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

Damiano Abram(✉), Peter Scholl, and Sophia Yakoubov

Aarhus University, Aarhus, Denmark
damiano.abram@cs.au.dk

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.

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

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.

© International Association for Cryptologic Research 2022
O. Dunkelman and S. Dziembowski (Eds.): EUROCRYPT 2022, LNCS 13275, pp. 790–820, 2022.
https://doi.org/10.1007/978-3-031-06944-4_27

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 (such as the modulus factorization) that should not be known to anyone.

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. 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].

[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.

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 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 ([ASY22])	any	public
Public-key PCF (pk-PCF, [OSY21])	fixed, reverse-samplable	private
Ideal pk-PCF ([ASY22])	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.) We defer the discussion of this last construction to the full version of this paper [ASY22, Section 7] due to space constraints.

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

[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.

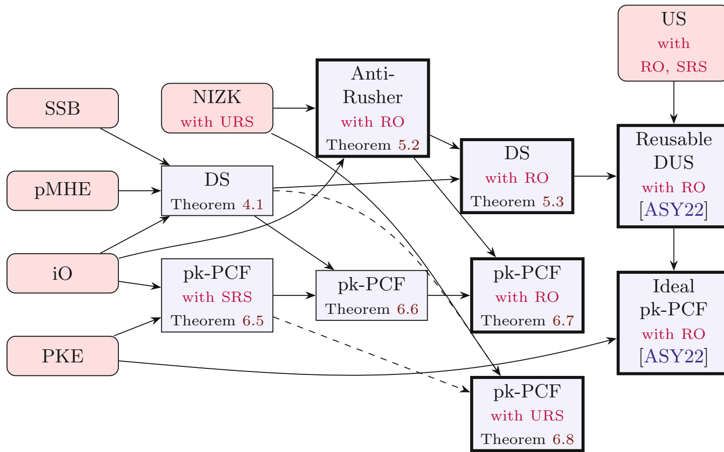


Fig. 2. In this table we describe the constructions in this paper. In pink are assumptions: they include somewhere statistically binding hash functions (SSB), multiparty homomorphic encryption with private evaluation (pMHE [AJJM20], a weaker form of multi-key FHE), indistinguishability obfuscation (iO), non-interactive zero knowledge proofs (NIZK), and universal samplers (US). In blue are constructions of distributed samplers (DS, Definition 3.1), reusable distributed universal samplers (reusable DUS, Definition 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. (Color figure online)

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, Sect. 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 (Sect. 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 ciphertext, 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* (Sect. 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 (Sect. 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 (Sect. 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 ([ASY22, 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.

Used Primitives. Our work relies on the following primitives.

- *Indistinguishability Obfuscation (iO).* [BGI+01] An obfuscator is an algorithm that rearranges a circuit Cr into another program Cr' with the same input-output behaviour, but being so different that it is impossible to tell what operations Cr initially performed. Specifically, security states that it is impossible to distinguish between the obfuscation of equivalent circuits. The

first indistinguishability obfuscator was designed by Garg *et al.* in [GGH+13]. Formal definitions of iO are given in [ASY22, Section 2.1].

- *Puncturable PRFs.*[KPTZ13, BW13, BGI14b] A puncturable PRF is a PRF F in which it is possible to puncture the keys in any position x. In other words, it means that from a key K, it is possible to derive another key $\hat{K}$ containing no information about $F_K(x)$ but still permitting to compute $F_K(y)$ for every $y \neq x$. It is easy to build puncturable PRF from the GGM construction [GGM86]. Formal definitions are given in [ASY22, Section 2.2].
- *Simulation-Extractable NIZKs.* [GO07] A NIZK for an NP relation $\mathcal{R}$ is a construction that allows proving the knowledge of a witness w for a statement x with only one round of interaction and without revealing any additional information about w. The zero-knowledge property is formalised by the existence of PPT simulators generating proofs without needing witnesses. The operation is performed exploiting a trapdoored CRS.
  We say that the NIZK is simulation-extractable if there exists an efficient algorithm that, in conjunction with the simulators, permits to extract the witness from any valid proof generated by the adversary.
  When the CRS is a random string of bits, we talk about NIZKs with URS. Formal definitions are given in [ASY22, Section 2.3].
- *Multiparty Homomorphic Encryption with Private Evaluation (pMHE).* MHE with private evaluation [AJJM20] is a construction that permits to evaluate circuits over encrypted values. It is possible to obtain partial decryptions with no interactions. Retrieving the actual plaintext requires however an additional round of communication as we need to pool the partial decryptions. MHE with private evaluation is a weaker version of multi-key FHE. The main differences is that there is actually no public key but only a private one that changes for every ciphertext. Furthermore, the encryption algorithm needs to know the parameters (input size, output size and depth) of the circuits we are going to evaluate. We can build pMHE from LWE [AJJM20]. Formal definitions are given in [ASY22, Section 2.4].
- *Somewhere Statistically Binding (SSB) Hashing.* [HW15] An SSB hash function is a keyed hash function with particular properties: every key hk hides an index i that specifies in which block the hash is statistically binding. Specifically, every pair of messages having the same digest under hk must coincide at the i-th block. It is possible to build SSB hash functions from fully homomorphic encryption [HW15]. Formal definitions are given in [ASY22, Section 2.5].

