On the (Im)possibility of Distributed Samplers: Lower Bounds and Party-Dynamic Constructions

Damiano Abram[1], Maciej Obremski[2], and Peter Scholl[1]

[1] Aarhus University
{damiano.abram, peter.scholl}@cs.au.dk
[2] National University of Singapore
obremski.math@gmail.com

Abstract. Distributed samplers, introduced by Abram, Scholl and Yakoubov (Eurocrypt '22), are a one-round, multi-party protocol for securely sampling from any distribution. We give new lower and upper bounds for constructing distributed samplers in challenging scenarios. First, we consider the feasibility of distributed samplers with a malicious adversary in the standard model; the only previous construction in this setting relies on a random oracle. We show that for any UC-secure construction in the standard model, even with a CRS, the output of the sampling protocol must have low entropy. This essentially implies that this type of construction is useless in applications.
Secondly, we study the question of building distributed samplers in the party-dynamic setting, where parties can join in an ad-hoc manner, and the total number of parties is unbounded. Here, we obtain positive results. First, we build a special type of unbounded universal sampler, which after a trusted setup, allows sampling from any distributed with unbounded size. Our construction is in the *shared randomness model*, where the parties have access to a shared random string, and uses indistinguishability obfuscation and somewhere statistically binding hashing. Next, using our unbounded universal sampler, we construct distributed universal samplers in the party-dynamic setting. Our first construction satisfies one-time selective security in the shared randomness model. Our second construction is reusable and secure against a malicious adversary in the random oracle model. Finally, we show how to use party-dynamic, distributed universal samplers to produce ideal, correlated randomness in the party-dynamic setting, in a single round of interaction.

1 Introduction

Many cryptographic protocols require public parameters to be generated in a secure manner. This is the case, for instance, with trusted parameters used in many succinct zero-knowledge proofs [BCCT12], or trusted RSA moduli used in cryptographic accumulators [Bd94]. Using incorrectly or insecurely generated parameters in these settings can have devastating results, often completely breaking the desired security properties. As a result, when such parameters are needed,

the parties involved may wish to run a secure multi-party computation protocol to generate them, guaranteeing security as long at least one of the parties is honest. However, this type of setup protocol is typically expensive to carry out and coordinate.

Universal samplers, introduced by Hofheinz et al. [HJK+16], offer a partial solution to this problem. A universal sampler produces a single set of public parameters, which can later be used to securely sample from *any* distribution. In their strongest form, note that universal samplers are inherently tied to the random oracle model: in fact, they can be seen as a type of random oracle for sampling from arbitrary, structured distributions, without leaking the underlying random coins in the process.

A downside of universal samplers is that they still require a trusted setup, even if it only needs to be done once. Distributed samplers, recently introduced by Abram, Scholl and Yakoubov [ASY22, AWZ23], work around this issue by allowing parameters to be sampled using a secure multi-party protocol with *minimal interaction*. Each party publishes a single message, after which all parties can obtain a sample from the desired distribution. More formally, a distributed sampler for n parties and a distribution $\mathcal{D}$ is defined by a pair of algorithms $(\mathsf{Gen}, \mathsf{Sample})$, such that given a set of messages $U_i = \mathsf{Gen}(1\!\!1^\lambda, i)$, for $i \in [n]$, one can compute a sample $R \leftarrow \mathsf{Sample}(U_1, \ldots, U_n)$. The security requirement essentially states that this one-round protocol must securely realize the ideal functionality for sampling from $\mathcal{D}$, even when up to $n-1$ parties are corrupted.

The basic definition considers a 'one-time' or static setting, where there is a single distribution $\mathcal{D}$ that is fixed ahead of time, and the parties can only obtain a single sample from $\mathcal{D}$. This can be considered either with security against a passive adversary, or an active adversary. Active security is particularly challenging, due to the need to handle a rushing adversary, who may choose their messages U_i *after* seeing the messages U_j of the other parties. This allows an attacker to "grind" different choices of their randomness, obtaining different U_i, until finding an output R that she likes. So, the best form of security one can hope for in this setting is a relaxation of the ideal functionality for sampling, where the adversary first obtains several samples from $\mathcal{D}$, before settling on a final output. A stronger variety of distributed samplers is one that is reusable, for an unbounded number of queries. This is known as a distributed universal sampler. Similarly to the case of a (non-distributed) universal sampler, this is only possible to construct in the random oracle model.

Abram et al [ASY22] constructed distributed samplers in the plain model (no CRS) for any distribution based on indistinguishability obfuscation and multi-key fully homomorphic encryption. Their first construction is secure only against a *semi-malicious*[3] and non-rushing adversary. This was then upgraded to malicious security in the programmable random oracle model, with a construction that is also reusable, and secure for adaptive choices of the desired distributions. On top of this, they showed how distributed samplers can be used for sampling

[3] A semi-malicious adversary is one who follows the protocol, but may choose their random tape arbitrarily.

arbitrary forms of *correlated randomness*, often used in MPC protocols, with a one-round protocol.

We note that the constructions in [ASY22] are proven secure assuming that the underlying primitives are secure against polynomially bounded adversaries. This is in contrast to similar primitives like non-interactive MPC [HIJ+17] or probabilistic iO [CLTV15], for which the only general constructions are based on subexponentially secure primitives. This highlights that the setting of computing randomized functionalities, where no party has a private input, seems easier than that of general computations.

1.1 Our Results

In this work, we further explore the feasibility of distributed samplers, pushing their lower and upper limits with both impossibility results and more powerful constructions. We focus on security in the UC model [Can01], which gives strong composability guarantees.

Impossibility of Distributed Samplers Without Random Oracles. We first pose the question: is it possible to build actively secure distributed samplers in the *standard model*, that is, without random oracles? As a starting point, we observe that actively secure distributed samplers cannot be built without a common reference string (CRS) in the UC model. This is an immediate consequence of the UC impossibility for same-output probabilistic functions of [CKL03], since the function $\mathcal{D}(1\!\!1^\lambda)$ we want to compute has an unpredictable output and no inputs.

We observe also that generic actively secure distributed samplers without a CRS cannot exist even in the standalone model with black-box simulation. If that was not the case, by sequentially composing a distributed sampler with 2-round active OT protocols in the CRS model such as [PVW08] or [DGH+20], we would obtain a 3-round OT protocol with active security and black-box simulation in the plain model. The latter is known to be impossible [HV16].

For this reason, we investigate the CRS model. At first glance, it seems that distributed samplers are then trivial: the CRS can directly encode a sample from the desired distribution. This solution does not even need interaction. However, interactive distributed samplers with a CRS may have some advantages over the trivial construction, if the CRS can be reused multiple times and/or is easier to generate, either by being short or unstructured (i.e. a uniformly random string). We prove that if the construction is secure against rushing adversaries in the UC model, none of the above properties can be satisfied in the standard model.

All of these impossibilities come from our main result, below. Although the impossibility is in the UC model, we show that it even holds for a restricted class of adversaries who always follow the protocol, but behave in a rushing manner, sending their messages after receiving those of the honest parties. This only strengthens our impossibility result.

Theorem 1.1 (Informal, c.f. Thm. 4.1). *For any distributed sampler secure against rushing adversaries in the UC model, for a distribution $\mathcal{D}$ where $\mathsf{H}_\infty(\mathcal{D}) = \omega(\log \lambda)$, we have that $\mathsf{H}(R \mid \sigma) = O(\log \lambda)$, where σ denotes the CRS and R the output of the distributed sampler.*[4]

This essentially rules out this flavour of distributed sampler for all practical applications, as we discuss in the following corollaries.

Corollary 1: the collision probability is large. An immediate consequence of small Shannon entropy is that the output of the distributed sampler has a high probability of a collision if the CRS is not changed. This implies that in applications where more than one sample from $\mathcal{D}$ is needed, the same CRS cannot be reused.

Corollary 2: the CRS must be long. Less trivially, we show that this means that the CRS must be at most $O(\lambda)$ bits smaller than the Yao incompressibility entropy of $\mathcal{D}$. Recall that this roughly measures the compressed size of a sample from $\mathcal{D}$, after applying any efficient compression algorithm. As a result, the CRS must be almost as long as an output of $\mathcal{D}$, after applying compression.

Corollary 3: the CRS must be ugly. Finally, we show that in meaningful scenarios, the CRS must inherently be *structured*, or "ugly", meaning that it requires private coins to sample. In practice, this type of CRS must be generated by a trusted party or multi-party computation protocol, whereas obtaining a CRS that can be sampled from uniform randomness is much easier, relying only on a public source of randomness (or a hash function modelled as a random oracle).

Conclusion. Put together, the above corollaries show that UC-secure distributed samplers in the standard model, with rushing adversaries, are essentially useless. Since the CRS can only be used once, is structured, and as long as an output of $\mathcal{D}$, in practice it will most likely be no easier to generate the CRS than to just generate a sample from $\mathcal{D}$.

Open questions. Our main impossibility result is in the UC model, with polynomial-time simulation and dishonest majority. Recall that in this setting, rewinding is not allowed and simulation is inherently black-box[5]. We leave to future work the question of proving impossibilities — or finding constructions — for different settings, such as an honest majority, rewinding and non-black-box simulation.

[4] We use H_∞ to denote the min-entropy. We use H to denote the Shannon entropy. We refer to Appendix A.7 for formal definitions.

[5] Although the UC model allows the simulator to depend on the real-world adversary, the notion of security is still black-box. Indeed, it can be proven that a protocol is UC-secure if and only if it is secure against the "dummy adversary", who simply follows the instructions given by the environment [Can01]. By reframing the model in this way, we obtain a form of black-box simulation: security requires the existence of a single simulator that works for every environment.

Positive Results: Party-Dynamic Distributed Samplers. On the positive side, we give new results in settings where the parties have access to a random oracle, or in some cases, a public source of uniform randomness, called the *shared randomness model*. We construct *party-dynamic* distributed universal samplers, where the messages are independent of the distribution we want to sample from, the set of participants and their number, which is a priori unbounded. We analyse two notions of security. In one-time, semi-malicious security, the messages are used to generate a single sample, and the underlying distribution and set of parties are chosen ahead of time. With reusable, active security, the same messages are used to generate samples for multiple distributions and multiple subsets of participants, both adaptively chosen by the adversary. Distributed universal samplers, i.e. distributed samplers where the messages are independent of the distribution, were already built in [ASY22]. Prior to this work, however, all constructions were tailored to a specific set of players, which forced a restart of the protocol if participants joined or left.

Applications. Constructions supporting dynamic participants are ideally suited to non-interactive setup ceremonies for SNARKs in a permisionless setting, such as blockchains. More generally, they can be used for trusted setup in MPC protocols: imagine a world where every institution (e.g. governments, NGOs, intergovernmental organisations, private companies,...) publishes a distributed universal sampler message on a public bulletin board. Any set of parties that wants to run an MPC protocol can now non-interactively generate any CRS or correlated randomness[6] they want by just combining the sampler messages of the institutions they trust. The desired randomness is secure as long as just one of the participant's randomness is kept private. Furthermore, since our construction is party-dynamic, new organisations can join the protocol at any time without requiring further action from the others. Of course, the use of iO makes our solution currently impractical. However, we highlight that the task of obfuscating circuits is only required by the institutions (which likely have more resources); the parties just need to evaluate the resulting programs. In other words, for our solution to become practical, obfuscating does not need to be extremely efficient, what matters is the efficiency of the evaluation.

Our results. A key tool we introduce for our party-dynamic constructions is an *unbounded universal sampler*. Universal samplers are a way of securely sampling from any distribution, after a trusted setup phase which outputs some public parameters, called the *sampler parameters*. Previous constructions [HJK+16, LZ17] require the sampler parameters to be at least as large as the maximum size of the distribution. In the unbounded setting, we impose no such constraint: the circuit-size of the distribution can be arbitrarily large. Since the sample may be bigger than the sampler parameters, this inherently means that we need some additional source of randomness (such as the shared randomness model, or random oracle). An immediate application of unbounded universal samplers is to

[6] Using a party-dynamic distributed correlation sampler, discussed below.

compile any protocol with a large CRS into one with a small, reusable CRS in the random oracle model. This technique was recently applied in a subsequent manuscript[7] [ABI+23] to build a private simultaneous messages protocol with succinct public parameters and messages that have logarithmic size in the function input.

Theorem 1.2 (Informal). *Assuming polynomially secure iO and somewhere statistically binding hashing, there exist unbounded universal samplers in the shared randomness model.*

Using the unbounded universal sampler, we obtain the following.

Theorem 1.3 (Informal). *Assuming polynomially secure iO and multi-key FHE, there exist party-dynamic distributed universal samplers for any distribution, which are:*

- *One-time secure against a non-rushing, semi-malicious adversary, in the shared randomness model*
- *Reusable and secure against a malicious and static adversary, in the UC model with local random oracle (and assuming NIZK)*

Party-Dynamic, Distributed Correlation Samplers. As an application of our party-dynamic distributed samplers, we show how they can be used to obtain party-dynamic, distributed, universal *correlation samplers*, where after each party publishing a single, short, message, any subset of parties can obtain large amounts of correlated samples $R_1, \ldots, R_n$, defined by some arbitrary, correlated distributions (adaptively chosen after the messages are sent). Formally, we phrase this construction in the language of (public-key) pseudorandom correlation functions [BCG+20, ASY22].

Theorem 1.4 (Informal). *Assuming polynomially secure iO and multi-key FHE, there exist party-dynamic, public-key, pseudorandom universal correlation functions, for adaptively-chosen correlations in the UC model with local random oracle.*

Such primitive can be used, for instance, to build party-dynamic MPC with an information-theoretical online phase [DPSZ12, IKM+13, IOZ14] and non-interactive offline phase: when a party joins the protocol, it just needs to sent its public-key for the pseudorandom correlation function. After that, it can immediately join the online phase, without the other players' need to regenerate their pseudorandom correlation function keys.

Roadmap. In Section 2, we present a technical overview of our results and a discussion on related work. We describe notation and preliminaries in Section 3 and Appendix A. In Section 4, we formalise our lower bounds. We discuss succinct and unbounded universal samplers in Section 5. Finally, we present our party-dynamic constructions in Section 6.

[7] The paper is also in submission to TCC 2023. We are happy to provide a copy on request.

2 Technical Overview

We now give a high-level overview of the techniques used to obtain our results.

2.1 (Im)possibility of Distributed Samplers without Random Oracle

As we motivated in the introduction, actively secure distributed samplers in the plain model with black-box simulation are impossible. In the CRS model, instead, they are trivial to build: the CRS can directly encode a sample from the underlying distribution. The result is a distributed sampler in which the parties do not even need to communicate, since they just output the CRS.

We study how interactive constructions can improve upon the trivial solution. In principle, the advantages can be multiple: the same CRS can be reused in many distributed sampler executions producing independent-looking outputs. Moreover, the CRS of distributed samplers can be nicer (i.e. easier to generate) than the direct encoding of a sample, for instance because of the smaller size, or because it is unstructured (i.e. a uniformly random string of bits). The result of our analysis is that none of the above properties can be satisfied: without a random oracle, distributed samplers essentially provide no advantage over the trivial solution. In order for this impossibility to hold, we do not even need to aim for active security, it suffices that the adversary is *strongly semi-malicious*: it may adaptively choose the randomness of the corrupted parties after seeing the honest messages, but all corrupted players follow the protocol.

On the Unpredictability of Distributed Samplers in the CRS Model. All the negative results mentioned above are consequences of the main theorem of this work: in a strongly semi-malicious distributed sampler, where the underlying distribution $\mathcal{D}$ has high min-entropy, namely $\mathsf{H}_\infty(\mathcal{D}) = \omega(\log \lambda)$, the Shannon entropy of the output conditioned on the CRS is $O(\log \lambda)$.

All through the paper we carefully juggle different variants of entropy, each bringing a unique set of properties we require during the proofs. Shannon entropy H has a powerful chain rule. Collision entropy H_2 gives us an elegant tool for building distinguishers, but lacks a chain rule and is not invariant under computational indistinguishability (i.e. for two computationally indistinguishable random variables, H_2 can be vastly different). We also use min-entropy H_∞, this is the smallest of the above mentioned and has the fewest properties. Our assumption on the entropy of the random source $\mathcal{D}$ is $\mathsf{H}_\infty(\mathcal{D}) = \omega(\log \lambda)$ (clearly the task is trivial if $\mathcal{D}$ is constant) – this becomes the weakest assumption one can make using any of the above notions (and thus makes our theorem stronger). Finally, Yao's entropy is the only entropy we use that remains invariant under computational indistinguishability (i.e. two computationally indistinguishable random variables have the same Yao entropy). For formal definitions please refer to Appendix A.7.

Distributed samplers against a rushing adversary. In order to understand the idea behind the result, we need to recall the definition of distributed samplers with security against an active adversary [ASY22]. The corresponding functionality provides the adversary with as many samples from the underlying distribution as the adversary wants. The adversary can then select one of these values; the functionality outputs it to all the honest parties. This kind of behaviour is needed to model the fact that, in the case of a rushing adversary, the corrupted parties see the honest messages before they publish their own. In other words, before committing to a choice, they can always test their candidate messages and discard them if they are not happy.

Definition 2.1 (Distributed sampler - security against rushing adversaries). *Let $\mathcal{D}(1\!\!1^\lambda)$ be an efficiently samplable distribution. An n-party actively secure (resp. strongly semi-maliciously secure) distributed sampler for $\mathcal{D}(1\!\!1^\lambda)$ is a one-round protocol implementing the functionality $\mathcal{F}_\mathcal{D}$ (see Fig. 1) against a static and active (resp. strongly semi-malicious) adversary corrupting up to $n-1$ parties.*

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

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 $R \xleftarrow{\$} \mathcal{D}(1\!\!1^\lambda)$ to every honest party and sends R to the adversary, then it halts.

Query. On input Query from the adversary, the functionality samples $R \xleftarrow{\$} \mathcal{D}(1\!\!1^\lambda)$ 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 $\mathsf{id} = \widehat{\mathsf{id}}$. Then, it outputs R to every honest party and terminates.

Fig. 1: The distributed sampler functionality for rushing adversaries

The security model. We consider the UC model against the "dummy adversary", the one that simply follows the instructions given by the environment. We recall that a protocol is UC-secure if and only if it is secure against the dummy adversary [Can01]. In this setting, there exists a unique simulator that works for every environment. Since the role of the adversary is essentially assumed by the environment, we will use the terms adversary and environment interchangeably. We work in the dishonest majority setting.

Our proof will only consider adversaries that behave honestly, i.e. they choose the randomness of the corrupted parties uniformly at random and they follow the protocol. Notice that since we are proving a lower bound, considering very weak adversaries such as the honest one makes our results even stronger.

In the ideal world, the outputs are restricted to a small set. The simulator of the distributed sampler needs to provide the honest parties' messages and the CRS to the adversary before learning the choices of the corrupted players. Since the simulator runs in polynomial time, the number of samples received from the functionality before the delivery is polynomially bounded. Let the corresponding set be Q.

Once the adversary supplies the corrupted messages, the output of the protocol is fixed (indeed, we cannot rewind the adversary, as the UC model does not allow it). If the latter belongs to Q, the simulator can easily instruct the functionality to output the right sample to all honest players. If instead that is not the case, the only choice left for the simulator is to keep querying the functionality for new samples and hope for a collision. Since the distribution has high min-entropy, this occurs with negligible probability. In other words, the output must belong to Q with overwhelming probability. If that does not happen, the adversary can easily distinguish the real protocol from the ideal world as the simulator is not able to make the honest parties output the right result.

In the real world, the output is easily predictable from the CRS and the messages of the honest parties. Let R denote the output of the distributed sampler, let σ be the CRS and let U_H and U_C denote the messages of the honest and the corrupted parties respectively. The fact that the CRS and the messages of the honest parties restrict the output is a set of polynomial size is a strong property. In particular, the latter implies that $\mathsf{H}(R|\sigma, U_H) = O(\log \lambda)$. This equality holds in the ideal world, but what about the real world? Unfortunately, Shannon's entropy does not behave well under computational indistinguishability, i.e. computationally indistinguishable random variables may have very different entropy. We prove, however, that if the adversary honestly follows the protocol in the real world, $\mathsf{H}(R|\,\sigma, U_H) = O(\log \lambda)$.

Consider the distinguisher that, after receiving the CRS and U_H, keeps regenerating the messages of the corrupted parties following the protocol, and stores the outputs obtained in this way. In the ideal world, the distinguisher will never obtain more than $q(\lambda)$ different samples, where $q(\lambda)$ is a polynomial upper-bound on the cardinality of Q, the set of values queried by the simulator to the functionality. We notice that without loss of generality $q(\lambda)$ is known to the distinguisher as the simulator is fixed.

Using a technical argument based on entropy, we show that if $\mathsf{H}(R|\,\sigma, U_H)$ is not $O(\log \lambda)$, in the real world, there exists a non-negligible function $\delta(\lambda)$ such that for every polynomial $j(\lambda)$, the j-th output obtained by the distinguisher differs from all the previous ones with probability at least $\delta(\lambda)$. The crucial point is that $\delta(\lambda)$ is independent of j. Indeed, as j increases, the probability of obtaining new outputs becomes lower (the probability of colliding with one of the previous outcomes gets higher and higher). If this probability decreases too fast, the number of different outputs obtained by the distinguisher may converge to a certain threshold smaller than $q(\lambda)$. Since the probability is always bounded from below by $\delta(\lambda)$, however, in the real world, the distinguisher is able to

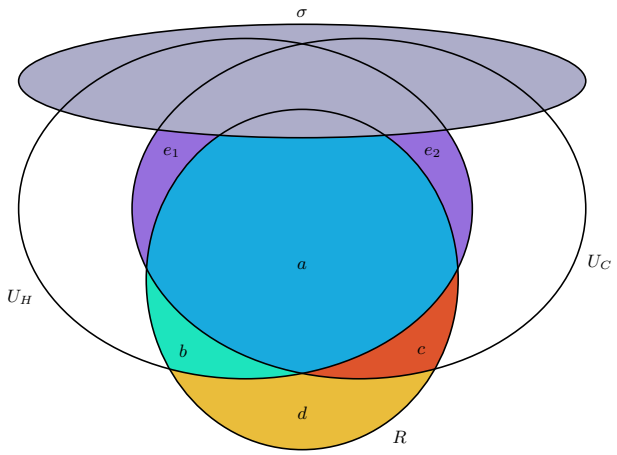

Fig. 2: Entropy diagram of the distributed sampler.

obtain more than $q(\lambda)$ different outputs in a polynomial number of steps. This is sufficient to break security of distributed samplers.

The final result: an easy application of the strong chain rule. At this point, proving our theorem becomes simple. Since we are considering an honest adversary, the result described in the previous paragraph immediately implies that $\mathsf{H}(R|\,\sigma, U_C)$ is also $O(\log \lambda)$. Furthermore, U_H is independent of U_C, given the CRS. In other words, $\mathsf{H}(U_H|\sigma) = \mathsf{H}(U_H|\sigma, U_C)$. By a simple application of the strong chain rule for Shannon's entropy, it is easy to show that $\mathsf{H}(R|\,\sigma) = O(\log \lambda)$.

Indeed, consider the entropy diagram in Fig. 2.[8] Observe that $\mathsf{H}(R|\sigma)$ corresponds to the union of the blue, red, green and yellow areas, i.e. $\mathsf{H}(R|\sigma) = a+b+c+d$. We know that $\mathsf{H}(R|\sigma, U_H)$ corresponds to the union of the red and yellow areas, so, $c + d = \mathsf{H}(R|\sigma, U_H)$. Similarly, $\mathsf{H}(R|\sigma, U_C)$ corresponds to the union of the green and yellow areas, so, $b + d = \mathsf{H}(R|\sigma, U_C)$. We also observe that the union of the blue and purple areas correspond to $\mathsf{H}(U_H|\sigma) - \mathsf{H}(U_H|\sigma, U_C) = 0$, so $a + e_1 + e_2 = 0$. Finally, we notice that both $e_1 + e_2$ and d are non-negative. Indeed, the former corresponds to $\mathsf{H}(U_H|\sigma, R) - \mathsf{H}(U_H|\sigma, R, U_C) \geq 0$, whereas the latter corresponds to $\mathsf{H}(R|\sigma, U_H, U_C) \geq 0$. The fact that $e_1 + e_2 \geq 0$ also implies that $a \leq 0$, so

$$\mathsf{H}(R|\sigma) = a + b + c + d \leq b + c + 2d = \mathsf{H}(R|\sigma, U_H) + \mathsf{H}(R|\sigma, U_C) = O(\log \lambda).$$

Bad News for Distributed Samplers. All the results we discuss below hold in absence of a random oracle and for distributed samplers that achieve UC-security against a strongly semi-malicious adversary.

[8] The diagram is not completely general as some of the intersections between the sets are empty, however, the figure is sufficiently generic to describe our argument.

Distributed sampler CRSs cannot be used twice. The first corollary of Theorem 1.1 is that two distributed sampler executions using the same CRS have colliding outputs with non-negligible probability. We recall that our theorem applies when the min-entropy of the underlying distribution is high, i.e. $\omega(\log \lambda)$. For all such distributions, the collision probability is negligible, i.e. two independent samples from $\mathcal{D}(\mathbb{1}^\lambda)$ will almost always be different. As a consequence, by reusing the same CRS twice, we obtain samples that do not look independent.

The reason at the base of our first corollary is that, by a simple application of Jensen's inequality, the average collision entropy $\widetilde{\mathsf{H}}_2(R|\,\sigma)$ is bounded from above by $\mathsf{H}(R|\,\sigma) = O(\log \lambda)$. We recall that the average collision entropy is defined as

$$\widetilde{\mathsf{H}}_2(R|\,\sigma) := -\log\big(\Pr[R = R']\big)$$

where R and R' are two distributed sampler outputs computed using the same CRS σ and the probability is also over the randomness of σ. We conclude that $\Pr[R = R'] \geq 1/\mathsf{poly}(\lambda)$.

Distributed sampler CRSs are long. We prove that CRSs of strongly semi-malicious distributed samplers cannot be small: they can be at most $O(\log \lambda)$ bits shorter than the Yao entropy of the underlying distribution $\mathsf{H}_{\mathsf{Yao}}(\mathcal{D})$[9].

We prove this result by first observing that $\mathsf{H}_{\mathsf{Yao}}(R|\,\sigma) = O(\log \lambda)$. Indeed, as we motivated in the previous paragraph, two distributed sampler executions using the same CRS have colliding outputs with non-negligible probability. We can therefore consider the Yao's compressor that outputs nothing and the associated decompressor that, provided with the CRS σ, reruns the distributed sampler protocol in its head and outputs the result R'. With $1/\mathsf{poly}(\lambda)$ probability, R' coincides with the input of the compressor.[10] This is enough to conclude that $\mathsf{H}_{\mathsf{Yao}}(R|\,\sigma) = O(\log \lambda)$.

We then show that $\mathsf{H}_{\mathsf{Yao}}(R|\,\sigma) \geq \mathsf{H}_{\mathsf{Yao}}(R) - |\sigma|$. We prove this by noticing that, given a compressor-decompressor pair (c', d') for $\mathsf{H}_{\mathsf{Yao}}(R|\,\sigma)$, we can build a compressor-decompressor pair (c, d) for $\mathsf{H}_{\mathsf{Yao}}(R)$ as follows: c provides its input R to the distributed sampler simulator, corrupting no party. It obtains a fake CRS σ' that looks like the real one. It then outputs $c'(R, \sigma')$ along with σ'. The decompressor d is exactly the same as d'. The success probability of (c, d) is the same as for (c', d') except for a negligible quantity. The size of the compressed string has however grown by $|\sigma|$ bits, increasing $\mathsf{H}_{\mathsf{Yao}}(R)$ by the same amount.

We point out that, in order to prove the above inequality, we cannot use the Yao chain rule of [KPW13, Appendix B] as their compressor for $\mathsf{H}_{\mathsf{Yao}}(R)$ has $O(2^{|\sigma|})$ size.

Distributed sampler CRSs are ugly. Suppose that there exists a strongly semi-malicious distributed sampler for the distribution $\mathcal{D}$ having CRS σ. We prove that it is possible to *non-interactively* and *securely* generate a sample from $\mathcal{D}$

[9] The Yao entropy of $\mathcal{D}$ roughly measures how much a sample from $\mathcal{D}$ can be compressed in polynomial time without losing information.

[10] We can make the decompressor deterministic using a PRF.

given only σ and public random coins. In other words, if there exists a distributed sampler with nice CRS, also the underlying distribution can be encoded in a nice CRS. The second solution may be preferable as it often requires less communication. As an additional corollary, if the distributed sampler uses a URS (i.e. the CRS is a random string of bits), we can sample from $\mathcal{D}$ using just public random coins. So, in the random oracle model, we would not even need a CRS.

Our idea is that, given σ and public random coins, each party can just rerun the distributed sampler protocol with σ as CRS and the public coins as randomness for the players. The result R is clearly indistinguishable from a sample from $\mathcal{D}$. However, in order to prove that this protocol is secure, we need to be able to simulate σ and the public coins, given R.

We simulate σ by feeding R to the distributed sampler simulator (we corrupt no party). Unfortunately, the simulator cannot provide us with the randomness used by the parties. We proceed by brute-force: we rerun the protocol in our head using the fake CRS and we hope that the output collides with R. If we fail, we retry sampling a new fake CRS. Once we succeed, we output the fake σ and the randomness of the parties that led to the collision.

By the first corollary of Theorem 1.1, we know that, *on average* over R, the collision probability is $1/\mathsf{poly}(\lambda)$. So, for a *polynomial fraction* of all possible values R, the simulation succeeds after a polynomial number of tries. For the remaining fraction of the support of $\mathcal{D}$, our approach fails, meaning that the CRS and the randomness of the parties might leak too much information about the output.

In other words, the sampling protocol we described is secure only for a polynomial fraction of the support of $\mathcal{D}$. The good news is that it is possible to tell if the result of our non-interactive sampling protocol lies in the secure subset or not: the parties can locally run the simulator. If it succeeds with sufficiently high frequency, they can be sure their output is secure, otherwise, they need to discard it, generate a new σ and public coins and rerun the protocol. Since there is a polynomial fraction of the support of $\mathcal{D}$ that will not be discarded, the players need a polynomial number of attempts before succeeding. We also point out that the distribution of the outputs will be biased, but not significantly: if $\mathcal{D}$ describes the distribution of another protocol's CRS, it is still secure to use the outputs of our procedure as CRSs for such protocol.

How General is the Impossibility? Our arguments seem to apply not only to the UC model but also to the more powerful settings of security with superpolynomial simulation, and standalone security with rewinding. Informally, what Theorem 1.1 is saying is that the size of the distributed sampler messages sets an information-theoretic bound B on the number of samples that a simulator can encode in the messages it produces. An adversary can rerun the distributed sampler protocol in its head a number of times that is significantly larger than B. In the real world, it is supposed to obtain more than B distinct outputs, on the other hand, in the ideal world, this does not happen. This suggests that the impossibility holds even if we rely on superpolynomial simulation.

Even rewinding does not seems to help: in the ideal world, with high probability, the output R of the distributed sampler has non-negligible probability of being resampled (i.e., if the distinguisher reruns the protocol in its head, regenerating the messages of the corrupted players, it has a high chance of reobtaining R after a few tries). This is because R was the result of the rewinding process. If R had a low probability of being resampled, the probability that rewinding output R would have been low in the first place. On the other hand, in the real world, R has very low probability of being resampled (we want $\mathsf{H}(R|\sigma, U_H) = \omega(\log \lambda)$, otherwise, we rerun into the problems of the UC model). This leads to a successful attack. Whether this ideas can be formalised will be part of future work.

2.2 Constructing Unbounded Universal Samplers

Our first positive result is a construction of an unbounded universal sampler in the shared randomness model. Recall that in a universal sampler (US), the trusted setup algorithm outputs some sampler parameters U, which are later used to securely sample from a distribution $\mathcal{D}$. Our goal is to ensure that the size of the circuit that samples from $\mathcal{D}$ may be unbounded, and in particular, independent of U.

Succinct, Bounded Universal Samplers. We start by building a US that is not totally unbounded, but is succinct, meaning that the size of U is only polylogarithmic in the maximum circuit size L of the supported distribution $\mathcal{D}$. To see the challenge in achieving this, recall that the sampler parameters in the selective, one-time universal sampler by Hofheinz *et al.* [HJK+16] consist of an obfuscated program. To sample from a distribution $\mathcal{D}$, the program is fed with the circuit describing $\mathcal{D}$. It then uses a puncturable PRF to generate random bits used to sample from $\mathcal{D}$ and outputs the result. If we want to obtain succinctness then there is no way the obfuscated program can evaluate the sampling circuit, which may now be significantly larger than the sampler parameters. Therefore, we cannot even provide $\mathcal{D}$ as input to the program, let alone evaluate it.

Taking advantage of the locality of garbled circuits. Our solution is to use garbled circuits. We obfuscate a program $\mathsf{SUSProg}$, which, instead of evaluating $\mathcal{D}$ itself, will output a garbling of $\mathcal{D}$ along with one random label for each input wire and both labels for each output wire. At any point in time, a party can evaluate the garbled circuit produced by $\mathsf{SUSProg}$ obtaining a sample from $\mathcal{D}$.

The big advantage of garbled circuits is its locality: as long as there is way to retrieve the labels associated with the input and output wires of any gate g, we can garble g without knowing the whole circuit to which g belongs. Specifically, each execution of $\mathsf{SUSProg}$ takes as input a single gate of $\mathcal{D}$ and outputs its garbling. The description of the gate will consists of a type (input, output, XOR or AND) and identifiers for the input and output wires of the gate[11]. Since the

[11] Notice that the terminology distinguishes between input gate and input wire of a gate. The first one is used to denote an input to the *circuit*, the second one is used

THE PROGRAM $\mathcal{P}_{\mathsf{SUS}}[K, \mathsf{hk}]$

Hardcoded: A PRF key K and an SSB hash key hk.
Input: SSB hash z of $\mathcal{D}$, index i, gate g and SSB proof π.

