Improved YOSO Randomness Generation with Worst-Case Corruptions

Chen-Da Liu-Zhang[1][0000-0002-0349-3838], Elisaweta Masserova[2][0009-0002-8970-9624], João Ribeiro[3⋆][0000-0002-9870-0501], Pratik Soni[4][0000-0002-3225-3323], and Sri AravindaKrishnan Thyagarajan[5][0000-0003-0114-7672]

[1] Lucerne University of Applied Sciences and Arts & Web3 Foundation, Zug, Switzerland
`chen-da.liuzhang@hslu.ch`
[2] Carnegie Mellon University, Pittsburgh, PA, USA
`elisawem@andrew.cmu.edu`
[3] IST - University of Lisbon, Lisboa, Portugal
`jribeiro@tecnico.ulisboa.pt`
[4] University of Utah, Salt Lake City, UT, USA
`psoni@cs.utah.edu`
[5] NTT Research, Sunnyvale, CA, USA
`t.srikrishnan@gmail.com`

Abstract. We study the problem of generating public unbiased randomness in a distributed manner within the recent You Only Speak Once (YOSO) framework for stateless multiparty computation, introduced by Gentry et al. in CRYPTO 2021. Such protocols are resilient to adaptive denial-of-service attacks and are, by their stateless nature, especially attractive in permissionless environments. While most works in the YOSO setting focus on independent random corruptions, we consider YOSO protocols with worst-case corruptions, a model introduced by Nielsen et al. in CRYPTO 2022.
Prior work on YOSO public randomness generation with worst-case corruptions designed information-theoretic protocols for t corruptions with either $n = 6t + 1$ or $n = 5t$ roles, depending on the adversarial network model. However, a major drawback of these protocols is that their communication and computational complexities scale exponentially with t. In this work, we complement prior inefficient results by presenting and analyzing simple and efficient protocols for YOSO public randomness generation secure against worst-case corruptions in the computational setting. Our first protocol is based on publicly verifiable secret sharing and uses $n = 3t+2$ roles. Since this first protocol requires setup and somewhat heavy cryptographic machinery, we also provide a second lighter protocol based on ElGamal commitments and verifiable secret sharing which uses $n = 5t + 4$ or $n = 4t + 4$ roles depending on the underlying network model. We demonstrate the practicality of our second protocol by showing experimental evaluations, significantly improving over

⋆ Work done while with NOVA LINCS and NOVA School of Science and Technology, Caparica, Portugal.

prior proposed solutions for worst-case corruptions, especially in terms of transmitted data size.

1 Introduction

Public randomness is a fundamental component of numerous financial and security protocols [18, 14]. Randomness usage is ubiquitous: From establishing fairness in the green card lottery, to assessing risk via Monte Carlo simulations, to generating the public parameters for the cryptographic protocols [1, 15]. In the past, public randomness was typically obtained via trusted third parties. However, with the emergence of blockchains and web3, there has been an increased effort to decentralize economic activities, and as a consequence, to decentralize public randomness generation as well [5, 21, 6, 13, 4].

A protocol for such distributed public randomness allows multiple mutually distrusting parties, each with their own source of randomness, to generate and agree on a public random value. However, designing a secure protocol which provides such a functionality is a notoriously hard task. Indeed, the cryptographic community put significant effort into designing distributed randomness generation protocols [5, 6, 21, 13, 4], as well as improving functionalities such as verifiable delay functions [3] and time-lock puzzles [22], which oftentimes serve as building blocks in such protocols.

Traditionally, those protocols consider *static* adversaries, where security is guaranteed as long as the adversary decides which parties to corrupt prior to the start of the execution. However, such an assumption seems unjustified, especially for protocols that run over long periods of time. A far more realistic setting would allow the adversary to corrupt parties *dynamically* during the course of the execution. Unfortunately, protocols in this *adaptive* setting tend to be costly. In the context of general multi-party computation, a novel approach to achieving adaptive security (in an arguably more efficient manner) has been recently proposed in the *You-Only-Speak-Once* (YOSO) line of work, introduced by Gentry, Halevi, Krawczyk, Magri, Nielsen, Rabin, and Yakoubov [10] and the Fluid MPC model introduced by Choudhuri, Goel, Green, Jain, and Kaptchuk [7]. Intuitively, protocols in the YOSO setting consider the notion of stateless ephemeral *roles*, where at a single point in time a small committee of such roles is required to perform certain actions, and produce a public output, along with messages to be sent to future roles. Roles are assigned to physical machines via a "role-assignment" functionality in the beginning of each round, in a way that makes it hard for the adversary to predict which physical machines will be participating as roles in a given committee. As roles are allowed to send only a single message (i.e., speak only once), and are torn down after the execution, adaptively corrupting a machine which executed a certain role in the past does not help the adversary. Due to these observations, assuming that the adversary can only corrupt a fraction of (a large total number of) physical machines, the protocols designed in the YOSO setting typically rely on the fact that the adversary's best option is to corrupt machines *at random*.

However, this assumption is viable only if role-assignment (which is typically separated from the multi-party protocol computing the function of interest) is truly secure. This makes role-assignment protocols hard to design, and the currently known constructions compromise either in terms of efficiency [11] or in terms of the supported corruption threshold [2].

The line of work designing Fluid MPC protocols [7–9] considers a worst-case corruption-per-committee model, where up to a certain minority fraction of parties are corrupted in each committee. In order to reduce trust in role-assignment even more, Nielsen, Ribeiro, and Obremski (NRO in the following) recently introduced a model for YOSO with *worst-case corruptions* [16], which we dub $\mathrm{YOSO}^{\mathsf{WCC}}$. In this model, prior to the start of the protocol, the adversary can choose any up to t roles to corrupt overall *across* all participating parties. The $\mathrm{YOSO}^{\mathsf{WCC}}$ model is tailored to the randomness generation setting, and the authors introduce two information-theoretic protocols which are secure given worst-case corruption of roles. Unfortunately, these protocols incur exponential communication- and computation complexities, which motivates us to ask the following question:

Can we design efficient distributed randomness generation protocols in the model of YOSO with worst-case corruptions?