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:*

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

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

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. 3) against a static and active adversary corrupting up to* $n-1$ *parties.*

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

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. 3. Distributed sampler functionality

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 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 Sect. 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 $\mathsf{sk}_1, \mathsf{sk}_2, \dots, \mathsf{sk}_n$, 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. 4. 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 $\mathcal{D}$, pMHE.Enc, and HE.PrivEval respectively. Moreover, we denote by B the maximum number of blocks in the messages fed into 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. 4 is an n-party distributed sampler with semi-malicious security for the distribution* $\mathcal{D}$*.*

We prove Theorem 4.1 in [ASY22, Appendix A]. 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 Sect. 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.

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 $U := (\mathsf{hk}, \mathsf{KGProg}, \mathsf{EvProg})$.

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

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 $s_1, s_2, \dots, s_n$, 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. $d_i \leftarrow \mathsf{pMHE.PrivEval}(\mathsf{sk}_i, \tilde{\mathcal{D}}, c_1, c_2, \dots, c_n;\ r_3)$
7. Output d_i.

Fig. 4. A distributed sampler with semi-malicious security

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 Sect. 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.

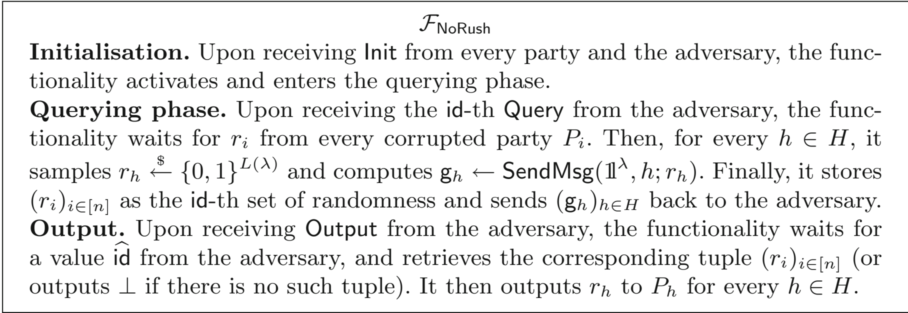
(In the case of distributed samplers, $\mathsf{SendMsg}$ corresponds to Gen, and Output corresponds to Sample.)

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. 5) for* $\mathsf{SendMsg}$ *against an active adversary.*

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.



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

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 $\mathsf{g}_h \leftarrow \mathsf{SendMsg}(\mathbb{1}^\lambda, h; r_h)$. Finally, it stores $(r_i)_{i\in[n]}$ as the id-th set of randomness 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 $(r_i)_{i\in[n]}$ (or outputs $\perp$ if there is no such tuple). It then outputs r_h to P_h for every $h \in H$.

Fig. 5. The anti-rushing functionality $\mathcal{F}_{\mathsf{NoRush}}$

Intuition Behind Our Anti-rushing Compiler. We define (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. 7. The unobfuscated program $\mathcal{P}_{\mathsf{AR}}$ is available in Fig. 6. 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.

