho_i \xleftarrow{\$} \{0,1\}^{L(\lambda)} \\ \forall i \in [n]: \quad U_i \leftarrow \mathsf{Gen}\big(1\!\!1^\lambda, i; \rho_i\big) \\ R \leftarrow \mathsf{Sample}(U_1, U_2, \ldots, U_n, \mathcal{D}) \end{array} \right\}$$

$$\left\{ \begin{pmatrix} (U_i)_{i\in[n]}, (\rho_i)_{i\in C} \\ R \end{pmatrix} \middle| \begin{array}{l} R \xleftarrow{\$} \mathcal{D} \\ (U_i)_{i\in[n]} \xleftarrow{\$} \mathsf{Sim}\big(1\!\!1^\lambda, C, \mathcal{D}, R, (\rho_i)_{i\in C}\big) \end{array} \right\}$$

Just as with a DS, we can adapt the above definition to passive security by sampling the randomness of the corrupted parties inside the game in the real world and by generating it using the simulator in the ideal world. In a definition for active security, we must also account for a rushing adversary, who may adaptively choose the sampler shares of the corrupted parties after seeing those of the honest parties. In other words, in the ideal world, the adversary would be allowed to select the final output R from a list of samples generated by the functionality. We do not consider this notion here, since our actively secure construction actually satisfies the stronger notion of a *reusable* DUS (see Section 7.1).

Universal samplers. We will build our one-time DUSs starting from their non-distributed version [HJK+16]. The corresponding definition is available in Section 2.6. A one-time universal sampler is a pair of PPT algorithms (US.Setup, US.Sample) with the same syntax as the reusable case. The main difference is that the random oracle is no longer needed. Security is defined by stating that no PPT adversary can distinguish the real samplers from fake ones specifically programmed to output an ideal sample R when used in conjunction with a distribution $\mathcal{D}$ selected ahead of time. The property is required to hold for every (ℓ, r, t)-distribution $\mathcal{D}$. In other words, the main novelty is that we now program the sampler for one specific distribution selected ahead of time, whereas in the reusable case, we did that for multiple distributions adaptively chosen by the adversary.

Construction of a one-time DUS. Given distributed samplers and a one-time universal sampler, it is quite straightforward to build a semi-maliciously secure one-time DUS. We can simply substitute the trusted dealer generating the one-time sampler U with a semi-maliciously secure DS for US.Setup. Security follows from the programmability of one-time universal samplers and the fact that, by definition, a DS implements the functionality that directly samples from the underlying distribution. This idea is formalised in the following theorem. We will not, however, provide a formal proof.

Theorem 7.5 (One-Time Distributed Universal Samplers). *Suppose that* (US.Setup, US.Sample) *is a one-time universal sampler for* (ℓ, r, t)*-distributions. Assume that* (DS.Gen, DS.Sample) *is a semi-maliciously secure distributed sampler for* US.Setup*. Then, there exists a one-time distributed universal samplers for* (ℓ, r, t)*-distributions with semi-malicious security.*

Reusable Distributed Universal Samplers. In a reusable DUS, the shares output by Gen can be reused to obtain an arbitrary number of samples, from distributions that are chosen adaptively by the adversary. Note that, as is the case for (non-distributed) universal samplers [HJK+16], this notion is impossible to realize in the standard model, and our construction will use a random oracle.

We model security by requiring that the sampler can be used to obtain a one-round protocol that securely realizes an ideal sampling functionality, which can be queried adaptively.

Definition 7.6 (Reusable Distributed Universal Sampler). *A distributed universal sampler* $\mathsf{DUS} = (\mathsf{Gen}, \mathsf{Sample})$ *for* (ℓ, r, t)*-distributions satisfies* reusable active security *if the corresponding one-round protocol* Π_{DUS} *implements the functionality* $\mathcal{F}_{\mathsf{DUS}}$ *(see Fig. 21) against a static and active PPT adversary.*

$\mathcal{F}_{\mathsf{DUS}}$

Initialisation. On input Init from every honest party and the adversary, the functionality activates and enters the querying phase.

Querying phase.

- On input $(\mathsf{Query}, \mathsf{id}, \mathcal{D})$ from the adversary, where $\mathcal{D}$ is an (ℓ, r, t)-distribution and the pair $(\mathsf{id}, \mathcal{D})$ has not been queried previously, the functionality samples $R \xleftarrow{\$} \mathcal{D}$, sends R to the adversary and stores $(\mathsf{id}, \mathcal{D}, R)$.
- On input $(\mathsf{Choice}, \widehat{\mathsf{id}})$ from the adversary, the functionality stores $\widehat{\mathsf{id}}$, ends the querying phase and begins the sampling phase. If $\widehat{\mathsf{id}} = \mathsf{Abort}$, the functionality outputs $\perp$ to every honest party and halts.

Sampling phase. On input an (ℓ, r, t)-distribution $\mathcal{D}$ from party P_i, where a tuple $(\widehat{\mathsf{id}}, \mathcal{D}, R)$ has not been stored previously, the functionality samples $R \xleftarrow{\$} \mathcal{D}$, stores $(\widehat{\mathsf{id}}, \mathcal{D}, R)$ and outputs R to P_i. If $(\widehat{\mathsf{id}}, \mathcal{D}, R)$ has been previously stored (in either the querying or sampling phase), the functionality outputs R to P_i.

Fig. 21. The Reusable Distributed Universal Sampler Functionality for Active Security

Adaptively secure universal samplers. We construct a reusable DUS starting from an adaptively secure universal sampler. We briefly recall the corresponding definition [HJK+16], the formal version of which is available in Section 2.6. An adaptively secure universal sampler is a pair of PPT algorithms $(\mathsf{US.Setup}, \mathsf{US.Sample})$, the first one of which is used by a trusted dealer to generate a sampler U. By feeding U and (ℓ, r, t)-distributions $\mathcal{D}$ to the second algorithm (and using the random oracle), anyone can obtain a sample R. Importantly, $\mathsf{US.Sample}$ is deterministic, so a set of parties can use U to generate public samples non-interactively. Security requires that no PPT adversary $\mathcal{A}$ can distinguish the real sampler U and the original oracle responses from fake ones specifically programmed to output ideal samples from the (ℓ, r, t)-distributions chosen by $\mathcal{A}$ on-the-fly. Hofheinz *et al.* [HJK+16] present an adaptively secure universal sampler for (ℓ, r, t)-distributions whose size is $\mathsf{poly}(\ell, r, t)$.

Our reusable distributed universal sampler. It turns out that designing a reusable DUS is rather straightforward. It suffices to generate the adaptively secure sampler U using a DS for $\mathsf{US.Setup}$. Observe that if we use the construction of [HJK+16], the size of the sampler shares is $\mathsf{poly}(\ell, r, t)$. Security follows from the

adaptive programmability of US and the fact that the DS implements the functionality that directly samples from the associated distribution. We formalise this in the theorem below, again, given without proof.

Theorem 7.7 (Reusable Distributed Universal Samplers in the Random Oracle Model). *Suppose that* $(\mathsf{US.Setup}, \mathsf{US.Sample})$ *is an adaptively secure universal sampler for* (ℓ, r, t)*-distributions using a random oracle* $\mathcal{H}$*. Assume that there exists an actively secure distributed sampler for* $\mathsf{US.Setup}$*. Then, there exists a reusable distributed universal sampler for* (ℓ, r, t)*-distributions with active security.*

7.2 Building Ideal Public Key PCFs upon Distributed Universal Samplers

If reusable distributed universal samplers exist, constructing ideal public key PCFs becomes easy. We can use the reusable DUS to produce correlated samples $(R_j)_{j\in[n]}$ and then protect their privacy using a PKE scheme. Specifically, each party P_i can generate a PKE pair $(\mathsf{sk}_i, \mathsf{pk}_i)$ and broadcast the public counterpart along with a reusable DUS share. In order to produce and deal correlated outputs $(R_j)_{j\in[n]}$ from a correlation function $\mathcal{C}$, it is sufficient to query the DUS with the distribution that samples from $\mathcal{C}$ and outputs the encryption of R_j under pk_j for every $j \in [n]$. In this way, only the party j can retrieve the value of the j-th sample R_j. If we rely on the DUS built on top of the US of Hofheinz *et al.* [HJK+16], the key size of our public key PCF is $\mathsf{poly}(\ell, r, n \cdot t)$.

Security. Proving the security of this construction is an easy task. We rely on UC composability, and substitute the DUS with the corresponding functionality $\mathcal{F}_{\mathsf{DUS}}$ (see Fig. 21). Then, by the IND-CPA security of the PKE scheme, we can substitute the ciphertexts addressed to the honest parties in the $\mathcal{F}_{\mathsf{DUS}}$ responses with the encryption of random values. We substitute the ciphertexts of the corrupt parties with encryptions of the elements provided by the functionality $\mathcal{F}_{\mathcal{C}}^{\mathsf{Ideal}}$ (see Fig. 20). Our result is formalised in the following theorem; we do not provide a formal proof.

Theorem 7.8 (Ideal Public Key PCFs in the Random Oracle Model). *If there exists an IND-CPA PKE scheme and an* n*-party reusable distributed universal sampler with active security, there exists an actively secure ideal public key PCFs for* (n, ℓ, r, t)*-correlations.*

References

AJJM20. Prabhanjan Ananth, Abhishek Jain, Zhengzhong Jin, and Giulio Malavolta. Multi-key fully-homomorphic encryption in the plain model. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part I*, volume 12550 of *LNCS*, pages 28–57. Springer, Heidelberg, November 2020.

BCCT12. Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In Shafi Goldwasser, editor, *ITCS 2012*, pages 326–349. ACM, January 2012.

BCG+19a. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 291–308. ACM Press, November 2019.

BCG+19b. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 489–518. Springer, Heidelberg, August 2019.

BCG+20a. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Correlated pseudorandom functions from variable-density LPN. In *61st FOCS*, pages 1069–1080. IEEE Computer Society Press, November 2020.

BCG+20b. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-LPN. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 387–416. Springer, Heidelberg, August 2020.

BCGI18. Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 896–912. ACM Press, October 2018.

Bd94. Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In Tor Helleseth, editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 274–285. Springer, Heidelberg, May 1994.

BGG19. Sean Bowe, Ariel Gabizon, and Matthew D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-SNARK. In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors, *FC 2018 Workshops*, volume 10958 of *LNCS*, pages 64–77. Springer, Heidelberg, March 2019.

BGI+01. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 1–18. Springer, Heidelberg, August 2001.

BGI+14a. Amos Beimel, Ariel Gabizon, Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, and Anat Paskin-Cherniavsky. Non-interactive secure multiparty computation. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 387–404. Springer, Heidelberg, August 2014.

BGI14b. Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 501–519. Springer, Heidelberg, March 2014.

BGM17. Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-SNARK parameters in the random beacon model. Cryptol-

ogy ePrint Archive, Report 2017/1050, 2017. https://eprint.iacr.org/2017/1050.

BHR12. Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 134–153. Springer, Heidelberg, December 2012.

BP15. Nir Bitansky and Omer Paneth. ZAPs and non-interactive witness indistinguishability from indistinguishability obfuscation. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part II*, volume 9015 of *LNCS*, pages 401–427. Springer, Heidelberg, March 2015.

BW13. Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 280–300. Springer, Heidelberg, December 2013.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

CDI05. Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 342–362. Springer, Heidelberg, February 2005.

CLTV15. Ran Canetti, Huijia Lin, Stefano Tessaro, and Vinod Vaikuntanathan. Obfuscation of probabilistic circuits and applications. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part II*, volume 9015 of *LNCS*, pages 468–497. Springer, Heidelberg, March 2015.

DHRW16. Yevgeniy Dodis, Shai Halevi, Ron D. Rothblum, and Daniel Wichs. Spooky encryption and its applications. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 93–122. Springer, Heidelberg, August 2016.

FLS90. Uriel Feige, Dror Lapidot, and Adi Shamir. Multiple non-interactive zero knowledge proofs based on a single random string (extended abstract). In *31st FOCS*, pages 308–317. IEEE Computer Society Press, October 1990.

GGH+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 FOCS*, pages 40–49. IEEE Computer Society Press, October 2013.

GGM86. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 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 Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

GO07. Jens Groth and Rafail Ostrovsky. Cryptography in the multi-string model. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 323–341. Springer, Heidelberg, August 2007.

GOS06. Jens Groth, Rafail Ostrovsky, and Amit Sahai. Non-interactive zaps and new techniques for NIZK. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 97–111. Springer, Heidelberg, August 2006.

GPSZ17. Sanjam Garg, Omkant Pandey, Akshayaram Srinivasan, and Mark Zhandry. Breaking the sub-exponential barrier in obfustopia. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017,*

Part III, volume 10212 of *LNCS*, pages 156–181. Springer, Heidelberg, April / May 2017.

HIJ+17. Shai Halevi, Yuval Ishai, Abhishek Jain, Ilan Komargodski, Amit Sahai, and Eylon Yogev. Non-interactive multiparty computation without correlated randomness. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part III*, volume 10626 of *LNCS*, pages 181–211. Springer, Heidelberg, December 2017.

HJK+16. Dennis Hofheinz, Tibor Jager, Dakshita Khurana, Amit Sahai, Brent Waters, and Mark Zhandry. How to generate and use universal samplers. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 715–744. Springer, Heidelberg, December 2016.

HW15. Pavel Hubacek and Daniel Wichs. On the communication complexity of secure function evaluation with long output. In Tim Roughgarden, editor, *ITCS 2015*, pages 163–172. ACM, January 2015.

