Le Mans: Dynamic and Fluid MPC for Dishonest Majority

Rahul Rachuri[1], Peter Scholl[1]

Aarhus University, {rachuri, peter.scholl}@cs.au.dk

Abstract. Most MPC protocols require the set of parties to be active for the entire duration of the computation. Deploying MPC for use cases such as complex and resource-intensive scientific computations increases the barrier of entry for potential participants. The model of Fluid MPC (Crypto 2021) tackles this issue by giving parties the flexibility to participate in the protocol only when their resources are free. As such, the set of parties is dynamically changing over time.

In this work, we extend Fluid MPC, which only considered an honest majority, to the setting where the majority of participants at any point in the computation may be corrupt. We do this by presenting variants of the SPDZ protocol, which support dynamic participants. Firstly, we describe a *universal preprocessing* for SPDZ, which allows a set of n parties to compute some correlated randomness, such that later on, any subset of the parties can use this to take part in an online secure computation. We complement this with a *Dynamic SPDZ* online phase, designed to work with our universal preprocessing, as well as a protocol for securely realising the preprocessing. Our preprocessing protocol is designed to efficiently use pseudorandom correlation generators, thus, the parties' storage and communication costs can be almost independent of the function being evaluated.

We then extend this to support a *fluid online phase*, where the set of parties can dynamically evolve during the online phase. Our protocol achieves *maximal fluidity* and security with abort, similarly to the previous, honest majority construction. Achieving this requires a careful design and techniques to guarantee a small state complexity, allowing us to switch between committees efficiently.

1 Introduction

Secure multi-party computation (MPC) allows a set of parties to jointly compute a function on their inputs, while preserving privacy, that is, not revealing anything more about the inputs than can be deduced from the output of the function. MPC can be applied in a wide range of situations, including secure aggregation, private training or evaluation of machine learning models, threshold signing and more.

Most MPC protocols work under the assumption that the set of parties involved in the computation is fixed throughout the protocol. Although committee-based MPC and player-replaceability schemes have existed for a while, recently more practically oriented models have been proposed such as Fluid

MPC [CGG$^+$21] and YOSO [GHK$^+$21]. These models support protocols with a *dynamically evolving* set of parties, where participants can join and leave the computation as desired, without interrupting the protocol. This enables a more flexible model, where parties can sign up to contribute their resources towards a large-scale, distributed computation, without having to commit for the duration of the entire protocol. This is particularly important for large-scale, long-running tasks such as complex scientific computations, such as Folding@home. In the *maximally fluid* setting, this concept is pushed to the limit, where each participant is only required to sign up for *a single round* of the protocol. This gives the most possible flexibility for any server who may wish to participate.

The YOSO (you only speak once) paradigm [GHK$^+$21] also considers maximally fluid MPC protocols, with some differences in the model. Unlike Fluid MPC, they separately study the role assignment problem, where they show how to leverage a blockchain to randomly assign the committee of parties who will take part in each round. With their mechanism, the identity of any member of the current committee is only revealed after they have published their message. This allows for much stronger security guarantees, since an adversary has no way to identify which servers are involved in the computation — and hence who to corrupt — until the role played by the server has already been terminated.

Both of these works give information-theoretically secure protocols in the *honest majority* setting, where in any given round of the protocol, the majority of the computing parties should be honest. Fluid MPC achieves security with abort, where a malicious party can prevent the protocol from terminating, while YOSO achieves the stronger notion of guaranteed output delivery (but is less efficient).

1.1 Our Contributions

In this work, we study MPC with dynamically evolving parties in the *dishonest majority* setting. This gives much stronger security guarantees, since we only require that in any given round of the computation, there is at least one honest party taking part. However, it is also more challenging than honest majority. We now elaborate on our contributions and some technical background.

The challenge of fluidity and dishonest majority. In the dishonest majority setting, most practical MPC protocols are based on authenticated secret-sharing using information-theoretic MACs, such as in the SPDZ [DPSZ12] or BDOZ [BDOZ11] protocols. These protocols rely on a preprocessing phase, using more expensive, "public-key" style cryptography, to generate a large amount of correlated randomness that is consumed in a lightweight online phase. Unfortunately, this means that each party has to maintain a *large state* (the correlated randomness), the size of which grows linearly with the complexity of the function being computed. This is problematic for achieving Fluid MPC, since when changing from one committee of parties to another, the natural approach is to securely transfer the entire state to the new committee. Ideally, we want this state transfer process to be *independent* of the function being computed, to avoid the communication complexity blowing up.

Key Tool: Universal Preprocessing for Dynamic Parties. Before aiming for Fluid MPC, we look at a simpler model which allows just a single change in the set of computing parties during the protocol. We consider a *universal preprocessing* phase, where all of the parties $P_1, \dots, P_n$ who may wish to be involved in the computation must take part. Later, any subset of the n parties can get together and run a fast, online protocol, without having to interact with anybody else. We assume the inputs to the protocol are provided by the online subset of parties (though with standard techniques such as [DDN+16], we can also support inputs from external parties).

Recall that in SPDZ, the parties need to preprocess authenticated multiplication triples, denoted $[\![a]\!], [\![b]\!], [\![c]\!]$, where a and b are secret, random finite field elements and $c = a \cdot b$. These values are secret-shared with MACs, given by

$$[\![x]\!] := (x^i, m^i, \Delta^i)_{i \in [n]}$$

where party P_i has the share Δ^i of the global MAC key $\Delta = \sum \Delta^i$, and also the shares x^i, m^i, satisfying $x = \sum x^i$ and $x \cdot \Delta = \sum m^i$ over the field.

Instead of producing fully authenticated triples like this, we produce a weaker form of *partial triple*, where c is unauthenticated, and not fully computed: every pair of parties (P_i, P_j) will get a two-party additive sharing of $a^i \cdot b^j$. This suffices to reconstruct a share c^i, by adding up P_i's relevant sharings of $a^i b^j$, together with $a^i b^i$.

Importantly, this also enables *any subset* of parties $\mathcal{P} \subset [n]$ to obtain a triple, by restricting to the shares a^i, b^i for $i \in \mathcal{P}$, and summing up the relevant shares of the products to get a c^i for this committee. A similar trick also works to get the MACs on a and b, since each MAC is just a secret-shared product with the fixed key Δ. Therefore, it's enough to give out two-party shares of $a^i \Delta^j$ and $b^i \cdot \Delta^j$ for every $i \neq j$.

We show how to realize this type of preprocessing using simple, pairwise correlations between every pair of parties, in the form of oblivious linear function evaluation (OLE) and vector-OLE. We ensure correctness of the authenticated $[\![a]\!], [\![b]\!]$ shares using a consistency check, which we formalize via a multi-party vector-OLE functionality. However, our protocol does not guarantee correctness of the shares of cross-products $a^i \cdot b^j$. We therefore model these errors via adversarial influence in the preprocessing functionality.

PCG-Friendliness. An important feature of our preprocessing protocol is that it is *PCG-friendly*, meaning that it can be implemented using *pseudorandom correlation generators* (PCGs) [BCG+19b]. A PCG allows two parties to take a pair of short, correlated seeds, and expand them to produce a much larger quantity of correlated randomness. There are efficient PCGs for vector-OLE, based on variants of the LPN assumption [BCGI18,BCG+19a,WYKW21], and for OLE under a variant of ring-LPN [BCG+20]. By supporting PCGs in our preprocessing, we obtain communication and storage complexities as small as $O(n \log|C|)$ field elements per party, for an arithmetic circuit C. Prior to our work, we stress that even with a statically chosen online phase, there was no practical, multi-party SPDZ-like protocol[1] that could support a preprocessing phase with this feature with good concrete efficiency — ours is the first protocol to support this "silent" feature.

Dynamic Variant of SPDZ Online Phase. One issue with our universal preprocessing is that, since the c terms of triples are not authenticated, we cannot use the same online phase as SPDZ. Instead, we modify the online phase so that in each multiplication, we first authenticate c before using a triple to multiply. Since a malicious party may have introduced errors in c, we then need to add a *verification phase*, to check the multiplications are correct. We do this following the approach of Chida et al. [CGH+18] (also used by the honest majority Fluid compiler of [CGG+21]). Here, as well as computing the circuit, the parties compute a randomised version of the circuit, where each wire value has been multiplied by a secret, random value $r \in \mathbb{F}_p$. At the end of the computation, the parties run a batch verification process to check consistency of the two computations. We show that this guarantees our protocol is correct, even with our weaker preprocessing protocol which allows malicious parties to introduce special types of errors into c.

Overall, the communication cost of our dynamic online protocol is only 4 field elements on top of the SPDZ online phase [DPSZ12,DKL+13], which costs 2 elements. However, this comes with the benefits of (1) a dynamically chosen online committee, and (2) a PCG-friendly preprocessing phase, where each party's communication and storage complexity is almost independent of the circuit size.

Maximally Fluid Online Phase. We now turn to the harder task of obtaining an online phase where the set of computing parties can dynamically change. We focus on the most challenging goal of *maximal fluidity*, where in each round, a different committee can sign up to receive one message from the previous committee, before sending one message and going offline.

This brings additional obstacles when it comes to preprocessing data, as well as verifying MACs on opened values during the online protocol. The first hurdle is that, even though our universal preprocessing allows any committee to obtain a multiplication triple, these triples end up being authenticated under different MAC keys, depending on the committee.

As a first attempt to deal with this MAC key inconsistency, one could have the current committee, $\mathcal{P}_{\mathsf{curr}}$, securely *reshare* their current state of intermediate computation values, including their MAC key $\Delta_{\mathcal{P}_{\mathsf{curr}}}$, to the next committee, $\mathcal{P}_{\mathsf{next}}$. To proceed further, however, $\mathcal{P}_{\mathsf{next}}$ will need authenticated triples under the same MAC key. Our preprocessing phase, on the other hand, only allows them to obtain triples under a different key $\Delta_{\mathcal{P}_{\mathsf{next}}}$. To avoid this issue, $\mathcal{P}_{\mathsf{curr}}$ would instead have to reshare *all of* the triples needed for the rest of the circuit evaluation, after which, $\mathcal{P}_{\mathsf{next}}$ would use some of these, reshare to the next committee and so on. This incurs a huge blow up in communication cost, which we would like to avoid.

[1] In the two party setting, an efficient PCG-based SPDZ preprocessing protocol was given in [BCG+19b].

Our method for dealing with this is a secure *key-switching* procedure, which allows $\mathcal{P}_{\mathsf{curr}}$ to transfer a shared $[\![x]\!]$ to $\mathcal{P}_{\mathsf{next}}$ in a single round, while switching to $\mathcal{P}_{\mathsf{next}}$'s MAC key. Another constraint we have from the model is that $\mathcal{P}_{\mathsf{next}}$ cannot send any messages to $\mathcal{P}_{\mathsf{curr}}$. At first glance, it may seem impossible, since $\mathcal{P}_{\mathsf{curr}}$ should not have any information on the next key. However, we show that by leveraging the power of our universal preprocessing, key-switching can be done with just a single set of messages from $\mathcal{P}_{\mathsf{curr}}$ to $\mathcal{P}_{\mathsf{next}}$.

In addition to securely switching keys, another challenge in our maximally fluid protocol is how to check MACs on opened values. We cannot use the batched MAC check from SPDZ, since this involves storing a large state, which has to be passed around until the end of the protocol. Instead, we modify this to an incremental procedure, where only a constant-sized state needs to be transferred in each round. We adopt a similar incremental protocol to verify multiplications, where, as in our Dynamic SPDZ protocol, we use the same randomised circuit idea as [CGH+18].

1.2 Related Work

Bracha [Bra85] introduced the idea of using committees in distributed protocols with a large number of parties, which has been used in a number of MPC protocols since. One recent example is [GSY21], which constructs committee-based MPC when up to 1/3 of the parties may be corrupt, achieving a construction that scales to hundreds of thousands of parties. Although part of their protocol is based on SPDZ, they do not support the notion of a dynamically chosen subset of parties from the preprocessing set carrying out the online computation. Concretely, their online phase for circuit evaluation costs 7x higher than SPDZ, whereas we estimate that we only suffer a 3x overhead. A detailed analysis of the costs is provided in Section 6.

Another relevant work is [SSW17], which outsources SPDZ preprocessing to an external set of parties. However, unlike our protocol, this requires resharing the entire preprocessing data from the external set to the online committee. We avoid this in Dynamic SPDZ, by relying on our universal preprocessing.

The area of proactive security has long considered the notion of an adversary who can corrupt different parties throughout the computation. These works typically use a proactive secret sharing scheme, where secrets are maintained by an ever-changing set of parties. Works such as [HJKY95,MZW+19] show security in the presence of a mobile adversary that can corrupt and uncorrupt parties at different points in the protocol. More recently, [BGG+20,GKM+20] construct secret-sharing protocols for the case of honest majority with active security. The model used in these papers also splits the work done by each committee into two parts, one used to do the computation with parties interacting only within the committee, and one used to perform a secure state-transfer to the committee that comes after them. The primary difference between Fluid SPDZ and proactive MPC is the motivation and the behaviour of the adversary. In proactive schemes, the adversary typically operates with a "corruption budget" that limits the adversary from being able to corrupt parties arbitrarily. We do not make such an assumption, and our motivation primarily comes from giving parties in a computation the ability to drop in and out, while minimising the minimum number of rounds they have to stay on for. In addition, we try to achieve a small *state complexity*, so that switching committees is not communication intensive.

2 Preliminaries and Security Model

2.1 Preliminaries

We use κ as the security parameter and ρ as the statistical security parameter. Bold letters such as $\boldsymbol{a}$ are used to indicate vectors, and $\boldsymbol{a}[i]$ refers to the i-th element of the vector. We write $[a, b]$ to denote the set of natural numbers $\{a, \ldots, b\}$ and $[a, b) = \{a, \ldots, b-1\}$.

Additional Functionalities. We make use of some standard functionalities in the paper, which are detailed in Appendix A. These include a functionality for oblivious transfer $\mathcal{F}_{\mathsf{OT}}$, coin-tossing $\mathcal{F}_{\mathsf{Rand}}$, commitment $\mathcal{F}_{\mathsf{Commit}}$, and a weak equality test $\mathcal{F}_{\mathsf{EQ}}$, that checks equality of two private inputs, while always revealing one party's input to the adversary.

