YOSO: You Only Speak Once
Secure MPC with Stateless Ephemeral Roles

Craig Gentry[1], Shai Halevi[1], Hugo Krawczyk[1], Bernardo Magri[2], Jesper Buus Nielsen*[2], Tal Rabin[1], and Sophia Yakoubov†[3]

[1]*Algorand Foundation*
[2]*Concordium Blockchain Research Center, Aarhus University*
[3]*Aarhus University, Denmark*

June 12, 2021

Abstract

The inherent difficulty of maintaining stateful environments over long periods of time gave rise to the paradigm of *serverless computing*, where mostly-stateless components are deployed on demand to handle computation tasks, and are teared down once their task is complete. Serverless architecture could offer the added benefit of improved resistance to targeted denial-of-service attacks, by hiding from the attacker the physical machines involved in the protocol until after they complete their work. Realizing such protection, however, requires that the protocol only uses stateless parties, where each party sends only one message and never needs to speaks again. Perhaps the most famous example of this style of protocols is the Nakamoto consensus protocol used in Bitcoin: A peer can win the right to produce the next block by running a local lottery (mining), all while staying covert. Once the right has been won, it is executed by sending a *single* message. After that, the physical entity never needs to send more messages.

We refer to this as the You-Only-Speak-Once (YOSO) property, and initiate the formal study of it within a new model that we call the YOSO model. Our model is centered around the notion of *roles*, which are stateless parties that can only send a single message. Crucially, our modelling separates the protocol design, that only uses roles, from the role-assignment mechanism, that assigns roles to actual physical entities. This separation enables studying these two aspects separately, and our YOSO model in this work only deals with the protocol-design aspect.

We describe several techniques for achieving YOSO MPC; both computational and information theoretic. Our protocols are synchronous and provide guaranteed output delivery (which is important for application domains such as blockchains), assuming honest majority of roles in every time step. We describe a practically efficient computationally-secure protocol, as well as a proof-of-concept information theoretically secure protocol.

Keywords. Blockchains, Secure MPC, Stateless Parties, YOSO.

*Partially funded by The Concordium Foundation; The Danish Independent Research Council under Grant-ID DFF-8021-00366B (BETHE); The Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM).

†Funded by the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No 669255 (MPCPRO).

Contents

1 Introduction

A somewhat surprising feature of our networked world is just how hard it is to keep a working stateful execution environment over long periods of time. Even in non-adversarial settings, it is a major challenge to keep a server operational and connected through software updates, local physical events, and global infrastructure interruptions. This becomes even harder in adversarial environments. Consider for example a network adversary targeting a specific protocol, watching the communication network and mounting a targeted denial of service (DoS) attack on any machine that sends a message in this protocol. In high-stake environments, one also must worry about near-instant malicious compromise, unleashed by well equipped adversaries with a stash of zero-day exploits.

One approach for mitigating this issue is the paradigm of *serverless computing*, where mostly-stateless components are deployed on demand to handle computation tasks, and are torn down once their task is complete. In addition to economic benefits, a protocol built from such components could offer better resistance against strong adversaries by hiding the physical machines that play a role in the protocol, until after they complete their work and send their messages. To realize this protection, however, the protocol must utilize only stateless components, making it harder to design.

Perhaps the best-known example of this style of protocol is the Nakamoto consensus protocol used in Bitcoin [23]. A salient property of the Bitcoin design is that a peer can win the right to produce the next block by running a local lottery (mining), all while staying covert. Once the right has been won, it is executed by sending a *single* message. After that, the physical entity never needs to send more messages.[1] Another example is the Algorand consensus protocol [8] with its player-replaceability property.

In this work we initiate a formal study of protocols of this style, which we refer to as You-Only-Speak-Once (YOSO). An important conceptual contribution of our work is the (relatively) clean modeling of such protocols, centered around their use of *roles* (which is the name we use for those one-time stateless parties). Crucially, our modeling separates the protocol design using roles from the role-assignment functionality that assigns the roles to actual physical machines.

This separation lets us study the protocol design problem on its own, freeing us from having to specify the role-assignment implementation which is necessarily very system dependent: a proof-of-work blockchain will have very different role-assignment mechanisms from a proof-of-stake blockchain, and a traditional cloud environment will use yet other mechanisms. However, all these systems could use the same protocol for secure computation once the roles have been properly assigned. On the technical side we make the following contributions:

- We present a formal model for defining and studying such protocols, called the YOSO model, which in particular codifies the separation between role-assignment and protocol execution and codifies the notion of only speaking once. The YOSO model is cast within the UC framework [5] and therefore can draw on the existing body of research on UC security. An overview of the model is provided in Section 2, and a more detailed treatment can be found in Appendix A.

- We also devise tools for working in the YOSO model, and describe two different secure MPC protocols. Our main solution presented in Section 3 is an information theoretic proof-of-concept protocol that provides statistical security [2]. Additionally, in Section 4

[1]The same entity may later win the right to produce another block, but no secret state has to be maintained between the production of the two blocks, so the next winner might logically and physically be considered a separate entity.

[2]As we explain below, the restrictions of working in the YOSO model are so severe that a priory it was not

we also describe a computationally-secure protocol. Both protocols are synchronous and provide guaranteed output delivery (which is important for our application domain), assuming an honest majority of roles in every protocol step.

- We show that an information theoretic secure YOSO MPC can be compiled into a natural UC secure protocol running on a toy model of a blockchain with role assignment. This is meant as a sanity check of the abstract role-based YOSO model. It shows that protocols developed in this model can indeed be compiled to practice. We show that if we start with a static-secure (analogously, adaptive-secure) YOSO protocol, we can get a static-secure (analogously, adaptive-secure) UC protocol with essentially the same corruption threshold.

1.1 The YOSO Model

We introduce the YOSO model to make it easy to start studying YOSO MPC independently of blockchain and role assignment.

Role-based computation In the YOSO model, participants in protocols are called *roles* rather than parties or nodes or machines. The reason for the name "roles" is that we usually think of these one-time parties as playing some role in a protocol. Some examples of roles include "Party #3 in the 2nd VSS protocol on the 8th round", "the prover in the 6th NIZK", etc. Formally, a role is just a stateless party that can only send a single message before it is destroyed, and a protocol is an interaction between roles. Throughout this manuscript we use the following terminology:

Roles: are abstract formal entities that perform the protocol actions and communicate with other roles.

Nodes/Machines: refer to stateful long-living entities that the adversary can identify and target for corruption. These can be physical or virtual machines, that would typically have some identifying characteristics such as an IP address that can be used by the adversary to attack them.

We sometimes use the term *parties*, but only in informal discussions and in contexts where the distinction between roles and machines is immaterial.

Importantly, roles are detached from machines, and mapping of machines to roles happens at execution time. A protocol in the YOSO model will inevitably be executed alongside a role-assignment functionality, and the security of the protocol will rely on the guarantees provided by that functionality. Ideally, this assignment should be unknown to the attacker until after the machine plays its role and sends a message, hence limiting the adversary's ability to target the role for corruption.

The YOSO model can be used with different role-assignment functionalities with different guarantees. In this work we mainly consider a simple random-assignment functionality: it assigns each role to a random machine from among a universe of available ones, and hides that assignment from the adversary (unless the chosen machine is already corrupted). An adversary that corrupts machines will therefore be unable to predict which roles will be corrupted; upon corruption of a machine the adversary will be handed the random roles that are mapped to that machine. This allows for a simplified view of the adversary where all corruptions are random.

clear to us that information-theoretical security is even possible in the "$2t + 1$ regime". Indeed this work began as an attempt to prove that no such protocols exist.

We note that when a YOSO protocol is executed (with some specific role-assignment functionality), it is likely that over time the same machine can be assigned to different roles. But its state of an already executed role will be erased at the end of each role execution, just before it sends its only message in the protocol. Therefore a future corruption will not corrupt an already executed role.

An adversary might, however, by corrupting a single machine corrupt several future roles. If there are few machines compared to roles this will lead to a higher variance in how many roles are corrupted. To mitigate this the level of tolerated corruption of roles needs to be set higher than the tolerated level of corruption of machines.

1.2 MPC in the YOSO Model

Motivation One use-case of YOSO-style protocols was recently considered by Choudhuri *et al.* [10]: it involves volunteer-based participation in a computation, where requiring that a party only speaks once enables participation from parties who cannot commit their resources for long. We note that such a context seems to require a closed system with known identities; otherwise an adversary can volunteer enough malicious participants to form a corrupt majority, at least for a brief window, which may suffice for endangering the security of the computation.

When it comes to open systems, a compelling motivation for these protocols is scalable computation in the presence of an adaptive fail-stop adversary (a powerful DoS adversary, as noted earlier). Imagine a large number — perhaps millions — of nodes that want to engage in a secure computation in the presence of such an adversary. Assuming that the DoS adversary cannot take down more than some threshold of the nodes, running an MPC protocol among all of them would achieve the nodes' goals. However, running classical MPC protocols among large numbers of nodes is expensive. All of the nodes typically need to communicate with all of their peers, creating a prohibitive communication load. YOSO MPC enables the computation to be run by a small subset of the nodes, with an independent subset — or committee — participating in every round. YOSO MPC thwarts an adaptive DoS adversary because the adversary is unable to predict which fail-stops will be useful to foil the security; thus it creates the opportunity for execution of the protocol with small committees resulting in communication that is sub-linear in the number of nodes in the network.

As a more concrete example of a scenario where such scalable computation would be necessary, consider "MPC as a service". That is, an outsourced computation service where clients submit inputs for a joint computation so that the privacy of the inputs and the correctness of the output are guaranteed, even if a fraction of the provider's servers are adversarially controlled. However, while full corruption of servers is expensive, dedicated denial of service against targeted servers is an easier attack to carry out, and the protocol should be able to withstand it. YOSO MPC offers a solution that remains secure under these realistic conditions.

Role Assignment for YOSO MPC In order to reap the benefits of such scalable YOSO MPC, it is important to assign YOSO MPC roles to machines in a scalable way without revealing the role assignment before the roles need to speak. Furthermore, the assigned machines should be able to receive secret messages (even while the message senders do not know their identities). This is challenging since, being able to speak only once, the machine having won a role cannot first make a public key available, and then receive messages and execute its role in the protocol. This would involve speaking at least twice.

One solution that was recently proposed by Benhamouda *et al.* [3] involves the use of nominating committees: each machine has a public key for an encryption scheme allowing the rerandomization of public keys. For each role R there will be a delegator role D. (We call R the delegate, and D the delegator.) First a machine is assigned a delegator role D using, e.g., cryptographic sortition (or just by solving some puzzle). Then the delegator D will pick, uniformly at random, another machine to play the delegate role R. It will take that machine's public key pk_i, rerandomize it into $\widetilde{pk_i}$, and publish $\widetilde{pk_i}$. Note that $\widetilde{pk_i}$ does not reveal the identity of the machine now assigned to R; however, it enables other roles to send secret messages to the delegate R by encrypting to $\widetilde{pk_i}$. Finally, the delegate R will execute the role. One drawback of this approach is that the role R will be corrupt if the delegator is corrupt *or* if the delegate is corrupt. This essentially doubles the corruption budget of the adversary.

It is an interesting research direction to develop more practical and more secure role assignment mechanisms. However, this is orthogonal to the design of MPC protocols which will be run by the roles, which is the focus of our work. In Section 5 we give a toy example of compiling a YOSO protocol to run on top of a blockchain with role assignment to illuminate this compelling use case.

Parameters of YOSO MPC Protocols When designing a YOSO MPC protocol there is a number of interesting parameters to consider. In addition to the many "generic" aspects of MPC (such as corruption type and threshold, hardness assumptions, trusted setup, security guarantees, etc.) YOSO MPC protocols have some new parameters in their design.

- *Future/Past Horizon:* When a role speaks, it may send private messages to roles intended to speak in future rounds. The future horizon describes how far into the future a role may need to speak (similarly past horizon is how far back a role may need to listen). The method of assigning roles impacts and is impacted by the future and past horizons and should be taken into consideration. For example, for proof-of-stake systems it is undesirable to assign roles in advance using the current stake distribution. Or if roles are assigned on the fly parties would need to read the history of communication far into the past. One should therefore try to use as short a future/past horizon as possible.

- *Dynamic and Public Execution Time:* Static execution time refers to the ability to know ahead of time when a role would speak in the protocol, contrasted with the dynamic case where the time to speak is only determined at run-time. As YOSO protocols are ideal for serverless architectures where servers are only running when they need to act, static execution time may save resources (e.g. cloud rental).

  A related distinction (in the dynamic case) is whether only the role itself can determine when it is going to speak, or whether it can be determined publicly. (This could make a difference, e.g., in agreement protocols that must accumulate enough votes before moving to the next phase, we may want to know if we still need to wait for the vote from the role or can we assume that it crashed and will never vote.)

1.2.1 YOSO MPC from Additive Homomorphic Threshold Encryption

Our first technical contribution is a YOSO MPC in the computational setting with guaranteed output delivery in a synchronous model, tolerating a dishonest minority of roles at any given round. Specifically, in every round we will have some number n of roles that will form an honest-majority committee. As stated, it falls to the role-assignment functionality to supply us with committees with honest majority; in this work we allow ourselves to just assume that we have them.

Given a supply of committees with honest majority, our construction is based on the CDN protocol [11]. Informally, CDN requires a system-wide public key pk for an additively homomorphic threshold encryption scheme, where the secret key sk is shared among the committee members (with each member i holding sk_i). The participants then perform the entire computation using additive homomorphism, interspersed with public decryption of masked intermediate values. The protocol uses Beaver triples that are generated on-the-fly to support multiplications; the secret key shares are used to open values in every round of Beaver triple use, and to obtain the computation output at the end.

We note that CDN is already almost a YOSO protocol: the only state the participants need is the secret key shares sk_i, and the only messages that they send are their decryption shares (with the ciphertexts all being public). Providing the participants with shares of the global secret key sk can be done, e.g., using the proactive handover protocol of Benhamouda *et al.* [3], which is a YOSO protocol. In each protocol round, committee members get their decryption shares, and then the committee decrypts the current batch of ciphertexts and reshares sk to the next committee.

To get a YOSO protocol, we also need to generate the Beaver triples YOSO-style. We will use two committees — C_A and C_B — to generate many triples of the form $(\mathsf{Enc}(a), \mathsf{Enc}(b), \mathsf{Enc}(ab))$, which will be consumed by future committees during multiplications. We first have members P_i of committee C_A individually choose random a_i's and publish the ciphertexts $\overline{a_i} = \mathsf{Enc}(a_i)$ along with NIZK proofs that these are valid ciphertexts. All parties can use additive homomorphism to obtain $\overline{a}$, an encryption of the sum a of the a_i's. Then members P_j of committee C_B will individually choose random b_j's and set $\overline{b_j} = \mathsf{Enc}(b_j)$, then use additive homomorphism to compute $\overline{c_j}$, an encryption of $b_j a$. P_j then publishes $(\overline{b_j}, \overline{c_j})$, along with proofs that they were generated properly. All parties can use additive homomorphism to obtain $\overline{b}$ and $\overline{c}$, encryptions of the sums b of the b_j's and c of the $b_j a$'s, respectively. $(\overline{a}, \overline{b}, \overline{c})$ form a Beaver triple. Note that as long as all the NIZK proofs are valid and there is at least one honest party in each committee C_A, C_B, the triple is indeed a Beaver triple for the values $a = \sum_i a_i$ and $b = \sum_j b_j$ which are unknown to the adversary.[3]

We note that another approach for achieving computational security would be to leverage *fully* homomorphic encryption (FHE). This requires an FHE scheme with a one-message threshold decryption procedure, and also one whose secret key could be maintained proactively using a YOSO protocol. Proactive maintenance of the secret key can be achieved, e.g., using the YOSO handover protocol of Benhamouda *et al.* [3], and one-round decryption can be achieved using the techniques from Asharov *et al.* [1] and Mukherjee-Wichs [22] (after a one-time trusted setup to generate the required evaluation key). In terms of complexity, an FHE-based solution may be more efficient in number of rounds and total communication, but it requires much more local computation, more per-round communication, and a more complicated trusted setup.

1.2.2 YOSO MPC from Information Theoretic Techniques.

Our second (and main) technical contribution is a proof-of-concept information theoretic YOSO protocol with guaranteed output delivery in a synchronous model, tolerating any dishonest minority of roles at any given committee. This protocol does not need any trusted setup, but it relies on secure point-to-point channels between roles,[4] as well as a totally-ordered broadcast. One consequence of this protocol is statistically unbiased coin-flip in the YOSO model, which (together with appropriate role-assignment) implies unbiased public randomness in public blockchains via a YOSO protocol.

[3]If we have many honest parties in C_A, C_B (say m of them in each committee), then we can improve efficiency and get $\Omega(m)$ triples at roughly the same bandwidth using standard techniques.

[4]We note again that such secure point-to-point channels would have to be implemented somehow, even though the receiving role may not have been assigned yet to a machine. This task falls to the role-assignment functionality, which we do not specify in this work.

We begin by observing that YOSO is easy in the semi-honest model, in fact semi-honest BGW [2] is basically already a YOSO protocol. The BGW protocol only uses secret sharing and reconstruction: secret sharing can be done to a future committee (instead of the current one) over point-to-point channels, and reconstruction can be done publicly. When implementing a circuit, each multiplication gate has two committees, one for each round in the multiplication protocol. For a gate with large fan-out, the gate committee will reshare their shares to the committees of all the downstream gates.

It is only when switching to the malicious model when things get hard, as YOSO seems to rule out many common information-theoretic techniques. In particular patterns such as "committing" to a value and then being challenged on it, or even just using the same secret value in many parts of the protocol, seem to inherently require a party to stick around and speak more than once. The same can be said for cut-and-choose techniques that have a party generating multiple values, being challenged to open (say) half of them, and if they are all valid then the other half is used in the protocol.

It is also easy to see that simplistic solutions such as one party sending all its secret state to another will not help: It would allow the adversary to get this secret value if either the sender or the receiver are corrupted, hence amplifying the adversary's power. A more promising avenue is to let a party share its secret state with future committees (maybe more than one), and have these committees emulate it in the future as needed. However, ensuring that a message from one party is recoverable intact by future committees is challenging; this is essentially a verifiable-secret-sharing (VSS) functionality. Ensuring that the party shares *the same message* to multiple committees poses more challenges still. In Section 3 we address these challenges by gradually developing stronger and stronger primitives that build on each other. Here we just give a hint for some of the observations that enable these tools, and the various steps that go into the construction.

Step 1, Future Broadcast (FBcast). In Section 3.2 we describe a Future Broadcast construction that enables a party to prepare a message that should be broadcast in a future round. This may be complicated in general, since we need to ensure that the message delivered in the future is in fact the message of the party creating it, the kind of authenticity that often requires VSS. But in our context we observe that we only need to ensure this authenticity for honest party messages, as faulty parties can say whatever they want at any time. Hence we can assume an honest dealer, which makes the design a lot easier.

Observe that in the computational setting this is straightforward to achieve. A party shares its value using a Shamir secret sharing and also provides every share holder with a digital signature on the share. When the value is reconstructed only shares with valid signatures are taken into the interpolation, if they all lie on a degree-t polynomial polynomial then the constant term is taken as the broadcasted message. In the IT setting we show that if the dealer is honest, information theoretic MACs are sufficient to replace digital signatures in this construction.

Step 2, Distributed Commitment (DC). The construction of shares with digital signatures in fact offers some guarantees also when the dealer is faulty. Even in that case, it fixes a value at the time of sharing that the dealer is committed to, i.e., the constant term of the polynomial of degree at most t interpolated through the honest parties' shares (which might be null). This is (roughly) what we call a Distributed Commitment functionality. We say that this is a (distributed) commitment, because as with regular commitments, at the time when the value is revealed a faulty dealer has the option not to expose its value at all and have the output be null. Achieving distributed commitment in the IT setting is more complicated as we do not have signatures.

The next step is therefore fortifying the IT MACs into IT signatures (IT-SIG) to assure that a value held by an honest party does in fact verify when presented. Our techniques build on the VSS tools of Rabin and Ben-Or [26], but those techniques have to be adjusted to the YOSO model as they are interactive. We transform the protocol from [26] into one where a party knows in advance all the messages that it may need to send in the future. This makes it possible to replace the multiple speaking rounds in the original protocol by having each party share its future messages using FBcast to deliver the IT-SIG design (Section 3.3).

Thus, we create an IT-SIG that provides enough of the digital signature properties for the purpose of realizing distributed commitments. See Section 3.4.

Step 3, Duplicate DC (DupDC) and VSS. Proceeding towards VSS, we again turn to Rabin and Ben-Or [26], who utilize DC to achieve VSS via yet another cut-and-choose proof. The complication in this proof is that one value needs to be used multiple times. In the YOSO model, this requires creating duplicates of the same committed value, each to be used in a different step of the proof. Letting the dealer run multiple DC's does not work. It is a subtle point to see why this is an issue. The first point is that we do not trust the honesty of any single party. Thus, we would need the dealer to prove that all the committed values are the same. This will create a problem because for the proof to go through the committee holding the sharing would need to talk. Once they talk they have exhausted their one opportunity to speak and now the duplicate of the value has been wasted. Thus, we need to create a mechanism that duplicates values without "wasting" them. Surprisingly, we observe that our DC protocol allows the share holders themselves to create duplicates of the commitment. This avoids the need for additional proofs, the committee of shareholders is mostly honest so all the duplicates will be the same by design (see Section 3.5). Here, yet again, we can make all elements of the proof public, thus informing all parties of the result of the computation. This enables us to finalize the design of the VSS (Section 3.6).

To eventually complete the design of the MPC we would also need duplicates of the VSS as the same value might go into multiple gates and the committee holding the value can only speak once. Luckily, we can derive the duplicates of the VSS directly from the duplicates of the DC.

Step 4, Augmented VSS (AugVSS). We need one more level of sharing which we call Augmented VSS. In this level of sharing we add the property that not only is a secret s shared via VSS but also that all the shares that define the sharing of s are VSSed. This will enable the MPC.

Step 5, Secure-MPC. Once we have AugVSS, getting information-theoretic secure-MPC can be done using standard techniques that need to be adapted to the YOSO model. We maintain the variant throughout the computation that the values on the wires are AugVSS. Hence we prove:

Theorem. (informal) *Any multiparty function F can be securely implemented by an information-theoretic YOSO protocol in a network with broadcast and secret point-to-point channels, resilient against a fraction $\tau < 1/2$ of random Byzantine corruptions. The protocol additionally tolerates any number of chosen, Byzantine corruptions of input roles and output roles.*

It is crucial to practice that we can tolerate chosen corruptions of input roles and output roles. Often the inputs and outputs are given by known clients that could more easily be targeted with an attack.

Epilogue, Public Randomness. The cut-and-choose protocols in our design are described using access to public randomness (which defines the challenges in those protocols). But where can we get this public randomness? Producing true randomness in a distributed setting seems to require MPC, creating a circular problem. Yet, we can show that our protocols remain secure when using *unpredictable* (high min-entropy) values, rather than truly random ones. Producing public unpredictable values in the honest-majority setting is much easier, and can even be done in a YOSO fashion. Thus, we can complete the MPC without the need for true randomness.

Of course, once we are able to get full-blown MPC, we can use it to produce completely uniform public randomness. This in particular solves the problem of obtaining public uniform randomness on a public blockchain using a YOSO protocol, a problem that was explored by a few previous works [6, 7].

On the impossibility of Garay *et al.* [15]. In [15] it was shown that any protocol in the information theoretic model with a sublinear message complexity (in the number of parties) cannot withstand adaptive corruptions of a fraction equal or greater than $1 - \sqrt{0.5}$ of the total number of parties. Yet, we claim that our IT protocol can withstand less than $n/2$ adaptive corruptions. This is not a contradiction. Our proof proceeds in two steps. In the first we prove that our IT protocol is adaptively secure without the assumption of sublinear message complexity. In the second part, when we prove the protocol that has sublinear message complexity, we need to combine our IT protocol with some role-assignment mechanism. This inevitably takes our protocol out of the IT model, making the lower bound of [15] not applicable.[5]

1.2.3 Compiling Abstract YOSO to Natural YOSO.

Our YOSO protocols are abstract in that they only consider abstract roles; we abstract away role assignment and machines. To show that protocols designed in our abstract YOSO model can be compiled in practice we give a simple UC functionality $\mathcal{F}_{\mathrm{RA}}$ modeling a blockchain with role assignment. It will allow parties to post on the blockchain and it will spit out a sequence of random public keys where the corresponding secret key is known by a random, secret physical machine. Given a protocol π for the YOSO model using hybrid functionalities $\mathcal{F}_{\mathrm{BC}}$ for broadcast and $\mathcal{F}_{\mathrm{SPP}}$ for secure point-to-point message transmission we can compile it to the UC $\mathcal{F}_{\mathrm{RA}}$-hybrid model. We use the blockchain for emulating $\mathcal{F}_{\mathrm{BC}}$. We emulate $\mathcal{F}_{\mathrm{SPP}}$ by encrypting under the public keys for future roles which occur on the blockchain.

We prove two results. We show that we can compile a YOSO protocol using hybrid functionalities $\mathcal{F}_{\mathrm{BC}}$ and $\mathcal{F}_{\mathrm{SPP}}$ and which IT YOSO implements a secure function evaluation of F against τ *random*, static corruptions into a UC secure protocol for the $\mathcal{F}_{\mathrm{RA}}$-hybrid model which tolerate ρ *chosen*, static corruptions for any $\rho < \tau$. We show the same for adaptive security.

We can get security against chosen corruptions from security against random corruptions because the adversary does not know the role-to-machine association chosen by $\mathcal{F}_{\mathrm{RA}}$. So corrupting a machine corrupts random roles.

1.3 Related Work

Protocols built out of ephemeral one-time roles became popular over the last decade with the emergence of public blockchains, whose defining feature is not relying on long-term participants with fixed identities. In particular, starting with Nakamoto's consensus protocol [23], these protocols became popular for achieving agreement in different settings, e.g., [21, 24, 8, 4].

[5]Specifically, the implementation of our communication channels which are needed to enable the solution can only be achieved in the computational setting (in our specific case we assume a PKI and more).

Only very recently did we start seeing attempts at using this style of protocols for other cryptographic tasks: Benhamouda *et al.* [3] described how to use such protocols for long-term maintenance of secrets on public blockchains, and mentioned the possibility of using these secrets for various tasks, including for general-purpose secure computation. Blum *et al.* [4] described how to implement input-free protocols in this model (such as coin tossing), and also described informally an FHE-based solution for functions with input (similar to the one sketched in Section 1.2.1 above).

Choudhuri *et al.* [10] described general-purpose secure-MPC protocols of this style (that they call *fluid*), but only achieving security with abort. The motivation of Choudhuri *et al.* was participants that volunteer for roles (in our terminology we would call it a volunteer-based role-assignment functionality). We note that this type of role assignment seems to be tailored to closed systems that have participants with known identities; otherwise an attacker can run a Sybil attack and get a majority of roles just by volunteering many times.

2 YOSO for the Working Cryptographer

The YOSO model can be cast within the UC framework [5] by identifying the roles in YOSO protocols with the party identifiers of the UC framework. This means that the roles are executed by the UC model, which completely abstracts away how these roles are actually assigned to physical machines; in fact, there is not even a notion of physical machines left. We then introduce a notion of random corruptions that are out of the control of the adversary. This can be used to model a set of roles which, in the now abstracted away real world, are hidden inside random physical machines, and the adversary can corrupt machines of its choosing.

Below we always use the term roles rather than parties, just to stress that we are in the YOSO model. This terminology is for didactic purposes only; a role in our formal model is identical to a party in the normal UC framework. The "speak once" aspect is enforced by our execution model, as we now explain.

2.1 YOSO Wrappers

To force roles to only speak once, we are explicitly "yosofying" them with a YOSO wrapper. Namely, our execution model postulates a wrapper around each role, that kills it immediately after the first time that it speaks. When that happens, the wrapper sends a Spoke token to the environment, the adversary and all its sub-routines (sub-protocols and ideal functionalities). Thereafter it responds with a Spoke token to the environment whenever activated, and only sends Spoke to the sub-routines that it is connected to.

Defining what it means for a role to "speak for the first time" is somewhat nontrivial. The main issue to tackle is whether sending messages to functionalities constitute speaking. To see the issue, consider a protocol Π (that implements some functionality $\mathcal{F}$), in which a role R must listen for many incoming messages before deciding to send a message. In this case, the $\mathcal{F}$-hybrid model could have the role R sending its input to $\mathcal{F}$ very early, but the implementation would have R actually speaking much later.

To account for that, we let functionalities reply to parties with a special Spoke token. The functionality can freely choose when to send this token, and the YOSO wrapper will kill the role as soon as it receives a Spoke token from any functionality. For example, a communication-channel functionality will reply with a Spoke token as soon as a party sends anything on it, while a higher-level functionality may trigger a Spoke token based on some input from the adversary. Note that when a communication channel outputs Spoke to a role, the role will pass it on to all its sub-routines and then its environment/outer protocol. Hence the entire composed role will be crashed.

We denote the "yosofied" role R by $\mathsf{YoS}(R)$, and the protocol that we get by yosofying all the roles in Π is denoted by $\mathsf{YoS}(\Pi)$.

2.2 Random Corruptions

In addition to the usual corruptions of the UC model we also model random corruptions in the YOSO model — that is, corruptions out of the control of the adversary.