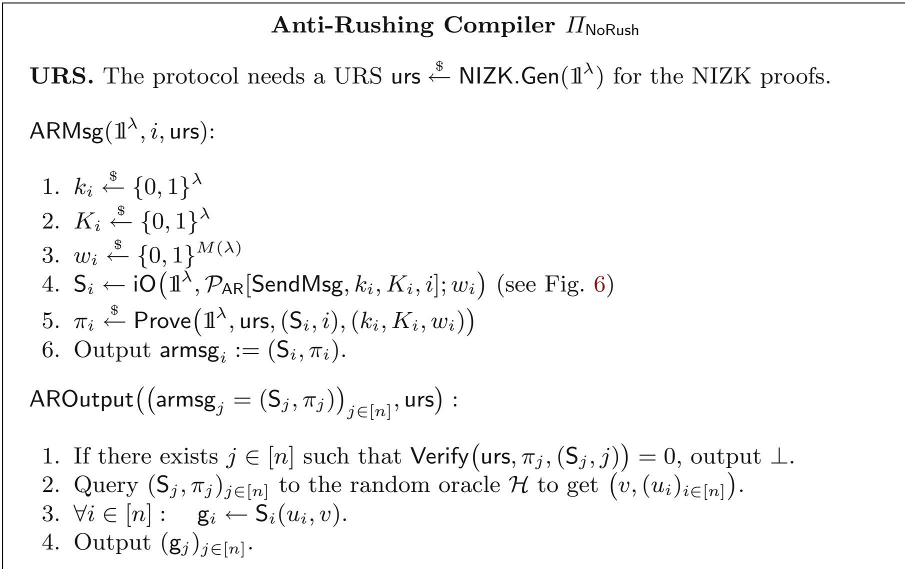
Finally, let $\mathsf{NIZK} = (\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Sim}_1, \mathsf{Sim}_2, \mathsf{Extract})$ be a simulation-extractable NIZK for the relation $\mathcal{R}$ describing the well-formedness of the obfuscated programs $(\mathsf{S}_i)_{i \in [n]}$. Formally, a statement 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
$$x^j \leftarrow \begin{cases} 0 & \text{if } y_j^0 = u^j, \\ 1 & \text{if } y_j^1 = u^j, \\ \perp & \text{otherwise.} \end{cases}$$
3. If $x^j \neq \perp$ for every $j \in [m]$, output $(x^1, x^2, \ldots, x^m)$.
4. Set $r \leftarrow F_K(u, v)$.
5. Output $\mathsf{g}_i \leftarrow \mathsf{SendMsg}(\mathbb{1}^\lambda, i; r)$.

Fig. 6. The anti-rushing program



Anti-Rushing Compiler Π_{NoRush}

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

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

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

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

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 $\big(v, (u_i)_{i\in[n]}\big)$.
3. $\forall i \in [n]: \quad \mathsf{g}_i \leftarrow \mathsf{S}_i(u_i, v)$.
4. Output $(\mathsf{g}_j)_{j\in[n]}$.

Fig. 7. Anti-rushing compiler

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. 7 realizes* $\mathcal{F}_{\mathsf{NoRush}}$ *for* $\mathsf{SendMsg}$ *in the random oracle model with a URS.*

We prove Theorem 5.2 in [ASY22, Appendix B].

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 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

[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].

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

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

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, $\mathsf{RSample}$ 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* $\mathbb{1}^\lambda$ *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.* Gen *needs* $L(\lambda)$ *bits of randomness.*

[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.

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

Initialisation.

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

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. $(R_i^1)_{i\in[n]} \xleftarrow{\$} \mathcal{C}(\mathbb{1}^\lambda, \mathsf{mk}'_1, \ldots, \mathsf{mk}'_n)$
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. 8. Correctness game for the public-key PCF

– 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* $\mathcal{G}_{\text{PCF-Corr}}(\lambda)$ *(see Fig. 8) 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* $\mathcal{G}_{\text{PCF-Sec}}^{C,(\rho_i)_{i\in C}}(\lambda)$ *(see Fig. 9) with noticeable advantage.*

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 $b = 1$.