2.2 Modelling Fluid MPC in Dishonest Majority

The remainder of this subsection covers definitions pertaining to the Fluid model. Computation broadly proceeds in 4 phases – preprocessing, input, execution, and output. This is similar to that of Fluid MPC [CGG^{+}21], with the addition of a preprocessing phase, which is used to generate data-independent information in the form of multiplication triples, to be used in the execution phase. In the preprocessing phase, we require all parties who wish to take part in the computation at some later point to be active, and after this they may go offline. The execution phase proceeds in epochs, where each epoch runs among a fixed set of parties, or committee. An epoch contains two parts, the *computation phase*, where the committee performs some computation, followed by a *hand-off phase*, used to securely transfer the current state to the next committee.

Fluidity. The computation phase of each epoch may take several rounds of interaction. Fluidity is defined as the minimum number of rounds in any given epoch of the execution phase. We say that a protocol achieves *maximal fluidity* if the epoch only lasts for one round. This means each server in the committee does some local computation, before sending a single message to the next committee in the hand-off phase. In the input and output phases, we not not measure fluidity, instead, the committee may interact for several rounds to share inputs or reconstruct the outputs.

A server is said to be "active" in the computation if it either performs computations or sends and/or receives messages. Therefore, a server participating in epoch i is active starting from the hand-off phase of epoch $i-1$, until the end of the hand-off phase of epoch i.

Committee formation. The committees used in each epoch may be either fixed ahead of time, or chosen on-the-fly throughout the computation. Fixing them ahead of time can be useful, for instance, in a volunteer sign-up based model, where servers can volunteer to participate in any epoch, and stay on for any number of epochs depending on their resource constraints. On the other hand, choosing committees on-the-fly may be desirable in settings closer to the YOSO model [GHK$^+$21], where a role-assignment mechanism is used to ensure that the next committee is only revealed at the last possible moment.

In this work, we do not distinguish between these two cases, and instead simply require that during the hand-off phase of epoch i, the current committee, denoted $\mathcal{P}_i$, knows the identities of the parties in the next committee $\mathcal{P}_{i+1}$. We make no assumptions or restrictions about the overlap between committees. As in [CGG$^+$21], the formation process can be modelled with an ideal functionality that samples and broadcasts committees according to the desired mechanism.

Corruption. Our model allows all-but-one of the servers who are active at the start of any given epoch to be corrupted, where the set of corrupt parties is fixed at the beginning of the epoch. Formally, this corresponds to an R-*adaptive adversary* from [CGG$^+$21]. Here, at the beginning of epoch i with committee $\mathcal{P}_i$, the adversary may adaptively choose a set of servers in $\mathcal{P}_i$ to be corrupted, and then learns the entire state of each corrupted server in any prior epochs. For the duration of epoch i, this set of corrupted parties is then fixed and cannot change. To rule out the adversary learning information on prior epochs, a server S may be corrupted in epoch i only if this does not lead to any prior epoch j with committee $\mathcal{P}_j$ becoming entirely corrupt.

We use this model for the online phase of our fluid MPC protocol. Note that for our dynamic SPDZ protocol, where the online committee does not change, this corresponds to the more common notion of static security. In the preprocessing phase for both dynamic SPDZ and our fluid MPC protocol, we have only proven security against a static adversary. While for fluid MPC, we would ideally also like the preprocessing to be adaptively secure, this is particularly challenging in the dishonest majority setting, and is known to imply strong primitives like non-committing encryption. In fact, since no practical adaptively secure preprocessing protocols are even known for the standard SPDZ protocol [DPSZ12], we view this as an interesting open problem.

2.3 Security Model

To model fluid MPC, we adopt the arithmetic black box model (ABB), which is an ideal functionality $\mathcal{F}_{\mathsf{ABB}}$ in the universal composability framework [Can01]. The functionality allows for a set of parties $P_1, \ldots, P_n$ to input their values, perform computations on them, and receive the outputs. The functionality is parameterised by a finite field $\mathbb{F}_p$, and supports native operations of addition and multiplication in the field.

We instantiate $\mathcal{F}_{\mathsf{ABB}}$ with the Dynamic SPDZ protocol ($\Pi_{\mathsf{SPDZ\text{-}Online}}$), which uses a preprocessing phase between a set of parties, and supports a dynamically chosen subset to perform the online phase. The preprocessing phase is used to set up partially authenticated, partially formed triples using pairwise MACs similar to BDOZ [BDOZ11] and TinyOT [HSS17]. We adapt the vector OLE from Wolverine [WYKW21], and PCGs from [BCG$^+$19a] and use them to form the partial triples.

To model Fluid MPC, we modify $\mathcal{F}_{\mathsf{ABB}}$ to support computations with dynamic committees, as functionality $\mathcal{F}_{\mathsf{DABB}}$ in Fig. 1. The main difference is that now, the functionality keeps track of the currently active committee in a variable $\mathcal{P}_{\mathsf{curr}}$. In operations which are part of the execution phase, where the committee may change, the functionality receives the identity of the next committee from the currently active parties (if it receives inconsistent inputs, we assume it aborts). In our protocol, the **Batch Multiply** command is the only part of the execution phase with interaction, so this is where any changes in committee might take place. We have $\mathcal{P}_{\mathsf{curr}}$ provide the next committee $\mathcal{P}_{\mathsf{next}}$ as input, and then wait for another message from $\mathcal{P}_{\mathsf{next}}$, who will provide a subsequent committee $\mathcal{P}'_{\mathsf{next}}$. This is because our multiplication protocol takes place over two rounds, so it inherently allows up to two committee changes whenever it is called (if we want to support maximal fluidity).

In practice, with our protocol it is possible to interleave multiplications, so that a new multiplication can be started before the old one has finished (reducing round complexity). However, for simplicity, we do not model this in $\mathcal{F}_{\mathsf{DABB}}$.

We instantiate $\mathcal{F}_{\mathsf{DABB}}$ with a Fluid Online ($\Pi_{\mathsf{Fluid\text{-}Online}}$) protocol. It extends the model of Fluid MPC [CGG$^+$21] which only works for the honest majority case, to the dishonest majority setting with active security. It uses the same preprocessing phase as Dynamic SPDZ, but the online phase supports committees switching. Parties can leave the computation by securely transferring their state to the subsequent committee, and rejoin the computation at a later point.

3 Universal Preprocessing for Dynamic Committees

In this section, we present the preprocessing phase used in our two online protocols. Our main design goals are (1) to allow a flexible and dynamic choice of participants during the online phase, and (2) to obtain a silent preprocessing phase, where the storage and communication complexities are (almost) independent of the function being computed. The section is organised in a top-down manner, where we start by describing an ideal preprocessing functionality, and then gradually explain our protocol for realising it.

Overview. In Fig. 2, we present an overview of the functionalities and protocols used for the preprocessing. In this section, we focus on realising $\mathcal{F}_{\mathsf{Prep}}$, using variants of oblivious linear function evaluation (OLE), as well as how to realise a multi-party variant of vector-OLE ($\mathcal{F}_{\mathsf{nVOLE}}$). Some of the remaining building blocks we use to implement this are deferred to Appendix D.

Functionality $\mathcal{F}_{\mathsf{DABB}}$

Parameters: Finite field $\mathbb{F}_p$, and set of parties $\mathcal{P}_{\mathsf{main}} = \{P_1, \ldots, P_n\}$. The functionality assumes all parties have agreed upon public identifiers id_x, for each variable x used in the computation. For a vector $\boldsymbol{x} = (x_1, \ldots, x_m)$, we write $\mathsf{id}_{\boldsymbol{x}} = (\mathsf{id}_{x_1}, \ldots, \mathsf{id}_{x_m})$.

Initialise: On input $(\mathsf{Init}, \mathcal{P}_{\mathsf{curr}})$ from P_i, for $i \in [1, n]$, where each P_i sends the same set $\mathcal{P}_{\mathsf{curr}} \subset \mathcal{P}_{\mathsf{main}}$, initialise $\mathcal{P}_{\mathsf{curr}}$ as the first active committee.

Input: On input $(\mathsf{Input}, \mathsf{id}_x, x)$ from some $P_i \in \mathcal{P}_{\mathsf{main}}$, and $(\mathsf{Input}, \mathsf{id}_x)$ from all parties in $\mathcal{P}_{\mathsf{curr}}$, store the pair (id_x, x).

Add: On input $(\mathsf{Add}, \mathsf{id}_{\boldsymbol{z}}, \mathsf{id}_{\boldsymbol{x}}, \mathsf{id}_{\boldsymbol{y}})$ from P_i, for every $P_i \in \mathcal{P}_{\mathsf{curr}}$, compute $\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{y}$ and store $(\mathsf{id}_{\boldsymbol{z}}, \boldsymbol{z})$.

Batch Multiply: On input $(\mathsf{Mult}, \mathcal{P}_{\mathsf{next}}, \mathsf{id}_{\boldsymbol{z}}, \mathsf{id}_{\boldsymbol{x}}, \mathsf{id}_{\boldsymbol{y}})$ from every $P_i \in \mathcal{P}_{\mathsf{curr}}$:

- Compute $\boldsymbol{z} = \boldsymbol{x} * \boldsymbol{y}$.
- Update $\mathcal{P}_{\mathsf{curr}} := \mathcal{P}_{\mathsf{next}}$.
- Wait to receive a message $(\mathsf{MultFinish}, \mathcal{P}'_{\mathsf{next}})$ from every $P_i \in \mathcal{P}_{\mathsf{curr}}$. Then, store the batch of products $(\mathsf{id}_{\boldsymbol{z}}, \boldsymbol{z})$ and update $\mathcal{P}_{\mathsf{curr}} := \mathcal{P}'_{\mathsf{next}}$.

Output: On input $(\mathsf{Output}, \mathsf{id}_{\boldsymbol{z}})$ from every $P_i \in \mathcal{P}_{\mathsf{curr}}$, where $\mathsf{id}_{\boldsymbol{z}}$ has been stored previously, retrieve $(\mathsf{id}_{\boldsymbol{z}}, \boldsymbol{z})$ and send it to the adversary. Wait for input from the adversary, if it is $\mathsf{Deliver}$, send the output to every $P_i \in \mathcal{P}_{\mathsf{curr}}$. Otherwise, abort.

Fig. 1: Functionality for a dynamic arithmetic black box

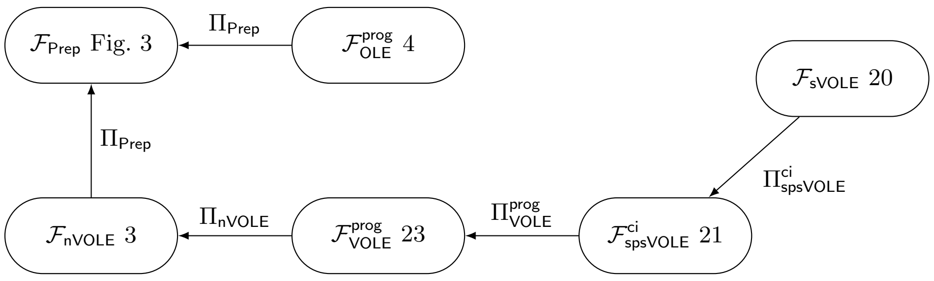
$\mathcal{F}_{\mathsf{Prep}}$ Fig. 3
Π_{Prep}
$\mathcal{F}^{\mathsf{prog}}_{\mathsf{OLE}}$ 4
$\mathcal{F}_{\mathsf{sVOLE}}$ 20
Π_{Prep}
$\Pi^{\mathsf{ci}}_{\mathsf{spsVOLE}}$
$\mathcal{F}_{\mathsf{nVOLE}}$ 3
Π_{nVOLE}
$\mathcal{F}^{\mathsf{prog}}_{\mathsf{VOLE}}$ 23
$\Pi^{\mathsf{prog}}_{\mathsf{VOLE}}$
$\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ 21

Fig. 2: Preprocessing Flow

3.1 Preprocessing Functionality

Let $\mathcal{P}_{\mathsf{main}} = \{P_1, \ldots, P_n\}$ be the set of all parties who may want to participate in the online phase.

Authenticated Secret Sharing. For the preprocessing, we use two kinds of secret sharing. $[x]$ denotes that $x \in \mathbb{F}_p$ is additively shared between the parties, that is, $x = x^1 + \ldots + x^n$ where P_i holds x^i. We also use pairwise authenticated shares, indicated by $\langle x \rangle$. Here, in addition to an additive share of x, each party holds an information-theoretic MAC on their share with every other party, who holds a corresponding MAC key. The MAC of P_i's share x^i under P_j's key is defined as $M^i_j = K^j_i + \Delta^j \cdot x^i$, where P_i holds the MAC M^i_j and P_j holds the local key K^j_i as well as the global key Δ^j (which is fixed for all MACs). While the shares x^i lie over the field $\mathbb{F}_p$, we allow MAC keys and MACs to be in an extension field $\mathbb{F}_{p^r}$, giving a forgery probability of p^{-r}, in case p is not large enough for the desired statistical security level.

If x is only shared between a smaller committee $\mathcal{P}_C \subset \mathcal{P}_{main}$, we write $[x]^{\mathcal{P}_C}$. Similarly, for pairwise MACs, we can consider a sharing between two (possibly overlapping) committees $\mathcal{P}_A, \mathcal{P}_B \subset \mathcal{P}_{main}$, where $\mathcal{P}_A$ holds shares and MACs on x, while $\mathcal{P}_B$ holds the corresponding MAC keys:

$$\langle x \rangle^{\mathcal{P}_A, \mathcal{P}_B} = \Big(\{x^i, \big(M^i_j\big)_{j \in \mathcal{P}_B}\}_{i \in \mathcal{P}_A}, \{\Delta^j, (K^j_i)_{i \in \mathcal{P}_A}\}_{j \in \mathcal{P}_B} \Big)$$

When the committees are clear from context, we will sometimes omit them and simply write $\langle x \rangle$ or $[x]$.

If all the parties in $\mathcal{P}$ of size n have a sharing $\langle x \rangle^{\mathcal{P}}$, where $x = x^1 + \cdots + x^n$, any two subsets $\mathcal{P}_A, \mathcal{P}_B$ can locally convert this into a sharing $\langle x' \rangle^{\mathcal{P}_A, \mathcal{P}_B}$ of a *different* value $x' = \sum_{i \in \mathcal{P}_A} x^i$. This procedure is done by simply restricting the relevant shares and MACs to those corresponding to the two committees. We denote it as follows:

$$\mathsf{RestrictShares}(\langle x \rangle^{\mathcal{P}}, \mathcal{P}_A, \mathcal{P}_B) \rightarrow \langle x' \rangle^{\mathcal{P}_A, \mathcal{P}_B}$$

In our protocols, we rely on the fact that if the original shares of x were uniformly random, then so is the resulting value x'.