1. If $\mathsf{SSB.Verify}\big(\mathsf{hk}, z, i, g, \pi\big) = 0$, output $\perp$.
2. $\mathsf{gk} \leftarrow F_1\big(K, z\big)$
3. Output $\mathsf{Garble}(\mathbb{1}^\lambda, g, \mathsf{gk})$

Fig. 3: Warm-up attempt for the unobfuscated SUS program

operations SUSProg needs to perform are now independent of $\mathcal{D}$, the size of SUSProg can remain small. A similar idea was adopted by Garg and Srinivasan for the construction of obfuscation for Turing machines [GS18].

Making the garbled gates consistent. The first problem is that we need to ensure that different gates are garbled consistently, in that whenever a wire of the circuit is re-used, the same wire labels are used. As a consequence, all the executions of SUSProg associated with $\mathcal{D}$ cannot be independent, they all need to have access to some common information.

To ensure this, we use a master garbling key gk to derive, using a PRF F, the randomness needed by the garbling and the random bits given as input to $\mathcal{D}$. Formally, the labels associated with a wire w will be $(k_w^0, k_w^1) \leftarrow F(\mathsf{gk}, w)$. For each input gate g, we also use F to sample a random input bit, and give out the corresponding wire label. For each XOR or AND gate g, we additionally use F to sample a permutation to reorder the ciphertexts. For each output gate, we provide both wire labels.

We observe that every execution of SUSProg associated with $\mathcal{D}$ needs to retrieve the same key gk. Furthermore, different distributions $\mathcal{D}$ and $\mathcal{D}'$ need to use independent-looking garbling keys. If that is not the case, we risk garbling different circuits using the same labels, which would compromise privacy.

We solve these issues by providing SUSProg also with a hash z of the circuit $\mathcal{D}$. Since the size of z is $O(\log L)$, we can input it to SUSProg without any troubles. The obfuscated program SUSProg will be equipped with a puncturable PRF F_1 and a key K. Using z as input for F_1, SUSProg will retrieve gk and use the latter to garble the provided gate. By the collision resistance of the hash function, different distributions will correspond to different hashes and so, by the security of the puncturable PRF, to independent-looking garbling keys. To make this argument compatible with indistinguishability obfuscation, we use a somewhere statistically binding (SSB) hash function [HW15].

to denote the input to a *gate*, i.e. the bit to which we apply an XOR or an AND. A similar discussion applies to output gates and output wires.

Limiting the leakage using SSB hashing. So far, nothing prevents the adversary from garbling a circuit using SUSProg while providing an inconsistent digest z. This means that the adversary can retrieve the randomness used to produce the sample from $\mathcal{D}$ by simply garbling the identity function along with $z = \mathsf{Hash}(\mathcal{D})$.

Luckily, SSB hash functions help us in countering this attack. Indeed, SSB hashing can be used to prove that a certain gate g is the i-th element in the preimage of z. So, if we provide the proof along with z, g, i and the SSB hash key hk, the obfuscated program is able to check if g really is the i-th gate of $\mathcal{D}$. If the verification succeeds, the program can garble g using gk, otherwise, it can simply output $\perp$.

SSB hash functions set an upper bound on the length of the messages that can be hashed. In our construction, we set this to $L(\lambda)$ blocks[12]. A nice feature of some SSB hashing schemes [HW15] is that both the hash key and the SSB proofs have size $O(\log L)$. Furthermore, the proofs can also be verified in $O(\log L)$ time. In other words, verifying the proofs in the code of SUSProg does not blow up the size of the program.

We present the construction so far in the program shown in Fig. 3. To summarise, the adversary can make SUSProg output only the garbling of $\mathcal{D}$ or independent-looking information. Indeed, any execution inputting a hash other than z would lead to an independent-looking garbling key and hence, independent-looking information. If instead z is input, all the adversary can receive is the garbled gates of $\mathcal{D}$. If it tries to provide a different gate, the hash check will fail.

Taking control over the outputs with a trapdoor. To prove security, we need to argue that our program reveals no information in addition to the output of the garbled circuit. This is formalised by saying that we can simulate SUSProg given a sample R from $\mathcal{D}$. Clearly, the simulated SUSProg needs to output R when run on $\mathcal{D}$. Unfortunately, our obfuscated program cannot satisfy this property in the current state. Indeed, the sample R may contain significantly more information than the size of SUSProg.

Here, we rely on the shared randomness model, where we require any party obtaining a sample to have a long, uniform string $\boldsymbol{u}$. Using $\boldsymbol{u}$, we equip SUSProg with a trapdoor that allows us to program its output in the security proof; we do this using the delayed backdoor programming technique from the adaptive universal sampler in [HJK+16], also used in the malicious constructions of [ASY22]. To garble the i-th gate g_i, we provide SUSProg with a u_i, corresponding to the i-th block of the randomness $\boldsymbol{u}$. We hardcode into our program an additional key k for a special kind of authenticated encryption scheme. In each execution, after verifying the SSB proofs, SUSProg tries to decrypt u_i using k. If decryption succeeds, the program outputs the underlying plaintext, otherwise it resumes its usual behaviour, i.e. it garbles the provided gate.

The encryption scheme, which is based on puncturable PRFs, is designed so that ciphertexts are indistinguishable from random strings, but the overwhelm-

[12] Each block will be the description of a different gate.

ing majority of strings are not valid ciphertexts. When a random u_i is input into $\mathsf{SUSProg}$, then, the probability of activating the trapdoor is negligible. In the simulation, however, $\boldsymbol{u}$ will be the encryption of a garbled circuit simulated using R and $\mathcal{D}$. By the security of iO, the adversary will not be able to tell if the output is generated using the trapdoor or the standard procedure.

Binding the trapdoor to the distribution. Finally, there is one weakness remaining in the construction: we need to bind the random string $\boldsymbol{u}$ to the distribution $\mathcal{D}$. At the moment, the adversary can easily tell if $\boldsymbol{u}$ hides the encryption of a random circuit or not. It can simply garble $\mathcal{D}$ twice, once using $\boldsymbol{u}$ and once inputting a random string. If the outputs differ, it must be that $\boldsymbol{u}$ activates the trapdoor.

Clearly, we cannot prevent the adversary from choosing the distribution and the random string as it pleases, however, we can make sure that for different choices of $(\mathcal{D}, \boldsymbol{u})$, we obtain independent-looking executions. Specifically, instead of equipping $\mathsf{SUSProg}$ with a hardcoded trapdoor key k, we generate k along with gk using the PRF F_1. Recall that the input given to F_1 is a hash of $\mathcal{D}$. In this way, different distributions would use different trapdoor keys and so $\boldsymbol{u}$ would activate the trapdoor only in conjunction with $\mathcal{D}$.

Finally, we also want to ensure that when given different random strings, the garbled circuit output by $\mathsf{SUSProg}$ changes. That corresponds to having a different garbling key gk. To ensure this, in each execution, we provide $\mathsf{SUSProg}$ also with an SSB hash h of $\boldsymbol{u}$. We then input h into the puncturable PRF F_1 along with z. In conclusion, we obtain a different garbling key and a different trapdoor key for every choice of distribution and random string. To ensure that the string u_i input to the program is consistent with the hash h, we additionally modify $\mathsf{SUSProg}$ to receive an SSB proof that u_i is the i-th block of the preimage of h. The program checks the proof and outputs the garbled gate only if the verification succeeds. Otherwise, it outputs $\perp$.

To summarise, if the adversary does not input (h, z) into $\mathsf{SUSProg}$, the program outputs information that looks independent of the sample R. If it inputs (z, h) instead, the adversary is forced to provide a pair (g_i, u_i) for a certain $i \in [L]$ where g_i denotes the i-th gate of $\mathcal{D}$. If this is the case, the adversary receives the scheduled garbling of g_i, otherwise, it receives $\perp$.

We present an informal description of the final version of the program in Fig. 4. For the complete, formal construction and its security proof, we refer to Section 5.1.

From Succinct to Unbounded Universal Samplers. Once we have a succinct, but bounded, US, it is quite straightforward to obtain an unbounded US. Our construction will simply run the setup procedure from the succinct US, and output its sampler parameters U. This already allows us to sample from any distribution $\mathcal{D}$ up to some polynomial bound. To sample from a larger $\mathcal{D}$, we simply use U to run the setup algorithm for a second succinct US, with a bound of twice the size (since the first US was succinct, this will always be possible for a sufficiently large security parameter). This process is then iterated until we have a sampler that can support the distribution $\mathcal{D}$.

THE PROGRAM $\mathcal{P}_{\mathsf{SUS}}[K, \mathsf{hk}]$

Hardcoded: A PRF key K and an SSB hash key hk.
Input: SSB hashes h and z of $\boldsymbol{u}$ and $\mathcal{D}$ respectively, index $i \in [L]$, random string v, gate g and SSB proofs π and π'.

1. $b \leftarrow \mathsf{SSB.Verify}\big(\mathsf{hk}, h, i, v, \pi\big)$
2. $b' \leftarrow \mathsf{SSB.Verify}\big(\mathsf{hk}, z, i, g, \pi'\big)$
3. If $b = 0$ or $b' = 0$, output $\perp$.
4. $(\mathsf{gk}, k) \leftarrow F_1\big(K, (h, z)\big)$
5. $x \leftarrow \mathsf{Dec}(k, v)$
6. If $x \neq \perp$, output x.
7. Otherwise, output $\mathsf{Garble}(\mathbb{1}^\lambda, g, \mathsf{gk})$

Fig. 4: Informal description of the unobfuscated SUS program

For technical reasons, to prove this construction secure we need an additional property of the unbounded US, which we call randomness extractability. Intuitively, this says that given a sampler output R and the randomness that was used to compute the sampler parameters, it is possible to extract the randomness that "explains" the output R from distribution $\mathcal{D}$. We show that this property holds for our construction, and in fact is easily achievable in a generic way for any universal sampler.

2.3 Building Unbounded and Party-Dynamic, Distributed Universal Samplers

Our next goal is to obtain unbounded *distributed universal samplers*, where the sampler is derived from n messages, one from each out of a set of n parties. As well as allowing the choice of distribution $\mathcal{D}$ to be unbounded, and not tied to the sampler parameters, here we also want the sampler to be *party-dynamic*, so the set of parties can be chosen dynamically from an unbounded set of possible parties.

A toy construction of a *bounded*, party-dynamic distributed sampler can be easily obtained from any n-party distributed sampler for a fixed number of parties: each party simply runs the i-party distributed sampler algorithm, for $i = 2, \ldots, n$, and publishes all the $n - 1$ messages. Of course, this construction requires the size of each message to scale at least linearly with n.

To get an unbounded construction, we modify this blueprint by instead having each party publish a single message consisting of an unbounded universal sampler. Later, to sample from a distribution with some size-n subset of the parties, those parties' unbounded USes will each be used to generate an n-party distributed sampler message on-the-fly. Since we use an unbounded US, this construction is inherently tied to the shared randomness model, where the subset

of n parties must all hold a common string of uniform bits to obtain their sample. We prove security in the one-time setting, against a non-rushing and semi-malicious adversary.

Modelling Active Security. In the non-rushing setting, modelling security is quite straightforward and similar to the case of non-party-dynamic definitions. When moving to an active adversary, however, we have to be careful how to define security. Recall that with a static of parties, active security of a distributed sampler is defined using an ideal functionality, which allows the adversary to obtain several samples from the distribution $\mathcal{D}$, before settling on one it likes. This corresponds to the fact that in a construction, every choice of a corrupt party's randomness may lead to a different result from $\mathcal{D}$.

In the party-dynamic setting, we consider a static adversary in the UC model: whenever a new party joins the system, the adversary must decide whether that party is corrupted or not. At the same time, we need a way to model the fact that the adversary can try candidate messages of a corrupt party P_j, obtaining different samples, before P_j has actually joined the system. To do this, we allow the adversary to input a label id_j, corresponding to a new choice of message for P_j, and can then obtain a sample for the desired subset of parties that includes P_j. When P_j eventually joins, the adversary can either choose one of the previously sent labels, "fixing" the relevant outputs to the corresponding samples, or choose a fresh label which leads to freshly sampled outputs.

Achieving Active Security and Reusability. Next, we upgrade our construction to be actively secure, and also reusable for an unbounded number of queries on arbitrary distributions. We do this in a black-box way, starting from any one-time secure, party-dynamic construction. The main idea is to have each party publish an *adaptive* (or reusable), bounded universal sampler [HJK+16] as its message, together with a NIZK showing that it is well-formed. Then, whenever a subset of parties wants to obtain a sample, the adaptive US is used to generate a message for a one-time, party-dynamic distributed US. By relying on a reusable US, we ensure that each message from the one-time, party-dynamic construction is only used once. Recall that our one-time, party-dynamic construction requires a source of public, shared randomness $\boldsymbol{u}$ to obtain the sample; to generate $\boldsymbol{u}$ in a reusable way, we use a random oracle.

There is still one problem with this approach, though. An adversary may still adaptively choose the messages of the corrupt parties, and the distribution $\mathcal{D}$, *after* seeing the honest parties' messages from the one-time, party-dynamic distributed US (which is not secure against a rushing adversary). To fix this, we again rely on the random oracle model. We force the adversary to commit to its messages before seeing these messages, by making it query the random oracle with input the subset of parties, distribution $\mathcal{D}$, and adaptive universal sampler messages. The output of the random oracle is a λ-bit tag, which is fed into the adaptive US before generating the one-time messages. Since the tag is unpredictable, this ensures that the adversary cannot learn any outputs without first committing to its messages.

Party-Dynamic, Public-Key Pseudorandom Universal Correlation Functions (PCFs). Our last construction is an application of party-dynamic distributed universal samplers, for generating correlated randomness. A public-key PCF [BCG+20, ASY22] is a one-round protocol for securely sampling from n correlated random variables, where each party obtains one of the outputs, while learning nothing of the other parties' outputs. We show how to build public-key PCFs in the party-dynamic setting where the correlation is adaptively chosen after the messages of the parties are sent. Our construction is quite simple, and follows the blueprint of the previous construction for a fixed number of parties [ASY22]: each party sends a public key for a PKE scheme, plus a message for a distributed universal sampler. The distributed universal sampler messages are then used to sample from the distribution that encrypts the n outputs of the correlation function under each of the parties' public keys, allowing only the correct party to recover its output. By relying on our party-dynamic distributed universal sampler, we immediately obtain a public-key, universal PCF in the party-dynamic setting.

We present the construction directly in the actively secure and reusable setting (in the random oracle model). Because of this, we achieve the stronger notion of an ideal public-key PCF, which securely realizes the ideal sampling functionality (with suitable relaxations to account for rushing adversaries). In contrast, without a random oracle, this type of PCF is impossible to achieve, unless one allows the parties' messages to be as long as the total output length of all queries to the correlation.

2.4 Related Work

Non-interactive key exchange. The setting of party-dynamic distributed samplers is similar to unbounded non-interactive key exchange (NIKE), which can be built using iO [KRS15]. NIKE is in some way similar to a distributed sampler for the uniform distribution, but it satisfies a weaker security definition: the output of the NIKE is guaranteed to look random only if no party is corrupted. This implies, for instance, that the derived output may depend only on the randomness of one party. Distributed samplers instead achieve security even when the adversary takes part in the computation. This difference allows NIKE to avoid many issues related to entropy.

One-round MPC. Distributed samplers can also be viewed as an inputless version of non-interactive MPC [HIJ+17]. We recall that non-interactive MPC unavoidably achieves a weak definition of security in which the adversary is allowed to learn the residual function (i.e. the function obtained by fixing the inputs of the honest parties while leaving the other inputs free). To achieve this, the primitive needs to rely on a PKI.

The fact that distributed samplers have no inputs gives a huge advantage: it allows us to satisfy a standard definition of security, without even needing PKIs. Notice that the naive idea of running an NIMPC protocol that, on input

$r_1, \ldots, r_n$, outputs $\mathcal{D}(1\!\!1^\lambda; r_1 \oplus \cdots \oplus r_n)$ does not give a distributed sampler for $\mathcal{D}$, due to the residual function attack.

Two-round reusable MPC. Another related primitive is multi-party, reusable non-interactive secure computation (MrNISC) [BL20], which performs MPC in the party-dynamic setting with minimal interaction. In their construction, based on LWE, parties use the first round to publish encryptions of their input, and later, can publish second round messages for computing any desired function with a subset of parties. While related to distributed samplers, MrNISC does not allow secret randomness to be used in the function, unless it is encoded as part of the inputs in the first round; therefore, it does not seem to help with building a distributed sampler.

iO for Turing machines. Our construction for unbounded universal samplers uses garbled circuits to achieve succinctness, in a similar way to constructions of iO for Turing machines [GS18]. One key difference, however, is that in our setting we are able to prove security relying only on polynomially secure primitives, while all existing constructions of iO for Turing machines rely on subexponentially secure primitives in their security proofs. We note that another construction of iO for Turing machines [BFK+19] uses the shared randomness model to avoid the size of the obfuscated program growing with a bound on the input. This is related to our use of shared randomness for removing the size dependency in our succinct universal sampler, however, the techniques are different.

Recent work on distributed samplers. In [AWZ23], Abram, Waters and Zhandry presented solutions to circumvent the impossibility proven in this paper. Instead of aiming for a simulation-based security definition, they show that, using strong primitives (including subexponential iO) but no random oracle, it is possible to implement game-based definitions for distributed samplers that allow removing trusted setups in one round while preserving the hardness of search problems and the security of most protocols against active adversaries.

3 Preliminaries

Notation. We denote the security parameter by λ. Even when not explicitly written, we assume that all random variables depend on λ. We use bold font to denote vectors, e.g. $\boldsymbol{v}$, single coordinates will be indicated using subscripts, e.g. v_i. The symbol $\sim_c$ denotes computational indistinguishability. We represent the set of corrupted players by C, the set of honest players is instead denoted by H. We indicate the bit-length of any string s by $|s|$. If c is a circuit, we use a similar notation $|c|$ to denote the number of gates. We use $\mathsf{struct}(c)$ to denote the structure of c. With an abuse of notation, we identify distributions $\mathcal{D}$ with circuits mapping uniformly random strings of bits into samples. We say that a distribution is efficient if its circuit has $\mathsf{poly}(\lambda)$ size.

If an algorithm Alg is assisted by an oracle $\mathcal{H}$ in its computations on input x, we write $\mathsf{Alg}^{\mathcal{H}}(x)$. We use a simple arrow $\leftarrow$ to assign the output of a deterministic algorithm $\mathsf{Alg}(x)$ or a fixed value c to a variable a, i.e. we write $a \leftarrow \mathsf{Alg}(x)$ and $a \leftarrow c$. If the algorithm is instead randomised, we write $a \xleftarrow{\$} \mathsf{Alg}(x)$. The notation assumes that in this case, $\mathsf{Alg}(x)$ is provided also with uniformly sampled randomness. We write instead $a \leftarrow \mathsf{Alg}(x; r)$ if we fix the randomness of the algorithm to be r. Finally, we write $a \xleftarrow{\$} X$ where X is a finite set, to denote that a is uniformly sampled from X. If instead a is sampled from a distribution $\mathcal{D}$, we write $a \xleftarrow{\$} \mathcal{D}$.

We present additional preliminaries, including a discussion about distributed samplers and entropy, in Appendix A.

4 Impossibility of Distributed Samplers without Random Oracle

We now present and prove our main theorem, namely that in a strong semi-malicious distributed sampler $\mathsf{H}(R|\sigma) = O(\log \lambda)$. The idea was sketched in the technical overview (see Section 2.1).

Theorem 4.1. *Let $\mathcal{D}(\mathbb{1}^\lambda)$ be an efficient distribution such that $\mathsf{H}_\infty(\mathcal{D}) = \omega(\log \lambda)$. In a strongly semi-maliciously secure distributed sampler for $\mathcal{D}(\mathbb{1}^\lambda)$ in the UC model, we have that $\mathsf{H}(R \mid \sigma) = O(\log \lambda)$.*

Proof. Consider the distributed sampler execution in which the adversary controls a subset C of parties, but behaves exactly as in the protocol. In particular, the adversary waits to receive the messages of the honest parties and then it generates random (and independent) messages for the corrupted parties.

Let Sample be the algorithm used by the parties to reconstruct their output. In order to be as general as possible, compared to Def. A.1, we change the syntax of the procedure by providing it also with the CRS σ, the index i of the party running it and the randomness used by P_i to generate its DS message U_i. Let $\mathsf{Gen}(\mathbb{1}^\lambda, \sigma, j)$ be the algorithm used by party P_j to generate its message. We assume that such algorithm requires $L_j(\lambda)$ bits of randomness. Suppose that the generation of the CRS requires $L(\lambda)$ bits of randomness.

Consider the security game of the protocol, we start by focusing on the ideal world. Since the simulator runs in polynomial time, there exists a polynomial upper bound $q(\lambda)$ on the number of samples the simulator queries to the functionality before providing (σ, U_H) to the adversary. Let Q be the set containing the responses to these queries.

Claim 4.1. *In the ideal world, with overwhelming probability, $R \in Q$.*

Proof of the claim. After the adversary provides U_C, the output of the protocol R is determined and known to the adversary. Notice that, if everybody is honest in the real world, all the parties obtain the same R with overwhelming probability, so R is well defined. If that was not the case, the adversary can easily distinguish the protocol from the simulation (in the ideal world, the honest parties

always output the same value, in the real one they would not). After receiving U_C, the simulator needs to communicate to the functionality that it must output R to the honest parties.

Now, observe that

$$\Pr[\mathcal{D}(1\!\!1^\lambda) = R] = \sum_x \Pr[R = x] \cdot \Pr[\mathcal{D}(1\!\!1^\lambda) = x] \leq$$
$$\leq \max_x \Pr[\mathcal{D}(1\!\!1^\lambda) = x] = 2^{-\mathsf{H}_\infty(\mathcal{D})} = 2^{-\omega(\log \lambda)}.$$

So, $\Pr[\mathcal{D}(1\!\!1^\lambda) = R]$ is negligible.

As a consequence, the simulator can make the honest parties output R only if $R \in Q$. Indeed, once R is fixed, the probability that any subsequent query to the functionality collides with R is negligible. We conclude that $R \in Q$ with overwhelming probability, otherwise it would be possible to distinguish between real world and ideal world. ■

The next claim is used to prove that, in the real world, $\mathsf{H}(R \,|\, U_H, \sigma) = O(\log \lambda)$. We introduce some notation.

Let $p(\lambda)$ be a polynomial and let ι be the index of a fixed corrupted party. Consider the algorithm $\mathcal{D}'_{U_H,\sigma}(1\!\!1^\lambda)$ defined as follows:

1. $\forall j \in C: \quad r_j \xleftarrow{\$} \{0,1\}^{L_j(\lambda)}$
2. $\forall j \in C: \quad U'_j \xleftarrow{\$} \mathsf{Gen}(1\!\!1^\lambda, \sigma, j; r_j)$
3. Output $R' \leftarrow \mathsf{Sample}(\sigma, U_H, U'_C, \iota, r_\iota)$

Let S be the random variable denoting the set of samples produced by running $\mathcal{D}'_{U_H,\sigma}(1\!\!1^\lambda)$ $p(\lambda)$ times.

Let E_0 be the random variable having value 1 if R is well defined (i.e. every party outputs the same value), 0 otherwise. Similarly, let E_1 be random variable having value 1 if $R \in S$, 0 otherwise. Finally, we define $M(\lambda) := L(\lambda) + \sum_{i\in[n]} L_i(\lambda)$.

Claim 4.2. *Suppose that, in the real world, $\mathsf{H}(R \,|\, U_H, \sigma)$ is not $O(\log \lambda)$. Then, for every $\lambda_0 \in \mathbb{N}$, there exists $\lambda \geq \lambda_0$ such that $\Pr[R \notin S | E_0 = 1] \geq 1/M(\lambda)$.*

Proof of the claim. Observe that, for every x, y, z, w, we have

$$\Pr[R = x \,|U_H = y, \sigma = z, S = w] = \Pr[R = x \,|U_H = y, \sigma = z].$$

Indeed, given $U_H = y$ and $\sigma = z$, the value of R is determined only by the randomness used to generate U_C. Such randomness is independent of S. So, $\mathsf{H}(R \,|\, U_H, \sigma, S) = \mathsf{H}(R \,|\, U_H, \sigma)$. We have that

$$\begin{aligned}\mathsf{H}(R \,|\, U_H, \sigma) &= \mathsf{H}(R \,|\, U_H, \sigma, S) \leq \mathsf{H}(R, E_0 \,|\, U_H, \sigma, S) = \\ &= \mathsf{H}(R \,|\, U_H, \sigma, S, E_0) + \mathsf{H}(E_0 \,|\, U_H, \sigma, S) \leq \\ &\leq \mathsf{H}(R \,|\, U_H, \sigma, S, E_0) + \mathsf{H}_0(E_0) = \\ &= \mathsf{H}(R \,|\, U_H, \sigma, S, E_0) + 1. \end{aligned} \tag{1}$$

Now, we have that $\mathsf{H}(R \,|\, U_H, \sigma, S, E_0)$ is equal to

$$\Pr[E_0 = 0] \cdot \mathsf{H}(R \,|\, U_H, \sigma, S, E_0 = 0) + \Pr[E_0 = 1] \cdot \mathsf{H}(R \,|\, U_H, \sigma, S, E_0 = 1). \quad (2)$$

We know that $\Pr[E_0 = 0]$ is negligible, moreover,

$$\mathsf{H}(R \,|\, U_H, \sigma, S, E_0 = 0) \leq \mathsf{H}_0(R) \leq \log\Big(2^{L(\lambda)} \cdot \prod_{i \in [n]} 2^{L_i(\lambda)}\Big) = M(\lambda).$$

We have proven that $\mathsf{H}(R \,|\, U_H, \sigma, S, E_0 = 0) = \mathsf{poly}(\lambda)$, so, putting it together with (1) and (2), we obtain

$$\mathsf{H}(R \,|\, U_H, \sigma) \leq \mathsf{H}(R \,|\, U_H, \sigma, S, E_0 = 1) + 1 + \mathsf{negl}(\lambda). \quad (3)$$

Now, we observe that

$$\begin{aligned}
\mathsf{H}(R \,|\, U_H, \sigma, S, E_0 = 1) &\leq \mathsf{H}(R, E_1 \,|\, U_H, \sigma, S, E_0 = 1) = \\
&= \mathsf{H}(R \,|\, U_H, \sigma, S, E_1, E_0 = 1) + \mathsf{H}(E_1 \,|\, U_H, \sigma, S, E_0 = 1) \leq \\
&\leq \mathsf{H}(R \,|\, U_H, \sigma, S, E_1, E_0 = 1) + \mathsf{H}_0(E_1) = \\
&= \mathsf{H}(R \,|\, U_H, \sigma, S, E_1, E_0 = 1) + 1. \quad (4)
\end{aligned}$$

Furthermore, $\mathsf{H}(R \,|\, U_H, \sigma, S, E_1, E_0 = 1)$ is equal to

$$\begin{aligned}
\Pr[E_1 = 0 | E_0 = 1] \cdot \mathsf{H}(R \,|\, U_H, \sigma, S, E_1 = 0, E_0 = 1) + \\
\Pr[E_1 = 1 | E_0 = 1] \cdot \mathsf{H}(R \,|\, U_H, \sigma, S, E_1 = 1, E_0 = 1). \quad (5)
\end{aligned}$$

We observe that

$$\begin{aligned}
\mathsf{H}(R \,|\, U_H, \sigma, S, &E_1 = 1, E_0 = 1) = \\
&= \sum_w \Pr[S = w] \cdot \mathsf{H}(R \,|\, U_H, \sigma, S = w, E_1 = 1, E_0 = 1) \leq \\
&\leq \sum_w \Pr[S = w] \cdot \mathsf{H}_0(R \,|\, S = w, E_1 = 1, E_0 = 1) \leq \\
&\leq \sum_w \Pr[S = w] \cdot \log\big(p(\lambda)\big) = \log\big(p(\lambda)\big). \quad (6)
\end{aligned}$$

We also notice that

$$\mathsf{H}(R \,|\, U_H, \sigma, S, E_1 = 0, E_0 = 1) \leq \mathsf{H}_0(R) \leq L(\lambda) + \sum_{i \in [n]} L_i(\lambda) = M(\lambda). \quad (7)$$

Now, suppose that there exists $\lambda_0 \in \mathbb{N}$ such that, for all $\lambda \geq \lambda_0$, $\Pr[E_1 = 0 | E_0 = 1] \leq 1/M(\lambda)$. We would have that

$$\begin{aligned}
\mathsf{H}(R \,|\, U_H, \sigma) &\leq \mathsf{H}(R \,|\, U_H, \sigma, S, E_0 = 1) + 1 + \mathsf{negl}(\lambda) \leq && \text{by (3)} \\
&\leq \mathsf{H}(R \,|\, U_H, \sigma, S, E_1, E_0 = 1) + 2 + \mathsf{negl}(\lambda) \leq && \text{by (4)} \\
&\leq \frac{1}{M(\lambda)} \cdot M(\lambda) + \log\big(p(\lambda)\big) + 2 + \mathsf{negl}(\lambda) = && \text{by (5),(6),(7)} \\
&= \log\big(p(\lambda)\big) + 3 + \mathsf{negl}(\lambda).
\end{aligned}$$

So, $\mathsf{H}(R \,|\, U_H, \sigma)$ would be $O(\log \lambda)$ contradicting our initial assumption. We conclude that for every $\lambda_0 \in \mathbb{N}$, there exists $\lambda \geq \lambda_0$ such that $\Pr[E_1 = 0 | E_0 = 1] \geq 1/M(\lambda)$. ■

Claim 4.3. *In the real world,* $\mathsf{H}(R \,|\, U_H, \sigma) = O(\log \lambda)$.

Proof of the claim. By contradiction suppose that, in the real world, $\mathsf{H}(R \,|\, U_H, \sigma)$ is not $O(\log \lambda)$.

Now, consider the adversary $\mathcal{A}$ that, given σ, U_H, samples $(q(\lambda)+1) \cdot \lambda \cdot M(\lambda)$ independent elements from $\mathcal{D}'_{U_H,\sigma}(\mathbb{1}^\lambda)$ and outputs 1 if and only if it obtains strictly more than $q(\lambda)$ distinct values in this way. We show that such adversary can distinguish between real world and ideal world with non-negligible advantage.

First of all, we notice that $\mathcal{A}$ runs in polynomial time. Let $R'_{j,\iota}$ be the output of the j-th execution of $\mathcal{D}'_{U_H,\sigma}(\mathbb{1}^\lambda)$. Observe that the distribution of $R'_{j,\iota}$ conditioned on U_H, σ is the same as the distribution of R_ι, the output of the ι-th party, conditioned on U_H, σ. By Claim 4.1,

$$\Pr[R'_{j,\iota} \notin Q] = \Pr[R_\iota \notin Q] \leq \Pr[E_0 = 0] + \Pr[R \notin Q] = \mathsf{negl}(\lambda).$$

Hence, by the union bound and due to the fact that $|Q| \leq q(\lambda)$, in the ideal world, $\mathcal{A}$ outputs 0 with overwhelming probability.

Now, let S_j be the random variable containing the values of the first $j-1$ samples from $\mathcal{D}'_{U_H,\sigma}(\mathbb{1}^\lambda)$. We know that, in the real world,

$$\begin{aligned}\Pr[R'_{j,\iota} \in S_j] &= \Pr[R_\iota \in S_j] = \\ &= \Pr[E_0 = 0] \cdot \Pr[R_\iota \in S_j | E_0 = 0] + \Pr[E_0 = 1] \cdot \Pr[R_\iota \in S_j | E_0 = 1] \leq \\ &\leq \mathsf{negl}(\lambda) + \Pr[R \in S_j | E_0 = 1].\end{aligned}$$

We conclude that, for every $\lambda_0 \in \mathbb{N}$, there exists a $\lambda \geq \lambda_0$ such that

$$\Pr[R'_{j,\iota} \in S_j] \leq 1 - \frac{1}{2M(\lambda)}$$

Now, we observe that, for every $\lambda_0 \in \mathbb{N}$, there exists a $\lambda \geq \lambda_0$ such that

$$\Pr\left[|S_{j \cdot \lambda \cdot M}| = |S_{(j+1) \cdot \lambda \cdot M}|\right] \leq \left(1 - \frac{1}{2M(\lambda)}\right)^{\lambda \cdot M(\lambda)}$$

and so, by the union bound, for the same values of λ,

$$\begin{aligned}\Pr[|S_{(q(\lambda)+1) \cdot \lambda \cdot M + 1}| \leq q(\lambda)] \leq &\Pr\left[\exists j \text{ s.t. } |S_{j \cdot \lambda \cdot M}| = |S_{(j+1) \cdot \lambda \cdot M}|\right] \leq \\ \leq &(q(\lambda)+1) \cdot \left(1 - \frac{1}{2M(\lambda)}\right)^{\lambda \cdot M(\lambda)}\end{aligned}$$

Observe that

$$\lim_{\lambda \to \infty} (q(\lambda)+1) \cdot \left(1 - \frac{1}{2M(\lambda)}\right)^{\lambda \cdot M(\lambda)} = 0,$$

so, $\Pr[|S_{(q(\lambda)+1)\cdot\lambda\cdot M+1}| \leq q(\lambda)]$ is definitively smaller than $1/3$. Therefore, for every $\lambda_0 \in \mathbb{N}$, there exists a $\lambda \geq \lambda_0$ such that

$$\begin{aligned}\mathrm{Adv}_{\mathcal{A}}(\lambda) = \Big|\Pr[\mathcal{A}(\mathbb{1}^\lambda) = 0|\mathrm{ideal}] - \Pr[\mathcal{A}(\mathbb{1}^\lambda) = 0|\mathrm{real}]\Big| \geq \\ \geq 1 - \mathsf{negl}(\lambda) - 1/3 \geq 1/2 - 1/3.\end{aligned}$$

We conclude that $\mathcal{A}$ distinguishes between the real world and the ideal world with non-negligible advantage. This contradicts the security of the distributed sampler, therefore, in the real world, $\mathsf{H}(R \,|\, U_H, \sigma) = O(\log \lambda)$. ■

Claim 4.4. *In the real world, we have that* $\mathsf{H}(R \,|\, U_C, \sigma) = O(\log \lambda)$.