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

Initialisation.
1. $b \xleftarrow{\$} \{0,1\}$
2. $\forall h \in H: \quad \rho_h \xleftarrow{\$} \{0,1\}^{L(\lambda)}$
3. $\forall i \in [n]: \quad (\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow \mathsf{Gen}(\mathbb{1}^\lambda, i; \rho_i)$
4. $(\mathsf{mk}_i)_{i\in C} \leftarrow \mathsf{Extract}(C, \rho_1, \rho_2, \dots, \rho_n)$.
5. $\forall h \in H: \quad \mathsf{mk}'_h \xleftarrow{\$} \mathsf{Secret}(\mathbb{1}^\lambda, h)$
6. Activate the adversary with $\mathbb{1}^\lambda$ 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. $\forall i \in [n]: \quad R_i^0 \leftarrow \mathsf{Eval}(i, \mathsf{pk}_1, \dots, \mathsf{pk}_n, \mathsf{sk}_i, x)$
2. $\forall i \in C: \quad R_i^1 \leftarrow R_i^0$
3. $(R_h^1)_{h\in H} \xleftarrow{\$} \mathsf{RSample}\big(\mathbb{1}^\lambda, C, (R_i^1)_{i\in C}, (\mathsf{mk}_i)_{i\in C}, (\mathsf{mk}'_h)_{h\in H}\big)$
4. Give $(R_1^b, R_2^b, \dots, R_n^b)$ to the adversary.

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

Fig. 9. Security game for the public-key PCF

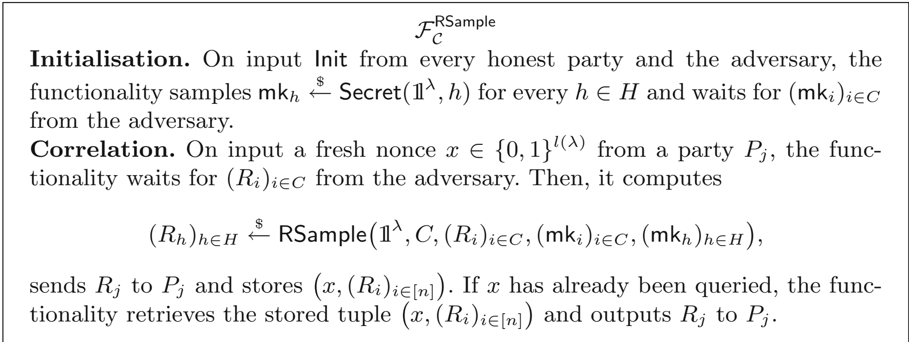
In our definition, nonces are adaptively chosen by the adversary; however, in a *weak* PCF [BCG+20a], the nonces are sampled randomly or selected by the adversary ahead of time. We can define a weak public-key PCF similarly, and use the same techniques as Boyle *et al.* [BCG+20a] to convert a weak public-key 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. 10) 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.

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.



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

Initialisation. On input Init from every honest party and the adversary, the functionality samples $\mathsf{mk}_h \xleftarrow{\$} \mathsf{Secret}(\mathbb{1}^\lambda, h)$ 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

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

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. 10. The actively secure public-key PCF functionality for reverse samplable correlation

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 Definition 6.2 to include an algorithm Setup that takes as input the security parameter $\mathbb{1}^\lambda$ 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 Definition 6.3, after minor tweaks to the games $\mathcal{G}_{\text{PCF-Corr}}(\lambda)$ and $\mathcal{G}_{\text{PCF-Sec}}^{C,(\rho_i)_{i \in C}}(\lambda)$ 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. Formal definitions of public-key PCF with trusted setup are available in [ASY22, Section 6.3].

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. 11 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 6.5 (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. 11 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* $\mathsf{poly}(\lambda)$*.*

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

We prove Theorem 6.5 in [ASY22, Appendix C].

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 $l(\lambda)$.