JLS21. Aayush Jain, Huijia Lin, and Amit Sahai. Indistinguishability obfuscation from well-founded assumptions. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2021, page 60–73, New York, NY, USA, 2021. Association for Computing Machinery.

KPTZ13. Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 669–684. ACM Press, November 2013.

KS08. Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Heidelberg, July 2008.

LZ17. Qipeng Liu and Mark Zhandry. Decomposable obfuscation: A framework for building applications of obfuscation from polynomial hardness. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 138–169. Springer, Heidelberg, November 2017.

OSY21. Claudio Orlandi, Peter Scholl, and Sophia Yakoubov. The rise of paillier: Homomorphic secret sharing and public-key silent OT. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 678–708. Springer, Heidelberg, October 2021.

PS19. Chris Peikert and Sina Shiehian. Noninteractive zero knowledge for NP from (plain) learning with errors. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 89–114. Springer, Heidelberg, August 2019.

A Proof of Theorem 5.2

Proof. The techniques used in this proof are inspired by [HJK+16]. We prove the security of the protocol Π_{NoRush} described in Fig. 13 by showing that it implements the functionality $\mathcal{F}_{\mathsf{NoRush}}$ (see Fig. 11) in the UC-model [Can01]. We achieve this plan through a series of hybrids allowing us to transition from the real protocol Π_{NoRush} (Hybrid 0) to the composition of $\mathcal{F}_{\mathsf{NoRush}}$ with a PPT

simulator (Hybrid 14). In all the stages, we assume that the keys k_h and K_h are uniformly sampled in $\{0,1\}^\lambda$ for every $h \in H$. Moreover, we assume, without loss of generality, that we deal with adversaries that always query the elements $(\mathsf{S}_i, \pi_i)_{i\in[n]}$ to the random oracle before broadcasting $(\mathsf{S}_i, \pi_i)_{i\in C}$ to the honest parties.

Hybrid 0. This is the initial stage, corresponding to the real world. The simulator generates the URS, the programs of the honest parties $(\mathsf{S}_h)_{h\in H}$, the corresponding NIZKs and the final outputs as per the protocol Π_{NoRush}. Moreover, it replies to the random oracle queries of the adversary sampling random strings. If any value is queried multiple times, the simulator takes care to answer always in the same way. Observe that with overwhelming probability, the final outputs are generated without using the trapdoor of the anti-rushing programs.

Hybrid 1. In this hybrid, we substitute the URS and the NIZKs proving the well-formedness of the programs of the honest parties with the outputs of the simulators $\mathsf{NIZK.Sim}_1$ and $\mathsf{NIZK.Sim}_2$. In this way, we remove any information concerning the keys of the honest parties from $(\pi_h)_{h\in H}$. Hybrid 1 is indistinguishable from Hybrid 0 due to the multi-theorem zero-knowledge of NIZK.

Formally speaking, the simulator generates the URS and the anti-rushing messages $(\mathsf{S}_h, \pi_h)_{h\in H}$ as follows (the red text indicates what changed since the last hybrid).

1. $\forall h \in H:$ $\quad \mathsf{S}_h \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}_{\mathsf{AR}}[\mathsf{SendMsg}, k_h, K_h, h])$
2. $(\mathsf{urs}, \tau) \xleftarrow{\$} \mathsf{NIZK.Sim}_1(1\!\!1^\lambda)$
3. $\forall h \in H:$ $\quad \pi_h \xleftarrow{\$} \mathsf{NIZK.Sim}_2\big(\mathsf{urs}, \tau, (\mathsf{S}_h, h)\big)$

Next, for every q from 1 to the number of random oracle queries issued by the adversary, we proceed from Hybrid 2.q to Hybrid 13.q.

Hybrid 2.q. In this hybrid, we schedule the q-th oracle response $\big(\hat{v}, (\hat{u}_i)_{i\in[n]}\big)$ before generating the programs of the honest parties. Furthermore, for every honest party h, we puncture the key K_h in $(\hat{u}_h, \hat{v})$ and we store it in S_h. We also program the latter to output the appropriate result when $(\hat{u}_h, \hat{v})$ is input. Observe that with overwhelming probability, such input does not activate the trapdoor in S_h. By the correctness of puncturing, the input-output behaviour of the honest parties' programs is the same as in the previous hybrid. Hence, indistinguishability holds by the security of iO.

The formal steps performed by the simulator in order to generate $(\mathsf{S}_h, \pi_h)_{h\in H}$ are described below.

1. $\hat{v} \xleftarrow{\$} \{0,1\}^\lambda$
2. $\forall i \in [n]:$ $\quad \hat{u}_i \xleftarrow{\$} \{0,1\}^{m\cdot\lambda}$
3. $\forall h \in H:$ $\quad \hat{K}_h \leftarrow \mathsf{Punct}\big(K_h, (\hat{u}_h, \hat{v})\big)$
4. $\forall h \in H:$ $\quad \hat{r}_h \leftarrow F_{K_h}(\hat{u}_h, \hat{v})$
5. $\forall h \in H:$ $\quad \hat{\mathsf{g}}_h \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, h; \hat{r}_h)$
6. $\forall h \in H:$ $\quad \mathsf{S}_h \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^1_{\mathsf{AR}}[\mathsf{SendMsg}, k_h, \hat{K}_h, h, \hat{u}_h, \hat{v}, \hat{\mathsf{g}}_h])$ (see Fig. 22)
7. $(\mathsf{urs}, \tau) \xleftarrow{\$} \mathsf{NIZK.Sim}_1(1\!\!1^\lambda)$
8. $\forall h \in H:$ $\quad \pi_h \xleftarrow{\$} \mathsf{NIZK.Sim}_2\big(\mathsf{urs}, \tau, (\mathsf{S}_h, h)\big)$

$\mathcal{P}^1_{\mathsf{AR}}[\mathsf{SendMsg}, k, K, i, \hat{u}, \hat{v}, \hat{\mathsf{g}}]$

Hard-coded. The algorithm $\mathsf{SendMsg}$, the PRF keys k and K and the index i of the party. Moreover, the scheduled oracle response $(\hat{u}, \hat{v})$ to the q-th query and the corresponding output $\hat{\mathsf{g}}$.
Input. Oracle responses $(u, v) \in \{0,1\}^{\lambda \cdot m(\lambda)} \times \{0,1\}^{\lambda}$.

1. If $(u, v) = (\hat{u}, \hat{v})$, output $\hat{\mathsf{g}}$
2. $(y_1^0, y_1^1, y_2^0, y_2^1, \ldots, y_m^0, y_m^1) \leftarrow F'_k(v)$
3. For every $j \in [m]$ set
$$x^j \leftarrow \begin{cases} 0 & \text{if } y_j^0 = u^j, \\ 1 & \text{if } y_j^1 = u^j, \\ \bot & \text{otherwise.} \end{cases}$$
4. If $x^j \neq \bot$ for every $j \in [m]$, output $(x^1, x^2, \ldots, x^m)$.
5. Set $r \leftarrow F_K(u, v)$.
6. Output $\mathsf{g} \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, i; r)$.

Fig. 22. The Anti-Rushing Program

Hybrid 3.q. In this hybrid, on behalf of every honest party h, the simulator generates the element $\hat{\mathsf{g}}_h$ hard-coded into S_h using true randomness $\hat{r}_h$ instead of the output of F. Moreover, if the anti-rushing messages $(\mathsf{armsg}_i)_{i \in C}$ of the corrupted parties are valid and correspond to the q-th oracle query, for every $h \in H$, the simulator directly outputs $\hat{r}_h$ to P_h, instead of using F_{K_h}. Observe that this hybrid is indistinguishable from the previous one due to the security of the puncturable PRF F.

The precise procedure used by the simulator to generate $(\mathsf{S}_h, \pi_h)_{h \in H}$ is the following.

1. $\hat{v} \xleftarrow{\$} \{0,1\}^\lambda$
2. $\forall i \in [n]: \quad \hat{u}_i \xleftarrow{\$} \{0,1\}^{m \cdot \lambda}$
3. $\forall h \in H: \quad \hat{K}_h \leftarrow \mathsf{Punct}\big(K_h, (\hat{u}_h, \hat{v})\big)$
4. $\forall h \in H: \quad \hat{r}_h \xleftarrow{\$} \{0,1\}^{L(\lambda)}$
5. $\forall h \in H: \quad \hat{\mathsf{g}}_h \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, h; \hat{r}_h)$
6. $\forall h \in H: \quad \mathsf{S}_h \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^1_{\mathsf{AR}}[\mathsf{SendMsg}, k_h, \hat{K}_h, h, \hat{u}_h, \hat{v}, \hat{\mathsf{g}}_h])$ (see Fig. 22)
7. $(\mathsf{urs}, \tau) \xleftarrow{\$} \mathsf{NIZK.Sim}_1(1\!\!1^\lambda)$
8. $\forall h \in H: \quad \pi_h \xleftarrow{\$} \mathsf{NIZK.Sim}_2\big(\mathsf{urs}, \tau, (\mathsf{S}_h, h)\big)$

Hybrid 4.q. For every honest party h, we now puncture the PRF key k_h in $\hat{v}$. Furthermore, we hard-code into S_h the output of $F'_{k_h}(\hat{v})$ and we use it to compute the result when $\hat{v}$ is input. Since the input-output behaviour of the program remains the same as in the previous hybrid, indistinguishability holds due to the security of iO.

The formal steps performed by the simulator for the generation of the the anti-rushing messages of the honest parties change as follows.

$\mathcal{P}^2_{\mathsf{AR}}[\mathsf{SendMsg}, k, K, i, \hat{u}, \hat{v}, \hat{\mathsf{g}}, (\hat{y}^b_j)_{j,b}]$

Hard-coded. The algorithm $\mathsf{SendMsg}$, the PRF keys k and K and the index i of the party. Moreover, the scheduled oracle response $(\hat{u}, \hat{v})$ to the q-th query and the corresponding output $\hat{\mathsf{g}}$. Finally, the PRF output $(\hat{y}^b_j)_{j,b}$.
Input. Oracle responses $(u, v) \in \{0,1\}^{\lambda \cdot m(\lambda)} \times \{0,1\}^\lambda$.

1. If $(u, v) = (\hat{u}, \hat{v})$, output $\hat{\mathsf{g}}$
2. If $v = \hat{v}$, for every $j \in [m]$, set

$$x^j \leftarrow \begin{cases} 0 & \text{if } \hat{y}^0_j = u^j, \\ 1 & \text{if } \hat{y}^1_j = u^j, \\ \bot & \text{otherwise.} \end{cases}$$

3. Otherwise, compute $(y^0_1, y^1_1, y^0_2, y^1_2, \ldots, y^0_m, y^1_m) \leftarrow F'_k(v)$ and, for every $j \in [m]$, set

$$x^j \leftarrow \begin{cases} 0 & \text{if } y^0_j = u^j, \\ 1 & \text{if } y^1_j = u^j, \\ \bot & \text{otherwise.} \end{cases}$$

4. If $x^j \neq \bot$ for every $j \in [m]$, output $(x^1, x^2, \ldots, x^m)$.
5. Set $r \leftarrow F_K(u, v)$.
6. Output $\mathsf{g} \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, i; r)$.

Fig. 23. The Anti-Rushing Program

1. $\hat{v} \xleftarrow{\$} \{0,1\}^\lambda$
2. $\forall i \in [n]: \quad \hat{u}_i \xleftarrow{\$} \{0,1\}^{m \cdot \lambda}$
3. $\forall h \in H: \quad \hat{K}_h \leftarrow \mathsf{Punct}\big(K_h, (\hat{u}_h, \hat{v})\big)$
4. $\forall h \in H: \quad \hat{k}_h \leftarrow \mathsf{Punct}'(k_h, \hat{v})$
5. $\forall h \in H: \quad (\hat{y}^b_{h,j})_{j,b} \leftarrow F'_{k_h}(\hat{v})$
6. $\forall h \in H: \quad \hat{r}_h \xleftarrow{\$} \{0,1\}^{L(\lambda)}$
7. $\forall h \in H: \quad \hat{\mathsf{g}}_h \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, h; \hat{r}_h)$
8. $\forall h \in H: \quad \mathsf{S}_h \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^2_{\mathsf{AR}}[\mathsf{SendMsg}, \hat{k}_h, \hat{K}_h, h, \hat{u}_h, \hat{v}, \hat{\mathsf{g}}_h, (\hat{y}^b_{h,j})_{j,b}])$ (see Fig. 23)
9. $(\mathsf{urs}, \tau) \xleftarrow{\$} \mathsf{NIZK.Sim}_1(1\!\!1^\lambda)$
10. $\forall h \in H: \quad \pi_h \xleftarrow{\$} \mathsf{NIZK.Sim}_2\big(\mathsf{urs}, \tau, (\mathsf{S}_h, h)\big)$

Hybrid 5.q. In this hybrid, on behalf of every honest party h, the simulator generates the values $(\hat{y}^b_{h,j})_{j,b}$ sampling them uniformly in $\{0,1\}^\lambda$ instead of using the PRF F'. This hybrid is indistinguishable from the previous one due to the security of the puncturable PRF F'.

The specific steps performed by the simulator for the generation of $(\mathsf{S}_h, \pi_h)_{h \in H}$ are the following.