Functionality (Fig. 3). The aim of $\mathcal{F}_{\mathsf{Prep}}$ is to allow arbitrary committees to obtain $[\cdot]$ and $\langle \cdot \rangle$-shared values, in the form of random authenticated field elements, and partial triples. The functionality begins with an initialization phase, which models the setting up of the necessary data to obtain up to m_R random values and m_T multiplication triples. Then, either the Rand or Trip command can be queried by a pair of dynamically-chosen committees ($\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}$), who obtain the appropriate shares. We assume that each query uses a distinct index k, which is necessary to ensure that in our protocol, the corresponding preprocessing data is not reused when another committee produces a triple.[2]

A key difference between our functionality and previous works like SPDZ [DPSZ12,DKL+13] is that our triples are only *partially authenticated*. In a random triple (a, b, c) where $c = a \cdot b$, the values a and b are authenticated with pairwise MACs, while c is only additively shared. This is a crucial aspect which allows our protocol to support dynamically-chosen parties, and also achieving a communication overhead that is significantly less than the circuit size.

[2] In our online phases, we assume the parties have a means of agreeing upon the ordering of committees to ensure that the indices queried to $\mathcal{F}_{\mathsf{Prep}}$ are not reused.

Functionality $\mathcal{F}_{\mathsf{Prep}}$

Parameters: Finite fields $\mathbb{F}_p$ and $\mathbb{F}_{p^r}$, parties $P_1, \ldots, P_n$, adversary $\mathcal{A}$ and set of honest parties $\mathcal{P}_H$.
Functionality: Generates triples with unauthenticated c, and authenticated random values.

Init: On receiving $(\mathsf{Init}, m_T, m_R)$ from P_i, for $i \in [1, n]$, where m_T is the upper bound on the number of triples and m_R on random values, sample a MAC key $\Delta^i \leftarrow \mathbb{F}_{p^r}$, send Δ^i to P_i and ignore subsequent Init commands from P_i.
Random Value: On input $(\mathsf{Rand}, \mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}, k)$ from every $P_i \in \mathcal{P}_{\mathsf{curr}} \cup \mathcal{P}_{\mathsf{next}}$, where $k \in [1, m_R]$ and Rand has not been queried before with the same k:

1. Sample shares $r^i \leftarrow \mathbb{F}_p$, for $i \in \mathcal{P}_{\mathsf{curr}}$.
2. For each $i \in \mathcal{P}_{\mathsf{curr}}$ and $j \in \mathcal{P}_{\mathsf{next}} \setminus \{i\}$, sample $K_i^j \leftarrow \mathbb{F}_{p^r}$ and let $M_j^i = K_i^j + \Delta^j \cdot r^i$.
3. Let $\langle r \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}} = \left(r^i, (M_j^i, K_i^j)_{j \in \mathcal{P}_{\mathsf{next}} \setminus \{i\}}\right)_{i \in \mathcal{P}_{\mathsf{curr}}}$, and output the relevant shares, MACs and MAC keys to the parties in $\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}$.

Triple: On input $(\mathsf{Trip}, \mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}, k)$, from every $P_i \in \mathcal{P}_{\mathsf{curr}} \cup \mathcal{P}_{\mathsf{next}}$, where $k \in [1, m_T]$ and Trip has not been queried before with the same k:

1. Run the steps from **Random Value** twice, to create sharings $\langle a \rangle, \langle b \rangle$.
2. *Additive errors:* Wait for $\mathcal{A}$ to input $\{\delta_a^i, \delta_b^i\}_{i \in \mathcal{P}_H \cap \mathcal{P}_{\mathsf{curr}}}$, each in $\mathbb{F}_p$. Let $c = a \cdot b + \sum_{i \in \mathcal{P}_H \cap \mathcal{P}_{\mathsf{curr}}} (a^i \cdot \delta_a^i + b^i \cdot \delta_b^i)$.
3. Sample shares $c^i \in \mathbb{F}_p$, for $i \in \mathcal{P}_{\mathsf{curr}}$, such that $\sum_{i \in \mathcal{P}_{\mathsf{curr}}} c^i = c$. Let $[c]^{\mathcal{P}_{\mathsf{curr}}} := (c^i)_{i \in \mathcal{P}_{\mathsf{curr}}}$.
4. Output $\langle a \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}}, \langle b \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}}, [c]^{\mathcal{P}_{\mathsf{curr}}}$ to the parties in $\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}$.

Corrupt parties: In addition to additive errors, corrupt parties may choose their own randomness for all sharings, namely r^i in Rand, a^i, b^i, c^i in Trip, as well as any MACs and MAC keys they receive. The honest parties' shares/MACs/keys are adjusted accordingly.

Fig. 3: Functionality for the preprocessing

3.2 Preprocessing Protocol

Our protocol for realising $\mathcal{F}_{\mathsf{Prep}}$ consists of two main building blocks: a 2-party OLE functionality, and an n-party vector-OLE (VOLE) functionality; we elaborate on these below, and later (in Section 3.3) show how they can be realized. These are used for computing the unauthenticated shares of c in multiplication triples, and authenticated shares of random values, respectively.

Programmable OLE. We use a functionality for *random, programmable oblivious linear evaluation* (OLE), $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$, shown in Fig. 4. This is a two-party functionality, which computes a batch of secret-shared products, i.e. random tuples

$(u_i, v_i), (w_i, x_i)$, where $w_i = u_i x_i + v_i$, over the field $\mathbb{F}_p$. The *programmability* requirement is that, for any given instance of the functionality, the party who obtains u_i or v_i can program these to be derived from a chosen random seed. This allows e.g. the same random u_i's to be used in a different instance of $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$. We model the programmability with a function $\mathsf{Expand} : S \to \mathbb{F}_{p^r}^m$, which deterministically expands the chosen seed into a vector of field elements. When instantiating the functionality, the expansion function will correspond to some kind of secure PRG.

Multi-party programmable VOLE. Vector oblivious linear evaluation (VOLE) can be seen as a batch of OLEs with the same x_i value in each tuple, that is, a vector $\boldsymbol{w} = \boldsymbol{u}x + \boldsymbol{v}$, where $x \in \mathbb{F}_p$ is a scalar given to one party. Here, while x lies in the field $\mathbb{F}_p$, the remaining values are in the extension field $\mathbb{F}_{p^r}$, since we use VOLE to generate MACs. In multi-party VOLE, shown as $\mathcal{F}_{\mathsf{nVOLE}}$ in Fig. 5, every pair of parties (P_i, P_j) is given a random VOLE instance $\boldsymbol{w}_j^i = \boldsymbol{u}^i x^j + \boldsymbol{v}_i^j$. The functionality guarantees *consistency*, in the sense that the same $\boldsymbol{u}^i$ or x^j values will be used in each of the instances involving P_i or P_j. While unlike the OLE functionality, the $\boldsymbol{u}^i, x^i$ values in $\mathcal{F}_{\mathsf{nVOLE}}$ are not programmable, we do require that the functionality outputs to P_i a short seed representing $\boldsymbol{u}^i$, so that P_i can later use this as an input to program $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$.

Functionality $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$

Parameters: Finite field $\mathbb{F}_{p^r}$, and expansion function $\mathsf{Expand} : S \to \mathbb{F}_p^m$ with seed space S and output length m.
The functionality runs between parties P_A and P_B.

On receiving s_a from P_A and s_b from P_B, where $s_a, s_b \in S$:

1. Compute $\boldsymbol{u} = \mathsf{Expand}(s_a)$, $\boldsymbol{x} = \mathsf{Expand}(s_b)$ and sample $\boldsymbol{v} \leftarrow \mathbb{F}_p^m$.
2. Output $\boldsymbol{w} = \boldsymbol{u} * \boldsymbol{x} + \boldsymbol{v}$ to P_A and $\boldsymbol{v}$ to P_B.

Corrupt parties: If P_B is corrupt, $\boldsymbol{v}$ may be chosen by $\mathcal{A}$. For a corrupt P_A, $\mathcal{A}$ can choose $\boldsymbol{w}$ (and then $\boldsymbol{v}$ is recomputed accordingly).

Fig. 4: Functionality for programmable OLE

Protocol. Given these building blocks, we use the preprocessing protocol Π_{Prep} (Fig. 6) to generate partially authenticated triples and authenticated random values between dynamically chosen committees. As discussed earlier, the key observation is that it suffices to generate a batch of *pairwise* secret-shared products, between every pair of parties, which can later be combined to produce preprocessing amongst an arbitrary subset of the parties.

The protocol is relatively straightforward, involving no interaction other than calling the relevant functionalities. In the Init phase of the protocol, each party

Functionality $\mathcal{F}_{\mathsf{nVOLE}}$

Parameters: Finite field $\mathbb{F}_{p^r}$, and expansion function $\mathsf{Expand} : S \to \mathbb{F}_p^m$ with seed space S and output length m. The functionality runs between $P_1, \ldots, P_n$.

Initialise: On receiving Init from P_i, for $i \in [1, n]$, sample $\Delta^i \leftarrow \mathbb{F}_{p^r}$, send it to P_i, and ignore all subsequent Init commands from P_i.
Extend: On receiving (Extend) from every $P_i \in \mathcal{P}$:

1. Sample $\mathsf{seed}^i \leftarrow S$, for each $P_i \in \mathcal{P}$.
2. Compute $\boldsymbol{u}^i = \mathsf{Expand}(\mathsf{seed}^i)$.
3. Sample $(\boldsymbol{v}_i^j)_{j \neq i} \leftarrow \mathbb{F}_{p^r}^m$ for $i \in \mathcal{P}, j \neq i$. Retrieve Δ^i and compute $\boldsymbol{w}_j^i = \boldsymbol{u}^i \cdot \Delta^j + \boldsymbol{v}_i^j$.
4. If P_B is corrupt, receive a set I from $\mathcal{A}$. If $\mathsf{seed} \in I$, send $\mathsf{success}$ to P_B and continue. Else, send abort to both parties, output seed to P_B and abort.
5. Output $\big((\mathsf{seed}^i, \boldsymbol{w}_j^i), \boldsymbol{v}_j^i\big)_{j \neq i}$ to P_i, for $P_i \in \mathcal{P}$.

Corrupt parties: A corrupt P_i can choose Δ^i and seed^i. It can also choose $\boldsymbol{w}_j^i$ (and $\boldsymbol{v}_i^j$ is recomputed accordingly) and $\boldsymbol{v}_j^i$.
Global key query: If P_i is corrupted, receive $(\mathsf{guess}, \boldsymbol{\Delta}')$ from $\mathcal{A}$ with $\boldsymbol{\Delta}' \in \mathbb{F}_{p^r}^n$. If $\boldsymbol{\Delta}' = \boldsymbol{\Delta}$, where $\boldsymbol{\Delta} = (\Delta^1, \ldots, \Delta^n)$, send $\mathsf{success}$ to P_i and ignore any subsequent global key query. Else, send $(\mathsf{abort}, \boldsymbol{\Delta})$ to P_i, abort to P_j and abort.

Fig. 5: Functionality for n-party VOLE

P_i initializes $\mathcal{F}_{\mathsf{nVOLE}}$, obtaining a random MAC key Δ^i. Parties use the **Extend** command of $\mathcal{F}_{\mathsf{nVOLE}}$ to authenticate their shares with every other party. Towards this, each P_i calls $\mathcal{F}_{\mathsf{nVOLE}}$ twice, which picks two random seeds s_a^i, s_b^i and expands them into the shares $\boldsymbol{a}^i, \boldsymbol{b}^i$. It outputs to P_i the pairwise MACs on its shares of the triples, along with the seeds. Each pair (P_i, P_j) then use $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ to obtain 2-party sharings of the products $\boldsymbol{a}^i * \boldsymbol{b}^j$, for each $j \neq i$.

Later, when a triple is required by the committees $\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}$, every party in the committee $\mathcal{P}_{\mathsf{curr}}$ sums up its pairwise shares of the product terms corresponding to one triple, obtaining a share of $a \cdot b$, where a, b are the sum of the corresponding shares within that committee. The second committee $\mathcal{P}_{\mathsf{next}}$ does not have any shares of $a \cdot b$, but instead obtains the MAC keys on the a, b shares from the previous $\mathcal{F}_{\mathsf{nVOLE}}$ outputs. To obtain authenticated random values, a similar procedure is done using only $\mathcal{F}_{\mathsf{nVOLE}}$ to add MACs.

Note that, if a corrupt party P_i inputs an inconsistent seed s_a^i or s_b^i into $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$, the resulting triple will be incorrect. This is modelled by the additive errors that may be introduced in $\mathcal{F}_{\mathsf{Prep}}$.

In Appendix B, we prove the following.

Theorem 1. *Suppose that* $\mathsf{Expand} : S \to \mathbb{F}_p^m$ *is a secure pseudorandom generator. Then, the protocol* Π_{Prep} *securely implements the functionality* $\mathcal{F}_{\mathsf{Prep}}$ *in the* $(\mathcal{F}_{\mathsf{nVOLE}}, \mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}})$*-hybrid model, when up to* $n-1$ *out of* n *parties are corrupted.*

Protocol Π_{Prep}

Parameters: Finite field $\mathbb{F}_{p^r}$, number of triples m_T, random values m_R, and expansion function $\mathsf{Expand} : S \to \mathbb{F}_p^m$ with seed space S and output length m.
Init: Run the following two stages among all the parties in $\mathcal{P}_{\mathsf{main}}$.
Triples setup: repeat the following, until $\geq m_T$ outputs have been obtained (each iteration produces m).

1. Each P_i calls $\mathcal{F}_{\mathsf{nVOLE}}$ with Init, receiving Δ^i.
2. Each P_i, for $i \in [1, n]$, calls $\mathcal{F}_{\mathsf{nVOLE}}$ twice, with input Extend and receives the seeds s_a^i, s_b^i. Use the outputs to define vectors of shares $\langle \boldsymbol{a} \rangle, \langle \boldsymbol{b} \rangle$ such that $\boldsymbol{a}^i = \mathsf{Expand}(s_a^i)$ and $\boldsymbol{b}^i = \mathsf{Expand}(s_b^i)$.
3. Every ordered pair (P_i, P_j) for $i, j \in [1, n]$ calls $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ with P_i sending s_a^i and P_j sending s_b^j, and it sends back $\boldsymbol{u}^{i,j}$ to P_i and $\boldsymbol{v}^{j,i}$ to P_j, such that $\boldsymbol{u}^{i,j} + \boldsymbol{v}^{j,i} = \boldsymbol{a}^i * \boldsymbol{b}^j$.