Proof of the claim. The messages of the corrupted parties are distributed as in the fully honest case. So, switching the role of honest and corrupted parties does not affect the distribution of $(\sigma, (U_i)_{i\in[n]}, R)$. By applying the result of Claim 4.3 on the new set of corrupted parties, we obtain that $\mathsf{H}(R \,|\, U_C, \sigma) = O(\log \lambda)$. ■

Claim 4.5. *In the real world,* $\mathsf{H}(R \,|\, \sigma) = O(\log \lambda)$.

Proof of the claim. By the strong chain rule of Shannon's entropy, we have that

$$\begin{aligned}&\mathsf{H}(R \,|\, U_H, \sigma) + \mathsf{H}(R \,|\, U_C, \sigma) - \mathsf{H}(R \,|\, U_H, U_C, \sigma) + \\ &+ \mathsf{H}(U_H \,|\, \sigma) - \mathsf{H}(U_H \,|\, U_C, \sigma) + \\ &+ \mathsf{H}(U_C \,|\, R, U_H, \sigma) - \mathsf{H}(U_C \,|\, R, \sigma) = \\ =&\underline{\mathsf{H}(R, U_H \,|\, \sigma)} - \underline{\mathsf{H}(U_H \,|\, \sigma)} + \underline{\mathsf{H}(R, U_C \,|\, \sigma)} - \underline{\mathsf{H}(U_C \,|\, \sigma)} + \\ &- \underline{\mathsf{H}(R, U_H, U_C \,|\, \sigma)} + \underline{\mathsf{H}(U_H, U_C \,|\, \sigma)} + \\ &+ \underline{\mathsf{H}(U_H \,|\, \sigma)} - \underline{\mathsf{H}(U_H, U_C \,|\, \sigma)} + \underline{\mathsf{H}(U_C \,|\, \sigma)} + \\ &+ \underline{\mathsf{H}(U_C, R, U_H \,|\, \sigma)} - \underline{\mathsf{H}(R, U_H \,|\, \sigma)} - \underline{\mathsf{H}(U_C, R \,|\, \sigma)} + \mathsf{H}(R \,|\, \sigma) = \\ =&\mathsf{H}(R \,|\, \sigma).\end{aligned}$$

We observe that $\mathsf{H}(U_C \,|\, R, U_H, \sigma) \leq \mathsf{H}(U_C \,|\, R, \sigma)$ and $\mathsf{H}(R \,|\, U_H, U_C, \sigma) \geq 0$. Moreover, since U_H and U_C are independent given σ, $\mathsf{H}(U_H \,|\, \sigma) = \mathsf{H}(U_H \,|\, U_C, \sigma)$. We conclude that, by Claim 4.3 and 4.4, $\mathsf{H}(R \,|\, \sigma) \leq \mathsf{H}(R \,|\, U_H, \sigma) + \mathsf{H}(R \,|\, U_C, \sigma) = O(\log \lambda)$. ■

□

4.1 Distributed Sampler CRSs Cannot be Used Twice

We now discuss the first consequence of Theorem 4.1. Suppose that our distribution $\mathcal{D}(\mathbb{1}^\lambda)$ has high min-entropy, i.e. $\mathsf{H}_\infty(\mathcal{D}) = \omega(\log \lambda)$. We observe that the probability that two independent samples from $\mathcal{D}(\mathbb{1}^\lambda)$ collide is negligible. Indeed, denoting the two independent outputs by R and R', we have

$$\Pr[R = R'] = 2^{-\mathsf{H}_2(\mathcal{D})} \leq 2^{-\mathsf{H}_\infty(\mathcal{D})} = 2^{-\omega(\log \lambda)}.$$

We show, however, that if we run a strongly semi-maliciously secure distributed sampler for $\mathcal{D}(1\!\!1^\lambda)$ twice using the same CRS, the outputs collide with non-negligible probability. As a consequence, we cannot hope to reuse the same CRS to generate independent looking samples.

Corollary 4.2. *Let $\mathcal{D}(1\!\!1^\lambda)$ be an efficiently samplable distribution such that $\mathsf{H}_\infty(\mathcal{D}) = \omega(\log\lambda)$. Consider a strongly semi-maliciously secure distributed sampler protocol for $\mathcal{D}(1\!\!1^\lambda)$ in the UC model and let R and R' denote the outputs of two protocol executions having the same CRS. Then, $\Pr[R = R'] \geq 1/\mathsf{poly}(\lambda)$.*

Proof. By Theorem 4.1, we know that $\mathsf{H}(R|\sigma) = O(\log\lambda)$. Now, we have that

$$\Pr[R = R'] = \sum_z \Pr[\sigma = z] \cdot \Pr[R = R'|\sigma = z] = \sum_z \Pr[\sigma = z] \cdot 2^{-\mathsf{H}_2(R|\sigma=z)} \geq$$
$$\geq \sum_z \Pr[\sigma = z] \cdot 2^{-\mathsf{H}(R|\sigma=z)}$$

We observe that $f(x) := 2^{-x}$ is a convex function, so, by Jensen's inequality

$$\Pr[R = R'] \geq \sum_z \Pr[\sigma = z] \cdot 2^{-\mathsf{H}(R|\sigma=z)} \geq 2^{-\sum_z \Pr[\sigma=z]\cdot\mathsf{H}(R|\sigma=z)} = 2^{-\mathsf{H}(R|\sigma)}.$$

Observe that $2^{-\mathsf{H}(R|\sigma)} = 1/\mathsf{poly}(\lambda)$. □

4.2 Distributed Sampler CRSs Cannot be Short

A second consequence of Theorem 4.1 is that the CRSs of strongly semi-maliciously secure distributed samplers cannot be short. Specifically, the bit-length of the CRS $|\sigma|$ must be at most $O(\log\lambda)$ bits shorter than $\mathsf{H}_{\mathsf{Yao}}(\mathcal{D})$.

The Yao entropy $\mathsf{H}_{\mathsf{Yao}}(R|\sigma)$ must be small. Although Yao's entropy and Shannon's entropy can assume very different values, we prove that if $\mathsf{H}(R|\sigma) = O(\log\lambda)$, also $\mathsf{H}_{\mathsf{Yao}}(R|\sigma) = O(\log\lambda)$. Indeed, by Corollary 4.2, we know that two distributed sampler executions using the same CRS have colliding outputs with non-negligible probability. We can therefore consider the compressor that on input (R, σ) outputs the empty string and the decompressor that on input σ, runs the distributed sampler using σ as CRS, outputting the result R'. Since there is a $1/\mathsf{poly}(\lambda)$ probability that $R = R'$, we conclude that $\mathsf{H}_{\mathsf{Yao}}(R|\sigma) = O(\log\lambda)$. Observe that we can make the decompressor deterministic using a PRF.

A chain rule for Yao's entropy. To conclude our argument, we show that

$$\mathsf{H}_{\mathsf{Yao}}(R|\sigma) \geq \mathsf{H}_{\mathsf{Yao}}(R) - |\sigma|. \tag{8}$$

By the security of distributed samplers in the fully honest case, R and $\mathcal{D}(1\!\!1^\lambda)$ are computationally indistinguishable, so, $\mathsf{H}_{\mathsf{Yao}}(R) = \mathsf{H}_{\mathsf{Yao}}(\mathcal{D})$. From this, we easily deduce that $\mathsf{H}_{\mathsf{Yao}}(\mathcal{D}) - |\sigma| \leq O(\log\lambda)$.

We highlight that in [KPW13, Appendix B], Krenn *et al.* proved the chain rule for Yao's entropy, which seems to immediately imply (8). Unfortunately, this is not the case. The idea at the base of their proof is that given a compressor-decompressor pair (c, d) for $\mathsf{H}_{\mathsf{Yao}}(R|\sigma)$, we can build a new compressor-decompressor pair for $\mathsf{H}_{\mathsf{Yao}}(R)$ with the same success probability as (c, d). On input R, the new compressor performs a brute-force search for a σ' such that $d(c(R, \sigma'), \sigma') = R$, then it outputs $c(R, \sigma'), \sigma'$. The decompressor instead is identical to d. Since, the output size of the new compressor is $|\sigma|$ bits larger than c's, we obtain that $\mathsf{H}_{\mathsf{Yao}}(R|\sigma) \geq \mathsf{H}_{\mathsf{Yao}}(R) - |\sigma|$. Observe however that if $|\sigma|$ is more than $O(\log \lambda)$, the new compressor does not run in polynomial time. That prevents us from using their result.

We notice that in our setting, the new compressor does not need to perform a brute-force search. Indeed, in order to obtain a σ', it can just feed R to the distributed sampler simulator for the fully honest case and pick the CRS contained in the simulated view. The latter is indistinguishable from the the real CRS used for the generation of R. This allows us to prove the chain rule even if $|\sigma|$ is more than $O(\log \lambda)$.

Corollary 4.3. *Suppose that* $\mathsf{H}_\infty(\mathcal{D}) = \omega(\log \lambda)$. *If OWFs exist, the CRS* σ *of a strongly semi-maliciously secure distributed sampler for* $\mathcal{D}(1\!\!1^\lambda)$ *in the UC model must satisfy* $\mathsf{H}_{\mathsf{Yao}}(\mathcal{D}) - |\sigma| \leq O(\log \lambda)$.

Proof. We start by proving the following claim.

Claim 4.1. *In the distributed sampler protocol,* $\mathsf{H}_{\mathsf{Yao}}(R|\sigma) = O(\log \lambda)$.

Proof of the claim. Consider the following randomised pair of compressor and decompressor:

- On input a pair (R, σ), the compressor c outputs the empty string.
- On input σ, the decompressor d runs the distributed sampler protocol in-its-head using σ as CRS. Then, it outputs the result R'.

Observe that $\Pr[d(c(R, \sigma), \sigma) = R] = \Pr[R = R']$. Since R and R' are the outputs of two distributed sampler executions using the same CRS, by Corollary 4.2, we know that

$$\Pr[d(c(R, \sigma), \sigma) = R] = \frac{1}{\mathsf{poly}(\lambda)}$$

Now, consider the deterministic decompressor d' that performs exactly the same operations as d, but uses a PRF F to generate its randomness, i.e. d' has a random PRF key K hard-coded in its circuit and it generates its randomness by computing $F(K, \sigma)$. By the security of the PRF

$$\Pr[d'(c(R, \sigma), \sigma) = R] \geq \Pr[d(c(R, \sigma), \sigma) = R] - \mathsf{negl}(\lambda) = \frac{1}{\mathsf{poly}(\lambda)}.$$

Notice that the probability in the first term is also over K. So,

$$\begin{aligned}\frac{1}{\mathsf{poly}(\lambda)} \leq \Pr[d'(c(R, \sigma), \sigma) = R] = \sum_x \frac{1}{|K|} \cdot \Pr[d'(c(R, \sigma), \sigma) = R | K = x] \leq \\ \leq \max_x \Pr[d'(c(R, \sigma), \sigma) = R | K = x].\end{aligned}$$

Let $\hat{x}$ be the value of x associated with the maximum, let $d'_{\hat{x}}$ be the decompressor having $K = \hat{x}$. We have proven that the pair $(c, d'_{\hat{x}})$ is successful in compressing and decompressing with probability greater than $1/\mathsf{poly}(\lambda)$. We conclude that $\mathsf{H}_{\mathsf{Yao}}(R|\sigma) = O(\log \lambda)$. ■

Claim 4.2. *In the distributed sampler protocol,* $\mathsf{H}_{\mathsf{Yao}}(R|\sigma) \geq \mathsf{H}_{\mathsf{Yao}}(R) - |\sigma|$.

Proof of the claim. Suppose that $\mathsf{H}_{\mathsf{Yao}}(R|\sigma) \leq k(\lambda)$ for some function $k(\lambda)$. Then, there exists a compressor-decompressor pair (c, d) such that

$$\Pr[d(c(R, \sigma'), \sigma') = R] \geq \frac{2^\ell}{2^k} - \mathsf{negl}(\lambda).$$

We now design a new compressor $\hat{c}$ for R: $\hat{c}$ feeds its input R to the distributed sampler simulator Sim for the fully-honest case. The latter provides a CRS σ' and the messages of all the parties. The compressor outputs $c(R, \sigma')$ as well as σ'.

Let σ be the CRS used to generate R. By the security of the distributed sampler in the fully honest case, we know that the triple $(\sigma, (U_i)_{i\in[n]}, R)$ in the protocol is computationally indistinguishable from $(\mathsf{Sim}(1\!\!1^\lambda, R'), R')$ where $R' \xleftarrow{\$} \mathcal{D}(1\!\!1^\lambda)$. We conclude that the pairs (R, σ) and (R, σ') are also computationally indistinguishable. As a consequence,

$$\Big|\Pr[d(c(R, \sigma), \sigma) = R] - \Pr[d(c(R, \sigma'), \sigma') = R]\Big| = \mathsf{negl}(\lambda).$$

We conclude that

$$\begin{aligned}\Pr[d(\hat{c}(R)) = R] = \Pr[d(c(R, \sigma'), \sigma') = R] \geq \Pr[d(c(R, \sigma'), \sigma') = R] - \mathsf{negl}(\lambda) \geq \\ \geq \frac{2^\ell}{2^k} - \mathsf{negl}(\lambda) = \frac{2^{\ell+|\sigma|}}{2^{k+|\sigma|}} - \mathsf{negl}(\lambda).\end{aligned}$$

Observe that $\ell + |\sigma|$ is the output size of $\hat{c}$. Notice also that $\hat{c}$ is not deterministic, however, we can make it so adopting the same technique used in Claim 4.1, i.e. by generating its randomness using a PRF. Specifically, consider the deterministic compressor $\hat{c}'$ which has a PRF key K hard-coded in its circuit and generates its randomness by computing $F(K, R)$, performing then the same operations as $\hat{c}$. By the security of the PRF, we have that

$$\Big|\Pr[d(\hat{c}'(R)) = R] - \Pr[d(\hat{c}(R)) = R]\Big| = \mathsf{negl}(\lambda).$$

So,

$$\Pr[d(\hat{c}'(R)) = R] \geq \frac{2^{\ell+|\sigma|}}{2^{k+|\sigma|}} - \mathsf{negl}(\lambda).$$

As in the proof of the previous claim, the probability in the first term is also over K. So,

$$\begin{aligned}\frac{2^{\ell+|\sigma|}}{2^{k+|\sigma|}} - \mathsf{negl}(\lambda) \leq \Pr[d(\hat{c}'(R)) = R] = \sum_x \frac{1}{|K|} \cdot \Pr[d(\hat{c}'(R)) = R | K = x] \leq \\ \leq \max_x \Pr[d(\hat{c}'(R)) = R | K = x].\end{aligned}$$

Let $\hat{x}$ be the value of x associated with the maximum, let $\hat{c}'_{\hat{x}}$ be the decompressor having $K = \hat{x}$. We have proven that the pair $(\hat{c}'_{\hat{x}}, d)$ succeeds in compressing and decompressing R with probability greater than $2^{\ell+|\sigma|}/2^{k+|\sigma|} - \mathsf{negl}(\lambda)$, so $\mathsf{H}_{\mathsf{Yao}}(R) \leq k(\lambda) + |\sigma|$. We conclude that $\mathsf{H}_{\mathsf{Yao}}(R|\sigma) \geq \mathsf{H}_{\mathsf{Yao}}(R) - |\sigma|$. ■

By Claims 4.1 and 4.2, we have $O(\log \lambda) = \mathsf{H}_{\mathsf{Yao}}(R|\sigma) \geq \mathsf{H}_{\mathsf{Yao}}(R) - |\sigma|$. We notice that, by the security of the distributed sampler in the fully honest case, R is computationally indistinguishable from $\mathcal{D}(1\!\!1^\lambda)$, so, $\mathsf{H}_{\mathsf{Yao}}(R) = \mathsf{H}_{\mathsf{Yao}}(\mathcal{D})$. We conclude that $\mathsf{H}_{\mathsf{Yao}}(\mathcal{D}) - |\sigma| \leq O(\log \lambda)$. □

4.3 Distributed Sampler CRSs Cannot be (too) Nice

The *niceness* of a CRS cannot be defined in a mathematical way. However, informally speaking, we can say that a CRS is nicer than another if it is easier to produce in an MPC setting, e.g. a uniformly random string of bits (i.e. a URS) is simpler to generate than a random RSA modulus of unknown factorisation. Indeed, if we aim for security with abort, we can generate a URS using a simple commit-then-reveal approach. On the other hand, all the state-of-the-art constructions for the generation of RSA moduli rely on rejection sampling: first the parties generate secret-shared (or encrypted) candidate primes p and q, they multiply them and apply expensive (bi)primality tests on the secret-shared data [FLOP18, HMR+19, CCD+20]. After sufficiently many trials (the number depends on the size of the modulus), the players obtain a valid RSA modulus with high probability. This results in rather complex protocols.

In the previous sections, we have seen that the CRS of distributed samplers cannot be used more then once and cannot be short. These facts suggest that directly encoding a sample from $\mathcal{D}(1\!\!1^\lambda)$ in a CRS is probably better than relying on a distributed sampler protocol for $\mathcal{D}(1\!\!1^\lambda)$. At least in the first case, the parties spare one round of interaction. But what if the CRS used by the distributed sampler is nicer than any encoding of $\mathcal{D}(1\!\!1^\lambda)$? We prove that this cannot happen as it is always possible to non-interactively generate samples from $\mathcal{D}(1\!\!1^\lambda)$ using only distributed sampler CRSs and public random coins.

Non-interactive generation of samples from $\mathcal{D}(1\!\!1^\lambda)$ using a distributed sampler CRS and random bits. In this section, we observe that a distributed sampler for $\mathcal{D}(1\!\!1^\lambda)$ allows us to construct an efficient deterministic function De that maps pairs consisting of a distributed sampler CRS σ and uniformly random bits r into values R that are computationally indistinguishable from $\mathcal{D}(1\!\!1^\lambda)$. This algorithm trivially outputs the result of the distributed sampler using σ as CRS and r as randomness for the parties. The interesting fact is that if the distributed sampler is strongly semi-maliciously secure in the UC model, the algorithm is efficiently invertible with non-negligible probability. Specifically, we prove that there exists an efficient PPT algorithm test, that outputs 1 with $1/\mathsf{poly}(\lambda)$ probability when run over a sample $R \xleftarrow{\$} \mathcal{D}(1\!\!1^\lambda)$. Moreover, if this event occur, it is possible to efficiently find a pair (σ, r) that looks random over $\mathsf{De}^{-1}(R)$.

In other words, if we provide the parties with a distributed sampler CRS σ and public uniformly random bits r, the parties can obtain a sample R from $\mathcal{D}(1\!\!1^\lambda)$ without any interaction. Furthermore, with $1/\mathsf{poly}(\lambda)$ probability, (σ, r) reveals no information in addition to what can be already inferred from R. Finally, the honest parties can tell when (σ, r) reveals too much information, having therefore the opportunity to reject R and restart.

Biases in the distribution. Notice that the selective rejection biases the distribution of the output. However, any cryptographic protocol basing its security on R will remain secure even if R is sampled according to the biased distribution. Indeed, at least a polynomial fraction of the samples is not rejected. If the protocol was insecure in the new setting, there would be a non-negligible probability of sampling a bad R even in the original protocol, which would therefore be insecure.

Why is De *invertible with non-negligible probability?* Our idea is based on the fact that in a strongly semi-malicious distributed sampler, $\mathsf{H}(R|\sigma)$ is small. In other words, the CRS of the protocol describes the output with high precision. Obtaining the exact CRS used for the generation of R is usually hard, however, the simulator for the fully-honest case can provide us with a functionally equivalent object.

In order to completely describe R, we need to extract also randomness r for the parties, so that, using r in conjunction with the CRS, we obtain R. Unfortunately, the simulator cannot provide much help here. Indeed, given $(\sigma', (U_i)_{i\in[n]}) \xleftarrow{\$} \mathsf{Sim}(1\!\!1^\lambda, R)$, it is usually hard to extract the randomness r used to generate $(U_i)_{i\in[n]}$. Actually, such r might not even exist. Luckily, in Corollary 4.2, we have proven that two executions of a strongly semi-maliciously secure distributed sampler using the same CRS have colliding outputs with $1/\mathsf{poly}(\lambda)$ probability. So, if we run the distributed sampler protocol again using σ' as CRS, we obtain R again with non-negligible probability. Clearly, in the new execution, the value of $(U_i)_{i\in[n]}$ has probably changed, however, this time we know the randomness r' used to generate the messages.

On average invertibility. We observe that the probability of succeeding in inverting De is also over the outcome of R. In other words, we just know that the *average* probability of inverting R is $1/\mathsf{poly}(\lambda)$. That does not mean that the overwhelming majority of values R is efficiently invertible: if R assumes an unlucky value, we can try to invert as many times as we want without any hope of succeeding. We could prove, however, that there always exists a polynomial fraction of the space of events for which R is easy to invert.

We also noticed that it is possible to efficiently test if R is easy to invert or not. Indeed, we can just try to invert it many times, if the success frequency is lower than a certain threshold, we can reject R, otherwise, we accept it. We used the Chernoff bound to find the threshold and the maximum number of inversion attempts. In particular, we needed test to succeed with at least $1/\mathsf{poly}(\lambda)$ probability.

THE GAME $\mathcal{G}^{\mathsf{Inv}}_{\mathcal{A}}(\mathbb{1}^\lambda)$

The challenger performs the following operations

1. $b \xleftarrow{\$} \{0,1\}$
2. $\sigma \xleftarrow{\$} \mathsf{CRS}(\mathbb{1}^\lambda)$, $r \xleftarrow{\$} \mathcal{U}(\mathbb{1}^\lambda)$
3. $R \leftarrow \mathsf{De}(\sigma, r)$
4. If $\mathsf{test}(R) = 0$, go back to step 2.
5. If $b = 0$, provide the adversary with (σ, r), otherwise provide it with $\mathsf{En}(R)$.

The adversary wins if it terminates its execution outputting b.

Fig. 5: Invertibility game

The special case of URSs. As a corollary of the result described in this section, if the distributed sampler uses a URS, it is possible to securely generate samples from $\mathcal{D}(\mathbb{1}^\lambda)$ using public random coins only. In particular, in the random oracle model, the parties can securely sample from $\mathcal{D}(\mathbb{1}^\lambda)$ without interacting and without needing any CRS.

We formalise our result below.

Corollary 4.4. *Suppose that* $\mathsf{H}_\infty(\mathcal{D}) = \omega(\log \lambda)$ *and there exists a strongly semi-maliciously secure distributed sampler for* $\mathcal{D}(\mathbb{1}^\lambda)$ *in the UC model. Let* $\mathsf{CRS}(\mathbb{1}^\lambda)$ *be the algorithm used to generate the CRS of the protocol. Then, there exist a deterministic polynomial algorithm* De *and PPT algorithms* En *and* test *such that*

- $\mathsf{De}\big(\mathsf{CRS}(\mathbb{1}^\lambda), \mathcal{U}(\mathbb{1}^\lambda)\big) \sim_c \mathcal{D}(\mathbb{1}^\lambda)$
- $\Pr[\mathsf{test}\big(\mathcal{D}(\mathbb{1}^\lambda)\big) = 1] \geq 1/\mathsf{poly}(\lambda)$
- *No PPT adversary can win the game* $\mathcal{G}^{\mathsf{Inv}}_{\mathcal{A}}(\mathbb{1}^\lambda)$ *(see Fig. 5) with non-negligible advantage.*

Proof. We start by defining the algorithm De, which on input σ and random string r, runs the distributed sampler protocol using σ as CRS and r as randomness for the parties, outputting the result.

Claim 4.1. $\mathsf{De}\big(\mathsf{CRS}(\mathbb{1}^\lambda), \mathcal{U}(\mathbb{1}^\lambda)\big) \sim_c \mathcal{D}(\mathbb{1}^\lambda)$

Proof of the claim. We observe that $R := \mathsf{De}\big(\mathsf{CRS}(\mathbb{1}^\lambda), \mathcal{U}(\mathbb{1}^\lambda)\big)$ is distributed exactly as the distributed sampler output. By the security of the distributed sampler, we conclude that $R \sim_c \mathcal{D}(\mathbb{1}^\lambda)$. ■

We now define the PPT algorithm Inv as follows: Inv feeds the input R to the distributed sampler simulator for the fully honest case. In this way, it obtains a fake CRS σ' and messages $(U_i)_{i\in[n]}$. Finally, Inv picks a uniformly random string r and outputs (σ', r).

Claim 4.2. *Let R denote a sample from $\mathcal{D}(1\!\!1^\lambda)$. Then, there exists a polynomial $q(\lambda)$ such that*

$$\Pr[\mathsf{De}(\mathsf{Inv}(R)) = R] \geq \frac{1}{q(\lambda)}$$

Proof of the claim. Let $(\sigma', r) := \mathsf{Inv}(R)$. By the security of distributed samplers, we know that (σ', R) is computationally indistinguishable from $(\sigma, \mathsf{De}(\sigma, r))$ where $\sigma \xleftarrow{\$} \mathsf{CRS}(1\!\!1^\lambda)$ and r is uniformly random. We conclude that for r and r' independent and uniformly random

$$\big(\sigma, \mathsf{De}(\sigma, r'), \mathsf{De}(\sigma, r)\big) \sim_c \big(\sigma', R, \mathsf{De}(\sigma', r)\big) \tag{9}$$

By Corollary 4.2, we know that

$$\Pr[\mathsf{De}(\sigma, r') = \mathsf{De}(\sigma, r)] \geq \frac{1}{\mathsf{poly}(\lambda)}$$

We conclude that, by (9),

$$\Pr[\mathsf{De}(\mathsf{Inv}(R)) = R] = \Pr[\mathsf{De}(\sigma', r') = R] \geq \frac{1}{\mathsf{poly}(\lambda)} - \mathsf{negl}(\lambda) \geq \frac{1}{\mathsf{poly}(\lambda)}$$

■

We now define the algorithm test as follows: on input R, test checks if $\mathsf{De}(\mathsf{Inv}(R)) = R$ for $4\lambda \cdot q(\lambda)$ times. If the equation is satisfied less than λ times, test outputs 0, otherwise it output 1.

In a similar way, we define En: on input R, En computes $(\sigma', r) \xleftarrow{\$} \mathsf{Inv}(R)$ and checks if $\mathsf{De}(\sigma', r) = R$. If that is the case, it outputs (σ', r). Otherwise, it repeats the operation. If the procedure fails for more than $8\lambda \cdot q(\lambda)$, En outputs $\perp$.

Claim 4.3.

$$\Pr[\mathsf{test}\big(\mathcal{D}(1\!\!1^\lambda)\big) = 1] \geq \frac{1}{8q(\lambda)}$$

Proof of the claim. We define $p_x := \Pr[\mathsf{De}(\mathsf{Inv}(x)) = x]$. Let R be a sample from $\mathcal{D}(1\!\!1^\lambda)$. By Claim 4.2, we know that

$$\mathbb{E}[p_R] = \sum_x \Pr[R = x] \cdot p_x = \Pr[\mathsf{De}(\mathsf{Inv}(R)) = R] \geq \frac{1}{q(\lambda)}$$

Since $0 \leq p_x \leq 1$, we have that $\mathbb{E}[p_R^2] \leq \mathbb{E}[p_R]$, so, by the Paley-Zygmund inequality,

$$\Pr\Big[p_R \geq \frac{1}{2q(\lambda)}\Big] \geq \Pr\Big[p_R > \frac{1}{2}\mathbb{E}[p_R]\Big] \geq \frac{1}{4} \cdot \frac{\mathbb{E}[p_R]^2}{\mathbb{E}[p_R^2]} \geq \frac{1}{4} \cdot \mathbb{E}[p_R] \geq \frac{1}{4q(\lambda)}$$

Define now $\Omega_{\mathsf{good}} = \{x | p_x \geq \frac{1}{2q(\lambda)}\}$. Suppose now that $x \in \Omega_{\mathsf{good}}$. If we run the check $\mathsf{De}(\mathsf{Inv}(x)) = x$ for $8\lambda \cdot q(\lambda)$ times, by the Chernoff bound, we know that it succeeds more than

$$\frac{1}{2} \cdot 4\lambda q(\lambda) \cdot p_x \geq \lambda$$

times with overwhelming probability. So if $x \in \Omega_{\mathsf{good}}$, $\mathsf{test}(x) = 1$ with overwhelming probability. We conclude that

$$\Pr[\mathsf{test}(R) = 1] \geq \Pr[\mathsf{test}(R) = 1 | R \in \Omega_{\mathsf{good}}] \cdot \Pr[R \in \Omega_{\mathsf{good}}] \geq$$
$$\geq \min_{x \in \Omega_{\mathsf{good}}} \Pr[\mathsf{test}(x) = 1] \cdot \Pr[R \in \Omega_{\mathsf{good}}] \geq \frac{1}{8q(\lambda)}$$

■

Claim 4.4. *No PPT adversary can win the game $\mathcal{G}^{\mathsf{Inv}}_{\mathcal{A}}(1\!\!1^\lambda)$ (see Fig. 5) with non-negligible advantage.*

Proof of the claim. As in the previous claim, let $p_x := \Pr[\mathsf{De}(\mathsf{Inv}(x)) = x]$. Define now $\Omega_{\mathsf{bad}} = \{x | p_x \leq \frac{1}{8q(\lambda)}\}$. Suppose that $x \in \Omega_{\mathsf{bad}}$. If we run the check $\mathsf{De}(\mathsf{Inv}(x)) = x$ for $4\lambda \cdot q(\lambda)$ times, by the Chernoff bound, we know that it succeeds less than

$$2 \cdot 4\lambda q(\lambda) \cdot p_x \leq \lambda$$

times with overwhelming probability. So, if $x \in \Omega_{\mathsf{bad}}$, $\mathsf{test}(x) = 0$ with overwhelming probability.

Let $R \xleftarrow{\$} \mathcal{D}(1\!\!1^\lambda)$. We observe that

$$\Pr[\mathsf{En}(R) = \bot | \mathsf{test}(R) = 1] \leq \Pr[R \in \Omega_{\mathsf{bad}} | \mathsf{test}(R) = 1] + \Pr[\mathsf{En}(R) = \bot | R \notin \Omega_{\mathsf{bad}}].$$

We know that

$$\Pr[R \in \Omega_{\mathsf{bad}} | \mathsf{test}(R) = 1] \leq \frac{\Pr[\mathsf{test}(R) = 1 | R \in \Omega_{\mathsf{bad}}]}{\Pr[\mathsf{test}(R) = 1]} \leq$$
$$\leq 8q(\lambda) \cdot \max_{x \in \Omega_{\mathsf{bad}}} \Pr[\mathsf{test}(x) = 1] = \mathsf{negl}(\lambda).$$

Furthermore,

$$\Pr[\mathsf{En}(R) = \bot | R \notin \Omega_{\mathsf{bad}}] \geq \min_{x \notin \Omega_{\mathsf{bad}}} \Pr[\mathsf{En}(x) = \bot].$$

Notice that for every $x \notin \Omega_{\mathsf{bad}}$, $p_x > \frac{1}{8q(\lambda)}$. Observe also that En tries to invert the input up to $8\lambda \cdot q(\lambda) > \lambda / p_x$ times, so, with overwhelming probability $\mathsf{En}(x) \neq \bot$. We conclude that $\Pr[\mathsf{En}(R) = \bot | \mathsf{test}(R) = 1] = \mathsf{negl}(\lambda)$.

Now, suppose that $\mathsf{En}(R) = (\sigma', r') \neq \bot$. We recall that σ' is obtained by running $\mathsf{Sim}(1\!\!1^\lambda, R)$, so by the security of distributed samplers $(R, \sigma') \sim_c (\mathsf{De}(\sigma, r), \sigma)$ where $\sigma \xleftarrow{\$} \mathsf{CRS}(1\!\!1^\lambda)$ and $r \xleftarrow{\$} \mathcal{U}(1\!\!1^\lambda)$. We also know that r' is random conditioned on satisfying $\mathsf{De}(\sigma', r') = R$. In other words, (R, σ', r') is computationally indistinguishable from $(\mathsf{De}(\sigma, r), \sigma, r)$. We conclude our proof observing that $(R, \mathsf{En}(R)) \sim_c (\mathsf{De}(\sigma, r), \mathsf{En}(\mathsf{De}(\sigma, r)))$. ■

□

5 Succinct and Unbounded Universal Samplers

Universal samplers. In [HJK+16], Hofheinz *et al.* introduced the notion of universal sampler: a particular type of trusted setup, called the sampler parameters, which allows non-interactively and securely generating samples from any distribution $\mathcal{D}$. The authors considered two notions of security. The first one is *selective, one-time security*, meaning that for a certain distribution $\mathcal{D}$ fixed before generating the sampler parameters, the construction reveals no information in addition to the corresponding sample. The second notion is *adaptive security*, meaning that the sampler parameters can generate samples from multiple distributions adaptively chosen by the adversary, even after seeing the parameters. If adaptive security holds, the construction is still guaranteed to reveal no information in addition to the outputs. Unfortunately, adaptive universal samplers can only exist in the random oracle model.

Limitations on the size of supported distributions. Independently of the notion of security, all known universal sampler constructions [HJK+16] suffer from a particular limitation: the circuit size of the supported distributions is bounded from above by the size of the sampler parameters. In this paper, we show how to get around this problem. We highlight that in order to obtain a truly non-interactive solution, we are forced to rely on a random oracle. Indeed, in the plain model, the parameters cannot be smaller than the Yao incompressibility entropy of the sample we want to produce.