1. $\hat{v} \xleftarrow{\$} \{0,1\}^\lambda$
2. $\forall i \in [n]: \quad \hat{u}_i \xleftarrow{\$} \{0,1\}^{m\cdot\lambda}$
3. $\forall h \in H: \quad \hat{K}_h \leftarrow \mathsf{Punct}\big(K_h, (\hat{u}_h, \hat{v})\big)$
4. $\forall h \in H: \quad \hat{k}_h \leftarrow \mathsf{Punct}'(k_h, \hat{v})$
5. $\forall h \in H, j \in [m]$ and $b \in \{0,1\}: \quad \hat{y}^b_{h,j} \xleftarrow{\$} \{0,1\}^\lambda$
6. $\forall h \in H: \quad \hat{r}_h \xleftarrow{\$} \{0,1\}^{L(\lambda)}$
7. $\forall h \in H: \quad \hat{\mathsf{g}}_h \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, h; \hat{r}_h)$
8. $\forall h \in H: \quad \mathsf{S}_h \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^2_{\mathsf{AR}}[\mathsf{SendMsg}, \hat{k}_h, \hat{K}_h, h, \hat{u}_h, \hat{v}, \hat{\mathsf{g}}_h, (\hat{y}^b_{h,j})_{j,b}])$ (see Fig. 23)
9. $(\mathsf{urs}, \tau) \xleftarrow{\$} \mathsf{NIZK.Sim}_1(1\!\!1^\lambda)$
10. $\forall h \in H: \quad \pi_h \xleftarrow{\$} \mathsf{NIZK.Sim}_2\big(\mathsf{urs}, \tau, (\mathsf{S}_h, h)\big)$

Hybrid 6.q. In this hybrid, we rely on an injective double-lengthening PRG $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$. Instead of hard-coding the values $(\hat{y}^b_{h,j})_{j,b}$ in S_h, the simulator now stores their images $(\hat{e}^b_{h,j})_{j,b}$ under the PRG G, i.e. $\hat{e}^b_{h,j} = G(\hat{y}^b_{h,j})$. Furthermore, when $\hat{v}$ is input in S_h, the program decodes now x^j by comparing $G(u^j)$ to $\hat{e}^0_{h,j}$ and $\hat{e}^1_{h,j}$. Observe that since G is injective, $u^j = \hat{y}^b_{h,j}$ if and only if $G(u^j) = \hat{e}^b_{h,j}$. In other words, the input-output behaviour of the programs of the honest parties did not change with respect to the previous hybrid. Therefore, indistinguishability holds by the security of iO.

The formal procedure performed by the simulator for the generation of $(\mathsf{S}_h, \pi_h)_{h\in H}$ is the following.

1. $\hat{v} \xleftarrow{\$} \{0,1\}^\lambda$
2. $\forall i \in [n]: \quad \hat{u}_i \xleftarrow{\$} \{0,1\}^{m\cdot\lambda}$
3. $\forall h \in H: \quad \hat{K}_h \leftarrow \mathsf{Punct}\big(K_h, (\hat{u}_h, \hat{v})\big)$
4. $\forall h \in H: \quad \hat{k}_h \leftarrow \mathsf{Punct}'(k_h, \hat{v})$
5. $\forall h \in H, j \in [m]$ and $b \in \{0,1\}: \quad \hat{y}^b_{h,j} \xleftarrow{\$} \{0,1\}^\lambda$
6. $\forall h \in H, j \in [m]$ and $b \in \{0,1\}: \quad \hat{e}^b_{h,j} \leftarrow G(\hat{y}^b_{h,j})$
7. $\forall h \in H: \quad \hat{r}_h \xleftarrow{\$} \{0,1\}^{L(\lambda)}$
8. $\forall h \in H: \quad \hat{\mathsf{g}}_h \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, h; \hat{r}_h)$
9. $\forall h \in H: \quad \mathsf{S}_h \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^3_{\mathsf{AR}}[\mathsf{SendMsg}, \hat{k}_h, \hat{K}_h, h, \hat{u}_h, \hat{v}, \hat{\mathsf{g}}_h, (\hat{e}^b_{h,j})_{j,b}])$ (see Fig. 24)
10. $(\mathsf{urs}, \tau) \xleftarrow{\$} \mathsf{NIZK.Sim}_1(1\!\!1^\lambda)$
11. $\forall h \in H: \quad \pi_h \xleftarrow{\$} \mathsf{NIZK.Sim}_2\big(\mathsf{urs}, \tau, (\mathsf{S}_h, h)\big)$

Hybrid 7.q. In this hybrid, instead of generating the values $(\hat{e}^b_{h,j})_{j,b}$ using the PRG G, the simulator samples them uniformly in $\{0,1\}^{2\lambda}$ for every $h \in H$. By the security of G, this hybrid is therefore indistinguishable from the previous one. Here, we are actually relying on the fact that, with overwhelming probability, the terms $(\hat{y}^b_{h,j})_{j,b}$ are not used for the generation the first $q-1$ oracle responses. This is a consequence of the fact that, with overwhelming probability, $\hat{v}$ is different from the nonces in the first $q-1$ oracle answers.

The procedure used by the simulator becomes now the following.

$\mathcal{P}^3_{\mathsf{AR}}[\mathsf{SendMsg}, k, K, i, \hat{u}, \hat{v}, \hat{\mathsf{g}}, (\hat{e}^b_j)_{j,b}]$

Hard-coded. The algorithm $\mathsf{SendMsg}$, PRF keys k and K and the index i of the party. Moreover, the scheduled oracle response $(\hat{u}, \hat{v})$ to the q-th query and the corresponding output $\hat{\mathsf{g}}$. Finally, the PRG outputs $(\hat{e}^b_j)_{j,b}$.
Input. Oracle responses $(u, v) \in \{0,1\}^{\lambda \cdot m(\lambda)} \times \{0,1\}^\lambda$.

1. If $(u, v) = (\hat{u}, \hat{v})$, output $\hat{\mathsf{g}}$
2. If $v = \hat{v}$, for every $j \in [m]$, set

$$x^j \leftarrow \begin{cases} 0 & \text{if } \hat{e}^0_j = G(u^j), \\ 1 & \text{if } \hat{e}^1_j = G(u^j), \\ \bot & \text{otherwise.} \end{cases}$$

3. Otherwise, compute $(y^0_1, y^1_1, y^0_2, y^1_2, \ldots, y^0_m, y^1_m) \leftarrow F'_k(v)$ and, for every $j \in [m]$, set

$$x^j \leftarrow \begin{cases} 0 & \text{if } y^0_j = u^j, \\ 1 & \text{if } y^1_j = u^j, \\ \bot & \text{otherwise.} \end{cases}$$

4. If $x^j \neq \bot$ for every $j \in [m]$, output $(x^1, x^2, \ldots, x^m)$.
5. Set $r \leftarrow F_K(u, v)$.
6. Output $\mathsf{g} \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, i; r)$.

Fig. 24. The Anti-Rushing Program

1. $\hat{v} \xleftarrow{\$} \{0,1\}^\lambda$
2. $\forall i \in [n]: \quad \hat{u}_i \xleftarrow{\$} \{0,1\}^{m \cdot \lambda}$
3. $\forall h \in H: \quad \hat{K}_h \leftarrow \mathsf{Punct}\big(K_h, (\hat{u}_h, \hat{v})\big)$
4. $\forall h \in H: \quad \hat{k}_h \leftarrow \mathsf{Punct}'(k_h, \hat{v})$
5. $\forall h \in H, j \in [m] \text{ and } b \in \{0,1\}: \quad \hat{e}^b_{h,j} \xleftarrow{\$} \{0,1\}^{2\lambda}$
6. $\forall h \in H: \quad \hat{r}_h \xleftarrow{\$} \{0,1\}^{L(\lambda)}$
7. $\forall h \in H: \quad \hat{\mathsf{g}}_h \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, h; \hat{r}_h)$
8. $\forall h \in H: \quad \mathsf{S}_h \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^3_{\mathsf{AR}}[\mathsf{SendMsg}, \hat{k}_h, \hat{K}_h, h, \hat{u}_h, \hat{v}, \hat{\mathsf{g}}_h, (\hat{e}^b_{h,j})_{j,b}])$ (see Fig. 24)
9. $(\mathsf{urs}, \tau) \xleftarrow{\$} \mathsf{NIZK.Sim}_1(1\!\!1^\lambda)$
10. $\forall h \in H: \quad \pi_h \xleftarrow{\$} \mathsf{NIZK.Sim}_2\big(\mathsf{urs}, \tau, (\mathsf{S}_h, h)\big)$

Hybrid 8.q. Starting from this hybrid, the program S_h generates the element $\hat{\mathsf{g}}_h$ using the trapdoor mechanism. Compared to the previous stage, the only thing that actually changes is how the values $(\hat{e}^b_{h,j})_{j,b}$ are generated. Specifically, the simulator first encodes the element $\hat{\mathsf{g}}_h$ as a bit string $\hat{x}_h$. Then, for every $j \in [m]$, it sets $\hat{e}^{\hat{x}^j_h}_{h,j}$ to $G(\hat{u}^j_h)$. The remaining value $\hat{e}^{1-\hat{x}^j_h}_{h,j}$ is instead sampled uniformly in $\{0,1\}^{2\lambda}$ as in the previous hybrid. Observe that in this way, even

if we remove the first line from $\mathcal{P}^3_{\mathsf{AR}}$, S_h keeps outputting $\hat{\mathsf{g}}_h$ when $(\hat{u}_h, \hat{v})$ is provided as input. We call the program obtained in this way $\mathcal{P}^4_{\mathsf{AR}}$.

It is possible to conclude that $\mathcal{P}^3_{\mathsf{AR}}$ and $\mathcal{P}^4_{\mathsf{AR}}$ have actually the same input-output behaviour. The claim is trivially verifiable when the input (u, v) coincides with the hard-coded pair $(\hat{u}, \hat{v})$ or $v \neq \hat{v}$. If instead $v = \hat{v}$ and $u \neq \hat{u}$, the matter is a little more complex.

Observe that the image of the PRG G has 2^λ elements and they are embedded into a space of cardinality $2^{2\lambda}$. Since the latter is much larger, with overwhelming probability, values uniformly sampled in $\{0,1\}^{2\lambda}$ do not belong to the image of G. In particular, this holds for those elements among $(\hat{e}^b_{h,j})_{j,b}$ that are sampled uniformly. For such values, there is no chance that $\hat{e}^b_{h,j} = G(\hat{u}^j_h)$. As a consequence, the output of $\mathcal{P}^3_{\mathsf{AR}}$ is never generated using the trapdoor when $v = \hat{v}$. In the case of $\mathcal{P}^4_{\mathsf{AR}}$ instead, the only input with $v = \hat{v}$ that activates the trapdoor is $(\hat{u}, \hat{v})$ due to the injectivity of G.

In this way, we have proven that Hybrid 8.q is indistinguishable from the previous one due to security of iO. The formal description of the steps performed by the simulator is available below.

1. $\hat{v} \xleftarrow{\$} \{0,1\}^\lambda$
2. $\forall i \in [n]:$ $\hat{u}_i \xleftarrow{\$} \{0,1\}^{m \cdot \lambda}$
3. $\forall h \in H:$ $\hat{K}_h \leftarrow \mathsf{Punct}\big(K_h, (\hat{u}_h, \hat{v})\big)$
4. $\forall h \in H:$ $\hat{k}_h \leftarrow \mathsf{Punct}'(k_h, \hat{v})$
5. $\forall h \in H:$ $\hat{r}_h \xleftarrow{\$} \{0,1\}^{L(\lambda)}$
6. $\forall h \in H:$ $\hat{\mathsf{g}}_h \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, h; \hat{r}_h)$
7. For every $h \in H$, rewrite $\hat{\mathsf{g}}_h$ as an m-bit string $\hat{x}_h$.
8. $\forall h \in H$ and $j \in [m]:$ $\hat{e}^{\hat{x}^j_h}_{h,j} \leftarrow G(\hat{u}^j_h)$
9. $\forall h \in H$ and $j \in [m]:$ $\hat{e}^{1-\hat{x}^j_h}_{h,j} \xleftarrow{\$} \{0,1\}^{2\lambda}$
10. $\forall h \in H:$ $\mathsf{S}_h \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^4_{\mathsf{AR}}[\mathsf{SendMsg}, \hat{k}_h, \hat{K}_h, h, \hat{v}, (\hat{e}^b_{h,j})_{j,b}])$ (see Fig. 25)
11. $(\mathsf{urs}, \tau) \xleftarrow{\$} \mathsf{NIZK.Sim}_1(1\!\!1^\lambda)$
12. $\forall h \in H:$ $\pi_h \xleftarrow{\$} \mathsf{NIZK.Sim}_2\big(\mathsf{urs}, \tau, (\mathsf{S}_h, h)\big)$

Hybrid 9.q. In this hybrid, the simulator changes the way it generates those values among $(\hat{e}^b_{h,j})_{j,b}$ that were previously sampled uniformly in $\{0,1\}^{2\lambda}$. For each of them, it indeed chooses a random λ-bit seed $\hat{y}^b_{h,j}$ and sets $\hat{e}^b_{h,j} \leftarrow G(\hat{y}^b_{h,j})$. This hybrid is therefore indistinguishable from the previous one by the PRG security of G. The procedure describing the steps of the simulator is now the following.