Random values setup: repeat the following, until $\geq m_R$ outputs have been obtained.

1. Every P_i, for $i \in [1, n]$, samples a seed $s_r^i \in S$ and calls $\mathcal{F}_{\mathsf{nVOLE}}$ with input (Extend, s_r^i) from P_i, forming $\langle \boldsymbol{r} \rangle$.

Triples: To get the k-th triple in committees $\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}$:

1. Let $\langle a' \rangle, \langle b' \rangle$ be the k-th shares from $\langle \boldsymbol{a} \rangle, \langle \boldsymbol{b} \rangle$. The parties run $\mathsf{RestrictShares}(\langle a' \rangle, \langle b' \rangle, \mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}})$ to obtain $\langle a \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}}, \langle b \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}}$.
2. Each $P_i \in \mathcal{P}_{\mathsf{curr}}$ computes $c^i = a^i \cdot b^i + \sum_{j \in \mathcal{P}_{\mathsf{curr}} \setminus \{i\}} (\boldsymbol{u}^{i,j}[k] + \boldsymbol{v}^{i,j}[k])$.
3. The parties output the triple $(\langle a \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}}, \langle b \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}}, [c]^{\mathcal{P}_{\mathsf{curr}}})$.

Random Values: To get the k-th random value in committees $\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}$, the parties take $\langle r' \rangle$, the k-th random value from $\langle \boldsymbol{r} \rangle$, and run $\mathsf{RestrictShares}$ to convert this to $\langle r \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}}$.

Fig. 6: Protocol for preprocessing

3.3 Instantiating Multi-Party VOLE

In multi-party VOLE, each party P_i runs an instance of random VOLE with every other party P_j. We model two-party random VOLE as the functionality $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ in Fig. 23, and show how to realize it in Section 3.3. To allow parties to use the *same* random input in different VOLE instances, the functionality is also programmable, similarly to $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$.

The main challenge in realizing $\mathcal{F}_{\mathsf{nVOLE}}$ is to guarantee that each party uses the same programmed input across every instance of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ with other parties. For instance, a corrupt party P_i could potentially use different Δ^i values as the sender, or different seeds for $\boldsymbol{u}^i$ as the receiver across instances. To prevent this, we use a consistency check to prevent parties from using different inputs across the instances. The check involves taking a random linear combination of the outputs of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ and opening the sum, and is similar to the $\Pi_{\mathsf{TripleBucketing}}$ protocol from [HSS17], except we work over a general finite field rather than $\mathbb{F}_2$.

Another difference is that we formalize the resulting protocol and show it realizes the multi-party VOLE functionality, while in [HSS17], the check was only used as part of a larger protocol. To prove this, we had to introduce the **Global key query** command in $\mathcal{F}_{\mathsf{nVOLE}}$, which allows corrupt parties to try to guess the honest parties' global scalars (MAC keys).

The final protocol for Π_{nVOLE} appears in Fig. 7.

Protocol Π_{nVOLE}

Parameters: Extension field $\mathbb{F}_{p^r}$, parties $P_1, \dots, P_n$.
Initialise: Each party P_i samples $\Delta^i \leftarrow \mathbb{F}_{p^r}$. Every ordered pair of parties (P_i, P_j) calls $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ with $(\mathsf{Init}, \Delta^i), \mathsf{Init}$ respectively.
Random Values: To create m authenticated random values $\langle r_1 \rangle, \dots, \langle r_m \rangle$,

1. Each party P_i samples a seed s^i.
2. Each ordered pair of parties (P_i, P_j) calls $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$, with P_i sending (Extend, s^i) and P_j sending Extend.
3. Use the outputs of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ to define $\langle r_1 \rangle, \dots, \langle r_m \rangle, \langle t \rangle \in \mathbb{F}_{p^r}$.
4. Each P_i does the following to check the consistency of inputs to $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$:
   (a) Call $\mathcal{F}_{\mathsf{Rand}}$ together with other parties to get random values $\chi_1, \dots, \chi_m \in \mathbb{F}_{p^r}$.
   (b) Locally compute
   $$\langle C \rangle = \sum_{i=1}^{m} \chi_i \cdot \langle r_i \rangle + \langle t \rangle$$
   (c) P_i has a share C^i, the MACs and keys $(M_j^i, K_j^i)_{j \neq i}$ from $\langle C \rangle$.
   (d) P_i rerandomizes the share locally by sending a zero share to the other parties. Call the randomised shares $\hat{C}^i$.
   (e) Broadcasts $\hat{C}^i$ and reconstructs $C = \sum_{i=1}^{n} \hat{C}^i$
   (f) P_i calls $\mathcal{F}_{\mathsf{Commit}}$ with $n+1$ values:
   $$C^i, \quad (Z_j^i)_{j \neq i} = M_j^i, \quad Z_i^i = (C^i - C) \cdot \Delta^i - \sum_{j \neq i} K_j^i$$
5. Parties open their commitments and check that $\sum_{i=1}^{n} Z_j^i = 0$, for $j \in [1, n]$. In addition, each P_i checks that $Z_i^j = K_j^i + C^j \cdot \Delta^i$. If any of the checks fail, abort.

Fig. 7: Protocol for Consistent VOLE

Consistency Check: Since $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ does not guarantee that each party uses the same seed s^i or scalar Δ^i with every other party, we need some sort of a consistency check to detect malicious behaviour. The high level idea is for parties to compute random linear combinations on the outputs of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$, securely open the sum and check that it is zero. This check is similar to the one from [HSS17], wherein it was used to check TinyOT triples.

The protocol starts with each (P_i, P_j) running $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ between them twice, once with P_i as the sender and once as the receiver. Recall that for a value v, P_i holds the share $\langle v \rangle = (v^i, \{M_j^i, K_j^i\}_{j \neq i})$. Using the outputs of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$, each P_i can define its shares of $\langle r_1 \rangle, \ldots, \langle r_m \rangle, \langle t \rangle \in \mathbb{F}_{p^r}$ locally. To compute a random linear combination, parties call $\mathcal{F}_{\mathsf{Rand}}$ and receive $\chi_1, \ldots, \chi_m \in \mathbb{F}_{p^r}$. They can locally compute shares of $\langle C \rangle$, and reconstruct C by broadcasting the shares. We wish to check $\sum_{i=1}^n Z_j^i = 0$ for $j \in [1, n]$, where $\{Z_j^i\}_{i \neq j} = M_j^i$ and $Z_i^i = (C^i - C) \cdot \Delta^i - \sum_{j \neq i} K_j^i$. Parties commit and open their shares, and locally check that each $\sum_{i=1}^n Z_j^i = 0$. If any of them fail, they abort.

An analysis of the check is provided in Appendix C, along with the proof for the following theorem:

Theorem 2. *Protocol* Π_{nVOLE} *UC-securely computes* $\mathcal{F}_{\mathsf{nVOLE}}$ *in the presence of a static malicious party corruption up to* $n - 1$ *in the* $(\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}})$*-hybrid model.*

The Missing Pieces: Programmable OLE and VOLE. We now describe how to realize the two missing building blocks used in our preprocessing protocol, namely 2-party programmable OLE and VOLE.

Realizing $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$. This can be realized in a number of ways, for instance, based on linearly homomorphic encryption [BDOZ11]. However, this would give a protocol with communication that scales *linearly* in m, the number of OLEs. Instead, we rely on the recent work of [BCG+20], which uses a variant of the ring-LPN assumption to obtain communication that is *logarithmic* in m. While the OLE functionality from [BCG+20] is not programmable, we observe that their protocol easily supports programmable inputs, so suffices for our application.

Realizing $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$. Unlike the OLE protocol from [BCG+20], this work starts with a building block called *single-point* VOLE, where the vector $\boldsymbol{u}$ contains a single, non-zero element, which is assumed to be sampled at random. When we need programmability, however, we cannot assume this. We therefore modify the underlying single-point VOLE from [WYKW21] to support programmable inputs, and show that the resulting protocol is still secure. We show how this can then be used to build programmable VOLE, with essentially the same steps as [WYKW21]. The full details of this are given in Appendix D.

4 Dynamic SPDZ

We now show how to use our preprocessing to obtain a dynamic variant of the SPDZ protocol [DPSZ12,DKL+13]. The preprocessing is performed between the entire set of parties $\mathcal{P}_{\mathsf{main}} = \{P_1, \ldots, P_n\}$, and later, when an *online phase* *committee* $\mathcal{P}_{\mathsf{curr}} \subset \mathcal{P}_{\mathsf{main}}$ wants to run MPC, they non-interactively select the relevant preprocessing data, and run our online phase. We consider evaluating arithmetic circuits over $\mathbb{F}_p$ for a large enough (superpolynomial) p, and will use $\mathcal{F}_{\mathsf{Prep}}$ entirely over $\mathbb{F}_p$ (i.e. not using the extension field $\mathbb{F}_{p^r}$).

Since our preprocessing is significantly weaker than SPDZ — due to faulty and partially authenticated triples — we cannot use the same online phase for multiplications. Instead, in our multiplication protocol, we will first have the parties add a MAC to the 'c' component of a triple (using a preprocessed random authenticated value), and then use the fully authenticated triple to multiply. Since the triples may be faulty, to verify multiplications we take the approach of [CGH⁺18], where parties compute two versions of the circuit: one with the actual inputs and one with a randomised version of the inputs. At the end of the protocol, they first run a MAC Check protocol to verify correctness of the opened values in multiplication, as in SPDZ. If this check succeeds, they open the random value used to compute the randomised circuit. Using that, they take a random linear combination of wires in both circuits and check that they are the consistent. We start by describing the online phase protocol $\Pi_{\mathsf{SPDZ\text{-}Online}}$, before analysing the verification process and concluding with a cost analysis.

SPDZ Sharing, Share Conversion and Opening. A SPDZ share of $v \in \mathbb{F}_p$ contains a vector of additive shares $([v], [\Delta], [\Delta \cdot v])$, where the shares are held by each P_i within the current committee $\mathcal{P}_{\mathsf{curr}}$. We denote this by $[\![\cdot]\!]^{\mathcal{P}_{\mathsf{curr}}}$, and omit $\mathcal{P}_{\mathsf{curr}}$ when it is clear from context. Note that the MAC key Δ is fixed for every sharing in the same committee.

Given a pairwise authenticated sharing $\langle x \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{curr}}}$, the parties can *locally* convert this into a SPDZ sharing with the procedure Π_{Convert}:

$$\Pi_{\mathsf{Convert}}(\langle x \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{curr}}}) : P_i \text{ outputs } \Big(x^i, \Delta^i, \Delta^i \cdot v^i + \sum_{j \in \mathcal{P}_{\mathsf{curr}}} (M_j^i - K_j^i)\Big)$$

where M_j^i, K_j^i are P_i's MACs and MAC keys from the $\langle \cdot \rangle$ sharing. By inspection, this gives a consistent sharing $[\![x]\!]^{\mathcal{P}_{\mathsf{curr}}}$.

We let Π_{Open} denote the opening protocol, which given $[\![x]\!]$ or $[x]$ has all parties send to each other their shares x^i and reconstruct $x = \sum x^i$. This procedure does not check the MACs, so it may be unreliable. To check the MAC on an opened value (after running Π_{Open}), we use the standard SPDZ MAC check protocol [DKL⁺13], shown in Fig. 8.

Online Protocol. $\Pi_{\mathsf{SPDZ\text{-}Online}}$ (Fig. 9) begins with each P_i in a set of parties $\mathcal{P}_{\mathsf{curr}} \subseteq \mathcal{P}_{\mathsf{main}}$ querying $\mathcal{F}_{\mathsf{Prep}}$ to receive an authenticated random value $\langle t \rangle$, where P_i knows t, and every other party has a share of the MAC. P_i uses this to generate $[\![\cdot]\!]$ sharing of its input x. This takes one round, where P_i sends $x + t$ to everyone else, along with a fresh sharing of x. The parties then use their MACs from $\langle t \rangle$ to obtain the MAC share for $[\![x]\!]$. For the randomised circuit evaluation (used to

Protocol $\Pi_{\mathsf{SPDZ\text{-}MAC}}$

Usage: Parties in $\mathcal{P}_{\mathsf{curr}}$ want to check the MACs on opened values $(A_1, \ldots, A_m)$.

1. Parties in $\mathcal{P}_{\mathsf{curr}}$ call $\mathcal{F}_{\mathsf{Rand}}$ to obtain random values $\chi_1, \ldots, \chi_m \in \mathbb{F}_p$.
2. Compute $A = \sum_{j=1}^{m} \chi_j \cdot A_j$ and $[\gamma] = \sum_{j=1}^{m} \chi_j \cdot [\Delta \cdot A_j]$.
3. Compute $[\sigma] = [\gamma] - [\Delta] \cdot A$. Each $P_i \in \mathcal{P}_{\mathsf{curr}}$ calls $\mathcal{F}_{\mathsf{Commit}}$ with input $[\sigma]$.
4. Parties open their commitments and check that $\sum_{i=1}^{n} [\sigma] = 0$. If not, output abort, else output $\mathsf{continue}$.

Fig. 8: Protocol to check MACs in Dynamic SPDZ

check multiplications), during initialization the parties first use $\mathcal{F}_{\mathsf{Prep}}$ to obtain a random sharing $[\![r]\!]$. Then, whenever an input $[\![x]\!]$ is authenticated, the parties multiply it with $[\![r]\!]$, using a triple from $\mathcal{F}_{\mathsf{Prep}}$.

Addition and multiplication by a public constant are standard operations, performed locally by every party on its shares. Multiplication is the more challenging operation as we do not have fully authenticated triples. The first step is to call $\mathcal{F}_{\mathsf{Prep}}$ twice to get two triples $([\![a]\!], [\![b]\!], [c])$, $([\![a']\!], [\![b']\!], [c'])$, as well as two random values $[\![l]\!], [\![l']\!]$, incrementing the corresponding counter after each call. $[\![l]\!], [\![l']\!]$ are used to authenticate $[c], [c']$ of the triples. This is done by computing $[l + c], [l' + c']$ locally, and opening the values by broadcasting the shares. Parties can then locally compute the MAC on c as $\Delta^i \cdot (l + c) - [\Delta \cdot l]$ for P_i. However, since we do not check the correctness at this point, the MACs in $[\![c]\!], [\![c']\!]$ might have an additive error chosen by the adversary. In addition, the c part of the triple may have errors, since this is allowed by $\mathcal{F}_{\mathsf{Prep}}$.