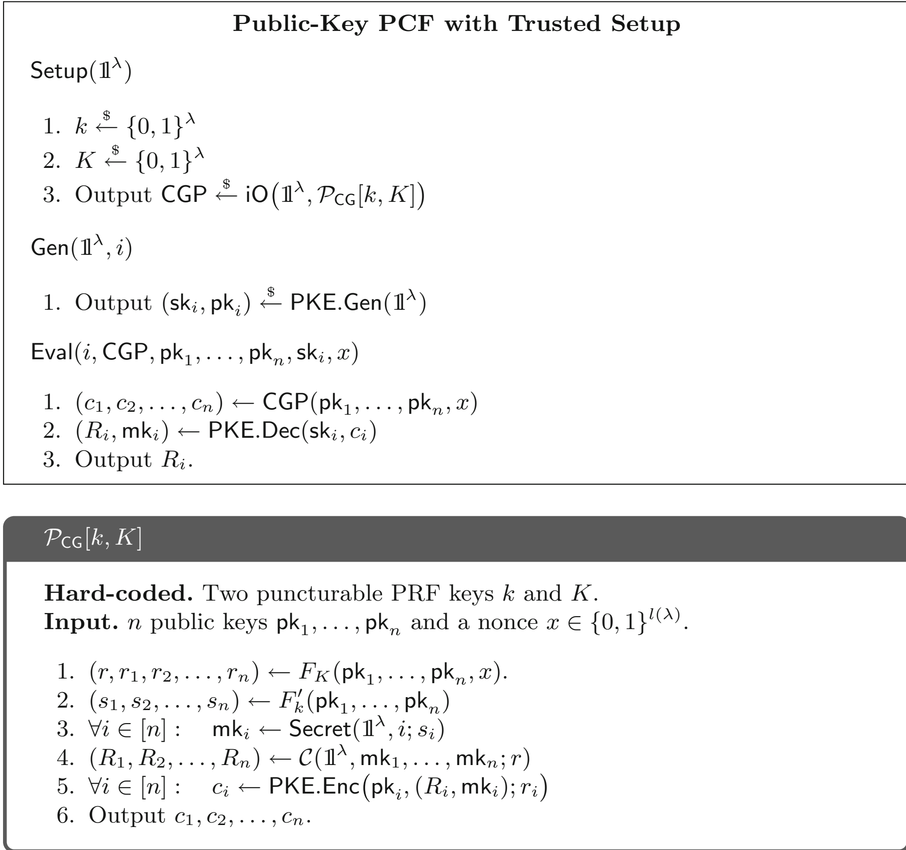


Fig. 11. A public-key PCF with trusted setup

Theorem 6.6 (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 6.6 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 Sect. 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 6.7 (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 6.8 (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 6.8 in [ASY22, Appendix D].

References

[AJJM20] Ananth, P., Jain, A., Jin, Z., Malavolta, G.: Multi-key fully-homomorphic encryption in the plain model. In: Pass, R., Pietrzak, K. (eds.) TCC 2020. LNCS, vol. 12550, pp. 28–57. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64375-1_2

[ASY22] Abram, D., Scholl, P., Yakoubov, S.: Distributed (correlation) samplers: how to remove a trusted dealer in one round. Cryptology ePrint Archive, Report 2022/? (2022)

[BCG+19a] Boyle, E., et al.: Efficient two-round OT extension and silent non-interactive secure computation. In: ACM CCS 2019. ACM Press (November 2019)

[BCG+19b] Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators: silent OT extension and more. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11694, pp. 489–518. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26954-8_16

[BCG+20a] Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Correlated pseudorandom functions from variable-density LPN. In: 61st FOCS. IEEE Computer Society Press (November 2020)

[BCG+20b] Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators from ring-LPN. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020. LNCS, vol. 12171, pp. 387–416. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56880-1_14

[BCGI18] Boyle, E., Couteau, G., Gilboa, N., Ishai, Y.: Compressing vector OLE. In: ACM CCS 2018. ACM Press (October 2018)

[BGI+01] Barak, B., et al.: On the (im)possibility of obfuscating programs. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 1–18. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44647-8_1

[BGI+14a] Beimel, A., Gabizon, A., Ishai, Y., Kushilevitz, E., Meldgaard, S., Paskin-Cherniavsky, A.: Non-interactive secure multiparty computation. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 387–404. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44381-1_22

[BGI14b] Boyle, E., Goldwasser, S., Ivan, I.: Functional signatures and pseudorandom functions. In: Krawczyk, H. (ed.) PKC 2014. LNCS, vol. 8383, pp. 501–519. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54631-0_29

[BHR12] Bellare, M., Hoang, V.T., Rogaway, P.: Adaptively secure garbling with applications to one-time programs and secure outsourcing. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 134–153. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34961-4_10

[BP15] Bitansky, N., Paneth, O.: ZAPs and non-interactive witness indistinguishability from indistinguishability obfuscation. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015. LNCS, vol. 9015, pp. 401–427. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46497-7_16