1. $\hat{v} \xleftarrow{\$} \{0,1\}^\lambda$
2. $\forall i \in [n]:$ $\hat{u}_i \xleftarrow{\$} \{0,1\}^{m \cdot \lambda}$
3. $\forall h \in H:$ $\hat{K}_h \leftarrow \mathsf{Punct}\big(K_h, (\hat{u}_h, \hat{v})\big)$
4. $\forall h \in H:$ $\hat{k}_h \leftarrow \mathsf{Punct}'(k_h, \hat{v})$
5. $\forall h \in H:$ $\hat{r}_h \xleftarrow{\$} \{0,1\}^{L(\lambda)}$

$\mathcal{P}^4_{\mathsf{AR}}[\mathsf{SendMsg}, k, K, i, \hat{v}, (\hat{e}^b_j)_{j,b}]$

Hard-coded. The algorithm $\mathsf{SendMsg}$, the PRF keys k and K and the index i of the party. Moreover, the scheduled nonce $\hat{v}$ for the q-th oracle query and the values $(\hat{e}^b_j)_{j,b}$.
Input. Oracle responses $(u, v) \in \{0,1\}^{\lambda \cdot m(\lambda)} \times \{0,1\}^\lambda$.

1. If $v = \hat{v}$, for every $j \in [m]$, set
$$x^j \leftarrow \begin{cases} 0 & \text{if } \hat{e}^0_j = G(u^j), \\ 1 & \text{if } \hat{e}^1_j = G(u^j), \\ \bot & \text{otherwise.} \end{cases}$$
2. Otherwise, compute $(y^0_1, y^1_1, y^0_2, y^1_2, \ldots, y^0_m, y^1_m) \leftarrow F'_k(v)$ and, for every $j \in [m]$, set
$$x^j \leftarrow \begin{cases} 0 & \text{if } y^0_j = u^j, \\ 1 & \text{if } y^1_j = u^j, \\ \bot & \text{otherwise.} \end{cases}$$
3. If $x^j \neq \bot$ for every $j \in [m]$, output $(x^1, x^2, \ldots, x^m)$.
4. Set $r \leftarrow F_K(u, v)$.
5. Output $\mathsf{g} \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, i; r)$.

Fig. 25. The Anti-Rushing Program

6. $\forall h \in H: \quad \hat{\mathsf{g}}_h \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, h; \hat{r}_h)$
7. For every $h \in H$, rewrite $\hat{\mathsf{g}}_h$ as an m-bit string $\hat{x}_h$.
8. $\forall h \in H$ and $j \in [m]: \quad \hat{y}^{\hat{x}^j_h}_{h,j} \leftarrow \hat{u}^j_h$
9. $\forall h \in H$ and $j \in [m]: \quad \hat{y}^{1-\hat{x}^j_h}_{h,j} \xleftarrow{\$} \{0,1\}^\lambda$
10. $\forall h \in H, j \in [m]$ and $b \in \{0,1\}: \quad \hat{e}^b_{h,j} \leftarrow G(\hat{y}^b_{h,j})$
11. $\forall h \in H: \quad \mathsf{S}_h \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^4_{\mathsf{AR}}[\mathsf{SendMsg}, \hat{k}_h, \hat{K}_h, h, \hat{v}, (\hat{e}^b_{h,j})_{j,b}])$ (see Fig. 25)
12. $(\mathsf{urs}, \tau) \xleftarrow{\$} \mathsf{NIZK.Sim}_1(1\!\!1^\lambda)$
13. $\forall h \in H: \quad \pi_h \xleftarrow{\$} \mathsf{NIZK.Sim}_2\big(\mathsf{urs}, \tau, (\mathsf{S}_h, h)\big)$

Hybrid 10.q. In this hybrid, instead of hard-coding the values $(\hat{e}^b_{h,j})_{j,b}$ in S_h, the simulator stores their preimages $(\hat{y}^b_{h,j})_{j,b}$ under the PRG G. Furthermore, when $\hat{v}$ is input into S_h, the program decodes x^j by comparing u^j to $\hat{y}^0_{h,j}$ and $\hat{y}^1_{h,j}$. Observe that since G is injective, $u^j = \hat{y}^b_{h,j}$ if and only if $G(u^j) = \hat{e}^b_{h,j}$. In other words, the input-output behaviour of the programs of the honest parties did not change with respect to the previous hybrid. Therefore, indistinguishability holds by the security of iO.

The formal procedure performed by the simulator for the generation of $(\mathsf{S}_h, \pi_h)_{h \in H}$ is the following.

1. $\hat{v} \xleftarrow{\$} \{0,1\}^\lambda$

$\mathcal{P}^5_{\mathsf{AR}}[\mathsf{SendMsg}, k, K, i, \hat{v}, (\hat{y}^b_j)_{j,b}]$

Hard-coded. The algorithm $\mathsf{SendMsg}$, the PRF keys k and K and the index i of the party. Moreover, the scheduled nonce $\hat{v}$ for the q-th oracle query and the values $(\hat{y}^b_j)_{j,b}$.
Input. Oracle responses $(u, v) \in \{0,1\}^{\lambda \cdot m(\lambda)} \times \{0,1\}^\lambda$.

1. If $v = \hat{v}$, for every $j \in [m]$, set
$$x^j \leftarrow \begin{cases} 0 & \text{if } \hat{y}^0_j = u^j, \\ 1 & \text{if } \hat{y}^1_j = u^j, \\ \bot & \text{otherwise.} \end{cases}$$
2. Otherwise, compute $(y^0_1, y^1_1, y^0_2, y^1_2, \ldots, y^0_m, y^1_m) \leftarrow F'_k(v)$ and, for every $j \in [m]$, set
$$x^j \leftarrow \begin{cases} 0 & \text{if } y^0_j = u^j, \\ 1 & \text{if } y^1_j = u^j, \\ \bot & \text{otherwise.} \end{cases}$$
3. If $x^j \neq \bot$ for every $j \in [m]$, output $(x^1, x^2, \ldots, x^m)$.
4. Set $r \leftarrow F_K(u, v)$.
5. Output $\mathsf{g} \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, i; r)$.

Fig. 26. The Anti-Rushing Program

2. $\forall i \in [n]: \quad \hat{u}_i \xleftarrow{\$} \{0,1\}^{m \cdot \lambda}$
3. $\forall h \in H: \quad \hat{K}_h \leftarrow \mathsf{Punct}\big(K_h, (\hat{u}_h, \hat{v})\big)$
4. $\forall h \in H: \quad \hat{k}_h \leftarrow \mathsf{Punct}'(k_h, \hat{v})$
5. $\forall h \in H: \quad \hat{r}_h \xleftarrow{\$} \{0,1\}^{L(\lambda)}$
6. $\forall h \in H: \quad \hat{\mathsf{g}}_h \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, h; \hat{r}_h)$
7. For every $h \in H$, rewrite $\hat{\mathsf{g}}_h$ as an m-bit string $\hat{x}_h$.
8. $\forall h \in H$ and $j \in [m]: \quad \hat{y}^{\hat{x}^j_h}_{h,j} \leftarrow \hat{u}^j_h$
9. $\forall h \in H$ and $j \in [m]: \quad \hat{y}^{1-\hat{x}^j_h}_{h,j} \xleftarrow{\$} \{0,1\}^\lambda$
10. $\forall h \in H: \quad \mathsf{S}_h \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^5_{\mathsf{AR}}[\mathsf{SendMsg}, \hat{k}_h, \hat{K}_h, h, \hat{v}, (\hat{y}^b_{h,j})_{j,b}])$ (see Fig. 26)
11. $(\mathsf{urs}, \tau) \xleftarrow{\$} \mathsf{NIZK.Sim}_1(1\!\!1^\lambda)$
12. $\forall h \in H: \quad \pi_h \xleftarrow{\$} \mathsf{NIZK.Sim}_2\big(\mathsf{urs}, \tau, (\mathsf{S}_h, h)\big)$

Hybrid 11.q. In this hybrid, we finally change the oracle response to the q-th query of the adversary, substituting the random values $(\hat{u}_h)_{h \in H}$ with the encryption of the elements $(\hat{\mathsf{g}}_h)_{h \in H}$.

Actually, the only thing that changes is the distribution of the terms $(\hat{y}^b_{h,j})_{j,b}$. Indeed, they are not uniform in $\{0,1\}^\lambda$ anymore, but they are generated by the simulator as $F'_{k_h}(\hat{v})$. Since $\hat{y}^{\hat{x}^j_h}_{h,j} = \hat{u}^j_h$ for every $j \in [m]$ and $h \in H$, we also modify the values $(\hat{u}_h)_{h \in H}$ accordingly.

Observe that this hybrid is indistinguishable from the previous one by the security of the puncturable PRF F'. The formal procedure used by the simulator for the generation of $(\mathsf{S}_h, \pi_h)_{h \in H}$ and the q-th oracle response becomes the following.

1. $\hat{v} \xleftarrow{\$} \{0,1\}^\lambda$
2. $\forall i \in C: \quad \hat{u}_i \xleftarrow{\$} \{0,1\}^{m \cdot \lambda}$
3. $\forall h \in H: \quad \hat{k}_h \leftarrow \mathsf{Punct}'(k_h, \hat{v})$
4. $\forall h \in H: \quad \hat{r}_h \xleftarrow{\$} \{0,1\}^{L(\lambda)}$
5. $\forall h \in H: \quad \hat{\mathsf{g}}_h \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, h; \hat{r}_h)$
6. For every $h \in H$, rewrite $\hat{\mathsf{g}}_h$ as an m-bit string $\hat{x}_h$.
7. $\forall h \in H: \quad (\hat{y}^b_{h,j})_{j,b} \leftarrow F'_{k_h}(\hat{v})$
8. $\forall h \in H$ and $j \in [m]: \quad \hat{u}^j_h \leftarrow \hat{y}^{\hat{x}^j_h}_{h,j}$
9. $\forall h \in H: \quad \hat{K}_h \leftarrow \mathsf{Punct}\big(K_h, (\hat{u}_h, \hat{v})\big)$
10. $\forall h \in H: \quad \mathsf{S}_h \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^5_{\mathsf{AR}}[\mathsf{SendMsg}, \hat{k}_h, \hat{K}_h, h, \hat{v}, (\hat{y}^b_{h,j})_{j,b}])$ (see Fig. 26)
11. $(\mathsf{urs}, \tau) \xleftarrow{\$} \mathsf{NIZK.Sim}_1(1\!\!1^\lambda)$
12. $\forall h \in H: \quad \pi_h \xleftarrow{\$} \mathsf{NIZK.Sim}_2\big(\mathsf{urs}, \tau, (\mathsf{S}_h, h)\big)$

Hybrid 12.q. In this hybrid, the simulator generates the programs of the honest parties by obfuscating $\mathcal{P}_{\mathsf{AR}}$, as in the original protocol. We however keep replying to the q-th oracle query of the adversary as in Hybrid 11.q. Indistinguishability from the previous stage is guaranteed by the security of iO. As a matter of fact, the programs $\mathcal{P}^5_{\mathsf{AR}}$ (as in Hybrid 11.q) and $\mathcal{P}_{\mathsf{AR}}$ have the same input-output behaviour.

Formally, the anti-rushing messages $(\mathsf{S}_h, \pi_h)_{h \in H}$ are generated as follows.

1. $\forall h \in H: \quad \mathsf{S}_h \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}_{\mathsf{AR}}[\mathsf{SendMsg}, k_h, K_h, h])$ (see Fig. 12)
2. $(\mathsf{urs}, \tau) \xleftarrow{\$} \mathsf{NIZK.Sim}_1(1\!\!1^\lambda)$
3. $\forall h \in H: \quad \pi_h \xleftarrow{\$} \mathsf{NIZK.Sim}_2\big(\mathsf{urs}, \tau, (\mathsf{S}_h, h)\big)$

The procedure for the generation of the q-th oracle response is instead the following.

1. $\hat{v} \xleftarrow{\$} \{0,1\}^\lambda$
2. $\forall i \in C: \quad \hat{u}_i \xleftarrow{\$} \{0,1\}^{m \cdot \lambda}$
3. $\forall h \in H: \quad \hat{r}_h \xleftarrow{\$} \{0,1\}^{L(\lambda)}$
4. $\forall h \in H: \quad \hat{\mathsf{g}}_h \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, h; \hat{r}_h)$
5. For every $h \in H$, rewrite $\hat{\mathsf{g}}_h$ as an m-bit string $\hat{x}_h$.
6. $\forall h \in H: \quad (\hat{y}^b_{h,j})_{j,b} \leftarrow F'_{k_h}(\hat{v})$
7. $\forall h \in H$ and $j \in [m]: \quad \hat{u}^j_h \leftarrow \hat{y}^{\hat{x}^j_h}_{h,j}$

Observe that the URS and the anti-rushing messages $(\mathsf{armsg}_h)_{h \in H}$ of the honest parties can be now generated independently of the q-th oracle response. It is therefore possible to produce the q-th oracle answer on the fly, after receiving the corresponding query.

Hybrid 13.q. In this hybrid, the simulator generates the elements $(\hat{\mathsf{g}}_h)_{h\in H}$ hidden in the q-th oracle response using the functionality $\mathcal{F}_{\mathsf{NoRush}}$. The operation is actually performed only if the q-th oracle query consists of n elements $(\mathsf{S}_i, \pi_i)_{i\in[n]}$ where the values $(\mathsf{S}_h, \pi_h)_{h\in H}$ coincide with the anti-rushing messages of the honest parties in the only round of interactions and, for every $i \in C$,

$$\mathsf{NIZK.Verify}\big(\mathsf{urs}, \pi_i, (\mathsf{S}_i, i)\big) = 1.$$

In such cases, the simulator extracts the witnesses from the NIZK proofs of the corrupted parties using $\mathsf{NIZK.Extract}$, obtaining the corresponding PRF keys k_i and K_i. By the extractability of NIZK, the procedure is successful with overwhelming probability. If the q-th oracle query does not satisfy the properties described above, the response is generated as in the previous stage.