We do this without changing the UC framework itself. Recall that in UC a corruption is implemented by the adversary just writing $(\mathtt{corrupt}, cp)$ on the backdoor tape of the party, where cp is some auxiliary information like the type of corruption: Byzantine, semi-honest, *et cetera*. There is no explicit mechanism in UC for limiting how many parties are corrupted or with which flavor. However, we often choose to analyze protocols under a restricted set of corruptions. This is simple to do by only quantifying over adversaries adhering to this restriction. This is easy to formulate for settings like "only semi-honest corruptions" or "at most a minority of the parties". However, it seems to be trickier for random corruptions: if the adversary corrupts a role R, how can we know that R was chosen at random? We need a precise meaning for this in order to be able to make precise security claims. For this purpose, we introduce a simple notion called the corruption controller ($\mathcal{CC}$), that runs as part of the environment. If an adversary wants to do a random corruption, it asks the environment, which will pass the request to the $\mathcal{CC}$. Then, the $\mathcal{CC}$ will sample the corruption and inform the adversary which role was corrupted (via the environment). If the environment sees the adversary is not respecting the decision of the $\mathcal{CC}$, then the environment will make a random guess in the security game. This enforces that no distinguishing advantage comes from executions violating the will of the $\mathcal{CC}$. We then only prove security under the class of environments having such a $\mathcal{CC}$ and using it as intended. We call this the class of controlled environments.

These random corruptions can be mixed freely with other corruption types, but it is illustrative to consider a generalization of the usual adversary structures to random corruptions. We codify the corruption power of the adversary by means of *corruption structure*.

Let Role be the set of (names of) roles in the system. A corruption structure on Role is a set of probability distributions over $2^{|\mathsf{Role}|}$. A static adversary would choose at the beginning of the execution a specific corruption distribution $C \in \mathcal{C}$ and give it to the $\mathcal{CC}$ via the environment. Then the $\mathcal{CC}$ samples $c \leftarrow C$ and give it to the adversary via the environment, and each role $\mathsf{R} \in \mathsf{Role}$ can now be corrupted if $\mathsf{R} \in c$. Note that a corruption structure with only point distributions (i.e. with a single probability-one pattern $c \in C$) corresponds exactly to standard static corruptions with these allowed patterns, coinciding with the notion of general adversary structure of Hirt and Maurer [19]. We stress that corruption structure represents our *assumption* about the corruption power of the adversary when designing the protocol. It is up to the role-assignment functionality to ensure that realistic adversaries will be unlikely to exceed this power.

When considering adaptive corruptions several choices are possible. We consider two in this work called sample corruptions and point corruptions. In sample corruptions the adversary gives a distribution on a set of roles and gets one of them corrupted, within some bound. In point corruptions the adversary can ask permission to corrupt a given role with some limited probability. If the corruption fails the role stays honest forever after. It is interesting future work to explore the relation between different notions of random corruptions.

On the first input TICK sample $(pk_{\mathsf{R}}, \mathsf{sk}_{\mathsf{R}}) \leftarrow \mathsf{Gen}$ for all $\mathsf{R} \in \mathsf{Correct} \cup \mathsf{Crash}$. Output pk to $\mathcal{O}$. For all $\mathsf{R} \in \mathsf{Leaky}$ output sk_{R} to $\mathcal{O}$. For each $\mathsf{R} \in \mathsf{Malicious}$ query $\mathcal{O}$ to get the keys $(pk_{\mathsf{R}}, \mathsf{sk}_{\mathsf{R}})$ for R. Then for each $\mathsf{R} \in \mathsf{Correct}$ output $(\mathsf{sk}_{\mathsf{R}}, \{ pk_{\mathsf{R}'} \}_{\mathsf{R}' \in \mathsf{Role}})$ to R.

Figure 1: The ideal functionality $\mathcal{F}_{\mathsf{Gen}}$ for a very simple PKI setup with key generator Gen.

2.3 YOSO Security

The notion of a protocol realizing a functionality is borrowed from the UC model. Namely, we say that Π YOSO-realises $\mathcal{F}$ for some class of environments (possibly using random corruptions) if $\mathsf{YoS}(\Pi)$ UC-realises $\mathcal{F}$. The considered class of environments should be a subset of the controlled environments.

It is easy to see that UC composition still holds for controlled environments. If an environment is composed with a protocol or simulator to define a new environment, as happens in the proof of the UC theorem, then this composed environment still uses the $\mathcal{CC}$ of the original one. The same holds when one composes an environment with a simulator. Therefore we get UC composition also for controlled environments.

YOSO composition then follows directly from UC composition. Let Π be a protocol for the $\mathcal{G}$-hybrid model and assume that Π YOSO-realises $\mathcal{F}$. Assume that Γ YOSO-realises $\mathcal{G}$. As usual in the UC framework let $\Pi^{\mathcal{G} \to \Gamma}$ be the protocol Π with calls to $\mathcal{G}$ replaced by calls to Γ. It follows that $\Pi^{\mathcal{G} \to \Gamma}$ YOSO-realises $\mathcal{F}$. To see this, note that the premises give us that $\mathsf{YoS}(\Pi)$ UC-realises $\mathcal{F}$ and that $\mathsf{YoS}(\Gamma)$ UC-realises $\mathcal{G}$. By the usual UC theorem we get that $\mathsf{YoS}(\Pi)^{\mathcal{G} \to \mathsf{YoS}(\Gamma)}$ UC-realises $\mathcal{F}$. Then use that by construction $\mathsf{YoS}(\Pi)^{\mathcal{G} \to \mathsf{YoS}(\Gamma)} = \mathsf{YoS}(\Pi^{\mathcal{G} \to \Gamma})$. This follows by the way the YoS wrapper passes around the SPOKE token to shut down entire composed parties.

2.4 Common Features, Functionalities, and Models

Synchrony. To simplify the treatment of synchronous clocks, we assume that in every round the environment sends a TICK message to all the roles *and also to all the functionalities* and the adversary, in addition to any other inputs that it wants to provide them. We use the model in [20] for this.

Communication channels and PKI. We assume at least an authenticated broadcast channel denoted $\mathcal{F}_{\mathsf{BC}}$, and usually also secure point-to-point channels $\mathcal{F}_{\mathsf{SPP}}$ (or at least authenticated channels $\mathcal{F}_{\mathsf{PP}}$). These functionalities are defined more or less as usual in the UC framework, except that in our case they return a SPOKE token to any role immediately in the step following the receipt of message from it.[6] These functionalities are formally presented in Appendix A. We also sometimes use a PKI functionality, which is specified in Figure 1.

YOSO Secure Function Evaluation. We consider secure function evaluation in the YOSO model. We assume that the roles of a protocol Π are divided into input roles, output roles and computation roles. The input roles receive inputs from the environment and the output roles will deliver the outputs back. The computation nodes carry out intermediary steps of the computation and do not interact with the environment.

As usual for UC-like models, to formulate the assertion that a function F could be computed securely we need to wrap that function by a compatible functionality $\mathcal{F}_{\mathrm{MPC}}^{F}$, as described in Appendix A (cf. Figure 13). Importantly, we assume that the roles receiving the output *do not*

[6]We allow a role to send messages on multiple channels in the same step, then it will receive SPOKE tokens from all of them in the next step.

speak in an implementation (so $\mathcal{F}_{\mathrm{MPC}}^{F}$ never sends SPOKE tokens to the output roles). Otherwise these output roles would not be able to contribute the result to the higher-level protocol.

By default, we assume that the roles receiving the inputs and the roles giving the outputs can be corrupted using the usual chosen corruptions. This is reasonable since in most of the meaningful high-level protocols, like elections, the inputs to the protocol are given by known machines that might be subject to targeted DoS attacks. Computation nodes however, are only subject to random corruptions; when running in the "real world" with a concrete role assignment mechanism, we get to execute computation roles on random machines.

We then say that Π YOSO securely implements F with a fraction τ random corruptions if Π implements $\mathcal{F}_{\mathrm{MPC}}^{F}$ against any number of chosen corruptions of input roles and output roles and random corruptions of up to a fraction τ of the computation roles.

The IT YOSO Model. We define the standard IT YOSO model to be the model with broadcast and secure point-to-point channels, unbounded environments, and poly-time protocols, ideal functionalities and simulators.

The Computational YOSO Model. The computational YOSO model is equipped with an authenticated broadcast channel, perhaps authenticated point-to-point channels, a PKI functionality (such as the one from Figure 1), and poly-time environments, protocols, ideal functionalities and simulators.

3 The Information-Theoretic $t < \frac{n}{2}$ MPC Protocol

In this section we describe an MPC protocol in the information theoretic YOSO model for $\tau < 1/2$ random Byzantine corruptions.

Theorem 1. *For any multiparty function F, there exists a poly-time protocol Π described below running with the network $(\mathcal{F}_{\mathsf{BC}}, \mathcal{F}_{\mathsf{SPP}})$ which YOSO-realizes the ideal functionality $\mathcal{F}_{MPC}^{F}$ in the information theoretic YOSO model. The protocol tolerates any number of chosen, Byzantine corruptions of input roles and output roles, and for any $\tau < 1/2$ it tolerates adaptive, Byzantine, random τ-point-corruptions of computation nodes.*

Recall that the reason we allow chosen corruptions of input roles and output roles is that in a real-life setting we cannot reasonably assume that it is unknown which machines will give input or get the outputs. So input and output roles could be targeted. On the other hand, we want to model that computation roles are run on random, secret machines, so we only allow random corruptions of computation nodes. Recall that τ-point corruptions just means that the adversary can point to a role R and ask for a corruption. Then the role is made corrupted with probability τ, and with probability $1 - \tau$ it will remain honest forever after. The type of random corruption it not essential for our proof. The reason why we prove security against point corruptions is that this is the type of corruption needed for the compilation result in Section 5.

Below we will phrase the protocol in terms of disjoint *committees* of size n. We call the roles in a committee *parties*. Let c be the number of committees that we need. We then start with $N = cn$ computation roles $\mathsf{R}_1, \ldots, \mathsf{R}_N$. We call the committees $\mathsf{C}_1, \ldots, \mathsf{C}_c$ where $\mathsf{C}_j = \{\mathsf{P}_1^j, \ldots, \mathsf{P}_n^j\}$ and $\mathsf{P}_i^j = \mathsf{R}_{i+(j-1)n}$. We call P_i^j *party i in committee j*. Notice that this grouping of roles into committees is static. This does not affect security as the adversary cannot bias corruption towards a specific committee. Each party is still subject only to τ-point corruption. If we set $\tau < 1/2$ then we can clearly pick n large enough that we can conclude from a tail bound that all committees have at most $t < n/2$ corrupted parties except with negligible probability. For the rest of the section we then assume that this has been done. From this point on the only assumption we need for security is that each committee has $t < n/2$ corrupted parties.

Note that we allow any number of corruptions among input roles and output roles. However, input roles and output roles are not part of committees, so this does not violate the honest majority assumption for committees.

Our protocol is adaptive secure. We will, however, below mainly prove static security and only briefly discuss adaptive security. The reason is that for point corruptions, the distinction between adaptive corruptions and static corruptions is minimal. An adaptive point corruption just means that the adversary chooses to be oblivious to whether a party is corrupt or not until the point corruption. This gives it no new powers over static corruptions. Note, in particular, that corruption control component $\mathcal{CC}$ could sample before the UC execution starts for each role R_i a bit b_i which is 1 with probability τ. If later the adversary does a point corruption of R_i it will become corrupted if and only if $b_i = 1$. Therefore, even in the adaptive case, the corruptions can be thought of as being static: they were chosen before the execution started. The only complication in proving adaptive security compared to proving static security is then that in the adaptive case, the simulator will not know b_i until the adversary does a point corruption of R_i. Below we phrase the proof in terms of static security. The proof can be adapted to the adaptive case using standard techniques.

The challenge in designing an information-theoretic MPC protocol in the YOSO model is in replacing the actions of parties that interact and speak multiple times in regular MPC protocols with parties (more precisely, roles) that speak only once. For this we introduce several tools and components for YOSO adaptation that may be useful for other protocols as well. A first such tool is *Future Broadcast (*FBcast*)* that allows a party P, that in the standard model would speak in several rounds, to send its future messages to future roles that will transmit the messages (either privately or through broadcast) when the time for those messages to be delivered comes. For example, consider a non-YOSO protocol where a party P transmits a message m at round i and a message m' at round $i + 3$. In the YOSO adaptation, the role representing the actions of P in round i will transmit m at round i and also, in the same round, apply $\mathsf{FBcast}(m')$ to pass message m' to a role that will speak m' in round $i + 3$. Note that this procedure is possible only in cases where the future message is known in advance. An interesting point to observe is that correctness of FBcast (in particular, in terms of correctness of messages sent "into the future"), needs only be guaranteed for original senders of m' which are honest as faulty ones can choose to speak any message of their choice whenever they speak. The sender P_i^j uses $\mathsf{FBcast}(m')$ to replace its own sending of m' in the future. In the emulated protocol a corrupt P_i^j could send $m'' \neq m'$ at this future point. So it is tolerable that $\mathsf{FBcast}(m')$ may open to $m'' \neq m'$ in the future when P_i^j is corrupt.

As a first application of FBcast, we use it to adapt the IT-SIGs of [26, 25] to the YOSO model and then use this YOSOfied primitive to build a Distributed Commitment (DC) protocol in the YOSO model. In it, a party (honest or faulty) commits to a value that it can later choose to reveal or not, but it cannot change the committed value. Furthermore, it is guaranteed that values committed by honest parties are always revealed correctly. We then use DC as an essential ingredient in the design of a YOSO Verifiable Secret Sharing (VSS) scheme which in turn is a central component of our YOSO information-theoretic MPC solution.

In various steps in our protocol we need access to some form of randomness and for clarity of presentation we will assume the presence of a beacon functionality. However, in actuality we need something much weaker than a truly random source to deliver our results, it is enough that the challenge cannot be guessed. Thus, we can have a very simple implementation of the beacon (see Section 3.11). We denote this functionality as $\mathcal{F}_{\mathsf{UPBeacon}}$ to reflect that it is an unpredictable beacon. During the analysis we at first assume it returns uniformly random elements. At the end we then return to why it is enough that it is unpredictable and how to implement it.

On (t, n)-Shamir secret sharing and honest majority. The solutions presented in this section make essential and repeated use of secret sharing techniques. In all cases, the underlying scheme is Shamir's scheme over a given field, and we assume all committees into which secrets are shared to have at least $t + 1$ honest parties where $t + 1 > n/2$. Thus, the polynomials defining shares are of degree t.

3.1 Information Theoretic and Homomorphic MAC

Message authentication codes (MAC) are used for verifying the authenticity of messages between a sender and receiver that share a secret key. Following the construction of [26] we have the following two protocols.

Three-party Setting. There exists (i) a sender S holding a message m, it chooses a key K and generates its corresponding MAC tag M computed under a key K; (ii) S sends the pair (m, M) to a receiver R; (iii) S sends the key K to a verifier V. The verification procedure combines the pair (m, M) held by R with the key K held by V.

For our purposes, we consider an information theoretic MAC function with the following properties: (i) producing a correct MAC without knowing the key succeeds with negligible probability even for an unbounded attacker; (ii) message hiding: nothing is learned about the message m from the key K; (iii) homomorphic: the MAC function is homomorphic with respect to appropriate group operations in the following sense. If $M_i = \mathsf{MAC}_{K_i}(m_i), i = 1, 2$, and the keys K_1, K_2 were computed by the same party (they might need to be correlated) then $M_1 + M_2 = \mathsf{MAC}_{K_1 +' K_2}(m_1 + m_2)$.

Such a MAC can be implemented as follows (all elements and operations are over a finite field, e.g., $\mathbb{Z}_p$): $K_i = (a, b_i)$, $M_i = am_i + b_i$ and $K_i +' K_j = (a, b_i + b_j)$. In the sequel, we will say that keys that share the same coefficient a but differ in b_i are *correlated*.

MAC with Distributed Public Verification. In the above setting, to verify a MAC one has to trust V to provide the correct key. In the scenarios in this paper, we often do not trust any single party individually, but rather can only count on committees with a majority of honest participants. Thus, we extend the basic 3-party scheme to one where the role of V is instantiated by an n-party committee $\mathsf{V} = \{V_1, \ldots, V_n\}$. Given a message m that S hands to R, S creates a MAC for m as follows. For $i = 1, \ldots, n$, S chooses keys K_i, computes $M_i = \mathsf{MAC}_{K_i}(m)$, and provides all M_i to R and K_i to V_i. When m needs to be verified, R first broadcasts m and the values M_i. Then, each V_i broadcasts K_i and the value m is accepted (i.e., the MAC validates) if and only if it holds that $M_i = \mathsf{MAC}_{K_i}(m)$ for at least $t + 1$ values of i.

The scheme guarantees that if S follows the protocol and $t + 1 > (n - 1)/2$ members of V are honest, then only a message m originating from S will be accepted. Note that the validation of m is public once R and members of V broadcast their values.

When the MAC in use is homomorphic, we have that if S MACs messages m_1, m_2 in the above way, with the same R and same committee V, then the message $m = m_1 + m_2$ can be validated as follows. R outputs m and $M_i = M_i^{(1)} + M_i^{(2)}$, $i = 1, \ldots, n$, and each V_i outputs $K_i^{(1)} +' K_i^{(2)}$. Here, $M_i^{(1)}, M_i^{(2)}$ are the MAC values received by R for m_1 and m_2, respectively, and $K_i^{(1)}, K_i^{(2)}$ are the keys received by V_i for m_1 and m_2, respectively. We therefore say that this *MAC procedure is homomorphic.*

FBcast.Share (Executed by S on input m)	FBcast.Reveal(with public verification)
Set two n-party committees, ShareHolder and ShareVerifier. 1. Compute a (t, n)-secret sharing $(m_1, \ldots, m_n)$ of m for $t = (n-1)/2$. 2. Generate keys $K_{i,j}$, $1 \leq i, j \leq n$ and compute $M_{i,j} = \mathsf{MAC}_{K_{i,j}}(m_i)$. 3. For $i = 1, \ldots, n$: Send $m_i, M_{i,1}, \ldots, M_{i,n}$ to ShareHolder$_i$; Send $K_{1,i}, \ldots, K_{n,i}$ to ShareVerifier$_i$.	1. ShareHolder$_i$ bcasts $m_i, M_{i,1}, \ldots, M_{i,n}$. 2. ShareVerifier$_i$ bcasts $K_{1,i}, \ldots, K_{n,i}$. 3. Accept m_i iff $M_{i,j} = \mathsf{MAC}_{K_{i,j}}(m_i)$ for at least $t+1$ of the keys. 4. If there are at least $t+1$ accepted shares and they all define a single polynomial of degree t then output the constant term. Otherwise, output "fail".

Figure 2: Future Broadcast Protocol

This protocol is inherently YOSO as each party speaks only once and we refer to it in the following as IT-MAC.

3.2 Future Broadcast

We introduce *Future Broadcast* (FBcast), a fundamental primitive in the YOSO setting that allows an *honest* party P that speaks at time t to prepare a message m for broadcasting at a future time t'. This is accomplished by having P simply secret share m to a committee that will broadcast m at time t', hence bypassing the limitation of speaking only once. To guarantee that the message can be reconstructed (in the case that P is honest and the committee has an honest majority), FBcast implements a robust secret sharing scheme. Namely, a scheme where correct reconstruction is guaranteed as long as the sharing was done correctly and at least $t+1$ honest parties provide their shares (i.e., bad shares from corrupt parties can be identified and eliminated). In settings where digital signatures are available, robust secret sharing is implemented by having the dealer sign its shares. In our information-theoretic setting, we achieve a similar effect using the IT-MAC procedure from Section 3.1 for verifying share integrity.

The FBcast protocol is presented in Figure 2. Its first phase, FBcast.Share, is executed by a party S on input message m. It consists of S secret sharing m with a committee ShareHolder where in addition to its share, each ShareHolder$_i$ receives a IT-MAC of the share computed by S using the above distributed MAC procedure. An additional committee, ShareVerifier, receives the MAC keys from S. When the value m needs to be broadcast in the future, FBcast.Reveal is performed following the distributed verification procedure: the ShareHolder members first broadcast their shares together with their MAC values, followed by a broadcast of keys held by ShareVerifier (note that ShareVerifier must speak after ShareHolder hence requiring two separate committees). Shares that do not pass verification are discarded and if those that remain interpolate to a single polynomial of degree t, the secret is reconstructed, otherwise reconstruction fails.

We denote by FBcast.Share$_S(m)$ the sharing by S of a value m for future revealing and FBcast.Reveal$_S(m)$ the revealing of m (executed by two committees), and refer to the whole protocol execution as FBcast$_S(m)$.

Analysis.

We show that FBcast satisfies the requirement that if S is honest and used m as input to FBcast.Share then m will be reconstructed when FBcast.Reveal is executed. For this we need to show that only m_i's that originated from S are accepted and that there are sufficiently many accepted shares to interpolate the polynomial. If m_i is accepted then the MAC was verified by a key broadcast by at least one honest $\mathsf{ShareVerifier}$. As S is honest, only m_i's created by S are accepted by an honest party. Furthermore, each share broadcasted by an honest $\mathsf{ShareHolder}$ is accepted as there will be at least $t+1$ honest $\mathsf{ShareVerifiers}$ whose broadcasted keys satisfy the MAC. By construction, no party speaks twice.

Homomorphism of FBcast.

Note that when used with a homomorphic MAC, FBcast inherits the homomorphic property of the distributed MAC scheme from Section 3.1. We denote this fact as $\mathsf{FB}_P(m_1) + \mathsf{FB}_P(m_2) = \mathsf{FB}_P(m_1 + m_2)$ for any messages m_1 and m_2 shared by the same party P. Yet, as the keys need to be correlated the creator of the MAC needs to know in advance what two values will be added. This is easily achievable in our protocols.

3.3 Homomorphic IT-SIG

Our protocols would benefit from a signature functionality in order to construct a VSS protocol. Of course in the information theoretic setting we cannot achieve the full properties of a signature, but we can achieve enough of the functionality to deliver the result. The property which we need is the following. Assume again the setting from the IT-MAC (Section 3.1). We would want to assure R that the message that it holds will be accepted by the committee V. In essence, that it has a "signature" on the message that it holds.

Unlike the transformation of the basic IT-MAC from [27] that did not require modification to comply with the YOSO model, the IT-SIG construction from that paper does require changes as it has interaction. Our protocol IT-SIG is described in Figure 3. It consists of two phases, $\mathsf{IT\text{-}SIG.Setup}$ and $\mathsf{IT\text{-}SIG.Reveal}$. In $\mathsf{IT\text{-}SIG.Setup}$, a sender S provides a receiver R with a value m and also provides verification information to a committee V of n verifiers $V_1, \ldots, V_n$. The goal is for R to disclose m in the $\mathsf{IT\text{-}SIG.Reveal}$ phase in a way that allows to *publicly verify* the correctness of m with the help of committee V and with the following guarantees, assuming that V contains an honest majority:

- If S and R are honest then the correct value m is disclosed and verified during $\mathsf{IT\text{-}SIG.Reveal}$ and no information on m is revealed prior to that.
- If both S and R are corrupt we make no requirement at all.
- If only S is corrupt, at the end of $\mathsf{IT\text{-}SIG.Setup}$, R holds a value m' that will pass verification in $\mathsf{IT\text{-}SIG.Reveal}$.
- If only R is corrupt, no value other than the m that originated with S in $\mathsf{IT\text{-}SIG.Setup}$ can pass verification in $\mathsf{IT\text{-}SIG.Reveal}$.

In addition, the protocol needs to satisfy the YOSO model where parties speak only once. We build it so that R speaks only once (either in $\mathsf{IT\text{-}SIG.Setup}$ or in $\mathsf{IT\text{-}SIG.Reveal}$) while in the case of S and the parties in V, from which the logic of the protocol requires more than one message, we resort to FBcast for distributing their future messages so that a different committee broadcasts them when needed, and all parties speak only once.

IT-SIG.Setup

1. On input m, the sender S:
   (a) Generates keys $K_{i,j}$, $1 \leq i \leq n, 1 \leq j \leq \kappa$ (for security parameter κ), and computes $M_{i,j} = \mathsf{MAC}_{K_{i,j}}(m)$.
   (b) Transfers $(m, \{M_{i,j}\}_{1 \leq i \leq n, 1 \leq j \leq \kappa})$ to receiver R and $\{K_{i,j}\}_{1 \leq j \leq \kappa}$ to V_i.
   (c) Executes $\mathsf{FBcast.Share}_S(m)$, and $\mathsf{FBcast.Share}_S(K_{i,j}), 1 \leq i \leq n, 1 \leq j \leq \kappa$.
2. Party V_i:
   (a) Chooses half of the indices at random, denoted by INX_i.
   (b) Broadcasts $K_{i,j}$ for $j \in INX_i$.
   (c) Executes $\mathsf{FBcast.Share}_{V_i}(K_{i,j})$, $j \notin INX_i$.
3. Execute $\mathsf{FBcast.Reveal}_S(K_{i,j})$ $j \in INX_i$ for all i; denote by $\bar{K}_{i,j}$ the reconstructed values.
4. If there exist indexes i and j for which $\mathsf{MAC}_{\bar{K}_{i,j}}(m) \neq M_{i,j}$ then R asks that $\mathsf{FBcast.Reveal}_S(m)$ be executed to reveal $\bar{m}$. If $\bar{m} = \perp$ set $\bar{m}$ to a default value.

IT-SIG.Reveal

1. If $\bar{m}$ was revealed in IT-SIG.Setup output this as S's message.
2. R broadcasts $(m, \{M_{i,j}\}_{1 \leq i \leq n, j \notin INX_i})$.
3. Set the number of votes for m to be the number of i's for which $\bar{K}_{i,j} \neq K_{i,j}$ for some $j \in INX_i$ from the setup.
4. For all i's not counted in the previous step, execute $\mathsf{FBcast.Reveal}_{V_i}(K_{i,j})$ for $j \notin INX_i$. If $\mathsf{MAC}_{K_{i,j}}(m) = M_{i,j}$ for any one of the recovered values then increment the vote by "1".
5. If vote is at least $t+1$ then output m as S's message. Otherwise, output $\perp$.

Figure 3: Information Theoretic SIG

Analysis

The following assumes an honest majority in committee V and that at most one of R and S is corrupted.

- Corrupt S: We need to show that at the end of IT-SIG.Setup, R holds a value m' that can pass verification in IT-SIG.Reveal. We set m' to the value m received from S except if a value $\bar{m}$ is revealed during Step 4 of IT-SIG.Setup in which case we set m' to $\bar{m}$. We first consider the case where R did not ask for the message to be revealed in step 4 and show that m will have at least $t+1$ votes in IT-SIG.Reveal. Indeed, for each honest V_i, either $\bar{K}_{i,j} \neq K_{i,j}$ for some $j \in INX_i$ and thus their vote is counted; otherwise, it holds that $\mathsf{MAC}_{K_{i,j}}(m) = M_{i,j}$ for all $j \in INX_i$. Thus, with overwhelming probability, there exists a $j \notin INX_i$ such that $\mathsf{MAC}_{K_{i,j}}(m) = M_{i,j}$, and a vote for i will be counted. This guarantees at least $t+1$ votes for the value m.

- Corrupt R: In this case we show that only the m that originated with S will pass verification in IT-SIG.Reveal. If the message associated with S is set to the value derived from $\mathsf{FBcast.Reveal}_S(m)$, it is certainly a message that originated with S. If it is set to the message published by R, then that message must get $t+1$ "votes". Votes can be generated by corrupt V_i publishing incorrect keys in Step 2b of IT-SIG.Setup; however, there are at most t such corrupt V_i. The only other way to generate a vote for an incorrect m is to forge a MAC M, which happens with negligible probability.

- If S and R are honest, then due to the message hiding property of the MAC function (that we assume), no information on m is revealed until IT-SIG.Reveal is executed. Indeed,

DC.Commit (executed by C on input m)	DC.Reveal
Let ShareHolder and ShareVerifier be two n-party committees. 1. Committer C computes a t-secret sharing of m, $(m_1, \dots, m_n)$ for $t \geq (n-1)/2$. 2. For $i = 1, \dots, n$: C executes IT-SIG.Setup on input m_i with $\mathsf{ShareHolder}_i$ as receiver R and the set ShareVerifier acting as the set of verifiers V (same ShareVerifier committee is used in all the invocations).	1. For $i = 1, \dots, n$, set $\bar{m}_i$ to the output of $\mathsf{IT\text{-}SIG.Reveal}_i$. 2. Take all $\bar{m}_i$ that are not $\perp$ and interpolate a polynomial through these points. If the polynomial is of degree t or less output its constant term, otherwise output $\perp$.

Figure 4: Distributed Commitment

the only case where R requests to broadcast m prior to IT-SIG.Reveal is when the keys broadcasted by S do not verify the MACs; this cannot be the case when S and R are both honest.

Homomorphism of IT-SIGs

The homomorphic properties of the MAC construction from Section 3.1, imply similar properties for IT-SIG in Figure 3 when the underlying MAC function is homomorphic. Namely, if m, m' are messages on which the (same) sender S runs IT-SIG.Setup with the same set V of verifiers and with correlated keys (i.e., corresponding keys use the same coefficient a in the scheme from Section 3.1), then an IT-SIG on $m + m'$ can be verified with committee V using the MAC keys held by V for m and for m'. This homomorphic property is used in an essential way when performing additions (also used for multiplications) in an arithmetic circuit as described in Section 3.10. A consequence of the need for correlated keys is that if two messages may need to be added in the future, this fact needs to be known at the time of generating the IT-SIG for both m_1 and m_2. In our application this is always the case as the need for additions is determined by the specific circuit being computed.

3.4 Distributed Commitment (DC)

The FB protocol does not offer any guarantees in the case when the dealer is faulty. Thus, we need to create a new primitive with slight better assurances. We present the *distributed commitment protocol* DC consisting of two phases, DC.Commit and DC.Reveal. In DC.Commit, a committer C commits to a value m that may later be revealed in DC.Reveal. More precisely, if C is honest, the revealed value is m, and m is hidden until it is revealed (which is exactly as in the case of FB). However, if C is corrupt, the execution of DC.Commit determines a single value m such that the output of DC.Reveal is guaranteed to be either $\perp$ or m (where m itself can be $\perp$). In other words, C can choose to prevent reconstruction, but if it allows for it to happen then it can only be to a value it is committed to at the end of DC.Commit. Reconstruction is public, namely, there will be public agreement on the output of DC.Reveal.

Protocol DC uses the scheme from Figure 3 in an essential way. In particular, in Step 3 of DC.Commit, for each m_i, C executes IT-SIG.Setup(m_i) with $\mathsf{ShareHolder}_i$ acting as the receiver and with ShareVerifier as the set V of verifiers. The n executions (one for each m_i) are performed in parallel using the same set ShareVerifier in all these executions.