As it is trivially possible to adapt known stateful randomness generation protocols to the $\mathrm{YOSO}^{\mathsf{WCC}}$ setting at the cost of having a very low adversarial threshold (see Section 1.3 for details), we further refine the question as follows:

Can we design efficient distributed randomness generation protocols in the model of YOSO with worst-case corruptions while optimizing the required number of roles?

1.1 Our Contributions

In this work, we answer the question above positively. As in NRO, we distinguish between two different adversarial models, the *sending-leaks* and *execution-leaks* models. Intuitively, in the execution-leaks model the adversary only obtains messages addressed to corrupted parties upon their execution. In the stronger sending-leaks model, the adversary obtains the messages addressed to corrupted parties immediately upon the sender sending the message. We design two randomness generation protocols in the sending-leaks model, along with an optimized version for the execution-leaks model, and prove these protocols secure. Our protocols are in the computational setting, meaning that the adversary we consider is computationally bounded.

In our first construction, we build upon a non-interactive *publicly verifiable secret sharing* (PVSS) protocol [20], which allows a dealer to share a secret in a single round among a set of parties (a subset of which can be corrupt) in a way that lets anyone verify that the dealer behaved correctly. Our PVSS-based randomness generation protocol requires $3t + 2$ roles, and has communication complexity that grows quadratically in the number of parties. While this construction requires setup and somewhat heavy cryptographic machinery in the form of simulation-extractable non-interactive zero-knowledge proofs, in our second construction we do not require setup and rely only on the usage of ElGamal commitments. In this construction, we build upon *verifiable secret sharing* (VSS) protocols [17], a notion that is similar to PVSS but requires more rounds. The communication complexity of this protocol is quadratic in the number of roles. The protocol requires $5t + 4$ roles in the sending-leaks model or $4t + 4$ roles in the execution-leaks model.

We implement our VSS-based construction and compare it to our implementation of the NRO scheme. Our evaluation shows that our protocol is not only asymptotically but also concretely efficient, and we outperform NRO for values as small as $t = 6$ (for running time) and $t = 3$ (for size of the transmitted data).

In the following, we first briefly outline our model, and then provide an overview of the main techniques and ideas used in our work.

1.2 Our Model and Security Goal

We now briefly outline the $\text{YOSO}^{\mathsf{WCC}}$ model we work in, following the communication model description of NRO [16]. We distinguish between stateless "roles" and physical machines which may run for a long time and retain state. Note that in the following we use the terms "role" and "party" interchangeably. We consider n parties $P_1, \dots, P_n$, which are executed one after the other. We assume that each party has its own internal source of randomness. We consider a computationally bounded adversary which is allowed to corrupt any t out of n parties before the protocol starts. Upon its execution, P_i can publicly broadcast a value x_i and send secret values $s_{i,j}$ to each "future" party P_j, i.e., any P_j such that $j > i$. We consider the following two adversarial network settings:

– In the *sending-leaks* model an adversary obtains a message $s_{i,j}$ sent by an honest P_i to a corrupt P_j as soon as P_i sent it. We call the corresponding adversary the *sending-leaks* adversary.
– In the *execution-leaks* model an adversary obtains a message $s_{i,j}$ sent by an honest P_i to a corrupt P_j only once P_j is activated. We call the corresponding adversary the *execution-leaks* adversary.

Our goal is the following: After the execution of all parties is complete, anyone (not just physical machines which acted as roles $P_1, \dots, P_n$) can obtain unbiased public randomness by applying a publicly known and deterministic extraction function to the values $(x_1, \dots, x_n)$. See Figure 1 for a visual representation of this process.

More formally, let λ denote a security parameter. Consider an interaction of an adversary A with the honest parties in the randomness generation protocol and let $\mathsf{OUT}(A)$ denote the coin output of this protocol with adversary A. Let $L(\lambda)$ denote the length of this output. Let D be a distinguisher. Consider the following experiment (for protocols which assume trusted setup, this setup is generated by the challenger):

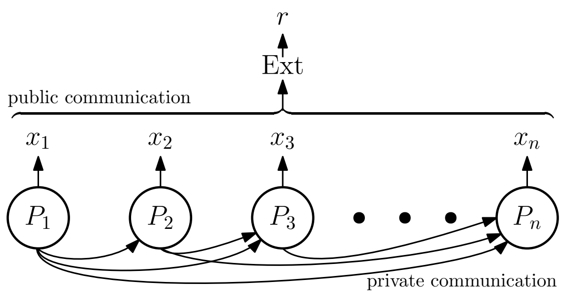

Fig. 1: Communication model from [16, Figure 1]. Parties P_i speak one after the other, send secrets to future parties P_j for $j > i$, and publish public values, which are available to all parties.

1. $b \xleftarrow{\$} \{0,1\}$.
2. $r \xleftarrow{\$} \{0,1\}^{L(\lambda)}$.
3. If $b = 0$, set $\mathsf{coin} \leftarrow \mathsf{OUT}(A)$. Otherwise, set $\mathsf{coin} \leftarrow r$.
4. $b' \leftarrow D(\mathsf{coin})$.

Then, we have the following formal security definition.