More precisely, the simulator now samples the nonce $\hat{v}$ and the terms $(\hat{u}_i)_{i\in C}$ uniformly and retrieves the randomness $\hat{r}_i$ used for the generation of $\hat{\mathsf{g}}_i = \mathsf{S}_i(\hat{u}_i, \hat{v})$ for every $i \in C$. Now, it is indeed possible to perform the operation as the PRF key K_i is no longer secret. Moreover, with overwhelming probability, the pair $(\hat{u}_i, \hat{v})$ does not activate the trapdoor in S_i. The values $(\hat{\mathsf{g}}_h)_{h\in H}$ are finally generated by sending $(\hat{\mathsf{g}}_i, \hat{r}_i)_{i\in C}$ to $\mathcal{F}_{\mathsf{NoRush}}$.

At the end, if the anti-rushing messages of the corrupted parties correspond to the q-th oracle query, the simulator retrieves the randomness of the honest parties $(\hat{r}_h)_{h\in H}$ by sending the label of the q-query to the functionality.

Observe that this hybrid is indistinguishable from the previous one by the witness extractability of NIZK. The formal steps used for the generation of the q-th oracle query become now the following.

1. $\hat{v} \xleftarrow{\$} \{0,1\}^\lambda$
2. $\forall i \in C: \quad \hat{u}_i \xleftarrow{\$} \{0,1\}^{m\cdot\lambda}$
3. $\forall i \in C: \quad (k_i, K_i, w_i) \leftarrow \mathsf{NIZK.Extract}\big(\mathsf{urs}, \tau, (\mathsf{S}_i, i), \pi_i\big)$
4. $\forall i \in C: \quad \hat{r}_i \leftarrow F_{K_i}(\hat{u}_i, \hat{v})$
5. $\forall i \in C: \quad \hat{\mathsf{g}}_i \leftarrow \mathsf{SendMsg}(1\!\!1^\lambda, i; \hat{r}_i)$
6. Send $\big(\mathsf{Query}, (\hat{\mathsf{g}}_i, \hat{r}_i)_{i\in C}\big)$ to $\mathcal{F}_{\mathsf{NoRush}}$, obtaining $\big(\mathsf{id}, (\hat{\mathsf{g}}_h)_{h\in H}\big)$ as a reply.
7. For every $h \in H$, rewrite $\hat{\mathsf{g}}_h$ as an m-bit string $\hat{x}_h$.
8. $\forall h \in H: \quad (\hat{y}^b_{h,j})_{j,b} \leftarrow F'_{k_h}(\hat{v})$
9. $\forall h \in H$ and $j \in [m]: \quad \hat{u}^j_h \leftarrow \hat{y}^{\hat{x}^j_h}_{h,j}$

Hybrid 14. This stage corresponds to the ideal world and it just formalises what has been achieved through the series of hybrids we described above.

At this point, the simulator generates the anti-rushing messages of the honest parties and the URS as in Hybrid 1. Moreover, it replies to all the oracle queries as in Hybrid 13.q, receiving ideal $(\mathsf{g}_h)_{h\in H}$ from the functionality $\mathcal{F}_{\mathsf{NoRush}}$. We recall that each of these samples is associated with a label id, so there is a one-to-one correspondence between labels and oracle queries. When the adversary selects the anti-rushing messages $(\mathsf{S}_i, \pi_i)_{i\in C}$ of the corrupted parties, the simulator sends the label of the corresponding oracle query to the functionality. The latter will take care of outputting the associated randomness to the honest parties. □

B Proof of Theorem 6.7

Proof. We show that the public key PCF with trusted setup described in Fig. 19 is semi-maliciously secure. We prove both correctness and security in one go. As a matter of fact, correctness can be regarded as the special case of security in which $C = \emptyset$. Observe that in such case, we can always assume that $\mathsf{RSample}$ samples directly from $\mathcal{C}$ using randomly chosen master secrets $(\mathsf{mk}'_i)_{i\in[n]}$.

We proceed by a sequence of 12 indistinguishable hybrids (some of them repeated for every possible nonce value) going from the real world (Hybrid 0) to the ideal world (Hybrid 12). The size of the nonce space will affect the proof only on the number of reductions needed. Specifically, the number of hybrids will be polynomial if and only if the cardinality of the nonce space is polynomial. In the other cases, we will need to assume the existence of sub-exponentially secure primitives.

We always assume that the PRF keys k and K hard-coded into the correlation generation program CGP are randomly sampled in $\{0,1\}^\lambda$. Moreover, in every stage, we assume that the PKE pairs of the parties are all generated according to the protocol, using the randomness parametrising the game in the case of the corrupted players. We denote the i-th pair by $(\hat{\mathsf{sk}}_i, \hat{\mathsf{pk}}_i)$.

Hybrid 0. This is the initial hybrid and corresponds to the execution of the security game with $b = 0$. The challenger creates the correlation generation program by obfuscating $\mathcal{P}_{\mathsf{CG}}$ (see Fig. 19). Also the keys of the honest parties are generated following the protocol. Those of the corrupted players are instead derived using the randomness parametrising the game. Finally, the challenger replies to all the sampling queries using Eval.

Hybrid 1. In this hybrid, we puncture the PRF key k in the list of public keys of the players. We also store, in the correlation generation program CGP, the master secrets corresponding to the punctured position and we use them to compute the output when the public keys of the parties are fed into it. In this way, the input-output behaviour of CGP does not change with respect to the previous hybrid. Therefore, indistinguishability follows from the security of obfuscation.

The formal steps performed by the challenger for the generation of CGP are now the following (the red text highlights what changed since the last stage).

1. $\hat{k} \leftarrow \mathsf{Punct}'\big(k, (\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n)\big)$
2. $(\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_n) \leftarrow F'_k(\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n)$
3. $\forall i \in [n]: \quad \hat{\mathsf{mk}}_i \leftarrow \mathsf{Secret}(1\!\!1^\lambda, i; \hat{s}_i)$
4. $\mathsf{CGP} \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^1_{\mathsf{CG}}[\hat{k}, K, (\hat{\mathsf{pk}}_i, \hat{\mathsf{mk}}_i)_{i\in[n]}])$ (see Fig. 27)

Hybrid 2. In this hybrid, we change how we produce the master secrets $(\hat{\mathsf{mk}}_i)_{i\in[n]}$. Specifically, instead of generating the randomness of Secret by means of F'_k, we sample it uniformly. By the security of the puncturable PRF, this hybrid is indistinguishable from the previous one.

Formally, the challenger generates CGP as follows.

1. $\hat{k} \leftarrow \mathsf{Punct}'\big(k, (\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n)\big)$

$\mathcal{P}^1_{\mathsf{CG}}[k, K, (\hat{\mathsf{pk}}_i, \hat{\mathsf{mk}}_i)_{i\in[n]}]$

Hard-coded. The puncturable PRF keys k and K, the public keys $(\hat{\mathsf{pk}}_i)_{i\in[n]}$ and the master secrets $(\hat{\mathsf{mk}}_i)_{i\in[n]}$.
Input. A nonce $x \in \{0,1\}^{l(\lambda)}$ and n public keys $\mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_n$.

1. $(r, r_1, r_2, \ldots, r_n) \leftarrow F_K(\mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_n, x)$.
2. If $\mathsf{pk}_i = \hat{\mathsf{pk}}_i$ for every $i \in [n]$, set $\mathsf{mk}_i \leftarrow \hat{\mathsf{mk}}_i$ for every $i \in [n]$.
3. Otherwise, perform the following operations
   (a) $(s_1, s_2, \ldots, s_n) \leftarrow F'_k(\mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_n)$
   (b) $\forall i \in [n]: \quad \mathsf{mk}_i \leftarrow \mathsf{Secret}(1\!\!1^\lambda, i; s_i)$
4. $(R_1, R_2, \ldots, R_n) \leftarrow \mathcal{C}(1\!\!1^\lambda, \mathsf{mk}_1, \mathsf{mk}_2, \ldots, \mathsf{mk}_n; r)$
5. $\forall i \in [n]: \quad c_i \leftarrow \mathsf{PKE.Enc}\big(\mathsf{pk}_i, (R_i, \mathsf{mk}_i); r_i\big)$
6. Output $c_1, c_2, \ldots, c_n$.

Fig. 27. The Correlation Generation Program

2. $\forall i \in [n]: \quad \hat{\mathsf{mk}}_i \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, i)$
3. $\mathsf{CGP} \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^1_{\mathsf{CG}}[\hat{k}, K, (\hat{\mathsf{pk}}_i, \hat{\mathsf{mk}}_i)_{i\in[n]}])$ (see Fig. 27)

We now consider the nonce space $\{0,1\}^{l(\lambda)}$ and we order it using the lexicographical order $<_{\text{lex}}$. Let ϵ denote the minimum and Ω the maximum. We apply the series of hybrids from 3 to 10 for every nonce $\hat{x}$, starting from ϵ and following the lexicographical order.

Hybrid 3.$\hat{x}$. In this hybrid, the challenger samples additional master secrets $(\mathsf{mk}'_h)_{h\in H}$ for the honest parties and hard-codes them into CGP along with $\hat{x}$ and H. When the nonce x input in CGP is strictly smaller than $\hat{x}$ and the provided public keys coincide with the ones of the parties, the program generates the samples $(R_i)_{i\in[n]}$ substituting $\hat{\mathsf{mk}}_h$ with mk'_h for every $h \in H$. Furthermore, when the nonce x is strictly smaller than $\hat{x}$, the challenger replies to the correlation queries using $\mathsf{RSample}$, providing it with $(\hat{\mathsf{mk}}_i)_{i\in[n]}$.

Observe that for $\hat{x} = \epsilon$, the input-output behaviour of CGP has not changed with respect to Hybrid 2. Moreover, the challenger never uses $\mathsf{RSample}$ to reply to the correlation queries. If instead $\hat{x} \neq \epsilon$, we will see that the input-output behaviour of CGP has not changed with respect to the previous hybrid either and the challenger replies to the correlation queries as it did before. We conclude that indistinguishability holds in both cases due to the security of the obfuscator.

The formal steps used by the challenger for the generation of CGP are now the following.

1. $\hat{k} \leftarrow \mathsf{Punct}'\big(k, (\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n)\big)$
2. $\forall i \in [n]: \quad \hat{\mathsf{mk}}_i \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, i)$
3. $\forall h \in H: \quad \mathsf{mk}'_h \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, h)$
4. $\mathsf{CGP} \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^{\hat{x},2}_{\mathsf{CG}}[\hat{k}, K, (\hat{\mathsf{pk}}_i, \hat{\mathsf{mk}}_i)_{i\in[n]}, (\mathsf{mk}'_h)_{h\in H}, H, \hat{x}])$ (see Fig. 28)

$\mathcal{P}^{\hat{x},2}_{\mathsf{CG}}[k, K, (\hat{\mathsf{pk}}_i, \hat{\mathsf{mk}}_i)_{i\in[n]}, (\mathsf{mk}'_h)_{h\in H}, H, \hat{x}]$

Hard-coded. The puncturable PRF keys k and K, the public keys $(\hat{\mathsf{pk}}_i)_{i\in[n]}$ and the master secrets $(\hat{\mathsf{mk}}_i)_{i\in[n]}$ and $(\mathsf{mk}'_i)_{i\in H}$, the set of honest parties H and the nonce $\hat{x}$.
Input. A nonce $x \in \{0,1\}^{l(\lambda)}$ and n public keys $\mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_n$.

1. $(r, r_1, r_2, \ldots, r_n) \leftarrow F_K(\mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_n, x)$.
2. If $\mathsf{pk}_i = \hat{\mathsf{pk}}_i$ for every $i \in [n]$ and $x \geq_{\text{lex}} \hat{x}$, set $\mathsf{mk}_i \leftarrow \hat{\mathsf{mk}}_i$ for every $i \in [n]$.
3. If $\mathsf{pk}_i = \hat{\mathsf{pk}}_i$ for every $i \in [n]$ and $x <_{\text{lex}} \hat{x}$, set $\mathsf{mk}_h \leftarrow \mathsf{mk}'_h$ for every $h \in H$ and $\mathsf{mk}_i \leftarrow \hat{\mathsf{mk}}_i$ for every $i \in C$.
4. Otherwise, perform the following operations
   (a) $(s_1, s_2, \ldots, s_n) \leftarrow F'_k(\mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_n)$
   (b) $\forall i \in [n]: \quad \mathsf{mk}_i \leftarrow \mathsf{Secret}(1\!\!1^\lambda, i; s_i)$
5. $(R_1, R_2, \ldots, R_n) \leftarrow \mathcal{C}(1\!\!1^\lambda, \mathsf{mk}_1, \mathsf{mk}_2, \ldots, \mathsf{mk}_n; r)$
6. $\forall i \in [n]: \quad c_i \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}_i, (R_i, \mathsf{mk}_i); r_i)$
7. Output $c_1, c_2, \ldots, c_n$.

Fig. 28. The Correlation Generation Program

The reply to $(\mathsf{Correlation}, x)$ when $x <_{\text{lex}} \hat{x}$ is instead computed as follows.