Observe that DC is a stronger primitive than FBcast. In FBcast, when the sender is corrupt, there is no guarantee about the reconstructed value (if any). In DC, at the end of DC.Commit, a single value m is determined so that in DC.Reveal, C only gets the choice between outputting m

or $\perp$. This is similar to a regular commitment in the computational setting where the committer is bound to the value but has the option not to reveal it.

Analysis

We show that at the end of DC.Commit a value m (or $\perp$) is determined, and during DC.Reveal, if C is honest m will be revealed, and if C is corrupt, either m or $\perp$ will be revealed.

In DC.Commit, C executes IT-SIG.Setup with at least $t+1$ honest parties acting as receivers R. For these honest parties, due to the properties of IT-SIG.Setup, it is guaranteed that the value they hold will be accepted in IT-SIG.Reveal. We claim that at the end of DC.Commit, a single value m is committed to, such that the output in DC.Reveal is either m or $\perp$ (where m itself can be $\perp$). To show this, we define m as the constant term of a polynomial of degree at most t interpolated through the set of shares held by the honest parties (this value might be $\perp$ if the set of points does not interpolate to such polynomial). We now show that this value m is the one to be output in DC.Reveal. When C is honest then only shares that were created by C are accepted and thus the polynomial will interpolate properly during DC.Reveal. If C is faulty we know that at least the shares of the honest parties will be included in the set of shares being interpolated and this is a set of at least $t+1$ shares. Thus, the message which is opened can only be m or $\perp$ if there is not a polynomial of degree at most t.
We denote by $\mathsf{DC}_P(m)$ the output of the execution of DC.Commit by party P on message m.

Homomorphism of DC.

Due to the homomorphic properties of the IT-SIG and FBcast, we have that for any two values m and m' committed by the same honest party P, it holds that $\mathsf{DC}_P(m)+\mathsf{DC}_P(m') = \mathsf{DC}_P(m+m')$. The same considerations for ensuring the homomorphism of IT-SIG described in Section 3.3 hold here too (i.e., the DC operations need to be performed by the same committer using correlated keys). In particular, if this property may be required in the future for two messages m, m', then this fact needs to be known at the time of running DC.Commit on these values (fortunately, for our application this requirement does hold). The question might be raised if we know that m and m' will be added why compute individual DC.Commit for both rather than the sum. The answer is that we will in fact need to utilize all three values.

3.5 Duplicate Commitments

In our protocols, we often need to use a committed value multiple times, thus requiring the decommitting parties to act in more than one round, a violation of the YOSO model. One possible solution is for the committer C to commit twice (or more) onto different committees to the same value and provide a proof of equality for the committed values; yet this proof of equality will "waste" the sharing, which is what we need to prevent. Instead, our approach is based on the observation that d duplicates can be achieved by having the parties in ShareHolder reshare the shares m_i received from C during DC (see Figure 4) with d different committees. Proofs of equality of committed values are not needed; it suffices that honest shareholders duplicate their shares correctly to guarantee that all duplicates commit to the same value. Here we are using in an essential way the fact that it is the shareholders that reshare their shares rather than C, and that we can rely on a majority of honest shareholders.

We define protocol DupDC that allows for the duplication of a DC-committed value m. Let d be the number of duplicates needed. In a first committing phase, DupDC.Commit, committer C runs DC.Commit with a committee ShareHolder, sharing its input m so that $\mathsf{ShareHolder}_i$ receives a share m_i on which IT-SIG.Setup of Figure 3 is performed. To generate d duplicates, for each

$i, 1 \leq i \leq n$, C runs d copies of $\mathsf{IT\text{-}SIG.Setup}$ on m_i, each copy with an independent set of MAC keys and a dedicated $\mathsf{ShareVerifier}$ committee. The d copies are verified by $\mathsf{ShareHolder}_i$, acting as receiver R, as specified by $\mathsf{IT\text{-}SIG.Setup}$. Finally, at the end of $\mathsf{DupDC.Commit}$, $\mathsf{ShareHolder}_i$ executes $\mathsf{FBcast.Share}$ for all the values that it holds, and executes d independent sharings of m_i into d separate $\mathsf{ShareHolder}$ committees.

The $\mathsf{DupDC.Reveal}$ phase follows $\mathsf{DC.Reveal}$ where the opening of m_i is implemented via share reconstruction by one of the d $\mathsf{ShareHolder}$ committees to which m_i was shared. Additional information that needs to be broadcast and verified as specified by $\mathsf{IT\text{-}SIG.Reveal}$ is performed via $\mathsf{FBcast.Reveal}$ by the FBcast committees created by $\mathsf{ShareHolder}_i$ during $\mathsf{DupDC.Commit}$.

It is straightforward to check that if the original committer C was honest, all duplicated values are correct DC commitments and they will open to the same committed value during $\mathsf{DupDC.Reveal}$. If C is dishonest, but $\mathsf{ShareHolder}_i$ is honest, and verification against a $\mathsf{ShareVerifier}$ committee fails during the $\mathsf{IT\text{-}SIG.Setup}$ actions, then the committed value is set to the one that is $\mathsf{FBcast.Reveal}$ as part of Step 4 in $\mathsf{IT\text{-}SIG.Setup}$. Otherwise, the value m_i can be reconstructed correctly by any of the d sharings of m_i shared by $\mathsf{ShareHolder}_i$. Since there is a majority ($t+1$ or more) of honest shareholders in each of the d $\mathsf{ShareHolder}$ committees, it is guaranteed that only the committed value or $\perp$ will be reconstructed in each of the d copies.

Analysis.

It follows from the properties of the DC and FBcast protocols. We note that C still has the option of not opening any subset of these duplicate commitments, but all those that will be open will be open to the same value.

3.6 Verifiable Secret Sharing Scheme

The distributed commitment DC functionality ensures that the committer, even a corrupt one, cannot change a value committed at the end of $\mathsf{DC.Commit}$; however, a corrupt committer can prevent reconstruction of the committed value during $\mathsf{DC.Reveal}$. In our applications, we need a commitment scheme with the property that if the commitment phase is successful then reconstruction of the committed value is guaranteed. We achieve this via *Verifiable Secret Sharing* (VSS), a protocol where a dealer secret shares a value s during a $\mathsf{VSS.Share}$ phase so that s is guaranteed to be reconstructed during $\mathsf{VSS.Reveal}$ from any subset of shareholders that includes $t+1$ honest ones. This is the case even for a corrupt dealer, as long it is not disqualified during $\mathsf{VSS.Share}$. Technically, VSS is similar to DC where the dealer proves all the shares to be on a polynomial of degree at most t.

Protocol $\mathsf{VSS.Share}$ proceeds as follows.

1. The dealer D chooses a random polynomial $f(x)$, s.t. $f(0) = s$ and an additional random polynomial $r(x)$, both of degree t. Let the coefficients of $f(x)$ and $r(x)$ be, respectively, f_j, r_j for $0 \leq j \leq t$.

2. Given a set $\mathsf{ShareHolder} = \{P_1, \ldots, P_n\}$, D computes $s_i = f(i), \rho_i = r(i)$ for $1 \leq i \leq n$ and transfers these values to P_i.

3. In the same step as above, D $\mathsf{DupDC.Commits}$ (a minimum of two duplicates; the exact number of duplicates is application-dependent) to all the values f_i, r_i. Due to the homomorphic properties of DC, this results in implicit $\mathsf{DC}_D(s_i)$ and $\mathsf{DC}_D(\rho_i)$

4. P_i uses $\mathsf{DupDC.Commit}$ to commit to s_i (a minimum of two duplicates; the exact number of duplicates is application-dependent) and $\mathsf{DC.Commit}$ to commit to ρ_i. Thus, creating

$\mathsf{DC}_{P_i}(s_i)$ and $\mathsf{DC}_{P_i}(\rho_i)$. It shares the ρ_i to one of the committees to which it duplicates the s_i.

5. Receive r from $\mathcal{F}_{\mathsf{UPBeacon}}$
6. To prove that P_i shared the same value as D, the values $\mathsf{DC}_D(s_i+r\cdot\rho_i)$ and $\mathsf{DC}_{P_i}(s_i+r\cdot\rho_i)$ are reconstructed using $\mathsf{DC.Reveal}$.
7. For any i where the reconstruction $\mathsf{DC}_D(s_i + r \cdot \rho_i)$ and $\mathsf{DC}_{P_i}(s_i + r \cdot \rho_i)$ return $\perp$ or the values are not equal execute $\mathsf{DC.Reveal}$ of D's sharing of s_i. If it returns $\perp$ disqualify the dealer.

Protocol $\mathsf{VSS.Reveal}$ proceeds as follows.

1. Execute $\mathsf{DC.Reveal}$ for all s_i shared by P_i
2. Interpolate a polynomial using all these share and output the constant term.

3.6.1 Analysis.

For the construction to be a VSS we need to ensure that the points shared using DC by P_i are in fact the points on the polynomial shared by D. This is achieved by testing equality of the sharings of P_i against the sharings of D. This is enabled by revealing a random blinding of the shares and testing for equality. It follows using a standard argument that if $s_i \neq s_i'$ then there is at most a single challenge r that will make the proof pass, showing that there is security except with probability $|\mathbb{F}|^{-1}$. Furthermore, the adversary can compute the r' that would make the proof pass before the beacon samples r, so it is enough that r cannot be guessed with non-negligible probability. Namely, then the proof passes then r' was a correct guess at the r which the beacon returned. We therefore later only have to implement $\mathcal{F}_{\mathsf{UPBeacon}}$ to be unpredictable.

Homomorphism of VSS.

VSS inherits the homomorphic properties of DC, Importantly, in the case of VSS, these properties hold even if the VSS was performed by two different dealers as long as it was done into *the same set of shareholders.* Namely, for two secrets m_1 and m_2, and two dealers D_1 and D_2, we have $\mathsf{VSS}_{D_1}(m_1) + \mathsf{VSS}_{D_2}(m_2) = \mathsf{VSS}(m_1 + m_2)$. Note that the right-hand side VSS is not associated to a specific dealer as it combines sharings of D_1 and D_2. The reason the homomorphism holds across dealers is due to the homomorphic properties of $\mathsf{DC}_{P_i}(\cdot)$ (that only hold for same committer) and the fact that the same P_i's act in both VSS dealings as shareholders.

3.7 Duplicate VSS

As in the case of DC, we also need duplicates of VSS values, e.g., when creating values on an arithmetic circuit that need to go into multiple gates. Recall that a VSS is a sharing of a value s where each share s_i of the sharing is shared as $\mathsf{DC}_{P_i}(s_i)$. It is easy to see that duplicating the $\mathsf{DC}_{P_i}(s_i)$ commitments results in duplicate VSSs.

3.8 Augmented VSS

In our application, particularly for the multiplication protocol, we need an *Augmented VSS (*AugVSS*)*, where not only the secret given as input is shared with VSS but also the shares resulting from $\mathsf{VSS}(s)$ are shared with VSS.

AugVSS is achieved via the following computation. The dealer D holding a value s defines a polynomial $f(x) = a_t x^t + ... + a_1 x + a_0$ where $a_0 = s$. It carries out $\mathsf{VSS}(a_i)$ for $0 \leq i \leq t$. Denote the polynomial sharing the a_i coefficient by $f_{a_i}(x)$. Then the parties carry out a computation to deliver to each party its share on the polynomial $f(x)$ by computing for party P_j the value $\Sigma_{i=0}^{t} j^i f_{a_i}(0)$. This delivers the desired properties. There is both a VSS of the value s and a VSS of all the shares of the parties on the polynomial $f(x)$. These VSS's are created via a computation of the linear combination of the VSSs of the coefficients of the polynomial $f(x)$. It is clear that AugVSS is also additively homomorphic.

3.9 Proof of Local Multiplication (PLM)

The following protocol assumes a duplicate VSS sharings for values a, b, c with the same two sets of committees, PLM Verification 1 and PLM Verification 2, for brevity, C and C'. There is no need to assume any knowledge on how these VSS's were created, the only assumption that is needed is that party P that is proving that $c = a \cdot b$ knows all three values a, b, c.

1. P chooses a random value b' and executes duplicate $\mathsf{VSS}_P(b')$ onto both committees C and C' and $\mathsf{VSS}_P(a \cdot b')$ with shareholder set C'.
2. Receive random e from $\mathcal{F}_{\mathsf{UPBeacon}}$;
3. Committee C reconstructs using $\mathsf{VSS.Reveal}$ the value $r = e \cdot b + b'$;
4. Committee C' reconstructs using $\mathsf{VSS.Reveal}$ the value $d = r \cdot a - e \cdot c - a \cdot b'$
5. Accept the proof if $d = 0$. Otherwise, determine that the proof has failed.

It follows using a standard argument that if if $c \neq ab$ then $d \neq 0$ except with probability $|\mathbb{F}|^{-1}$. In particular, there is a single e which will let the proof pass. Hence it is enough that e cannot be guessed with non-negligible probability. The rest of the argument for the correctness of the proof follows from the properties of the VSS.

3.10 YOSO MPC

Using the tools developed up to now we can show how to do secure function evaluation (or MPC) in the YOSO model. That is, we are given an arithmetic circuit $\mathcal{C}$, with m secret inputs provided by m parties (roles), and we show how to privately compute the circuit on the inputs, in the YOSO model.

Let $\mathcal{C}$ be a given arithmetic circuit with m inputs $x_1, \ldots, x_m$ and gates $g_1, \ldots, g_\ell$. In the YOSO computation of $\mathcal{C}$ we maintain the following invariant. For each gate g_i with input values v_{i1}, v_{i2}, there is a gate input committee C_i that contains an AugVSS sharing of v_{i1} and an AugVSS sharing of v_{i2}. For each gate g_i, there will be a gate output committee that will hold the AugVSS sharing of v_{i3} (where v_{i3} represents the output of the computation of the gate on the two inputs). The gate output committee will create a collection of d duplicates of the AugVSS of v_{13}, where d is the number of gates to which v_{13} enters as an input.

The top level AugVSS sharing held by the input committee will be backed-up by lower level sharings committees for each of the sharings, i.e. VSS, DC, FB and so on. Assume that committee C holds the DC of the AugVSS of a, then that same committees will hold the DC of AugVSS of b, etc.

The multiplication operation require closer attention to the details of the committees. The gate input committee of a multiplication gate will duplicate the inputs onto three committees, the computing committee and two verification committees. Other sharings required in the computation will be shared onto these committees as well. Details appear below.

Addition Addition simply uses the homomorphic property of AugVSS.

Multiplication

1. The input committee duplicates the sharing of a and b to the two PLM verification committees (see Section 3.9).
2. In the same step as above, each P_i from the input committee that hold shares a_i and b_i of a and b, respectively, performs $\mathsf{AugVSS}(\gamma_i)$ for $\gamma_i = a_i \cdot b_i$ onto the computing committee and the two PLM verification committees and performs the first sharing step of the PLM protocol, to prove that $\gamma_i = a_i \cdot b_i$. Note that $\mathsf{AugVSS}(\gamma_i)$ results in $\mathsf{VSS}_{P_i}(\gamma_i)$ and sharings of all the shares of this polynomial, denote those by $\mathsf{VSS}_{P_i}(\gamma_{i,j})$.
3. For any i for which the AugVSS or the PLM procedures fail, the committee that holds a_i and b_i uses $\mathsf{VSS.Reveal}$ to publicly reconstruct these values. Going forward, when the protocol uses the value γ_i, its value is set to the product $a_i \cdot b_i$ of the reconstructed values.
4. The computing committee holds the AugVSS of all γ_i's. Let $\mathsf{VSS}(c) = \Sigma_{i=1}^{2t+1} \lambda_i \mathsf{VSS}_{P_i}(\gamma_i)$ and let $\mathsf{VSS}(c_i) = \Sigma_{j=1}^{2t+1} \lambda_j \mathsf{VSS}_{P_j}(\gamma_{j,i})$ for the appropriate Lagrange coefficients λ_i (see below).
5. The computing committee transfers the value to the output committee of the gate. In practice, these two committees can be the same as the computing committee never speaks.

The multiplication protocol follows the design of [16]. The correctness of the AugVSS sharing of the multiplication $c = a \cdot b$ follows from: (i) the verified correctness of $\mathsf{VSS}(\gamma_i)$ and PLM; (ii) the public availability of γ_i values for those i where verification failed (these values are available because in AugVSS, not only the secret is shared but also its shares). (iii) the existence of Lagrange coefficients λ_i for which $c = a \cdot b = \Sigma_{i=1}^{2t+1} \lambda_i (a_i \cdot b_i)$. These Lagrange coefficients λ_i are defined as follows. Let $f_{\gamma_i}(x)$ be the degree-t polynomial shared by P_i in the input committee with $f_{\gamma_i}(0) = \gamma_i$. The coefficients λ_i are those that interpolate the polynomials $f_{\gamma_i}(x)$ into a degree-t polynomial $F(x)$ of degree t such that $F(0) = c$, namely, $F(x) = \Sigma_{i=1}^{2t+1} \lambda_i f_{\gamma_i}(x)$. The share s_j computed by party P_j in the computing committee satisfies $s_j = \Sigma_{i=1}^{2t+1} \lambda_i s_{ij} = \Sigma_{i=1}^{2t+1} \lambda_i f_{\gamma_i}(j) = F(j)$, hence a correct sharing of c with corresponding polynomial $F(x)$.

Security argument

Security follows using standard arguments. In particular, the simulator proceeds as follows. Use the AugVSS's to reconstruct the inputs of the corrupted parties. Input these to $\mathcal{F}_{\mathrm{MPC}}^{F}$ where F denotes the function computed by $\mathcal{C}$. Use dummy inputs of the honest parties in the simulation. Run the simulated protocol honestly with these dummy inputs. When processing an output gate, learn the correct output from $\mathcal{F}_{\mathrm{MPC}}^{F}$. Then from the t simulated shares of the corrupted parties and the output acting as share $t+1$ compute the matching shares of the honest parties. Then send these in the simulation. Furthermore, the simulation of the IT-MAC and IT-SIG are straightforward.

To prove adaptive security the simulator will for each committee C_j start out with a set C_j of size t playing the role of the corrupted parties and will simulate as in the static case with C_j being corrupted. If party P_i^j in C_j becomes corrupted and $\mathsf{P}_i^j \notin C_j$ then the simulator will swap P_i^j with an honest party in C_j and then patch the view of the party to get a simulated state of P_i^j. If P_i^j holds a share on a random, unknown polynomial of degree at most t, the share will just be simulated by a random field element. If P_i^j holds a share on a random, known polynomial of degree at most t, as is the case for a reconstructed output of the computation, then the simulator will know the output and will with the additional t simulated shares of C_j have $t+1$ simulated shares. From these it can compute the corresponding simulated share of P_i^j and claim this as the state of P_i^j. In general the adaptive patching follows using standard techniques from MPC and can be done along the lines in [14] where the patching technique is used to prove [2] adaptive secure in the UC model.

3.11 Implementing $\mathcal{F}_{\mathsf{UPBeacon}}$

We now return to implementing $\mathcal{F}_{\mathsf{UPBeacon}}$.

By inspection we did not use $\mathcal{F}_{\mathsf{UPBeacon}}$ to implement our commitments. We only used it in VSS and proofs of local multiplication. We can therefore use the commitment to implement $\mathcal{F}_{\mathsf{UPBeacon}}$ without any circularity problems. We discuss how to flip a an L-bit string e. If we flip a field element then $L = \log_2(|\mathbb{F}|)$. Select k coin roles $\mathsf{C}_1, \ldots, \mathsf{C}_k$. Each of them commit to a uniformly random L-bit string e_i. Then in sequence they open $e_1, \ldots, e_k$. Then they sum the values which were opened correctly. Notice that corrupt parties can choose to open or not open their commitment which biases the coin. We consider the probability of guessing e. With probability $1/2$ the role C_k is honest and then the coin is uniform. In this case the guessing probability is 2^{-L}. To the total guessing probability this contributes with 2^{-L-1}. With probability $1/2^2$ the role C_{k-1} is honest and C_k is corrupt. In this case C_k can remove one bit of min entropy and the guessing probability is 2^{-L+1}. To the total guessing probability this contributes with 2^{-L-1}. Summing over all L possible positions of the last honest C_i we get a total guessing probability of $L2^{-L-1}$. On top of that comes the case where all C_i are corrupt where the guessing probability of 1. This gives a total of $L2^{-L-1} + 2^{-L} = (1/2L+1)2^{-L}$. So, if we pick a finite field $\mathbb{F}$ of sufficiently super-polynomial size, for instance exponential, we can flip unguessable field elements.

Then simply note that in all cases where we used $\mathcal{F}_{\mathsf{UPBeacon}}$ we only needed to flip unguessable field elements.

3.12 Unbiased Randomness YOSO Style

A number of works have explored how to generate random beacons in a blockchain setting [6, 7]. Using our MPC protocol we can proceed to develop the first completely uniform beacon, $\mathcal{F}_{\mathsf{Beacon}}$, on the blockchain implemented by a YOSO protocol. In the above section we proved how to get $\mathcal{F}_{\mathrm{MPC}}^{F}$ from $\mathcal{F}_{\mathsf{BC}}, \mathcal{F}_{\mathsf{SPP}}$ under 49% Byzantine corruptions. Note the we can in turn implement $\mathcal{F}_{\mathsf{Beacon}}$ given $\mathcal{F}_{\mathrm{MPC}}^{F}$. Have k computation roles act as input roles and let each of them input a uniformly random field elements. Sum them and output the sum to all parties. Except with negligible probability there will be an honest input role and the output is uniform. This show how to get statistically unbiased coin-flip from $\mathcal{F}_{\mathsf{BC}}$ and $\mathcal{F}_{\mathsf{SPP}}$.

4 The Computational $t < \frac{n}{2}$ MPC Protocol

In this section we build a computationally secure YOSO MPC protocol. This is the first YOSO protocol to achieve guaranteed output delivery and require only an honest majority in each committee; Benhamouda *et al.* [3] require an honest supermajority, while Choudhuri *et al.* [10] only achieve security with abort.

Our protocol is heavily based on the CDN protocol [11]. CDN uses threshold linearly homomorphic encryption, where n parties have shares sk_i of a decryption key sk. The input values to the computation are encrypted under the corresponding encryption key pk; the linear operations are performed using the homomorphism, and the multiplications are done using Beaver triples (encryptions of random a and b, and of their product ab).

CDN is naturally almost entirely YOSO. The only technical challenge in making it entirely YOSO is passing the shared secret key, which is used for the decryption of masked intermediate values during Beaver triple use and for the recovery of the output, from committee to committee. This can be achieved by using *key rerandomizable threshold encryption*, a linearly homomorphic version of which can be instantiated as a variant of Paillier encryption with a Shamir shared secret key. Passing (rerandomized shares of) the secret key from committee to committee is done in a way similar to the YOSO handover protocol of Benhamouda *et al.* [3].

CDN does require some setup — shares of the decryption key sk must be distributed to the first committee by a trusted third party, or generated in a secure distributed way. However, this setup is fairly minimal. In contrast, the Beaver triples can be generated on-the-fly, by two consecutive committees C_A and C_B. These committees are different from the committees who hold shares of the decryption key in two ways. Committees C holding decryption key shares should (a) have an honest majority (to balance our need for guaranteed output delivery with our need for privacy against the adversary), and (b) be able to receive private messages from the previous committee. In contrast, the committees C_A and C_B generating our Beaver triples (a) can tolerate a dishonest majority, requiring only one honest party, and (b) do not need to be able to receive any private messages. The fact that the Beaver triple committees can tolerate a dishonest majority means that they can be smaller than our key-holding committees; the fact that they do not need to receive any private messages means that the role-assignment mechanism for selecting them can be simpler. We do not elaborate on either of these observations, especially since the latter is a property of the underlying role-assignment functionality, which we do not study in this work. However, these two observations together lead us to believe that the overhead for generating Beaver triples for our CDN protocol is minimal.

In Section 4.1we introduce the building blocks our protocol needs. In Section 4.2, we give a detailed description of the protocol, and in Section 4.3, we show the security of the protocol.

4.1 Building Blocks

The building blocks we leverage are non-interactive zero-knowledge arguments of knowledge (Section 4.1.1), and a linearly homomorphic key rerandomizable threshold encryption scheme (Section 4.1.2).

4.1.1 Non-Interactive Zero-Knowledge Arguments of Knowledge

Syntax A non-interactive zero-knowledge argument of knowledge ($\mathsf{NIZKAoK}$) scheme has the following algorithms, as described by Groth and Maller [18]:

$\mathsf{Setup}(1^\kappa, \mathcal{R}) \to (crs, td)$: An algorithm that, given the security parameter, sets up the global common reference string crs and the trapdoor td for the $\mathsf{NIZKAoK}$ system.

$\mathsf{P}(crs, \phi, w) \to \pi$: An algorithm that, given the common reference string crs for a relation $\mathcal{R}$, a statement ϕ and a witness w, returns a proof π that $(\phi, w) \in \mathcal{R}$.

$\mathsf{V}(crs, \phi, \pi) \to$ `accept/reject`: An algorithm that, given the common reference string crs for a relation $\mathcal{R}$, a statement ϕ and a proof π, checks whether π proves the existence of a witness w such that $(\phi, w) \in \mathcal{R}$.

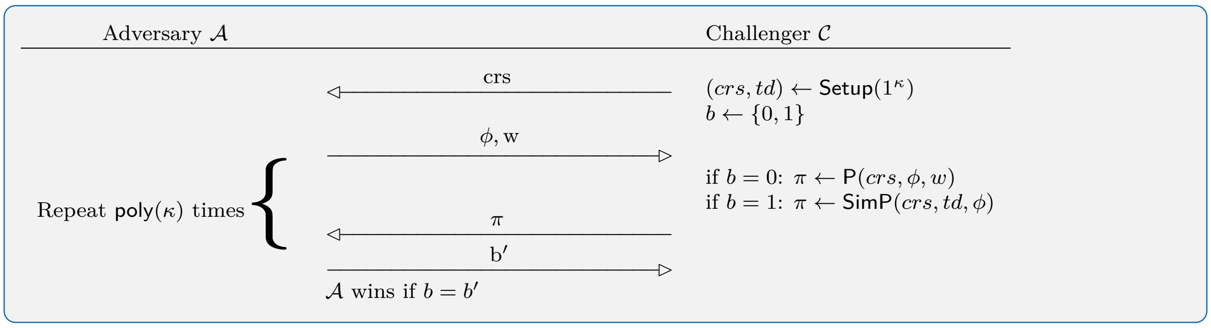

Adversary $\mathcal{A}$
Challenger $\mathcal{C}$
crs
$(crs, td) \leftarrow \mathsf{Setup}(1^\kappa)$
$b \leftarrow \{0,1\}$
ϕ, w
Repeat $\mathsf{poly}(\kappa)$ times
if $b = 0$: $\pi \leftarrow \mathsf{P}(crs, \phi, w)$
if $b = 1$: $\pi \leftarrow \mathsf{SimP}(crs, td, \phi)$
π
b′
$\mathcal{A}$ wins if $b = b'$

Figure 5: Security game for the zero knowledge property of the NIZKAoK.

$\mathsf{SimP}(crs, td, \phi) \rightarrow \pi$: An algorithm that, given the common reference string crs for a relation $\mathcal{R}$, the trapdoor td and a statement ϕ, simulates a proof of the existence of a witness w such that $(\phi, w) \in \mathcal{R}$.

Security Properties Of course, a NIZKAoK scheme must be *correct* (that is, verification using an honestly produced proof must return `accept`). The important security properties of a NIZKAoK scheme are *zero knowledge*, *knowledge soundness*, and *simulation extractability*, described below.

Definition 1 (Zero Knowledge for NIZKAoK). *Informally, a* NIZKAoK *scheme has* zero knowledge *if a proof does not leak any more information than the truth of the statement.*

More formally, let $\kappa \in \mathbb{N}$ be the security parameter, and let $\mathsf{NIZKAoK} = (\mathsf{Setup}, \mathsf{P}, \mathsf{V}, \mathsf{SimP})$ *be a* NIZKAoK *scheme. Consider the game between a probabilistic polynomial-time adversary $\mathcal{A}$ and a challenger $\mathcal{C}$ described in Figure 5.*

NIZKAoK *has* zero knowledge *if for any sufficiently large security parameter κ, for any probabilistic polynomial-time adversary $\mathcal{A}$, there exists a negligible function negl in the security parameter κ such that the probability that $\mathcal{A}$ wins the game is less than $\frac{1}{2} + negl(\kappa)$.*

Informally, *knowledge soundness* is the property that guarantees that it is always possible to extract a valid witness from a proof that verifies. *Simulation extractability* is a stronger version of knowledge soundness, where it is always possible to extract a valid witness from a proof that verifies even if the adversary has access to a simulation oracle. This is a flavor of non-malleability; an adversary should not even be able to modify a simulated proof in order to forge a proof.