Let P_i be an honest party in $\mathcal{P}_{\mathsf{curr}}$. In a triple (a, b, c), c^i can have additive errors of the form $\{\delta_a^{j,i} \cdot b^i + \delta_b^{j,i} \cdot a^i\}_{j \in \mathcal{P}_{\mathcal{A}}}$, where $\delta_a^{j,i}, \delta_b^{j,i}$ are chosen by a malicious P_j in $\mathcal{F}_{\mathsf{Prep}}$. We show in Appendix E that these errors do not give the adversary any additional power compared to injecting additive errors to the output of multiplications in the online phase, and will be detected by our verification procedure. Using the potentially inconsistent triples, parties then compute the multiplications $x \cdot y$, $rx \cdot y$ by opening $[\![x - a]\!], [\![y - b]\!], [\![rx - a']\!], [\![y - b']\!]$ in the standard way of using Beaver triples. To open $[\![\cdot]\!]$-shared values, parties broadcast arithmetic shares of the value and continue with the computation. At the end of the protocol, the verification phase computes a MAC Check on all the authenticated values that had been opened. The protocol for the online phase of Dynamic SPDZ appears in Fig. 9.

Note that for a multiplication $x \cdot y$, it is important that $[l + c]$ is not opened in the same round as $[\![x - a]\!], [\![y - b]\!]$. This is because if we do, a rushing adversary can perform the following attack: To make the illustration simpler, we consider only two parties P_i, P_j in the committee. Suppose the adversary P_j introduces an error $\delta_b^{j,i} \cdot a^i$ with an honest party P_i, using the errors in $\mathcal{F}_{\mathsf{Prep}}$. The adversary then waits until it receives $x - a$, and when opening $[l + c]$, injects another additive

Protocol $\Pi_{\textsf{SPDZ-Online}}$

Init: Each $P_i \in \mathcal{P}_{\textsf{main}}$ sends $(\textsf{Init}, m_T, m_R)$ to $\mathcal{F}_{\textsf{Prep}}$ and receives Δ^i. Later, when $\mathcal{P}_{\textsf{curr}} \subseteq \mathcal{P}_{\textsf{main}}$ wants to run the online phase, each $P_i \in \mathcal{P}_{\textsf{curr}}$ sets $\textsf{count} = 0, \textsf{rcount} = 0$, and calls $\mathcal{F}_{\textsf{Prep}}$ with $(\textsf{Rand}, \mathcal{P}_{\textsf{curr}}, \mathcal{P}_{\textsf{curr}}, \textsf{rcount})$ to obtain $[\![r]\!]$.
Input: To share an input x, P_i inputs $(\textsf{Rand}, P_i, \mathcal{P}_{\textsf{curr}}, \textsf{rcount})$ to $\mathcal{F}_{\textsf{Prep}}$ to get $\langle t \rangle$, where P_i knows t. Then,

1. P_i samples shares of x such that $x = \sum_{j \in \mathcal{P}_{\textsf{curr}}} x^j$ and sends $(x^j, x+t)$ to each $P_j \in \mathcal{P}_{\textsf{curr}}$. P_i sets its share $(\Delta \cdot x)^i = \Delta^i \cdot (x+t) - (\Delta t)^i$, where $(\Delta t)^i = \Delta^i \cdot t - \sum_{j \in \mathcal{P}_{\textsf{curr}} \setminus \{P_i\}} M^i_j$.
2. Each $P_j \in \mathcal{P}_{\textsf{curr}} \setminus \{P_i\}$ sets its share to be $[\![x]\!] = (x^j, \Delta^j \cdot (x+t) - (\Delta t)^j)$, where $(\Delta t)^j = K^j_i$.
3. Each $P_i \in \mathcal{P}_{\textsf{curr}}$ runs **Multiplication** below on $[\![x]\!]$ and $[\![r]\!]$ to get $[\![r \cdot x]\!]$.[a]

Addition: To perform addition, $[\![z]\!] = [\![x]\!] + [\![y]\!]$, each $P_i \in \mathcal{P}_{\textsf{curr}}$ locally adds their shares of $[\![x]\!], [\![y]\!]$, and $[\![rx]\!], [\![ry]\!]$ to get $[\![x+y]\!], [\![r(x+y)]\!]$.
Addition by Constant: To compute $[\![z]\!] = [\![x+c]\!]$, a designated party (say P_j) adds c to its share x^j, and all parties add $\Delta^i c$ to their MAC share.
Multiplication by Constant: To compute $[\![z]\!] = k \cdot [\![x]\!]$, each $P_i \in \mathcal{P}_{\textsf{curr}}$ locally multiply the public constant k to shares of $[\![x]\!]$ to get $[\![kx]\!], [\![r \cdot (kx)]\!]$.
Multiplication: To compute $[\![z]\!] = [\![x]\!] \cdot [\![y]\!]$ and $[\![rz]\!] = [\![rx]\!] \cdot [\![y]\!]$, each $P_i \in \mathcal{P}_{\textsf{curr}}$:

1. Calls $\mathcal{F}_{\textsf{Prep}}$ twice with inputs $(\textsf{Trip}, \mathcal{P}_{\textsf{curr}}, \mathcal{P}_{\textsf{curr}}, \textsf{count})$, incrementing $\textsf{count}$ after each call. $\mathcal{F}_{\textsf{Prep}}$ outputs shares of the triples $(\langle a \rangle, \langle b \rangle, [c]), (\langle a' \rangle, \langle b' \rangle, [c'])$.
2. Calls $\mathcal{F}_{\textsf{Prep}}$ with $(\textsf{Rand}, \mathcal{P}_{\textsf{curr}}, \mathcal{P}_{\textsf{curr}}, \textsf{rcount})$ twice to receive $\langle l \rangle, \langle l' \rangle$. Increment $\textsf{rcount}$ after each call.
3. Applies $\Pi_{\textsf{Convert}}$ on $(\langle a \rangle, \langle b \rangle, \langle a' \rangle, \langle b' \rangle, \langle l \rangle, \langle l' \rangle)$ to get $[\![\cdot]\!]$ shares.
4. Runs $\Pi_{\textsf{Open}}$ on $[e] = [x-a], [d] = [y-b], [e'] = [rx - a']$ and $[d'] = [y - b'])$.
5. Runs $\Pi_{\textsf{Open}}$ on $[l+c], [l'+c']$ and computes the multiplications as:

$$[\Delta \cdot c] = (l+c) \cdot \Delta^j - [\Delta \cdot l], \quad [\Delta \cdot c'] = (l'+c') \cdot \Delta^j - [\Delta \cdot l]$$
$$[\![z]\!] = e \cdot d + e \cdot [\![b]\!] + d \cdot [\![a]\!] + [\![c]\!]$$
$$[\![rz]\!] = e' \cdot d' + e' \cdot [\![b']\!] + d' \cdot [\![a']\!] + [\![c']\!]$$

Reconstruction: First, run $\Pi_{\textsf{SPDZ-Verify}}$ to check the multiplications. Then, to output $[\![z]\!]$, run $\Pi_{\textsf{Open}}$ on $[z]$, then use $\Pi_{\textsf{SPDZ-MAC}}$ to check its MAC.

[a] We actually only use one triple to multiply x and r, skipping the extra product in the protocol.

Fig. 9: Protocol for the online phase of Dynamic SPDZ

error given by $\big((x-a) + a^j\big) \cdot \delta_b^{j,i}$. Therefore, the triple will now be:

$$\begin{aligned}[\![a]\!], [\![b]\!], [\![c]\!] &= \{[c] + \delta_b^{j,i} \cdot a^i + [(x-a) + a^j] \cdot \delta_b^{j,i}, [\Delta \cdot c]\} \\ &= \{[c] + x \cdot \delta_b^{j,i}, [\Delta \cdot c]\}\end{aligned}$$

This results in the adversary mounting a selective failure attack, since the error now depends on the secret wire value x. It can be avoided by making the adversary add the additive error prior to learning $x - a$. A simple way of achieving this is to authenticate c one round prior to opening $x - a$. Although this costs an additional round, the authentication step of a triple for the current layer can easily be merged with the opening of $x - a$ from the previous layer. This is still secure because the triples are independent and the adversary does not gain anything by opening the independently masked c in the previous layer.

The verification phase, described in Fig. 10, is run before outputting any result of a computation. First, the parties check the MACs on all the values that were opened over the course of the computation. If the check fails, the parties abort. Otherwise, they proceed by checking correctness of multiplications, with the check from [CGH+18], which involves checking a random linear combination of the inputs and outputs, and randomised versions of them. Parties start by calling $\mathcal{F}_{\mathsf{Coin}}$ to receive random challenges $\alpha_1, \ldots, \alpha_N$ and $\beta_1, \ldots, \beta_M \in \mathbb{F}_p$. They locally compute $[\![u]\!] = \sum_{i=1}^{N} \alpha_i \cdot [\![rz_i]\!] + \sum_{i=1}^{M} \beta_i \cdot [\![\alpha v_i]\!]$ and $[\![w]\!] = \sum_{i=1}^{N} \alpha_i \cdot [\![z_i]\!] + \sum_{i=1}^{M} \beta_i \cdot [\![v_i]\!]$. If no cheating had occurred, opening $[\![u]\!] - r \cdot [\![w]\!]$ should result in zero. To check this, parties securely reconstruct $[\![r]\!]$ using Π_{Open}, locally compute $[\![u]\!] - r \cdot [\![w]\!]$. If the opened value is not zero, they reject.

Protocol $\Pi_{\mathsf{SPDZ\text{-}Verify}}$

Verification: Let $\{v_i, rv_i\}_{i \in [M]}$ be the input wires of the circuit, and $\{z_i, rz_i\}_{i \in [N]}$ be the output wires of multiplication gates of the circuit.

1. Parties start by running $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to check MACs on all the values opened in multiplications and inputs previously. If $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ fails, abort, else continue.
2. Parties call $\mathcal{F}_{\mathsf{Coin}}$ to receive $\alpha_1, \ldots, \alpha_N$, $\beta_1, \ldots, \beta_M \in \mathbb{F}_p$
3. Parties locally compute
$$[\![u]\!] = \sum_{i=1}^{N} \alpha_i \cdot [\![rz_i]\!] + \sum_{i=1}^{M} \beta_i \cdot [\![rv_i]\!]$$
$$[\![w]\!] = \sum_{i=1}^{N} \alpha_i \cdot [\![z_i]\!] + \sum_{i=1}^{M} \beta_i \cdot [\![v_i]\!]$$
4. Parties open $[\![r]\!]$ by broadcasting shares of $[r]$ and running $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ on it.
5. Parties locally compute $[\![u]\!] - r[\![w]\!]$, open it and run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$. If the MAC check passes and $u - rw = 0$, parties Accept it and go to reconstruction, else Reject.

Fig. 10: Protocol for the verification phase in Dynamic SPDZ

The analysis of the verification phase proceeds similarly to that of [CGH$^+$18], except we also need to deal with the additional errors from our preprocessing functionality. We prove the following in Appendix E.

Lemma 1. *Suppose $\mathcal{A}$ introduces additive errors of the form $\delta_a^{j,i}, \delta_b^{j,i} \neq 0$, for malicious parties P_j and honest P_i in $\mathcal{F}_{\mathsf{Prep}}$, and in $\Pi_{\mathsf{SPDZ\text{-}Online}}$ additive errors $\delta_c, \delta_{c'} \neq 0$ when authenticating triples a, b, c and a', b', c' respectively. If any errors are non-zero, then the Verification phase in $\Pi_{\mathsf{SPDZ\text{-}Online}}$ fails with probability less than $2/p$.*

The following theorem, proven in Appendix E, shows that the protocol securely realizes the standard arithmetic black-box functionality, $\mathcal{F}_{\mathsf{ABB}}$ (recall, this is identical to $\mathcal{F}_{\mathsf{DABB}}$ in Fig. 1, except the operations are all carried out in one committee, $\mathcal{P}_{\mathsf{curr}}$).

Theorem 3. *Protocol $\Pi_{\mathsf{SPDZ\text{-}Online}}$ UC-securely computes $\mathcal{F}_{\mathsf{ABB}}$ in the presence of a static malicious adversary corrupting up to all-but-one of the parties in $\mathcal{P}_{\mathsf{curr}}$, in the $(\mathcal{F}_{\mathsf{Prep}}, \mathcal{F}_{\mathsf{Coin}})$-hybrid model.*

Complexity Analysis. Compared with the standard SPDZ online phase [DKL$^+$13], our dynamic variant is more expensive, since we need to verify multiplications. Instead of 2 openings of $[\![\cdot]\!]$-shared values per multiplication, as in SPDZ, we need 4 openings of $[\![\cdot]\!]$-shared values, plus 2 openings of $[\cdot]$ sharings. This leads the overall online communication and the storage complexity to be around 3x that of SPDZ. However, our preprocessing protocol from Section 3 is vastly more efficient than any SPDZ preprocessing, since it is the only protocol that is PCG-friendly, allowing N triples to be preprocessed with communication scaling in $O(\log N)$. Furthermore, this comes with the additional flexibility of dynamically choosing the set of parties in the online phase.

5 Fluid SPDZ

In this section, we show how to run Fluid SPDZ, which is a SPDZ-like online phase that supports fluidity. We base ourselves on the universal preprocessing from Section 3, where the entire set of parties, $\mathcal{P}_{\mathsf{main}}$, is involved. Later, in the online phase, we start with a subset of parties $\mathcal{P}_{\mathsf{curr}} \subset \mathcal{P}_{\mathsf{main}}$, and this committee can later evolve in a dynamic way (in contrast to Dynamic SPDZ, where the committee is fixed once the online phase begins). As discussed in Section 2, we assume when the committee changes at the end of an epoch, the current committee is made aware of the identity of the next committee who they hand-off their state to. We show how to leverage $\mathcal{F}_{\mathsf{Prep}}$ to achieve a *maximally fluid* online phase, where each epoch may last only one round. In our protocol, we will denote the current committee in a given epoch by $\mathcal{P}_{\mathsf{curr}}$. Before going into the main online protocol, we cover some key building blocks necessary to support fluidity, and describe how we adapt the SPDZ MAC check protocol to work in this context.