1. $(c_i)_{i\in[n]} \leftarrow \mathsf{CGP}(\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n, x)$
2. $\forall i \in C: \quad (R_i, \mathsf{mk}_i) \leftarrow \mathsf{PKE.Dec}(\hat{\mathsf{sk}}_i, c_i)$
3. $(R_h)_{h\in H} \xleftarrow{\$} \mathsf{RSample}(1\!\!1^\lambda, C, (R_i)_{i\in C}, (\hat{\mathsf{mk}}_i)_{i\in C}, (\hat{\mathsf{mk}}_h)_{h\in H})$

Hybrid 4.$\hat{x}$. In this hybrid, we puncture the PRF key K in the tuple consisting of the public keys of the parties and the nonce $\hat{x}$, i.e. $(\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n, \hat{x})$. Moreover, we program CGP to output the appropriate ciphertexts when the punctured position is given as input. Since the input-output behaviour of the program has not changed with respect to the previous hybrid, indistinguishability follows from the security of iO.

The formal procedure used by the challenger for the generation of CGP is now the following.

1. $\hat{k} \leftarrow \mathsf{Punct}'(k, (\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n))$
2. $\forall i \in [n]: \quad \hat{\mathsf{mk}}_i \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, i)$
3. $\forall h \in H: \quad \mathsf{mk}'_h \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, h)$
4. $\hat{K} \leftarrow \mathsf{Punct}(K, (\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n, \hat{x}))$
5. $(\hat{r}, \hat{r}_1, \hat{r}_2, \ldots, \hat{r}_n) \leftarrow F_K(\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n, \hat{x})$
6. $(\hat{R}_i)_{i\in[n]} \leftarrow \mathcal{C}(1\!\!1^\lambda, \hat{\mathsf{mk}}_1, \hat{\mathsf{mk}}_2, \ldots, \hat{\mathsf{mk}}_n; \hat{r})$
7. $\forall i \in [n]: \quad \hat{c}_i \leftarrow \mathsf{PKE.Enc}(\hat{\mathsf{pk}}_i, (\hat{R}_i, \hat{\mathsf{mk}}_i); \hat{r}_i)$
8. $\mathsf{CGP} \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^{\hat{x},3}_{\mathsf{CG}}[\hat{k}, \hat{K}, (\hat{\mathsf{pk}}_i, \hat{\mathsf{mk}}_i)_{i\in[n]}, (\mathsf{mk}'_h)_{h\in H}, H, \hat{x}, (\hat{c}_i)_{i\in[n]}])$ (see Fig. 29)

Hybrid 5.$\hat{x}$. In this hybrid, we change how we generate the samples $(\hat{R}_i)_{i\in[n]}$ and encrypt them. Specifically, instead of producing the randomness using the

$\mathcal{P}^{\hat{x},3}_{\mathsf{CG}}[k, K, (\hat{\mathsf{pk}}_i, \hat{\mathsf{mk}}_i)_{i\in[n]}, (\mathsf{mk}'_h)_{h\in H}, H, \hat{x}, (\hat{c}_i)_{i\in[n]}]$

Hard-coded. The puncturable PRF keys k and K, the public keys $(\hat{\mathsf{pk}}_i)_{i\in[n]}$ and the master secrets $(\hat{\mathsf{mk}}_i)_{i\in[n]}$ and $(\mathsf{mk}'_i)_{i\in H}$, the set of honest parties H, the nonce $\hat{x}$ and the ciphertexts $(\hat{c}_i)_{i\in[n]}$.
Input. A nonce $x \in \{0,1\}^{l(\lambda)}$ and n public keys $\mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_n$.

1. If $\mathsf{pk}_i = \hat{\mathsf{pk}}_i$ for every $i \in [n]$ and $x = \hat{x}$, output $(\hat{c}_i)_{i\in[n]}$.
2. $(r, r_1, r_2, \ldots, r_n) \leftarrow F_K(\mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_n, x)$.
3. If $\mathsf{pk}_i = \hat{\mathsf{pk}}_i$ for every $i \in [n]$ and $x \geq_{\text{lex}} \hat{x}$, set $\mathsf{mk}_i \leftarrow \hat{\mathsf{mk}}_i$ for every $i \in [n]$.
4. If $\mathsf{pk}_i = \hat{\mathsf{pk}}_i$ for every $i \in [n]$ and $x <_{\text{lex}} \hat{x}$, set $\mathsf{mk}_h \leftarrow \mathsf{mk}'_h$ for every $h \in H$ and $\mathsf{mk}_i \leftarrow \hat{\mathsf{mk}}_i$ for every $i \in C$.
5. Otherwise, perform the following operations
   (a) $(s_1, s_2, \ldots, s_n) \leftarrow F'_k(\mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_n)$
   (b) $\forall i \in [n]: \quad \mathsf{mk}_i \leftarrow \mathsf{Secret}(1^\lambda, i;\, s_i)$
6. $(R_1, R_2, \ldots, R_n) \leftarrow \mathcal{C}(1^\lambda, \mathsf{mk}_1, \mathsf{mk}_2, \ldots, \mathsf{mk}_n;\, r)$
7. $\forall i \in [n]: \quad c_i \leftarrow \mathsf{PKE.Enc}\big(\mathsf{pk}_i, (R_i, \mathsf{mk}_i);\, r_i\big)$
8. Output $c_1, c_2, \ldots, c_n$.

Fig. 29. The Correlation Generation Program

PRF F_K, we sample it uniformly. Observe that this hybrid and the previous one are indistinguishable by the security of the puncturable PRF F.

The formal procedure used by the challenger to generate CGP becomes the following.

1. $\hat{k} \leftarrow \mathsf{Punct}'\big(k, (\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n)\big)$
2. $\forall i \in [n]: \quad \hat{\mathsf{mk}}_i \xleftarrow{\$} \mathsf{Secret}(1^\lambda, i)$
3. $\forall h \in H: \quad \mathsf{mk}'_h \xleftarrow{\$} \mathsf{Secret}(1^\lambda, h)$
4. $\hat{K} \leftarrow \mathsf{Punct}\big(K, (\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n, \hat{x})\big)$
5. $(\hat{R}_i)_{i\in[n]} \xleftarrow{\$} \mathcal{C}(1^\lambda, \hat{\mathsf{mk}}_1, \hat{\mathsf{mk}}_2, \ldots, \hat{\mathsf{mk}}_n)$
6. $\forall i \in [n]: \quad \hat{c}_i \xleftarrow{\$} \mathsf{PKE.Enc}\big(\hat{\mathsf{pk}}_i, (\hat{R}_i, \hat{\mathsf{mk}}_i)\big)$
7. $\mathsf{CGP} \xleftarrow{\$} \mathsf{iO}(1^\lambda, \mathcal{P}^{\hat{x},3}_{\mathsf{CG}}[\hat{k}, \hat{K}, (\hat{\mathsf{pk}}_i, \hat{\mathsf{mk}}_i)_{i\in[n]}, (\mathsf{mk}'_h)_{h\in H}, H, \hat{x}, (\hat{c}_i)_{i\in[n]}])$ (see Fig. 29)

Hybrid 6.$\hat{x}$. In this hybrid, the challenger replies to the query $(\mathsf{Correlation}, \hat{x})$ directly sending the values $(\hat{R}_i)_{i\in[n]}$ sampled by $\mathcal{C}$ during the generation of PCG. This hybrid is indistinguishable from the previous one by the correctness of iO and PKE.

Hybrid 7.$\hat{x}$. In this hybrid, we change how we produce the samples $(\hat{R}_i)_{i\in[n]}$. Specifically, we first generate $(R'_i)_{i\in[n]}$ using the correlation function $\mathcal{C}$ and substituting mk'_h to $\hat{\mathsf{mk}}_h$ for every $h \in H$. Then, we obtain $(\hat{R}_h)_{h\in H}$ by feeding the original master secrets $(\hat{\mathsf{mk}}_i)_{i\in[n]}$ and $(R'_i)_{i\in C}$ into $\mathsf{RSample}$. Finally, we set $\hat{R}_i$ to R'_i for every $i \in C$. Observe that this hybrid is indistinguishable from the previous one by the reverse samplability of $(\mathsf{Secret}, \mathcal{C})$.

The formal steps performed by the challenger for the generation of CGP become the following.

1. $\hat{k} \leftarrow \mathsf{Punct}'\big(k, (\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n)\big)$
2. $\hat{K} \leftarrow \mathsf{Punct}\big(K, (\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n, \hat{x})\big)$
3. $\forall i \in [n]: \quad \hat{\mathsf{mk}}_i \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, i)$
4. $\forall h \in H: \quad \mathsf{mk}'_h \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, h)$
5. $\forall i \in C: \quad \mathsf{mk}'_i \leftarrow \hat{\mathsf{mk}}_i$
6. $(R'_i)_{i\in[n]} \xleftarrow{\$} \mathcal{C}(1\!\!1^\lambda, \mathsf{mk}'_1, \mathsf{mk}'_2, \ldots, \mathsf{mk}'_n)$
7. $(\hat{R}_h)_{h\in H} \xleftarrow{\$} \mathsf{RSample}\big(1\!\!1^\lambda, C, (R'_i)_{i\in C}, (\hat{\mathsf{mk}}_i)_{i\in C}, (\hat{\mathsf{mk}}_h)_{h\in H}\big)$
8. $\forall i \in C: \quad \hat{R}_i \leftarrow R'_i$
9. $\forall i \in [n]: \quad \hat{c}_i \xleftarrow{\$} \mathsf{PKE.Enc}\big(\hat{\mathsf{pk}}_i, (\hat{R}_i, \hat{\mathsf{mk}}_i)\big)$
10. $\mathsf{CGP} \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^{\hat{x},3}_{\mathsf{CG}}[\hat{k}, \hat{K}, (\hat{\mathsf{pk}}_i, \hat{\mathsf{mk}}_i)_{i\in[n]}, (\mathsf{mk}'_h)_{h\in H}, H, \hat{x}, (\hat{c}_i)_{i\in[n]}])$ (see Fig. 29)

Hybrid 8.$\hat{x}$. In this stage, instead of hard-coding into CGP the encryption of $(\hat{R}_h, \hat{\mathsf{mk}}_h)$, for every $h \in H$, we store the encryption of (R'_h, mk'_h). The challenger, however, replies to the query $(\mathsf{Correlation}, \hat{x})$ by feeding the actual samples of the corrupted parties and the original master secrets $(\hat{\mathsf{mk}}_i)_{i\in[n]}$ into $\mathsf{RSample}$. Observe that the challenger does not need to know the secret-keys of the honest parties to reply to the queries $(\mathsf{Correlation}, x)$ with $x \neq \hat{x}$. The knowledge of the keys K and k permits indeed to recompute the samples $(R_i)_{i\in[n]}$. This fact allows us to reduce the indistinguishability between Hybrid 7 and 8 to the IND-CPA security of PKE.

The formal steps performed by the challenger for the generation of CGP become the following.

1. $\hat{k} \leftarrow \mathsf{Punct}'\big(k, (\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n)\big)$
2. $\hat{K} \leftarrow \mathsf{Punct}\big(K, (\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n, \hat{x})\big)$
3. $\forall i \in [n]: \quad \hat{\mathsf{mk}}_i \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, i)$
4. $\forall h \in H: \quad \mathsf{mk}'_h \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, h)$
5. $\forall i \in C: \quad \mathsf{mk}'_i \leftarrow \hat{\mathsf{mk}}_i$
6. $(R'_i)_{i\in[n]} \xleftarrow{\$} \mathcal{C}(1\!\!1^\lambda, \mathsf{mk}'_1, \mathsf{mk}'_2, \ldots, \mathsf{mk}'_n)$
7. $\forall i \in [n]: \quad \hat{c}_i \xleftarrow{\$} \mathsf{PKE.Enc}\big(\hat{\mathsf{pk}}_i, (R'_i, \mathsf{mk}'_i)\big)$
8. $\mathsf{CGP} \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^{\hat{x},3}_{\mathsf{CG}}[\hat{k}, \hat{K}, (\hat{\mathsf{pk}}_i, \hat{\mathsf{mk}}_i)_{i\in[n]}, (\mathsf{mk}'_h)_{h\in H}, H, \hat{x}, (\hat{c}_i)_{i\in[n]}])$ (see Fig. 29)

Furthermore, the response to $(\mathsf{Correlation}, \hat{x})$ is now computed as follows.

1. $(c_i)_{i\in[n]} \leftarrow \mathsf{CGP}(\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n, \hat{x})$
2. $\forall i \in C: \quad (R_i, \mathsf{mk}_i) \leftarrow \mathsf{PKE.Dec}(\hat{\mathsf{sk}}_i, c_i)$
3. $(R_h)_{h\in H} \xleftarrow{\$} \mathsf{RSample}\big(1\!\!1^\lambda, C, (R_i)_{i\in C}, (\hat{\mathsf{mk}}_i)_{i\in C}, (\hat{\mathsf{mk}}_h)_{h\in H}\big)$

Hybrid 9.$\hat{x}$. In this hybrid, we change how we generate the samples $(R'_i)_{i\in[n]}$ and we encrypt them. Specifically, instead of sampling the randomness for $\mathcal{C}$ and $\mathsf{PKE.Enc}$ uniformly, we rely on the output of the PRF F_K when its nonce is

the position where we had previously punctured. Indistinguishability from the previous stage holds by the security of the puncturable PRF F.

The formal steps performed by the challenger for the generation of CGP become the following.