Definition 1 (Computationally secure YOSO$^{\mathsf{WCC}}$ randomness generation). *A YOSO$^{\mathsf{WCC}}$ randomness generation protocol with n parties is* (t,n)-computationally secure in the sending-leaks (resp. execution-leaks) model *if for all PPT sending-leaks (resp. execution-leaks) adversaries A that corrupt t out of n parties and all PPT distinguishers D in the above security game it holds that*

$$\left|\Pr[b = b'] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda).$$

1.3 Our Techniques

First, note that, as pointed out by NRO, any stateful r-round multiparty computation protocol which is secure against t out of n corruptions can be ported to the YOSO$^{\mathsf{WCC}}$ setting as follows: Use r roles $P_{i,r}$ to implement the behavior of each participant P_i of the stateful protocol over r rounds. Role $P_{i,k}$ mimics the behavior of P_i in round k of the stateful protocol, with the caveat that it additionally sends its state to the future role $P_{i,k+1}$. Unfortunately, this approach is costly in terms of the required number of roles: It requires $n \cdot r$ roles, while tolerating only t corrupted parties.

To address this issue, we design randomness generation protocols which are tailored to the YOSO$^{\mathsf{WCC}}$ setting. For simplicity, say we wish to generate only a single random bit $r \in \{0,1\}$.

First idea. Our first idea is the following: As each party has its own source of randomness, we could set $n = t+1$ and simply XOR all values r_i, where r_i is the random bit generated by P_i, i.e., set $r = \bigoplus_{i \in [t+1]} r_i$. As at least one party out of $t+1$ is honest, the XOR should result in an unbiased bit. However, we need to be careful – we must not let a corrupt party see the values of the honest parties before supplying its own r_i. Thus, intuitively, we have to make each party *commit* to the randomness it wishes to contribute prior to revealing the values of other parties. This approach requires a party to speak two times: Once when committing to a value, and once when opening it. This can be naively achieved by using two roles to implement P_i, and having the first role privately send its state to its counterpart.

Perhaps surprisingly, this approach still does not achieve what we want: As corrupt parties can refuse to open the committed values, in our protocol we must specify how to proceed in such a case. We can either choose to ignore each such party P_i, thereby making their contribution equal to $r_i = 0$ (first case in the following), or set $r_i = 1$ (second case). In both cases, a corrupt P_{t+1} can bias the outcome of the final XOR by committing to $r_{t+1} = 1$ in the first case and $r_{t+1} = 0$ in the second case, and then adaptively deciding whether to open the value or not during the execution of its second role, thereby setting the result r to the value of its choice. As the second role of P_{t+1} is the last party speaking, all values supplied by the honest parties are known upon its execution.

Utilizing PVSS. We address the issue above by ensuring that the coin output is fixed *prior* to the reveal phase. We begin by considering a setting with trusted setup. In this case, we can rely on a (t, n)-*publicly verifiable secret sharing* (PVSS) protocol. Using such a protocol, a dealer can secret share its secret among n parties in a way that any $t+1$ parties can reconstruct the secret, but any t (potentially corrupted) parties have no information about the secret. Moreover, public verifiability ensures that anyone (even non-recipients) can verify that the dealer sharing has been performed correctly, i.e., there exists a *unique* secret which can be later reconstructed by any set of $t+1$ recipient parties.

Intuitively, this fixes the secret at the end of the commit/sharing phase, and if the adversary corrupts at most t parties, it does not learn any information about the secret. If we ensure that the secret reconstruction starts only after the sharing phase of *all* secrets is complete, the adversary can no longer bias the outcome. However, there is one caveat: As anyone must be able to verify that the sharing was done correctly, the dealer cannot send the shares to the parties via *private* communication. Instead, the dealer publishes encryptions of the shares of the parties with respect to their corresponding public keys. In a scenario such as ours, where we run not only one, but *multiple* PVSS protocols, publicly revealing encryptions of the shares makes PVSS susceptible to malleability attacks. To prevent such attacks from adversarial dealers, we make use of a PVSS protocol with appropriate non-malleability properties. Such properties can be achieved, for example, via simulation-extractable non-interactive zero-knowledge proofs [12].

If we use a $(t, 2t+1)$-PVSS protocol, the above protocol requires only $3t+2$ roles in total: $t+1$ dealers and $2t+1$ parties who obtain the secret shares. This protocol allows us to achieve the following result:

Theorem 1 (informal). *Assuming public key encryption and simulation-extractable NIZKs, there exists a computationally secure randomness generation protocol with $3t+2$ roles in the sending-leaks model, where t is the number of corruptions.*

We give a formal description of our PVSS-based construction in Section 2.

Removing Trusted Setup. While the protocol above enjoys good efficiency properties and requires only a small number of parties, it relies on somewhat heavy cryptographic assumptions and a trusted setup. In our second and main construction we address these limitations.

Our idea is to utilize *verifiable secret sharing* (VSS), which is similar to PVSS, except that it does not provide public verifiability. Instead, we only have the so-called "strong commitment" property, which states that the shares of the honest parties define a secret (which could be $\perp$).

At a high level, as a first step we will design a YOSO$^{\mathsf{WCC}}$-friendly VSS scheme. Then, as in the PVSS-based construction, we will let $t+1$ dealers each share their secret randomness using this VSS. However, as mentioned above, this time we cannot rely on the public verifiability property of the secret sharing scheme. We used this property in the previous construction to determine whether a dealer behaved honestly during the commit/secret sharing phase. This, in turn, allowed us to circumvent the issue where a malicious party commits to some randomness, and *after seeing honest values* decides whether to open this randomness or not. In VSS, the dealer is allowed to send shares to the parties *privately*, and thus when a dealer and a share recipient are in dispute, from the perspective of an external party it may not be immediately possible to tell whether the dealer or the share recipients behaved maliciously. Handling this requires further interaction and results in more roles in our YOSO$^{\mathsf{WCC}}$ VSS-based protocol, which we outline in the following.