Definition 2 (Simulation Extractability for NIZKAoK). *Informally, a* NIZKAoK *scheme has* simulation extractability *if it is always possible to extract a valid witness from a proof that verifies.*

More formally, let $\kappa \in \mathbb{N}$ be the security parameter, and let $\mathsf{NIZKAoK} = (\mathsf{Setup}, \mathsf{P}, \mathsf{V}, \mathsf{SimP})$ *be a* NIZKAoK *scheme. Consider the game between a probabilistic polynomial-time adversary $\mathcal{A}$ and a challenger $\mathcal{C}$ described in Figure 6, where $\tau_{\mathcal{A}}$ denotes the adversary's inputs and outputs, including its randomness:*

NIZKAoK *has* simulation extractability *if for any sufficiently large security parameter κ, for any probabilistic polynomial-time adversary $\mathcal{A}$, there exists an extraction algorithm* $\mathsf{Extract}_{\mathcal{A}}$ *and a negligible function negl in the security parameter κ such that the probability that $\mathcal{A}$ wins the game is less than $negl(\kappa)$.*

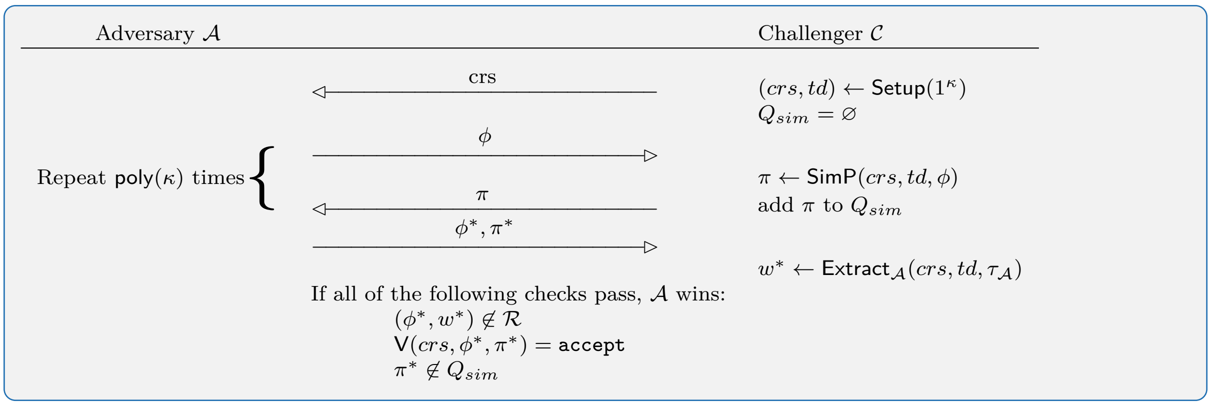

Adversary $\mathcal{A}$
Challenger $\mathcal{C}$
crs
$(crs, td) \leftarrow \mathsf{Setup}(1^\kappa)$
$Q_{sim} = \varnothing$
Repeat $\mathsf{poly}(\kappa)$ times
ϕ
π
$\pi \leftarrow \mathsf{SimP}(crs, td, \phi)$
add π to Q_{sim}
ϕ^*, π^*
$w^* \leftarrow \mathsf{Extract}_\mathcal{A}(crs, td, \tau_\mathcal{A})$
If all of the following checks pass, $\mathcal{A}$ wins:
$(\phi^*, w^*) \notin \mathcal{R}$
$\mathsf{V}(crs, \phi^*, \pi^*) = \mathsf{accept}$
$\pi^* \notin Q_{sim}$

Figure 6: Security game for the simulation extractability property of the NIZKAoK.

4.1.2 Linearly Homomorphic Key Rerandomizable Threshold Encryption

Syntax A linearly homomorphic (over ring $\mathbb{R}$) key rerandomizable threshold encryption scheme has the following algorithms:

$\mathsf{TKGen}(1^\kappa) \to (tpk, tsk_1, \ldots, tsk_n)$: An algorithm that, given the security parameter, sets up the public key tpk and the shares $tsk_1, \ldots, tsk_n$ of the secret key.

$\mathsf{TEnc}(tpk, m; \rho) \to \beta$: An algorithm that, given the public key, a message $m \in \mathbb{R}$ and randomness ρ, outputs an encryption β of m.

$\mathsf{TPDec}(tpk, tsk_i, \beta) \to d_i$: An algorithm that, given the public key, a share tsk_i of the secret key and a ciphertext β, outputs a partial decryption d_i.

$\mathsf{TDec}(tpk, \{d_i\}_{i \in S, |S| \geq t}) \to m$: An algorithm that, given sufficiently many partial decryptions, returns the decrypted message m.

$\mathsf{TEval}(tpk, \beta_1, \ldots, \beta_k, \lambda_1, \ldots, \lambda_k) \to \beta$: A deterministic algorithm that, given the public key, ciphertexts $\beta_1, \ldots, \beta_k$ corresponding to messages $m_1, \ldots, m_k \in \mathbb{R}^k$ and coefficients $\lambda_1, \ldots, \lambda_k \in \mathbb{R}^k$, outputs a ciphertext β that encrypts $\sum_{i=1}^{k} \lambda_i m_i \in \mathbb{R}$.

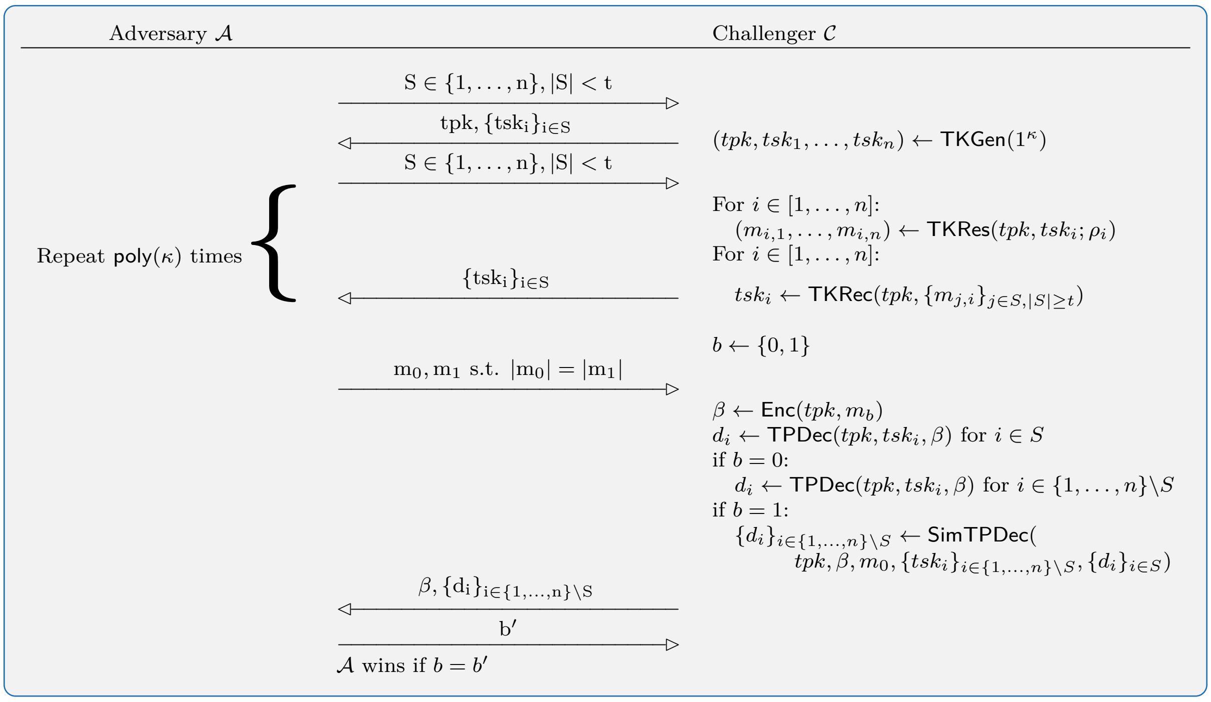
$\mathsf{TKRes}(tpk, tsk_i; \rho_i) \to (m_{i,1}, \ldots, m_{i,n})$: An algorithm that, given the public key and a share of a secret key, produces n messages to help with the rerandomization of the secret key sharing.

$\mathsf{TKRec}(tpk, \{m_{j,i}\}_{j \in S, |S| \geq t}) \to tsk_i$: An algorithm that, given sufficiently many messages for the rerandomization of the secret key sharing, outputs a share of the secret key.

$\mathsf{SimTPDec}(tpk, \beta, m, \{tsk_i\}_{i \in \{1,\ldots,n\} \setminus S}, \{d_i\}_{i \in S, |S| \geq t}) \to \{d_i\}_{i \in \{1,\ldots,n\} \setminus S}$: A simulation algorithm that, given a ciphertext, a target message, and partial decryptions belonging to corrupt parties, simulates partial decryptions belonging to honest parties that cause TDec to output the desired message.

Security Properties Of course, a linearly homomorphic key rerandomizable threshold encryption scheme must be *correct* in several ways:

- Decryption on honestly produced ciphertext and keys must return the appropriate message.


Adversary $\mathcal{A}$
Challenger $\mathcal{C}$
$\mathrm{S} \in \{1, \ldots, \mathrm{n}\}, |\mathrm{S}| < \mathrm{t}$
$\mathrm{tpk}, \{\mathrm{tsk_i}\}_{\mathrm{i \in S}}$
$\mathrm{S} \in \{1, \ldots, \mathrm{n}\}, |\mathrm{S}| < \mathrm{t}$
Repeat $\mathsf{poly}(\kappa)$ times
$\{\mathrm{tsk_i}\}_{\mathrm{i \in S}}$
$\mathrm{m_0}, \mathrm{m_1}$ s.t. $|\mathrm{m_0}| = |\mathrm{m_1}|$
$\beta, \{\mathrm{d_i}\}_{\mathrm{i \in \{1,\ldots,n\} \setminus S}}$
b'
$\mathcal{A}$ wins if $b = b'$

```
$(tpk, tsk_1, \ldots, tsk_n) \leftarrow \mathsf{TKGen}(1^\kappa)$
For $i \in [1, \ldots, n]$:
   $(m_{i,1}, \ldots, m_{i,n}) \leftarrow \mathsf{TKRes}(tpk, tsk_i; \rho_i)$
For $i \in [1, \ldots, n]$:
   $tsk_i \leftarrow \mathsf{TKRec}(tpk, \{m_{j,i}\}_{j \in S, |S| \geq t})$
$b \leftarrow \{0, 1\}$
$\beta \leftarrow \mathsf{Enc}(tpk, m_b)$
$d_i \leftarrow \mathsf{TPDec}(tpk, tsk_i, \beta)$ for $i \in S$
if $b = 0$:
   $d_i \leftarrow \mathsf{TPDec}(tpk, tsk_i, \beta)$ for $i \in \{1, \ldots, n\} \setminus S$
if $b = 1$:
   $\{d_i\}_{i \in \{1,\ldots,n\} \setminus S} \leftarrow \mathsf{SimTPDec}($
         $tpk, \beta, m_0, \{tsk_i\}_{i \in \{1,\ldots,n\} \setminus S}, \{d_i\}_{i \in S})$
```

Figure 7: Security game for the partial decryption simulatability property of the TE.

- Decryption must remain correct after homomorphic evaluation.
- Decryption must remain correct after a rerandomization of the secret key sharing.

These correctness properties are intuitive, and we do not formalize them here. The important security property of a TE scheme is *partial decryption simulatability*, described below. Note that it trivially implies chosen plaintext security.

Definition 3 (Partial Decryption Simulatability for TE)**.** *Informally, a* TE *scheme has* partial decryption simulatability *if for any honestly produced ciphertext, desired message m and fewer than t partial decryptions, the algorithm* $\mathsf{SimTPDec}$ *produces remaining partial decryptions which cause* TDec *to return m.*

More formally, let $\kappa \in \mathbb{N}$ be the security parameter, and let $\mathsf{TE} = (\mathsf{TKGen}, \mathsf{TEnc}, \mathsf{TPDec}, \mathsf{TDec}, \mathsf{TEval}, \mathsf{TKRes}, \mathsf{TKRec}, \mathsf{SimTPDec})$ *be a* TE *scheme. Consider the game between a probabilistic polynomial-time adversary $\mathcal{A}$ and a challenger $\mathcal{C}$ described in Figure 7.*

TE *has* partial decryption simulatability *if for any sufficiently large security parameter κ, for any probabilistic polynomial-time adversary $\mathcal{A}$, there exists a negligible function negl in the security parameter κ such that the probability that $\mathcal{A}$ wins the game is less than $\frac{1}{2} + negl(\kappa)$.*

Instantiation We can instantiate such a linearly homomorphic key rerandomizable threshold encryption scheme by Shamir sharing a Paillier decryption key [12]. Recall that Paillier encryption works modulo n^2 where $n = pq$ and $p = 2p' + 1$, and $q = 2q' + 1$ (for prime p, q, p', q'). The plaintext space is $\mathbb{Z}_n$ and the ciphertext space is $\mathbb{Z}^*_{n^2}$. Encryption of x is given by

$$\beta = \mathsf{Enc}(pk, m) = (n+1)^m r^n \bmod n^2,$$

for $pk = n$ and uniformly random $r \in \mathbb{Z}^*_{n^2}$. Let $n' = p'q'$. The secret key d is a number such that $d \bmod n = 1$ and $d \bmod n' = 0$.[7] The element $n+1$ has order n in $\mathbb{Z}^*_{n^2}$. Any element r^n is in a sub-group of order $\phi(n) = (p'-1)(q'-1)$, and $(r^n)^2$ is in a sub-group of order $n' = p'q'$. Therefore, we can decrypt as

$$\mathsf{Dec}(d, \beta) = \frac{\beta^{2d} \mod n^2 - 1}{2n} = \frac{(n+1)^{2m} \mod n^2 - 1}{2n} = \frac{(2mn+1)-1}{2n} = m.$$

This allows a very simple threshold decryption by interpolation in the exponent. The secret key d is initially secret shared using a Shamir secret sharing modulo nn'. Party i has share i. Partial decryption will basically consist of β^{2d_i} with some subtle extra details described in the cited papers. When resharing, the parties cannot reshare tsk_i modulo nn', since they do not know this modulus (this knowledge would leak the secret key, and the factorization of n). Instead, they will sample a sufficiently large secret sharing of tsk_i over the integers using a polynomial p_i.

Given $t+1$ shares $p_i(j)$ the next party j can compute via Lagrange interpolation its new key share $d'_j = \sum_i \lambda_i p_i(j)$. The party cannot reduce its key share modulo nn', since it still does not know the modulus. This, however, will not matter as d_j is later used to compute $\beta^{2d'_j}$, and β^2 has an order which divides the modulus.

More troublesome are the Lagrange coeficients λ_i, which should also be computed modulo the unknown nn'. It can, however, be seen that the computation of these only involves division by numbers between 2 and $n-2$. So, if we let $\Delta = (n-1)!$, then we can compute $\Delta\lambda_i$ over the integers. This will allow us to interpolate a fresh sharing of Δd. The next decryption will yield $\Delta x \bmod n$ as opposed to x, but Δ is invertible modulo n, so the parties can still retrieve x. After l key refreshings, the parties have to divide out Δ^l. See [12] for the details.

Damgård and Koprowski [13] show how to generate the setup necessary for this scheme, but their protocol is not YOSO. It is an interesting open problem to design a practical YOSO protocol for setting up the threshold decryption material.

4.2 Protocol Description

In this section we describe in detail our MPC protocol Π_{CDN}. The roles $\mathsf{R} \in \Pi_{\mathsf{CDN}}.\mathsf{Role}$ (i.e., the parties in the protocol Π_{CDN}) are initially divided into a sequence of committees C of a size n large enough that each of them have honest majority except with negligible probability; we denote the committee of a round l by C_l, and the i-th committee member of C_l by $C_l[i]$.

We describe our protocol in terms of individual operations. Some operations (e.g. Input) are executed by a single role R; others (e.g. Add) are public operations executed by all roles; yet others (e.g. Mult or $\mathsf{MakeBeaver}$) are executed by one or more committees. We subscript each operation with the roles executing it.

Note that several functions could be evaluated given the same setup. However, we focus on a single function evaluation for simplicity.

Let $\mathsf{TE} = (\mathsf{TKGen}, \mathsf{TEnc}, \mathsf{TPDec}, \mathsf{TDec}, \mathsf{TEval}, \mathsf{TKRes}, \mathsf{TKRec})$ be a linearly homomorphic key rerandomizable threshold encryption scheme, let $\mathsf{PKE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ be a public key encryption scheme, and let $\mathsf{NIZKAoK} = (\mathsf{Setup}, \mathsf{P}, \mathsf{V}, \mathsf{SimP})$ be a simulation extractable non-interactive zero-knowledge argument of knowledge. We use $\mathsf{NIZKAoK}$ for the following relations:

[7]Typically, d is chosen such that $d \bmod \phi(n) = 0$; however, in order to get the threshold case to work, we use n' instead of $\phi(n)$ to avoid factors of two which cause complications.

$$\mathcal{R}_{\mathsf{Dec}} = \left\{ \begin{array}{l} \phi = \quad (pk, tpk, \{pk'_j\}_{j\in[n]}, \{\overline{m_j}\}_{j\in[n]}, \{\overline{m'_j}\}_{j\in S}, d, \beta) \\ w = \quad (\mathsf{sk}, tsk, \rho, \{\rho'_j\}_{j\in[n]}) \end{array} \middle| \begin{array}{l} \mathsf{sk} \text{ is the secret key corresp. to } pk \\ \wedge\ m'_j \leftarrow \mathsf{Dec}(\mathsf{sk}, \overline{m'_j}) \text{ for } j \in S \\ \wedge\ tsk \leftarrow \mathsf{TKRec}(tpk, \{m'_j\}_{j\in S}) \\ \wedge\ \{m_j\}_{j\in[n]} \leftarrow \mathsf{TKRes}(tpk, tsk; \rho) \\ \wedge\ \overline{m_j} \leftarrow \mathsf{Enc}(pk, m_j; \rho_j) \text{ for } j \in [n] \\ \wedge\ d \leftarrow \mathsf{TPDec}(tpk, tsk, \beta). \end{array} \right\},$$

$$\mathcal{R}_{\mathsf{Beaver}} = \left\{ \begin{array}{l} \phi = (\overline{a}, \overline{b}, \overline{c}) \\ w = (b, \rho) \end{array} \middle| \begin{array}{l} \overline{b} = \mathsf{TEnc}(tpk, b; \rho) \\ \wedge\ \overline{c} = \mathsf{TEval}(tpk, \overline{a}, b) \end{array} \right\}.$$

We assume that in our encryption scheme PKE, the public key commits to a single decryption key; so, given a ciphertext, an adversary should not be able to come up with an alternative decryption key which can decrypt this ciphertext to an incorrect message.

Protocol Π_{CDN}

Setup(1^κ):
- The ideal functionality $\mathcal{F}_{\mathsf{Gen}}$ initializes a PKI for Π such that each role $\mathsf{R} \in \Pi_{\mathsf{CDN}}.\mathsf{Role}$ is assigned a key pair $(pk_{\mathsf{R}}, \mathsf{sk}_{\mathsf{R}})$ for the PKE scheme.
- Run $(tpk, tsk_1, \ldots, tsk_n) \leftarrow \mathsf{TKGen}(1^\kappa)$. The public key tpk is published and each share tsk_i of the secret key is given to the corresponding committee member $C_0[i]$ of the initial committee C_0.
- Initialize the NIZKs by running $(crs_{\mathsf{Dec}}, td) \leftarrow \mathsf{Setup}(1^\kappa, \mathcal{R}_{\mathsf{Dec}})$, and $(crs_{\mathsf{Beaver}}, td) \leftarrow \mathsf{Setup}(1^\kappa, \mathcal{R}_{\mathsf{Beaver}})$. Publish $crs_{\mathsf{Dec}}, crs_{\mathsf{Beaver}}$.

Input$_{\mathsf{R}}(x)$: The role R samples randomness ρ, computes an encryption of its secret input x as $\overline{x} \leftarrow \mathsf{TEnc}(tpk, x; \rho)$, amd broadcast it by sending the message $(\textsc{Input}, \mathsf{R}, \overline{x})$ to $\mathcal{F}_{\mathsf{BC}}$.

Decrypt$_{C_l}(\beta)$: Each member $C_l[i]$ of the current committee C_l does the following:

- Reconstructs its key share.
  - Let $S \subseteq \{1, \ldots, n\}$ include j if and only if the proof $\pi_{\mathsf{Dec},l-1,j}$ verifies.
  - $C_l[i]$ decrypts key shares received from the $C_{l-1}[j]$, $j \in S$ as $m_{l-1,j,i} \leftarrow \mathsf{Dec}(\mathsf{sk}_{C_l[i]}, \overline{m_{l-1,j,i}})$, and
  - Reconstructs its key share as $tsk_i \leftarrow \mathsf{TKRec}(tpk, \{m_{l-1,j,i}\}_{j\in S})$.
- Reshares its key share.
  - $C_l[i]$ samples randomness $\rho_{l,i}$ and computes a re-sharing of its key share as $(m_{l,i,1}, \ldots, m_{l,i,n}) \leftarrow \mathsf{TKRes}(tpk, tsk_i; \rho_{l,i})$, and
  - For each $j \in [n]$ it samples randomness $\rho_{l,i,j}$ and encrypts each share to the next committee as $\overline{m_{l,i,j}} \leftarrow \mathsf{Enc}(pk_{C_{l+1}[j]}, m_{l,i,j}; \rho_{l,i,j})$.
- Computes the partial decryption $d_i \leftarrow \mathsf{TPDec}(tpk, tsk_i, \beta)$.
- Proves that it did everything correctly.

$$\pi_{\mathsf{Dec},l,i} \leftarrow \mathsf{P}\left(\begin{array}{l} crs_{\mathsf{Dec}}, \\ \phi = (tpk, pk_{C_l[i]}, \{pk_{C_{l+1}[j]}\}_{j\in[n]}, \{\overline{m_{l-1,j,i}}\}_{j\in S}, \{\overline{m_{l,i,j}}\}_{j\in[n]}, d_i, \beta), \\ w = (\mathsf{sk}_{C_l[i]}, tsk_i, \rho_{l,i}, \{\rho_{l,i,j}\}_{j\in[n]}) \end{array} \right)$$

- Broadcast by sending the message $\big(\textsc{Input}, C[i], (d_i, \{\overline{m_{l,i,j}}\}_{j\in[n]}, \pi_{\mathsf{Dec},l,i})\big)$ to $\mathcal{F}_{\mathsf{BC}}$.
- Let $S \subseteq \{1, \ldots, n\}$ include i if and only if the proof $\pi_{\mathsf{Dec},l,i}$ verifies.
- All roles $\mathsf{R} \in \Pi_{\mathsf{CDN}}.\mathsf{Role}$ compute the final decryption $x \leftarrow \mathsf{TDec}(tpk, \{d_i\}_{i\in S}, \beta)$.

Note that if multiple ciphertexts are to be decrypted in round l, the same committee can decrypt them all in parallel.

Add$(\overline{a}, \overline{b})$: All roles $\mathsf{R} \in \Pi_{\mathsf{CDN}}.\mathsf{Role}$ compute $\mathsf{TEval}(tpk, (\overline{a}, \overline{b}), (1, 1))$.

MakeBeaver$_{C_A, C_B}$: Committees C_A and C_B together produce a Beaver triple. (Multiplication then reduces to a linear operations and decryptions, described below.)

- Each member $C_A[i]$ of C_A does the following:
  - Picks a random value a_i,
  - Encrypts it as $\overline{a_i} \leftarrow \mathsf{TEnc}(tpk, a_i)$, and
  - Broadcast by sending the message $(\textsc{Input}, C_A[i], \overline{a_i})$ to $\mathcal{F}_{\mathsf{BC}}$.
- All roles $\mathsf{R} \in \Pi_{\mathsf{CDN}}.\mathsf{Role}$ compute $\overline{a} \leftarrow \mathsf{TEval}(tpk, \{\overline{a_i}\}_{i\in C_l}, (1)^n)$.

- Each member $C_B[j]$ of C_B does the following:
  - Picks a random value b_j.
  - Encrypts it as $\overline{b_j} \leftarrow \mathsf{TEnc}(tpk, b_j; \rho_{b,j})$.
  - Computes $\overline{c_j} \leftarrow \mathsf{TEval}(tpk, \overline{a}, b_j)$.
  - Proves that it did everything correctly.

$$\pi_{\mathsf{Beaver},j} \leftarrow \mathsf{P}\left(\begin{array}{l} crs_{\mathsf{Beaver}}, \\ \phi = (\overline{a}, \overline{b_j}, \overline{c_j}), \\ w = (b_j, \rho_{b,j}) \end{array}\right)$$

  - Broadcast by sending the message $\big(\textsc{Input}, C_B[j], (\overline{b_i}, \overline{c_j}, \pi_{\mathsf{Beaver},j})\big)$ to $\mathcal{F}_{\mathsf{BC}}$.
- Let $S \subseteq \{1, \ldots, n\}$ include j if and only if the proof $\pi_{\mathsf{Beaver},j}$ verifies.
- All roles $\mathsf{R} \in \Pi_{\mathsf{CDN}}.\mathsf{Role}$ compute $\overline{b} \leftarrow \mathsf{TEval}(tpk, \{\overline{b_j}\}_{j \in S}, (1)^{|S|})$, and $\overline{c} \leftarrow \mathsf{TEval}(tpk, \{\overline{c_j}\}_{j \in S}, (1)^{|S|})$.

Note that the entirety of Beaver triple generation can be carried out without the use of tsk, and without the committee members needing to receive any private messages. Additionally, note that Beaver triple generation committees can have a dishonest majority, and can therefore be smaller.

$\mathsf{Mult}_{C_l, C_{l+1}}(\overline{x}, \overline{y}, \overline{a}, \overline{b}, \overline{c})$: Let $\overline{x}$ and $\overline{y}$ be the ciphertexts corresponding to the values being multiplied, and $\overline{a}, \overline{b}, \overline{c}$ be a Beaver triple previously produced (and verified).

- All roles $\mathsf{R} \in \Pi_{\mathsf{CDN}}.\mathsf{Role}$ compute $\overline{\epsilon} \leftarrow \mathsf{TEval}(tpk, (\overline{x}, \overline{a}), (1, 1))$, and $\overline{\delta} \leftarrow \mathsf{TEval}(tpk, (\overline{y}, \overline{b}), (1, 1))$.
- The roles in committee C_l execute $\epsilon = \mathsf{Decrypt}_{C_l}(\overline{\epsilon})$ and (in parallel) $\delta = \mathsf{Decrypt}_{C_l}(\overline{\delta})$.
- All roles $\mathsf{R} \in \Pi_{\mathsf{CDN}}.\mathsf{Role}$ compute $\overline{z} = \overline{xy} = \mathsf{TEval}(tpk, (\overline{y}, \overline{a}, \overline{c}), (\epsilon, -\delta, 1))$.
- The roles in committee C_{l+1} execute $z = \mathsf{Decrypt}_{C_{l+1}}(\overline{z})$.

4.3 Security Analysis

Next, we state the theorem claiming that the protocol Π_{CDN} described above securely implements MPC in the YOSO model.

Theorem 2. *For any multiparty function F, the protocol Π_{CDN} described above running with the network $(\mathcal{F}_{\mathsf{Gen}}, \mathcal{F}_{\mathsf{BC}}, \mathcal{F}_{\mathsf{PP}})$ and with $t < N/2 \cdot (1 - \varepsilon)$ random Byzantine corruptions, YOSO-securely implements the ideal functionality $\mathcal{F}_{MPC}F$.*

sketch. We describe several hybrid games leading to a full simulator for Π_{CDN}. By the last game, the simulator no longer requires the honest input roles' inputs.

Game 0: During Setup, the simulator stores the $\mathsf{NIZKAoK}$ trapdoor td. The simulator executes all honest roles honestly.

Game 1: The simulator simulates the honest roles' proofs using SimP. This is indistinguishable from the previous game by the zero knowledge property of $\mathsf{NIZKAoK}$ (Definition 1).

Game 2: The simulator extracts witnesses from all proofs produced by malicious parties. If any of the proofs verify but fail to produce a valid witness, the simulator aborts. By the simulation extractability property of the $\mathsf{NIZKAoK}$ (Definition 2), the simulator only aborts with negligible probability. Notice that, at this point, thanks to the information it extracts, the simulator

- Always knows all decryption key shares $tsk_1, \ldots, tsk_n$;
- Always knows all plaintexts corresponding to ciphertexts.

Game 3: Let S be the set of corrupt roles in this round. During $\mathsf{Decrypt}(\beta)$, if this is the decryption of the output, the simulator ...

- Lets m be the plaintext corresponding to β.
- Computes corrupt roles' decryption shares as $d_i \leftarrow \mathsf{TPDec}(tpk, tsk_i, \beta)$ for $i \in S$, and
- Computes the honest roles' decryption shares as $\{d_i\}_{i\in\{1,\ldots,n\}\setminus S} \leftarrow \mathsf{SimTPDec}(tpk, \beta, m, \{tsk_i\}_{i\in\{1,\ldots,n\}\setminus S}, \{d_i\}_{i\in S,|S|\geq t})$.

This is indistinguishable from the previous game because plaintexts obtained during multiplication are randomly distributed, and by the partial decryption simulatability property of TE (Definition 3).

Game 4: During $\mathsf{Decrypt}(\beta)$, if this is the decryption of the output, the simulator sets m to be the MPC output given by $\mathcal{F}_{\text{MPC}}F$. This is indistinguishable from the previous game because, by inspection, all steps of the computation were correct, so the output obtained from $\mathcal{F}_{\text{MPC}}F$ is the same as the plaintext corresponding to the output ciphertext.

Game 5: Let S be the set of corrupt roles in this round. During Mult, the simulator picks a different x_i' (and, similarly, y_i') on behalf of one of the honest roles. However, during the corresponding $\mathsf{Decrypt}$, the simulator ...

- lets m be the plaintext corresponding to the $a + x$ using to the original x_i (or, similarly, the plaintext corresponding to the $b + y$ using the original y_i).
- computes corrupt roles' decryption shares as $d_i \leftarrow \mathsf{TPDec}(tpk, tsk_i, \beta)$ for $i \in S$, and
- computes the honest roles' decryption shares as $\{d_i\}_{i\in\{1,\ldots,n\}\setminus S} \leftarrow \mathsf{SimTPDec}(tpk, \beta, m, \{tsk_i\}_{i\in\{1,\ldots,n\}\setminus S}, \{d_i\}_{i\in S,|S|\geq t})$.

This is indistinguishable from the previous game by the partial decryption simulatability property of TE (Definition 3).

Game 6: During $\mathsf{Decrypt}(\beta)$, if this is a decryption done during multiplication, the simulator sets m to be a uniform random value. This is indistinguishable from the previous game because the distributions are identical; there is now no information about the original x_i and y_i in the view of the adversary.

Game 7: During Input, the simulator encrypts 0 on behalf of all of the honest input roles. (It continues to force all decryptions to This is indistinguishable from the previous game by the chosen plaintext security of TE . Note that at this point, the simulator does not require honest parties' inputs.