Succinct and unbounded universal samplers. Rather than aiming full-on to our objective, we linger in the plain model for a little longer and we rephrase the definition of universal sampler. We relax the sampling algorithm, Sample, to take as input a long string of uniform bits $\boldsymbol{u}$, as well as the trusted sampler parameters U. In our final solution, called an *unbounded universal sampler* (UUS), U will impose no restrictions on the set of supported distributions. Instead, we require the length of $\boldsymbol{u}$ to depend on the distribution. The good news is that if $\boldsymbol{u}$ is too short to sample from our distribution, it is not hard to extend it (e.g. using a coin tossing protocol or in the random oracle model). Notice that by extending $\boldsymbol{u}$, we increase the entropy of the setup and so, get around the impossibility.

In our quest for unbounded universal samplers, we introduce an intermediate stepping stone of a *succinct universal sampler* (SUS). An SUS differs from an unbounded one as its sampler parameters U set a polynomial upper bound L on the circuit size of the supported distributions. However, we also require that U can be generated using a circuit of $\mathsf{poly}(\lambda, \log L)$ size. This implies that the size of U is also $\mathsf{polylog}(L)$.

We first recall the definition of a universal sampler from [HJK+16], which we refer to as a bounded universal sampler. For now, we focus on selective, one-time security. Compared with [HJK+16], we explicitly give the size bound as an input to Setup, instead of fixing it in advance. We also allow Sample to take as input some public random coins $\boldsymbol{u}$, of length $p(\lambda, |\mathcal{D}|)$ bits, where p is some polynomial and $\mathcal{D}$ is the distribution being sampled from.

Definition 5.1 (Bounded Universal Sampler). *Let $p(\lambda, X)$ be a polynomial. A (bounded)* universal sampler *is a pair of algorithms* $(\mathsf{Setup}, \mathsf{Sample})$ *with the following syntax:*

1. Setup *is a PPT algorithm taking as input the security parameter* $\mathbb{1}^\lambda$ *and a polynomial bound* $L(\lambda)$*. The output is a sampler* U*.*
2. Sample *is a deterministic algorithm taking as input a sampler* U*, a distribution* $\mathcal{D}(\lambda)$ *with circuit size* $|\mathcal{D}| \leq L(\lambda)$ *and a random string* $\boldsymbol{u} \in \{0,1\}^{p(\lambda,|\mathcal{D}|)}$*. The output is a sample* R*.*

We say that the US satisfies selective one-time security if there exists a PPT simulator Sim *such that, for every polynomial* $L(\lambda)$ *and distribution* $\mathcal{D}$ *with* $|\mathcal{D}| \leq L(\lambda)$*, no PPT adversary can distinguish between*

$$\left\{ U, \boldsymbol{u}, R \;\middle|\; \begin{array}{l} U \xleftarrow{\$} \mathsf{Setup}(\mathbb{1}^\lambda, L) \\ \boldsymbol{u} \xleftarrow{\$} \{0,1\}^{p(\lambda,|\mathcal{D}|)} \\ R \leftarrow \mathsf{Sample}(U, \mathcal{D}, \boldsymbol{u}) \end{array} \right\} \quad \textit{and} \quad \left\{ U, \boldsymbol{u}, R \;\middle|\; \begin{array}{l} R \xleftarrow{\$} \mathcal{D} \\ (\boldsymbol{u}, U) \xleftarrow{\$} \mathsf{Sim}(\mathbb{1}^\lambda, L, \mathcal{D}, R) \end{array} \right\}$$

The definition of selective, one-time security states that, for any distribution $\mathcal{D}$ fixed ahead of time, $R = \mathsf{Sample}(U, \mathcal{D}, \boldsymbol{u})$ is indistinguishable from a random sample from $\mathcal{D}$. Furthermore, the pair $(U, \boldsymbol{u})$ leaks no information in addition to R. We also point out that for bounded universal samplers, the length of $\boldsymbol{u}$ does not need to depend on $\mathcal{D}$, we can simply set it to $p(\lambda, L)$. Finally, we notice that the selective, one-time universal sampler of [HJK+16] is a bounded universal sampler where $p(\lambda, X) = 0$.

We also consider the following succinctness property (which is not satisfied by [HJK+16]).

Definition 5.2 (Succinct, Bounded Universal Sampler). *We say that a bounded US is* succinct *if the circuit size of* $\mathsf{Setup}(\mathbb{1}^\lambda, L)$ *is* $|\mathsf{Setup}(\mathbb{1}^\lambda, L)| \leq \mathsf{poly}(\lambda, \log L)$*.*

In our final goal of an unbounded universal sampler, we remove the size constraint on $\mathcal{D}$.

Definition 5.3 (Unbounded Universal Sampler). *An* unbounded universal sampler *is defined the same way as a bounded US, except that (1) the* Setup *algorithm omits the* $L(\lambda)$ *argument, and (2) the* Sample *algorithm only requires that* $|\mathcal{D}| = \mathsf{poly}(\lambda)$ *(and still* $\boldsymbol{u} \in \{0,1\}^{p(\lambda,|\mathcal{D}|)}$*).*

Randomness extractability in universal samplers. We now introduce a new property, which is needed later for our party-dynamic distributed US (Section 6). We say that a universal sampler is *randomness extractable* if knowing the random coins used for generating the sampler parameters allows us to retrieve the randomness used to produce the sampled output. In other words, if R is the universal sampler output for a distribution $\mathcal{D}$, given the randomness used to compute the CRS, we can extract ρ such that $R = \mathcal{D}(\mathbb{1}^\lambda; \rho)$. Since we use this property

to learn information about the samples produced by an adversarially generated sampler, we ask extractability to hold for every choice of the coins used by the adversary. Furthermore, we allow the extractor to simulate the public random coins $\boldsymbol{u}$ used to generate R from the sampler, giving the option of inserting a trapdoor to help.

Definition 5.4 (Randomness Extractable Universal Sampler). *Suppose that* $(\mathsf{Setup}, \mathsf{Sample})$ *is a universal sampler, let* $M(\lambda)$ *denote the bit-length of the randomness needed by* Setup*. We say that* $(\mathsf{Setup}, \mathsf{Sample})$ *is* randomness extractable *if there exists a PPT algorithm* $\mathsf{Extract}$ *such that, for any* $\rho_0 \in \{0,1\}^{M(\lambda)}$*, polynomial* $L(\lambda)$ *and supported distribution* $\mathcal{D}(\mathbb{1}^\lambda)$*, it holds that*

$$\Pr\left[\mathcal{D}(\mathbb{1}^\lambda;\, \rho) = R \;\middle|\; \begin{array}{l} U \leftarrow \mathsf{Setup}(\mathbb{1}^\lambda, L;\, \rho_0) \\ (\rho, \boldsymbol{u}) \xleftarrow{\$} \mathsf{Extract}(\mathbb{1}^\lambda, \mathcal{D}, L, \rho_0) \\ R \leftarrow \mathsf{Sample}(U, \mathcal{D}, \boldsymbol{u}) \end{array}\right] = 1 - \mathsf{negl}(\lambda)$$

and the following distributions are indistinguishable

$$\left\{\boldsymbol{u}, \rho_0 \middle| \boldsymbol{u} \xleftarrow{\$} \{0,1\}^{p(\lambda, |\mathcal{D}|)}\right\}, \quad \left\{\boldsymbol{u}, \rho_0 \middle| (\rho, \boldsymbol{u}) \xleftarrow{\$} \mathsf{Extract}(\mathbb{1}^\lambda, \mathcal{D}, L, \rho_0)\right\}.$$

We define the randomness extractability property also for unbounded universal samplers. In such case, we need to modify the syntax: we remove $L(\lambda)$ *from the inputs of* Setup *and* $\mathsf{Extract}$*.*

We note that the construction of [HJK+16] is randomness extractable, because the Setup algorithm samples a PRF key that is used to generate the randomness ρ for the sampled output. In cases where the property does not obviously hold, though, we note that it can be obtained without loss of generality. Indeed, if $(\mathsf{Setup}, \mathsf{Sample})$ is not randomness extractable, we can easily build a universal sampler $(\mathsf{Setup}', \mathsf{Sample}')$ that is randomness extractable. The new scheme works exactly as $(\mathsf{Setup}, \mathsf{Sample})$ but the random coins $\boldsymbol{u}$ are slightly longer: they now encode a public key pk for a PKE scheme with pseudorandom public keys[13]. Instead of directly sampling from a distribution $\mathcal{D}$, the new universal sampler samples from the distribution that outputs $(R, \mathsf{Enc}_{\mathsf{pk}}(\rho))$ where $R = \mathcal{D}(\mathbb{1}^\lambda; \rho)$. The extractor can generate the random coins $\boldsymbol{u}$ such that it knows the secret key for pk, which it uses as a trapdoor to learn the randomness ρ used to generate R.

5.1 Our Succinct Universal Sampler

We now present a succinct universal sampler based on polynomially secure iO and SSB hash functions. We refer back to Section 2.2 for an informal overview of the techniques and their motivation. Below, we give the detailed construction and a brief description.

[13] PKE with pseudorandom public keys is easily built from standard assumptions such as DDH or LWE.

Detailed construction. The formal description of our succinct universal sampler is presented in Fig. 6. The description of the unobfuscated program we designed is in Fig. 7.

Recall that $L(\lambda)$ is the bound on the circuit size of the distribution $\mathcal{D}$. Let $m(\lambda)$ be an upper bound on the bit-length of a garbled gate. The construction uses the following ingredients:

- SSB hash function $\mathsf{SSB} = (\mathsf{Gen}, \mathsf{Hash}, \mathsf{Open}, \mathsf{Verify})$, with hash length $\ell_{\mathsf{Hash}}(\lambda)$. Used to compress the circuit of $\mathcal{D}$, and the random coins $\boldsymbol{u}$, into short digests.
- iO scheme iO.
- Puncturable PRF (F_1, Punct_1): maps $2\ell_{\mathsf{Hash}}$-bit nonces into two λ-bit strings gk and k. The key gk is used for randomness in garbling, while k is the authenticated encryption key.
- Puncturable PRF (F_2, Punct_2): maps $(\log L)$-bit nonces into $2m(\lambda)$ λ-bit strings $(y_1^0, y_1^1, \ldots, y_m^0, y_m^1)$. This PRF is used to generate randomness for the encryption scheme in the trapdoor.
- Garbling and evaluation functions $\mathsf{GC.Garble}, \mathsf{GC.Eval}$

We now expand upon the flavour of Yao's garbled circuits we use. Given a λ-bit garbling key gk, we define the function $G \leftarrow \mathsf{GC.Garble}(1\!\!1^\lambda, g, \mathsf{gk})$, which outputs a garbling of the gate g using gk as source of randomness. Formally, the labels of any wire w connected to g are obtained as $(k_w^0, k_w^1) \leftarrow F(\mathsf{gk}, w)$. If g is an input gate, the algorithm also assigns a pseudorandom value for that input, given by the bit $b_g \leftarrow F(\mathsf{gk}, g)$, and then outputs only the label associated with that bit, i.e. $X_g \leftarrow \mathsf{En}(b_g, e_g)$ where e_g represent the encoding information of the input gate g. If instead, g is an XOR or an AND gate, the permutation applied on the ciphertexts in the garbling is $\tau_g \leftarrow F(\mathsf{gk}, g)$[14]. This is all the randomness the garbler needs.

The encryption scheme used in the trapdoor encrypts an $m(\lambda)$-bit message x using a nonce $i \in [L]$ and the key k, and works as follows. It first computes $(y_1^0, y_1^1, \ldots, y_m^0, y_m^1) \leftarrow F_2(k, i)$, and then outputs the string $(v^1, v^2, \ldots, v^m)$ where $v^j = y_j^0$ if $x_j = 0$, $v^j = y_j^1$ otherwise. Decryption is performed by reversing the operations. If there exists an index j such that $v^j \notin \{y_j^0, y_j^1\}$, decryption fails. The nonce i will correspond to the index of the gate we want to garble.

The sampler parameters output by Sample contain an obfuscation of the program $\mathcal{P}_{\mathsf{SUS}}$ (Fig. 7). When generating a sample from a distribution $\mathcal{D}$, with auxiliary random coins $\boldsymbol{u}$, the program is run on input a hash h of the random coins, and a hash z of the circuit description of $\mathcal{D}$. Recall that $\boldsymbol{u}$ and $\mathcal{D}$ themselves are too long to be input to the program directly. The program also takes as input a gate index i, and the corresponding gate g', and also the i-th portion of the random coins, denoted v.

First, the program checks the hashes are valid, to prevent malicious queries for inconsistent gates or random coins, and then it generates the garbling key gk and trapdoor encryption key k. It then checks for a trapdoor, outputting the embedded message if one exists, and otherwise, outputs the garbling of gate g'.

[14] We assume that F generates sufficiently long outputs. We truncate the excess bits.

Succinct Universal Sampler

Let $p(\lambda, X) = \lambda \cdot m(\lambda) \cdot X$ where $m(\lambda)$ is the output size of Garble.

$\mathsf{Setup}\big(1\!\!1^\lambda, L(\lambda)\big)$:

1. $\mathsf{hk} \xleftarrow{\$} \mathsf{SSB.Gen}(1\!\!1^\lambda, L, 0)$
2. $K \xleftarrow{\$} \{0,1\}^\lambda$
3. $\mathsf{SUSProg} \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}_{\mathsf{SUS}}[K, \mathsf{hk}])$
4. Output $U := (\mathsf{hk}, \mathsf{SUSProg})$

$\mathsf{Sample}(U = (\mathsf{hk}, \mathsf{SUSProg}), \mathcal{D}, \boldsymbol{u})$:

Let $\boldsymbol{g} := (g_i)_{i \in [|\mathcal{D}|]}$ be a description of the gates of $\mathcal{D}$. Let u_i be the i-th $\big(\lambda \cdot m(\lambda)\big)$-bit block of $\boldsymbol{u}$.

1. $h \leftarrow \mathsf{SSB.Hash}(\mathsf{hk}, \boldsymbol{u})$
2. $z \leftarrow \mathsf{SSB.Hash}(\mathsf{hk}, \boldsymbol{g})$
3. $\forall i \in [|\mathcal{D}|]: \quad \pi_i \xleftarrow{\$} \mathsf{SSB.Open}(\mathsf{hk}, \boldsymbol{u}, i)$
4. $\forall i \in [|\mathcal{D}|]: \quad \pi'_i \xleftarrow{\$} \mathsf{SSB.Open}(\mathsf{hk}, \boldsymbol{g}, i)$
5. $\forall i \in [|\mathcal{D}|]: \quad G_i \leftarrow \mathsf{SUSProg}(h, z, u_i, g_i, \pi_i, \pi'_i)$ (see Fig. 7)
6. output $R \leftarrow \mathsf{GC.Eval}(\boldsymbol{G})$

Fig. 6: A succinct universal sampler

$\mathcal{P}_{\mathsf{SUS}}[K, \mathsf{hk}]$

Hardcoded: A PRF key K and an SSB hash key hk.
Input: Hashes h and z, index $i \in [L]$, random string v, gate g' and SSB proofs π and π'.

1. $b \leftarrow \mathsf{SSB.Verify}\big(\mathsf{hk}, h, i, v, \pi\big)$
2. $b' \leftarrow \mathsf{SSB.Verify}\big(\mathsf{hk}, z, i, g', \pi'\big)$
3. If $b = 0$ or $b' = 0$, output $\perp$.
4. $(\mathsf{gk}, k) \leftarrow F_1\big(K, (h, z)\big)$
5. $(y_1^0, y_1^1, \ldots, y_m^0, y_m^1) \leftarrow F_2\big(k, i\big)$
6. For every $j \in [m]$ define

$$x_j \leftarrow \begin{cases} 0 & \text{if } y_j^0 = v^j, \\ 1 & \text{if } y_j^1 = v^j, \\ \perp & \text{otherwise.} \end{cases}$$

7. If $x_j \in \{0,1\}$ for every $j \in [m]$, output x.
8. Otherwise, output $\mathsf{Garble}(1\!\!1^\lambda, g', \mathsf{gk})$

Fig. 7: The unobfuscated succinct universal sampler program

Theorem 5.5. *Let $L(\lambda)$ be a polynomial. If $\mathsf{SSB} = (\mathsf{Gen}, \mathsf{Hash}, \mathsf{Open}, \mathsf{Verify})$ is an SSB hash function, iO is an indistinguishability obfuscator, (F_1, Punct_1) and (F_2, Punct_2) are puncturable PRFs and Yao's garbled circuits are secure, the construction in Fig. 6 is a bounded universal sampler satisfying selective one-time security and randomness extractability.*

Moreover, suppose that iO is an obfuscator for the class of circuits of size $s(\lambda)$. If the circuits for $\mathsf{SSB.Gen}$, $\mathsf{SSB.Verify}$ and for the evaluation of F_2 on punctured and unpunctured keys have $\mathsf{poly}(\lambda, \log L)$ size and the circuit for iO has $\mathsf{poly}(\lambda, s)$ size, the universal sampler in Fig. 6 is succinct.

We observe that the SSB construction of [HW15], puncturable PRFs based on GGM [KPTZ13, BW13, BGI14, GGM86] and most of the iO schemes satisfy the properties for succinctness. We also point out that $L(\lambda)$, actually, does not need to be polynomial. Unfortunately, however, in order to satisfy selective one-time security for a super-polynomial $L(\lambda)$, we need to rely on a subexponentially secure iO scheme and on a subexponentially hiding SSB hash function.

Proof. We prove selective one-time security independently on whether $L(\lambda)$ is polynomial or not. We do this through a series of computationally indistinguishable hybrids. The only difference between the two cases will be that the number of hybrids will be polynomial if and only if $L(\lambda)$ is polynomial. If $L(\lambda)$ is super-polynomial, we need subexponentially secure primitives in order to achieve indistinguishability in spite of the exponential number of hybrids.

We mark changes using red font. Let $\mathcal{D}$ be the distribution addressed by the selective, one-time security game. Let $\boldsymbol{g}$ be the circuit describing $\mathcal{D}$. In the proof, we define $\widehat{h}$ and $\widehat{z}$ as $\mathsf{SSB.Hash}(\mathsf{hk}, \boldsymbol{u})$ and $\mathsf{SSB.Hash}(\mathsf{hk}, \boldsymbol{g})$ respectively. We always assume that the key K is sampled uniformly over $\{0,1\}^\lambda$.

Hybrid 0. This corresponds to the left distribution in Def. 5.1. The adversary is provided with an honestly generated $U = (\mathsf{hk}, \mathsf{SUSProg}) \xleftarrow{\$} \mathsf{Setup}\big(\mathbb{1}^\lambda, L(\lambda)\big)$, a random sting $\boldsymbol{u} \xleftarrow{\$} \{0,1\}^{p(\lambda, |\mathcal{D}|)}$ and $R \leftarrow \mathsf{Sample}(U, \mathcal{D}, \boldsymbol{u})$.

Hybrid 1. In this hybrid, the challenger punctures the PRF key K in $(\widehat{h}, \widehat{z})$ and hardcodes the result in $\mathsf{SUSProg}$. Furthermore, it programs the correct output in $\mathsf{SUSProg}$ by storing $(\widehat{\mathsf{gk}}, \widehat{k}) \leftarrow F_1\big(K, (\widehat{h}, \widehat{z})\big)$ in it. Notice that by the correctness of the puncturable PRF F_1, the input-output behaviour of $\mathcal{P}_{\mathsf{SUS}}$ remains the same. So, by the security of iO, Hybrid 1 is indistinguishable from Hybrid 0.

More formally, the computations performed by the challenger for the generation of $\mathsf{SUSProg}$ become the following (all the other operations in the game remain the same).

1. $\overline{K} \leftarrow \mathsf{Punct}_1\big(K, (\widehat{h}, \widehat{z})\big)$
2. $(\widehat{\mathsf{gk}}, \widehat{k}) \leftarrow F_1\big(K, (\widehat{h}, \widehat{z})\big)$
3. $\mathsf{SUSProg} \xleftarrow{\$} \mathsf{iO}(\mathbb{1}^\lambda, \mathcal{P}^1_{\mathsf{SUS}}[\overline{K}, \mathsf{hk}, \widehat{h}, \widehat{z}, \widehat{\mathsf{gk}}, \widehat{k}])$ (see Fig. 8)

Hybrid 2. In this hybrid, the challenger samples the keys $\widehat{\mathsf{gk}}$ and $\widehat{k}$ uniformly in $\{0,1\}^\lambda$ instead of computing $F_1\big(K, (\widehat{h}, \widehat{k})\big)$. Notice that this hybrid is indistinguishable from Hybrid 1 by the security of the puncturable PRF F_1.

$\mathcal{P}^1_{\mathsf{SUS}}[\overline{K}, \mathsf{hk}, \widehat{h}, \widehat{z}, \widehat{\mathsf{gk}}, \widehat{k}]$

Hardcoded: A punctured PRF key $\overline{K}$, an SSB hash key hk, the hashes $\widehat{h}$ and $\widehat{z}$ and the keys $(\widehat{\mathsf{gk}}, \widehat{k})$.
Input: Hashes h and z, index $i \in [L]$, random string v, gate g' and SSB proofs π and π'.

1. $b \leftarrow \mathsf{SSB.Verify}(\mathsf{hk}, h, i, v, \pi)$
2. $b' \leftarrow \mathsf{SSB.Verify}(\mathsf{hk}, z, i, g', \pi')$
3. If $b = 0$ or $b' = 0$, output $\perp$.
4. If $h = \widehat{h}$ and $z = \widehat{z}$, set $\mathsf{gk} \leftarrow \widehat{\mathsf{gk}}$ and $k \leftarrow \widehat{k}$.
5. Otherwise, $(\mathsf{gk}, k) \leftarrow F_1\big(\overline{K}, (h, z)\big)$.
6. $(y_1^0, y_1^1, \ldots, y_m^0, y_m^1) \leftarrow F_2(k, i)$
7. For every $j \in [m]$ define

$$x_j \leftarrow \begin{cases} 0 & \text{if } y_j^0 = v^j, \\ 1 & \text{if } y_j^1 = v^j, \\ \perp & \text{otherwise.} \end{cases}$$

8. If $x_j \in \{0, 1\}$ for every $j \in [m]$, output x.
9. Otherwise, output $\mathsf{Garble}(\mathbb{1}^\lambda, g', \mathsf{gk})$

Fig. 8: The unobfuscated succinct universal sampler program – Hybrid 1

More formally, the operations performed by the challenger for the generation of $\mathsf{SUSProg}$ become the following.

1. $\overline{K} \leftarrow \mathsf{Punct}_1\big(K, (\widehat{h}, \widehat{z})\big)$
2. $\widehat{\mathsf{gk}} \xleftarrow{\$} \{0,1\}^\lambda$
3. $\widehat{k} \xleftarrow{\$} \{0,1\}^\lambda$
4. $\mathsf{SUSProg} \xleftarrow{\$} \mathsf{iO}(\mathbb{1}^\lambda, \mathcal{P}^1_{\mathsf{SUS}}[\overline{K}, \mathsf{hk}, \widehat{h}, \widehat{z}, \widehat{\mathsf{gk}}, \widehat{k}])$ (see Fig. 8)

We now introduce a counter l and we initially set it to 0. For each value assumed by l, we proceed from Hybrid $3.l.0$ to Hybrid $3.l.4$. We then increment l by 1 and we repeat the procedure starting from Hybrid $3.(l+1).0$. We stop when $l = |\mathcal{D}| + 1$. At that point, we move to Hybrid $4.l.0$ keeping l unvaried.

Hybrid $3.l.0$. In this hybrid, the challenger makes the hash key hk statistically binding at index l. Furthermore, the challenger changes the program $\mathcal{P}_{\mathsf{SUS}}$. Specifically, if the inputs h and z coincide with $\widehat{h}$ and $\widehat{z}$ and $i < l$, the program tries now to decrypt v using $\widehat{k}$. If the operation succeeds, $\mathcal{P}_{\mathsf{SUS}}$ outputs the result, otherwise, it outputs $\perp$. All the rest, including the generation of $\boldsymbol{u}$, remains as in the previous hybrid.

Notice that if $l = 0$, i cannot be smaller than l, so the new block of code is never executed. Moreover, hk was already statistically binding in 0. Since the input-output behaviour of the program remains unvaried, by the security of iO,

$\mathcal{P}^2_{\mathsf{SUS}}[\overline{K}, \mathsf{hk}, \widehat{h}, \widehat{z}, \widehat{\mathsf{gk}}, \widehat{k}, l]$

Hardcoded: A punctured PRF key $\overline{K}$, an SSB hash key hk, the hashes $\widehat{h}$ and $\widehat{z}$, the keys $(\widehat{\mathsf{gk}}, \widehat{k})$ and the index l.
Input: Hashes h and z, index $i \in [L]$, random string v, gate g' and SSB proofs π and π'.

1. $b \leftarrow \mathsf{SSB.Verify}(\mathsf{hk}, h, i, v, \pi)$
2. $b' \leftarrow \mathsf{SSB.Verify}(\mathsf{hk}, z, i, g', \pi')$
3. If $b = 0$ or $b' = 0$, output $\perp$.
4. If $h = \widehat{h}$ and $z = \widehat{z}$, set $\mathsf{gk} \leftarrow \widehat{\mathsf{gk}}$ and $k \leftarrow \widehat{k}$.
5. Otherwise, $(\mathsf{gk}, k) \leftarrow F_1(\overline{K}, (h, z))$.
6. $(y_1^0, y_1^1, \ldots, y_m^0, y_m^1) \leftarrow F_2(k, i)$
7. For every $j \in [m]$ define

$$x_j \leftarrow \begin{cases} 0 & \text{if } y_j^0 = v^j, \\ 1 & \text{if } y_j^1 = v^j, \\ \perp & \text{otherwise.} \end{cases}$$

8. If $x_j \in \{0,1\}$ for every $j \in [m]$, output x.
9. If $h = \widehat{h}$ and $z = \widehat{z}$ and $i < l$, output $\perp$.
10. Otherwise, output $\mathsf{Garble}(1\!\!1^\lambda, g', \mathsf{gk})$

Fig. 9: The unobfuscated succinct universal sampler program – Hybrid $3.l.0$.

no PPT adversary can distinguish between Hybrid 2 and Hybrid 3.0.0. If instead $l > 0$, Hybrid $3.l.0$ is identical to Hybrid $3.(l-1).4$.

Formally, the challenger produces $\mathsf{SUSProg}$ by computing

$$\mathsf{SUSProg} \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^2_{\mathsf{SUS}}[\overline{K}, \mathsf{hk}, \widehat{h}, \widehat{z}, \widehat{\mathsf{gk}}, \widehat{k}, l]) \quad \text{(see Fig. 9)}.$$

Hybrid $3.l.1$. In this hybrid, the challenger punctures $\widehat{k}$ in l and hardcodes the punctured key in $\mathsf{SUSProg}$. Moreover, it computes $\widehat{G}_l \leftarrow \mathsf{Garble}(1\!\!1^\lambda, g_l, \widehat{\mathsf{gk}})$ and, for every $j \in [m]$ and $b \in \{0,1\}$, sets

$$\widehat{y}_j^b \leftarrow u_l^j \quad \text{if } b = \widehat{G}_l^j, \qquad \widehat{y}_j^b \xleftarrow{\$} \{0,1\}^\lambda \quad \text{otherwise.}$$

Finally, it hardcodes $(\widehat{y}_1^0, \widehat{y}_1^1, \ldots, \widehat{y}_m^0, \widehat{y}_m^1)$ into the program. Such a tuple will be used by $\mathsf{SUSProg}$ when $(h, z, i) = (\widehat{h}, \widehat{z}, l)$, instead of computing $F_2(\widehat{k}, l)$. The last modification we apply to $\mathsf{SUSProg}$ is that when $(h, z, i) = (\widehat{h}, \widehat{z}, l)$ and the decryption of v in the trapdoor fails, the program outputs $\perp$. The code of the unobfuscated version of $\mathsf{SUSProg}$ can be found in Fig. 10.

We argue that the new program maintains the same input-output behaviour as in the previous hybrid. As a consequence, by the security of iO, no PPT adversary can distinguish between Hybrid $3.l.0$ and $3.l.1$. We analyse the inputs case by case:

– If $(h, z, i) \neq (\hat{h}, \hat{z}, l)$, the input-output behaviour of $\mathcal{P}^2_{\mathsf{SUS}}$ (see Fig. 9) and $\mathcal{P}^3_{\mathsf{SUS}}$ (see Fig. 10) are the same by the correctness of puncturable PRFs. Indeed, line 6 of $\mathcal{P}^3_{\mathsf{SUS}}$ is never run. Moreover, line 10 is run only if it was run in $\mathcal{P}^2_{\mathsf{SUS}}$.
– If $(h, z, i) = (\hat{h}, \hat{z}, l)$ but $(v, g') \neq (u_l, g_l)$, with overwhelming probability over the randomness of hk, both $\mathcal{P}^2_{\mathsf{SUS}}$ and $\mathcal{P}^3_{\mathsf{SUS}}$ output $\perp$. This is because the SSB key is binding at position l.
– If $(h, z, i, v, g') = (\hat{h}, \hat{z}, l, u_l, g_l)$ but either π or π' does not verify, both $\mathcal{P}^2_{\mathsf{SUS}}$ and $\mathcal{P}^3_{\mathsf{SUS}}$ output $\perp$.
– If $(h, z, i, v, g') = (\hat{h}, \hat{z}, l, u_l, g_l)$ and both π and π' verify, the output of $\mathcal{P}^3_{\mathsf{SUS}}$ is $\widehat{G}_l$. Indeed, for every $j \in [m]$, we have $x_j = \widehat{G}^j_l$ as

$$\widehat{y}_j^{\widehat{G}^j_l} = u_l^j = v^j.$$

Moreover, for every $j \in [m]$, we have $\widehat{y}_j^{1-\widehat{G}^j_l} \neq u_l^j$ with overwhelming probability. The output of $\mathcal{P}^2_{\mathsf{SUS}}$ was the same. Indeed, since u_l was sampled independently of $\widehat{k}$, with overwhelming probability, there existed a $j \in [m]$ such that $u_l^j \notin \{y_j^0, y_j^1\}$ (we recall that y_j^0, y_j^1 were computed by evaluating the PRF F_2 with key $\widehat{k}$ in position l).

Formally, the challenger generates $\mathsf{SUSProg}$ as follows.

1. $\widehat{k} \xleftarrow{\$} \{0,1\}^\lambda$
2. $\overline{k} \leftarrow \mathsf{Punct}_2(\widehat{k}, l)$
3. $\widehat{G}_l \leftarrow \mathsf{Garble}(1\!\!1^\lambda, g_l, \widehat{\mathsf{gk}})$
4. $\forall j \in [m]: \quad \widehat{y}_j^{\widehat{G}^j_l} \leftarrow u_l^j$
5. $\forall j \in [m]: \quad \widehat{y}_j^{1-\widehat{G}^j_l} \xleftarrow{\$} \{0,1\}^\lambda$
6. $\mathsf{SUSProg} \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^3_{\mathsf{SUS}}[\overline{K}, \mathsf{hk}, \widehat{h}, \widehat{z}, \widehat{\mathsf{gk}}, \overline{k}, l, (\widehat{y}_j^b)_{j,b}])$ (see Fig. 10)

Hybrid 3.l.2. In this hybrid, the challenger generates all the values $(\widehat{y}_j^b)_{j,b}$ as

$$(\widehat{y}_1^0, \widehat{y}_1^1, \ldots, \widehat{y}_m^0, \widehat{y}_m^1) \leftarrow F_2(\widehat{k}, l).$$

Moreover, for every $j \in [m]$, the challenger sets $u_l^j \leftarrow \widehat{y}_j^b$ where $b = \widehat{G}^j_l$. Observe that this hybrid is indistinguishable from Hybrid 3.l.1 by the security of the puncturable PRF F_2. Formally, the challenger now generates $\mathsf{SUSProg}$ and u_l as follows:

1. $\widehat{k} \xleftarrow{\$} \{0,1\}^\lambda$
2. $\overline{k} \leftarrow \mathsf{Punct}_2(\widehat{k}, l)$
3. $\widehat{G}_l \leftarrow \mathsf{Garble}(1\!\!1^\lambda, g_l, \widehat{\mathsf{gk}})$
4. $(\widehat{y}_j^b)_{j,b} \leftarrow F_2(\widehat{k}, l)$
5. $\forall j \in [m]: \quad u_l^j \leftarrow \widehat{y}_j^{\widehat{G}^j_l}$

$\mathcal{P}^3_{\mathsf{SUS}}[\overline{K}, \mathsf{hk}, \widehat{h}, \widehat{z}, \widehat{\mathsf{gk}}, \overline{k}, l, (\widehat{y}^b_j)_{j,b}]$

Hardcoded: A punctured PRF keys $\overline{K}$ and $\overline{k}$, an SSB hash key hk, the hashes $\widehat{h}$ and $\widehat{z}$, the key $\widehat{\mathsf{gk}}$, the index l and the values $(\widehat{y}^b_j)_{j,b}$.
Input: Hashes h and z, index $i \in [L]$, random string v, gate g' and SSB proofs π and π'.