1. $\hat{k} \leftarrow \mathsf{Punct}'\big(k, (\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n)\big)$
2. $\hat{K} \leftarrow \mathsf{Punct}\big(K, (\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n, \hat{x})\big)$
3. $(\hat{r}, \hat{r}_1, \hat{r}_2, \ldots, \hat{r}_n) \leftarrow F_K(\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n, \hat{x})$
4. $\forall i \in [n]: \quad \hat{\mathsf{mk}}_i \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, i)$
5. $\forall h \in H: \quad \mathsf{mk}'_h \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, h)$
6. $\forall i \in C: \quad \mathsf{mk}'_i \leftarrow \hat{\mathsf{mk}}_i$
7. $(R'_i)_{i\in[n]} \leftarrow \mathcal{C}(1\!\!1^\lambda, \mathsf{mk}'_1, \mathsf{mk}'_2, \ldots, \mathsf{mk}'_n; \hat{r})$
8. $\forall i \in [n]: \quad \hat{c}_i \leftarrow \mathsf{PKE.Enc}\big(\hat{\mathsf{pk}}_i, (R'_i, \mathsf{mk}'_i); \hat{r}_i\big)$
9. $\mathsf{CGP} \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^{\hat{x},3}_{\mathsf{CG}}[\hat{k}, \hat{K}, (\hat{\mathsf{pk}}_i, \hat{\mathsf{mk}}_i)_{i\in[n]}, (\mathsf{mk}'_h)_{h\in H}, H, \hat{x}, (\hat{c}_i)_{i\in[n]}])$ (see Fig. 29)

Hybrid 10.$\hat{x}$. In this hybrid, we do not puncture K anymore and we remove $(\hat{c}_i)_{i\in[n]}$ from CGP. When the public keys of the parties and the nonce $\hat{x}$ are input into the program, we compute the output running the same procedure as if the nonce was strictly smaller than $\hat{x}$. Observe that the input-output behaviour of the program is the same as in the previous hybrid, therefore indistinguishability follows from the security of iO.

The formal procedure adopted by the challenger for the generation of CGP becomes the following.

1. $\hat{k} \leftarrow \mathsf{Punct}'\big(k, (\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n)\big)$
2. $\forall i \in [n]: \quad \hat{\mathsf{mk}}_i \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, i)$
3. $\forall h \in H: \quad \mathsf{mk}'_h \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, h)$
4. $\mathsf{CGP} \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^{\hat{x},4}_{\mathsf{CG}}[\hat{k}, K, (\hat{\mathsf{pk}}_i, \hat{\mathsf{mk}}_i)_{i\in[n]}, (\mathsf{mk}'_h)_{h\in H}, H, \hat{x}])$ (see Fig. 30)

The next step of the proof is to repeat Hybrid 3-10 for the following value of the nonce $\hat{x}$. When the procedure has been applied to all the elements of the nonce space, we move to Hybrid 11.

Hybrid 11. In this stage, we change how we generate the master secrets $(\mathsf{mk}'_h)_{h\in H}$ and $(\hat{\mathsf{mk}}_i)_{i\in C}$. Specifically, instead of sampling the randomness fed into Secret uniformly, we rely on the output of the PRF F'_k when its nonce is the position where we had previously punctured. Observe that this hybrid is indistinguishable from the previous one by the security of the puncturable PRF F'.

The formal procedure used by the challenger for the generation of CGP becomes the following.

1. $\hat{k} \leftarrow \mathsf{Punct}'\big(k, (\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n)\big)$
2. $(\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_n) \leftarrow F'_k(\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n)$
3. $\forall i \in [n]: \quad \mathsf{mk}'_i \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, i; \hat{s}_i)$
4. $\forall i \in C: \quad \hat{\mathsf{mk}}_i \leftarrow \mathsf{mk}'_i$

$\mathcal{P}^{\hat{x},4}_{\mathsf{CG}}[k, K, (\hat{\mathsf{pk}}_i, \hat{\mathsf{mk}}_i)_{i\in[n]}, (\mathsf{mk}'_h)_{h\in H}, H, \hat{x}]$

Hard-coded. The puncturable PRF keys k and K, the public keys $(\hat{\mathsf{pk}}_i)_{i\in[n]}$ and the master secrets $(\hat{\mathsf{mk}}_i)_{i\in[n]}$ and $(\mathsf{mk}'_i)_{i\in H}$, the set of honest parties H and the nonce $\hat{x}$.
Input. A nonce $x \in \{0,1\}^{l(\lambda)}$ and n public keys $\mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_n$.

1. $(r, r_1, r_2, \ldots, r_n) \leftarrow F_K(\mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_n, x)$.
2. If $\mathsf{pk}_i = \hat{\mathsf{pk}}_i$ for every $i \in [n]$ and $x >_{\text{lex}} \hat{x}$, set $\mathsf{mk}_i \leftarrow \hat{\mathsf{mk}}_i$ for every $i \in [n]$.
3. If $\mathsf{pk}_i = \hat{\mathsf{pk}}_i$ for every $i \in [n]$ and $x \leq_{\text{lex}} \hat{x}$, set $\mathsf{mk}_h \leftarrow \mathsf{mk}'_h$ for every $h \in H$ and $\mathsf{mk}_i \leftarrow \hat{\mathsf{mk}}_i$ for every $i \in C$.
4. Otherwise, perform the following operations
   (a) $(s_1, s_2, \ldots, s_n) \leftarrow F'_k(\mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_n)$
   (b) $\forall i \in [n]: \quad \mathsf{mk}_i \leftarrow \mathsf{Secret}(1\!\!1^\lambda, i; s_i)$
5. $(R_1, R_2, \ldots, R_n) \leftarrow \mathcal{C}(1\!\!1^\lambda, \mathsf{mk}_1, \mathsf{mk}_2, \ldots, \mathsf{mk}_n; r)$
6. $\forall i \in [n]: \quad c_i \leftarrow \mathsf{PKE.Enc}\big(\mathsf{pk}_i, (R_i, \mathsf{mk}_i); r_i\big)$
7. Output $c_1, c_2, \ldots, c_n$.

Fig. 30. The Correlation Generation Program

5. $\forall h \in H: \quad \hat{\mathsf{mk}}_h \xleftarrow{\$} \mathsf{Secret}(1\!\!1^\lambda, h)$
6. $\mathsf{CGP} \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^{\Omega,4}_{\mathsf{CG}}[\hat{k}, K, (\hat{\mathsf{pk}}_i, \hat{\mathsf{mk}}_i)_{i\in[n]}, (\mathsf{mk}'_h)_{h\in H}, H, \Omega])$ (see Fig. 30)

Hybrid 12. This is the final stage and corresponds to the ideal world. In this hybrid, we do not puncture k anymore and we generate CGP by obfuscating the original program $\mathcal{P}_{\mathsf{CG}}$ (see Fig. 19). Observe that the input-output behaviour of CGP is the same as in Hybrid 11. Indeed, there is no $x >_{\text{lex}} \hat{x}$ because $\hat{x}$ has reached the maximum. Indistinguishability holds by the security of iO.

Also notice that the challenger replies to every query $(\mathsf{Correlation}, x)$ using the following procedure.

1. $(c_i)_{i\in[n]} \leftarrow \mathsf{CGP}(\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n, x)$
2. $\forall i \in C: \quad (R_i, \mathsf{mk}_i) \leftarrow \mathsf{PKE.Dec}(\hat{\mathsf{sk}}_i, c_i)$
3. $(R_h)_{h\in H} \xleftarrow{\$} \mathsf{RSample}\big(1\!\!1^\lambda, C, (R_i)_{i\in C}, (\hat{\mathsf{mk}}_i)_{i\in C}, (\hat{\mathsf{mk}}_h)_{h\in H}\big)$

Furthermore, observe that the master secrets $(\hat{\mathsf{mk}}_h)_{h\in H}$ are sampled at random using Secret and they are independent of the public keys and CGP. Finally, we formalise the operations of the extractor.

$\mathsf{Extract}(C, \mathsf{CGP}, \rho_1, \ldots, \rho_n)$

1. $\forall i \in [n]: \quad (\hat{\mathsf{sk}}_i, \hat{\mathsf{pk}}_i) \leftarrow \mathsf{PKE.Gen}(1\!\!1^\lambda; \rho_i)$
2. $(c_i)_{i\in[n]} \leftarrow \mathsf{CGP}(\hat{\mathsf{pk}}_1, \hat{\mathsf{pk}}_2, \ldots, \hat{\mathsf{pk}}_n, \epsilon)$
3. $\forall i \in C: \quad (R_i, \mathsf{mk}_i) \leftarrow \mathsf{PKE.Dec}(\hat{\mathsf{sk}}_i, c_i)$
4. Output $(\mathsf{mk}_i)_{i\in C}$.

Observe that in every hybrid the size of CGP is polynomial in the length $l(\lambda)$ of the nonces. Furthermore, the size of the keys is always independent of the size of the nonce space. □

C Proof of Theorem 6.10

Proof. Let $\mathsf{pkPCFS} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval})$ be the sub-exponentially and semi-maliciously secure public key PCF with trusted setup for $(\mathsf{Secret}, \mathcal{C})$. Assume that $\mathsf{DS} = (\mathsf{Gen}, \mathsf{Sample})$ is a sub-exponentially and semi-maliciously secure distributed sampler for $\mathsf{pkPCFS.Setup}$. Moreover, suppose that the algorithms $\mathsf{DS.Gen}$ and $\mathsf{pkPCFS.Gen}$ need $L(\lambda)$ and $L'(\lambda)$ bits of randomness for their execution respectively. We rely on a PRG G mapping a λ-bit seed into a pseudo-random string of $L(\lambda) + L'(\lambda)$ bits. We assume that the output of G is naturally split into two blocks of length $L(\lambda)$ and $L'(\lambda)$ bits respectively. Finally, let $\mathsf{NIZK} = (\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$ be a simulation-extractable NIZK proving the well-formedness of sampler shares and PCF public keys. Specifically, in the relation corresponding to NIZK, the statement consists of a tuple (U, pk, i), whereas the witness is a pair (s, sk) such that

$$U = \mathsf{DS.Gen}(1\!\!1^\lambda, i; r), \qquad (\mathsf{sk}, \mathsf{pk}) = \mathsf{pkPCFS.Gen}(1\!\!1^\lambda, i; r'), \qquad (r, r') = G(s).$$

Our actively secure public key PCF $\Pi_{\mathsf{exp}\text{-}\mathcal{C}}$ is described in Fig. 31. We now prove that no PPT adversary is able to distinguish between $\Pi_{\mathsf{exp}\text{-}\mathcal{C}}$ and the composition of $\mathcal{F}_{\mathcal{C}}^{\mathsf{RSample}}$ with a PPT simulator we are going to present. The proof proceeds by a series of 6 indistinguishable hybrids (some of them repeated multiple times) going from $\Pi_{\mathsf{exp}\text{-}\mathcal{C}}$ (Hybrid 0) to the ideal world (Hybrid 6).

$\Pi_{\mathsf{exp}\text{-}\mathcal{C}}$

URS. The protocol needs a URS $\mathsf{urs} \xleftarrow{\$} \mathsf{NIZK.Gen}(1\!\!1^\lambda)$ for the NIZK proofs.
Initialisation. Each party P_i performs the following steps.

1. $s_i \xleftarrow{\$} \{0,1\}^\lambda$
2. $(r_i, r_i') \leftarrow G(s_i)$
3. $U_i \leftarrow \mathsf{DS.Gen}(1\!\!1^\lambda, i; r_i)$
4. $(\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow \mathsf{pkPCFS.Gen}(1\!\!1^\lambda, i; r_i')$
5. $\pi_i \xleftarrow{\$} \mathsf{NIZK.Prove}\big(1\!\!1^\lambda, \mathsf{urs}, (U_i, \mathsf{pk}_i, i), (s_i, \mathsf{sk}_i)\big)$
6. Broadcast $(U_i, \mathsf{pk}_i, \pi_i)$ and wait for a similar message from every other party.
7. If there exists $j \in [n]$ such that $\mathsf{NIZK.Verify}\big(\mathsf{urs}, \pi_j, (U_j, \mathsf{pk}_j, j)\big) = 0$ abort.
8. $S \leftarrow \mathsf{DS.Sample}(U_1, U_2, \ldots, U_n)$

Correlation. On input a nonce $x \in \{0,1\}^{l(\lambda)}$, each party P_i outputs $R_i \leftarrow \mathsf{pkPCFS.Eval}\big(i, S, (\mathsf{pk}_j)_{j\in[n]}, \mathsf{sk}_i, x\big)$.

Fig. 31. Actively Secure Public Key PCF based on Sub-Exponentially Secure Primitives

Hybrid 0. This hybrid coincides with the real world. The simulator runs the protocol $\Pi_{\mathsf{exp}\text{-}\mathcal{C}}$ on behalf of the honest parties. Specifically, it starts its execution producing the URS for the NIZK proofs, it generates the sampler shares and the

keys of the honest parties, proves their well-formedness and sends everything except the private keys to the adversary. Moreover, the simulator replies to the correlation queries using $\mathsf{pkPCFS.Eval}$ as in $\Pi_{\mathsf{exp}\text{-}\mathcal{C}}$.

Hybrid 1. In this hybrid, we change how we generate the URS and the NIZK proofs of the honest parties. Specifically, we substitute them with the output of the simulators $\mathsf{NIZK.Sim}_1$ and $\mathsf{NIZK.Sim}_2$. Indistinguishability between Hybrid 0 and 1 follows from the multi-theorem zero-knowledge of NIZK. Formally, the steps performed by the simulator for the generation of the messages of the honest parties are the following (the red text indicates what changed since the last hybrid).