$\square$

5 Compiling Abstract YOSO to Natural YOSO

We explore the issue of compiling from the (abstract) YOSO model into a natural YOSO model with explicit role assignment. The example is only meant as a toy example to demonstrate the feasibility, not as a treatment of actual real-world role assignment.

The basic idea of our compilation is to have a UC ideal functionality spit out on a blockchain a stream of public keys. Each public key has a corresponding physical machine. This physical machine knows the secret key. The other machines do not know which machine corresponds to which public key in the key stream. The public keys can then be associated to roles in the protocol being executed using some open role assignment based on the execution schedule of the protocol being compiled. As a result random machines are now associated to roles.

Each public key in the key stream consist of two sub public keys, one for encrypting and one for verifying signatures. To send a secret message to a role, encrypt under the encryption key associated to a role. When executing a role, sign the outgoing message using the signature key of the role. Before sending the message, delete the secret keys and all other internal state and then post the signed ciphertexts on the blockchain. Notice that if a machine is corrupted then the roles executed by the machine are corrupted. However, since the mapping from public keys to machines is random and hidden there is no way to corrupt a particular role. Corrupting a machine corresponds to corrupting the random roles the machine have been assigned. Furthermore, a role already executed by a machine had its internal state deleted, so it remains honest even though the machine later becomes corrupted.

We want to model synchronous computation. Therefore we use the UC model with synchronous computation from [20]. All parties and ideal functionalities are aware which round they are currently in by accessing a global clock functionality.

We now go a bit more into detail. We assume we have an information theoretic secure YOSO protocol Π for the setting with just a public broadcast channel and perfectly secure channels to future roles. Note that this protocol might have been constructed in a modular way using the YOSO framework using several intermediary ideal functionalities and hybrid settings. We just assume that when all is plugged together we end up with a protocol using only broadcast and secure channels. We assume it is secure against some fraction τ of random corruptions of the computation roles $\mathsf{Role}^{\mathrm{CMP}}$. We assume the protocol can tolerate any number of chosen corruptions of input roles and output roles. We assume the protocol has public activation such that it can be predicted just from contents on the blockchain which role is to execute next.

The compilation is inspired by a blockchain setting where machines can post on a blockchain but where we want to ensure that each machine speak only once.[8] Messages can be sent to future roles via the blockchain which all machines can read. The problem is how to send a *secret* message via a public blockchain. We focus of the specific setting from [3] where a machine M is given a role R by a random public key pk appearing on the blockchain for which M knows the secret key. The machine M is picked at random among all N machines such that the adversary cannot corrupt a specific role. This public key is then assigned to one of the future roles. The listening schedule can be used to assign the keys to roles which are soon to receive messages. In Figure 8 we sketch a concrete infrastructure for generating the random keys pk.

Below we will assume that we have an ideal functionality $\mathcal{F}_{\mathrm{RA}}$ which spits out to machines such random keys pk. For simplicity we use the same $\mathcal{F}_{\mathrm{RA}}$ to model the blockchain. We then show how to UC securely compile a YOSO protocol to the $\mathcal{F}_{\mathrm{RA}}$-hybrid model.

Before we continue let us make a caveat. The focus on the present example is on illustrating the feasibility of compiling YOSO protocols into the usual UC framework. We will therefore skip many of the details that would show up in practice. Some of the simplifications we make are:

1. We assume the set of parties is fixed. No new parties arrive during the execution of the MPC. In practice one would have to deal with parties joining the blockchain during the MPC, in particular if the MPC runs for a long time.

2. In some cases when a computation takes place on a blockchain, the parties are selected according to how much stake they have. We assume a flat stake distribution where each

[8]Machines might by chance get several roles assigned but we will at least ensure that each role is executed at uniformly random points to hide among all other machines from denial of service attacks.

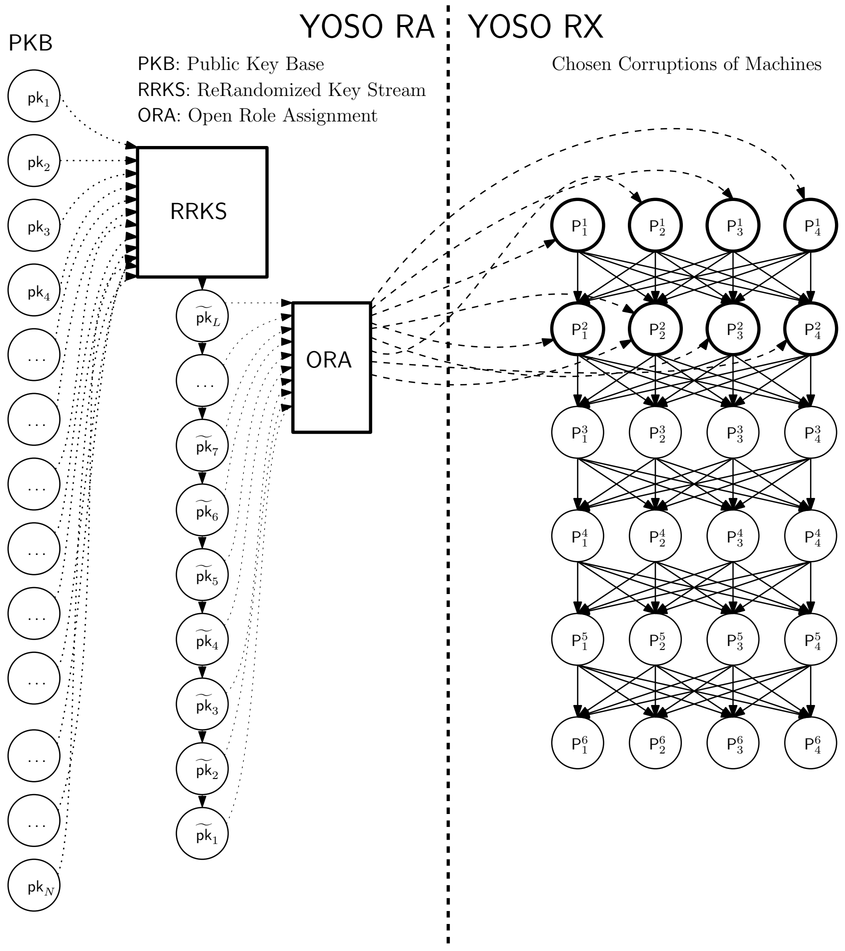

YOSO RA
YOSO RX
PKB
PKB: Public Key Base
RRKS: ReRandomized Key Stream
ORA: Open Role Assignment
Chosen Corruptions of Machines
RRKS
ORA
pk_1
pk_2
pk_3
pk_4
pk_N
$\widetilde{pk}_L$
$\widetilde{pk}_7$
$\widetilde{pk}_6$
$\widetilde{pk}_5$
$\widetilde{pk}_4$
$\widetilde{pk}_3$
$\widetilde{pk}_2$
$\widetilde{pk}_1$
P^1_1 P^1_2 P^1_3 P^1_4
P^2_1 P^2_2 P^2_3 P^2_4
P^3_1 P^3_2 P^3_3 P^3_4
P^4_1 P^4_2 P^4_3 P^4_4
P^5_1 P^5_2 P^5_3 P^5_4
P^6_1 P^6_2 P^6_3 P^6_4

Figure 8: Illustration of running a YOSO protocol on top of a concrete role assignment mechanism. There is a publicly known key base $pk_1, \dots, pk_N$ with machine M_i having public key pk_i. These keys were registered on the blockchain at some point. Some protocol is creating a rerandondomized key stream (RRKS). Each $\widetilde{pk}_j$ is a rerandomized version of some pk_i for a random index i. Machine M_i can decrypt under $\widetilde{pk}_j$. The index i is pseudo random to the other parties even when they are given $\widetilde{pk}_j$. An open role assignment (ORA) then assigns these rerandomized keys to future roles in the order in which they need to be spoken to according to the listening schedule of the protocol being compiled. By *open* we mean that a given position in the key stream is always assigned to the same role. This maintains that random machines get assigned to role.

party should be elected to a role with the same probability. In practice one would have to deal with non-uniform stake distribution and evolving stake distribution. However, these are largely orthogonal issues.

3. We assume that we can assign roles uniformly at random. In practice role assignment protocols might have a bias, like not having the same machine repeat twice in a row. As an example the protocol in [17] gives a slightly non-uniform assignment. Dealing with non-uniform assignment is not essential to demonstrate the idea of compiling a YOSO protocol to UC.

4. We assume that when the role assignment generates keys, then the secret keys are always unknown to the adversary when they are assigned to an honest party. This is not always the case. In [3] there is a constant probability that the secret keys assigned to an honest party leaks to the adversary. In [17] the secret keys of an honest party are always hidden. The proof for the weaker case is similar to the strong case with more corruptions. We therefore opt for the simpler case.

5. We do not concern ourselves with how to implement $\mathcal{F}_{\mathrm{RA}}$. We in particular do not claim that the role assignment protocol in [3] or [17] UC implements our very idealized $\mathcal{F}_{\mathrm{RA}}$. In particular the feature of forward security is hard to implement with adaptive security. It is an interesting future work to consider UC secure implementations of $\mathcal{F}_{\mathrm{RA}}$, or similar but more realistic models of role assignment.

5.1 The Big Picture

We now want to compile a YOSO protocol $\Pi^{\textsc{Abstract}}$ using only broadcast and secure channels into a natural YOSO protocol $\Pi^{\textsc{Natural}}$ using a blockchain and role assignment. An easy way to formalize this is to compile the YOSO protocol $\Pi^{\textsc{Abstract}}$ back to the YOSO model, but now for a corruption setting where we allow *chosen* corruption of the computation roles. To distinguish the two settings we call the computation roles in $\Pi^{\textsc{Natural}}$ computation machines (we still call the computation roles in $\Pi^{\textsc{Abstract}}$ computation roles). To be able to describe $\Pi^{\textsc{Natural}}$ it is convenient to have a lean notation for how roles in $\Pi^{\textsc{Abstract}}$ executes. For this purpose we introduce the Simple YOSO (SYOSO) model which gives a very simplified model of IT YOSO protocols only using broadcast and secure channels. We then first cast $\Pi^{\textsc{Abstract}}$ into a protocol $\Gamma^{\textsc{Abstract}}$ in the SYOSO. Then we compile $\Gamma^{\textsc{Abstract}}$ into a protocol $\Pi^{\textsc{Natural}}$ running on top of $\mathcal{F}_{\mathrm{RA}}$. We do not propose SYOSO as its own model, but rather as a technical stepping stone for the proof abstracting away some of the unnecessary details.

5.2 Simplified YOSO Model (SYOSO)

For convenience of notation we first specify the Simple YOSO (SYOSO) model which gives a very simplified model of IT YOSO protocols only using broadcast and secure channels. It also only considers secure function evaluation.

To conveniently describe how to run YOSO protocols on top of $\mathcal{F}_{\mathrm{RA}}$ we will assume that the parties act in sequence. The protocols we described in previous chapters acted in rounds. However, we assumed a rushing adversary, so security does not suffer from unrolling each round into a sequential activation. We note that this sequential execution is just for convenience of description. In practice the roles which do not depend on each other can still be executed in parallel.

When a role R executes it will post a public message m_{R} on the ledger and send secret messages to some future roles. We set up some notation for executing SYOSO protocols. An activation sequence ActSeq will be a sequence of pairs $(\mathsf{R}, m_{\mathsf{R}})$, where R is the role which acted and m_{R} is the public message it sent. If the role was to send a message but it did not, then we set $m_{\mathsf{R}} = \textsc{NoMsg}$. We assume that it can be efficiently computed from an activation sequence which role R is to act next, which roles $\mathcal{I}$ have sent secret messages to R and which roles $\mathcal{O}$ are to get secret messages from R. This follows if the protocol Π we start from has public activation. We write

$$\mathsf{UpNext}(\mathsf{ActSeq}) = (\mathsf{R}, \mathcal{I}_{\mathsf{R}}, \mathcal{O}_{\mathsf{R}}) \ ,$$

where each $\mathsf{S} \in \mathcal{I}_{\mathsf{R}}$ occurs in ActSeq and each each $\mathsf{S} \in \mathcal{O}_{\mathsf{R}}$ does not occur in $\mathsf{ActSeq} \cup \{\mathsf{R}\}$. This just says that roles do not send messages to roles which already executed and they do not try to receive from roles which did not yet execute. Furthermore, the UpNext function should guarantee that $\mathsf{S} \in \mathcal{I}_{\mathsf{R}}$ iff $\mathsf{R} \in \mathcal{O}_{\mathsf{S}}$. We assume the execution of the role is given by a randomized algorithm $\mathsf{ExecRole}$, where

$$\mathsf{ExecRole}(\mathsf{ActSeq}, x, \{(\mathsf{S}, m_{\mathsf{S},\mathsf{R}})\}_{\mathsf{S}\in\mathcal{I}}) = (\mathsf{R}, y, m_{\mathsf{R}}, \{(\mathsf{S}, m_{\mathsf{R},\mathsf{S}})\}_{\mathsf{S}\in\mathcal{O}}) \ ,$$

where R is the role which was executed, x is a possible secret input, y is a possible secret output, m_{R} is the message broadcast by R, and $m_{\mathsf{S},\mathsf{R}}$ is a secret message from sender S to receiver R. Here $x = \perp$ if $\mathsf{R} \not\in \mathsf{Role}^{\textsc{In}}$ and $y = \perp$ if $\mathsf{R} \not\in \mathsf{Role}^{\textsc{Out}}$. Note that $\mathsf{ExecRole}$ can use UpNext to compute which role R it is to execute. We let it output R for notational convenience later.

We consider secure function evaluation of a function

$$(y_1, \ldots, y_m) = F(x_1, \ldots, x_n)$$

with n secret inputs and m secret outputs. We assume the input roles $\mathsf{I}_1, \ldots, \mathsf{I}_n$ are fixed, that the output roles $\mathsf{O}_1, \ldots, \mathsf{O}_m$ are fixed, and that the activation sequence begins with $\mathsf{I}_1, \ldots, \mathsf{I}_n$ and ends with $\mathsf{O}_1, \ldots, \mathsf{O}_m$. Note that this in particular means that $\mathsf{UpNext}(()) = (\mathsf{I}_1, \varnothing, \mathcal{O})$ for some $\mathcal{O}$.

We assume for convenience that input roles do not communicate and output roles do not communicate. This is the case for the protocol we gave above. All entities tacitly take the security parameter κ as input.

An execution $\textsc{Exec}^{\textsc{static},\tau}_{\Gamma,\mathcal{A},\mathcal{E}}(\kappa)$ with τ static corruptions proceeds as below.

1. Let $\mathcal{A}$ specify any number of static Byzantine corruptions of $\mathsf{Role}^{\textsc{In}}$ and $\mathsf{Role}^{\textsc{Out}}$. Pick τ uniformly random Byzantine corruptions of $\mathsf{Role}^{\textsc{Cmp}}$. Let $\mathsf{Corrupt} \subset \mathsf{Role}$ by the set of corrupted parties let $\mathsf{Honest} = \mathsf{Role} \setminus \mathsf{Corrupt}$. Now let $\mathcal{E}$ and $\mathcal{A}$ interact as an adversary and environment in the UC framework and let $\mathcal{A}$ attack the protocol as follows.
2. Run $\mathcal{E}$ to get inputs x_i for $\mathsf{I}_i \in \mathsf{Honest}$.
3. Let $\mathsf{ActSeq} = ()$ be the empty sequence.
4. Let $(\mathsf{R}, \mathcal{I}_\mathsf{R}, \mathcal{O}_\mathsf{R}) = \mathsf{UpNext}(\mathsf{ActSeq})$.
5. If $\mathsf{R} = \mathsf{I}_i$ and $\mathsf{I}_i \in \mathsf{Honest}$ then let $x = x_i$, otherwise let $x = \perp$.
6. If $\mathsf{R} \in \mathsf{Honest}$ then sample
$$(y, m_\mathsf{R}, \{(\mathsf{S}, m_{\mathsf{R},\mathsf{S}})\}_{\mathsf{S} \in \mathcal{O}_\mathsf{R}}) \leftarrow \mathsf{ExecRole}(\mathsf{ActSeq}, x, \{(\mathsf{S}, m_{\mathsf{S},\mathsf{R}})\}_{\mathsf{S} \in \mathcal{I}_\mathsf{R}})$$
otherwise sample
$$(m_\mathsf{R}, \{(\mathsf{S}, m_{\mathsf{R},\mathsf{S}})\}_{\mathsf{S} \in \mathcal{O}_\mathsf{R}}) \leftarrow \mathcal{A} \ .$$
7. Append $(\mathsf{R}, m_\mathsf{R})$ to ActSeq and input m_R to $\mathcal{A}$.
8. For $\mathsf{S} \in \mathcal{O}_\mathsf{R} \cap \mathsf{Corrupt}$, input $m_{\mathsf{R},\mathsf{S}}$ to $\mathcal{A}$.
9. For $\mathsf{S} \in \mathcal{O}_\mathsf{R} \cap \mathsf{Honest}$, input $|m_{\mathsf{R},\mathsf{S}}|$ to $\mathcal{A}$.
10. If $\mathsf{R} = \mathsf{O}_j$ and $\mathsf{R} \in \mathsf{Honest}$ then let $y_j = y$ and input y_j to $\mathcal{E}$.
11. If $\mathsf{R} \neq \mathsf{O}_m$ then go to Step 4.
12. Run $\mathcal{E}$ to get $z \in \{0,1\}$ and output z.

We can define adaptive security similar to YOSO. On an adaptive corruption of R the adversary learns the messages $m_{\mathsf{S},\mathsf{R}}$ sent to R and may change them if R did not execute yet.

A simulation $\textsc{Ideal}_{F,\mathcal{S},\mathcal{E}}(\kappa)$ proceeds as follows:

1. Let $\mathcal{S}$ specify any number of static Byzantine corruptions of $\mathsf{Role}^{\textsc{In}}$ and $\mathsf{Role}^{\textsc{Out}}$. Pick τ uniformly random Byzantine corruptions of $\mathsf{Role}^{\textsc{Cmp}}$. Let $\mathsf{Corrupt} \subset \mathsf{Role}$ by the set of corrupted parties let $\mathsf{Honest} = \mathsf{Role} \setminus \mathsf{Corrupt}$. Now let $\mathcal{E}$ and $\mathcal{S}$ interact as simulator and environment in the UC framework and let $\mathcal{S}$ attack the protocol as follows.
2. Run $\mathcal{E}$ to get inputs x_i for $\mathsf{I}_i \in \mathsf{Honest}$.
3. If at some point the simulator $\mathcal{S}$ outputs a value x_i for each $\mathsf{I}_i \in \mathsf{Corrupt}$, then compute $(y_1, \ldots, y_m) = F(x_1, \ldots, x_n)$ and give y_j for each $\mathsf{O}_j \in \mathsf{Corrupt}$ to $\mathcal{S}$.
4. For $j = 1, \ldots, m$, if $\mathsf{O}_j \in \mathsf{Honest}$ then input y_j to $\mathcal{E}$.
5. Run $\mathcal{E}$ to get $z \in \{0,1\}$ and output z.

We call Γ SYOSO secure if for all PPT $\mathcal{A}$ there exists a PPT $\mathcal{S}$ such that

$$\textsc{Exec}_{\Gamma,\mathcal{A},\mathcal{E}}(\kappa) \approx \textsc{Ideal}_{F,\mathcal{S},\mathcal{E}}(\kappa) \ .$$

It is easy to verify that YOSO security implies SYOSO security up to some syntactic "massaging." In both cases the simulator has to do straight-line simulation of the honest parties in an environment picking the inputs while only having access to evaluating f on inputs where the inputs of the corrupted parties are chosen by $\mathcal{S}$.

5.3 Toy UC Model of a Global Ledger and Role Assignment

We would then like to be able to emulate the chosen corruption of computation machines in $\Pi^{\textsc{Natural}}$ using random corruption of computation roles in $\Gamma^{\textsc{Abstract}}$. For this purpose we use $\mathcal{F}_{\mathrm{RA}}$ to hide which computation machine executes which computation roles. In a bit more details we let $\mathsf{Machine}^{\textsc{In}} = \mathsf{Role}^{\textsc{In}}$ and $\mathsf{Machine}^{\textsc{Out}} = \mathsf{Role}^{\textsc{Out}}$. We let $\mathsf{Machine}^{\textsc{Cmp}}$ be a subset of all pid's such that $\mathsf{Machine}^{\textsc{Cmp}} \cap (\mathsf{Machine}^{\textsc{In}} \cup \mathsf{Machine}^{\textsc{Out}}) = \varnothing$. For $\mathsf{R} \in \mathsf{Role}^{\textsc{In}}$ we let $\mathsf{M}_\mathsf{R} = \mathsf{R} \in \mathsf{Machine}^{\textsc{In}}$, i.e., we let machine R execute role R. Similarly, for $\mathsf{R} \in \mathsf{Role}^{\textsc{Out}}$ we let $\mathsf{M}_\mathsf{R} = \mathsf{R} \in \mathsf{Machine}^{\textsc{Out}}$. We then use an ideal functionality $\mathcal{F}_{\mathrm{RA}}$ for role assignment to define a random map $\mathsf{RA} : \mathsf{Role}^{\textsc{Cmp}} \to \mathsf{Machine}^{\textsc{Cmp}}$ and we let $\mathsf{M}_\mathsf{R} = \mathsf{RA}(\mathsf{R})$. We then run the YOSO protocol on a blockchain while letting M_R execute role R. Recall that the reason why we do not assign input roles and output roles at random is to model the practical reality that often some specific parties are to give inputs and see the outputs.

5.3.1 The Timed Ledger with Role Assignment

We model the blockchain as a very simplistic timed ledger. A ledger is just a growing sequence of messages which all parties agree on. A timed ledger is one where messages get timestamps when posted. This timestamp is somewhat close to the real time. As an example consider a proof of stake blockchain where blocks are associated to slots and slot numbers are closely associated to physical time. Here a message can be timestamped via the slot number of the block in which it occurs. For simplicity we use a simple model of timed ledger. We will use $\Delta_{\textsc{Post}}$ to bound the time it takes to post something to the ledger. We will make the very simplistic assumption that when something is posted on a blockchain then all honest parties will see it at the same local clock time and will output the message with a timestamp which is this local clock time.[9] It is possible to compile to blockchains with weaker delivery guarantees, but for our toy example of a compilation nothing is lost in making this simplifying choice. We will also assume that we have a mechanism for generating fresh public keys which appear on the ledger and where a random secret party knows the corresponding secret keys. For simplicity we will allow the adversary to decide when a given role R has its keys generated. In terms of safety this is the conservative modeling: if you can prove security in this model, then the protocol is secure with any order of key generation. In terms of liveness the model is problematic as it allows the adversary a denial of service attack of not generating any keys; one for instance cannot prove guaranteed output delivery in this setting. One can solve this issue by requiring that the compiled protocol is non-trivial: *if* the adversary eventually generate keys for all honest roles *then* the protocol eventually generates outputs.

The toy ledger with role assignment is given in Figure 9.

Notice that in $\mathcal{F}_{\mathrm{RA}}$, if a machine M_R with role R becomes corrupted then $\mathcal{F}_{\mathrm{RA}}$ leaks the secret keys of the role R if the GENERATE message is still not output to the parties. This is a simplistic model of the following real world setting. When M_R is to learn the secret keys dk_R and sk_R some message will be sent which only M_R can decrypt. Imagine that this message is posted on the ledger. While this message is in transit the machine M_R cannot avoid having the ability to decrypt it. The purpose is, after all, that it lets M_R learn $(\mathsf{dk}_\mathsf{R}, \mathsf{sk}_\mathsf{R})$. However, after the message is learned by M_R and M_R decrypted it to learn $(\mathsf{dk}_\mathsf{R}, \mathsf{sk}_\mathsf{R})$ we assume that M_R can forget

[9]One inefficient way to implement this is as follows. Assume an upper bound $\Delta_{\textsc{Block}}$ on the time it takes for a block with timestamp t to become known to a party, i.e., a block with timestamp t arrives at all parties before time $t + \Delta_{\textsc{Block}}$. Note that on a Nakamoto style blockchain the time t at which a block "arrives" is the time at which it is known to be in the stable common prefix forever except with negligible probability, which can be much later than when it is seen for the first time. Assume furthermore that local clocks drift at most $\Delta_{\textsc{Clock}}$ from the global time. Now when a block B with timestamp t arrives wait until local time $t + \Delta_{\textsc{Block}} + \Delta_{\textsc{Clock}}$ and then output the messages in B with timestamp $t + \Delta_{\textsc{Block}} + \Delta_{\textsc{Clock}}$.

Init Let Posted be the empty set, let $\mathsf{Ordered}$ be the empty sequence. In any round let t^{Now} denote the current time. We assume the ledger is parametrized by a set $\mathsf{Machine}$ which is the set of parties allowed to use it. They are divided into $\mathsf{Machine}^{\text{RAN}}$ and $\mathsf{Machine}^{\text{FIXED}}$ with random roles respectively fixed roles.

Post On input (POST, m) from $\mathsf{pid} \in \mathsf{Machine}$, output $(\text{POST}, \mathsf{pid}, m)$ to the adversary and add $(\mathsf{pid}, t_{\text{Now}}, m)$ to the set Posted.

Order On input $(\text{ORDER}, \mathsf{pid}, m)$ from the adversary where some $(\mathsf{pid}, t, m) \in \mathsf{Posted}$ and no $(\mathsf{pid}, t', m) \in \mathsf{Ordered}$, append $(\mathsf{pid}, t_{\text{Now}}, m)$ to $\mathsf{Ordered}$. Execute this command automatically when $(\mathsf{pid}, t, m) \in \mathsf{Posted}$ and no $(\mathsf{pid}, t', m) \in \mathsf{Ordered}$ and $t = t_{\text{Now}} - \Delta_{\text{POST}}$.

Read On input (READ) from $\mathsf{pid} \in \mathsf{Machine}$, leak (READ) to the adversary and output $\mathsf{Ordered}$ to pid.

Generate On input $(\text{GENERATE}, \mathsf{R})$ from the adversary, sample $(\mathsf{ek_R}, \mathsf{dk_R}) \leftarrow \mathsf{Enc.Gen}(1^\kappa)$ and sample $(\mathsf{vk_R}, \mathsf{sk_R}) \leftarrow \mathsf{Sig.Gen}(1^\kappa)$. If $\mathsf{R} \in \mathsf{Role}^{\text{FIXED}}$ then let $\mathsf{pid} = \mathsf{R}$. Otherwise sample a uniformly random $\mathsf{pid} \in \mathsf{Machine}^{\text{RAN}}$. Then leak $m = (\text{GENERATE}, \mathsf{R}, \mathsf{ek_R}, \mathsf{vk_R})$ to the adversary and add $(\mathsf{ra}, t_{\text{Now}}, m)$ to Posted. Let $M_\mathsf{R} = \mathsf{pid}$. When (ra, t', m) is later added to $\mathsf{Ordered}$ output $(\text{GENERATE}, \mathsf{R}, \mathsf{ek_R}, \mathsf{dk_R}, \mathsf{vk_R}, \mathsf{sk_R})$ to $\mathsf{M_R}$.

Forward Security When $\mathsf{M_R}$ becomes corrupted, output $(\text{GENERATE}, \mathsf{R}, \mathsf{ek_R}, \mathsf{dk_R}, \mathsf{vk_R}, \mathsf{sk_R})$ to the adversary if $(\text{GENERATE}, \mathsf{R}, \mathsf{ek_R}, \mathsf{dk_R}, \mathsf{vk_R}, \mathsf{sk_R}) \notin \mathsf{Ordered}$.

Figure 9: Toy Timed Ledger $\mathcal{F}_{\text{RA}}$ with Role Assignment

the ability to decrypt the message, say by deleting or evolving the secret key needed to decrypt these role assignment messages. This would be part of the protocol implementing $\mathcal{F}_{\text{RA}}$. The value $(\mathsf{dk_R}, \mathsf{sk_R})$ might still be around for the sake of the outer protocol Π^{NATURAL}, but for the sake of $\mathcal{F}_{\text{RA}}$ it should no longer be available to $\mathsf{M_R}$ if $\mathsf{M_R}$ is corrupted in the implementation of $\mathcal{F}_{\text{RA}}$. This part of the model is essential for being able to prove adaptive security: it guarantees that if $\mathsf{M_R}$ is adaptively corrupted after executing R it is the same as R having been honest.[10]

Notice that if an honest role R posts a message m to the ledger and is later corrupted, then the message m is still delivered. This holds even of R is corrupted while m is still in transit (it is in Posted and not in $\mathsf{Ordered}$). This is sometimes called atomic message delivery. It is needed for adaptive corruption to be meaningful in the YOSO framework. We use random role assignment to keep the role-machine associating hidden until the machine has to send its single message. If a denial of service attacker is strong enough to kill a machine *after* role execution and take back the message there is little we can do. We therefore assume atomic message delivery.

A compiled SYOSO role R is ready to execute when it is its turn according to ActSeq, and all the roles it has to send messages to already have public keys on $\mathcal{F}_{\text{RA}}$. When a SYOSO role R is ready to be executed by a party pid it will post the public message m_R on $\mathcal{F}_{\text{RA}}$ and send secret messages to some future roles by encrypting under their public keys. The public message m_R will be signed by R, as will the ciphertexts. Before posting, the party pid deletes the secret keys $\mathsf{sk_R}$ and $\mathsf{dk_R}$.

We now describe this protocol in more detail. We set up some notation for executing a SYOSO protocol in practice. We use the same functions UpNext and $\mathsf{ExecRole}$ as given by the SYOSO protocol Γ. The activation sequence ActSeq will be the sequence of pairs $(\mathsf{R}, m_\mathsf{R})$,

[10]Looking a bit forward, our simulation strategy will then be to ensure that if $\mathsf{M_R}$ is corrupted *before* executing R, then we assign it one of the statically corrupted roles R. The simulator can do this as the role R of M is unknown to the adversary until M is corrupted and the simulator can therefore program $\mathcal{F}_{\text{RA}}$ to assign a given R to M at corruption time.