We build our protocol around the well-known Pedersen VSS [17]. The standard stateful version of this VSS proceeds in the following four rounds, where s is the secret that is being shared, and g and h are generators of a group where computing discrete logarithms is hard:

1. The dealer D chooses two degree-t polynomials

$$f_1(x) = a_0 + a_1x + \cdots + a_tx^t,$$
$$f_2(x) = b_0 + b_1x + \cdots + b_tx^t$$

such that $b_0 = s$. Then, D broadcasts commitments

$$(c_0, c_1, \cdots, c_t) = (g^{a_0}h^{b_0}, g^{a_1}h^{b_1} \cdots, g^{a_t}h^{b_t}),$$

and sends $r_i = f_1(i)$ and $s_i = f_2(i)$ to each P_i, $i \in [n]$.

2. Each party R_i checks whether $g^{r_i} h^{s_i} = \prod_{k=0}^{t} c_k^{i^k}$. If not, R_i broadcasts $\mathsf{Complain}$.
3. D broadcasts all shares from parties who complained. If any share that D broadcasts does not satisfy the above relation, D is deemed corrupt and the execution halts. Otherwise, each P_i who complained replaces its old share with the new (r_i, s_i).
4. Each R_i outputs r_i, s_i. The value $s = f_2(0)$ is the reconstructed secret.

Note that in the construction above, the dealer as well as each share recipient R_i may need to speak twice – the dealer is required to come back in the third round to resolve the complaints, and each R_i might complain in the second round, and is then required to output its share in the fourth round. We adapt this scheme to the YOSO$^{\mathsf{WCC}}$ setting in two steps: First, we use two roles D and D' for the dealer, and let D not only execute the first round of the protocol above, but also send it state privately to D'. Second, we use two roles R_i and R_i' for each share recipient, and also let R_i not only execute the second round of the protocol above, but send its state to its counterpart R_i'.

A final issue remains: Currently, we assume that g, h are publicly known values, and the construction above is secure assuming that $\log_g h$ is *not known to any party*. We would now like to remove this setup. The strawman idea is to simply have each dealer supply its own pair of g and h. However, if a malicious dealer colludes with a party R_i, then R_i can cheat by providing an invalid opening (which still verifies correctly), thus changing the reconstructed secret value. To fix this, we substitute computationally Pedersen commitments by *unconditionally binding* ElGamal commitments. In more detail, we now compute the commitment $(c_0, c_1, \cdots, c_t)$ as follows:

$$(c_0, c_1, \cdots, c_t) = \big((g^{a_0}, h^{a_0} \cdot g^{b_0}), (g^{a_1}, h^{a_1} \cdot g^{b_1}), \cdots, (g^{a_t}, h^{a_t} \cdot g^{b_t})\big).$$

Pipelining. When implemented naively, the construction outlined above requires $6t + 4$ roles, and the construction is secure in the sending-leaks model. To further decrease the number of roles, we carefully parallelize the execution of both dealer roles with the receiver roles. More concretely, instead of letting the $t + 1$ dealers share secrets towards a fixed set of $2t + 1$ receivers, the recipient set for the i-th dealer is set to be the $2t + 1$ roles that immediately succeed that particular dealer. Moreover, we observe that the conflicts regarding the i-th dealer can also be immediately resolved after the corresponding set of $2t + 1$ receiver roles have been executed.

This means that the total number of roles (after resolving complaints from all dealers) is now $3t + 3$, i.e., this linearization of roles allows us to decrease the total number of roles by roughly t in the sharing phase of the dealers. For further details, see Section 3.

Additional optimization in the execution-leaks model. We make the observation that in the execution-leaks model we can further reduce the final set of receivers R'_i by t roles. The idea is that each original receiver R_i (from the sharing phase) follows the procedure of round two, but in addition sends its shares to *all* roles R'_j (instead of only R'_i as before) if its shares verify correctly. This step does not reveal information on the shares, since the channels to the future roles do not reveal any information until the corresponding recipient role is executed. In the reconstruction step, we can let each R'_j publish the received shares from all parties that it got the shares from. See the full version for details.

We arrive at the final theorem.

Theorem 2 (informal). *Assuming ElGamal commitments, there exists a computationally secure randomness generation protocol with $5t+4$ roles (resp. $4t+4$ roles) in the sending-leaks (resp. execution-leaks model), where t is the number of corruptions.*

2 PVSS-based YOSO$^{\mathsf{WCC}}$ Randomness Generation

We introduce a randomness generation scheme which relies on publicly verifiable secret sharing (PVSS). Before going into our protocol, we briefly explain what a PVSS is.

2.1 Publicly Verifiable Secret Sharing

Recall the definition of Publicly Verifiable Secret Sharing (PVSS) from [4]. In PVSS, a dealer D shares a secret to a set of n parties $\mathcal{P} = \{P_1, \cdots, P_n\}$. A (t, n)-PVSS protocol ensures that a secret is split in a way that allows $t+1$ parties to reconstruct a secret, but at the same time, knowing t shares does not reveal any information about the secret. Any external verifier V is able to check that D acts honestly. More formally, a PVSS protocol consists of the algorithms $(\mathsf{Setup}, \mathsf{Dist}, \mathsf{Verif}, \mathsf{Reconstr\text{-}Dec}, \mathsf{Reconstr\text{-}Pool})$, where $\mathsf{Setup} = (\mathsf{Setup}_\pi, \mathsf{Setup}_{\mathsf{PKI}})$, and which denote the following:

- **Setup:** Consists of $(\mathsf{Setup}_\pi, \mathsf{Setup}_{\mathsf{PKI}})$, which take security parameter λ as input. In Setup_π, the parameters of the proof system are generated in a trusted fashion. Using $\mathsf{Setup}_{\mathsf{PKI}}$, every party generates a public key pk_i and withholds the corresponding secret key sk_i.
- **Distribution:** The dealer creates shares $s_1, \cdots, s_n$ for the secret s, encrypts share s_i with the key pk_i for $i = \{1, \cdots, n\}$ and publishes these encryptions $\hat{s}_i$, together with a proof $\textsc{proof}_D$ that these are indeed encryptions of a valid sharing of some secret.
- **Verification:** In this phase, any external V (not necessarily being a participant in the protocol) can verify non-interactively, given all the public information until this point, that the values $\hat{s}_i$ are encryptions of a valid sharing of some secret.
- **Reconstruction:** This phase is divided in two.
  Decryption of the shares: This phase can be carried out by any set Q of $t+1$ or more parties. Every party P_i in Q decrypts the share s_i from the ciphertext $\hat{s}_i$

by using its secret key sk_i, and publishes s_i together with a (non-interactive) zero-knowledge proof PROOF_i that this value is indeed a correct decryption of $\hat{s}_i$.
Share pooling: Any external verifier V (not necessarily being a participant in the protocol) can now execute this phase. V first checks whether the proofs PROOF_i are correct. If the check passes for less than $t+1$ parties in Q then V aborts; otherwise V applies a reconstruction procedure to the set s_i of shares corresponding to parties P_i that passed the checks.

A PVSS protocol $(\mathsf{Setup}, \mathsf{Dist}, \mathsf{Verif}, \mathsf{Reconstr\text{-}Dec}, \mathsf{Reconstr\text{-}Pool})$ must provide three security guarantees: Correctness, Verifiability and IND1-Secrecy. These properties are defined below:

– **Correctness:** If the dealer and all players in Q are honest, then all checks in the verification and reconstruction phases pass, and the secret can be reconstructed from the information published by the players in Q during reconstruction.
– **Verifiability:** If the check in the Verification phase passes, then with high probability the values $\hat{s}_i$ are encryptions of a valid sharing of some secret. Furthermore, if the check in the Reconstruction phase passes, then the values s_i are indeed the shares of the secret distributed by D.
– **IND1-Secrecy:** Prior to the reconstruction phase, the public information together with the secret keys sk_i of any set of at most t players gives no information about the secret.

2.2 Our PVSS-Based Randomness Generation Protocol

Our protocol is in the sending-leaks model (thus also secure in the execution-leaks model). We describe the scheme and outline the security proof. For the complete proof, see the full version.

The high-level idea of the scheme is the following: given $n = 3t + 2$ parties, split them into two groups $\mathcal{P}$ and $\mathcal{P}'$ of size $t + 1$ and $2t + 1$, respectively. We dub the parties from the first group *dealers*, denoted by $P_1, P_2, \cdots, P_{t+1}$, and the parties from the second group *decryptors*, denoted by $P'_1, P'_2, \cdots, P'_{2t+1}$. Let $(\mathsf{Setup} := (\mathsf{Setup}_\pi, \mathsf{Setup}_{\mathsf{PKI}}), \mathsf{Dist}, \mathsf{Verif}, \mathsf{RDec}, \ \mathsf{RPool})$ denote a $(t, 2t + 1)$-PVSS protocol. The protocol starts with a "sharing" phase, where every P_i is executed one after another and acts as a PVSS dealer distributing its secret to the decryptors in $\mathcal{P}'$. Then, decryptors $P'_i \in \mathcal{P}'$ are executed one after another, and each decryptor P'_i executes the share decryption part of the PVSS reconstruction phase for each dealer P_i. Finally, any party C can execute the share pooling phase of the PVSS reconstruction phase in order to obtain the secret shared by each dealer. We give the full scheme in Protocol 1.

For security, we need our PVSS to be non-malleable, which can be naively achieved by using simulation-extractable NIZKs [12] as PVSS proofs. Intuitively, a strawman PVSS scheme which provides the required non-malleability works as follows: Share the secret using a (t, n) secret sharing scheme (e.g, Shamir's secret

Protocol 1 Randomness Beacon from PVSS in the Sending-Leaks Model.

Setup: PVSS Setup_π algorithm is executed in a trusted fashion to obtain the common reference string crs. Public key of every party in the protocol is generated according to $\mathsf{Setup}_{\mathsf{PKI}}$.

Sharing phase: Each party P_i, $i \in [t+1]$ does the following:

1. P_i samples x_i from $\{0, 1\}$ uniformly at random.
2. P_i uses PVSS algorithm Dist as the dealer to distribute shares of x_i to the parties $P'_1, \cdots, P'_{2t+1}$:
$$(\{\hat{s}_j^{(i)}\}_{j\in[2t+1]}, \textsc{proof}_D^{(i)}) \leftarrow \mathsf{Dist}(x_i, \{pk_{P'_j}\}_{j\in[2t+1]}, \mathsf{crs}).$$
3. P_i publishes $(\{\hat{s}_j^{(i)}\}_{j\in[2t+1]}, \textsc{proof}_D^{(i)})$.

Reconstruction phase: Each party P'_j, $j \in [2t+1]$ does the following:

1. For each P_i, P'_j uses $\mathsf{Verif}(\{\hat{s}_j^{(i)}\}_{j\in[2t+1]}, \textsc{proof}_D^{(i)}, \mathsf{crs})$ to verify that P_i dealt a valid secret. For each P_i who passed the check, P'_j verifies that the proof $\textsc{proof}_D^{(i)}$ and every encryption $\hat{s}_m^{(i)}$ distributed by P_i is not the same as one distributed by any dealer P_k, where $k < i$. Denote P_i as *valid* if so.
2. For each valid P_i, P'_j uses the PVSS algorithm $\mathsf{RDec}(\hat{s}_j^{(i)}, sk'_{P'_j}, \mathsf{crs})$ to obtain $(s_j^{(i)}, \textsc{proof}_j^{(i)})$, and publishes this pair.