Simple Resharing. We use a standard method for resharing an additively shared value $[x]^{\mathcal{P}_{\mathsf{curr}}}$ from committee $\mathcal{P}_{\mathsf{curr}}$ into committee $\mathcal{P}_{\mathsf{next}}$, as shown in Fig. 11. To reduce communication, we assume a setup where every pair of parties shares a common PRG seed. (If this is not available, note that we can still have parties in $\mathcal{P}_{\mathsf{curr}}$ sample and send the PRG seeds, which saves communication when a large batch of values is being reshared).

Protocol Π_{Reshare}

Setup: Each pair of parties $P_i, P_j \in \mathcal{P}_{\mathsf{main}}$ has a common PRG seed $s^{i,j}$.
Usage: $\mathcal{P}_{\mathsf{curr}}$ reshares $[x]^{\mathcal{P}_{\mathsf{curr}}}$ to $\mathcal{P}_{\mathsf{next}}$. Parties in $\mathcal{P}_{\mathsf{next}}$ are indexed from 1 to m.

1. Each $P_i \in \mathcal{P}_{\mathsf{curr}}$ computes $x^{i,j} \in \mathbb{F}_p$ as a fresh output of a PRG applied to $s^{i,j}$, for $j = 2, \ldots, m$. P_i defines $x^{i,1} = x^i - \sum_{j=2}^{m} x^{i,j}$.
2. Each P_i sends $x^{i,1}$ to P_1 in $\mathcal{P}_{\mathsf{next}}$. Each $P_j \in \mathcal{P}_{\mathsf{next}}$ defines its share as $x^j = \sum_{i \in \mathcal{P}_{\mathsf{curr}}} x^{i,j}$ (where if $j \neq 1$, $x^{i,j}$ is computed from the PRG).

Fig. 11: Protocol for resharing values across committees

Resharing with MACs: the Key-Switch Procedure. Since our protocol uses SPDZ $[\![\cdot]\!]$-sharing, simple resharing is not enough to securely transfer the state from one committee to another. We also need a way to securely reshare a value $[\![x]\!]$, while *switching* to a different MAC key, which is held by the second committee.

Our solution is to use the *key-switch protocol*, $\Pi_{\mathsf{Key\text{-}Switch}}$, shown in Fig. 12. This securely transfers $[\![x]\!]$ from $\mathcal{P}_{\mathsf{curr}}$ to $\mathcal{P}_{\mathsf{next}}$, while switching to the appropriate MAC key. The protocol proceeds as follows: each party $P_i \in \mathcal{P}_{\mathsf{curr}}$ starts with a random value r^i that is pairwise authenticated with every party in $\mathcal{P}_{\mathsf{curr}} \cup \mathcal{P}_{\mathsf{next}}$ — that is, P_i holds a MAC on t^i under P_j's MAC key, for each $P_j \in \mathcal{P}_{\mathsf{curr}} \cup \mathcal{P}_{\mathsf{next}}$. This can easily be obtained by a call to $\mathcal{F}_{\mathsf{Prep}}$ using the Rand command. Each P_i can then obtain $[\Delta_{\mathcal{P}_{\mathsf{curr}}} \cdot t]$, where $t = \sum_{i \in \mathcal{P}_{\mathsf{curr}}} t^i$, by combining the relevant MAC shares as in Π_{Convert}, thus forming $[\![t]\!]$. The idea now is for $\mathcal{P}_{\mathsf{curr}}$ to open the masked value $x + t$, which $\mathcal{P}_{\mathsf{next}}$ can use to obtain $[\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot x] = [\Delta_{\mathcal{P}_{\mathsf{next}}}] \cdot (x + t) - [\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t]$. All that remains is for parties in $\mathcal{P}_{\mathsf{next}}$ to get $[\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t]$. Note that $\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t = \sum_{i \in \mathcal{P}_{\mathsf{curr}}} \sum_{j \in \mathcal{P}_{\mathsf{next}}} M_j^i - K_i^j$. Therefore, the parties in $\mathcal{P}_{\mathsf{curr}}$ can reshare $M = \sum_{j \in \mathcal{P}_{\mathsf{next}}} M_j^i$ to parties in $\mathcal{P}_{\mathsf{next}}$, who then locally sum the shares and their keys to obtain shares of $\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t = M - \sum_{i \in \mathcal{P}_{\mathsf{curr}}} K_i^j$. Security of $\Pi_{\mathsf{Key\text{-}Switch}}$ is stated in Lemma 2, and analysed in Appendix F.

Lemma 2. *If parties in $\mathcal{P}_{\mathsf{curr}}$ follow the protocol, $\Pi_{\mathsf{Key\text{-}Switch}}$ leads to a consistent sharing of $[\![x]\!]^{\mathcal{P}_{\mathsf{curr}}}$, and its transcript is simulatable by random values.*

Fluid MAC Check: The MAC Check protocol from SPDZ (Fig. 8) is designed to check a large batch of MACs at the end of the computation. In the fluid

Protocol $\Pi_{\mathsf{Key\text{-}Switch}}$

Input: $[\![x]\!] = ([x], [\Delta_{\mathcal{P}_{\mathsf{curr}}} \cdot x])$ in $\mathcal{P}_{\mathsf{curr}}$.
Output: $[\![x]\!] = ([x], [\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot x])$ in $\mathcal{P}_{\mathsf{next}}$.

1. Each $P_i \in \mathcal{P}_{\mathsf{curr}}$ calls $\mathcal{F}_{\mathsf{Prep}}$ with $(\mathsf{Rand}, \mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{curr}} \cup \mathcal{P}_{\mathsf{next}}, \mathsf{rcount})$ to receive $t^i, \{M_j^i\}_{j \in \mathcal{P}_{\mathsf{curr}} \cup \mathcal{P}_{\mathsf{next}}}$, while $P_j \in \mathcal{P}_{\mathsf{curr}} \cup \mathcal{P}_{\mathsf{next}}$ receives K_i^j.
2. $\mathcal{P}_{\mathsf{curr}}$ uses Π_{Convert} to form $[\![t]\!]^{\mathcal{P}_{\mathsf{curr}}}$. Each $P_i \in \mathcal{P}_{\mathsf{curr}}$ computes $M^i = \sum_{j \in \mathcal{P}_{\mathsf{next}}} M_j^i$ to obtain $[M]$.
3. Parties in $\mathcal{P}_{\mathsf{curr}}$ run $\Pi_{\mathsf{Open}}([\![x + t]\!])$ and $\Pi_{\mathsf{Reshare}}([M], [x])$, all to $\mathcal{P}_{\mathsf{next}}$.
4. Each $P_j \in \mathcal{P}_{\mathsf{next}}$ computes $K^j = \sum_{i \in \mathcal{P}_{\mathsf{curr}}} K_i^j$ to obtain $[K]$, and then defines $[\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t] = [M] - [K]$
5. Finally, P_j can compute its share of the MAC $[\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot x]$ as $[\Delta_{\mathcal{P}_{\mathsf{next}}}] \cdot (x + t) - [\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t]$. $\mathcal{P}_{\mathsf{next}}$ outputs $[x], [\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot x]$.

Fig. 12: Protocol to switch MAC keys

setting, however, this means that parties need to keep track of all the opened values and MACs by resharing them across committees, which blows up the complexity of the protocol. An alternative would be to check MACs on values as soon as they are opened over the course of the computation. A maximally fluid instantiation of this would run over 4 epochs. We propose an incremental approach with maximal fluidity, which runs over only 2 epochs.

$\Pi_{\mathsf{Fluid\text{-}MAC}}$, detailed in Fig. 13, has two subprotocols. During the online computation, parties run **Compress MACs** to incrementally update the MAC check state, a shared value $[\sigma]$ (which is initially zero). At the end of the computation, the final committee runs **Check MACs** to verify all the MACs. Let $(A_1, \ldots, A_m)$ be a set of opened values that $\mathcal{P}_i$ wants to check the MACs on. We assume that $\mathcal{P}_{i+1}$ holds the shared state $[\sigma']$, from prior epochs. The protocol begins with $\mathcal{P}_i$, which opens a random challenge β from $\mathcal{F}_{\mathsf{Prep}}$ to $\mathcal{P}_{i+1}$; since β is obtained in $\langle \cdot \rangle$ form, $\mathcal{P}_{i+1}$ can locally check the MACs on β to verify this. By taking a linear combination with powers of β, $\mathcal{P}_{i+1}$ computes $[\sigma] = [\sigma'] + \gamma^k - [\Delta_{\mathcal{P}_i}] \cdot A$, where $A = \sum_{j=1}^{m} \beta^j \cdot A_j$ and $\gamma^k = \sum_{j=1}^{m} \beta^j \cdot [\Delta_{\mathcal{P}_i} \cdot A_j]$.

At the end of the protocol, when a committee wants to complete the MAC Check, all it has to do is securely open $[\sigma]$ and check that it is zero.

Fluid Verify: In $\Pi_{\mathsf{Fluid\text{-}Verify}}$, parties in a given committee, say $\mathcal{P}_{i+1}$, want to verify the outputs of multiplication gates using the randomised circuit outputs, similar to the verification method from Section 4. As in the Fluid MAC check, we carry out the check incrementally throughout the computation, where in the first phase, the parties open a random value, which is expanded into challenges $\alpha_i \in \mathbb{F}_p$, used to update the sharings $[\![u]\!], [\![w]\!]$, corresponding to the tally of randomised multiplications and actual multiplications. These are maintained as state, until the final verification phase where we open $[\![r]\!]$ and check that

Protocol $\Pi_{\textsf{Fluid-MAC}}$

Usage: Parties in $\mathcal{P}_i$ want to check the MACs values $(A_1, \dots, A_m)$ opened to them. We assume $\mathcal{P}_{i+1}$ gets the MAC state $[\sigma']$ from a previous run of $\Pi_{\textsf{Fluid-MAC}}$.

Compress MACs: Compute a compressed version of the MACs:

Committee i:

1. Each $P_j \in \mathcal{P}_i$ calls $\mathcal{F}_{\textsf{Prep}}$ with input $(\textsf{Rand}, \mathcal{P}_i, \mathcal{P}_{i+1}, \textsf{rcount})$ to receive $\langle \beta^j \rangle$.
2. $\textsf{Hand-off}$: Send β^j, M_k^j to each $P_k \in \mathcal{P}_{i+1}$, along with $A_1, \dots, A_m$. Reshare $[\sigma'], [\Delta_{\mathcal{P}_i}], [\Delta_{\mathcal{P}_i} \cdot A_1], \dots, [\Delta_{\mathcal{P}_i} \cdot A_m]$.

Committee $i+1$:

3. P_k locally checks $M_k^j = \beta^j \cdot \Delta^k + K_j^k$ for all $j \in \mathcal{P}_i$, and aborts if any of them fail. Let $\beta = \sum_{j \in \mathcal{P}_i} \beta^j$.
4. It updates $[\sigma']$ as $[\sigma] = [\sigma'] + \gamma^k - [\Delta_{\mathcal{P}_i}] \cdot A$, where $A = \sum_{j=1}^{m} (\beta)^j \cdot A_j$ and $\gamma^k = \sum_{j=1}^{m} (\beta)^j \cdot [\Delta_{\mathcal{P}_i} \cdot A_j]$ (here, $(\beta)^j$ is the j-th power of β).

Check MACs: (**Committee $i+2$**)

5. Set $\sigma^j = \sum_{k \in \mathcal{P}_{i+1}} [\sigma^k]$. Each $P_j \in \mathcal{P}_{i+2}$ calls $\mathcal{F}_{\textsf{Commit}}$ to commit to σ^j.
6. Open all commitments, and if they are consistent, $\textsf{Accept}$ if $\sum_{j \in \mathcal{P}_{i+2}} \sigma^j = 0$. Else, $\textsf{Reject}$.

Fig. 13: MAC Check protocol for a fluid committee

$[\![u]\!] - r \cdot [\![w]\!] = 0$. The underlying technique is similar to the one used in [CGG+21], and the protocol appears in Appendix F.

Fluid Online: We now describe how the online phase works. $\Pi_{\textsf{Fluid-Online}}$ begins the same way as $\Pi_{\textsf{SPDZ-Online}}$ with a set of parties $\mathcal{P}_{\textsf{curr}} \subseteq \mathcal{P}_{\textsf{main}}$, running Input and Initialise phases. These are used to set up the preprocessing functionality, and create authenticated sharings of the inputs. During these two phases, we assume that the committee does not change. Addition and multiplication by a public constant are local operations, so they are naturally maximally fluid operations.

Multiplication needs to be spread out over multiple epochs to do it in a maximally fluid way. To evaluate one multiplication between x, y, we need to perform two multiplications: $x \cdot y$ and $rx \cdot y$. At a high level, we can think of parties doing two things in $\Pi_{\textsf{Fluid-Mult}}$. The first is computing output shares of the multiplications $[\![z]\!], [\![rz]\!]$. The second thing is running the MAC check and the verification protocols in an incremental way, so that we retain a small state complexity throughout the computation. Both of these parts are run in parallel between the committees $\mathcal{P}_{\textsf{curr}-1}, \mathcal{P}_{\textsf{curr}}, \mathcal{P}_{\textsf{curr}+1}$.

The full online phase is given in Fig. 14. Below, we focus on describing the multiplication protocol, shown in Fig. 15.

Protocol $\Pi_{\textsf{Fluid-Online}}$

Init: Every $P_i \in \mathcal{P}_{\textsf{curr}} \subseteq \mathcal{P}_{\textsf{main}}$ sets $\textsf{count} = 0, \textsf{rcount} = 0$. P_i inputs $(\textsf{Rand}, \mathcal{P}_{\textsf{curr}}, \mathcal{P}_{\textsf{curr}}, \textsf{rcount})$ to $\mathcal{F}_{\textsf{Prep}}$ and receives $\langle r \rangle$. P_i sends $(\textsf{Init}, m_T, m_R)$ to $\mathcal{F}_{\textsf{Prep}}$ and receives Δ^i.
Input: To form $[\![\cdot]\!]$-sharing of an input x possessed by $P_i \in \mathcal{P}_{\textsf{main}}$,

1. P_i along with parties in $\mathcal{P}_{\textsf{curr}}$ runs $\Pi_{\textsf{Key-Switch}}$, where P_i (acting as $\mathcal{P}_{\textsf{curr}}$) inputs $[\![x]\!]$ under its key and parties in $\mathcal{P}_{\textsf{curr}}$ (as $\mathcal{P}_{\textsf{next}}$) receive $[\![x]\!]$ under their key.
2. Parties in $\mathcal{P}_{\textsf{curr}}$ input $(\textsf{Trip}, \mathcal{P}_{\textsf{curr}}, \mathcal{P}_{\textsf{curr}}, \textsf{count})$ to $\mathcal{F}_{\textsf{Prep}}$ and receive $(\langle a \rangle, \langle b \rangle, [c])$.
3. Then they engage to perform the multiplication of $\{[\![x_i]\!]\}_{i \in \mathcal{P}_{\textsf{curr}}}$ with $[\![r]\!]$ to produce $\{[\![r \cdot x_i]\!]\}_{i \in \mathcal{P}_{\textsf{curr}}}$.