The protocol Π^{NATURAL} running in the $\mathcal{F}_{\text{RA}}$-hybrid model with parties $\mathsf{Machine}$. We assume that input machines $\mathsf{Machine}^{\text{IN}}$ have their inputs provided before the execution starts. The protocol uses a labelled CCA secure encryption scheme Enc and an unforgeable signature scheme.

Role Assignment If $\mathcal{F}_{\text{RA}}$ outputs $(\text{GENERATE}, \mathsf{R}, \mathsf{ek}_\mathsf{R}, \mathsf{dk}_\mathsf{R}, \mathsf{vk}_\mathsf{R}, \mathsf{sk}_\mathsf{R})$ to M then M stores $\mathsf{SKeys}_\mathsf{R} = (\mathsf{R}, \mathsf{dk}_\mathsf{R}, \mathsf{sk}_\mathsf{R})$.

Ready to Receive We say that $\mathsf{R} \in \mathsf{Role}$ is *ready to receive* if $(\mathsf{ra}, t', (\text{GENERATE}, \mathsf{R}, \mathsf{ek}_\mathsf{R}, \mathsf{vk}_\mathsf{R}))$ occurs in $\mathsf{Ordered}$. If so, we say that it was ready at ledger time t'.

Ready to Send When it first happens that a message $(\cdot, t', m)$ is posted to $\mathsf{Ordered}$ such that $\mathsf{UpNext}(\mathsf{ActSeq}) = (\mathsf{R}, \mathcal{I}_\mathsf{R}, \mathcal{O}_\mathsf{R})$ then we say that R is *ready to send*. We say R was ready at ledger time t'.

Ready to Execute When it first happens that a message $(\cdot, t', m)$ is posted to $\mathsf{Ordered}$ such that R is ready to send and it holds for all $\mathsf{S} \in \mathcal{O}_\mathsf{R}$ that S is ready to receive we say that R is *ready to execute*. We say R was ready at ledger time t'. Note that this is a publicly observable event which all machine agree on given the same $\mathsf{Ordered}$.

Time out If it happened at ledger time t' that R is ready to execute and the local time is now $t = t' + \Delta_{\text{POST}}$ and no message of the form $(\mathsf{R}, m_\mathsf{R}, M, \sigma)$ was posted with $\mathsf{Sig.Ver}(\mathsf{vk}_\mathsf{R}, (m_\mathsf{R}, M), \sigma) = \top$, then virtually inject a message $(\mathsf{R}, m_\mathsf{R} = \text{NOMSG}, \text{NOMSG}, \text{NOMSG})$ into $\mathsf{Ordered}$ at ledger time $t' + \Delta_{\text{POST}}$. By this we mean that all machines behave as if that message is in $\mathsf{Ordered}$.

Role Execution When it happens that R is ready to execute at time t and M is the machine which has $\mathsf{SKeys}_\mathsf{R} = (\mathsf{R}, \mathsf{dk}_\mathsf{R}, \mathsf{sk}_\mathsf{R})$ stored, then M proceeds as follows at time t.

1. For each S in $\mathcal{I}_\mathsf{R}$ inspect $\mathsf{Ordered}$ to find $(\mathsf{S}, m_\mathsf{S}, M, \sigma)$. We know it is there as $\mathsf{S} \in \mathcal{I}_\mathsf{R}$ and we enforce by convention that the roles execute in turn according to UpNext on $\mathsf{Ordered}$. If $m_\mathsf{S} = \text{NOMSG}$, or S posted the message before it was ready to execute, or there are no keys $(\mathsf{S}, \mathsf{ek}_\mathsf{S}, \mathsf{vk}_\mathsf{S})$ on $\mathsf{Ordered}$ or $\mathsf{Sig.Ver}(\mathsf{vk}_\mathsf{S}, (m_\mathsf{S}, M), \sigma) = \bot$, then set $m_{\mathsf{S},\mathsf{R}} = \text{NOMSG}$. Otherwise parse M as $\{(\mathsf{R}', M_{\mathsf{S},\mathsf{R}'})\}_{\mathsf{R}' \in \mathcal{O}_\mathsf{S}}$ and retrieve $M_{\mathsf{S},\mathsf{R}}$. If this is not possible let $m_{\mathsf{S},\mathsf{R}} = \text{NOMSG}$. Otherwise let $m_{\mathsf{S},\mathsf{R}} = \mathsf{Enc.Dec}^\mathsf{S}(\mathsf{dk}_\mathsf{R}, M_{\mathsf{S},\mathsf{R}})$. Here $^\mathsf{S}$ is the label.
2. If $\mathsf{R} = \mathsf{I}_i$ then retrieve input x_i and let $x = x_i$. Otherwise let $x = \bot$.
3. Sample $(\mathsf{R}, y, m_\mathsf{R}, \{(\mathsf{S}, m_{\mathsf{R},\mathsf{S}})\}_{\mathsf{S} \in \mathcal{O}_\mathsf{R}}) \leftarrow \mathsf{ExecRole}(\mathsf{ActSeq}, x, \{(\mathsf{S}, m_{\mathsf{S},\mathsf{R}})\}_{\mathsf{S} \in \mathcal{I}_\mathsf{R}})$
4. For $\mathsf{S} \in \mathcal{O}_\mathsf{R}$ compute $M_{\mathsf{R},\mathsf{S}} \leftarrow \mathsf{Enc.Enc}^\mathsf{R}(\mathsf{ek}_\mathsf{R}, m_{\mathsf{R},\mathsf{S}})$ and let $M = \{(\mathsf{S}, M_{\mathsf{R},\mathsf{S}})\}_{\mathsf{S} \in \mathcal{O}_\mathsf{R}}$. Here $^\mathsf{R}$ is the label.
5. Compute $\sigma = \mathsf{Sig.Sign}(\mathsf{sk}_\mathsf{R}, (m_\mathsf{R}, M))$.
6. If $\mathsf{R} = \mathsf{O}_j$ let $y_j = y$ and output y_j.
7. Erase sk_R and dk_R and all internal values used during the above computation, except $(\mathsf{R}, m_\mathsf{R}, M, \sigma)$.
8. Post $(\mathsf{R}, m_\mathsf{R}, M, \sigma)$ on $\mathcal{F}_{\text{RA}}$.

Figure 10: The protocol Π^{NATURAL} compiled from Γ^{ABSTRACT}.

where R is the role which acted and m_{R} is the public message it sent on $\mathcal{F}_{\mathrm{RA}}$. To keep this synchronized with the same notion in the SYOSO model we make the convention that we ignore $(\mathsf{R}, m_{\mathsf{R}})$ if it occurs on the ledger out of turn, i.e., it occurs at a point in time where it was not the case that $\mathsf{UpNext}(\mathsf{ActSeq}) = (\mathsf{R}, \cdot, \cdot)$. In the exposition below we will tacitly behave as if any messages sent on $\mathsf{Ordered}$ out of turn is not there. If the role was to send a message but did not, then we will use a timeout to set $m_{\mathsf{R}} = \text{NoMsg}$, more on this below. Notice that the current activation sequence can be read from $\mathcal{F}_{\mathrm{RA}}$, it is an efficient function $\mathsf{ActSeq}(\mathsf{Ordered})$ of $\mathsf{Ordered}$. Since $\mathsf{Ordered}$ is implicitly given at all parties we will write ActSeq to mean $\mathsf{ActSeq}(\mathsf{Ordered})$. With this notation the protocol is given in Figure 10.

5.4 Theorem Statements

We prove three results. We show that we can compile a statically secure YOSO protocol into a statically secure natural YOSO protocol for the $\mathcal{F}_{\mathrm{RA}}$-hybrid model, with a slight reduction in the corruption level tolerated. The new corruption level can be arbitrarily close to the original, but not equal. We also show that we can compile an adaptively secure YOSO protocol into an adaptively secure natural YOSO protocol for the $\mathcal{F}_{\mathrm{RA}}$-hybrid model, with a slight reduction in corruption level.

Theorem 3. *Let $\tau > 0$ and $\rho > 0$ be constants with $\rho < \tau$.*

- *If there exists Π^{Abstract} for the synchronous YOSO model using hybrid functionalities $\mathcal{F}_{\mathsf{BC}}$ and $\mathcal{F}_{SPP}$ and which IT YOSO implements F against τ random, static corruptions, then there exist Π_1^{Natural} for the $\mathcal{F}_{RA}$-hybrid model which implements F against ρ chosen, static corruptions.*
- *If there exists Π^{Abstract} for the synchronous YOSO model using hybrid functionalities $\mathcal{F}_{\mathsf{BC}}$ and $\mathcal{F}_{SPP}$ and which IT YOSO implements F against τ random, adaptive point-corruptions, then there exist Π_2^{Natural} for the $\mathcal{F}_{RA}$-hybrid model which implements F against ρ chosen, adaptive point corruptions.*

The protocols Π_i^{Natural} use a number of computation machines which is polynomial in the number of computation roles of Π^{Abstract}. The degree of the polynomial depends on the constant gap $\rho - \tau$. The compiled protocols use a labeled CCA secure encryption scheme and an unforgeable signature scheme. The compiled protocol Π_2 in addition assumes that the encryption scheme is a public key non-committing encryption scheme.

For the public-key non-committing scheme we can use the two-round NCE scheme from [9]. The public key is the first message from the receiver to the sender. The ciphertext is the second message from the sender to the receiver. To make it labeled CCA one can send the ciphertext of the NCE encrypted by a labeled CCA scheme.

We comment on the relation between τ and ρ. Let R be the number of roles assigned at random. Let M be the number of machines they are assigned to. Consider throwing R balls uniformly at random into M bins. If you corrupt a fraction ρ of bins you will on average corrupt a fraction ρ of roles. So if you assume M grows with the security parameter, and you set $\rho < \tau$ with a big enough margin to account for the variation, you can ensure that you corrupt at most a fraction τ roles, except with negligible probability. The variance can be made arbitrarily small by increasing M, which is how we get the above theorems. If M is small and fixed then many roles get assigned to the same machine and the variance goes up. This will lower the tolerated ρ. We do not discuss how to compute ρ from given τ, R and M. We simply assume that we can set M arbitrarily to "squeeze" ρ to τ .

5.5 Analysis

We now prove Theorem 3. Let M be the number of computation machines $\Pi^{\textsc{Natural}}$ and let R be the number of computation roles in $\Gamma^{\textsc{Abstract}}$. Assume that R grows at least linearly in the security parameter κ. If not, then it can be padded to do so by introducing dummy roles. Let $M = R^{2+c}$ for a positive integer c. We assign R roles at random to M machines. This means that the probability that any two roles are assigned to the same machine is less than R^{-2-c}. Therefore the expected number of collisions is less than R^{-c}. For any constant δ we can set c large enough that the fraction of roles in a collision is less than δ except with negligible probability. This means that for any $\rho < \tau$ we can ensure that corrupting ρ machines will corrupt at most τ roles except with negligible probability. Below we assume that this has been assured by setting M large enough.

We now prove that $\Pi^{\textsc{Natural}}$ UC implements $\mathcal{F}_{\mathrm{MPC}}^{F}$ in the $\mathcal{F}_{\mathrm{RA}}$-hybrid model with τ adaptive, chosen corruptions of $\mathsf{Machine}^{\mathrm{CMP}}$. Along the way we will also prove the static case as a warm-up case.

It is clear that by construction and by the unforgeability of the signature scheme, the messages computed and sent (and laster received) by the honest parties are computed correctly according to $\mathsf{ExecRole}$. Therefore, if the number of corrupted roles can be kept below τ the outputs of the computation at honest roles will be correct and match those of F. Below we then focus on how to simulate the transcript of the protocol to $\mathcal{E}$ in an indistinguishable manner. We go via a number of hybrid distributions to prove this. In all the processes and hybrids we describe below we assume that any number of chosen, static corruptions are allowed among role $\mathsf{Role}^{\mathrm{IN}}$ and $\mathsf{Role}^{\mathrm{OUT}}$.

- Let $\textsc{Exec}^{\textsc{adaptive},\rho}_{\mathcal{F}_{\mathrm{MPC}}^{F},\mathcal{S}_{\mathrm{UC}},\mathcal{E}}$ denote the ideal process with ideal functionality $\mathcal{F}_{\mathrm{MPC}}^{F}$ and with adaptive, chosen corruptions of up to ρ parties in $\mathsf{Machine}^{\mathrm{CMP}}$.
- Let $\textsc{Exec}^{\textsc{adaptive},\rho}_{\Pi^{\textsc{Natural}},\mathcal{A},\mathcal{E}}$ denote the UC execution of $\Pi^{\textsc{Natural}}$ in the $\mathcal{F}_{\mathrm{RA}}$-hybrid model with ρ adaptive, chosen corruptions of $\mathsf{Machine}^{\mathrm{CMP}}$ and adversary $\mathcal{A}$.
- Let $\textsc{Ideal}^{\textsc{adaptive},\tau}_{F,\mathcal{S}_{\mathrm{SYOSO}},\mathcal{E}}$ denote the ideal SYOSO ideal process with τ adaptive, random corruptions of $\mathsf{Role}^{\mathrm{CMP}}$.
- Let $\textsc{Exec}^{\mathrm{SYOSO},\textsc{adaptive},\tau}_{\Gamma^{\textsc{Abstract}},\mathcal{B},\mathcal{E}}$ denote the real-world SYOSO protocol with τ adaptive, random corruptions of $\mathsf{Role}^{\mathrm{CMP}}$ and adversary $\mathcal{B}$.

We have to prove that for all PPT $\mathcal{A}$ there exists a PPT simulator $\mathcal{T}$ such that

$$\textsc{Exec}^{\textsc{adaptive},\rho}_{\Pi^{\textsc{Natural}},\mathcal{A},\mathcal{E}} = \textsc{Exec}^{\textsc{adaptive},\rho}_{\mathcal{F}_{\mathrm{MPC}}^{F},\mathcal{T},\mathcal{E}} \ .$$

We give a hybrid argument. The first hybrids are mainly syntactic or by assumption, so we will only sketch the proofs. We can prove that for all PPT $\mathcal{S}_{\mathrm{SYOSO}}$ there exists PPT $\mathcal{S}_{\mathrm{UC}}$ such that

$$\textsc{Ideal}^{\textsc{adaptive},\tau}_{F,\mathcal{S}_{\mathrm{SYOSO}},\mathcal{E}} = \textsc{Exec}^{\textsc{adaptive},\rho}_{\mathcal{F}_{\mathrm{MPC}}^{F},\mathcal{S}_{\mathrm{UC}},\mathcal{E}} \ .$$

This follows by construction of the processes. In both processes the simulator has the same powers and obligations, so $\mathcal{S}_{\mathrm{UC}} = \mathcal{S}_{\mathrm{SYOSO}}$ up to a thin wrapper.

Synchronous YOSO security for the model with broadcast and secure channels imply SYOSO security by construction, and by SYOSO security we have that for all PPT $\mathcal{B}$ there exists PPT $\mathcal{S}_{\mathrm{SYOSO}}$ such that

$$\textsc{Exec}^{\mathrm{SYOSO},\textsc{adaptive},\tau}_{\Gamma^{\textsc{Abstract}},\mathcal{B},\mathcal{E}} \approx \textsc{Ideal}^{\textsc{adaptive},\tau}_{F,\mathcal{S}_{\mathrm{SYOSO}},\mathcal{E}} \ .$$

This means that we have reduced the proof obligation to proving that for all PPT $\mathcal{A}$ there exist PPT $\mathcal{S}$ such that

$$\textsc{Exec}^{\textsc{adaptive},\rho}_{\Pi^{\textsc{Natural}},\mathcal{A},\mathcal{E}} \approx \textsc{Exec}^{\text{SYOSO},\textsc{adaptive},\tau}_{\Gamma^{\textsc{Abstract}},\mathcal{S},\mathcal{E}} \ .$$

If we can lift this proof obligation then we can in particular for all PTT $\mathcal{A}$ construct a PPT simulator $\mathcal{T} = \mathcal{S} \circ \mathcal{S}_{\text{SYOSO}} \circ \mathcal{S}_{\text{UC}}$ composed of $\mathcal{S}$, $\mathcal{S}_{\text{SYOSO}}$, and $\mathcal{S}_{\text{UC}}$ such that

$$\textsc{Exec}^{\textsc{adaptive},\rho}_{\Pi^{\textsc{Natural}},\mathcal{A},\mathcal{E}} \approx \textsc{Exec}^{\textsc{adaptive},\rho}_{\mathcal{F}^{F}_{\text{MPC}},\mathcal{T},\mathcal{E}} \ .$$

Note that there is a slight misuse of notation in claiming $\textsc{Ideal}^{\textsc{adaptive},\tau}_{F,\mathcal{S}_{\text{SYOSO}},\mathcal{E}} = \textsc{Exec}^{\textsc{adaptive},\rho}_{\mathcal{F}^{F}_{\text{MPC}},\mathcal{S}_{\text{UC}},\mathcal{E}}$. In $\textsc{Exec}^{\textsc{adaptive},\rho}_{\mathcal{F}^{F}_{\text{MPC}},\mathcal{S}_{\text{UC}},\mathcal{E}}$ the corruption aggregation of the UC framework will report corrupted *machines* to $\mathcal{E}$. In the process $\textsc{Ideal}^{\textsc{adaptive},\tau}_{F,\mathcal{S}_{\text{SYOSO}},\mathcal{E}}$ we work with roles. We should allow $\mathcal{S}_{\text{UC}}$ to simulate this difference which it formally cannot right now as the corruptions are reported by the corruption aggregation mechanism. Similarly for the simulator $\mathcal{S}$ which is simulating machines from roles. We will not go into the details of patching the processes to allow this bridging. Note in particular that when we use $\mathcal{S}$ and $\mathcal{S}_{\text{SYOSO}}$ as components in $\textsc{Exec}^{\textsc{adaptive},\rho}_{\mathcal{F}^{F}_{\text{MPC}},\mathcal{S}\circ\mathcal{S}_{\text{SYOSO}}\circ\mathcal{S}_{\text{UC}},\mathcal{E}}$ we do not have this formal mismatch, as both $\textsc{Exec}^{\textsc{adaptive},\rho}_{\mathcal{F}^{F}_{\text{MPC}},\mathcal{S}\circ\mathcal{S}_{\text{SYOSO}}\circ\mathcal{S}_{\text{UC}},\mathcal{E}}$ and $\textsc{Exec}^{\textsc{adaptive},\rho}_{\Pi^{\textsc{Natural}},\mathcal{A},\mathcal{E}}$ work with machines. So the swap from machines to roles to machines happens internally in $\mathcal{T} = \mathcal{S} \circ \mathcal{S}_{\text{SYOSO}} \circ \mathcal{S}_{\text{UC}}$.

Similarly we can reduce the proof obligation for the static case to proving that for all $\mathcal{A}$ there exist $\mathcal{S}$ such that

$$\textsc{Exec}^{\textsc{static},\rho}_{\Pi^{\textsc{Natural}},\mathcal{A},\mathcal{E}} \approx \textsc{Exec}^{\text{SYOSO},\textsc{static},\tau}_{\Gamma^{\textsc{Abstract}},\mathcal{T},\mathcal{E}} \ .$$

The proofs for the two cases are very similar. To be able to reuse most of the proof structure we will initially do one proof that for all $\mathcal{A}$ there exist $\mathcal{S}$ such that

$$\textsc{Exec}^{\textsc{adaptive},\rho}_{\Pi^{\textsc{Natural}},\mathcal{A},\mathcal{E}} \approx \textsc{Exec}^{\text{SYOSO},\textsc{static},\tau}_{\Gamma^{\textsc{Abstract}},\mathcal{S},\mathcal{E}} \ .$$

Note that we attempt to prove adaptive security from static security.

5.5.1 Programming $\mathcal{F}_{\mathbf{RA}}$ in the Simulation

During the execution or simulation of $\textsc{Exec}^{\textsc{adaptive}}_{\Pi^{\textsc{Natural}},\mathcal{A},\mathcal{E}}$ let $\mathsf{Machine}^{\textsc{Cor}}$ denote the currently corrupt machines in $\mathsf{Machine}$ and let $\mathsf{Machine}^{\textsc{Hon}} = \mathsf{Machine} \setminus \mathsf{Machine}^{\textsc{Cor}}$. Let $\mathsf{Corrupt}$ be the statically corrupted roles in $\textsc{Exec}^{\text{SYOSO},\textsc{static},\tau}_{\Gamma^{\textsc{Abstract}},\mathcal{S},\mathcal{E}}$.

The main challenge in proving $\textsc{Exec}^{\textsc{adaptive},\rho}_{\Pi^{\textsc{Natural}},\mathcal{A},\mathcal{E}} \approx \textsc{Exec}^{\text{SYOSO},\textsc{static},\tau}_{\Gamma^{\textsc{Abstract}},\mathcal{S},\mathcal{E}}$ is that in $\textsc{Exec}^{\textsc{adaptive}}_{\Pi^{\textsc{Natural}},\mathcal{A},\mathcal{E}}$ we have chosen and adaptive corruptions, while in $\textsc{Exec}^{\text{SYOSO},\textsc{adaptive},\tau}_{\Gamma^{\textsc{Abstract}},\mathcal{S},\mathcal{E}}$ we have random and static corruptions. Note, however, that $\mathcal{A}$ expects to see a copy of $\mathcal{F}_{\text{RA}}$ which is not present in $\textsc{Exec}^{\text{SYOSO},\textsc{static},\tau}_{\Gamma^{\textsc{Abstract}},\mathcal{S},\mathcal{E}}$. So it is $\mathcal{S}$ which simulates $\mathcal{F}_{\text{RA}}$ to $\mathcal{A}$. We will use this to map roles $\mathsf{R} \in \mathsf{Role}^{\textsc{Cmp}} \cap \mathsf{Corrupt}$ to corrupt machines M and map $\mathsf{R} \in \mathsf{Role}^{\textsc{Cmp}} \cap \mathsf{Honest}$ to machines M which are honest at the time of execution.

There are some caveats to this, as we sometimes have to map corrupt roles to honest machines, but first we look at the basic simulation strategies.

5.5.2 Delayed Role Assignment

When the simulator $\mathcal{S}$ simulates $\mathcal{F}_{\text{RA}}$ it will simulate it exactly as in the protocol, with the following differences.

When keys are generated for $\mathsf{R} \in \mathsf{Role}^{\textsc{Comp}} \cap \mathsf{Honest}$ then it will not pick M_{R} until it is time for R to be executed. This is possible as the behavior of $\mathcal{F}_{\text{RA}}$ is independent of M_{R} to $\mathcal{A}$ when M_{R} is honest. We call this *delayed honest role assignment* below. Furthermore, when it is time for R to be executed, $\mathcal{S}$ will assign R to a uniformly random $\mathsf{M} \in \mathsf{Machine}^{\textsc{Hon}}$, i.e., a uniformly random machine M which is honest *at the time of execution*. Then $\mathcal{S}$ will simulate that M_{R} executes R as detailed below. Notice that this operation is atomic, so $\mathcal{E}$ cannot corrupt the simulated M_{R} while it executes R. And afterwards, when the simulated message is posted to the simulated $\mathcal{F}_{\text{RA}}$ a corruption of M_{R} will have to leak nothing new: in the real world protocol M_{R} would have deleted all internal state. And, importantly, note that when it is time for R to be executed, then $\mathcal{F}_{\text{RA}}$ delivered the Generate-message for R by construction. Therefore $\mathcal{F}_{\text{RA}}$ will not have to later hand the secret keys for R to $\mathcal{A}$ either. All in all, this ensure that when the execution of an honest R has been done it is (1) by an honest M_{R} at the time and (2) the secret keys sk_{R} and dk_{R} never have to be handed to $\mathcal{A}$.

When keys are generated for $\mathsf{R} \in \mathsf{Role}^{\textsc{Comp}} \cap \mathsf{Corrupt}$ then with some carefully calculated probability α the simulator $\mathcal{S}$ will pick M_{R} to be a uniformly random machine from $\mathsf{Machine}^{\textsc{Cor}}$, assign R to M_{R} immediately, and then output the secret keys to the adversary immediately as $\mathcal{F}_{\text{RA}}$ would do. With probability $1 - \alpha$ we instead from now on treat R as honest. We do delayed honest role assignment of R to some future honest machine and once M_{R} is chosen and R is executed the simulator just executes R correctly. This *corrupt-to-honest conversion* has the rationale that the current number of adaptively corrupted machines might be much smaller than the number of statically corrupted roles. It would therefore give a wrong distribution to assign all keys for corrupt roles to currently corrupted machines. Think of a single corrupted machine which would receive a fraction τ of all roles. We cannot somehow put R in the pocket and assign it to the next adaptively corrupt machine: this would give an odd distribution on role assignment where roles which already had keys generated hit newly corrupted machines too frequent. We therefore assign it uniformly at random to an honest machine. We can postpone this assignment as we do with postponed honest role assignment. Importantly, we never assign honest roles to currently corrupted machines. We return to how to simulate the correct distribution of role assignment below. We first show how to simulate the different combinations of role versus machines and corrupt versus honest.

5.5.3 Corrupt Role, Honest Machine (Case Orange)

Assume corrupted R is assigned to honest M_{R} and now $\mathcal{S}$ has to simulate the execution of R. This is the easy case. Later we call it the orange case. Since the role is corrupted, the simulator $\mathcal{S}$ received m_{R} and each $m_{\mathsf{S},\mathsf{R}}$ in $\textsc{Exec}^{\text{SYOSO},\tau}_{\Gamma^{\textsc{Abstract}},\mathcal{S},\mathcal{E}}$. It can then compute $(y, m_{\mathsf{R}}, \{(\mathsf{S}, m_{\mathsf{R},\mathsf{S}})\}_{\mathsf{S}\in\mathcal{O}_{\mathsf{R}}}) \leftarrow \mathsf{ExecRole}(\mathsf{ActSeq}, x, \{(\mathsf{S}, m_{\mathsf{S},\mathsf{R}})\}_{\mathsf{S}\in\mathcal{I}_{\mathsf{R}}})$ and output $(m_{\mathsf{R}}, \{(\mathsf{S}, m_{\mathsf{R},\mathsf{S}})\}_{\mathsf{S}\in\mathcal{O}_{\mathsf{R}}})$ in $\textsc{Exec}^{\text{SYOSO},\tau}_{\Gamma^{\textsc{Abstract}},\mathcal{S},\mathcal{E}}$ to simulate the execution of an honest role R. Then let M_{R} send encrypted and signed $(m_{\mathsf{R}}, \{(\mathsf{S}, m_{\mathsf{R},\mathsf{S}})\}_{\mathsf{S}\in\mathcal{O}_{\mathsf{R}}})$ in the simulated $\textsc{Exec}^{\text{ADAPTIVE}}_{\Pi^{\textsc{Natural}},\mathcal{A},\mathcal{E}}$.

5.5.4 Corrupt Role, Corrupt Machine (Case Red)

To simulate a corrupt role R executed by a corrupt machine M_{R} the simulator proceeds as follows. Later we call this the red case. Recall that M_{R} is controlled by $\mathcal{A}$ in the simulated $\textsc{Exec}^{\text{ADAPTIVE}}_{\Pi^{\textsc{Natural}},\mathcal{A},\mathcal{E}}$. The simulator runs $\mathcal{A}$ until M_{R} posts $(\mathsf{R}, m_{\mathsf{R}}, M, \sigma)$ on $\mathcal{F}_{\text{RA}}$ (or a timeout is called on behalf of M_{R}). If a message is posted before all $\mathsf{S} \in \mathcal{O}_{\mathsf{R}}$ were ready to receive then let

$m_{\mathsf{R}} = \textsc{NoMsg}$ and let $m_{\mathsf{R},\mathsf{S}} = \textsc{NoMsg}$ for all $\mathsf{S} \in \mathcal{O}_{\mathsf{R}}$. The protocol has the same behavior as it ignores messages posted before S was ready to execute, which it is not unless all its receivers are ready to receive. Otherwise, use the secret key sk_{S} of each $\mathsf{S} \in \mathcal{O}_{\mathsf{R}}$ and decrypt $m_{\mathsf{R},\mathsf{S}}$ from M as S would have done in the protocol. Then output $(m_{\mathsf{R}}, \{(\mathsf{S}, m_{\mathsf{R},\mathsf{S}})\}_{\mathsf{S} \in \mathcal{O}_{\mathsf{R}}})$. Note that we here might decrypt under the secret key of an honest receiver S. However, this could have been done via a decryption oracle. And, the receiver S will decrypt messages from R under label R, so we never use the oracle to decrypt ciphertexts for S sent by an honest R'. We can therefore assume that these are still indistinguishable.

5.5.5 Honest Role, Corrupt Machine (Case Cyan)

Due to delayed honest role assignment this case will not happen.

5.5.6 Honest Role, Honest Machine (Case Green)

When an honest role R is to be executed by an honest M_{R} then $\mathcal{S}$ proceeds as follows. Later we call this the green case. Learn m_{R} from $\textsc{Exec}^{\text{SYOSO},,\tau}_{\Gamma^{\textsc{Abstract}},\mathcal{S},\mathcal{E}}$. For $\mathsf{S} \in \mathcal{O} \cap \mathsf{Corrupt}$ learn $m_{\mathsf{R},\mathsf{S}}$ from $\textsc{Exec}^{\text{SYOSO},,\tau}_{\Gamma^{\textsc{Abstract}},\mathcal{S},\mathcal{E}}$ and let $m'_{\mathsf{R},\mathsf{S}} = m_{\mathsf{R},\mathsf{S}}$. For $\mathsf{S} \in \mathcal{O} \cap \mathsf{Honest}$ learn $|m_{\mathsf{R},\mathsf{S}}|$ from $\textsc{Exec}^{\text{SYOSO},,\tau}_{\Gamma^{\textsc{Abstract}},\mathcal{S},\mathcal{E}}$ and let $m'_{\mathsf{R},\mathsf{S}} = 1^{|m_{\mathsf{R},\mathsf{S}}|}$. Then let M_{R} send encrypted and signed $(m_{\mathsf{R}}, \{(\mathsf{S}, m'_{\mathsf{R},\mathsf{S}})\}_{\mathsf{S} \in \mathcal{O}_{\mathsf{R}}})$ in the simulated $\textsc{Exec}^{\text{ADAPTIVE}}_{\Pi^{\textsc{Natural}},\mathcal{A},\mathcal{E}}$. Notice that if $m'_{\mathsf{R},\mathsf{S}} \neq m_{\mathsf{R},\mathsf{S}}$ then S is honest. As argued above we never need to decrypt the ciphertext $M'_{\mathsf{R},\mathsf{S}}$ in the simulation, so we could ask a challenge oracle for the IND-CCA game to encrypt us $m'_{\mathsf{R},\mathsf{S}}$ or $m_{\mathsf{R},\mathsf{S}}$ at this point.