[BW13] Boneh, D., Waters, B.: Constrained pseudorandom functions and their applications. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT 2013. LNCS, vol. 8270, pp. 280–300. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-42045-0_15

[Can01] Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: Proceedings of the 42nd FOCS. IEEE Computer Society Press (October 2001)

[CDI05] Cramer, R., Damgård, I., Ishai, Y.: Share conversion, pseudorandom secret-sharing and applications to secure computation. In: Kilian, J. (ed.) TCC 2005. LNCS, vol. 3378, pp. 342–362. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30576-7_19

[CLTV15] Canetti, R., Lin, H., Tessaro, S., Vaikuntanathan, V.: Obfuscation of probabilistic circuits and applications. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015. LNCS, vol. 9015, pp. 468–497. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46497-7_19

[DHRW16] Dodis, Y., Halevi, S., Rothblum, R.D., Wichs, D.: Spooky encryption and its applications. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. LNCS, vol. 9816, pp. 93–122. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53015-3_4

[FLS90] Feige, U., Lapidot, D., Shamir, A.: Multiple non-interactive zero knowledge proofs based on a single random string (extended abstract). In: Proceedings of the 31st FOCS. IEEE Computer Society Press (October 1990)

[GGH+13] Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B.: Candidate indistinguishability obfuscation and functional encryption for all circuits. In: Proceedings of the 54th FOCS. IEEE Computer Society Press (October 2013)

[GGM86] Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. J. ACM (4) (1986)

[GMW87] Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: Proceedings of the 19th ACM STOC. ACM Press (May 1987)

[GO07] Groth, J., Ostrovsky, R.: Cryptography in the multi-string model. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 323–341. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74143-5_18

[GOS06] Groth, J., Ostrovsky, R., Sahai, A.: Non-interactive Zaps and new techniques for NIZK. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 97–111. Springer, Heidelberg (2006). https://doi.org/10.1007/11818175_6

[GPSZ17] Garg, S., Pandey, O., Srinivasan, A., Zhandry, M.: Breaking the subexponential barrier in obfustopia. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10212, pp. 156–181. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56617-7_6

[HIJ+17] Halevi, S., Ishai, Y., Jain, A., Komargodski, I., Sahai, A., Yogev, E.: Non-interactive multiparty computation without correlated randomness. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017. LNCS, vol. 10626, pp. 181–211. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70700-6_7

[HJK+16] Hofheinz, D., Jager, T., Khurana, D., Sahai, A., Waters, B., Zhandry, M.: How to generate and use universal samplers. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016. LNCS, vol. 10032, pp. 715–744. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53890-6_24

[HW15] Hubacek, P., Wichs, D.: On the communication complexity of secure function evaluation with long output. In: Proceedings of the ITCS 2015. ACM (January 2015)

[JLS21] Jain, A., Lin, H., Sahai, A.: Indistinguishability obfuscation from well-founded assumptions. In: Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2021, pp. 60–73, New York, NY, USA. Association for Computing Machinery (2021)

[KPTZ13] Kiayias, A., Papadopoulos, S., Triandopoulos, N., Zacharias, T.: Delegatable pseudorandom functions and applications. In: Proceedings of the ACM CCS 2013. ACM Press (November 2013)

[KS08] Kolesnikov, V., Schneider, T.: Improved garbled circuit: free XOR gates and applications. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008. LNCS, vol. 5126, pp. 486–498. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70583-3_40

[LZ17] Liu, Q., Zhandry, M.: Decomposable obfuscation: a framework for building applications of obfuscation from polynomial hardness. In: Kalai, Y., Reyzin, L. (eds.) TCC 2017. LNCS, vol. 10677, pp. 138–169. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70500-2_6

[OSY21] Orlandi, C., Scholl, P., Yakoubov, S.: The rise of paillier: homomorphic secret sharing and public-key silent OT. In: Canteaut, A., Standaert, F.-X. (eds.) EUROCRYPT 2021. LNCS, vol. 12696, pp. 678–708. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-77870-5_24

[PS19] Peikert, C., Shiehian, S.: Noninteractive zero knowledge for NP from (plain) learning with errors. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11692, pp. 89–114. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26948-7_4