Addition: To perform addition, $[\![z]\!] = [\![x]\!] + [\![y]\!]$, each $P_i \in \mathcal{P}_{\textsf{curr}}$ locally adds their shares of $[\![x]\!], [\![y]\!], [\![rx]\!], [\![ry]\!]$ to get $[\![x+y]\!], [\![r(x+y)]\!]$.
Addition by Constant: To compute $[\![z]\!] = [\![x+c]\!]$, a designated party (say $P_j \in \mathcal{P}_{\textsf{curr}}$) adds c to its share x^j, and all the other parties add $\Delta^i c$ to their MAC share.
Multiplication by Constant: To compute $[\![z]\!] = k \cdot [\![x]\!]$, each $P_i \in \mathcal{P}_{\textsf{curr}}$ locally multiply the public constant k to shares of $[\![x]\!]$ to get $[\![kx]\!], [\![r \cdot (kx)]\!]$.
Multiplication: To compute $[\![z]\!] = [\![x]\!] \cdot [\![y]\!]$ and $[\![rz]\!] = [\![rx]\!] \cdot [\![y]\!]$ in $\mathcal{P}_{\textsf{curr}}$, run $\Pi_{\textsf{Fluid-Mult}}$ among $(\mathcal{P}_{\textsf{curr-1}}, \mathcal{P}_{\textsf{curr}}, \mathcal{P}_{\textsf{curr+1}})$.

Verify and Reconstruct:

1. Parties in the final committee, say $\mathcal{P}_{final}$, run **Check MACs** of $\Pi_{\textsf{Fluid-MAC}}$. If $\Pi_{\textsf{Fluid-MAC}}$ fails, $\textsf{Reject}$, else continue.
2. Parties execute **Final Check** phase of $\Pi_{\textsf{Fluid-Verify}}$. If the result is $\textsf{Accept}$, for each output wire z, they open $[\![z]\!]$ by broadcasting their shares to the other parties and running both phases of $\Pi_{\textsf{Fluid-MAC}}$. If $\Pi_{\textsf{Fluid-MAC}}$ fails, $\textsf{Reject}$.

Fig. 14: Protocol for a maximally fluid online phase

Computing the output shares. In order for the current committee $\mathcal{P}_{\textsf{curr}}$ to evaluate the multiplications, we start with the committee of the previous epoch $\mathcal{P}_{\textsf{curr-1}}$. We want to use $\mathcal{P}_{\textsf{curr-1}}$ to set up an authenticated triple for $\mathcal{P}_{\textsf{curr}}$ to use. Towards this, $\mathcal{P}_{\textsf{curr-1}}$ calls $\mathcal{F}_{\textsf{Prep}}$ to receive two triples - $(\langle a \rangle, \langle b \rangle, [c])$ and $(\langle a' \rangle, \langle b' \rangle, [c'])$. In addition, they also call it using $\textsf{Rand}$ to receive authenticated shares of two random values $\langle l \rangle$ and $\langle l' \rangle$, to be used to authenticate $[c], [c']$. Parties use $\Pi_{\textsf{Convert}}$ to locally go from $\langle \cdot \rangle$ to $[\![\cdot]\!]$ shares of the triples and the random values. To transfer the triples to $\mathcal{P}_{\textsf{curr}}$ such that the MACs are under their key, $\mathcal{P}_{\textsf{curr-1}}$ runs the $\Pi_{\textsf{Key-Switch}}$ protocol with $\mathcal{P}_{\textsf{curr}}$, on $([\![a]\!], [\![b]\!]), ([\![a']\!], [\![b']\!]), [\![l]\!], [\![l']\!]$ and opens $[l+c], [l'+c']$ to them. As a result, $\mathcal{P}_{\textsf{curr}}$ can locally get authenticated shares of the triples under the MAC key $\Delta_{\mathcal{P}_{\textsf{curr}}}$. Using shares of the triples, they locally compute $[\![x-a]\!], [\![y-b]\!], [\![x-a']\!], [\![y-b']\!]$ and open them to $\mathcal{P}_{\textsf{curr+1}}$. $\mathcal{P}_{\textsf{curr+1}}$ can compute $[\![z]\!], [\![rz]\!]$ using the standard Beaver multiplication technique.

Security of the Online Protocol. We now briefly discuss security of the online protocol, $\Pi_{\mathsf{Fluid\text{-}Online}}$. As argued in Appendix F, the values sent in the key-switch protocol are always indistinguishable from random, and any errors in the resulting sharing will always be detected by a MAC check. Regarding $\Pi_{\mathsf{Fluid\text{-}MAC}}$ and $\Pi_{\mathsf{Fluid\text{-}Verify}}$, note that these protocols both follow essentially the same set of steps as the Dynamic SPDZ protocols ($\Pi_{\mathsf{SPDZ\text{-}MAC}}$ and $\Pi_{\mathsf{SPDZ\text{-}Verify}}$). The key differences are (1) the random challenges are obtained by opening random authenticated sharings, instead of $\mathcal{F}_{\mathsf{Coin}}$, and (2) the final check values are computed incrementally, instead of immediately. For (1), because the sharings are authenticated and MACs immediately checked, they are still uniformly random until the time of opening. For (2), note that since each challenge is only opened after the corresponding value being checked has been made public, its randomness still contributes in the same way as Dynamic SPDZ, to prevent cheating.

During the multiplication protocol, $\Pi_{\mathsf{Fluid\text{-}Mult}}$, the parties run the same computations as in Dynamic SPDZ, with the difference that in each round, the state is securely transferred using Π_{Reshare} or $\Pi_{\mathsf{Key\text{-}Switch}}$, and the MAC check and verification procedures are run in the background. Hence, security can be proven similarly to the proof of Theorem 3. We obtain the following.

Theorem 4. *Let $\mathcal{A}$ be an* R*-adaptive adversary in $\Pi_{\mathsf{Fluid\text{-}Online}}$. Then, the protocol UC-securely computes $\mathcal{F}_{\mathsf{DABB}}$ in the presence of $\mathcal{A}$ in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model.*

6 Cost Analysis

Table 1: Cost estimates for various protocols (comm. in # field elements)

Protocol	Online comm.	Preproc. comm.	Storage
SPDZ [KPR18,KOS16]	$2\|C\|$	$O(n\|C\|)$	$O(\|C\|)$
SPDZ (with our preproc.)	$2\ \|C\|$	$O(\|C\|) + O(n\log(\|C\|)$	$O(\|C\|) + O(n\log(\|C\|)$
Dynamic SPDZ	$6\|C\|$	$O(n\log(\|C\|)$	$O(n\log(\|C\|)$
Fluid SPDZ	$O(n_c\|C\|)$	$O(n\log(\|C\|)$	$O(n\log(\|C\|)$

In Table 1 we give some efficiency estimates for our protocols, in terms of the per-party communication and storage costs. n is the number of parties, while n_c is the average committee size in the online phase. First, in the preprocessing, our dynamic and fluid protocols have significantly smaller storage and communication compared with previous SPDZ protocols (if n is small, relative to the circuit size). As mentioned in Section 4, we can also use our preprocessing to get a modified version of SPDZ, with the same online cost as regular SPDZ, by verifying the multiplication triples in the offline phase. This gives the best preprocessing complexity for any SPDZ-like protocol with the same online phase.

The online complexities for all protocols apart from Fluid are just $O(1)$ field elements per multiplication, while with Fluid SPDZ, we get $O(n_c)$. This is because for the other protocols, we assume the players follow the "king" approach to open values [DN07], where parties send their shares to a designated party, who sums them up and sends back the result.

Protocol $\Pi_{\mathsf{Fluid\text{-}Mult}}$

Usage: $\mathcal{P}_{\mathsf{curr}}$ wants to evaluate multiplications $z = x \cdot y, rz = rx \cdot y$.

Committee $\mathcal{P}_{\mathsf{curr\text{-}1}}$:

1. Calls $\mathcal{F}_{\mathsf{Prep}}$ twice with $(\mathsf{Trip}, \mathcal{P}_{\mathsf{curr\text{-}1}}, \mathcal{P}_{\mathsf{curr\text{-}1}}, \mathsf{count})$, incrementing count after each call. $\mathcal{F}_{\mathsf{Prep}}$ outputs shares of the triples $(\langle a \rangle, \langle b \rangle, [c])$, $(\langle a' \rangle, \langle b' \rangle, [c'])$.
2. Calls $\mathcal{F}_{\mathsf{Prep}}$ with $(\mathsf{Rand}, \mathcal{P}_{\mathsf{curr\text{-}1}}, \mathcal{P}_{\mathsf{curr\text{-}1}}, \mathsf{rcount})$ twice to receive $\langle l \rangle, \langle l' \rangle$, incrementing rcount after each call.
3. Applies Π_{Convert} to get on $(\langle a \rangle, \langle b \rangle, \langle a' \rangle, \langle b' \rangle, \langle l \rangle, \langle l' \rangle)$ to get $[\![\cdot]\!]$ shares. Locally computes $[l + c], [l' + c']$.
4. Hand-off:
   (a) Run $\Pi_{\mathsf{Key\text{-}Switch}}$ on $([\![a]\!], [\![b]\!])$, $([\![a']\!], [\![b']\!])$, $[\![l]\!], [\![l']\!]$, and $[\![r]\!]$.
   (b) Run Π_{Open} on $[l + c], [l' + c']$.

Committee $\mathcal{P}_{\mathsf{curr}}$:

5. Locally computes

$$[c] = (l + c) - [l], \quad [c'] = (l' + c') - [l']$$
$$[\Delta_{\mathcal{P}_{\mathsf{curr}}} \cdot c] = [\Delta_{\mathcal{P}_{\mathsf{curr}}}] \cdot (l + c) - [\Delta_{\mathcal{P}_{\mathsf{curr}}} \cdot l]$$
$$[\Delta_{\mathcal{P}_{\mathsf{curr}}} \cdot c'] = [\Delta_{\mathcal{P}_{\mathsf{curr}}}] \cdot (l' + c') - [\Delta_{\mathcal{P}_{\mathsf{curr}}} \cdot l']$$

6. In addition, they also compute $[\![x - a]\!], [\![y - b]\!], [\![x - a']\!], [\![y - b']\!]$.
7. Executes Steps 1, 2 in **Incremental Verification** of $\Pi_{\mathsf{Fluid\text{-}Verify}}$ and **Compress MACs** in $\Pi_{\mathsf{Fluid\text{-}MAC}}$.
8. Hand-off : In parallel to the Hand-off in **Incremental Verification** and **Compress MACs**,
   (a) Run $\Pi_{\mathsf{Key\text{-}Switch}}$ on $([\![a]\!], [\![b]\!], [\![c]\!])$, $([\![a']\!], [\![b']\!], [\![c']\!])$, $[\![r]\!]$, and $[\![\boldsymbol{m}]\!]$, where $[\![\boldsymbol{m}]\!]$ is the set of wires not used in a multiplication in the current layer.
   (b) Run Π_{Open} on $[\![x - a]\!], [\![y - b]\!], [\![rx - a']\!], [\![y - b']\!]$.

Committee $\mathcal{P}_{\mathsf{curr+1}}$:

9. Locally executes the remaining steps of key-switch, and evaluates the multiplications as:

$$e = x - a, d = y - b, \quad e' = rx - a', d' = y - b'$$
$$[\![z]\!] = e \cdot d + e \cdot [\![b]\!] + d \cdot [\![a]\!] + [\![c]\!]$$
$$[\![rz]\!] = e' \cdot d' + e' \cdot [\![b']\!] + d' \cdot [\![a']\!] + [\![c']\!]$$

10. Executes Steps 3 and 4 in **Incremental Verification** of $\Pi_{\mathsf{Fluid\text{-}Verify}}$ on $[\![z]\!], [\![rz]\!]$ and in the **Compress MACs** phase in $\Pi_{\mathsf{Fluid\text{-}MAC}}$ on $(x - a, y - b, rx - a', y - b')$.

Fig. 15: Protocol for a maximally fluid multiplication

Although this takes an additional round, it reduces the communication complexity of opening a value from $O(n^2)$ to $O(n)$. While the king approach is also possible in Fluid MPC, it is harder to estimate the costs of this, since the parties need to reshare part of their current state to the king.

In Table 1 we present asymptotic estimates of the cost of variants of our protocols against the current best SPDZ protocols [KPR18,KOS16]. The primary improvement comes from our preprocessing, which can be used to run a traditional SPDZ online phase without any fluidity, at the same cost as the other approaches. It has an additional factor of $O(|C|)$ in the preprocessing compared to Dynamic and Fluid SPDZ because we also authenticate and check the triples in the preprocessing. Comparing Dynamic SPDZ with [KPR18,KOS16] shows that we can support dynamic participants at the cost of a small overhead in the online phase, and a vastly more cheaper preprocessing phase, making it practically efficient.

To get an idea of the concrete efficiency of our universal preprocessing, we give some communication estimates based on existing VOLE and OLE protocols. For producing $N = 2^{20}$ triples, each pair of the n parties needs a VOLE of length $4N$ and an OLE of length N field elements. Using state-of-the-art LPN-based VOLE [WYKW21] and OLE [BCG+20], this can be done with a total of around 4MB of communication per pair of parties. For example, using Dynamic SPDZ with 10 parties, each party can use under 40MB of bandwidth, to gain the ability to do MPC with any subset of parties later on.

6.1 Concrete Costs and Optimizations for $\Pi_{\mathsf{Fluid\text{-}Online}}$

In this section, we estimate the concrete communication cost per party running $\Pi_{\mathsf{Fluid\text{-}Online}}$. Note that running the online phase in a maximally fluid way, as described in Fig. 15, allows for multiplications to be interleaved across committees. This means that parties in a committee, say $\mathcal{P}_i$, may be involved in three multiplications in parallel. This can be seen as running three instances of $\Pi_{\mathsf{Fluid\text{-}Online}}$ in parallel, with $\mathcal{P}_i$ playing different roles $(\mathcal{P}_{\mathsf{curr\text{-}1}}, \mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{curr+1}})$ across the three instances in parallel. In addition, we can reduce the number of random challenges that need to be opened as part of **Compress MACs** and **Incremental Verification** due to the interleaving.