1. $b \leftarrow \mathsf{SSB.Verify}(\mathsf{hk}, h, i, v, \pi)$
2. $b' \leftarrow \mathsf{SSB.Verify}(\mathsf{hk}, z, i, g', \pi')$
3. If $b = 0$ or $b' = 0$, output $\perp$.
4. If $h = \widehat{h}$ and $z = \widehat{z}$, set $\mathsf{gk} \leftarrow \widehat{\mathsf{gk}}$ and $k \leftarrow \overline{k}$.
5. Otherwise, $(\mathsf{gk}, k) \leftarrow F_1(\overline{K}, (h, z))$.
6. If $h = \widehat{h}$ and $z = \widehat{z}$ and $i = l$, set $y^b_j \leftarrow \widehat{y}^b_j$ for every $j \in [m]$ and $b \in \{0, 1\}$.
7. Otherwise, compute $(y^0_1, y^1_1, \ldots, y^0_m, y^1_m) \leftarrow F_2(k, i)$.
8. For every $j \in [m]$ define

$$x_j \leftarrow \begin{cases} 0 & \text{if } y^0_j = v^j, \\ 1 & \text{if } y^1_j = v^j, \\ \perp & \text{otherwise.} \end{cases}$$

9. If $x_j \in \{0, 1\}$ for every $j \in [m]$, output x.
10. If $h = \widehat{h}$ and $z = \widehat{z}$ and $i \leq l$, output $\perp$.
11. Otherwise, output $\mathsf{Garble}(1\!\!1^\lambda, g', \mathsf{gk})$

Fig. 10: The unobfuscated succinct universal sampler program – Hybrid $3.l.1$

6. $\mathsf{SUSProg} \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^3_{\mathsf{SUS}}[\overline{K}, \mathsf{hk}, \widehat{h}, \widehat{z}, \widehat{\mathsf{gk}}, \overline{k}, l, (\widehat{y}^b_j)_{j,b}])$ (see Fig. 10)

Hybrid $3.l.3$. In this hybrid, rather than obfuscating $\mathcal{P}^3_{\mathsf{SUS}}$ (see Fig. 10), the challenge generates $\mathsf{SUSProg}$ using $\mathcal{P}^2_{\mathsf{SUS}}$ (see Fig. 9) hardcoding $l + 1$ instead of l. Specifically, it sets

$$\mathsf{SUSProg} \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^2_{\mathsf{SUS}}[\overline{K}, \mathsf{hk}, \widehat{h}, \widehat{z}, \widehat{\mathsf{gk}}, \widehat{k}, l + 1]).$$

The generation of u_l remains as in Hybrid $3.l.2$. We observe that the input-output behaviour of $\mathsf{SUSProg}$ remains the same as in the previous hybrid. Indeed, the changes can affect only executions where $(h, z, i) = (\widehat{h}, \widehat{z}, l)$. Since in the previous hybrid $(\widehat{y}^b_j)_{j,b} = F_2(\widehat{k}, l)$, the program $\mathsf{SUSProg}$ behaves as before even in the above case. We conclude that Hybrid $3.l.2$ and Hybrid $3.l.3$ are indistinguishable by the security of iO.

Hybrid $3.l.4$. In this hybrid, the challenger makes hk statistically binding at index $l + 1$

$$\mathsf{hk} \xleftarrow{\$} \mathsf{SSB.Gen}(1\!\!1^\lambda, L, l + 1).$$

We conclude that Hybrid $3.l.4$ is indistinguishable from Hybrid $3.l.3$ by the hiding property of SSB hashing.

At this point, we go to Hybrid $3.(l+1).0$ incrementing l by 1. If $l = |\mathcal{D}| + 1$, notice that $\boldsymbol{u}$ hides now an encryption of a garbling of $\boldsymbol{G}$. In this case, we proceed to Hybrid $4.l.0$ keeping l unvaried. We then continue to Hybrid $4.l.1$. At that point, we increment again l and we move on to Hybrid $4.(l+1).0$. We stop when $l = L(\lambda)$.

Hybrid $4.l.0$. This hybrid is identical to Hybrid $3.l.3$: the challenger increments by 1 the threshold l stored in $\mathsf{SUSProg}$. Specifically, when $(h, z) = (\widehat{h}, \widehat{z})$, $i = l$ and the decryption in the trapdoor fails, the new program $\mathcal{P}_{\mathsf{SUS}}$ immediately outputs $\perp$. Previously, instead, $\mathsf{SUSProg}$ garbled the provided gate using $\widehat{\mathsf{gk}}$. We observe that hk is statistically binding at index l, so, when $i = l$ the input-output behaviour of $\mathsf{SUSProg}$ remains the same as before. Indeed, with overwhelming probability, there exists no (g', π') such that $\mathsf{SSB.Verify}(\mathsf{hk}, \widehat{z}, l, g', \pi') = 1$. We conclude that if $l = |\mathcal{D}| + 1$, Hybrid $4.l.0$ is indistinguishable from Hybrid $3.(l-1).4$ by the security of iO. Furthermore, for the same reason, Hybrid $4.l.0$ is indistinguishable from Hybrid $4.(l-1).1$ for $l > |\mathcal{D}| + 1$.

Formally, the challenger now generates $\mathsf{SUSProg}$ as

$$\mathsf{SUSProg} \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^2_{\mathsf{SUS}}[\overline{K}, \mathsf{hk}, \widehat{h}, \widehat{z}, \widehat{\mathsf{gk}}, \widehat{k}, l+1]) \quad \text{(see Fig. 9)}.$$

Hybrid $4.l.1$. This hybrid is identical to Hybrid $3.l.4$: the challenger makes hk statistically binding at index $l+1$

$$\mathsf{hk} \xleftarrow{\$} \mathsf{SSB.Gen}(1\!\!1^\lambda, L, l+1).$$

We conclude that Hybrid $4.l.1$ is indistinguishable from Hybrid $4.l.0$ by the hiding property of SSB hashing.

At this point, we increment l. If $l < L(\lambda)$, we move again to Hybrid $4.l.0$, otherwise we proceed to Hybrid 5.

Hybrid 5. In this hybrid, we notice that $\mathsf{SUSProg}$ does not use $\widehat{\mathsf{gk}}$ anymore. So, we remove it from the program. Notice that the input-output behaviour of $\mathsf{SUSProg}$ remains unvaried, so Hybrid 5 is indistinguishable from Hybrid $4.L.1$ by the security of obfuscation.

Formally, the challenger generates now $\mathsf{SUSProg}$ as

$$\mathsf{SUSProg} \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^4_{\mathsf{SUS}}[\overline{K}, \mathsf{hk}, \widehat{h}, \widehat{z}, \widehat{k}]) \quad \text{(see Fig. 11)}.$$

Hybrid 6. We observe that the program $\mathsf{SUSProg}$ contains no information about $\widehat{\mathsf{gk}}$. Therefore, in this hybrid, the challenger generates the garbled circuit hidden in $\boldsymbol{u}$ using true randomness instead of extracting it from $\widehat{\mathsf{gk}}$ using a PRF. We conclude that this hybrid is indistinguishable from Hybrid 5 by the PRF security.

Let $\ell_{\mathcal{D}}$ denote the bit-length of the randomness need by $\mathcal{D}$. Formally, the challenger generates now $\boldsymbol{u}$ as follows:

1. $\widehat{k} \xleftarrow{\$} \{0,1\}^\lambda$
2. $(\widehat{G}, e, d) \xleftarrow{\$} \mathsf{Garble}(1\!\!1^\lambda, \mathcal{D})$

$\mathcal{P}^4_{\mathsf{SUS}}[\overline{K}, \mathsf{hk}, \widehat{h}, \widehat{z}, \widehat{k}]$

Hardcoded: A punctured PRF key $\overline{K}$, an SSB hash key hk, the hashes $\widehat{h}$ and $\widehat{z}$ and the key $\widehat{k}$.
Input: Hashes h and z, index $i \in [L]$, random string v, gate g' and SSB proofs π and π'.

1. $b \leftarrow \mathsf{SSB.Verify}(\mathsf{hk}, h, i, v, \pi)$
2. $b' \leftarrow \mathsf{SSB.Verify}(\mathsf{hk}, z, i, g', \pi')$
3. If $b = 0$ or $b' = 0$, output $\bot$.
4. If $h = \widehat{h}$ and $z = \widehat{z}$, set $k \leftarrow \widehat{k}$.
5. Otherwise, $(\mathsf{gk}, k) \leftarrow F_1(\overline{K}, (h, z))$.
6. $(y_1^0, y_1^1, \ldots, y_m^0, y_m^1) \leftarrow F_2(k, i)$
7. For every $j \in [m]$ define

$$x_j \leftarrow \begin{cases} 0 & \text{if } y_j^0 = v^j, \\ 1 & \text{if } y_j^1 = v^j, \\ \bot & \text{otherwise.} \end{cases}$$

8. If $x_j \in \{0,1\}$ for every $j \in [m]$, output x.
9. If $h = \widehat{h}$ and $z = \widehat{z}$, output $\bot$.
10. Otherwise, output $\mathsf{Garble}(1\!\!1^\lambda, g', \mathsf{gk})$

Fig. 11: The unobfuscated succinct universal sampler program – Hybrid 5

3. $r \xleftarrow{\$} \{0,1\}^{\ell_{\mathcal{D}}}$
4. $X \leftarrow \mathsf{En}(r, e)$
5. $(\widehat{G}_i)_{i \in [|\mathcal{D}|]} \leftarrow (X, \widehat{G}, d)$
6. $\forall i \in [|\mathcal{D}|]: \quad (y_{i,j}^b)_{j,b} \leftarrow F_2(\widehat{k}, i)$
7. $\forall i \in [|\mathcal{D}|]$ and $j \in [m]: \quad u_i^j \leftarrow y_{i,j}^b$ where $b = \widehat{G}_i^j$.

Hybrid 7. This hybrid corresponds to the right distribution in Def. 5.1. The challenger generates now the garbled circuit hidden in $\boldsymbol{u}$ using the simulator $\mathsf{GC.Sim}(1\!\!1^\lambda, \mathsf{struct}(\mathcal{D}), R)$ where $R \xleftarrow{\$} \mathcal{D}$. By the security of garbled circuits, Hybrid 6 and Hybrid 7 are indistinguishable. Formally, the operations performed by the SUS simulator $\mathsf{Sim}(1\!\!1^\lambda, L, \mathcal{D}, R)$ are the following.

1. $\mathsf{hk} \xleftarrow{\$} \mathsf{SSB.Gen}(1\!\!1^\lambda, L, L)$
2. $\widehat{k} \xleftarrow{\$} \{0,1\}^\lambda$
3. $(\widehat{G}, X, d) \xleftarrow{\$} \mathsf{GC.Sim}(1\!\!1^\lambda, \mathsf{struct}(\mathcal{D}), R)$
4. $(\widehat{G}_i)_{i \in [|\mathcal{D}|]} \leftarrow (X, \widehat{G}, d)$
5. $\forall i \in [|\mathcal{D}|]: \quad (y_{i,j}^b)_{j,b} \leftarrow F_2(\widehat{k}, i)$
6. $\forall i \in [|\mathcal{D}|]$ and $j \in [m]: \quad u_i^j \leftarrow y_{i,j}^b$ where $b = \widehat{G}_i^j$.
7. $\widehat{h} \leftarrow \mathsf{SSB.Hash}(\mathsf{hk}, \boldsymbol{u})$

8. $\widehat{z} \leftarrow \mathsf{SSB.Hash}(\mathsf{hk}, \boldsymbol{g})$
9. $K \xleftarrow{\$} \{0,1\}^\lambda$
10. $\overline{K} \leftarrow \mathsf{Punct}_1\big(K, (\widehat{h}, \widehat{z})\big)$
11. $\mathsf{SUSProg} \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, \mathcal{P}^4_{\mathsf{SUS}}[\overline{K}, \mathsf{hk}, \widehat{h}, \widehat{z}, \widehat{k}])$ (see Fig. 11)
12. Output $(\mathsf{hk}, \mathsf{SUSProg})$.

Succinctness. We analyse the size of the unobfuscated programs $\mathcal{P}_{\mathsf{SUS}}$, $\mathcal{P}^1_{\mathsf{SUS}}$, $\mathcal{P}^2_{\mathsf{SUS}}$, $\mathcal{P}^3_{\mathsf{SUS}}$ and $\mathcal{P}^4_{\mathsf{SUS}}$. We observe that the only parts that depend on L are the input $i \in [L]$, the SSB proofs π and π', their verification, the evaluation of the puncturable PRF F_2, the hash key hk, the punctured key $\overline{k}$ and the index l. Also the description of the gate g' and the circuit for Garble may depend on L as length of the identifiers of the wires grows with L. The dependency is however logarithmic. We notice that the size of of i and l is $\log L$. Also the comparisons that are sometimes computed between i and l and $|\mathcal{D}|$ can be computed using $O(\log L)$ gates.

By hypothesis, the circuits needed for the evaluation of $F_2(k, i)$ and $F_2(\overline{k}, i)$ have $O(\log L)$ size. We conclude that also the size of $\overline{k}$ is $O(\log L)$.

Finally, by the assumptions on SSB, the size of the circuit describing $\mathsf{SSB.Gen}$ and $\mathsf{SSB.Verify}$ is also $O(\log L)$. As a consequence, the size of hk, π and π' is also $O(\log L)$. We conclude that all programs $\mathcal{P}_{\mathsf{SUS}}, \mathcal{P}^1_{\mathsf{SUS}}, \mathcal{P}^2_{\mathsf{SUS}}, \mathcal{P}^3_{\mathsf{SUS}}$ and $\mathcal{P}^4_{\mathsf{SUS}}$ have size smaller than $q(\lambda, \log L)$ for a certain polynomial $q(\lambda, X)$.

Now, let iO be an indistinguishability obfuscator for the class of circuits having size smaller than $q(\lambda, \log L)$. Notice that such class contains all the programs we used in the hybrids. We know that the size of the circuit describing iO is $\mathsf{poly}(\lambda, q(\lambda, \log L))$. We conclude that the size of the circuit describing $\mathsf{Setup}\big(1\!\!1^\lambda, L(\lambda)\big)$ is $\mathsf{poly}(\lambda, \log L)$.

Randomness Extractability. The extractor is provided with $\mathcal{D}$, $L(\lambda)$ and the random coins ρ_0 fed into Setup. The coins allow us to retrieve the PRF key K and the hash key hk used in the construction.

The extractor can sample a random $\boldsymbol{u} \xleftarrow{\$} \{0,1\}^{p(\lambda, |\mathcal{D}|)}$. With overwhelming probability, the execution of $\mathsf{Sample}(U, \mathcal{D}, \boldsymbol{u})$ does not activate the trapdoor in $\mathsf{SUSProg}$. We can argue for this using entropy. We observe that for every $i \in [L]$, the Yao entropy of u_i is $2\lambda \cdot m(\lambda)$, so, for every compressor-decompressor pair (c, d) where c has an $\ell(\lambda)$-bit output, we have

$$\Pr[d\big(c(u_i)\big) = u_i] \leq \frac{2^{\ell(\lambda)}}{2^{2\lambda \cdot m(\lambda)}} + \mathsf{negl}(\lambda).$$

Now consider the circuit c that on input u_i, performs the same operations as in $\mathcal{P}_{\mathsf{SUS}}$ (see Fig. 7) and outputs (k, x). In particular, the compressor c generates K and hk using ρ_0 as randomness. Then, it hashes $\boldsymbol{u}$ and the circuit for $\mathcal{D}$ obtaining h and z. Finally it runs $\mathcal{P}_{\mathsf{SUS}}$ on input (h, z) along with i and u_i[15]. We also

[15] The compressor can skip the verification of the SSB proofs.

consider the decompressor d that on input (k, x) computes $(y_1^0, y_1^1, \ldots, y_m^0, y_m^1) \leftarrow F_2(k, i)$ and outputs, for every $j \in [m]$, y_j^b where $b = x_j$. If such b does not exists for any j, the decompressor outputs $\perp$.

We notice that c uses $\boldsymbol{u}$ in its code without knowing anything about it except for its i-th entry u_i. We fix the values $(u_j)_{j \neq i}$ so that $\Pr[d\big(c(u_i)\big) = u_i]$ is maximal. Observe that the output size of c is $\ell(\lambda) = \lambda + m(\lambda)$.

The probability that the $\boldsymbol{u}$ generated by the extractor triggers the trapdoor in $\mathsf{SUSProg}$ when used in conjunction with the index i is smaller than the probability that $d\big(c(u_i)\big) = u_i$. In other words, it is bounded by

$$\Pr[d\big(c(u_i)\big) = u_i] \leq \frac{2^{\lambda + m(\lambda)}}{2^{2\lambda \cdot m(\lambda)}} + \mathsf{negl}(\lambda).$$

We observe that the RHS of the above equation is negligible.

Since, the probability of triggering the trapdoor is negligible, it means that the outputs of $\mathsf{SUSProg}$ are generated by directly garbling $\mathcal{D}$ and using gk as source of randomness. We notice that also the bits input into the circuit are generated using gk. The garbling key gk can be computed as $(\mathsf{gk}, k) \leftarrow F_1\big(K, (h, z)\big)$. So, the extractor is able to retrieve the randomness used to generate the sample with overwhelming probability. □

5.2 Building Unbounded Universal Samplers from Polynomially Secure Primitives

We now explain how we can use succinct universal samplers to build an unbounded universal sampler. By combining the result in this section with Theorem 5.5, we conclude that it is possible to design unbounded universal samplers based on polynomially secure primitives only (including iO).

The construction is rather simple: we start from a succinct universal sampler having bound $\widehat{L}(\lambda)$. This sampler immediately allows to sample from any distribution of size smaller than $\widehat{L}(\lambda)$. If instead our distribution $\mathcal{D}(1\!\!1^\lambda)$ has size $s(\lambda) > L(\lambda)$, we use the succinct universal sampler to generate a succinct universal sampler with bound $2\widehat{L}(\lambda)$. If $s(\lambda) \leq 2\widehat{L}(\lambda)$, we can now use the new universal sampler to compute the output, otherwise, we repeat the operation, generating another universal sampler with twice as big circuit bound. Since $\mathcal{D}(1\!\!1^\lambda)$ is efficiently samplable, we are sure that we stop after a polynomial number of iterations. We formalise our idea in Fig. 12.

Theorem 5.6. *If* $(\mathsf{SUS.Setup}, \mathsf{SUS.Sample})$ *is a succinct universal sampler satisfying selective, one-time security, then the construction* $(\mathsf{UUS.Setup}, \mathsf{UUS.Sample})$ *in Fig. 12 is an unbounded universal sampler satisfying selective, one-time security. Moreover, if* $(\mathsf{SUS.Setup}, \mathsf{SUS.Sample})$ *is randomness extractable, also* $(\mathsf{UUS.Setup}, \mathsf{UUS.Sample})$ *is randomness extractable.*

Proof. We start by arguing that $\widehat{L}(\lambda)$ exists.

Unbounded Universal Sampler

Let $\widehat{L}(\lambda)$ be a polynomial such that, for every $i \in \mathbb{N}$ and $\lambda \in \mathbb{N}$, $\left|\mathsf{SUS.Setup}\left(1\!\!1^\lambda, 2^{i+1} \cdot \widehat{L}(\lambda)\right)\right| \leq 2^i \cdot \widehat{L}(\lambda)$.

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

1. output $U \xleftarrow{\$} \mathsf{SUS.Setup}\left(1\!\!1^\lambda, \widehat{L}(\lambda)\right)$

$\mathsf{UUS.Sample}(U, \mathcal{D}(1\!\!1^\lambda), \boldsymbol{u})$: Define $t := \lceil \log(|\mathcal{D}|/L) \rceil$. For every $i \in [t]$, define $\mathcal{D}_i$ to be the circuit describing $\mathsf{SUS.Setup}(1\!\!1^\lambda, 2^i \cdot \widehat{L})$. Rewrite $\boldsymbol{u}$ as $(u_0, u_1, \ldots, u_t)$ where u_{i-1} is a random $p(\lambda, |\mathcal{D}_i|)$-bit string and u_t is $p(\lambda, |\mathcal{D}|)$-bit long.

1. $U_0 \leftarrow U$
2. For $i = 1, 2, \ldots, t$: $\quad U_i \leftarrow \mathsf{SUS.Sample}(U_{i-1}, \mathcal{D}_i, u_{i-1})$
3. Output $R \leftarrow \mathsf{SUS.Sample}(U_t, \mathcal{D}, u_t)$

Fig. 12: An unbounded universal sampler

Claim 5.1. *There exists a polynomial $\widehat{L}(\lambda)$ such that, for every $i \in \mathbb{N}$ and $\lambda \in \mathbb{N}$,*

$$\left|\mathsf{SUS.Setup}\left(1\!\!1^\lambda, 2^{i+1}\widehat{L}(\lambda)\right)\right| \leq 2^i \cdot \widehat{L}(\lambda).$$

Proof of the claim. Since $(\mathsf{SUS.Setup}, \mathsf{SUS.Sample})$ is succinct, we know that there exists a polynomial $q(\lambda, L)$ such that

$$|\mathsf{SUS.Setup}(1\!\!1^\lambda, L)| \leq q(\lambda, \log L)$$

for sufficiently large λ and L. Define $c := \deg q(X, Y) + 1$, we know that

$$|\mathsf{SUS.Setup}(1\!\!1^\lambda, L)| \leq q(\lambda, \log L) \leq (\lambda \cdot \log L)^c$$

for $\lambda \geq \lambda_0$ and $L \geq L_0$. As a consequence, for every $\lambda \geq \lambda_0$, $i \in \mathbb{N}$ and $L \geq L_0$,

$$|\mathsf{SUS.Setup}(1\!\!1^\lambda, 2^{i+1} \cdot L)| \leq q\big(\lambda, \log(2^{i+1} \cdot L)\big) \leq \big(\lambda \cdot \log(2^{i+1} \cdot L)\big)^c.$$

Now, define $L'(\lambda) := \lambda^{c+1}$ and observe that

$$\frac{\big(\lambda \cdot \log\big(2^{i+1} \cdot L'(\lambda)\big)\big)^c}{2^i \cdot L'(\lambda)} = \frac{\big((c+1) \cdot \log \lambda + i + 1\big)^c}{2^i \cdot \lambda} \leq \frac{\big((c+1) \cdot \log \lambda + (i+1) \cdot 2^{-i/c}\big)^c}{\lambda}$$

We observe that $\lim_{i \to \infty} (i+1) \cdot 2^{-i/c} = 0$, to there exists a constant d such that

$$\frac{\big(\lambda \cdot \log\big(2^{i+1} \cdot L'(\lambda)\big)\big)^c}{2^i \cdot L'(\lambda)} \leq \frac{\big((c+1) \cdot \log \lambda + d\big)^c}{\lambda} \xrightarrow[\lambda \to \infty]{} 0.$$

As a consequence, there exists $\lambda_1 \in \mathbb{N}$ such that $\big(\lambda \cdot \log\big(2^{i+1} \cdot L'(\lambda)\big)\big)^c \leq 2^i \cdot L'(\lambda)$ for every $\lambda \geq \lambda_1$ and $i \in \mathbb{N}$. We define $\widehat{L}(\lambda) := L'(\lambda + \overline{\lambda})$ where

$$\overline{\lambda} = \min\{\lambda \in \mathbb{N} | \lambda \geq \lambda_0, \lambda \geq \lambda_1, L'(\lambda) \geq L_0\}.$$

Notice that $L'(\lambda)$ is an increasing function, so $\widehat{L}(\lambda) \geq L_0$ for every $\lambda \in \mathbb{N}$. ■

The above claim shows that the succinct universal sampler U_i can sample from the distribution $\mathcal{D}_{i+1}$. We observe that $t = \mathsf{polylog}(\lambda)$. Indeed, we know that $|\mathcal{D}(\mathbb{1}^\lambda)|$ is smaller than a polynomial $s(\lambda)$, so $t \leq \log s(\lambda) - \log \widehat{L}(\lambda) + 1 = \mathsf{polylog}(\lambda)$. We also notice that all the bounds input in $\mathsf{SUS.Setup}$ are smaller than $2^t \cdot \widehat{L}(\lambda) = \mathsf{poly}(\lambda)$. In other words, all the bounds input into $\mathsf{SUS.Setup}$ are polynomial.

Claim 5.2. *The construction in Fig. 12 satisfies selective one-time security.*

Proof of the claim. We proceed with a series of $t+1$ indistinguishable hybrids. Starting from the real execution of the universal sampler and moving towards the simulated one. We present the simulator in the last stage.

Hybrid 0. This hybrid corresponds to the left distribution in Def. 5.3: we generate U as in the construction and we sample $\boldsymbol{u}$ uniformly at random.

Hybrid 1. In this hybrid, we generate U and u_0 using the SUS simulator. Specifically, we start by sampling $U_1 \xleftarrow{\$} \mathsf{SUS.Setup}\big(\mathbb{1}^\lambda, 2 \cdot \widehat{L}(\lambda)\big)$. Then, we set $(u_0, U) \xleftarrow{\$} \mathsf{SUS.Sim}(\mathbb{1}^\lambda, \widehat{L}(\lambda), \mathcal{D}_1, U_1)$. All the elements $(u_i)_{i>0}$ are sampled uniformly at random. Notice that Hybrid 1 is indistinguishable from Hybrid 0 by the selective one-time security of the succinct universal sampler.

Hybrid i for $i = 2, \ldots, t$. This hybrid is identical to Hybrid $i-1$ except that we change the way we generate U_{i-1} and u_{i-1}. Similarly to Hybrid 1, we compute

$$U_i \xleftarrow{\$} \mathsf{SUS.Setup}\big(\mathbb{1}^\lambda, 2^i \cdot \widehat{L}(\lambda)\big).$$

Then, we feed U_i in the SUS simulator

$$(u_{i-1}, U_{i-1}) \xleftarrow{\$} \mathsf{SUS.Sim}(\mathbb{1}^\lambda, 2^{i-1} \cdot \widehat{L}(\lambda), \mathcal{D}_i, U_i).$$

Notice that U_{i-1} is then immediately fed into another execution of $\mathsf{SUS.Sim}$. This hybrid is indistinguishable from Hybrid $i-1$ by the selective one-time security of the succinct universal sampler.

Hybrid $t+1$. In this hybrid, corresponding to the right distribution in Def. 5.3, we change the way in which we generate U_t and u_t. We repeat the same procedure as before. Specifically, we sample $R \xleftarrow{\$} \mathcal{D}(\mathbb{1}^\lambda)$ and we compute

$$(u_t, U_t) \xleftarrow{\$} \mathsf{SUS.Sim}(\mathbb{1}^\lambda, 2^t \cdot \widehat{L}(\lambda), \mathcal{D}, R).$$

Once again, this hybrid is indistinguishable from the previous one by the selective one-time security of the succinct universal sampler.

To summarise, the operations performed by the unbounded universal sampler simulator $\mathsf{UUS.Sim}\big(\mathbb{1}^\lambda, \mathcal{D}(\mathbb{1}^\lambda), R\big)$ are the following.

1. $t \leftarrow \lceil \log(|\mathcal{D}|/L) \rceil$
2. $(u_t, U_t) \xleftarrow{\$} \mathsf{SUS.Sim}(\mathbb{1}^\lambda, 2^t \cdot \widehat{L}(\lambda), \mathcal{D}, R)$
3. For $i = t, t-1, \ldots, 1$: $\quad (u_{i-1}, U_{i-1}) \xleftarrow{\$} \mathsf{SUS.Sim}(\mathbb{1}^\lambda, 2^{i-1} \cdot \widehat{L}(\lambda), \mathcal{D}_i, U_i)$
4. Output $\boldsymbol{u} := (u_0, u_1, \ldots, u_t)$ and U_0.

Clearly, we have that $\mathsf{UUS.Sample}(U_0, \mathcal{D}, \boldsymbol{u}) = R$. ■

Claim 5.3. *If* $(\mathsf{SUS.Setup}, \mathsf{SUS.Sample})$ *is randomness extractable, also the unbounded universal sampler* $(\mathsf{UUS.Setup}, \mathsf{UUS.Sample})$ *is randomness extractable.*

Proof of the claim. We consider the extractor $\mathsf{UUS.Extract}(\mathbb{1}^\lambda, \mathcal{D}(\mathbb{1}^\lambda), \rho_0)$ which performs the following operations

1. For $i = 1, 2, \ldots, t$: $\quad (\rho_i, u_{i-1}) \xleftarrow{\$} \mathsf{SUS.Extract}\big(\mathbb{1}^\lambda, \mathcal{D}_i(\mathbb{1}^\lambda), 2^{i-1} \cdot \widehat{L}(\lambda), \rho_{i-1}\big)$
2. $(\rho, u_t) \xleftarrow{\$} \mathsf{SUS.Extract}\big(\mathbb{1}^\lambda, \mathcal{D}(\mathbb{1}^\lambda), 2^t \cdot \widehat{L}(\lambda), \rho_t\big)$
3. output ρ and $\boldsymbol{u} := (u_0, u_1, \ldots, u_t)$.

We argue that the value $\boldsymbol{u}$ simulated by the extractor is indistinguishable from random even for an adversary knowing ρ_0, though a series of $t+1$ indistinguishable hybrids. Along the way, we also show that R, the sample from $\mathcal{D}(\mathbb{1}^\lambda)$ generated by the universal sampler, coincides with $\mathcal{D}(\mathbb{1}^\lambda; \rho)$ with overwhelming probability.

Hybrid 0. This hybrid corresponds to the real execution of the unbounded universal sampler: we generate U using ρ_0 as randomness and we sample $\boldsymbol{u}$ uniformly. We provide the adversary with $(\rho_0, \boldsymbol{u})$.

Hybrid i **for** $i = 1, 2, \ldots, t$**.** This hybrid is identical to Hybrid $i-1$ except that we change the way we generate u_{i-1}. Specifically, we compute

$$(\rho_i, u_{i-1}) \xleftarrow{\$} \mathsf{SUS.Extract}\big(\mathbb{1}^\lambda, \mathcal{D}_i(\mathbb{1}^\lambda), 2^{i-1} \cdot \widehat{L}(\lambda), \rho_{i-1}\big).$$

We know that, with overwhelming probability, we have $U_i = \mathcal{D}_i(\mathbb{1}^\lambda; \rho_i)$. Here U_i denotes the i-th succinct universal sampler generated in $\mathsf{UUS.Sample}(U, \mathcal{D}(\mathbb{1}^\lambda), \boldsymbol{u})$. Furthermore, we know that the distribution of $(\rho_0, \boldsymbol{u})$ is indistinguishable from the one in Hybrid $i-1$ thanks to the property of the SUS extractor.

Hybrid $t+1$**.** This hybrid corresponds to the simulated execution in which $\boldsymbol{u}$ is generated by the extractor. Compared to Hybrid t, we change the way in which we generate u_t. Specifically, we compute

$$(\rho, u_t) \xleftarrow{\$} \mathsf{SUS.Extract}\big(\mathbb{1}^\lambda, \mathcal{D}(\mathbb{1}^\lambda), 2^t \cdot \widehat{L}(\lambda), \rho_t\big).$$

We know that, with overwhelming probability, we have $R = \mathcal{D}(\mathbb{1}^\lambda; \rho)$. Furthermore, we know that the distribution of $(\rho_0, \boldsymbol{u})$ is indistinguishable from the one in Hybrid t thanks to the property of the SUS extractor. ■

□

6 Party-Dynamic Distributed Universal Samplers

Distributed universal samplers. In [ASY22], the authors designed an n-party distributed universal sampler (DUS). This is a particular distributed sampler that is not tailored to any specific distribution $\mathcal{D}$: the latter can be provided as input to the sampling algorithm. In particular, the messages published by the

parties are independent of $\mathcal{D}$. Abram, Scholl and Yakoubov present two types of constructions. The first one achieves weakly semi-malicious security as long as we generate only one sample and the corresponding distribution is fixed before the generation of the messages. The second one achieves active security and permits reusing the same messages to sample from multiple distributions (of bounded size). The latter can be adaptively chosen by the environment after fixing the messages. While the first construction was built in the plain model, the second one had to rely on a random oracle.

Party-dynamic distributed universal samplers. In this section, we define and construct an even more powerful DUS. Not only we allow the messages to be independent of the distribution we sample from, we also require that the messages are independent of the group of participants, setting no upper bound on their number. We call the result a *party-dynamic distributed universal sampler*.

Following the blueprint of [ASY22], we define party-dynamic DUSs in two flavours: with one-time, weakly semi-malicious security and with reusable, active security. In the first case, the construction guarantees security for only one sample and only as long as the distribution, the subset of parties taking part to the computation as well as the randomness of the corrupted players are fixed before the generation of the honest messages. Furthermore, the adversary is not allowed to deviate from the protocol except for the choice of the randomness of the corrupted parties. In the second case, instead, the adversary is free to misbehave as it pleases. It can activate parties at any point in time and reuse the same messages to sample from multiple distributions and with different subsets of participants. Furthermore, all its decisions, including the messages of the corrupted parties, can be made after seeing the honest-parties messages. The result is a very powerful primitive: every player just needs to publish a single messages on a bulletin board upon activation. No subsequent communication is needed, the players can gather in subsets (potentially more than one at the same time) and use the already published messages to sample from any distribution. All the parties in the same group are guaranteed to receive the same output without learning any additional information.

The relation between party-dynamic DUSs and random oracles. Both notions of party-dynamic distributed universal samplers are inherently tied to random oracles, the second one more than the first. In the reusable version of the primitive, the random oracle is fundamental to gain control over the adaptive choices made by the adversary. Without it, we would run into the same impossibility results described in Section 4. The other reason why we need the random oracle is instead connected to the entropy of the outputs. In particular, the size of a DUS message needs to be larger than the Yao entropy of the outputs that can be produced from it. If we consider the ideal execution of a DUS, a honest message should be able to output ideal samples[16] even if all the other players are corrupted. Since we aim to produce an independent sample from any queried

[16] With *ideal samples* we mean truly random samples from the queried distributions.

distribution and for any subset of parties involved in the computation, without random oracle, the size of the messages would blow up. This would happen even if we bounded the set of parties and the set of supported distributions. We highlight that our construction sets no such bounds.

The entropy argument described above applies, to some extent, also to one-time party-dynamic distributed samplers. Here, our setting is easier as the messages are used to create a single sample, however, the entropy of such sample can be arbitrarily big.

6.1 One-Time, Party-Dynamic DUS

In this section, we formalise the definition of one-time party-dynamic DUS with weakly semi-malicious security. We then present the first construction of this kind.

To get around the entropy problem highlighted in the above paragraph, we adopt the same approach used for UUSes: we split the messages into a small structured part and a long random string of bits. We require the structured part to be independent of the sampled distributions and the group of participants. The size of the random string is instead allowed to change based on the number of parties and the distribution we want to sample from. In the end, the total entropy in the construction will be sufficient to achieve security, but since we can build the unstructured part of the messages using a random oracle (or any other source of public randomness), all the information sent by the players will be independent of the group of participants and the distributions. We therefore obtain exactly what we aimed for.