Any party C can use the PVSS algorithm RPool on information published by the parties $P'_1, \cdots, P'_{2t+1}$ to obtain x_i. Output $\bigoplus_{i\in I} x_i$, where I denotes an index set of dealers for which C obtained the secret using RPool.

sharing [19]), encrypt each share using a public key of the corresponding share receiver, and append a simulation-extractable NIZK proof confirming that the dealer knows the shares underlying the ciphertexts, and these shares correspond to the (t, n) secret sharing. The reconstruction works by having each receiver decrypt its share, and publish a proof confirming that it knows a secret key such that the decryption of the corresponding ciphertext results in the stated value. See the full version for details. The communication complexity is $O(n^2|c| + n|p|)$, where $|c|$ is the length of a single ciphertext, and $|p|$ of a proof.

Theorem 3. *Assuming public key encryption and simulation-extractable NIZKs, there exists a YOSO*$^{\mathsf{WCC}}$ $(t, 3t+2)$*-computationally secure randomness generation protocol in the sending-leaks model.*

3 Randomness Generation from ElGamal Commitments

We now describe our randomness generation protocol that is secure against computational adversaries, and *does not require any setup assumptions*. We provide two variations of this protocol: One for the sending-leaks model and another for the execution-leaks model. To reduce the number of roles, we use pipelining in both versions. For simplicity, we first describe the protocol without pipelining.

Construction for the sending-leaks model. As mentioned in Section 1.3, the high-level idea of the construction is the following: as a first step, we "linearize" a custom version of Pedersen's VSS protocol [17] where a single party shares a random value in our stateless model. Recall that in each linearization we have the roles:

- Party D, who acts as the dealer distributing the secrets (publishing commitments to the coefficients of t-degree polynomial and bilaterally sending to each receiver a share evaluation), and sends its state to its counterpart D'.
- $2t+1$ receivers R_i, who receive and verify the secret shares, complain about the shares if applicable, and otherwise send these to the counterpart R'_i.
- Party D' who obtains a state from D and uses it to publish the shares of the receivers that complained.
- $2t+1$ receivers R'_i who receive the shares from their counterparts R_i, as well as set their shares to the ones broadcast by D' (if the counterpart R_i complained), and publicly reveal these shares.

We use $t+1$ such linearized VSS to share $t+1$ random values, and output the final coin as the xor the results. In more detail, we let $n = 6t+4$, and divide the n parties into a group $\mathcal{D}$ of size $t+1$, group $\mathcal{R}$ of size $2t+1$, group $\mathcal{D}'$ of size $t+1$, and group $\mathcal{R}'$ of size $2t+1$. These parties execute the following roles:

- Each $D_i \in \mathcal{D}$ acts as the dealer D in the i-th linearization.
- Each $R_i \in \mathcal{R}$ executes the role the i-th receiver R_i in *each* of the $t+1$ linearizations.
- Each $D'_i \in \mathcal{D}'$ acts as the dealer D' in the i-th linearization.
- Each $R'_i \in \mathcal{R}'$ executes the role of the i-th receiver R'_i in *each* of the $t+1$ linearizations.

We denote a client who wishes to obtain the result of the protocol by C (C can be external, but can also be one of D_i, R_i, D'_i, R'_i).

The protocol starts with a "sharing" phase, where each D_i is executed one after another and shares its secret via Shamir's secret sharing to the receivers in $\mathcal{R}$, while committing to it using ElGamal's commitments. Additionally, each D_i sends its state to its counterpart D'_i. Then, parties in $\mathcal{R}$ are executed one after another, verify the shares they receive, and complain about the dealers who sent inconsistent shares. Finally, each dealer D'_i uses the state it received from its counterpart D_i to publicly respond to the complains. After this, the sharing phase is completed and the "reconstruction phase" begins. Here, each $R'_i \in \mathcal{R}'$ simply outputs the shares it received. Every party C who is interested in the output verifies the published shares, uses the ones that passed the verification to reconstruct the secret dealt by a particular dealer, and computes the xor of all secrets dealt by the dealers who were not deemed corrupt (i.e., publicly sent inconsistent information as a respond to a complain during the sharing phase). See Protocol 2 for details.

Execution-leaks variant. For the execution-leaks model, we similarly implement the behavior of each dealer using two roles – one responsible for the sharing of a secret, and one responsible for addressing the complaints. However, instead of implementing each R_i using two roles, we have $2t+1$ parties R_i and $t+1$ parties R'_j (where R'_i can *not* be thought of as a counterpart of R_i). Each R_i follows the procedure of round two, and if its shares verify, it additionally sends its shares to *each* R'_j. Finally, each R'_j publishes all shares (from all parties got the shares from) which verified correctly. See the full version for details.

Pipelining optimization. Implemented naively, in the sending-leaks model the protocol described above requires $6t+4$ parties, and its execution-leaks variant requires $5t+4$ parties. To reduce this number, we propose the following modification to both the sending-leaks and the execution-leaks protocols: Instead of combining multiple linearized VSS by having $t+1$ dealers, each of whom shares secrets among the *same set* $\mathcal{R}$ of $2t+1$ parties, we let each dealer share secrets among the *next* $2t+1$ parties. In the sending-leaks model, we now have $n = 5t+4$ parties P_i, where depending on the index i, party P_i executes the following roles:

- For $1 \leq i \leq t+1$, party P_i executes the role of the dealer D in i-th VSS linarization. If additionally $i > 1$, P_i also executes the role R_{i-j} in j-th VSS linearization, where $j < i$.
- For $t+2 \leq i \leq 3t+2$, party P_i executes the role R_{i-j} in j-th VSS linearization, where $j < i$. If additionally $i > 2t+2$, P_i also executes the role of the dealer D' in the $i-2t-2$-th VSS linearization.
- For $i = 3t+3$, party P_i executes the role D' in the $t+1$-st VSS linearization.
- For $3t+4 \leq i \leq 5t+4$, P_i executes the role R'_{i-3t-3} for each linearization.