Notice that here we send an encryption of $1^{|m_{\mathsf{R},\mathsf{S}}|}$ to the currently honest machine S. If M_{S} was to become corrupted before the role R was executed this would be a problem as $\mathcal{A}$ would have to be given sk_{S} and would see $1^{|m_{\mathsf{R},\mathsf{S}}|}$ instead of $m_{\mathsf{R},\mathsf{S}}$. Note however that since $\mathsf{S} \in \mathcal{O}_{\mathsf{R}}$ and R was just executed, it follows that S has not yet been in turn to be executed. Therefore S is per *delayed honest role assignment* not yet assigned to a machine M. More importantly, when S is later executed $\mathcal{S}$ will pick a machine M_{S} which is honest at the time.

5.5.7 Getting the Right Distribution of Role Assignment, Static from Static Case

Above we argued that by a reduction to IND-CCA the above simulation cases can be shown indistinguishable from the execution. We now argue that we can simulate the correct distribution of role assignment while only using the above simulation cases. We should ensure that the number of roles assigned to machines has the correct distribution. At the same time we should not assign unexecuted honest roles to currently corrupted machines. We *are* allowed to assign corrupted roles to honest machines.

As a warm up, let us first consider the case where there are ρ static corruptions of computation machines in $\textsc{Exec}^{\text{ADAPTIVE},\rho}_{\Pi^{\textsc{Natural}},\mathcal{A},\mathcal{E}}$. We will play two balls in bins games; the machines $\mathsf{Machine}^{\textsc{Cmp}}$ will be the bins, and the roles $\mathsf{Role}^{\textsc{Cmp}}$ will be the balls. Call $\mathsf{Machine}^{\textsc{Red}} = \mathsf{Machine}^{\textsc{Cor}} \cap \mathsf{Machine}^{\textsc{Cmp}}$ the red machines. Call $\mathsf{Machine}^{\textsc{Green}} = \mathsf{Machine}^{\textsc{Hon}} \cap \mathsf{Machine}^{\textsc{Cmp}}$ the green machines. Let $R = |\mathsf{Role}^{\textsc{Cmp}}|$ and let $M = |\mathsf{Machine}^{\textsc{Cmp}}|$.

Consider the following process P_1 for coloring some roles $\mathsf{R} \in \mathsf{Role}^{\textsc{Cmp}}$ as red or green. First throw roles $\mathsf{R} \in \mathsf{Role}^{\textsc{Cmp}}$ uniformly at random into machines $\mathsf{Machine}^{\textsc{Cmp}}$. Then color R red if $\mathsf{R} \in \mathsf{Machine}^{\textsc{Red}}$ and otherwise color it green. Notice that this is how the protocol colors roles. It is the machines which are corrupt. Roles are assigned to machines, and roles inherit the color from the machine.

Consider then the following process P_2 for coloring some roles $\mathsf{R} \in \mathsf{Role}^{\textsc{Cmp}}$ as red or green. First assign R yellow balls uniformly at random to machines $\mathsf{Machine}^{\textsc{Cmp}}$. Call the balls in red machines red and call the balls in green machines green. Let r be the number of red balls. By assumption on ρ and τ it will always be the case that $R\tau \geq r$, except with negligible probability. Pick $R\tau - r$ green balls and color them orange. Now there are $R\tau$ green balls and $R(1-\tau)$ balls which are red or orange. Then color τR uniformly random roles $\mathsf{R} \in \mathsf{Role}^{\text{CMP}}$ as red and color the rest as green. Then for all computation roles $\mathsf{R} = \mathsf{R}_1, \ldots, \mathsf{R}_R$ in some possibly adversarially chosen order, proceed as follows. If R is green, then find a uniformly random green ball and swap the ball with R. If R is red, find a uniformly random red or orange ball and replace it by R. If R is swapped with an orange role, color R green.

Notice that here the roles get colors before being placed in machines. Then we place them at random but under the condition that green roles are placed in green machines. This is possible as $R\tau - r$ is positive. These two processes clearly give the same distribution of roles unto machines and both ensure that all roles R in red machines are corrupted. This means that $\mathcal{S}$ can assign roles to machines as follows.

First we let $\mathcal{S}$ mentally throw R yellow balls uniformly at random into machines $\mathsf{Machine}^{\text{CMP}}$. Call the balls in red machines red and call the balls in green machines green. Let r be the number of red balls. Pick $R\tau - r$ green balls and color them orange. Then it learns the τ corrupted computation roles R from $\text{EXEC}^{\text{SYOSO},\tau}_{\Gamma^{\text{ABSTRACT}},\mathcal{S},\mathcal{E}}$. Then for all computation roles $\mathsf{R} = \mathsf{R}_1, \ldots, \mathsf{R}_R$ in the activation order of Γ^{ABSTRACT}, proceed as follows. If R is honest, then find a uniformly random green ball in a green machine M, remove the green ball, let $\mathsf{M}_\mathsf{R} = \mathsf{M}$ and simulate M_R executing R using Case Green. If R is corrupt, then find a uniformly random red or orange ball in a machine M, remove the ball from M, let $\mathsf{M}_\mathsf{R} = \mathsf{M}$ and simulate M_R executing R. This means simulating a corrupt role on a corrupt machine (Case Red) or simulating a corrupt role on an honest machine (Case Orange). This completes the proof for the static-static case.

5.5.8 Getting the Right Distribution of Role Assignment, Adaptive from Adaptive Case

We now turn our attention to the case of simulating chosen adaptive corruptions of machines from adaptive random point-corruptions of roles. The simulator proceeds as $\mathcal{S}$ in the above case except that it does not know the set $\mathsf{Corrupt}$ statically, it will define this set adaptively using point-corruptions. First we let $\mathcal{S}$ throw R yellow balls uniformly at random into machines $\mathsf{Machine}^{\text{CMP}}$. As above this is to ensure that the distribution of roles unto machines has the right distribution. Now we want to map honest roles to honest machines and corrupt roles to corrupt machines.[11]

When a role R is about to get keys in the simulation, the simulator needs to pick a machine M for R. The simulator knows that it will swap R for a random yellow ball. It will count the current number of yellow balls in currently red machines (call these the red balls) and the current number of yellow balls in currently green machines (call these the green balls). Then it calculates the probability α that if R is swapped with a random yellow ball then R will become red, this is just the fraction of red balls to red plus green balls. Then $\mathcal{S}$ asks for a point corruption of R with probability α. If R becomes corrupted, then swap R with a random yellow ball in a red machine. If R becomes honest, then later swap R with a random yellow ball in a green machine, but do not choose the machine yet. Do postponed honest role assignment. The reason why we do postponed honest role assignment is that if we pick M_R now, then M_R might become corrupted before we can execute R. However, the role R will remain honest forever after the point corruption judged it to be honest.

This obviously gives the correct distribution of roles unto machines and ensure that we can

[11]In the adaptive case we do not need to assign corrupt roles to honest machines as we do not need to work with a surplus of corrupt roles.

simulate with Case Green and Case Red. There are only two things left to handle. (1) We need to argue that $\alpha \leq \tau$ in the point corruptions, and (2) in the adaptive case we get the additional case to simulate when a *role* R moves from honest to corrupted.[12] We handle this case first. Notice that if a role R moves from honest to corrupted, it is because $\mathcal{S}$ just did a point corruption of R. So R is about to be executed. This means that all $\mathsf{S} \in \mathcal{I}_{\mathsf{R}}$ already sent $M_{\mathsf{S},\mathsf{R}} = \mathsf{Enc}.\mathsf{Enc}^{\mathsf{S}}(\mathsf{ek}_{\mathsf{R}}, 1^{|m_{\mathsf{S},\mathsf{R}}|})$ to R while R was honest. Now that R becomes adaptively corrupted the simulator learns $m_{\mathsf{S},\mathsf{R}}$ and has to simulate the internal state of R to $\mathcal{A}$ as R became corrupted *before* executing. This is where we use that the encryption scheme is non-committing. When given $m_{\mathsf{R},\mathsf{S}}$ the simulator patches sk_{R} to make $M_{\mathsf{S},\mathsf{R}}$ decrypt to $m_{\mathsf{S},\mathsf{R}}$. Then it shows sk_{R} to $\mathcal{A}$.

We then argue that $\alpha \leq \tau$. Recall that we compute α as the current fraction of red balls to red plus green balls. Consider the points in the simulation where the set of corrupted servers increases. There are at most ρM such points and M is polynomial in the security parameter. Consider the fixed set of corrupted machines at such a point in time. Even if the size of this set is at a maximal ρM then it follows from how we pick ρ and τ that if we throw R yellow balls at random then less than a fraction τ will hit the corrupt machines except with negligible probability. Then do a union bound over these polynomially many points in time. The negligible probability is not affected by this polynomial union bound. This completes the proof for the adaptive-adaptive case.

References

[1] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*, pages 483–501. Springer, 2012.

[2] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10. ACM, 1988.

[3] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a public blockchain keep a secret? In Rafael Pass and Krzysztof Pietrzak, editors, *Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part I*, volume 12550 of *Lecture Notes in Computer Science*, pages 260–290. Springer, 2020.

[4] Erica Blum, Jonathan Katz, Chen-Da Liu Zhang, and Julian Loss. Asynchronous byzantine agreement with subquadratic communication. *IACR Cryptol. ePrint Arch.*, 2020:851, 2020.

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

[12] Recall that we already handled the case where a honest *machine* becomes corrupted using delayed role assignment. We ensure to never assign a future honest role to a machine which later becomes corrupted by delayed honest role assignment. Honest roles which have not yet been executed are not assigned to machines. And once they are to be executed we execute them on an honest machine and then delete their state.

[6] Ignacio Cascudo and Bernardo David. SCRAPE: Scalable randomness attested by public entities. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17*, volume 10355 of *LNCS*, pages 537–556. Springer, Heidelberg, July 2017.

[7] Ignacio Cascudo and Bernardo David. ALBATROSS: publicly attestable batched randomness based on secret sharing. *IACR Cryptol. ePrint Arch.*, 2020:644, 2020.

[8] Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 777:155–183, 2019.

[9] Seung Geol Choi, Dana Dachman-Soled, Tal Malkin, and Hoeteck Wee. Improved non-committing encryption with applications to adaptively secure protocols. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2009.

[10] Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. Fluid MPC: secure multiparty computation with dynamic participants. *IACR Cryptol. ePrint Arch.*, 2020:754, 2020.

[11] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 280–299. Springer, Heidelberg, May 2001.

[12] Ivan Damgård, Mads Jurik, and Jesper Buus Nielsen. A generalization of paillier's public-key system with applications to electronic voting. *Int. J. Inf. Sec.*, 9(6):371–385, 2010.

[13] Ivan Damgård and Maciej Koprowski. Practical threshold RSA signatures without a trusted dealer. In Birgit Pfitzmann, editor, *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding*, volume 2045 of *Lecture Notes in Computer Science*, pages 152–165. Springer, 2001.

[14] Ivan Damgård and Jesper Buus Nielsen. Adaptive versus static security in the UC model. In Sherman S. M. Chow, Joseph K. Liu, Lucas Chi Kwong Hui, and Siu-Ming Yiu, editors, *Provable Security - 8th International Conference, ProvSec 2014, Hong Kong, China, October 9-10, 2014. Proceedings*, volume 8782 of *Lecture Notes in Computer Science*, pages 10–28. Springer, 2014.

[15] Juan A. Garay, Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. The price of low communication in secure multi-party computation. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 420–446. Springer, Heidelberg, August 2017.

[16] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In Brian A. Coan and Yehuda Afek, editors, *17th ACM PODC*, pages 101–111. ACM, June / July 1998.

[17] Craig Gentry, Shai Halevi, Bernardo Magri, Jesper Buus Nielsen, and Sophia Yakoubov. Random-index PIR with applications to large-scale secure MPC. "ePrint report 2020/1248", 2020.

[18] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 581–612. Springer, Heidelberg, August 2017.

[19] Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, January 2000.

[20] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 477–498. Springer, Heidelberg, March 2013.

[21] Silvio Micali. Very simple and efficient byzantine agreement. In Christos H. Papadimitriou, editor, *ITCS 2017*, volume 4266, pages 6:1–6:1, 67, January 2017. LIPIcs.

[22] Pratyay Mukherjee and Daniel Wichs. Two round multiparty computation via multi-key FHE. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 735–763. Springer, Heidelberg, May 2016.

[23] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[24] Rafael Pass and Elaine Shi. The sleepy model of consensus. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 380–409. Springer, Heidelberg, December 2017.

[25] Tal Rabin. Robust sharing of secrets when the dealer is honest or cheating. *J. ACM*, 41(6):1089–1109, 1994.

[26] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In David S. Johnson, editor, *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*, pages 73–85. ACM, 1989.

[27] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In *21st ACM STOC*, pages 73–85. ACM Press, May 1989.

A Details of the YOSO Model

The intended purpose of our YOSO model is to have an abstract model of YOSO protocols where the discovered techniques apply to a variety of real-world application domains. In normal MPC the standard model has been one with a fixed set of so-called parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ connected by ideal authenticated channels or secure channels, and possibly with a broadcast channel. This abstract model did not concern itself with how these roles met, who channels were set up between them *et cetera.*

A lot of the many constructive techniques and proof techniques developed in MPC research over the past three decades were developed in this abstract standard model. To actually run an MPC protocol in practice a lot of independent details need to be considered. However, details like on how setting up a PKI in practice and using it to establish secure channels are entire research areas unto themselves. Yet, MPC has developed largely independently of the research on PKI and key exchange. We believe that there is a need for a similar clean model for developing MPC techniques applicable to the variety of blockchain inspired settings. Ideally the model should invite participation from MPC researchers who do not know or care about what a blockchain is and how it works.

A.1 Role Assignment versus Role Execution

The model is targeted at protocols where pieces of a protocol are executed by random machines and where no one knows who these machines are before they execute their role. After that, some new machine is chosen at random to execute the next piece of the protocol. Adding the next block to a Nakamoto style consensus is a prominent example; if a machine speaks in such a protocol it never has to speak again, i.e., it does not need to keep any secret state. We therefore call such protocol YOSO (you only speak once).

We will now dive a bit into the YOSO-style protocols and describe our model. Protocols in the YOSO model have two main components.

Role Assignment: Some mechanism allows a chosen physical entity to take on a particular role, like extending on a given block, producing a block for a given slot in proof-of-stake consensus, or being party number 7 in round number 42 in a player-replaceable model. Other mechanisms involve volunteering and permissioned assignment of roles, and nomination. The techniques going into this aspect are often largely independent of those going into how the actual protocol works.

Role Execution: Once a physical entity was assigned the right to execute a role and received the necessary messages, it will send out a set of messages in one atomic step. These messages are designed to obtain some complex goal like reaching agreement or running an MPC. This aspect is capturing the more abstract logic of how the protocol works.

It appears that even though these components are often entangled in current presentations of such protocols, the actual techniques for realizing each component can be developed in parallel and later be adapted to work together. Ideally, we should have a single clean abstraction of role acquisition, which could be implemented by one line of research and then be used for implementing a variety of abstract protocols by other teams.

There are many subtle details of YOSO-type protocols which are hard to capture in one simple model. An example is a detail such as "when can a role be spoken to?" To send a message to an unknown machine, the other machines need to learn the public key of the receiver machine, and these public keys typically become gradually available by some role assignment protocol. Sometimes the role execution itself can influence future role assignments; e.g., in a blockchain proof-of-stake mechanism the lottery might depend on how much stake machines have, and the MPC might be executing a ledger layer allowing stake to be shifted. Other protocols have other entangled aspects. It is therefore impossible to make an abstraction which is both simple and that allows to analyse all practical protocols. When we had to make a choice, we opted for simplicity, for the reasons discussed above. This means that the YOSO model cannot be used to modularly analyse all practical protocols. Rather, the main intended purpose is to allow theoretical studies and discover general techniques which can be used as ingredients in other practical constructions and theoretical results. Before describing the details of our model, we first motivate our design choices next.

A.2 Natural versus Abstract YOSO

There are two different approaches one could take to modeling YOSO protocols in the UC framework. We call them the natural model and the abstract model.

In the natural model we explicitly represent both physical machines and abstract roles. The roles are assigned at random to machines and the adversary is not told which machine has which role until the protocol execution reveals this. The adversary can make chosen corruptions of machines as is usual in UC, but since it does not know which machine has which role these chosen corruption of machines are *de facto* random corruptions of roles. In this model the parties of the UC model are identified with the machines, i.e., a UC party identifier pid names a machine. There is then a mechanism to inform machines which role they have been assigned, and to enforce that they only execute the assigned role(s).

In the abstract model there are no machines, only roles. An advantage is that we abstract away the unnecessary notion of physical machines. In this model we identify the parties of the UC model with the roles, i.e., identifier names a role in the protocol. A problem which arises here is that the adversary knows the party identifier pid and can choose to corrupt a specific pid. We therefore lose the property that the physical location of the hardware executing a role is hidden from the adversary. We fix this by introducing a new type of corruption where the adversary gets to corrupt a random party and where this randomness is out of the control of the adversary. This gives the same effect as in the natural model, where a random role is corrupted, but now without explicitly modeling the physical machines.

A.3 A Case Against Natural UC YOSO Model

We will later go for an abstract model, but to be able to compare the two options we now sketch a natural model and discuss some drawbacks. Let us call the model the Natural YOSO (NYOSO) model.

We cast the NYOSO model within the UC model. We describe a mechanism for decoupling the machine which executes a given role from the name of the role. The goal is to hide from the environment and adversary which machine executed a given role. This will ensure that the environment/adversary may choose to corrupt a specific machine but cannot a priori choose to corrupt a given role. We will let the name of the machines in our model be a party identifier pid, i.e., we identify machines with parties. We then introduce machinery which allows machines to execute some given secret roles. To allow composition and allow the same pid to play the same role R across several protocols we will use an ideal functionality ARA for Abstract Role Assignment. It will sample role assignments at random and keep track of which pid is assigned to a given role R,[13] to later inform parties which roles they are assigned to. Other ideal functionalities can also ask the ARA whether a given pid is allowed to execute a given role R. Corrupted parties can do what they want.

YOSO protocols and YOSO ideal functionalities will be specified obliviously of this ARA. When specifying a YOSO protocol we take the pid to be the role R. Also, YOSO ideal functionalities take inputs for roles and gives outputs for roles. To run these up against an ARA we therefore use wrappers. As an example, for a YOSO ideal functionality $\mathcal{F}$, the wrapper $\mathcal{F}' = \mathsf{YoS}(\mathcal{F})$ runs as follows: If it gets an input from pid wanting to given input m on behalf of role R, then the wrapper consults with ARA to determine whether pid was assigned role R. If this is the case then $\mathcal{F}$ is given input m as if coming from party identifier R. In particular, the ideal functionality $\mathcal{F}$ is oblivious of who is executing role R. Similarly outputs from $\mathcal{F}$ to role R are handed to the assigned party pid after consulting with ARA.

We now highlight an important drawback of natural models. Recall that the UC theorem says that if γ emulates $\mathcal{G}$ and π is a protocol for the $\mathcal{G}$-hybrid model emulating $\mathcal{F}$, then $\pi^{\mathcal{G}\to\gamma}$ also emulates $\mathcal{F}$. Here $\pi^{\mathcal{G}\to\gamma}$ is the protocol π with calls to $\mathcal{G}$ replaced by calls to γ. We recall some technical steps from the proof of the UC theorem. When we want to show the security

[13]This ARA could be a global ideal functionality or a normal one, this is not important for the present discussion.

of $\pi^{\mathcal{G}\to\gamma}$ in environment $\mathcal{E}$ then we create a new environment $\mathcal{E}_\pi$ which internally runs all of π except that calls to $\mathcal{G}$ are made to the hybrid functionality $\mathcal{G}$ in the execution. Then we show that running $\mathcal{E}$ with π gives the same effect as running $\mathcal{E}_\pi$ with $\mathcal{G}$ and we show that running $\mathcal{E}$ with $\pi^{\mathcal{G}\to\gamma}$ gives the same effect as running $\mathcal{E}_\pi$ with γ. It then follows from γ emulating $\mathcal{G}$ that $\pi^{\mathcal{G}\to\gamma}$ emulates $\mathcal{F}$.

Assume we try the same proof strategy to prove a composition theorem for YOSO protocols. In the composed protocol $\pi^{\mathcal{G}\to\gamma}$ a party with party identifier pid will be composed of two machines, P^{pid} running as part of π and Q^{pid} running as part of γ.

Notice that the part P^{pid} in the copy of π in $\mathcal{E}' = \mathcal{E}_\pi$ needs to know which role it executes. So we need to give $\mathcal{E}'$ access to ARA. But if we give environments access to ARA they can learn which machines run which roles, defying the purpose of having sampled the role assignment at random. There is seemingly no way around this. After all, the party P^{pid} in π running role R must know which role it executes to be able to execute the correct code. Even if P^{pid} is not explicitly told what role it runs, its input-output behavior towards $\mathcal{G}$ will *a priori* reveal which program it is running, which most likely reveals information on its role. Therefore it will be known to $\mathcal{E}'$ which role P^{pid} is executing in π. This leaks to $\mathcal{E}'$ which role the corresponding party Q^{pid} is executing in γ. Now $\mathcal{E}'$ can go and corrupt the specific role it is interested in.

This problem seems inherent in proving general composition of natural YOSO models. The problem is not insurmountable, but one has to create NYOSO models with classes of environments where some have access to ARA and some do not, or define that environments only use access to ARA in “benign” ways. Both directions lead to significant changes to the UC model and complicated models defying the elegance of having general composition. We could then just say that we do not want to do modular analysis of YOSO protocols. This would defy the purpose of having a UC model of YOSO protocols of course.

A.4 The (Abstract) YOSO Model

As argued above, natural YOSO models are cumbersome for modular analysis. We therefore propose the methodology of developing, composing and analyzing YOSO protocols in an abstract model where only roles are considered, and then compiling the final protocol to a natural model if/when they are to be run with a given role assignment mechanism. This allows to do simple composable analysis of YOSO protocols as the ”role level” without caring about how roles will later be assigned. We show a toy example of such a compilation in Section 5. We now go into the details of our UC YOSO model.

We consider a finite set of roles Role, which might depend on the security parameter κ. For the sake of the abstract model, we can think of the roles as descriptions of roles in the protocol, as e.g. “party 7 in round 8”. We use R to denote a generic role $\mathsf{R} \in \mathsf{Role}$ and often write $\mathsf{R}_1, \dots, \mathsf{R}_n$ when we have an ordered sequence of roles, for instance a committee. Roles are reminiscent of the abstract names $\mathsf{P}_1, \dots, \mathsf{P}_n$ which are often used in MPC; they are just abstract names to which we can later associate a program.

We identify Role with the set of party identifiers pid in the UC model, and the role names are hence passed around as the pid in UC.

Random Corruptions. We allow the same classes of corruptions as the UC model. However, we extend with a notion of random corruptions. To be able to give precise statements about what a random corruption is we introduce the corruption control (CC) component $\mathcal{CC}$. It is a poly-time ITI. It is meant to codify in a precise manner statements about proving security only against some restricted class of corruptions.

The corruption control component is only meant to ever talk to a UC environment $\mathcal{E}$, so we describe the behavior of $\mathcal{CC}$ in the context of an environment $\mathcal{E}$. They will form a single entity $\mathcal{E}^{\mathcal{CC}}$ called a controlled environment. Recall that the UC framework has a corruption aggregation component. It allows the environment to learn which corruptions have been performed. We will use *corruption information* to denote the information the environment would get by querying the corruption aggregation component. The $\mathcal{CC}$ allows the following two types of interactions with $\mathcal{E}$.

Judgement: The environment might query $\mathcal{CC}$ with $(\texttt{legal?}, ci)$, where ci is the current corruption information. In response it returns a judgement $\top$ or $\bot$. This judgement should be a monotone safety property: If there has not been any corruptions yet, then it returns $\top$. Furthermore, if the current corruption information ci makes $\mathcal{CC}$ return $\bot$ in its current state, then ci and all future ci' will make $\mathcal{CC}$ return $\bot$ if there are no future negotiations. As a simple example of a $\mathcal{CC}$ consider a $\mathcal{CC}$ parameterized by a set of n parties and which returns $\top$ iff less than $n/2$ of these parties have ever been corrupted.

Negotiation: The environment might query $\mathcal{CC}$ with $(\texttt{negotiate}, aux)$ with some auxiliary information aux. In response the $\mathcal{CC}$ will return some bitstring a. A negotiation might change which ci's are legal in the future. As a simple example, let's say the $\mathcal{CC}$ has the behavior that all roles R start out with status `untested`. If queried with $(\texttt{legal?}, ci)$, it will return $\top$ iff all R which were ever corrupted have status `corruptable`. On input $(\texttt{negotiate}, \mathsf{R})$ where R has status `untested` it samples a uniformly random bit. If it comes out 0, then R gets status `honest`. Otherwise R gets status `corruptable`. The $\mathcal{CC}$ then replies with a being the new status of R to let the adversary know the result of the negotiation.

When run in the UC experiment the controlled environment $\mathcal{E}^{\mathcal{CC}}$ runs as a proxy for $\mathcal{E}$, with the following differences.

1. When $\mathcal{E}$ makes its guess b (its output bit for the UC execution), then $\mathcal{E}^{\mathcal{CC}}$ inputs $(\texttt{legal?}, ci)$ to $\mathcal{CC}$ for the current ci. If the reply is $\top$, then $\mathcal{E}^{\mathcal{CC}}$ terminates with guess b. If the reply is $\bot$ the controlled environment samples a uniformly random bit b' and terminates with guess b'.

2. If the adversary sends $(\texttt{negotiate}, aux)$ to $\mathcal{E}^{\mathcal{CC}}$, then $\mathcal{E}^{\mathcal{CC}}$ first sends $(\texttt{legal?}, ci)$ to $\mathcal{CC}$ for the current ci. If the reply is $\bot$ then $\mathcal{E}^{\mathcal{CC}}$ terminates with a random guess in the UC experiment, as above. If the reply is $\top$, then it sends $(\texttt{negotiate}, aux)$ to $\mathcal{CC}$ and sends the reply a back to the adversary.

The purpose of the random guess b' is to ensure that if the adversary makes an illegal corruption, then the environment outputs the same distribution in the real world and the simulation. Therefore any distinguishing advantage of a controlled environment comes from an execution with legal corruptions. The reason for first querying on $(\texttt{legal?}, ci)$ is to ensure that a negotiation does not turn an already illegal corruption legal. If also provides $\mathcal{CC}$ with the current ci for the negotiation.

Recall that during the proof of the UC theorem, an environment $\mathcal{E}_\pi$ is defined which internally runs all of π except that calls to $\mathcal{G}$ are made to the hybrid functionality $\mathcal{G}$ in the execution. It is easy to see that if $\mathcal{E}$ is a controlled environment then $\mathcal{E}_\pi$ is a controlled environment. In particular, $(\mathcal{E}^{\mathcal{CC}})_\pi = (\mathcal{E}_\pi)^{\mathcal{CC}}$. The same holds for the case of pulling a simulator into an environment. Therefore we get UC composition for the class of controlled environments.

We can now use the $\mathcal{CC}$ to define and limit random corruptions. We do not put any further restrictions on how this is done. The CC can also be used to negotiation other types of corruption than random corruptions of course. Two types of corruptions we will consider later are sample corruptions and point corruptions.

By a *sample corruption* over a set Role of roles we mean that the adversary submits a poly-time distribution C on Role. Then $\mathcal{CC}$ samples $\mathsf{R} \in \mathsf{Role}$ using C and now allows R to be corrupted. We can let $\mathcal{CC}$ restrict the distributions and total corruptions. We might for instance say that the corruptions must be uniform on the currently uncorrupted R in Role and that at most a fraction τ of the roles may be corrupted in total.

By a *point corruption* of $\mathsf{R} \in \mathsf{Role}$ we mean that the adversary submits R and a probability ρ. Then $\mathcal{CC}$ flips a bit which is 1 with probability ρ. We require that this can be done in poly-time. If the bit is 1 then R is considered corruptible. Otherwise it is consider honest forever after. We can let $\mathcal{CC}$ restrict the queries. We might for instance say that the corruptions must not be with probability above 49% and that each R might be negotiated at most once.

You Only Speak Once. Another key aspect of our model is that we want to consider protocols where each party only sends messages at one point in time. This again gives some challenges with composition. Consider a protocol π for the $\mathcal{G}$-hybrid model. Consider then a protocol $\pi^{\mathcal{G}\to\gamma}$ where we replace $\mathcal{G}$ by a protocol γ. Roles R in $\pi^{\mathcal{G}\to\gamma}$ is a composition of two parties, the part P^{R} acting in π and the part Q^{R} acting in γ. Let us denote the composed role by $R^{\mathsf{R}} = P^{\mathsf{R}} \circ Q^{\mathsf{R}}$. If at some point during a call from π to γ the protocol γ has Q^{pid} send a message, then this counts as R^{R} having sent a message. Therefore P^{R} cannot later send a message. However, when π is in the $\mathcal{G}$-hybrid model, then a call from π goes to $\mathcal{G}$ and hence there is no notion of some Q^{R} having sent a message. Therefore P^{R} *would* now be allowed to later send a message. This gives different behavior when running with $\mathcal{G}$ and γ, which leads to problems with modular analysis.