Definition 6.1 (One-time, party-dynamic distributed universal sampler). *Let $p_{\mathcal{D}}(\lambda, X)$ be a polynomial for any efficiently samplable distribution $\mathcal{D}(\mathbb{1}^\lambda)$. A* party-dynamic, distributed universal sampler *is a pair of PPT algorithms* $(\mathsf{Gen}, \mathsf{Sample})$ *with the following syntax:*

1. Gen *takes as input the security parameter $\mathbb{1}^\lambda$. The output is the distributed sampler message U. We assume that* Gen *needs $M(\lambda)$ bits of randomness.*
2. Sample *is a deterministic algorithm taking as input a set of parties $\mathcal{P}$ of any size $n = \mathsf{poly}(\lambda)$, a distribution $\mathcal{D}(\mathbb{1}^\lambda)$ of circuit size $\mathsf{poly}(\lambda)$, n messages $(U_i)_{i\in\mathcal{P}}$, n random strings $(u_i)_{i\in[n]}$ of size $p_{\mathcal{D}}(\lambda, n)$ each. The output is a sample R.*

We say that the distributed sampler is weakly semi-maliciously, one-time secure *if there exists a PPT simulator* Sim *such that, for every set $\mathcal{P} \in 2^{\{0,1\}^*}$ of size $\mathsf{poly}(\lambda)$, every subset $C \subsetneq \mathcal{P}$ of corrupted parties, associated randomness $(r_i)_{i\in C}$, and every efficiently samplable distribution $\mathcal{D}(\mathbb{1}^\lambda)$, the following two distributions*

are computationally indistinguishable.

$$\left\{ \begin{array}{c} (U_i)_{i\in\mathcal{P}}, (u_i)_{i\in\mathcal{P}} \\ (r_i)_{i\in C}, R \end{array} \middle| \begin{array}{ll} r_i \xleftarrow{\$} \{0,1\}^{M(\lambda)} & \forall i \in H \\ U_i \leftarrow \mathsf{Gen}(\mathbb{1}^\lambda; r_i) & \forall i \in \mathcal{P} \\ u_i \xleftarrow{\$} \{0,1\}^{p_{\mathcal{D}}(\lambda, |\mathcal{P}|)} & \forall i \in \mathcal{P} \\ R \leftarrow \mathsf{Sample}\big(\mathcal{P}, (U_j)_{j\in\mathcal{P}}, (u_j)_{j\in\mathcal{P}}, \mathcal{D}\big) & \end{array} \right\}$$

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

The above definition states that even for the worst choice of the randomness of the corrupted parties, the adversary cannot distinguish between the real DUS messages and fake ones specifically crafted to output an ideal sample from $\mathcal{D}$ when used in conjunction with the corrupted messages. In other words, the construction does not reveal any information i addition to the output. We highlight that in order for this definition to work, it is fundamental that the adversary provides the corrupted randomness, the set of participant and the distribution before the honest messages are dealt. We also notice that we do not allow the adversary to control the unstructured part of the corrupted messages: we are modelling the fact that the latter will be generated by the random oracle.

Our Construction. The idea at the base of our one-time, party-dynamic DUS is very simple: we let each party publish the structured part of an unbounded universal sampler. If a subset of n participants, for any $n \in \mathbb{N}$, wants to sample from a distribution $\mathcal{D}$, they generate random n-party distributed sampler messages for the distribution $\mathcal{D}$ using their unbounded universal samplers. In particular, the UUS of party P_j is used to produce the DS message of P_j. In this way, the adversary obtains the DS messages of the honest parties without learning any other information. All is left to do is to reconstruct the distributed sampler output, which will look like a sample from $\mathcal{D}$. As long as one party is honest, no additional information about the output is revealed.

The formal description of our solution is presented in Fig. 13.

Theorem 6.2. *Suppose that* $\mathsf{UUS} = (\mathsf{Setup}, \mathsf{Sample})$ *is an unbounded universal sampler with selective, one-time security and randomness extractability. For any* $n \in \mathbb{N}$ *and efficient distribution* $\mathcal{D}$*, let* $\mathsf{DS}_n^{\mathcal{D}}$ *be an* n*-party distributed sampler for* $\mathcal{D}$ *with weak, semi-malicious security. Then, the construction in Fig. 13 is a party-dynamic distributed universal sampler with weakly semi-malicious, one-time security.*

Proof. We prove the result though a series of computationally indistinguishable hybrids. Let $\mathcal{D}$ be the efficient distribution we want to sample from. Let $\mathcal{P}$ be the subset of parties that needs to compute the sample, we denote $|\mathcal{P}|$ by n. In each hybrid, we mark the changed operations using red font.

Party-Dynamic Distributed Sampler with Weakly Semi-Malicious Security

Let $\mathsf{DS}_n^{\mathcal{D}}$ denote an n-party, weakly semi-malicious distributed sampler for the distribution $\mathcal{D}$. For every $i \in [n]$, let $\mathcal{D}'_i$ be the distribution described by $\mathsf{DS}_n^{\mathcal{D}}.\mathsf{Gen}(\mathbb{1}^\lambda, i)$. We define $p_{\mathcal{D}}(\lambda, n) := q\big(\lambda, \max_{i\in[n]}|\mathcal{D}'_i|\big)$ where q is the polynomial describing the length of the random strings input in the unbounded universal sampler UUS.

$\mathsf{Gen}(\mathbb{1}^\lambda)$:

1. Output $U \xleftarrow{\$} \mathsf{UUS.Setup}(\mathbb{1}^\lambda)$.

$\mathsf{Sample}\big(\mathcal{P}, (U_i)_{i\in\mathcal{P}}, (u_i)_{i\in\mathcal{P}}, \mathcal{D}\big)$:
Let n be $|\mathcal{P}|$.

1. Relabel the indices in $\mathcal{P}$ as $1, \ldots, n$.
2. $\forall i \in \mathcal{P}$: $\quad V_i \leftarrow \mathsf{UUS.Sample}(U_i, \mathcal{D}'_i, u_i)$.
3. output $R \leftarrow \mathsf{DS}_n^{\mathcal{D}}.\mathsf{Sample}(V_1, V_2, \ldots, V_n)$.

Fig. 13: Party-dynamic distributed sampler with one-time, weakly semi-malicious security

Hybrid 0. This hybrid corresponds to the real world: we generate the unbounded universal samplers of the honest parties according to the protocol and the random strings $(u_j)_{j\in\mathcal{P}}$ are sampled at random.

Hybrid 1. In this hybrid, we generate the messages published by the honest parties and the corresponding random strings using the unbounded universal sampler simulator. Specifically, we generate $(U_i, u_i)_{i\in\mathcal{P}}$ by performing the following operations:

1. $\forall i \in \mathcal{P} \cap C$: $\quad U_i \leftarrow \mathsf{UUS.Setup}(\mathbb{1}^\lambda; r_i)$
2. $\forall i \in \mathcal{P} \cap C$: $\quad u_i \xleftarrow{\$} \{0,1\}^{p_{\mathcal{D}}(\lambda,n)}$
3. $\forall i \in \mathcal{P} \cap H$: $\quad \widehat{V}_i \xleftarrow{\$} \mathsf{DS}_n^{\mathcal{D}}.\mathsf{Gen}(\mathbb{1}^\lambda, i)$
4. $\forall i \in \mathcal{P} \cap H$: $\quad (u_i, U_i) \xleftarrow{\$} \mathsf{UUS.Sim}(\mathbb{1}^\lambda, \mathcal{D}'_i, \widehat{V}_i)$.

Notice that, with overwhelming probability, for every $i \in \mathcal{P} \cap H$, we have

$$\widehat{V}_i = \mathsf{UUS.Sample}(U_i, \mathcal{D}'_i, u_i).$$

This hybrid is indistinguishable from the previous one by the selective one-time security of UUS.

Hybrid 2. In this hybrid, we change how we generate the random strings of the corrupted parties. Specifically, we use the unbounded universal sampler extractor for their generation. Formally, we generate $(U_i, u_i)_{i\in\mathcal{P}}$ as follows:

1. $\forall i \in \mathcal{P} \cap C$: $\quad (\rho_i, u_i) \xleftarrow{\$} \mathsf{UUS.Extract}(\mathbb{1}^\lambda, \mathcal{D}'_i, r_i)$

2. $\forall i \in \mathcal{P} \cap H :$ $\widehat{V}_i \xleftarrow{\$} \mathsf{DS}_n^{\mathcal{D}}.\mathsf{Gen}(1\!\!1^\lambda, i)$
3. $\forall i \in \mathcal{P} \cap H :$ $(u_i, U_i) \xleftarrow{\$} \mathsf{UUS.Sim}(1\!\!1^\lambda, \mathcal{D}'_i, \widehat{V}_i)$
4. $\forall i \in \mathcal{P} \cap C :$ $U_i \leftarrow \mathsf{UUS.Setup}(1\!\!1^\lambda; r_i)$.

Notice that, with overwhelming probability, for every $i \in \mathcal{P} \cap C$, we have

$$\mathsf{UUS.Sample}(U_i, \mathcal{D}'_i, u_i) = \mathsf{DS}_n^{\mathcal{D}}.\mathsf{Gen}(1\!\!1^\lambda, i;\ \rho_i).$$

This hybrid is indistinguishable from the previous one by the randomness extractability of UUS.

Hybrid 3. In this hybrid, we generate the distributed sampler messages $(\widehat{V}_i)_{i \in \mathcal{P} \cap H}$ using the simulator for $\mathsf{DS}_n^{\mathcal{D}}$. Formally, the operations we perform for the generation of $(U_i, u_i)_{i \in \mathcal{P}}$ become the following:

1. $\forall i \in \mathcal{P} \cap C :$ $U_i \leftarrow \mathsf{UUS.Setup}(1\!\!1^\lambda; r_i)$
2. $\forall i \in \mathcal{P} \cap C :$ $(\rho_i, u_i) \xleftarrow{\$} \mathsf{UUS.Extract}(1\!\!1^\lambda, \mathcal{D}'_i, r_i)$
3. $R \xleftarrow{\$} \mathcal{D}$
4. $(\widehat{V}_i)_{i \in \mathcal{P} \cap H} \xleftarrow{\$} \mathsf{DS}_n^{\mathcal{D}}.\mathsf{Sim}\big(1\!\!1^\lambda, \mathcal{P} \cap C, R, (\rho_i)_{i \in \mathcal{P} \cap C}\big)$
5. $\forall i \in \mathcal{P} \cap H :$ $(u_i, U_i) \xleftarrow{\$} \mathsf{UUS.Sim}(1\!\!1^\lambda, \mathcal{D}'_i, \widehat{V}_i)$.

Notice that with overwhelming probability the final output is now R. This hybrid is indistinguishable from Hybrid 2 by the weakly semi-malicious security of $\mathsf{DS}_n^{\mathcal{D}}$.

We summarise the operations performed by the one-time, party-dynamic simulator $\mathsf{Sim}\big(1\!\!1^\lambda, \mathcal{P}, \mathcal{P} \cap C, \mathcal{D}, R, (r_i)_{i \in \mathcal{P} \cap C}\big)$ are the following.

1. $\forall i \in \mathcal{P} \cap C :$ $U_i \leftarrow \mathsf{UUS.Setup}(1\!\!1^\lambda; r_i)$
2. $\forall i \in \mathcal{P} \cap C :$ $(\rho_i, u_i) \xleftarrow{\$} \mathsf{UUS.Extract}(1\!\!1^\lambda, \mathcal{D}'_i, r_i)$
3. $(\widehat{V}_i)_{i \in \mathcal{P} \cap H} \xleftarrow{\$} \mathsf{DS}_n^{\mathcal{D}}.\mathsf{Sim}\big(1\!\!1^\lambda, \mathcal{P} \cap C, R, (\rho_i)_{i \in \mathcal{P} \cap C}\big)$
4. $\forall i \in \mathcal{P} \cap H :$ $(u_i, U_i) \xleftarrow{\$} \mathsf{UUS.Sim}(1\!\!1^\lambda, \mathcal{D}'_i, \widehat{V}_i)$.
5. Output $(U_i, u_i)_{i \in \mathcal{P}}$.

□

6.2 Reusable, Maliciously Secure Construction

In this section, we show how we can upgrade our one-time, party-dynamic distributed universal sampler with weakly semi-malicious security so that we can reuse the messages published by the parties to sample from multiple distributions and for different subsets of parties, achieving at the same security against a fully malicious adversary. We call a construction satisfying the above properties a *reusable, party-dynamic DUS* with malicious security.

The security model. Formally, in our security model, we let the adversary choose when to make a party join the construction. At that point, the adversary can decide whether to corrupt the party or not, the state of the corruption cannot be changed afterwards. Upon activation, each party is asked to publish a single message on a bulletin board. If the party is corrupt, we let the adversary choose

the message. At any point, any subset of active parties can pool their messages on the bulletin board and use them to sample from any distribution they desire. We set no bound on the circuit size of the distribution. Furthermore, we can reuse the same messages to sample from different distributions and for different subsets of parties. We let the environment control from which distributions we sample and the corresponding subset of parties. The correctness of the protocol requires that all the honest parties belonging to the chosen subset, output the same sample. Security instead, states that the adversary cannot learn any information in addition to the outputs of the honest parties, which must be random.

Influence of the adversary. We observe that a reusable, party-dynamic distributed universal sampler with active security unavoidably allows some influence to the environment. Indeed, the adversary can always activate the corrupted parties after the honest parties, benefiting from the opportunity of choosing the malicious messages when the honest ones are already fixed. In particular, the adversary can generate the corrupted messages multiple times in its head and test them on the honest ones. In this way, it obtains multiple samples from different distributions and for different subsets of players. The adversary can then select the messages of the corrupted parties corresponding to the group of samples it likes the most. If the environment ever recreates the tested executions, the adversary is guaranteed that the honest players will output the precomputed values.

Formalising the model. We formalise the definition of reusable, party-dynamic DUS with malicious security using the UC model. In the correspondent functionality $\mathcal{F}_{\mathsf{pdDUS}}$, each candidate message for a party P_j is modelled as a different label id_j. Before taking its final decision on the corrupted messages, the adversary is allowed to test the candidates $(\mathsf{id}_j)_{j\in C}$ by querying them to the functionality along with the distribution and the subset of parties they would be used for. The answer is a random sample from the selected distribution. At the moment of P_j's activation, the adversary needs to take a binding decision, specifying a label $\widehat{\mathsf{id}}_j$. If any of the tested executions is ever recreated by the environment, the functionality outputs the test response to all honest players involved in the computation.

Definition 6.3 (Reusable, party-dynamic DUS with malicious security). *A reusable, party-dynamic distributed universal sampler with malicious security is a protocol implementing the functionality $\mathcal{F}_{\mathsf{pdDUS}}$ (see Fig. 14) against an active adversary in the UC model. Each party is required to send at most one message during its whole execution.*

To further motivate why we need a random oracle, notice that if the construction of Def. 6.3 existed in the plain model (with or without) CRS, it would be possible to construct adaptively secure universal samplers in the plain model [HJK+16]. Furthermore, they could be used to build actively secure distributed samplers that violate the results described in Section 4.

THE FUNCTIONALITY $\mathcal{F}_{\mathsf{pdDUS}}$

Initialisation. The functionality initialises the set of honest parties H, of corrupted parties C and queries Q to $\emptyset$.

Query. On input $(\mathsf{Query}, \mathcal{P}, (\mathsf{id}_j)_{j\in\mathcal{P}\setminus H}, \mathcal{D})$ from the adversary where $\mathcal{P}$ is a subset of parties, the functionality performs the following operations:

- If Q contains a tuple $(\mathcal{P}, (\mathsf{id}_j)_{j\in\mathcal{P}\setminus H}, \mathcal{D}, R)$, send R to the adversary.
- Otherwise, sample $R \xleftarrow{\$} \mathcal{D}$, store $(\mathcal{P}, (\mathsf{id}_j)_{j\in\mathcal{P}\setminus H}, \mathcal{D}, R)$ in Q and send R to the adversary.

Join. On input Join from a party P_i where $i \notin C \cup H$, the functionality waits for a message from the adversary.

- If the adversary sends $(\mathsf{corrupt}, \widehat{\mathsf{id}}_i)$, the functionality sets $C \leftarrow C \cup \{i\}$ and stores $(i, \widehat{\mathsf{id}}_i)$.
- Otherwise, it sets $H \leftarrow H \cup \{i\}$.

Sample. On input $(\mathsf{Sample}, \mathcal{P}, \mathcal{D})$ from a honest party P_i where $i \in \mathcal{P} \subseteq H \cup C$, the functionality performs the following operations

- If there exists a $j \in \mathcal{P} \cap C$ such that $\widehat{\mathsf{id}}_j = \perp$, output $\perp$ to P_i.
- If there exists a tuple $\big(\mathcal{P}, (\widehat{\mathsf{id}}_j)_{j\in\mathcal{P}\cap C}, \mathcal{D}, R\big) \in Q$, output R to P_i.
- Otherwise, sample $R \xleftarrow{\$} \mathcal{D}$, output R to P_i and store $\big(\mathcal{P}, (\widehat{\mathsf{id}}_j)_{j\in\mathcal{P}\cap C}, \mathcal{D}, R\big)$ in Q.

Fig. 14: Reusable, party-dynamic distributed universal sampler functionality

On the labelling system in $\mathcal{F}_{\mathsf{pdDUS}}$. At first, it may seem that there exist easier ways of labelling the queries in $\mathcal{F}_{\mathsf{pdDUS}}$. It turns out, however, that simpler solutions make the functionality weaker. We observe that it is important that the adversary specifies the labels fixing the outputs upon activation of the corrupted parties: if we allow the adversary to supply the chosen label at the time of sampling, the adversary can base its decision on all information received[17] since the activation of the corrupted parties. In many contexts, this would be problematic.

We also observe that abstracting every test query under a single, monolithic identifier (i.e. the identifier cannot be split into the contributions of the single parties) makes our model less expressive. Indeed, in the protocol, when the adversary fixes the messages of the corrupted parties in a subset $\mathcal{P}$, it inevitably fixes also all outputs relative to subsets $\mathcal{P}' \subseteq \mathcal{P}$. Modelling this fact using monolithic identifiers is complex.

Adaptive Universal Samplers and Randomness Extractability. The reusable, party-dynamic DUS we will present in the next section relies on an

[17] The functionality $\mathcal{F}_{\mathsf{pdDUS}}$ can be used as a resource for another protocol in which the parties are constantly interacting. Sampling may occur several rounds after the players' activation.

adaptive (bounded) universal sampler [HJK+16]. The latter differs from its selectively, one-time secure version as its CRS can be reused multiple times to sample from different distributions. Even if such distributions are adaptively chosen by the adversary after receiving the CRS, the construction is still guaranteed to reveal no information in the addition to the outputs.

Below, we recall the formal definition of adaptive universal sampler [HJK+16]. We point out that such construction can exist only in the programmable random oracle model, so, its definition is intrinsically connected to such model.

Definition 6.4 (Adaptive and bounded universal sampler). *An* adaptive and bounded universal sampler *is a pair of PPT algorithms* $(\mathsf{Setup}, \mathsf{Sample})$ *with the following syntax:*

- Setup *is a PPT algorithm taking as input the security parameter* $\mathbb{1}^\lambda$ *and a bound* $L(\lambda)$*. The output is an adaptive sampler* adU*.*
- Sample *is a deterministic algorithm having access to a random oracle* $\mathcal{H}$*. The inputs are a sampler* adU *and a distribution* $\mathcal{D}$ *having circuit size at most* $L(\lambda)$*. The output is a sample* R*.*

The sampler satisfied adaptive security if there exist PPT simulators Sim *and* SimRO *such that, for every bound* $L(\lambda)$*, no PPT adversary* $\mathcal{A}$ *can win the game* $\mathcal{G}^{\mathsf{adUS}}_{\mathcal{A}}(\mathbb{1}^\lambda)$ *(see Fig. 15) with non-negligible advantage.*

Essentially, the above definition states that the real universal sampler cannot be distinguished from a fake one which can be programmed to output ideal samples via the random oracle.

Randomness extractable unbounded universal sampler. Our reusable, party-dynamic DUS requires an additional property from adaptive distributed samplers. Specifically, we require that if we know the randomness used to generate the universal sampler, then, we can efficiently extract the randomness used to generate the outputs. We formalise the definition below. Using the same techniques in the proof of Theorem 5.5, it is easy to prove that the adaptive distributed sampler of [HJK+16] is randomness extractable.

Definition 6.5 (Randomness extractable adaptive universal sampler). *Suppose that* $(\mathsf{Setup}, \mathsf{Sample})$ *is an adaptive universal sampler with random oracle* $\mathcal{H}$*. Let* $M(\lambda)$ *denote the bit-length of the randomness needed by* Setup*. We say that* $(\mathsf{Setup}, \mathsf{Sample})$ *is* randomness extractable *if there exists a PPT algorithm* $\mathsf{Extract}$ *having access to* $\mathcal{H}$ *such that, for any PPT adversary* $\mathcal{A}$ *and polynomial* $L(\lambda)$*, it holds that*

$$\Pr\left[\mathcal{D}(\rho) = R \;\middle|\; \begin{array}{l} (\mathcal{D}, r) \xleftarrow{\$} \mathcal{A}^{\mathcal{H}}(\mathbb{1}^\lambda, L(\lambda)) \\ U \leftarrow \mathsf{Setup}(\mathbb{1}^\lambda, L(\lambda); r) \\ \rho \leftarrow \mathsf{Extract}^{\mathcal{H}}(\mathbb{1}^\lambda, \mathcal{D}, L(\lambda), r) \\ R \leftarrow \mathsf{Sample}^{\mathcal{H}}(U, \mathcal{D}) \end{array}\right] = 1 - \mathsf{negl}(\lambda)$$

The above probability is taken also over the random coins of the oracle.

THE GAME $\mathcal{G}^{\mathsf{adUS}}_{\mathcal{A}}(1\!\!1^{\lambda})$

Initialisation. The challenger instantiates a random oracle $\mathcal{H}$ and a sampling oracle $\mathcal{O}$. The latter is a equipped with a truly random function F outputting $L(\lambda)$ bits. Upon receiving any distribution $\mathcal{D}$ having circuit size at most $L(\lambda)$, the oracle $\mathcal{O}$ outputs the sample $R \leftarrow \mathcal{D}(F(\mathcal{D}))$.
Then, the challenger performs the following operations:

1. $b \xleftarrow{\$} \{0,1\}$
2. $\mathsf{adU}_0 \xleftarrow{\$} \mathsf{Setup}\big(1\!\!1^{\lambda}, L(\lambda)\big)$
3. $(\mathsf{adU}_1, \tau) \xleftarrow{\$} \mathsf{Sim}^{\mathcal{O}}(1\!\!1^{\lambda}, L(\lambda))$
4. provide $\mathcal{A}$ with adU_b.

Oracle query. On input $(\mathsf{oracle}, \mathcal{D})$ from the adversary, the challenger performs the following operations:

1. $r_0 \leftarrow \mathcal{H}(\mathcal{D})$
2. $(r_1, \tau) \xleftarrow{\$} \mathsf{SimRO}^{\mathcal{O}}(\tau, \mathcal{D})$
3. provide $\mathcal{A}$ with r_b.

Sample query. On input $(\mathsf{sample}, \mathcal{D})$ from the adversary, the challenger performs the following operations:

1. $R_0 \leftarrow \mathsf{Sample}^{\mathcal{H}}(\mathsf{adU}_0, \mathcal{D})$
2. $R_1 \leftarrow \mathcal{O}(\mathcal{D})$
3. provide $\mathcal{A}$ with R_b.

The adversary wins if it ends its execution outputting b.

Fig. 15: The adaptive universal sampler game

Building Reusable, Party-Dynamic DUS with Malicious Security. We now explain how to upgrade a one-time, party-dynamic DUS with weakly semi-malicious security into a reusable one achieving security against an active adversary.

The main challenge: from selectively chosen inputs to adaptively chosen ones. One-time, party-dynamic DUSs permit any subset of parties to sample from any distribution. The main challenge, however, is that they achieve security only if the messages are used only once and the distribution and the corresponding subset of parties are fixed before the generation of the messages. In reusable, party-dynamic DUSs, this does not happen: the environment can choose the distributions and the subsets of parties as well as the messages of the corrupted parties after seeing the honest messages. Furthermore, we reuse the messages to sample multiple times.

Reusing the same message to sample from multiple distributions and for multiple subsets of parties. Our idea is to non-interactively generate new, independent-

Reusable, Party-Dynamic Distributed Universal Sampler with Malicious Security

CRS. Provide all the parties with $\mathsf{urs} \xleftarrow{\$} \mathsf{NIZK.Gen}(1\!\!1^\lambda)$.

Join. In order to join the protocol, party P_i performs the following operations:

1. $r_i \xleftarrow{\$} \{0,1\}^{M(\lambda)}$
2. $\mathsf{adU}_i \leftarrow \mathsf{adUS.Setup}\big(1\!\!1^\lambda, L(\lambda); r_i\big)$
3. $\pi_i \xleftarrow{\$} \mathsf{NIZK.Prove}(1\!\!1^\lambda, \mathsf{urs}, \mathsf{adU}_i, r_i)$
4. Publish (adU_i, π_i) on the public bulletin board.

Sample. On input a set of parties $\mathcal{P}$ and a distribution $\mathcal{D}$, each party P_i for $i \in \mathcal{P}$ performs the following operations:

1. Retrieve the messages $(\mathsf{adU}_j, \pi_j)_{j\in\mathcal{P}}$ on the bulletin board.
2. If there exists $j \in \mathcal{P}$ such that $\mathsf{NIZK.Verify}(\mathsf{urs}, \pi_j, \mathsf{adU}_j) = 0$, output $\perp$.
3. Query $\big(\mathcal{P}, (\mathsf{adU}_j, \pi_j)_{j\in\mathcal{P}}, \mathcal{D}\big)$ to the random oracle $\mathcal{H}$. Let $h \in \{0,1\}^\lambda$ be the answer.
4. For every $j \in \mathcal{P}$, let $\mathcal{D}_{j,h}$, be the distribution that outputs (j,h) along with a random sample from $\mathsf{pdDUS.Gen}(1\!\!1^\lambda)$.
5. $\forall j \in \mathcal{P}: \quad U_j \leftarrow \mathsf{adUS.Sample}^{\mathcal{H}}(\mathsf{adU}_j, \mathcal{D}_{j,h})$
6. For every $i \in \mathcal{P}$, query $\big(\mathcal{P}, (U_j)_{j\in\mathcal{P}}, \mathcal{D}, i\big)$ to the one-time, party-dynamic DUS random oracle. Let u_i be the $p_{\mathcal{D}}(\lambda, |\mathcal{P}|)$-bit response.
7. Output $R \leftarrow \mathsf{pdDUS.Sample}\big(\mathcal{P}, (U_j)_{j\in\mathcal{P}}, (u_j)_{j\in\mathcal{P}}, \mathcal{D}\big)$.

Fig. 16: Reusable, party-dynamic DUS with malicious security

looking one-time, party-dynamic DUS messages for every distribution $\mathcal{D}$ and subset of parties $\mathcal{P}$. Clearly, it is sufficient to produce only the structured part of the messages, the generation of the unstructured part is instead entrusted to the random oracle. We achieve our goal by making each party P_j publish an adaptive universal sampler adU_j. We generate the one-time, party-dynamic DUS messages by querying the corresponding distributions. Notice that, now, we never use any of the one-time DUS messages twice.

A minor issue we encounter is that, if an adaptive universal sampler is queried multiple times with the same distribution, the output remains always the same. We solve this problem by parametrising the queried distributions with a different tag $h \in \{0,1\}^\lambda$ and the identifier j of the addressed party: the distribution labelled with (j,h) outputs h and j along with a random one-time, party-dynamic DUS message. Since all the queried distributions are now different, the adaptive universal samplers are guaranteed to output independent-looking samples.

Dealing with the adaptive choices of the adversary using a random oracle. We observe that the solution described in the previous paragraph is not secure yet. The main issue is that there is nothing that prevents the adversary from adaptively choosing the subset of parties $\mathcal{P}$, the distribution $\mathcal{D}$ and the messages of

the corrupted parties after seeing the one-time, party dynamic DUS messages of the honest players.

Following the blueprint of [HJK+16] and [ASY22], we solve this issue using the random oracle. In particular, we force the adversary to query the subset of parties, the corresponding adaptive universal samplers and the distribution it wants to sample from to the random oracle. The response is the tag $h \in \{0,1\}^\lambda$ parametrising the distributions of the one-time, party-dynamic DUS messages. In other words, the adversary cannot learn any outputs without first presenting its plans to the random oracle. In the security proof, this allows us to use the one-time, party-dynamic DUS simulator and program the final output. Specifically, we generate the honest, one-time, party-dynamic DUS messages $(U_j)_{j\in\mathcal{P}\cap H}$ by feeding an ideal sample R from $\mathcal{D}$ into the corresponding simulator. Then, we use the adaptive universal sampler programmability to make adU_j output U_j for every $j \in \mathcal{P}\cap H$. In this way, the output of the construction is guaranteed to be R.

Detecting malformed messages and extracting the randomness of the corrupted parties. We observe that the one-time, party-dynamic DUS simulator needs to be provided with the randomness of the corrupted parties. We need to find a way to extract it. In the current state, the construction suffers also from another vulnerability: nothing prevents the corrupted parties from publishing malformed and potentially malicious messages. We solve both issues using simulation-extractable NIZKs. The latter bases its security on a CRS. In many instantiations, however, the CRS is unstructured, so, we can use the random oracle to generate it without any interaction.

We modify our construction so that each party P_j, now, publishes a proof of well-formedness π_j along with its adaptive universal sampler. Before outputting any sample, the parties involved in the computation check the NIZK proofs of the other participants. If any verification fails, the players output $\perp$. We slightly change also the queries issued to the random oracle by appending the well-formedness proofs. In this way, in the security proof, we can extract the randomness used to generate $(\mathsf{adU}_j)_{j\in\mathcal{P}\setminus H}$. Using the randomness extractability of the adaptive universal samplers, we can then retrieve the random coins used for the generation of the one-time, party-dynamic DUS messages of the corrupted parties.

Formal description. The precise description of our reusable, party-dynamic DUS with malicious security is in Fig. 16. The construction relies on an adaptive universal sampler $\mathsf{adUS} = (\mathsf{Setup}, \mathsf{Sample})$ satisfying randomness extractability. We use $M(\lambda)$ to denote the bit-length of the randomness needed by $\mathsf{adUS.Setup}$. We also use a one-time, party-dynamic DUS $\mathsf{pdDUS} = (\mathsf{Gen}, \mathsf{Sample})$ satisfying weakly semi-malicious security. We denote by $\mathcal{D}_{j,h}$ the distribution that outputs the tag (j,h) along with a sample from $\mathsf{pdDUS.Gen}(1\!\!1^\lambda)$. We choose $L(\lambda)$ to be a polynomial upper bound on the circuit length of $\mathcal{D}_{j,h}$. Finally, we rely on a simulation-extractable NIZK scheme $\mathsf{NIZK} = (\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$ having

unstructured CRS. The corresponding language is

$$\left\{(U, r) \middle| U = \mathsf{adUS.Setup}\left(1\!\!1^{\lambda}, L(\lambda); r\right)\right\}.$$

THE RESOURCE $\mathcal{F}_{\mathsf{Bulletin}}$

Publish. On input $(\mathsf{Publish}, m_i)$ from party P_i, the functionality stores (i, m_i). Subsequent queries of this kind from party P_i are ignored.

Read. On input (Read, j) from a party P_i, retrieve the pair (j, m_j) if it was previously stored, and send m_j to P_i.

Fig. 17: The bulletin board resource

Finally, the construction assumes the existence of a bulletin board functionality $\mathcal{F}_{\mathsf{Bulletin}}$ (see Fig. 17). The latter permits each party P_i to publish a single message. At any point in the future, the other players can retrieve the message published by P_i without needing further communication from P_j. It is possible to implement such a functionality using blockchains.

Since there exist multiple primitives using the random oracle in our protocol, we assume that each oracle query is prepended with a description of its context. For instance, any query relative to P_j's adaptive universal sampler will be prepended with adU_j.

Theorem 6.6. *Assume that* $\mathsf{NIZK} = (\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$ *is a simulation-extractable NIZK,* $\mathsf{adUS} = (\mathsf{Setup}, \mathsf{Sample})$ *is an adaptively secure universal sampler and* $\mathsf{pdDUS} = (\mathsf{Gen}, \mathsf{Sample})$ *is a party-dynamic DUS with weakly semi-malicious, one-time security. Then, the construction in Fig. 16 is a reusable, party dynamic DUS with malicious security in the* $\mathcal{F}_{\mathsf{Bulletin}}$*-hybrid model with random oracle.*

Proof. We prove the security of the construction through a sequence of computationally indistinguishable hybrids. Hybrid 0 corresponds to the real protocol, the last hybrid corresponds to the ideal world. We present the precise description of the simulator at the end. To simplify the notation, we assume that we have access to multiple random oracles: the random oracle $\mathcal{H}$ of the construction, a random oracle $\mathcal{H}_{\mathsf{adU}}$ for every possible adaptive universal samplers adU and the one used by the one-time party-dynamic DUS $\mathcal{H}_{\mathsf{pdDUS}}$. It is easy to implement the three oracles using only one[18]. In each hybrid, we mark the changed operations using red font.

Hybrid 0. This hybrid corresponds to the real world. The simulator provides the parties with $\mathsf{urs} \xleftarrow{\$} \mathsf{NIZK.Gen}(1\!\!1^{\lambda})$. When a honest party joins, the simulator generates a reusable, party-dynamic message following the protocol and

[18] It is sufficient to prepend the label of the addressed oracle to each query.

publishes it on the bulletin board. The samples output by the honest parties are also generated as in the protocol. Finally, the random oracle queries are answered using random strings.