Proofs of the following theorems can be found in the full version.

Theorem 4. *Assuming ElGamal commitments, there is a YOSO*$^{\mathsf{WCC}}$ $(t, 5t+4)$*-secure computational randomness generation protocol in the sending-leaks model.*

Theorem 5. *Assuming ElGamal commitments, there is a YOSO*$^{\mathsf{WCC}}$ $(t, 4t+4)$*-secure computational randomness generation protocol in the execution-leaks model.*

4 Implementation and Evaluation

We now evaluate our randomness generation scheme in the sending-leaks model from Section 3. In the following, we first compare it to our implementation of the randomness extraction protocol from [16]. We evaluate the work required to be performed by each role in the full version. Our implementation is available at https://github.com/yosorand/yoso-rand-elgamal and required ≈ 300 lines of code in Rust. We ran all our experiments single-threaded on a MacBook Pro with 32GB of RAM, and an Apple M1 Pro SoC.

Protocol 2 SL Randomness Generation from ElGamal Commitments

Sharing phase:
Each D_i, $i \in [t+1]$ does the following:

1. D_i chooses random degree-t polynomials f_1 and f_2:
$$f_1 = a_0 + a_1 x + \cdots + a_t x^t \text{ and } f_2 = b_0 + b_1 x + \cdots + b_t x^t.$$
2. D_i chooses a pair of generators (g, h).
3. D_i commits to f_1 and f_2 via broadcasting (g, h) along with
$$(c_0, c_1, \cdots, c_t) = \Big((g^{a_0}, h^{a_0} \cdot g^{b_0}), (g^{a_1}, h^{a_1} \cdot g^{b_1}), \cdots, (g^{a_t}, h^{a_t} \cdot g^{b_t})\Big).$$
4. D_i sends $r_j = f_1(j)$ and $s_j = f_2(j)$ to each P_j, $j \in [2t+1]$; and sends polynomials f_1 and f_2 to D'_i.

Each R_i, $i \in [2t+1]$ does the following:

1. For each dealer D_j, R_i checks whether the share (r_i, s_i) it obtained from D_j, and the commitments to f_1 and f_2 distributed by D_j satisfy
$$g^{r_i} = \prod_{k=0}^{t} (g^{a_k})^{i^k} \text{ and } h^{r_i} \cdot g^{s_i} = \prod_{k=0}^{t} \Big(h^{a_k} \cdot g^{b_k}\Big)^{i^k}$$
If not, R_i broadcasts $\mathsf{Complain} - \mathsf{D_j}$.
2. R_i sends all shares (r_i, s_i) that passed verification to R'_i.

Each D'_i, $i \in [t+1]$ does the following:

1. D'_i broadcasts shares of parties who complained about D_i. If any share broadcast by D'_i does not pass the check above, D'_i is deemed corrupt.

Reconstruction phase:
Each R'_i, $i \in [2t+1]$ does the following:

1. If R_i complained about D_j, and D'_j was not deemed corrupt, R'_i sets its corresponding share to s_i and r_i broadcast by D'_j.
2. R'_i outputs all shares (s_i, r_i) it obtained for non-corrupt dealers.

Client C does the following:

1. For each D'_i who was not deemed corrupt, C uses any $t+1$ shares s_j and r_j that pass the verification check against the corresponding commitment to reconstruct the value $s_i = f_i(0)$, where f_i is the polynomial f_2 dealt by D_i/D'_i.
2. Let H denote the index set of dealers D'_i which were not deemed corrupt. C outputs $\bigoplus_{i \in H} s_i$.

In our proof-of-concept implementation, we simulate the communication layer (i.e., the broadcast and point-to-point channels), and assume that the channels are authenticated. In our implementation all parties behave honestly, which corresponds to the worst case in terms of communication and computation complexity (same for NRO).

In terms of the running times (see Table 1), as expected, for very small values of t the NRO protocol is faster than our protocol. However, due to the exponential computational complexity of the NRO protocol, we outperform NRO already for $t = 6$, and the NRO scheme becomes impractical for values as small as $t = 8$. This gap will only increase as t grows.

t	Our scheme	NRO
1	314	0.64
2	821	2
3	1589	24
4	2793	236
5	4285	2463
6	6312	24387
7	8886	233328
8	11966	-

Table 1: Running time comparison, all times in milliseconds.

t	Our scheme	NRO
1	0.0031	0.0003
2	0.0067	0.004
3	0.0115	0.039
4	0.0176	0.378
5	0.0249	3.399
6	0.0336	29.182
7	0.0436	242.327
8	0.0548	-

Table 2: Overall communication sizes in MB.

Note that while in our evaluation we assume that all parties are single-threaded, our scheme is easily parallelizable: As a party often executes multiple roles for *independent* protocol executions of the linearized VSS, those roles can be executed by different cores. This slashes the cost roughly by a factor which corresponds to the number of cores available.

Finally, we report the overall data sizes that parties need to transmit both in our protocol and the NRO protocol (see Table 2). Again, while the NRO protocol is very efficient on very small values of t, our scheme outperforms it already for $t = 3$ (and stays remarkably low for larger values of t). This gap will only grow, as the NRO protocol has exponential communication complexity. As before in our running time experiment, we did not obtain the final values for the NRO scheme due to the timeout.