For composition to work out we need to introduce some bookkeeping about when a party has already "spoken" inside an ideal functionality $\mathcal{G}$. And for γ to emulate $\mathcal{G}$ we need that they "speak" at the same time. This will allow to forbid P^{R} to speak in the future at the correct times also in the $\mathcal{G}$-hybrid model.

We use a very simple mechanism for this. All communication takes place via ideal functionalities. We introduce a special token SPOKE which ideal functionalities can return to parties. If a party P^{R} gets output SPOKE from $\mathcal{G}$ then we say that R *spoke* in $\mathcal{G}$.

When a part in a composed party like $R^{\mathsf{R}} = P^{\mathsf{R}} \circ Q^{\mathsf{R}}$ gets input SPOKE from a sub-party or its outer protocol, then it inputs SPOKE to all its (other) sub-parties in a canonical order and then returns SPOKE to its outer protocol. This ensures that if some component in a composed party "spoke" then all other components are made aware of this. After this the party will ignore all future inputs and never give output again. We enforce this behavior using a wrapper $\mathsf{YoS}(P)$ which will normally forward all inputs and outputs to and from P, but which in response to a SPOKE token will pass it around to all other wrappers as specified and then start ignoring inputs by immediatly returning SPOKE on all future inputs; we say that $\mathsf{YoS}(P)$ *crashed*. For this purpose the wrapper keeps track of all sub-routines that P took part in and will inform them about future SPOKE tokens. This means that P will never see the token. Once it spoke it will just never get activated again. Notice that $\mathsf{YoS}(P \circ Q) = \mathsf{YoS}(P) \circ \mathsf{YoS}(Q)$, which we use in the YOSO composition theorem later. Recall that in UC the sub-routines can be created dynamically. This leaves the corner case where a future sub-routine created by another party contacts $\mathsf{YoS}(P)$ after it crashed. By design $\mathsf{YoS}(P)$ returns SPOKE in response to these unsolicited sub-routine contacts.

Notice that by design, if a role $\mathsf{YoS}(P)$ gets input SPOKE from the environment $\mathcal{E}$, then $\mathsf{YoS}(P)$ will crash. This allows the environment to do a trivial denial of service attack on a protocol by crashing all roles. This looks like a problem, but it is foremost a necessary feature.[14] It makes sense for some sub-routines to learn that the role it is part of spoke in some other part of the protocol. To avoid unintended denial of service attacks, one can as usual choose to analyse protocols in a restricted class of environments, for instance environments which never inputs SPOKE to an internal honest role of a protocol. As a consequence one can of course later only compose the protocol into an outer protocol with this property. That means that honest outer protocols should not unintentionally speak on behalf of an internal role of an inner protocol. This can be ensured by proper scoping of roles during protocol design. Such bookkeeping is an unfortunate but necessary part of doing modular analysis of YOSO protocols.

Notice that a corrupted party is free to not run the YoS wrapper. So corrupted parties can send multiple message and keep talking to ideal functionalities after speaking. This also holds for adaptive corruption. If an honest party spoke and then crashed, then it is so to say uncrashed if adaptively corrupted. The adversary can now interact with the ideal functionalities that the honest party used in the past.

A.5 YOSO Security and The Composition Theorem

We define security via the UC definition of security. For simplicity we stick to the case of securely implementing an ideal functionality. We use $\pi \leq^{\mathrm{UC}} \mathcal{F}$ to denote that π UC securely implements $\mathcal{F}$ against the class of controlled environments. Recall briefly that it means that for all PPT $\mathcal{A}$ there exists a PPT $\mathcal{S}$ such that $\textsc{Exec}_{\pi,\mathcal{A},\mathcal{E}} \approx \textsc{Exec}_{\mathcal{F},\mathcal{S},\mathcal{E}}$ for all PPT $\mathcal{E}$ controlled environments. When we consider information theoretic security we allow $\mathcal{E}$ to be unbounded.

Definition 4 (YOSO Security). *Let π be a protocol for the UC model and let $\mathcal{F}$ be an ideal functionality. We say that π YOSO securely implements $\mathcal{F}$ if $\mathsf{YoS}(\pi) \leq^{UC} \mathcal{F}$. We write $\pi \leq^{YOSO} \mathcal{F}$. We say that the protocol IT YOSO securely implements $\mathcal{F}$ if security holds for unbounded environments and with statistical indistinguishability.*

Theorem 4 (YOSO Composition). *Let π and γ be protocols for the UC model. Assume that π runs in the $\mathcal{G}$ hybrid model. If $\pi \leq^{YOSO} \mathcal{F}$ and If $\gamma \leq^{YOSO} \mathcal{G}$ then $\pi^{\mathcal{G}\to\gamma} \leq^{YOSO} \mathcal{F}$.*

Proof. From $\pi \leq^{\mathrm{YOSO}} \mathcal{F}$ we get that $\mathsf{YoS}(\pi) \leq^{\mathrm{UC}} \mathcal{F}$. From $\gamma \leq^{\mathrm{YOSO}} \mathcal{G}$ we get that $\mathsf{YoS}(\gamma) \leq^{\mathrm{UC}} \mathcal{G}$. From the UC theorem we get that $\mathsf{YoS}(\pi)^{\mathcal{G}\to\mathsf{YoS}(\gamma)} \leq^{\mathrm{UC}} \mathcal{F}$. Now observe that by construction $\mathsf{YoS}(\pi)^{\mathcal{G}\to\mathsf{YoS}(\gamma)} = \mathsf{YoS}(\pi^{\mathcal{G}\to\gamma})$. This gives us that $\mathsf{YoS}(\pi^{\mathcal{G}\to\gamma}) \leq^{\mathrm{UC}} \mathcal{F}$, which implies that $\pi^{\mathcal{G}\to\gamma} \leq^{\mathrm{YOSO}} \mathcal{F}$, □

A.6 Synchronous YOSO

We now give a model of synchronous YOSO protocols. We base it on the model from [20]. All parties and ideal functionalities are aware which round they are currently in by accessing a global clock functionality. The exact details of the model are not important to the exposition here, and the interested reader is encouraged to consult [20]. The crucial point is that the model allows the computation to proceed in synchronous rounds where in each round each honest party gets to give an input. Between rounds each honest party has ample time to compute the next message, i.e., polynomial time. If the corrupted parties do not give an input in a round, a dummy one is enforced. This way, crashed or malicious parties cannot deadlock the computation and crashes can be detected.

[14]There is no reasonable way around this "problem". A role Q does not know when it talks to the environment or a role R in an outer protocol. And when it talks to an outer protocol we need it to respect a SPOKE token to get $\mathsf{YoS}(P \circ Q) = \mathsf{YoS}(P) \circ \mathsf{YoS}(Q)$. Another way to look at it is that if P can crash a sub-party Q, then when P is pulled into the environment during the proof of the UC theorem, then the environment should be allowed to crash Q when P instructs this.

A.6.1 Some Ideal Functionalities

We now proceed to specify our computation model by giving some ideal functionalities for communicating. Before that we give some conventions for specifying YOSO ideal functionalities.

We assume that all inputs to ideal functionalities are of the form $(c, \mathsf{R}, \cdots)$ where c is the command type, R the role (party identifier) of the role who gave the input, and the rest is the payload. We will not be explicit about session identifiers.

We assume that $\mathcal{F}$ is parametrized by a set of roles $\mathcal{F}.\mathsf{Role} \subset \mathsf{Role}$. There is a "promise" that an implementation of $\mathcal{F}$ will not touch other roles than these in $\mathcal{F}.\mathsf{Role}$. The ideal functionality can further specify input, computation and output roles, $\mathcal{F}.\mathsf{Role}^{\text{IN}}, \mathcal{F}.\mathsf{Role}^{\text{CMP}}, \mathcal{F}.\mathsf{Role}^{\text{OUT}} \subset \mathcal{F}.\mathsf{Role}$, where

$$\mathcal{F}.\mathsf{Role} = \mathcal{F}.\mathsf{Role}^{\text{IN}} \cup \mathcal{F}.\mathsf{Role}^{\text{CMP}} \cup \mathcal{F}.\mathsf{Role}^{\text{OUT}} \ .$$

We require that $(\mathcal{F}.\mathsf{Role}^{\text{IN}} \cup \mathcal{F}.\mathsf{Role}^{\text{OUT}}) \cap \mathcal{F}.\mathsf{Role}^{\text{CMP}} = \varnothing$. The input roles and output roles might overlap. We think of $\mathsf{R} \in \mathsf{Role}^{\text{IN}}$ as those roles taking inputs, $\mathsf{R} \in \mathsf{Role}^{\text{COMP}}$ as those doing the internal computation, and $\mathsf{R} \in \mathsf{Role}^{\text{OUT}}$ as those giving outputs.

To take part in the protocol, computation roles need to send messages, so we expect that roles in $\mathsf{Role}^{\text{CMP}}$ will speak during an implementation of $\mathcal{F}$. We will therefore enforce that they output SPOKE. We do not require the same for input roles, as we might want to have ideal functionalities where input roles do not communicate. An example could be an ideal functionality for signature schemes. Output roles on the other hand will also have associated restrictions. In any protocol, the only way a role can contribute information to the future execution is by sending a message.[15] However, in YOSO protocols you can only speak once. Therefore output roles should not speak during the implementation of a protocol, as they might have to speak in the outer protocol to contribute the output to a larger computation. All in all, we have the following conventions.

External Roles: $\mathcal{F}$ will ignore all inputs from $\mathsf{R} \notin \mathcal{F}.\mathsf{Role}$ and never give outputs to $\mathsf{R} \notin \mathcal{F}.\mathsf{Role}$, neither normal output nor SPOKE. This ensures that $\mathcal{F}$ does not "pollute" other roles which should have been used by other protocols. This allows to scope the role space.

Computation Roles: When $\mathcal{F}$ is initialized it will give an adversarially delayed output of SPOKE to all $\mathsf{R} \in \mathcal{F}.\mathsf{Role}^{\text{CMP}}$. When using $\mathcal{F}$ in a hybrid model, this ensures that an outer protocol should not expect to use $\mathsf{R} \in \mathcal{F}.\mathsf{Role}^{\text{CMP}}$. When using $\mathcal{F}$ in the simulation as the ideal functionality being implemented, it allows the simulator to output SPOKE on behalf of all computation roles at the same time as the role output SPOKE in the computation being simulated.

Input Roles: $\mathcal{F}$ never outputs SPOKE to $\mathsf{R} \in \mathcal{F}.\mathsf{Role}^{\text{IN}}$ until R gave an input. It might or might not output SPOKE to R after an input was given.

Output Roles: $\mathcal{F}$ never gives output SPOKE to $\mathsf{R} \in \mathcal{F}.\mathsf{Role}^{\text{OUT}} \setminus \mathcal{F}.\mathsf{Role}^{\text{IN}}$.

Generic Computation Nodes: We say that the computation roles are generic if the behavior of $\mathcal{F}$ does not depend on $\mathsf{Role}^{\text{CMP}}$. This means that $\mathcal{F}$ does not look at the inputs to $\mathsf{R} \in \mathsf{Role}^{\text{CMP}}$, does not depend on whether they are corrupted or not, and do not depend on the value of the $\mathsf{Role}^{\text{CMP}}$, except that it outputs SPOKE to all $\mathsf{R} \in \mathcal{F}.\mathsf{Role}^{\text{CMP}}$. In particular, we can specify $\mathcal{F}$ without knowing the set of computation nodes. When the computation roles are generic we do not explicitly mention $\mathsf{Role}^{\text{COMP}}$. We think of it as

[15]In a synchronous models this is not technically true. One can send a bit b at time t by sending a PING if $b = 1$ and sending nothing if $b = 0$. Still the role needs to have the *option* to send a message to be able to send information.

The ideal functionality is conventionally YOSO with $\mathsf{Role}^{\mathrm{IN}} = \mathsf{Role}^{\mathrm{OUT}}$ and in addition has the following behavior.

- Initially create a map $y : \mathbb{N} \times \mathsf{Role} \to \mathsf{Msg}_\perp$ with $y(r, \mathsf{R}) = \perp$ for all r, R. Below we use $y(r, \cdot)$ to denote the map $y' : \mathsf{Role} \to \mathsf{Msg}_\perp$ with $y'(\mathsf{R}) = y(r, \mathsf{R})$.
- On input $(\textsc{Send}, \mathsf{R}, x_\mathsf{R} \in \mathsf{Msg})$ in round r proceed as follows:
  1. Update $y(r, \mathsf{R}) = x_\mathsf{R}$. *Store inputs of the round.*
  2. Output $(\mathsf{R}, x_\mathsf{R})$ to $\mathcal{S}$. *Leak messages in a rushing fashion.*

  If R is honest the command is only executed if $y(r, \mathsf{R}) = \perp$. If R is honest then give output SPOKE to R.
- On input $(\textsc{Read}, \mathsf{R}, r')$ in round r where $r' < r$ output $y(r, \cdot)$ to R.

Figure 11: The IF $\mathcal{F}_{\mathsf{BC}}$ for broadcast.

The ideal functionality is conventionally YOSO with $\mathsf{Role}^{\mathrm{IN}} = \mathsf{Role}^{\mathrm{OUT}}$ and in addition has the following behavior.

- Initially create a map $y : \mathbb{N} \times \mathsf{Role} \times \mathsf{Role} \to \mathsf{Msg}_\perp$ with $y(r, \mathsf{R}, \mathsf{S}) = \perp$ for all $r, \mathsf{R}, \mathsf{S}$.
- On input $(\textsc{Send}, \mathsf{S}, \mathsf{R}, x_{\mathsf{S},\mathsf{R}} \in \mathsf{Msg})$ in round r proceed as follows:
  1. Update $y(r, \mathsf{R}, \mathsf{S}) = x_{\mathsf{S},\mathsf{R}}$. *Store inputs of the round.*
  2. Output $(\mathsf{S}, \mathsf{R}, |x_{\mathsf{S},\mathsf{R}}|)$ to $\mathcal{S}$. *Leak the sending of a message in a rushing fashion.*

  If S is honest the command is only executed if $y(r, \mathsf{R}, \mathsf{S}) = \perp$. If S is honest then *at the end of the round* give output SPOKE to S. If R is corrupt (or later corrupted) then output $x_{\mathsf{S},\mathsf{R}}$ to $\mathcal{S}$ (at the time of corruption).
- On input $(\textsc{Read}, \mathsf{R}, r')$ in round r where $r' < r$ output $y(r, \mathsf{R})$ to R.

Figure 12: The IF $\mathcal{F}_{\mathrm{SPP}}$ for secure message transmission on point-to-point channels.

being provided later to match the computation nodes of the protocol π implementing $\mathcal{F}$. This conveniently allows us not to have to specify up front what computation roles a protocol implementing $\mathcal{F}$ will use.

Ideal functionalities can output to the adversary or simulator and receive inputs back. We use $\mathcal{S}$ to name the entity that the ideal functionalities talks to, be it a simulator or adversary. We call an ideal functionality that meets all these conventions *conventionally YOSO*. Note that these conventions are not enforced by the YOSO model. We just give them a name to not have to mention them every time we specify a conventional ideal functionality.

We now give the ideal functionalities for broadcast and secure message transmission. We work with a finite message space Msg and has a special symbol $\perp \notin \mathsf{Msg}$. Let $\mathsf{Msg}_\perp = \mathsf{Msg} \cup \{\perp\}$. The ideal functionality for broadcast is given in Figure 11. Notice that $\mathcal{F}_{\mathsf{BC}}$ outputs SPOKE to R the first time R sends a message. This models the YOSO behavior that the role R can send messages in at most one round.

The ideal functionality for secure message transmission is given in Figure 12. Notice that $\mathcal{F}_{\mathrm{SPP}}$ outputs SPOKE to S after the first round when S sends a message. This models the YOSO behavior that the role R can send messages in at most one round. It *does* allow that a role sends

The ideal functionality is conventionally YOSO and in addition has the following behavior.

- Initially set the stage to be GettingInputs. We set a default input for all roles, which they may overwrite later: for all $\mathsf{R} \in \mathsf{Role}^{\mathrm{IN}}$ let $x_{\mathsf{R}} = 0$.
- On input $(\textsc{Input}, \mathsf{R} \in \mathsf{Role}^{\mathrm{IN}}, x \in \mathsf{Msg})$ proceed as follows:
  1. Store $x_{\mathsf{R}} = x$.
  2. If $\mathsf{R} \in \mathsf{Honest}$ then output $(\textsc{Input}, \mathsf{R}, |x|)$ to $\mathcal{S}$.
  3. If or when $\mathsf{R} \in \mathsf{Leaky} \cup \mathsf{Malicious}$ output $(\textsc{Input}, \mathsf{R}, x)$ to $\mathcal{S}$.

  If R is honest then output Spoke to R. If R is honest then consider only the first input of the form $(\textsc{Input}, \mathsf{R}, \cdot)$ and only consider this input if it is given in round 1.
- On Evaluated from $\mathcal{S}$ in a round $r > 1$ and when the stage is GettingInputs, set the stage to be Evaluated and compute $\{y_{\mathsf{R}}\}_{\mathsf{R} \in \mathsf{Role}^{\mathrm{OUT}}} = F(\{x_{\mathsf{R}}\})_{\mathsf{R} \in \mathsf{Role}^{\mathrm{IN}}}$. Store y_{R} for all $\mathsf{R} \in \mathsf{Correct}$ and output to $\mathcal{S}$ the value $\{y_{\mathsf{R}}\}_{\mathsf{R} \in \mathsf{Role}^{\mathrm{OUT}} \cap (\mathsf{Malicious} \cup \mathsf{Leaky})}$.
- On input $(\textsc{Read}, \mathsf{R} \in \mathsf{Role}^{\mathrm{OUT}})$ when the stage is Evaluated output y_{R} to R.

Figure 13: Functionality $\mathcal{F}_{\mathrm{MPC}}^{F}$, wrapping a function $F(x_1, \ldots, x_n) \to (y_1, \ldots, y_m)$. Roles in $\mathsf{Role}^{\mathrm{IN}}$ hold input values and roles in $\mathsf{Role}^{\mathrm{OUT}}$ receive output values.

messages to several other roles in the same round. It is possible to consider weaker forms of YOSO secure message transmission where only one (or a small constant) number of messages can be sent.

A.6.2 Synchronous YOSO MPC

As usual for UC-like models, to formulate the assertion that a function F could be computed securely we need to wrap that function by a ideal functionality $\mathcal{F}_{\mathrm{MPC}}^{F}$. The wrapper functionality for the YOSO model is depicted in Figure 13. The ideal functionality evaluates a function $F : (\mathsf{Role}^{\mathrm{IN}} \to \mathsf{Msg}) \to (\mathsf{Role}^{\mathrm{OUT}} \to \mathsf{Msg})$, i.e., given an input for each input role it provides an output for each output role. We formulate the ideal functionality for a setting where we might have both Byzantine and semi-honest corruptions. We use Malicious to denote the set of Byzantine or active corrupted roles, we use Leaky to denote semi-honest corruptions that behave correctly but leak their internal state, and finally we use Honest to denote the roles which are not corrupted. We let $\mathsf{Correct} = \mathsf{Leaky} \cup \mathsf{Honest}$. These are the roles computing the correct program, though some might be leaky.

We assume that all input roles get input in round 1. This means that the protocol starts running in round 1 and that all honest parties must provide their input there. If some input role did not get an input $x \in \mathsf{Msg}$ by round 1, it will simply use 0. The protocol can then go through two stages, GettingInputs and Evaluated. The stage can be GettingInputs for a long time, but only corrupted parties can give inputs in later rounds. This models that we need the honest parties to give input in the first round, but corrupted parties might only be committed to their inputs in a later round.

The adversary $\mathcal{S}$ decides when it is time to give outputs by inputting Evaluated. We assume that the output roles do not speak in an implementation. This is specified by letting $\mathcal{F}_{\mathrm{MPC}}^{F}$ not output Spoke for the output roles.

When studying YOSO MPC we will assume *chosen* corruptions of input roles and output roles and *random* corruptions of computation roles. This is to model a situation where the inputs to the computation are provided by some known parties. These parties might therefore be the target of a denial of service attack. Similarly for the parties who are to learn the outputs: these might well be some known servers. It is therefore reasonable to assume targeted corruption of the input roles and output roles. For the computation roles we assume they are assigned to unknown machines using some role assignment mechanism so we allow only random corruptions of computation roles. By a τ-secure YOSO MPC we mean one secure against any number of corruptions of parties in $\mathsf{Role}^{\text{IN}} \cup \mathsf{Role}^{\text{OUT}}$ and uniformly random corruptions of a fraction τ of the parties in $\mathsf{Role}^{\text{CMP}}$.

Definition 5 (YOSO MPC). *We say that π YOSO-securely realizes F against τ corruption if $\pi \leq^{YOSO} \mathcal{F}_{MPC}^{F}$ for the set of environments allowed to corrupt any number of roles in $\mathsf{Role}^{IN} \cup \mathsf{Role}^{OUT}$ and a uniformly random fraction τ of Role^{CMP}. We say that the protocol IT YOSO-securely realizes F if security holds for unbounded environments and with statistical indistinguishability.*

When considering τ-security with *adaptive* corruptions we can consider two types of corruption, namely *sampling* corruption and *point* corruption. Sampling corruption means that the adversary can ask for an additional corruption, and then a uniformly random uncorrupted $\mathsf{R} \in \mathsf{Role}^{\text{COMP}}$ will be corrupted; this is allowed until a fraction τ is corrupted. In point corruption the adversary can ask to corrupt a given $\mathsf{R} \in \mathsf{Role}^{\text{COMP}}$ and specify a probability $\alpha \leq \tau$, and then R is announced corrupt with probability α and honest with probability $(1-\alpha)$. The adversary cannot ask for the same R twice, so once R is announced honest it remains honest. It is of course possible to consider other variations of random, adaptive corruptions.

We can simulate sample corruptions from point corruptions. When sampling a new corrupted role, keep making random point corruptions with the right marginal probability until some R is announced corrupt and then return R. It does not seem that you can simulate point corruptions from sample corruptions. In fact, we conjecture that security against point corruption is strictly stronger. Note, in particular, that it seems stronger to have a protocol secure against point corruptions when using it in a bigger protocol. During simulation in the hybrid model for the protocol a simulator which has access to point corruptions might learn that a role will never become corrupted in the future. The simulator might conceivably exploit this. With sample corruptions any honest role might become corrupt at some future point. It is an interesting open problem to find a separating example.

A.6.3 Parameters of the Models

It is interesting to study restricted versions of the YOSO model to determine what features allow YOSO MPC with certain efficiency and security properties.

A.6.4 Horizons

With the $\mathcal{F}_{\mathsf{BC}}$ and $\mathcal{F}_{\text{SPP}}$ functionalities we proposed (Figure 11 and Figure 12 respectively), any role can send messages to any other role. In implementing role assignment it will typically not be the case that all roles are assigned immediately. In fact, there are several settings where this is not desirable, as roles should be assigned according to recent conditions: which machines are alive, which machines hold how much stake in the system *et cetera*. For the purpose of studying the impact of the scheduling of assignments on the possibility of YOSO MPC it is convenient to have a notion of *horizon*. For this we associate with each round a time t. It starts out being $t = 0$ and increases by 1 at the beginning of each round. Let $\mathsf{Time} = \mathbb{N}$.

Listening Schedule. A *listening schedule* is a function $\mathsf{Listening} : \mathsf{Time} \to \mathcal{P}(\mathsf{Role})$ which at each time describes which roles are "listening". We say that a protocol is $\mathsf{Listening}$-admissible if no honest party ever sends to a role which is not listening. Note that in practice "listening" might just mean that the public key of the role is known and that ciphertexts sent to the role can be deposited somewhere for later pick up, for instance on a blockchain. The machine (later) assigned the role might not need to be awake yet, in fact, the machine might not even need to know that it was assigned the role yet. For $\mathsf{R} \in \mathsf{Role}$ we let $\mathsf{Listening}(\mathsf{R}) = \{\ t \in \mathsf{Time} | \mathsf{R} \in \mathsf{Listening}(t)\}$. For simplicity we will assume the model is equipped with a listening schedule, by default the dummy one with $\mathsf{Listening} = \mathsf{Role}$.

Liveness Schedule. A *liveness schedule* is a function $\mathsf{Active} : \mathsf{Time} \to \mathcal{P}(\mathsf{Role})$ which at each time describe which roles may act. We say that a protocol is Active-admissible if an honest R only speaks when $\mathsf{R} \in \mathsf{Active}(t)$. For $\mathsf{R} \in \mathsf{Role}$ we let $\mathsf{Active}(\mathsf{R}) = \{\ t \in \mathsf{Time} | \mathsf{R} \in \mathsf{Active}(t)\}$. For simplicity we will assume the model is equipped with a liveness schedule, by default the dummy one with $\mathsf{Active} = \mathsf{Role}$.

Round Based. We say that a protocol is round based if each role is assigned to a particular round using a function $\mathsf{Round} : \mathsf{Time} \to \mathcal{P}(\mathsf{Role})$. We let $\mathsf{Role}_t = \mathsf{Round}(t)$ and assume that $\mathsf{Role} = \bigcup_{r=1}^{\infty} \mathsf{Role}_r$, where the union is disjoint. We let $\mathsf{Round}(\mathsf{R})$ be the t such that $\mathsf{R} \in \mathsf{Round}(t)$.

Future Horizon. For a round based protocol we define a future horizon and past horizon. We say a protocol has *future horizon* $\mathsf{fh} \in \mathbb{N}$ if

$$\mathsf{Listening}(t) = \bigcup_{h=t+1}^{t+\mathsf{fh}} \mathsf{Round}(t)\ .$$

Past Horizon. We say a protocol has *past horizon* $\mathsf{ph} \in \mathbb{N}$ if

$$\mathsf{Active}(t) = \bigcup_{h=t-\mathsf{ph}+1}^{t} \mathsf{Round}(t)\ .$$

The past horizon limits how long the machines assigned with a round has to linger to be able to act. In a serverless architecture where a machine is rented or spun up only when it needs to act, this controls how long the machine needs to be up. Note that past horizon 1 means that only roles in $\mathsf{Round}(t)$ may trigger. Future horizon 1 means that only $\mathsf{Round}(t+1)$ my receive messages. Hence $(\mathsf{ph}, \mathsf{fh}) = (1, 1)$ is the minimal meaningful model if information is to be passed from one round to the next using the point-to-point channels.

Broadcast Horizon. Note that a role can always post to the broadcast channel and any role can see all previous messages on the broadcast channel. The horizons so far only affect what can be sent on the point-to-point channels. In practice the future horizon mainly affects secret values, as authenticated values for future roles could in principle be stored on the broadcast channel. We might want to restrict the use of the broadcast channel as a storage medium. This can be done via the *broadcast horizon* $\mathsf{bh} \in \mathbb{N} \cup \{\infty\}$.

If bh is finite then a role may only read messages from $\mathcal{G}_{\mathsf{BC}}$ sent in the time interval $[t + 1 - \mathsf{bh}, t]$. If $\mathsf{bh} = 1$ a role can only see the messages broadcast in the previous round, i.e., the broadcast channel behaves as a single shot broadcast protocol. If $\mathsf{bh} = \infty$ the broadcast channel behaves as a ledger/persistent totally ordered broadcast channel which stores previous values and allows these to be retrieved when a role wakes up. If $\mathsf{bh} = 0$ there is no broadcast channel.

The above horizon notions are for a model with only point-to-point channels. If a PKI is included in the model, then the broadcast channel can be used to send secure message by dumping encryptions for future roles on the blockchain. In that case it does not make sense to consider a broadcast horizon longer than the future horizon.

A.6.5 Adaptive versus Static Triggering

We can also make a distinction between whether parties speak at fixed points in time or can be dynamic.

Public Triggering. We say that a protocol has *public triggering* if all roles can efficiently predict when all other roles will speak at least one round before they do so. Public triggering is useful in publicly verifying whether a role refused to execute its role. In a serverless setting it is helpful in knowing when to spin up a machine to execute a given role. If a role does not have public triggering, then the machine that should execute the role should in principle always be alive.

Static vs. Adaptive Triggering. We say that a protocol has *static triggering* if $\forall \mathsf{R} \in \mathsf{Role}\,(\,|\mathsf{Active}(\mathsf{R})| \leq 1\,)$. We say that a protocol has *adaptive triggering* if it does not have static triggering. Clearly static triggering implies public triggering. We conjecture that static triggering is weaker than adaptive public triggering.

Static Communication Pattern. We can restrict the model using a static communication pattern $\mathsf{Comm} : \mathsf{Role} \to \mathcal{P}(\mathsf{Role}) \times \mathcal{P}(\mathsf{Role})$, where $(R, S) = \mathsf{Comm}(\mathsf{P})$ restricts P to receive from $\mathsf{R}' \in R$ and send to $\mathsf{S} \in S$. For simplicity we assume all protocols have a communication restriction Comm, by default the dummy one $\mathsf{Comm} = (\mathsf{Role}, \mathsf{Role})$. We say that a protocol has out degree δ and in degree ι if it holds for all $\mathsf{R} \in \mathsf{Role}$ and $\mathsf{Comm}(\mathsf{R}) = (R, S)$ that $|R| \leq \iota$ and $|S| \leq \delta$. A basic assumption about the YOSO model is that DoS attacks are a threat and that a machine executing a role has time to send its message before being attacked. For this to hold in practice it could be important that the machine does not have to send an excessively large message (or many messages). In particular it can be interesting to limit the number of roles a given role can send to.

Note that if a model has static triggering and a static communication graph, there is no need for cycles in the communication graph as backwards edges will always point to roles which already spoke. If there is no broadcast channel the model in this case then strongly resembles a circuit model with the individual roles acting as gates. It is interesting to explore connections between YOSO MPC and notions of robust and leakage resilient circuits.

A.6.6 Complexity

As usual it is interesting to study communication complexity, both in terms of bits, messages, and number of active roles. A seemingly interesting parameter to study is *communication depth.* It is the longest chain of roles passing information along the chain. It is interesting to relate it to round complexity in the standard model of MPC.