Hybrid 1. In this hybrid, we generate the URS and the zero-knowledge proofs of the honest parties using the simulators for the simulation-extractable NIZK. Specifically, urs is generated as

$$(\mathsf{urs}, \tau) \xleftarrow{\$} \mathsf{NIZK.Sim}_1(1\!\!1^\lambda)$$

Whenever an honest party P_i joins, the simulator generates its message (adU_i, π_i) as follows:

1. $\mathsf{adU}_i \xleftarrow{\$} \mathsf{adUS.Setup}\big(1\!\!1^\lambda, L(\lambda)\big)$
2. $\pi_i \xleftarrow{\$} \mathsf{NIZK.Sim}_2(\mathsf{urs}, \tau, \mathsf{adU}_i)$

This hybrid is indistinguishable from Hybrid 0 due to the multi-theorem zero-knowledge property of NIZK.

Hybrid 2. In this hybrid, instead of generating the adaptive samplers for the honest parties using $\mathsf{adUS.Setup}$, we use the simulator $\mathsf{adUS.Sim}$. We also use $\mathsf{adUS.SimRO}$ to simulate the queries to the corresponding random oracles. Formally, we generate the message of the honest party P_i as follow:

1. We instantiate a sampling oracle $\mathcal{O}_i$ equipping it with a random function F_i. On input a distribution $\mathcal{D}$ of size at most $L(\lambda)$, $\mathcal{O}_i$ outputs $\mathcal{D}(F_i(\mathcal{D}))$.
2. $(\mathsf{adU}_i, \tau_i) \xleftarrow{\$} \mathsf{adUS.Sim}^{\mathcal{O}_i}\big(1\!\!1^\lambda, L(\lambda)\big)$
3. $\pi_i \xleftarrow{\$} \mathsf{NIZK.Sim}_2(\mathsf{urs}, \tau, \mathsf{adU}_i)$

We reply to the oracle queries to $\mathcal{H}_{\mathsf{adU}_i}$ using $\mathsf{adUS.SimRO}^{\mathcal{O}_i}(\tau_i)$. Since adU_i has large entropy, the probability that the adversary as already queried $\mathcal{H}_{\mathsf{adU}_i}$ before is negligible. We conclude that this hybrid is indistinguishable from the previous one by the adaptive security of $\mathsf{adUS} = (\mathsf{Gen}, \mathsf{Sample})$.

We now repeat Hybrid 3.t for $t = 1, 2, \ldots, T$. Here, T denotes a polynomial upper bound on the number of oracle queries issued by the adversary.

Hybrid 3.t. We introduce some notation. Assume that

$$q := \big(\widehat{\mathcal{P}}, (\widehat{\mathsf{adU}}_j, \widehat{\pi}_j)_{j \in \widehat{\mathcal{P}}}, \widehat{\mathcal{D}}\big)$$

is the t-th random oracle query issued by the adversary where

- $\mathsf{NIZK.Verify}(\mathsf{urs}, \widehat{\pi}_j, \widehat{\mathsf{adU}}_j) = 1$ for every $j \in \widehat{\mathcal{P}}$
- $(\widehat{\mathsf{adU}}_j, \widehat{\pi}_j) = (\mathsf{adU}_j, \pi_j)$ for every $j \in \widehat{P} \cap H$.

We define the set $W \subseteq \widehat{\mathcal{P}} \setminus H$ containing the indexes j such that $(\widehat{\mathsf{adU}}_j, \widehat{\pi}_j)$ is the copy of an honest party's message on the bulletin board. Let $Z = \widehat{\mathcal{P}} \setminus (H \cup W)$ and let $\widehat{h} \in \{0,1\}^\lambda$ denote the answer of $\mathcal{H}$ to the query q. For every $j \in Z$, we define the element $\widehat{U}_j$ such that

$$\mathsf{adUS.Sample}^{\mathcal{H}_{\widehat{\mathsf{adU}}_j}}\big(\widehat{\mathsf{adU}}_j, \mathcal{D}_{j,\widehat{h}}\big) = (j, \widehat{h}, \widehat{U}_j).$$

In this hybrid, for every $j \in \widehat{\mathcal{P}} \cap H$, we change the answer of the sampling oracle $\mathcal{O}_j$ on input $\mathcal{D}_{j,\widehat{h}}$. For every $j \in W$, we also change the answer of the sampling oracle of the copied party on input $\mathcal{D}_{j,\widehat{h}}$. Specifically, instead of outputting a random sample from $\mathsf{pdDUS.Gen}(1\!\!1^\lambda)$, we use the simulator for the one-time party dynamic DUS with weakly semi-malicious security pdDUS. In such computation, we treat the parties in $W \cup (\widehat{\mathcal{P}} \cap H)$ as honest (the adversary knows nothing about how their messages have been generated).

We notice that simulator for pdDUS needs to be provided with the randomness $(\rho_j)_{j\in Z}$ used for the generation of $(\widehat{U}_j)_{j\in Z}$. We extract such randomness in two steps: first, using the simulation-extractability of NIZK, we retrieve the random coins used to generate $\widehat{\mathsf{adU}}_j$ for every $j \notin \widehat{\mathcal{P}} \cap H$. Then, using the randomness extractability of adUS, we obtain $(\rho_j)_{j\in\widehat{\mathcal{P}}\setminus H}$.

We also provide $\mathsf{pdDUS.Sim}$ with a sample R from $\widehat{\mathcal{D}}$. We obtain the latter by querying the functionality with

$$\big(\mathsf{Query}, \widehat{\mathcal{P}}, (\widehat{\mathsf{adU}}_j, \widehat{\pi}_j)_{j\in\widehat{\mathcal{P}}\setminus H}, \widehat{\mathcal{D}}\big).$$

The simulator of pdDUS provide us also with random strings $(\widehat{u}_j)_{j\in\widehat{\mathcal{P}}}$. From now on, if $(\widehat{\mathcal{P}}, (\widehat{U}_j)_{j\in\widehat{\mathcal{P}}}, \widehat{\mathcal{D}}, i)$ with $i \in \widehat{\mathcal{P}}$ is ever queried to the one-time, party-dynamic DUS oracle, we reply with $\widehat{u}_i$.

Formally, in this hybrid the simulator performs the following steps:

1. Send $\big(\mathsf{Query}, \widehat{\mathcal{P}}, (\widehat{\mathsf{adU}}_j, \widehat{\pi}_j)_{j\in\widehat{\mathcal{P}}\setminus H}, \widehat{\mathcal{D}}\big)$ to the functionality. Let R be the reply.
2. $\forall j \in Z: \quad r_j \leftarrow \mathsf{NIZK.Extract}(\mathsf{urs}, \tau, \widehat{\mathsf{adU}}_j, \widehat{\pi}_j)$
3. $\forall j \in Z: \quad \rho_j \leftarrow \mathsf{adUS.Extract}^{\mathcal{H}_{\widehat{\mathsf{adU}}_j}}\big(1\!\!1^\lambda, \mathcal{D}_{j,\widehat{h}}, L(\lambda), r_j\big)$
4. $(\widehat{U}_j, \widehat{u}_j)_{j\in\widehat{\mathcal{P}}} \xleftarrow{\$} \mathsf{pdDUS.Sim}\big(1\!\!1^\lambda, \widehat{\mathcal{P}}, Z, \widehat{\mathcal{D}}, R, (\rho_j)_{j\in Z}\big)$
5. For every $j \in \widehat{\mathcal{P}} \cap H$, if $\mathcal{D}_{j,\widehat{h}}$ is queried to $\mathcal{O}_j$, we reply with $\widehat{U}_j$.
6. For every $j \in W$ who copied the honest party P_i, if $\mathcal{D}_{j,\widehat{h}}$ is queried to $\mathcal{O}_i$, we reply with $\widehat{U}_j$.
7. If $(\widehat{\mathcal{P}}, (\widehat{U}_j)_{j\in\widehat{\mathcal{P}}}, \widehat{\mathcal{D}}, i)$ with $i \in \widehat{\mathcal{P}}$ is ever queried to $\mathcal{H}_{\mathsf{pdDUS}}$, we reply with $\widehat{u}_i$.

We notice that by the simulation extractability of NIZK, for every $j \in Z$, we have that $\widehat{\mathsf{adU}}_j = \mathsf{adUS.Setup}(1\!\!1^\lambda, L(\lambda); r_j)$. Furthermore, by the randomness extractability of adUS, for every $j \in Z$, we have that $\widehat{U}_j = \mathsf{pdDUS.Gen}(1\!\!1^\lambda; \rho_j)$. Both equations holds with overwhelming probability.

By the weakly semi-malicious, one-time security of pdDUS, we also know that

$$(\widehat{U}_j, \widehat{u}_j)_{j\in\widehat{\mathcal{P}}} \xleftarrow{\$} \mathsf{pdDUS.Sim}\big(1\!\!1^\lambda, \widehat{\mathcal{P}}, Z, \widehat{\mathcal{D}}, R, (\rho_j)_{j\in Z}\big)$$

is indistinguishable from

$$\left\{ (U_j, u_j)_{j\in\widehat{\mathcal{P}}} \;\middle|\; \begin{array}{ll} U_j \xleftarrow{\$} \mathsf{pdDUS.Gen}(1\!\!1^\lambda) & \forall j \in \widehat{\mathcal{P}} \setminus Z \\ U_j \leftarrow \widehat{U}_j & \forall j \in Z \\ u_j \xleftarrow{\$} \{0,1\}^{p_{\mathcal{D}}(\lambda,n)} & \forall j \in \widehat{\mathcal{P}} \end{array} \right\}$$

Indistinguishability holds even if the adversary knows the inputs to $\mathsf{pdDUS.Sim}$. We observe that the responses of $\mathcal{H}_{\mathsf{pdDUS}}$ that we changed look like random strings of the right size. Furthermore, with overwhelming probability, the adversary has never issued the modified queries before. This is a consequence of the fact that $\widehat{U}_j$ has $\omega(\log \lambda)$ min-entropy for every $j \in H$ (otherwise, pdDUS would not be secure). As a consequence, the adversary cannot notice that we changed the oracle responses only after receiving q.

Finally, we observe that, with overwhelming probability, any $\mathcal{O}_i$ (including $\mathcal{O}_j$) is never queried with $\mathcal{D}_{j,\widehat{h}}$ before q is sent either. Indeed, the executions of $\mathsf{adUS.Sim}$ and $\mathsf{adUS.SimRO}$ are independent of $\widehat{h}$ until the query q is issued. Moreover, for any i, $\mathsf{adUS.Sim}$ and $\mathsf{adUS.SimRO}$ can issue only a polynomial number of queries to $\mathcal{O}_i$, while $\widehat{h}$ is uniformly distributed over $\{0,1\}^\lambda$. Since the answers of $\mathcal{O}_i$ to different queries look independent, the adversary cannot detect if we change the answer of $\mathcal{O}_i$ to $\mathcal{D}_{j,\widehat{h}}$ only after receiving q. Notice also that if $i \in \widehat{\mathcal{P}}$, we are changing the answer of $\mathcal{O}_i$ to two different queries: $\mathcal{D}_{j,\widehat{h}}$ and $\mathcal{D}_{i,\widehat{h}}$.

By the above arguments, we conclude that Hybrid 3.1 is indistinguishable from Hybrid 2. Furthermore, for every $t > 1$, Hybrid 3.t is indistinguishable from Hybrid 3.$(t-1)$.

We conclude our proof by observing that, by the security of adUS, for every $j \in \widehat{\mathcal{P}} \cap H$,

$$\mathsf{adUS.Sample}^{\mathsf{SimRO}(\tau_j)}(\mathsf{adU}_j, \mathcal{D}_{j,\widehat{h}}) = \mathcal{O}_j(\mathcal{D}_{j,\widehat{h}}) = (j, \widehat{h}, \widehat{U}_j).$$

Moreover, for any $j \in W$ copying the messages of the honest party P_i, we have

$$\mathsf{adUS.Sample}^{\mathsf{SimRO}(\tau_i)}(\widehat{\mathsf{adU}}_j, \mathcal{D}_{j,\widehat{h}}) = \mathcal{O}_i(\mathcal{D}_{j,\widehat{h}}) = (j, \widehat{h}, \widehat{U}_j).$$

The above equations hold except with negligible probability. We conclude that, if the adversary activates and corrupts all the parties in $\widehat{\mathcal{P}} \setminus H$ choosing $(\widehat{\mathsf{adU}}_j, \widehat{\pi}_j)_{j \in \widehat{\mathcal{P}} \setminus H}$ as their messages, the sample output by any honest party P_j on input $(\widehat{P}, \widehat{\mathcal{D}})$ is

$$\mathsf{pdDUS.Sample}\big(\widehat{\mathcal{P}}, (\widehat{U}_j)_{j \in \widehat{\mathcal{P}}}, (\widehat{u}_j)_{j \in \widehat{\mathcal{P}}}, \widehat{\mathcal{D}}\big).$$

By the one-time security of pdDUS, the latter coincides with the sample provided by the functionality. In other words, we can let the functionality deal the outputs of the honest parties without the adversary noticing it.

We summarise the operations performed by our simulator in Fig. 18.

□

6.3 Party-Dynamic, Ideal Public-Key PCFs.

Public-key PCFs. Public-key pseudorandom correlation functions (PCFs), introduced by [OSY21] and formalised in [ASY22], are one-round protocols permitting a set of players to generate large amounts of correlated randomness in a distributed way and with minimal communication. In the construction, each party generates a key pair broadcasting the public counterpart. Using the public

THE SIMULATOR Sim

Initialisation. All the oracle queries are initially answered using random elements. The simulator keeps a log of the queries elements and if a value is queried multiple times, it replies always in the same way. The simulator generates the CRS as follows:

1. $(\mathsf{urs}, \tau) \xleftarrow{\$} \mathsf{NIZK.Sim}_1(1\!\!1^\lambda)$
2. Output urs.

Join. When a corrupted party P_j joins the protocol, the simulator performs the following operation:

1. Receive (adU_j, π_j) from the adversary.
2. If $\mathsf{NIZK.Verify}(\mathsf{urs}, \pi_j, \mathsf{adU}_j) = 0$, send $(\mathsf{corrupt}, \widehat{\mathsf{id}}_j = \bot)$ to the functionality.
3. Otherwise, send $(\mathsf{corrupt}, \widehat{\mathsf{id}}_j = (\mathsf{adU}_j, \pi_j))$.
4. Publish (adU_j, π_j) on the bulletin board on behalf of P_j.

If P_j joins instead as a honest party, the simulator performs the following steps:

1. Instantiate a sampling oracle $\mathcal{O}_j$. On input any distribution $\mathcal{D}$ of size at most $L(\lambda)$, the oracle replies with $R \xleftarrow{\$} \mathcal{D}$. The simulator keeps a log of all the responses of $\mathcal{O}_j$. If the same distribution is queried multiple times, the simulator always provides the same answer.
2. $(\mathsf{adU}_j, \tau_j) \xleftarrow{\$} \mathsf{adUS.Sim}^{\mathcal{O}_j}\big(1\!\!1^\lambda, L(\lambda)\big)$
3. $\pi_j \xleftarrow{\$} \mathsf{NIZK.Sim}_2(\mathsf{urs}, \tau, \mathsf{adU}_j)$
4. Publish (adU_j, π_j) on the bulletin board on behalf of P_j.
5. Queries to $\mathcal{H}_{\mathsf{adU}_j}$ are from now on replied using $\mathsf{adUS.SimRO}^{\mathcal{O}_j}(\tau_j)$.

Queries to $\mathcal{H}$. All queries to $\mathcal{H}$ are answered with random elements in $\{0,1\}^\lambda$. However, the simulator performs the following operations if the query is of the form $\big(\widehat{\mathcal{P}}, (\widehat{\mathsf{adU}}_j, \widehat{\pi}_j)_{j\in\widehat{\mathcal{P}}}, \widehat{\mathcal{D}}\big)$ where $\mathsf{NIZK.Verify}(\mathsf{urs}, \widehat{\pi}_j, \widehat{\mathsf{adU}}_j) = 1$ for every $j \in \widehat{\mathcal{P}}$ and $(\widehat{\mathsf{adU}}_j, \widehat{\pi}_j) = (\mathsf{adU}_j, \pi_j)$ for every $j \in \widehat{P} \cap H$. If $Z = \emptyset$, the simulator changes nothing.

1. Let $\widehat{h}$ be the answer of $\mathcal{H}$.
2. $\forall j \in Z: \quad r_j \leftarrow \mathsf{NIZK.Extract}(\mathsf{urs}, \tau, \widehat{\mathsf{adU}}_j, \widehat{\pi}_j)$
3. $\forall j \in Z: \quad \rho_j \leftarrow \mathsf{adUS.Extract}^{\mathcal{H}_{\widehat{\mathsf{adU}}_j}}\big(1\!\!1^\lambda, \mathcal{D}_{j,\widehat{h}}, L(\lambda), r_j\big)$
4. query $\big(\mathsf{Query}, \widehat{\mathcal{P}}, (\widehat{\mathsf{adU}}_j, \widehat{\pi}_j)_{j\in\widehat{\mathcal{P}}\setminus H}, \widehat{\mathcal{D}}\big)$ to the functionality. Let R be the answer.
5. $(\widehat{U}_j, \widehat{u}_j)_{j\in\widehat{\mathcal{P}}} \xleftarrow{\$} \mathsf{pdDUS.Sim}\big(1\!\!1^\lambda, \widehat{\mathcal{P}}, Z, \widehat{\mathcal{D}}, R, (\rho_j)_{j\in Z}\big)$
6. If $\mathcal{O}_j$ is ever queried with $\mathcal{D}_{j,\widehat{h}}$ for any $j \in \widehat{\mathcal{P}} \cap H$, from now on, the sampling oracle replies with $\widehat{U}_j$.
7. For every $j \in W$ who copied the honest party P_i, if $\mathcal{D}_{j,\widehat{h}}$ is queried to $\mathcal{O}_i$, we reply with $\widehat{U}_j$.
8. If the one-time, party-dynamic DUS random oracle is ever queried with $(\widehat{\mathcal{P}}, (\widehat{U}_j)_{j\in\widehat{\mathcal{P}}}, \widehat{\mathcal{D}}, i)$ for any $i \in \widehat{\mathcal{P}}$, from now on, the simulator replies with $\widehat{u}_i$.

Fig. 18: The reusable, party-dynamic DUS simulator

keys of all participants and their own secret key, the players are able to generate large amounts of correlated randomness. Clearly, each party can obtain only the material addressed to it, the other players' outputs remain secret.

An issue with entropy (again). One of the most important qualities we require from a public-key PCF is that the public keys are small (i.e. sublinear in size) compared to the amount of produced material. In this way, we can design protocols for the generation of correlated material with minimal communication complexity. This requirement conflicts however with entropy. Indeed, the size of the keys should be at least equal to the Yao entropy of the outputs. In other words, if we want to generate ideal correlated samples in the plain model, the size of the public keys must be at least linear in the size of the produced material.

Known solutions. In the last years, two solutions to the above problem were found: weakening the security definition of public-key PCFs [BCG+19] or relaying on the random oracle to non-interactively introduce entropy in the construction [ASY22]. The first line of research led to the standard definition of PCFs. The notion asks that the adversary is unable to distinguish between the real outputs of the honest parties and fake ones produced from the outputs of the corrupted players conditioned on satisfying the desired correlation rule. Not all types of correlation permit this kind of operation, so the standard notion of public-key PCFs is restricted to a particular family of functions called *reverse samplable*[19]. We also notice that this weaker definition of PCF does not prevent the adversary from having very strong influence on the protocol. For instance, a PCF can be secure even if the corrupted parties are allowed to choose their output. This might be problematic in some contexts.

The second line of research led to the definition of ideal, public-key PCFs [ASY22]. These constructions satisfy a stronger security definition: the protocol directly implements the functionality that generates the desired correlated material and distributes it to the parties. Since it is a one-round protocol, in the actively secure case, the functionality allows some limited influence in the form of providing multiple samples (a polynomial number and only the parts addressed to the corrupted parties) and letting the adversary choose which to output the the honest players. Due to the random oracle, ideal public-key PCFs do not need to be tailored to any specific correlation, the latter can be provided as input to the sampling algorithm. The big advantage of ideal public-key PCFs is also that they do not restrict to reverse samplable correlation. Furthermore, they limit the influence of the adversary. The big disadvantage is that they need a random oracle.

Generating Correlated Randomness for a Dynamically Changing Set of Parties. In [ASY22], Abram, Scholl and Yakoubov presented the first ideal public-key PCF, which was built using obfuscation and distributed samplers. The construction is tailored to a certain number of parties n, meaning that if the

[19] For instance, it is impossible to generate garbled circuits using standard PCFs.

THE FUNCTIONALITY $\mathcal{F}_{\mathsf{ideal\text{-}pkPCF}}$

Initialisation. The functionality initialises the set of honest parties H, of corrupted parties C and queries Q to $\emptyset$.

Query. On input $(\mathsf{Query}, \mathcal{P}, (\mathsf{id}_j)_{j\in\mathcal{P}\setminus H}, \mathcal{C})$ from the adversary where $\mathcal{P}$ is a subset of parties and $\mathcal{C}$ is a $|\mathcal{P}|$-party correlation function, the functionality performs the following operations:

- If Q contains a tuple $(\mathcal{P}, (\mathsf{id}_j)_{j\in\mathcal{P}\setminus H}, \mathcal{C}, (R_j)_{j\in\mathcal{P}})$, send $(R_j)_{j\in\mathcal{P}\setminus H}$ to the adversary.
- Otherwise, sample $(R_j)_{j\in\mathcal{P}} \xleftarrow{\$} \mathcal{C}$, store $(\mathcal{P}, (\mathsf{id}_j)_{j\in\mathcal{P}\setminus H}, \mathcal{C}, (R_j)_{j\in\mathcal{P}})$ in Q and send $(R_j)_{j\in\mathcal{P}\setminus H}$ to the adversary.

Join. On input Join from a party P_i where $i \notin C \cup H$, the functionality waits for a message from the adversary.

- If the adversary sends $(\mathsf{corrupt}, \widehat{\mathsf{id}}_i)$, the functionality sets $C \leftarrow C \cup \{i\}$ and stores $(i, \widehat{\mathsf{id}}_i)$.
- Otherwise, it sets $H \leftarrow H \cup \{i\}$.

Sample. On input $(\mathsf{Sample}, \mathcal{P}, \mathcal{C})$ from a honest party P_i where $i \in \mathcal{P} \subseteq H \cup C$ and $\mathcal{C}$ is a $|\mathcal{P}|$-party correlation function, the functionality performs the following operations

- If there exists a $j \in \mathcal{P} \cap C$ such that $\widehat{\mathsf{id}}_j = \perp$, output $\perp$ to P_i.
- If there exists a tuple $\left(\mathcal{P}, (\widehat{\mathsf{id}}_j)_{j\in\mathcal{P}\cap C}, \mathcal{C}, (R_j)_{j\in\mathcal{P}}\right) \in Q$, output R_i to P_i.
- Otherwise, sample $(R_j)_{j\in\mathcal{P}} \xleftarrow{\$} \mathcal{C}$, store $\left(\mathcal{P}, (\widehat{\mathsf{id}}_j)_{j\in\mathcal{P}\cap C}, \mathcal{C}, (R_j)_{j\in\mathcal{P}}\right)$ in Q and output R_i to P_i.

Fig. 19: Reusable, party-dynamic, ideal public-key PCF functionality

set of participants changes, the players are forced to restart the protocol. In this section, we formalise an even stronger definition of ideal public-key PCF in which the messages of the parties are independent of the set of participants. Following the same approach that we used for party-dynamic DUSs, we could start by considering the weaker one-time, semi-maliciously secure definition. Since we have to rely on the random oracle anyway and our construction is fairly simple, we go straight to the reusable, actively secure version.

Definition 6.7 (Reusable, party-dynamic, ideal public-key PCF with malicious security). *A reusable, party-dynamic, ideal public-key PCF with malicious security is a protocol implementing the functionality $\mathcal{F}_{\mathsf{ideal\text{-}pkPCF}}$ (see Fig. 19) against an active adversary in the UC model. Each party is required to send at most one message during its whole execution.*

Notice that the above definition follows the blueprint of Def. 6.3. As before, the adversary is free to misbehave as it pleases. The environment has total control over when to activate parties. Moreover, it is completely free to choose

from which correlation functions to sample and the set of parties involved in the computation. The messages of the honest parties can be reused to sample from multiple correlations and for different subsets of players (even more than one at the same time). The environment is also free to make these choices as well as to choose the corrupted messages after seeing the messages of the honest parties. We highlight that the state of corruption is chosen upon activation of a party and cannot be changed afterwards. So, we achieve static security.

We represent all possible messages of a corrupted player P_j via a label id_j. The functionality allows the adversary to test the messages of the corrupted parties before publishing them. This is modelled by the querying procedure: after providing the set of participants, the candidate corrupted messages and the correlation function, the functionality provides the adversary with the corresponding samples of the corrupted players, while keeping those of the honest parties secret. If the adversary decides to use the queried corrupted messages and the environment recreates the tested situation, the functionality reveals the honest samples that were previously kept secret by outputting them to the honest players. The adversary specifies the chosen message for a corrupted P_j by providing the corresponding label $\widehat{\mathsf{id}}_j$ to the functionality upon P_j's activation.

A Reusable, Party-Dynamic, Ideal Public-Key PCF. We use the simple idea of [ASY22]: we let each party publish the public counterpart of a PKE key. Let $\mathcal{C}$ the correlation function we want to sample from, let $\mathcal{P}$ be the set of participants. We use a reusable, party-dynamic DUS to sample from the distribution $\mathcal{D}_{\mathcal{C}}$ that runs $\mathcal{C}$ and, for every $j \in \mathcal{P}$, encrypts its j-th output under P_j's public key. In this way, only P_j is able to retrieve its sample.

We formally describe the construction in Fig. 20. We work in the $(\mathcal{F}_{\mathsf{pdDUS}}, \mathcal{F}_{\mathsf{Bulletin}})$-hybrid model (see Fig. 14 and Fig. 17). We also rely on an IND-CPA public key encryption scheme $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$.

Theorem 6.8. *If* $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is an IND-CPA public-key encryption scheme, the construction in Fig. 20 is a reusable, party-dynamic, ideal public-key PCF with malicious security in the* $(\mathcal{F}_{\mathsf{pdDUS}}, \mathcal{F}_{\mathsf{Bulletin}})$*-hybrid model.*

Proving Theorem 6.8 is rather straightforward. The result is immediately implied by the IND-CPA security of PKE. For this reason, we do not provide a formal proof.

Public-key PCF with master secrets. We notice that our construction has some advantages over the ideal public-key PCF of [ASY22]. Indeed, thanks to the unbounded universal sampler hidden in $\mathcal{F}_{\mathsf{pdDUS}}$, we set no bound on the circuit size of the correlation functions we sample from. As a consequence, differently from the solution of [ASY22], our ideal public key PCF supports also master secrets. We say that a correlation function has master secrets if it is parametrised by random values, one for each party, which must remain private. An example of this kind of correlation is authenticated beaver triples: each sample from the correlation is authenticated using the same MAC key. Each party holds a

A Reusable, Party-Dynamic, Ideal Public-Key PCF

Join. In order to join the protocol, each party P_i performs the following operations:

1. $(\mathsf{pk}_i, \mathsf{sk}_i) \xleftarrow{\$} \mathsf{PKE.Gen}(1\!\!1^\lambda)$
2. Send Join to $\mathcal{F}_{\mathsf{pdDUS}}$.
3. Publish pk_i on $\mathcal{F}_{\mathsf{Bulletin}}$ and keep sk_i secret.

Sample. On input a subset of players $\mathcal{P}$ where $i \in \mathcal{P}$ and a $|\mathcal{P}|$-party correlation function $\mathcal{C}$, each party P_i performs the following operations:

1. Read the public keys $(\mathsf{pk}_j)_{j\in\mathcal{P}}$ from $\mathcal{F}_{\mathsf{Bulletin}}$.
2. Let $\mathcal{D}_{\mathcal{C}}$ be the distribution that computes $(R_j)_{j\in\mathcal{P}} \xleftarrow{\$} \mathcal{C}$, derives $c_j \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk}_j, R_j)$ for every $j \in \mathcal{P}$ and outputs $(c_j)_{j\in\mathcal{P}}$.
3. Send $(\mathsf{Sample}, \mathcal{P}, \mathcal{D}_{\mathcal{C}})$ to $\mathcal{F}_{\mathsf{pdDUS}}$. If the result is $\perp$, P_i outputs $\perp$, otherwise, let $(c_j)_{j\in\mathcal{P}}$ be the result.
4. Output $R_i \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}_i, c_i)$.

Fig. 20: A reusable, party-dynamic, ideal public-key PCF

share of the key, such share must remain private. We say that a public-key PCF supports master secrets if it allows the generation of multiple samples using the same master secrets while leaking no information about the secrets of the honest parties. We notice that the master secrets are not necessarily input by the parties, they can also be sampled at random by the public-key PCF itself. We refer to [ASY22] for a more formal definition.

How to sample with master secrets. In our construction, the parties can input a special distribution $\mathcal{D}^{\mathsf{ms}}_{\mathcal{C}}$ into $\mathcal{F}_{\mathsf{pdDUS}}$: $\mathcal{D}^{\mathsf{ms}}_{\mathcal{C}}$ generates the master secrets for all parties and uses them to produce multiple samples from the correlation function. Then, it encrypts each result under the public keys of the participants. If $\mathcal{F}_{\mathsf{pdDUS}}$ is implemented using our unbounded universal sampler, the parties do not even need to retrieve all the generated samples in one go. They can indeed use the unbounded universal sampler to garble only the parts of $\mathcal{D}^{\mathsf{ms}}_{\mathcal{C}}$ that they need (for instance the part computing the first ℓ outputs). If, at a later stage, the parties need additional correlated material, they can garble and evaluate a new piece of $\mathcal{D}^{\mathsf{ms}}_{\mathcal{C}}$. The downside is that before even beginning the garbling, the parties need to hash $\mathcal{D}^{\mathsf{ms}}_{\mathcal{C}}$ and the random oracle responses. The operation requires linear computation in the amount of generated material. Our solution has also another disadvantage: the amount of correlated randomness we can generate using the same master secrets is polynomially bounded. The bound is chosen when $\mathcal{D}^{\mathsf{ms}}_{\mathcal{C}}$ is input in $\mathcal{F}_{\mathsf{pdDUS}}$. When the parties deplete their source of correlated material,

they are forced to query $\mathcal{D}_{\mathcal{C}}^{\mathsf{ms}}$ again[20]. The new batch of correlated material will however use independent master secrets.

We highlight that it is actually possible to use the techniques we just described also in the construction of [ASY22], in that case, however, the size of the PCF public keys would blow up linearly in the amount of generated correlation.

Acknowledgements

The authors would like to thank Ivan Damgård, Jesper Buus Nielsen and Sophia Yakoubov for their feedback and support.

This work has been carried out with the generous support of the Independent Research Fund Denmark (DFF) under project number 0165-00107B (C3PO), the Aarhus University Research Foundation (AUFF) and the grant MOE2019-T2-1-145, "Foundations of quantum-safe cryptography".

References

ABI+23. Damiano Abram, Amos Beimel, Yuval Ishai, Eyal Kushilevitz, and Varun Narayanan. Cryptography from Planted Graphs: Security with Logarithmic-Size Messages , 2023.

ASY22. Damiano Abram, Peter Scholl, and Sophia Yakoubov. Distributed (Correlation) Samplers: How to Remove a Trusted Dealer in One Round. In *EUROCRYPT 2022*. Springer, 2022.

AWZ23. Damiano Abram, Brent Waters, and Mark Zhandry. Security-Preserving Distributed Samplers: How to Generate any CRS in One Round without Random Oracles. In *CRYPTO 2023*. Springer, 2023.

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 *ITCS 2012*. ACM, January 2012.

BCG+19. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In *CRYPTO 2019, Part III*, LNCS. Springer, Heidelberg, August 2019.

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

Bd94. Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In *EUROCRYPT'93*, LNCS. Springer, Heidelberg, May 1994.

BFK+19. Saikrishna Badrinarayanan, Rex Fernando, Venkata Koppula, Amit Sahai, and Brent Waters. Output compression, MPC, and iO for turing machines. In *ASIACRYPT 2019, Part I*, LNCS. Springer, Heidelberg, December 2019.

[20] We can parametrise $\mathcal{D}_{\mathcal{C}}^{\mathsf{ms}}$ using a public nonce $x \in \{0,1\}^*$: we ask $\mathcal{D}_{\mathcal{C}}^{\mathsf{ms}}$ to output x along with its outputs. If we query $\mathcal{D}_{\mathcal{C}}^{\mathsf{ms}}$ for different nonces x, $\mathcal{F}_{\mathsf{pdDUS}}$ provides us with independent looking values.

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 *CRYPTO 2001*, LNCS. Springer, Heidelberg, August 2001.

BGI14. Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In *PKC 2014*, LNCS. Springer, Heidelberg, March 2014.

BHR12. Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *ACM CCS 2012*. ACM Press, October 2012.

BL20. Fabrice Benhamouda and Huijia Lin. Mr NISC: Multiparty reusable non-interactive secure computation. In *TCC 2020, Part II*, LNCS. Springer, Heidelberg, November 2020.

BP15. Nir Bitansky and Omer Paneth. ZAPs and non-interactive witness indistinguishability from indistinguishability obfuscation. In *TCC 2015, Part II*, LNCS. Springer, Heidelberg, March 2015.

BW13. Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *ASIACRYPT 2013, Part II*, LNCS. Springer, Heidelberg, December 2013.

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

CCD+20. Megan Chen, Ran Cohen, Jack Doerner, Yashvanth Kondi, Eysa Lee, Schuyler Rosefield, and abhi shelat. Multiparty generation of an RSA modulus. In *CRYPTO 2020, Part III*, LNCS. Springer, Heidelberg, August 2020.