Acknowledgements

This work was supported by a Protocol Labs Cryptonet Network Grant RFP-013 “Stateless Distributed Randomness Generation”. C. Liu-Zhang’s research was also supported by the Hasler Foundation Project no 23090 and ETH Zurich Leading House Research Partnership Grant RPG-072023-19. J. Ribeiro’s research was also supported by NOVA LINCS (ref. UIDB/04516/2020) with the financial support of FCT - Fundação para a Ciência e a Tecnologia. Elisaweta Masserova was supported by a gift from Bosch and NSF Grants No. 1801369 and 2224279. We thank Jay Bosamiya for helping us with the implementation of this work.

References

1. Baignères, T., Delerablée, C., Finiasz, M., Goubin, L., Lepoint, T., Rivain, M.: Trap me if you can – million dollar curve. IACR Cryptol. ePrint Arch. (2015), http://eprint.iacr.org/2015/1249
2. Benhamouda, F., Gentry, C., Gorbunov, S., Halevi, S., Krawczyk, H., Lin, C., Rabin, T., Reyzin, L.: Can a public blockchain keep a secret? In: Pass, R., Pietrzak, K. (eds.) Theory of Cryptography. pp. 260–290. Springer International Publishing, Cham (2020)
3. Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: Advances in Cryptology – CRYPTO 2018. pp. 757–788. Springer (2018). https://doi.org/10.1007/978-3-319-96884-1_25
4. Cascudo, I., David, B.: SCRAPE: Scalable Randomness Attested by Public Entities. In: International Conference on Applied Cryptography and Network Security. pp. 537–556 (2017)
5. Cascudo, I., David, B.: ALBATROSS: Publicly AttestabLe BATched Randomness based On Secret Sharing. In: Advances in Cryptology – ASIACRYPT 2020. pp. 311–341. Springer International Publishing, Cham (2020)
6. Choi, K., Manoj, A., Bonneau, J.: SoK: Distributed randomness beacons. In: 44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023. pp. 75–92. IEEE (2023). https://doi.org/10.1109/SP46215.2023.10179419
7. Choudhuri, A.R., Goel, A., Green, M., Jain, A., Kaptchuk, G.: Fluid mpc: Secure multiparty computation with dynamic participants. In: Malkin, T., Peikert, C. (eds.) Advances in Cryptology – CRYPTO 2021. pp. 94–123. Springer International Publishing, Cham (2021)
8. David, B., Deligios, G., Goel, A., Ishai, Y., Konring, A., Kushilevitz, E., Liu-Zhang, C.D., Narayanan, V.: Perfect mpc over layered graphs. In: Annual International Cryptology Conference. pp. 360–392. Springer (2023)
9. Deligios, G., Goel, A., Liu-Zhang, C.D.: Maximally-fluid mpc with guaranteed output delivery. Cryptology ePrint Archive, Paper 2023/415 (2023), https://eprint.iacr.org/2023/415, https://eprint.iacr.org/2023/415
10. Gentry, C., Halevi, S., Krawczyk, H., Magri, B., Nielsen, J.B., Rabin, T., Yakoubov, S.: YOSO: You Only Speak Once. In: Annual International Cryptology Conference. pp. 64–93 (2021)
11. Gentry, C., Halevi, S., Magri, B., Nielsen, J.B., Yakoubov, S.: Random-index PIR and applications. In: Theory of Cryptography - 19th International Conference, TCC 2021, Raleigh, NC, USA, November 8-11, 2021, Proceedings, Part III. Lecture Notes in Computer Science, vol. 13044, pp. 32–61. Springer (2021)
12. Groth, J.: Simulation-sound NIZK proofs for a practical language and constant size group signatures. In: Lai, X., Chen, K. (eds.) Advances in Cryptology – ASIACRYPT 2006. pp. 444–459. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
13. Groth, J.: Non-interactive distributed key generation and key resharing. IACR Cryptol. ePrint Arch. (2021), https://eprint.iacr.org/2021/339
14. Kelsey, J., Brandão, L.T.A.N., Peralta, R., Booth, H.: A reference for randomness beacons: Format and protocol version 2. Tech. rep., National Institute of Standards and Technology (2019), https://csrc.nist.gov/pubs/ir/8213/ipd
15. Lenstra, A.K., Wesolowski, B.: A random zoo: sloth, unicorn, and trx. IACR Cryptol. ePrint Arch. (2015), http://eprint.iacr.org/2015/366

16. Nielsen, J.B., Ribeiro, J., Obremski, M.: Public randomness extraction with ephemeral roles and worst-case corruptions. In: Dodis, Y., Shrimpton, T. (eds.) Advances in Cryptology – CRYPTO 2022. pp. 127–147. Springer Nature Switzerland, Cham (2022)
17. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Feigenbaum, J. (ed.) Advances in Cryptology — CRYPTO '91. pp. 129–140. Springer Berlin Heidelberg, Berlin, Heidelberg (1992)
18. Rabin, M.O.: Transaction protection by beacons. Journal of Computer and System Sciences **27**(2), 256–267 (1983)
19. Shamir, A.: How to share a secret. Communications of the ACM **22**(11), 612–613 (1979)
20. Stadler, M.: Publicly verifiable secret sharing. In: Maurer, U.M. (ed.) Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding. Lecture Notes in Computer Science, vol. 1070, pp. 190–199. Springer (1996)
21. Syta, E., Jovanovic, P., Kogias, E.K., Gailly, N., Gasser, L., Khoffi, I., Fischer, M.J., Ford, B.: Scalable bias-resistant distributed randomness. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 444–460 (2017). https://doi.org/10.1109/SP.2017.45
22. Thyagarajan, S.A.K., Castagnos, G., Laguillaumie, F., Malavolta, G.: Efficient CCA timed commitments in class groups. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. p. 2663–2684. CCS '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3460120.3484773