1. $\forall h \in H: \quad (r_h, r'_h) \leftarrow G(s_h)$
2. $\forall h \in H: \quad U_h \xleftarrow{\$} \mathsf{DS.Gen}(1\!\!1^\lambda, h)$
3. $\forall h \in H: \quad (\mathsf{sk}_h, \mathsf{pk}_h) \leftarrow \mathsf{pkPCFS.Gen}(1\!\!1^\lambda, h; r'_h)$
4. $(\mathsf{urs}, \tau) \xleftarrow{\$} \mathsf{NIZK.Sim}_1(1\!\!1^\lambda)$
5. $\forall h \in H: \quad \pi_h \xleftarrow{\$} \mathsf{NIZK.Sim}_2\big(\mathsf{urs}, \tau, (U_h, \mathsf{pk}_h, h)\big)$

Hybrid 2. In this hybrid, we change how we generate the randomness $(r_h, r'_h)_{h\in H}$ of the honest parties. Specifically, instead of expanding the seed s_h, we sample r_h and r'_h uniformly in $\{0,1\}^{L(\lambda)}$ and $\{0,1\}^{L'(\lambda)}$ respectively. Observe that this hybrid is indistinguishable from the previous one by the PRG security of G.

Next, we repeat the hybrids from 3 to 5 for every possible choice of the seeds of the corrupted parties $\hat{\rho} := (\hat{s}_i)_{i\in C}$. We follow the lexicographical order starting from the minimum.

Hybrid 3.$\hat{\rho}$. In this hybrid, we change how we generate the sampler shares of the honest parties. Let $(\hat{r}_i, \hat{r}'_i)$ be $G(\hat{s}_i)$ for every $i \in C$. We rely on $\mathsf{DS.Sim}$, providing it with the randomness $(\hat{r}_i)_{i\in C}$ and an element $\hat{S}$ produced by $\mathsf{pkPCFS.Setup}$.

We also change the way we reply to the correlation queries. At the beginning of its execution, the simulator samples a random mk'_h for every $h \in H$. When it receives the messages of the corrupted parties, the simulator extracts the witnesses from the NIZK proofs, obtaining the seeds $\rho = (s_i)_{i\in C}$ and the corresponding secret keys. The operation can be performed due to simulation-extractability. Furthermore, the simulator derives the master secrets $(\mathsf{mk}_i)_{i\in C}$ of the corrupt parties by computing $(r_i, r'_i) \leftarrow G(s_i)$ for every $i \in C$ and running $\mathsf{pkPCFS.Extract}$ on $(r'_i)_{i\in[n]}$ and S. The reply to the correlation queries is then computed as follows.

- If $\rho = \hat{\rho}$, the simulator substitutes the output of $\mathsf{DS.Sample}$ with $\hat{S}$ in $\mathsf{pkPCFS.Eval}$. It answers with the results.
- If $\rho >_{\text{lex}} \hat{\rho}$, the simulator replies as in the real protocol.
- If $\rho <_{\text{lex}} \hat{\rho}$, the simulator extracts the outputs of the corrupted parties by relying on their private keys, feeds the obtained values into $\mathsf{RSample}$ along with $(\mathsf{mk}_i)_{i\in C}$ and $(\mathsf{mk}'_h)_{h\in H}$ and, at the end, answers with the results.

Observe that this hybrid is indistinguishable from the previous one by the semi-malicious security of DS. Notice that when $\hat{\rho}$ is minimum, the simulator

never relies on $\mathsf{RSample}$. The formal steps used by the simulator to generate $(U_h)_{h\in H}$ become the following.

1. $\forall i \in C: \quad (\hat{r}_i, \hat{r}'_i) \leftarrow G(\hat{s}_i)$
2. $\hat{S} \xleftarrow{\$} \mathsf{pkPCFS.Setup}(1\!\!1^\lambda)$
3. $(U_h)_{h\in H} \xleftarrow{\$} \mathsf{DS.Sim}\big(1\!\!1^\lambda, C, \hat{S}, (\hat{r}_i)_{i\in C}\big)$

The reply to $(\mathsf{Correlation}, x)$ is instead computed as follows.

1. $S \leftarrow \mathsf{DS.Sample}(U_1, U_2, \ldots, U_n)$
2. $\forall i \in C: \quad (s_i, \mathsf{sk}_i) \leftarrow \mathsf{NIZK.Extract}\big(\mathsf{urs}, \tau, (U_i, \mathsf{pk}_i, i), \pi_i\big)$
3. $\forall i \in C: \quad (r_i, r'_i) \leftarrow G(s_i)$
4. $(\mathsf{mk}_i)_{i\in C} \leftarrow \mathsf{pkPCFS.Extract}(C, S, r'_1, r'_2, \ldots, r'_n)$
5. $\rho \leftarrow (s_i)_{i\in C}$
6. If $\rho = \hat{\rho}$, compute $R_h \leftarrow \mathsf{pkPCFS.Eval}\big(h, \hat{S}, (\mathsf{pk}_i)_{i\in[n]}, \mathsf{sk}_h, x\big)\ \forall h \in H$.
7. If $\rho >_{\text{lex}} \hat{\rho}$, compute $R_h \leftarrow \mathsf{pkPCFS.Eval}\big(h, S, (\mathsf{pk}_i)_{i\in[n]}, \mathsf{sk}_h, x\big)\ \forall h \in H$.
8. If $\rho <_{\text{lex}} \hat{\rho}$, compute
   (a) $\forall i \in C: \quad R_i \leftarrow \mathsf{pkPCFS.Eval}\big(i, S, (\mathsf{pk}_j)_{j\in[n]}, \mathsf{sk}_i, x\big)$
   (b) $(R_h)_{h\in H} \xleftarrow{\$} \mathsf{RSample}\big(1\!\!1^\lambda, C, (R_i)_{i\in C}, (\mathsf{mk}_i)_{i\in C}, (\mathsf{mk}'_h)_{h\in H}\big)$

Hybrid 4.$\hat{\rho}$. In this hybrid, we change how we generate the outputs of the honest parties when the seeds of the corrupted players coincide with $\hat{\rho}$. Specifically, we now reverse sample them. We can indeed retrieve the samples addressed to the corrupted parties using their private keys. The latter can be extracted from the corresponding NIZK proofs along with the seeds of the corrupted players. Using $\mathsf{pkPCFS.Extract}$, it is also possible to derive the master secrets $(\mathsf{mk}_i)_{i\in C}$.

Observe that this hybrid is indistinguishable from the previous one by the semi-malicious security of pkPCFS. As a matter of fact, if the randomness ρ chosen by the adversary is different from $\hat{\rho}$, the two stages are perfectly identical. If instead $\rho = \hat{\rho}$, we can reduce distinguishability between Hybrid 2.$\hat{\rho}$ and 3.$\hat{\rho}$ to the security game $\mathcal{G}^{C,(\hat{r}'_i)_{i\in C}}_{\text{SetupSec}}(\lambda)$.

When $\rho = \hat{\rho}$, the simulator replies to $(\mathsf{Correlation}, x)$ using the following procedure.

1. $\forall i \in C: \quad (s_i, \mathsf{sk}_i) \leftarrow \mathsf{NIZK.Extract}\big(\mathsf{urs}, \tau, (U_i, \mathsf{pk}_i, i), \pi_i\big)$
2. $\forall i \in C: \quad (r_i, r'_i) \leftarrow G(s_i)$
3. $(\mathsf{mk}_i)_{i\in C} \leftarrow \mathsf{pkPCFS.Extract}(C, \hat{S}, r'_1, r'_2, \ldots, r'_n)$
4. $\forall i \in C: \quad R_i \leftarrow \mathsf{pkPCFS.Eval}\big(i, \hat{S}, (\mathsf{pk}_j)_{j\in[n]}, \mathsf{sk}_i, x\big)$
5. $(R_h)_{h\in H} \xleftarrow{\$} \mathsf{RSample}\big(1\!\!1^\lambda, C, (R_i)_{i\in C}, (\mathsf{mk}_i)_{i\in C}, (\mathsf{mk}'_h)_{h\in H}\big)$

Hybrid 5.$\hat{\rho}$. In this hybrid, we revert to the original procedure for the generation of the sampler shares of the honest parties. Specifically, we do not rely anymore on $\mathsf{DS.Sim}$, but we use $\mathsf{DS.Gen}$. Observe that this hybrid is indistinguishable from the previous one by the semi-malicious security of DS.

If $\rho = \hat{\rho}$, the procedure used by the simulator to reply to $(\mathsf{Correlation}, x)$ becomes the following.

1. $S \leftarrow \mathsf{DS.Sample}(U_1, U_2, \ldots, U_n)$

2. $\forall i \in C: \quad (s_i, \mathsf{sk}_i) \leftarrow \mathsf{NIZK.Extract}\big(\mathsf{urs}, \tau, (U_i, \mathsf{pk}_i, i), \pi_i\big)$
3. $\forall i \in C: \quad (r_i, r_i') \leftarrow G(s_i)$
4. $(\mathsf{mk}_i)_{i \in C} \leftarrow \mathsf{pkPCFS.Extract}(C, S, r_1', r_2', \ldots, r_n')$
5. $\forall i \in C: \quad R_i \leftarrow \mathsf{pkPCFS.Eval}\big(i, S, (\mathsf{pk}_j)_{j \in [n]}, \mathsf{sk}_i, x\big)$
6. $(R_h)_{h \in H} \xleftarrow{\$} \mathsf{RSample}\big(\mathbb{1}^\lambda, C, (R_i)_{i \in C}, (\mathsf{mk}_i)_{i \in C}, (\mathsf{mk}_h')_{h \in H}\big)$

The next step is to repeat Hybrid 3-5 for the next value $\hat{\rho}$ of the seeds of the corrupted parties. If $\hat{\rho}$ has reached the maximum, we move to Hybrid 6.

Hybrid 6. This hybrid corresponds to the ideal world and summarises what we have achieved so far. In this final stage, the simulator selects the sampler shares and the keys of the honest parties as in the original protocol, using however true randomness instead of expanding PRG seeds. The URS and the NIZK proofs are generated by relying on the simulators $\mathsf{NIZK.Sim}_1$ and $\mathsf{NIZK.Sim}_2$ as in Hybrid 1.

When the simulator receives the messages of the corrupted parties from the adversary, it extracts their seeds and private keys from the zero-knowledge proofs using NIZK.Extract. At that point, it has all the necessary information to retrieve the master secrets $(\mathsf{mk}_i)_{i \in C}$ of the corrupted players by means of pkPCFS.Extract. The values are sent to $\mathcal{F}_{\mathcal{C}}^{\mathsf{RSample}}$.

Upon receiving any query $(\mathsf{Correlation}, x)$, the simulator is also able to compute the outputs of the corrupted parties as it knows their private keys. So it is just sufficient to forward the results to the functionality. The latter will take care of the generation and distribution of the samples of the honest players using RSample. The formal description of the simulator is available is Fig. 32.

□

$\mathcal{S}_{\mathsf{exp}\text{-}\mathcal{C}}$

Initialisation.

1. $\forall h \in H: \quad r'_h \xleftarrow{\$} \{0,1\}^{L'(\lambda)}$
2. $\forall h \in H: \quad U_h \xleftarrow{\$} \mathsf{DS.Gen}(1\!\!1^\lambda, h)$
3. $\forall h \in H: \quad (\mathsf{sk}_h, \mathsf{pk}_h) \leftarrow \mathsf{pkPCFS.Gen}(1\!\!1^\lambda, h;\, r'_h)$
4. $(\mathsf{urs}, \tau) \xleftarrow{\$} \mathsf{NIZK.Sim}_1(1\!\!1^\lambda)$
5. $\forall h \in H: \quad \pi_h \xleftarrow{\$} \mathsf{NIZK.Sim}_2\big(\mathsf{urs}, \tau, (U_h, \mathsf{pk}_h, h)\big)$
6. Send $(U_h, \mathsf{pk}_h, \pi_h)_{h\in H}$ to the adversary and wait for $(U_i, \mathsf{pk}_i, \pi_i)_{i\in C}$ as a reply.
7. If there exists $i \in C$ such that $\mathsf{NIZK.Verify}\big(\mathsf{urs}, \pi_i, (U_i, \mathsf{pk}_i, i)\big) = 0$, send Abort to the functionality.
8. $S \leftarrow \mathsf{DS.Sample}(U_1, U_2, \ldots, U_n)$
9. $\forall i \in C: \quad (s_i, \mathsf{sk}_i) \leftarrow \mathsf{NIZK.Extract}\big(\mathsf{urs}, \tau, (U_i, \mathsf{pk}_i, i), \pi_i\big)$
10. $\forall i \in C: \quad (r_i, r'_i) \leftarrow G(s_i)$
11. $(\mathsf{mk}_i)_{i\in C} \leftarrow \mathsf{pkPCFS.Extract}(C, S, r'_1, r'_2, \ldots, r'_n)$
12. Send $(\mathsf{mk}_i)_{i\in C}$ to the functionality.

Correlation. On input $(\mathsf{Correlation}, x)$ where $x \in \{0,1\}^{l(\lambda)}$, the simulator sends to the functionality $R_i \leftarrow \mathsf{pkPCFS.Eval}\big(i, S, (\mathsf{pk}_j)_{j\in[n]}, \mathsf{sk}_i, x\big)$ for every $i \in C$.

Fig. 32. Simulator for $\Pi_{\mathsf{exp}\text{-}\mathcal{C}}$