CKL03. Ran Canetti, Eyal Kushilevitz, and Yehuda Lindell. On the limitations of universally composable two-party computation without set-up assumptions. In *EUROCRYPT 2003*, LNCS. Springer, Heidelberg, May 2003.

CLTV15. Ran Canetti, Huijia Lin, Stefano Tessaro, and Vinod Vaikuntanathan. Obfuscation of probabilistic circuits and applications. In *TCC 2015, Part II*, LNCS. Springer, Heidelberg, March 2015.

DGH+20. Nico Döttling, Sanjam Garg, Mohammad Hajiabadi, Daniel Masny, and Daniel Wichs. Two-round oblivious transfer from CDH or LPN. In *EUROCRYPT 2020, Part II*, LNCS. Springer, Heidelberg, May 2020.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO 2012*, LNCS. Springer, Heidelberg, August 2012.

FLOP18. Tore Kasper Frederiksen, Yehuda Lindell, Valery Osheter, and Benny Pinkas. Fast distributed RSA key generation for semi-honest and malicious adversaries. In *CRYPTO 2018, Part II*, LNCS. Springer, Heidelberg, August 2018.

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*. 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*. IEEE Computer Society Press, October 2013.

GGM86. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, (4), October 1986.

GO07. Jens Groth and Rafail Ostrovsky. Cryptography in the multi-string model. In *CRYPTO 2007*, LNCS. Springer, Heidelberg, August 2007.

GOS06. Jens Groth, Rafail Ostrovsky, and Amit Sahai. Non-interactive zaps and new techniques for NIZK. In *CRYPTO 2006*, LNCS. Springer, Heidelberg, August 2006.

GS18. Sanjam Garg and Akshayaram Srinivasan. A simple construction of iO for turing machines. In *TCC 2018, Part II*, LNCS. Springer, Heidelberg, November 2018.

HIJ+17. Shai Halevi, Yuval Ishai, Abhishek Jain, Ilan Komargodski, Amit Sahai, and Eylon Yogev. Non-interactive multiparty computation without correlated randomness. In *ASIACRYPT 2017, Part III*, LNCS. Springer, Heidelberg, December 2017.

HILL99. Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, (4), 1999.

HJK+16. Dennis Hofheinz, Tibor Jager, Dakshita Khurana, Amit Sahai, Brent Waters, and Mark Zhandry. How to generate and use universal samplers. In *ASIACRYPT 2016, Part II*, LNCS. Springer, Heidelberg, December 2016.

HLR07. Chun-Yuan Hsiao, Chi-Jen Lu, and Leonid Reyzin. Conditional computational entropy, or toward separating pseudoentropy from compressibility. In *EUROCRYPT 2007*, LNCS. Springer, Heidelberg, May 2007.

HMR+19. Carmit Hazay, Gert Læssøe Mikkelsen, Tal Rabin, Tomas Toft, and Angelo Agatino Nicolosi. Efficient RSA key generation and threshold paillier in the two-party setting. *Journal of Cryptology*, (2), April 2019.

HV16. Carmit Hazay and Muthuramakrishnan Venkitasubramaniam. What security can we achieve within 4 rounds? In *SCN 16*, LNCS. Springer, Heidelberg, August / September 2016.

HW15. Pavel Hubacek and Daniel Wichs. On the communication complexity of secure function evaluation with long output. In *ITCS 2015*. ACM, January 2015.

IKM+13. Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. On the power of correlated randomness in secure computation. In *TCC 2013*, LNCS. Springer, Heidelberg, March 2013.

IOZ14. Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. In *CRYPTO 2014, Part II*, LNCS. Springer, Heidelberg, August 2014.

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*, 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 *ACM CCS 2013*. ACM Press, November 2013.

KPW13. Stephan Krenn, Krzysztof Pietrzak, and Akshay Wadia. A counterexample to the chain rule for conditional HILL entropy - and what deniable encryption has to do with it. In *TCC 2013*, LNCS. Springer, Heidelberg, March 2013.

KRS15. Dakshita Khurana, Vanishree Rao, and Amit Sahai. Multi-party key exchange for unbounded parties from indistinguishability obfuscation. In *ASIACRYPT 2015, Part I*, LNCS. Springer, Heidelberg, November / December 2015.

LZ17. Qipeng Liu and Mark Zhandry. Decomposable obfuscation: A framework for building applications of obfuscation from polynomial hardness. In *TCC 2017, Part I*, LNCS. Springer, Heidelberg, November 2017.

OPWW15. Tatsuaki Okamoto, Krzysztof Pietrzak, Brent Waters, and Daniel Wichs. New realizations of somewhere statistically binding hashing and positional accumulators. In *ASIACRYPT 2015, Part I*, LNCS. Springer, Heidelberg, November / December 2015.

OSY21. Claudio Orlandi, Peter Scholl, and Sophia Yakoubov. The rise of paillier: Homomorphic secret sharing and public-key silent OT. In *EUROCRYPT 2021, Part I*, LNCS. Springer, Heidelberg, October 2021.

PS19. Chris Peikert and Sina Shiehian. Noninteractive zero knowledge for NP from (plain) learning with errors. In *CRYPTO 2019, Part I*, LNCS. Springer, Heidelberg, August 2019.

PVW08. Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In *CRYPTO 2008*, LNCS. Springer, Heidelberg, August 2008.

Rey11. Leonid Reyzin. Some notions of entropy for cryptography - (invited talk). In *ICITS 11*, LNCS. Springer, Heidelberg, May 2011.

Sha48. C. E. Shannon. A mathematical theory of communication. *Bell Sys. Tech. J.*, 27:623–656, 1948.

SW14. Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *46th ACM STOC*. ACM Press, May / June 2014.

Yao82. Andrew Chi-Chih Yao. Theory and applications of trapdoor functions (extended abstract). In *23rd FOCS*. IEEE Computer Society Press, November 1982.

Yao86. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*. IEEE Computer Society Press, October 1986.

A Additional Preliminaries

In this appendix, we present the notions we used for our proofs and constructions. In particular, we recall the definitions of *distributed samplers*, *indistinguishability obfuscation*, *puncturable PRFs*, *SSB hash functions*, *garbled circuits* and *simulation-extractable NIZKs*. Then, in Appendix A.7, we recall various definitions of entropy and basic results.

A.1 Distributed Samplers

Distributed samplers (DS) are a strong primitive allowing n parties to securely generate CRSs with a single round of interaction. Specifically, a distributed sampler for the distribution $\mathcal{D}(\mathbb{1}^\lambda)$ is a one-round protocol that generates a sample R from $\mathcal{D}(\mathbb{1}^\lambda)$ without revealing any information except R itself.

The notion was introduced for the first time by Abram, Scholl and Yakoubov in [ASY22]. In the paper, the authors show how to build the primitive from indistinguishability obfuscation and multi-key FHE. In this section, we recall their definition considering multiple adversarial models.

We start by consider security against a weakly semi-malicious adversary, i.e. a *non-rushing* adversary that, as in the semi-honest model, follows the protocol, but before beginning the execution, it chooses the random tapes of the corrupted parties as it prefers. If the adversary follows the protocol but instead chooses the randomness of the corrupted parties after seeing the honest messages, we say that we are dealing with a *strongly semi-malicious* adversary.

Definition A.1 (Weakly semi-maliciously secure distributed sampler). *Let $\mathcal{D}(\mathbb{1}^\lambda)$ be an efficiently samplable distribution. An n-party distributed sampler (DS) for $\mathcal{D}(\mathbb{1}^\lambda)$ is a pair of PPT algorithms* $(\mathsf{Gen}, \mathsf{Sample})$ *having the following syntax:*

1. Gen *is a probabilistic algorithm taking as input the security parameter $\mathbb{1}^\lambda$ and the index i of the party running it. The output is the distributed sampler message U_i of the i-th party. We assume that* Gen *needs $M(\lambda)$ bits of randomness.*
2. Sample *is a deterministic algorithm taking as input n distributed sampler messages $(U_j)_{j\in[n]}$, one for each party. The output is a sample R.*

We say that the distributed sampler is weakly semi-maliciously *secure if there exists a PPT simulator* Sim *such that, for every subset $C \subsetneq [n]$ of corrupted parties and associated randomness $(\rho_i)_{i\in C}$, the following two distributions are computationally indistinguishable.*

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

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

The security definition essentially states that even for the worst randomness choice of the corrupted parties, the honest messages leak no information except the output itself. Observe that if we run Sample over the simulated messages, the output coincides with R with overwhelming probability. Notice that any adversary corrupting no party but just listening to the conversations is always able to obtain the output. Indeed, the latter is just a deterministic function of the transcript.

It is possible to reformulate the above definition by saying that weakly semi-maliciously secure distributed sampler is a one-round protocol implementing the functionality that provides all the parties with the same sample R from $\mathcal{D}(\mathbb{1}^\lambda)$. Unfortunately, it is impossible to implement the above functionality against rushing adversaries. Indeed, after receiving the messages of the honest parties, the adversaries can always rerun the protocol in its head multiple times, changing only the messages of the corrupted parties. In this way, the attacker obtains multiple samples from $\mathcal{D}(\mathbb{1}^\lambda)$, it can therefore choose the one it likes the most and send the corresponding corrupted messages in the protocol. In other words,

the adversary can always choose the output among a set of polynomially many samples. For this reason, in the rushing setting, distributed samplers are defined as in Def. 2.1.

A.2 Indistinguishability Obfuscation

Some of the constructions presented in this work are based on indistinguishability obfuscation [BGI+01, GGH+13]. An indistinguishability obfuscator is a cryptographic primitive that on input a circuit in a certain class outputs an equivalent circuit, i.e. a circuit with *exactly* the same input-output behaviour. The operations performed by the output circuit are however so different from the original ones that it is impossible to tell how the input circuit was behaving. In the context of obfuscation, we often refer to circuits as *programs*. We recall now the formal definition.

Definition A.2 (Indistinguishability obfuscator). *Let $\{\mathcal{L}_\lambda\}_{\lambda\in\mathbb{N}}$ be a class of circuits where each element $c \in \mathcal{L}_\lambda$ maps an $\mathsf{inp}(\lambda)$-bit input into an $\mathsf{out}(\lambda)$-bit output. An* indistinguishability obfuscator *for $\{\mathcal{L}_\lambda\}_{\lambda\in\mathbb{N}}$ is a PPT algorithm iO satisfying the following properties:*

- ***Correctness***. *For every $\lambda \in \mathbb{N}$, $x \in \{0,1\}^{\mathsf{inp}(\lambda)}$ and $c \in \mathcal{L}_\lambda$, we have that*

$$\Pr\left[c'(x) = c(x) \middle| c' \xleftarrow{\$} \mathsf{iO}(1\!\!1^\lambda, c)\right] = 1.$$

- ***Security***. *For every circuits $c_0, c_1 \in \mathcal{L}_\lambda$ such that $c_0(x) = c_1(x)$ for every $x \in \{0,1\}^{\mathsf{inp}(\lambda)}$, we have*

$$\mathsf{iO}(1\!\!1^\lambda, c_0) \sim_c \mathsf{iO}(1\!\!1^\lambda, c_1).$$

We point out that every obfuscator is tailored to a specific class of circuits. As the latter grows, the size of the obfuscated programs and of the obfuscator itself often increases.

The first candidate indistinguishability obfuscator was designed by Garg *et al.* in [GGH+13] based on non-standard assumptions. The work opened the way to a vast line of research focused on weakening the assumptions needed by iO [JLS21]. All the constructions presented so far, however, either rely on subexponentially secure primitives or an exponential number of polynomially secure ones.

A.3 Puncturable PRFs

This paper also makes use of puncturable PRFs [KPTZ13, BW13, BGI14]. As for standard PRFs, the latter is a primitive that using a random secret key, maps a nonce into a pseudorandom string. These constructions however satisfy an additional property: it is possible to remove from any key K all the information about the expansion of a chosen nonce x. The operation is called puncturing and basically, provides a modified key $\overline{K}$. The latter can be expanded as the original one for every nonce except x, obtaining exactly the same outputs. However, the expansion of x using K looks random even for an adversary holding $\overline{K}$. We recall the formal definition below.

Definition A.3 (Puncturable PRF). *A puncturable PRF with output space* $(\mathcal{Z}_\lambda)_{\lambda\in\mathbb{N}}$ *and nonce space* $(\mathcal{Y}_\lambda)_{\lambda\in\mathbb{N}}$ *is a pair of PPT algorithms* (F, Punct) *satisfying the following properties*

- ***correctness****. For every* $\lambda \in \mathbb{N}$*, and* $K \in \{0,1\}^\lambda$ *and* $x, y \in \mathcal{Y}_\lambda$ *such that* $x \neq y$*, we have that*

$$\Pr\left[F(K,y) = F(\overline{K},y)\middle|\overline{K} \leftarrow \mathsf{Punct}(K,x)\right] = 1.$$

- ***Security****. For every* $x \in \mathcal{Y}_\lambda$*, no PPT adversary can distinguish between*

$$\left\{\overline{K}, z \middle| \begin{array}{l} K \xleftarrow{\$} \{0,1\}^\lambda \\ \overline{K} \leftarrow \mathsf{Punct}(K,x) \\ z \leftarrow F(K,x) \end{array}\right\} \qquad \left\{\overline{K}, z \middle| \begin{array}{l} K \xleftarrow{\$} \{0,1\}^\lambda \\ \overline{K} \leftarrow \mathsf{Punct}(K,x) \\ z \xleftarrow{\$} \mathcal{Z}_\lambda \end{array}\right\}$$

It is easy to build puncturable PRFs using the GGM construction [BW13, GGM86]. Puncturable PRFs are part of the standard toolkit for iO based cryptography [SW14].

A.4 Somewhere Statistically Binding Hash Functions

We recall the definition of *somewhere statistically binding hash functions* (SSB hashing) [HW15, OPWW15]. Informally speaking, an SSB hash function is a hash function with block alphabet Σ hashing messages of length at most $L(\lambda)$. The particular property of the construction is that it hides an index i in its hash key hk. As for any hash function, despite being hard to find, there always exist messages $\boldsymbol{x} \neq \boldsymbol{x}'$ having colliding hashes $\mathsf{Hash}(\mathsf{hk}, \boldsymbol{x}) = \mathsf{Hash}(\mathsf{hk}, \boldsymbol{x}')$. SSB hash functions satisfy however an additional binding property: if the hash of $\boldsymbol{x}$ collides with the hash of $\boldsymbol{x}'$, the i-th block of $\boldsymbol{x}$ is guaranteed to coincide with the i-th block in $\boldsymbol{x}'$.

The construction allows also the generation of (usually short) proofs proving that a certain value $x \in \Sigma$ coincides with the j-th block of a preimage of a digest h. The index j does not need to be the one hidden in the hash key. The proofs are not necessarily zero-knowledge. We recall the formal definition.

Definition A.4 (SSB hash function). *An* somewhere statistically binding *(SSB) hash function with block alphabet* Σ *and output length* $\ell_{\mathsf{Hash}}(\lambda)$ *is a tuple of PPT algorithms* $(\mathsf{Gen}, \mathsf{Hash}, \mathsf{Open}, \mathsf{Verify})$ *with the following syntax:*

- Gen *is a PPT algorithm taking as input the security parameter* $1\!\!1^\lambda$*, a bound* $L \leq 2^\lambda$ *and an index* $i \in [L]$*. The output is an SSB hash key* hk*.*
- Hash *is a deterministic algorithm taking as input an SSB hash key* hk *and a message* $\boldsymbol{x} \in \Sigma^{\leq L}$*. the output is a digest* $h \in \{0,1\}^{\ell_{\mathsf{Hash}}(\lambda)}$*.*
- Open *is a PPT algorithm taking as input an SSB hash key* hk*, a message* $\boldsymbol{x} \in \Sigma^{\leq L}$ *and an index* $i \in [L]$*. The output is an SSB proof* π*.*

– Verify *is a deterministic algorithm taking as input an SSB hash key* hk*, a digest* h*, an index* $i \in [L]$*, a value* $x \in \Sigma$ *and an SSB proof* π*. The output is a bit* $b \in \{0,1\}$*.*

We also require the following properties:

– ***Correctness****. For any* $\lambda \in \mathbb{N}$*, bound* $L \leq 2^\lambda$*, indexes* $i, j \in [L]$ *and message* $x \in \Sigma^{\leq L}$ *k-length at most* L*, we have*

$$\Pr\left[\mathsf{Verify}(\mathsf{hk}, h, j, x_j, \pi) = 1 \;\middle|\; \begin{array}{l} \mathsf{hk} \xleftarrow{\$} \mathsf{Gen}(1\!\!1^\lambda, L, i) \\ h \leftarrow \mathsf{Hash}(\mathsf{hk}, \boldsymbol{x}) \\ \pi \xleftarrow{\$} \mathsf{Open}(\mathsf{hk}, \boldsymbol{x}, j) \end{array}\right] = 1$$

– ***Index hiding****. For every bound* $L(\lambda) \leq 2^\lambda$ *and indexes* $i, j \in [L(\lambda)]$*,*

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

– ***Somewhere statistically binding****. For every bound* $L(\lambda) \leq 2^\lambda$ *and index* $i \in [L(\lambda)]$*, we have*

$$\Pr\left[\begin{array}{c} \exists (x, x', h, \pi, \pi') \text{ s.t.} \\ x \neq x' \\ \mathsf{Verify}(\mathsf{hk}, h, i, x, \pi) = 1 \\ \mathsf{Verify}(\mathsf{hk}, h, i, x', \pi') = 1 \end{array} \;\middle|\; \mathsf{hk} \xleftarrow{\$} \mathsf{Gen}(1\!\!1^\lambda, L(\lambda), i)\right] = \mathsf{negl}(\lambda).$$

We point out that thanks to both correctness, hiding and binding, SSB hash functions are also collision resistant. Furthermore, we also highlight that in order for binding to hold, the digest length ℓ_{Hash} must be at least $\log|\Sigma|$.

SSB hash functions can be built from various cryptographic assumptions including FHE [HW15], DDH and DCR [OPWW15].

A.5 Garbled Circuits

This work uses garbled circuits [Yao86, BHR12]. A garbling scheme is a cryptographic primitive allowing us to encrypt a circuit and its inputs. The result of such operation is called the *garbling*. A party provided with the garbled circuit is able to retrieve its output without learning any additional information except the structure of the circuit.

We point out that garbled circuits and obfuscation are very different concepts. Garbled circuits can be usually evaluated on at most one input and the structure of the circuit is always leaked. Obfuscated programs instead have no bound on the number of times they can be evaluated and they never leak any information about the original circuit. The difference between the two notions is also mirrored by the assumptions needed for their construction: while all the candidate obfuscators are based on subexponentially secure primitives, garbled circuits can be built from one-way functions.

We recall the formal definition of garbling scheme.

Definition A.5 (Garbling scheme). *A* garbling scheme *is a tuple of PPT algorithms* $(\mathsf{Garble}, \mathsf{Eval}, \mathsf{En}, \mathsf{De})$ *with the following syntax:*

- Garble *is a PPT algorithm taking as input the security parameter* $\mathbb{1}^\lambda$ *and a circuit* c*. The output is the garbled circuit* G*, the encoding information* e *and the decoding information* d*.*
- En *is a deterministic algorithm taking as input a value* x *and the encoding information* e*. The output is the input information* X*.*
- Eval *is a deterministic algorithm taking as input a garbled circuit* G *and the input information* X*. The output is the output information* Y*.*
- De *is a deterministic algorithm taking as input the output information* Y *and the decoding information* d*. The output is a value* y*.*

We require the scheme to satisfy the following properties:

- ***Correctness****. For any* $\lambda \in \mathbb{N}$*, circuit* c *and input* x*,*

$$\Pr\left[\mathsf{De}(Y,d) = c(x) \;\middle|\; \begin{array}{l}(G,e,d) \xleftarrow{\$} \mathsf{Garble}(\mathbb{1}^\lambda, c)\\ X \leftarrow \mathsf{En}(x,e)\\ Y \leftarrow \mathsf{Eval}(G,X)\end{array}\right] = 1 - \mathsf{negl}(\lambda).$$

- ***Security****. There exists a PPT simulator* Sim *such that, for every circuit* c *and input* x*, the following distributions are computationally indistinguishable:*

$$\left\{G, X, d \;\middle|\; \begin{array}{l}(G,e,d) \xleftarrow{\$} \mathsf{Garble}(\mathbb{1}^\lambda, c)\\ X \leftarrow \mathsf{En}(x,e)\end{array}\right\} \quad \textit{and} \quad \left\{\mathsf{Sim}\big(\mathbb{1}^\lambda, \mathsf{struct}(c), c(x)\big)\right\}.$$

A famous garbling scheme. The constructions in this paper do not make a black-box use of the primitive. We therefore sketch how it is possible to garble and evaluate a circuit using a 2-keyed PRF only. The scheme below is credited to [Yao86].

The labels of the wires. Suppose that c is a binary circuit made of XOR and AND gates. We associate every wire w of c with two random strings $k_w^0, k_w^1 \in \{0,1\}^\lambda$, called the *labels* of w. The two labels are associated with the value that w can assume, namely 0 and 1 and they correspond to keys of the PRF. At the right time, the party performing the evaluation will learn only one of the two labels for every wire, specifically, the one associated with the value of w in that particular execution of the circuit. The evaluator will not know however if the known label is k_w^0 or k_w^1.

Garbling the gates. Each gate g in c is garbled by "encrypting" the labels of the output wire under the labels of the corresponding inputs. The operations is performed using the 2-keyed PRF. Specifically, let u, v be the input wires and let w be the output wire. For every pair $(b_1, b_2) \in \{0,1\} \times \{0,1\}$, we encrypt $k_w^{g(b_1,b_2)}$ using $k_u^{b_1}$ and $k_v^{b_2}$. The evaluator is provided with all the four ciphertexts

generated in this way. We permute their order to avoid leaking the value of the inputs to which they are associated. Notice that the evaluator can retrieve a label of the output wire if and only if it knows the labels of the corresponding inputs. In other words, it can decrypt only one of the four ciphertexts, in the other cases, it will obtain random looking strings. If we properly pad the plaintexts before encryption, the evaluator can learn when the decryption succeeds and when it fails.

The encoding and decoding information. In order to allow the evaluator compute $c(x)$ without learning any additional information about x, we provide it with the labels of the input wires associated with x, i.e. for every i, we reveal $k_{w_i}^{x_i}$ where w_i is the i-th input wire and x_i is the i-th bit in x. This is sufficient to trigger the chain of decryptions that leads to labels of the output wires. In order to decode the latter, we provide the evaluator with both labels k_w^0 and k_w^1 for every output wire w.

The locality of garbling. An important property of the garbling scheme we just described is its *locality*: as long as we know the labels associated with the input and output wires of a gate, we can garble it without knowing any information about the rest of the circuit. This will be fundamental in the constructions presented in this paper.

A.6 Simulation-Extractable NIZKs

The last cryptographic primitive we need in in this paper is multi-theorem simulation-extractable NIZKs [GO07]. A NIZK (or non-interactive zero-knowledge proof) is a construction proving that a public input x, called the statement, belongs to an NP language L. The construction relies on a CRS. Given the latter and a witness w for x, it is possible to efficiently generate a proof π. The proof can easily be verifier on the CRS and the statement without further interaction.

The construction satisfies multi-theorem zero-knowledge, meaning that, even if we prove multiple statements using the same CRS, the adversary cannot distinguish between the real proofs and fake ones simulated without using the witnesses by relying on a trapdoor embedded in the CRS. The construction satisfies also simulation extractability, meaning that the trapdoor in the CRS allows to efficiently extract the witness from the valid proofs generated by any PPT adversary. The condition holds even if the adversary is provided with simulated proofs for multiple statements chosen ahead of time. In other words, a simulation-extractable NIZK is a proof of knowledge.

Below, we recall the formal definition of multi-theorem simulation-extractable NIZK.

Definition A.6 (Multi-theorem simulation-extractable NIZK). *A multi-theorem simulation-extractable NIZK (non-interactive zero-knowledge proof) for an NP relation $\mathcal{R}$ is a triple of PPT algorithms* $(\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$ *with the following syntax:*

– Gen *is a PPT algorithm taking as input the security parameter* $1\!\!1^\lambda$ *and outputs a CRS* σ.
– Prove *is a randomised algorithm taking as input the security parameter* $1\!\!1^\lambda$, *a CRS* σ, *a statement* x *and a witness* w. *The output is a proof* π *for* x.
– Verify *is a deterministic algorithm taking as input a CRS* σ, *a proof* π *and a statement* x. *The output is a bit* $b \in \{0,1\}$.

We require the following properties:

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

$$\Pr\left[\mathsf{Verify}(\sigma,\pi,x)=1 \,\middle|\, \begin{array}{l}\sigma \xleftarrow{\$} \mathsf{Gen}(1\!\!1^\lambda)\\ \pi \xleftarrow{\$} \mathsf{Prove}(1\!\!1^\lambda,\sigma,x,w)\end{array}\right] = 1-\mathsf{negl}(\lambda).$$

– ***Multi-Theorem Zero-Knowledge.*** *There exists PPT simulators* Sim_1 *and* Sim_2 *such that, for every polynomial* $L(\lambda)$ *and tuple of pairs* $(x_i,w_i)_{i\in[L]} \in \mathcal{R}$, *no PPT adversary can distinguish between the following distributions*

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

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

– ***Simulation Extractability*** *There exists a PPT extractor* $\mathsf{Extract}$ *such that, for every polynomial* $L(\lambda)$, *statements* $(x_i,w_i)_{i\in[L]} \in \mathcal{R}$ *and PPT adversary* $\mathcal{A}$, *we have*

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

We point out that it is possible to build multi-theorem simulation-extractable NIZK where the CRS is unstructured [FLS90, GOS06, PS19, BP15], i.e. it can be derived in a secure way from a random string of bits. Unstructured CRSs can always be generated without interaction in the random oracle model.

A.7 Notions of Entropy

In information theory, entropy is used to measure the unpredictability of random variables. After almost a century of research, several definitions have been formalised. In this appendix, we recall some of the important notions and the related properties. We start with Shannon's entropy [Sha48].

Definition A.7 (Shannon's entropy). *Let X be a random variable having finite support. The Shannon's entropy of X is*

$$\mathsf{H}(X) := -\sum_x \Pr[X = x] \cdot \log\big(\Pr[X = x]\big).$$

We recall also the notion of conditional Shannon's entropy.

Definition A.8 (Conditional Shannon's entropy). *Let X and Y be random variables having finite support and let E be an event. The Shannon's entropy of X conditioned on E is*

$$\mathsf{H}(X|E) := -\sum_x \Pr[X = x|E] \cdot \log\big(\Pr[X = x|E]\big).$$

The Shannon's entropy of X conditioned on Y is instead

$$\mathsf{H}(X|Y) := \sum_y \Pr[Y = y] \cdot \mathsf{H}(X|Y = y).$$

Shannon's entropy satisfies an important property called *the strong chain rule*. We recall it below.

Theorem A.9 (Strong chain rule). *Let X and Y be random variables with finite support. Then,*

$$\mathsf{H}(X, Y) = \mathsf{H}(Y) + \mathsf{H}(X|Y).$$

Notice that (X, Y) is a random variable, so $\mathsf{H}(X, Y)$ is defined as in Def. A.7. We also recall the following properties of Shannon's entropy.

Lemma A.10. *Let X, Y and Z be random variables with finite support. Then,*

- *If X is uniform over a set of cardinality m, $\mathsf{H}(X) = \log m$.*
- *If X is independent of Y, given Z, $\mathsf{H}(X|Y, Z) = \mathsf{H}(X|Z)$.*
- $\mathsf{H}(X|Y, Z) \leq \mathsf{H}(X|Z)$.
- *If f is a deterministic function, $\mathsf{H}(f(X)) \leq \mathsf{H}(X)$.*

We now recall other definitions of entropy that are used to prove our results.

Definition A.11 (Max entropy). *Let X be a random variable with finite support, let E be an event. We define the max entropy of X to be*

$$\mathsf{H}_0(X) = \log|\mathsf{Supp}(X)|.$$

We define the max entropy of X conditioned on E to be

$$\mathsf{H}_0(X|E) = \log|\mathsf{Supp}(X|E)|.$$

Definition A.12 (Min entropy). *Let X be a random variable with finite support, let E be an event. We define the min entropy of X to be*

$$\mathsf{H}_\infty(X) = -\log\big(\max_x \Pr[X = x]\big).$$

We define the min entropy of X conditioned on E to be

$$\mathsf{H}_\infty(X|E) = -\log\big(\max_x \Pr[X = x|E]\big).$$

Finally, we recall the definition of collision entropy.

Definition A.13 (Collision entropy). *Let X and Y be random variables with finite support, let E be an event. We define the collision entropy of X to be*

$$\mathsf{H}_2(X) = -\log\Big(\sum_x \Pr[X = x]^2\Big) = -\log\big(\Pr[X = X']\big),$$

where X' is independent and identically distributed to X. We define the collision entropy of X conditioned on E to be

$$\mathsf{H}_2(X|E) = -\log\Big(\sum_x \Pr[X = x|E]^2\Big).$$

The average collision entropy of X given Y is instead

$$\widetilde{\mathsf{H}}_2(X|Y) = -\log\Big(\sum_{x,y} \Pr[Y = y] \cdot \Pr[X = x|Y = y]^2\Big).$$

All the above definitions of entropy are not equivalent. For instance, Shannon's entropy can assume values that are significantly larger than min and collision entropy. The definitions are however related by the following well-known inequalities.

Theorem A.14. *Let X be a random variable with finite support, let E be an event. We have that*

$$0 \leq \mathsf{H}_\infty(X) \leq \mathsf{H}_2(X) \leq \mathsf{H}(X) \leq \mathsf{H}_0(X),$$

$$0 \leq \mathsf{H}_\infty(X|E) \leq \mathsf{H}_2(X|E) \leq \mathsf{H}(X|E) \leq \mathsf{H}_0(X|E) \leq \mathsf{H}_0(X).$$

Yao's Incompressibility Entropy. All the entropy notions we recalled above are great for measuring information theoretic properties, however, they all suffer from an important disadvantage, namely, they do not behave well under computational indistinguishability. Specifically, if $X \sim_c X'$, the entropy of X can be significantly different from the entropy of X', it does not matter which of the above definitions we consider.

We solve this issue by relying on a notion of computational entropy [Yao82, HLR07]. We recall the definition.

Definition A.15 (Yao's entropy). *Let $(X_\lambda)_{\lambda\in\mathbb{N}}$ be an ensemble of random variables. We say that the Yao entropy of X is smaller or equal to $k(\lambda)$, written $\mathsf{H}_{\mathsf{Yao}}(X) \leq k(\lambda)$, if there exists a pair of polynomial sized deterministic circuits $(c_\lambda, d_\lambda)_{\lambda\in\mathbb{N}}$ such that*

$$\Pr[d(c(X)) = X] \geq \frac{2^{\ell(\lambda)}}{2^{k(\lambda)}} - \mathsf{negl}(\lambda).$$

In the above formula, $\ell(\lambda)$ denotes the output size of c_λ. The circuit c_λ is called a compressor, whereas d_λ is called a decompressor.

In [HLR07], Hsiao, Lu and Reyzin generalised the definition to the conditional case. We recall it below.

Definition A.16 (Conditional Yao's entropy). *Let $(X_\lambda)_{\lambda\in\mathbb{N}}$ and $(Y_\lambda)_{\lambda\in\mathbb{N}}$ be two ensembles of random variables. We say that the Yao entropy of X conditioned on Y is smaller or equal to $k(\lambda)$, written $\mathsf{H}_{\mathsf{Yao}}(X|Y) \leq k(\lambda)$, if there exists a pair of polynomial sized deterministic circuits $(c_\lambda, d_\lambda)_{\lambda\in\mathbb{N}}$ such that*

$$\Pr[d(c(X,Y),Y) = X] \geq \frac{2^{\ell(\lambda)}}{2^{k(\lambda)}} - \mathsf{negl}(\lambda).$$

In the above formula, $\ell(\lambda)$ denotes the output size of c_λ. The circuit c_λ is called a compressor, whereas d_λ is called a decompressor. If $\mathsf{H}_{\mathsf{Yao}}(X|Y) \leq k(\lambda)$ where $k(\lambda)$ is $O(\log\lambda)$, we will simply write that $\mathsf{H}_{\mathsf{Yao}}(X|Y) = O(\log\lambda)$.

Essentially, Yao's incompressibility entropy measures how much it is possible to compress, in polynomial time, samples from a distribution X given that the outcome of the possibly correlated random variable Y is known.

We observe that Yao's entropy can assume values that are significantly larger than Shannon's entropy. Examples of this kind are the outputs of PRGs. In some particular cases, however, also the opposite is true. For instance, there exist distributions X such that $\mathsf{H}_\infty(X) = O(\log\lambda)$ but $\mathsf{H}(X) = \omega(\log\lambda)$. For all such X, we have $\mathsf{H}_{\mathsf{Yao}}(X) = O(\log\lambda)$ (consider the compressor that outputs the empty string and the decompressor that outputs the most likely element).

The following well-known lemma formalises the fact that Yao's entropy preserves under computational indistinguishability.

Lemma A.17. *Let $(X_\lambda, Y_\lambda)_{\lambda\in\mathbb{N}}$ and $(X'_\lambda, Y'_\lambda)_{\lambda\in\mathbb{N}}$ be two ensembles of random variables such that $(X_\lambda, Y_\lambda) \sim_c (X'_\lambda, Y'_\lambda)$. Then, $\mathsf{H}_{\mathsf{Yao}}(X|Y) \leq k(\lambda)$ if and only if $\mathsf{H}_{\mathsf{Yao}}(X'|Y') \leq k(\lambda)$.*

We highlight that Yao's entropy is not the only notion of computational entropy [Rey11, HILL99, HLR07]. Among all the studied notions, it is however the one assuming highest values [HLR07]. We decided to use Yao's entropy exactly for this reason, making the results presented in this paper as strong as possible.