To calculate the concrete cost, we assume that the circuit has a uniform width of m, and the committees are of size n_c. The number of elements per party per epoch can then be estimated by the following formula: $14 \cdot m \cdot n_c + 42 \cdot m + 13 \cdot n_c + 20$. If the circuit is wide, i.e. $m \gg n_c$, the amortised cost per multiplication becomes $14 \cdot n_c + 42$. The cost of adding an additional party to the computation will roughly be 14 elements.

Though we presented maximally fluid protocols, in practice one could relax the model by allowing each epoch to last more than one round. The motivation to do so is to save in terms of the concrete communication cost. For instance, assume that the fluidity is four rounds instead of one. As the multiplication in $\Pi_{\mathsf{Fluid\text{-}Online}}$ takes three rounds (including computing **Compress MACs** and **Incremental Verification**), this means the committee that starts the multiplication will be the one to finish it as well. There will not be a need for state transfer during the multiplication, essentially getting rid of all the Key-Switch operations in

$\Pi_{\mathsf{Fluid\text{-}Online}}$. Transferring the state after the multiplication is also cheaper, as the committee will only have to Key-Switch output wires of the multiplication, the MAC key, and the random value $[\![r]\!]$. The cost of running the Fluid online with a fluidity of four is $6 \cdot m + 4 \cdot n_c$, where $6 \cdot m$ is the cost for authenticating $2m$ triples and opening the Beaver triple intermediate values, and the $4 \cdot n_c$ is for the random challenges that need to be opened for **Compress MACs** and **Incremental Verification**. With a wide enough circuit, the amortised cost per multiplication per party comes down to about 6 elements, matching the cost of Dynamic SPDZ.

Acknowledgements

This work has been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreements No. 803096 (SPEC)), the Digital Research Centre Denmark (DIREC), and the Aarhus University Research Foundation (AUFF) and the Independent Research Fund Denmark under project number 0165-00107B.

References

ADI$^+$17. Benny Applebaum, Ivan Damgård, Yuval Ishai, Michael Nielsen, and Lior Zichron. Secure arithmetic computation with constant computational overhead. In *CRYPTO 2017, Part I*, LNCS. Springer, Heidelberg, August 2017.

BCG$^+$19a. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In *ACM CCS 2019*. ACM Press, November 2019.

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

BCG$^+$20. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-LPN. In *CRYPTO 2020, Part II*, LNCS. Springer, Heidelberg, August 2020.

BCGI18. Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In *ACM CCS 2018*. ACM Press, October 2018.

BDOZ11. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT 2011*, LNCS. Springer, Heidelberg, May 2011.

BGG$^+$20. 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 *TCC 2020, Part I*, LNCS. Springer, Heidelberg, November 2020.

Bra85. Gabriel Bracha. An $O(\lg n)$ expected rounds randomized byzantine generals protocol. In *17th ACM STOC*. ACM Press, May 1985.

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

CGG$^+$21. Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. Fluid MPC: Secure multiparty computation with dynamic participants. In *CRYPTO 2021, Part II*, LNCS. Springer, Heidelberg, August 2021.

CGH$^+$18. Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In *CRYPTO 2018, Part III*, LNCS. Springer, Heidelberg, August 2018.

DDN$^+$16. Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. Confidential benchmarking based on multiparty computation. In *FC 2016*, LNCS. Springer, Heidelberg, February 2016.

DKL$^+$13. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS 2013*, LNCS. Springer, Heidelberg, September 2013.

DN07. Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO 2007*, LNCS. Springer, Heidelberg, August 2007.

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

GHK$^+$21. Craig Gentry, Shai Halevi, Hugo Krawczyk, Bernardo Magri, Jesper Buus Nielsen, Tal Rabin, and Sophia Yakoubov. YOSO: You only speak once - secure MPC with stateless ephemeral roles. In *CRYPTO 2021, Part II*, LNCS. Springer, Heidelberg, August 2021.

GKM$^+$20. Vipul Goyal, Abhiram Kothapalli, Elisaweta Masserova, Bryan Parno, and Yifan Song. Storing and retrieving secrets on a blockchain. Cryptology ePrint Archive, Report 2020/504, 2020. `https://eprint.iacr.org/2020/504`.

GSY21. S. Dov Gordon, Daniel Starin, and Arkady Yerukhimovich. The more the merrier: Reducing the cost of large scale MPC. In *EUROCRYPT 2021, Part II*, LNCS. Springer, Heidelberg, October 2021.

HJKY95. Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *CRYPTO'95*, LNCS. Springer, Heidelberg, August 1995.

HSS17. Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In *ASIACRYPT 2017, Part I*, LNCS. Springer, Heidelberg, December 2017.

KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS 2016*. ACM Press, October 2016.

KPR18. Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT 2018, Part III*, LNCS. Springer, Heidelberg, April / May 2018.

MZW$^+$19. Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. CHURP: Dynamic-committee proactive secret sharing. In *ACM CCS 2019*. ACM Press, November 2019.

SSW17. Peter Scholl, Nigel P. Smart, and Tim Wood. When it's all just too much: Outsourcing MPC-preprocessing. In *16th IMA International Conference on Cryptography and Coding*, LNCS. Springer, Heidelberg, December 2017.

WYKW21. Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. 42nd IEEE Symposium on Security and Privacy (Oakland 2021), 2021.

Supplementary Material

A Additional Functionalities

In our protocols, we use standard functionalities for commitments and oblivious transfer. We also use a weak equality test, the functionality for which appears in Fig. 17. Rest of the functionalities are shown in $\mathcal{F}_{\mathsf{Commit}}$ Fig. 18, and $\mathcal{F}_{\mathsf{OT}}$ Fig. 19.

Functionality $\mathcal{F}_{\mathsf{Rand}}$

The functionality runs between a set of parties $\mathcal{P}$ and an adversary $\mathcal{A}$.
Upon receiving a description of a domain $\mathbb{F}_{p^r}^m$ from every party in $\mathcal{P}$, uniformly sample $(x_1, \ldots, x_m) \leftarrow \mathbb{F}_{p^r}^m$ and send this to $\mathcal{A}$. If $\mathcal{A}$ responds with Deliver, send $x_1, \ldots, x_m$ to all parties and terminate. Otherwise, if $\mathcal{A}$ sends Abort, send Abort to all parties and terminate.

Fig. 16: Ideal functionality for coin tossing

Functionality $\mathcal{F}_{\mathsf{EQ}}$

This functionality receives a value V_A from P_A and V_B from P_B, checks if $V_A = V_B$, and reveals P_A's input to P_B.
Equality Check: On input (EQ, V_i) from P_i for $i \in [A, B]$:

1. Send V_A to P_B.
2. If P_B is honest, output success or fail depending on $V_A \stackrel{?}{=} V_B$ to P_A.
3. If P_B is corrupted, output to P_A whatever P_B sends.

Fig. 17: Functionality to for a weak equality check

B Security of Π_{Prep}

Theorem 5 (Theorem 1, restated). *Suppose that* Expand $: S \to \mathbb{F}_p^m$ *is a secure pseudorandom generator. Then, the protocol* Π_{Prep} *securely implements the functionality* $\mathcal{F}_{\mathsf{Prep}}$ *in the* $(\mathcal{F}_{\mathsf{nVOLE}}, \mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}})$*-hybrid model, when up to* $n-1$ *out of* n *parties are corrupted.*

Proof. Since the protocol involves no interaction other than with $\mathcal{F}_{\mathsf{nVOLE}}$ and $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$, simulation is quite straightforward. Let A be the set of corrupt parties. We construct a simulator, $\mathcal{S}$, as follows. For each $i \in A$, $\mathcal{S}$ receives Δ^i from $\mathcal{A}$ and

Functionality $\mathcal{F}_{\mathsf{Commit}}$

The functionality runs between a set of parties $\mathcal{P}$ and an adversary $\mathcal{A}$.

Commit: On input $(\mathsf{commit}, P_i, x, \tau_x)$ from P_i, where τ_x is a previously unused identifier, store (P_i, x, τ_x) and send (P_i, τ_x) to all parties.
Open: On input $(\mathsf{open}, P_i, \tau_x)$ from P_i, retrieve x and send (x, i, τ_x) to all parties.

Fig. 18: Ideal functionality for commitments

Functionality $\mathcal{F}_{\mathsf{OT}}$

On receiving (m_0, m_1) from P_A (sender), where $|m_0|= |m_1|$, and $b \in \{0, 1\}$ from P_B (receiver), output m_b to P_B.

Fig. 19: Functionality to for oblivious transfer

forwards it to $\mathcal{F}_{\mathsf{Prep}}$. We focus on the setup for triple generation; the simulation for random values is simpler. $\mathcal{S}$ receives the corrupt parties' seeds s_a^i, s_b^i as input to $\mathcal{F}_{\mathsf{nVOLE}}$, as well as the MACs and MAC key outputs which are chosen by the corrupt parties. $\mathcal{S}$ then computes the expanded shares $\boldsymbol{a}^i = \mathsf{Expand}(s_a^i)$ and $\boldsymbol{b}^i = \mathsf{Expand}(s_b^i)$. For each $i \in A$ and honest P_j, it receives seeds $s_a^{i,j}, s_b^{i,j}$ as input to the $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ instances between P_i and P_j. For any instance where $s_a^{i,j} \neq s_a^i$, $\mathcal{S}$ computes the additive error multipliers $\boldsymbol{\delta}_b^{i,j} = \mathsf{Expand}(s_a^{i,j}) - \boldsymbol{a}^i$, and similarly computes $\boldsymbol{\delta}_a^{i,j} = \mathsf{Expand}(s_b^{i,j}) - \boldsymbol{b}^i$. For $j \in [n] \setminus A$, let $\boldsymbol{\delta}_b^j = \sum_{i \in A} \boldsymbol{\delta}_b^{i,j}$, and $\boldsymbol{\delta}_a^j = \sum_{i \in A} \boldsymbol{\delta}_a^{i,j}$.

Finally, $\mathcal{S}$ sends the error terms $\boldsymbol{\delta}_a^j, \boldsymbol{\delta}_b^j$ to $\mathcal{F}_{\mathsf{Prep}}$, as well as the corrupted parties' expanded shares $\boldsymbol{a}^i, \boldsymbol{b}^i$ (for $i \in A$), MACs, MAC keys and c^i shares (all computed the same way as in the protocol).

We now argue indistinguishability of the ideal and real executions. Since the corrupt parties receive no information during the protocol, we only need to look at the distribution of the parties' outputs. Let $\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}$ be two committees which query the **Triples** command, and suppose each committee has at least one honest party (for an entirely corrupt committee, indistinguishability of the corresponding outputs is trivial). Each sharing $\langle a \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}}, \langle b \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}}$ is defined from a subset of the original sharings $\langle a \rangle, \langle b \rangle$, where each honest party's share a^i, b^i was derived as an output of Expand on an independent random seed. Hence, by a standard hybrid argument, these shares are computationally indistinguishable from random values. The MACs and MAC keys held by the two committees on $\langle a \rangle, \langle b \rangle$ are perfectly indistinguishable, because in both worlds, corrupt parties choose their own values, while values between a pair of honest parties are sampled at random. Finally, we need to consider the shares c^i, for $i \in \mathcal{P}_{\mathsf{curr}}$. In the real world, we have

$$\begin{aligned} c = \sum_{i \in \mathcal{P}_{\mathsf{curr}}} c^i &= \sum_{i \in \mathcal{P}_{\mathsf{curr}}} (a^i b^i + \sum_{j \neq i} (u^{i,j} + v^{i,j})) \\ &= \sum_{i \in \mathcal{P}_{\mathsf{curr}}} (a^i b^i + \sum_{j \neq i} (u^{i,j} + v^{j,i})) \\ &= \sum_{i \in \mathcal{P}_{\mathsf{curr}}} (a^i b^i + \sum_{j \neq i} (a^{i,j} b^{j,i}) \end{aligned}$$

where $a^{i,j}, b^{i,j}$ equal a^i, b^i if P_i is honest, or if P_i is corrupt, derived from the seed used by P_i with P_j in $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$. Plugging in $a^{i,j} = \delta_b^{i,j} + a^i$ and $b^{j,i} = \delta_a^{j,i} + b^j$, we have

$$\begin{aligned} c &= \sum_{i \in \mathcal{P}_{\mathsf{curr}}} (a^i b^i + \sum_{j \neq i} (a^i + \delta_b^{i,j}) \cdot (b^j + \delta_a^{j,i}) \\ &= ab + \sum_{i \in \mathcal{P}_{\mathsf{curr}}} \sum_{j \neq i} (a^i \delta_a^{j,i} + b^j \delta_b^{i,j}) \\ &= ab + \sum_{i \in \mathcal{P}_{\mathsf{curr}}} (a^i \delta_a^i + b^i \delta_b^i) \end{aligned}$$

where δ_a^i, δ_b^i are defined as in the error vectors from the simulation, and we have assumed that, for any i, j where both P_i and P_j are corrupt, $\delta_a^{i,j}$ and $\delta_b^{j,i}$ are both zero (since here, simulation is trivial).

It follows that the way c is computed in the real world, above, is identical to that in the ideal world. Furthermore, the randomness of the individual c^i shares is guaranteed, because of the randomly sampled outputs of $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ between two honest parties.

C Security of Π_{nVOLE}

In this section, we give the complete security proof for the multi-party VOLE protocol, Π_{nVOLE}.

C.1 Analysis of the Consistency Check

Since $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ does not guarantee that each party P_i uses the same seed s^i with every other party, we need some sort of a consistency check to detect malicious behaviour. The high level idea is for parties to compute a random linear combination on the outputs of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$, securely open the sum and check that it is zero. The check is similar to the one from [HSS17], wherein it was used to check TinyOT triples.

Recalling the notation for a 2-party MAC between (P_i, P_j), P_i holds the values (x^i, M_j^i), where $M_j^i(x^i) = K_i^j(x^i) + x^i \cdot \Delta^j$. K_i^j is the local key that P_j has with P_i, and Δ^j is the global key that is supposed to be kept the same across interactions with different parties.

We formalise the security of the consistency check used in Fig. 7. There are two sources of errors a corrupt P_B can use, which are:

1. Providing inconsistent inputs (Δ) when acting as the sender in the $\mathsf{Initialise}$ command of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ with 2 different honest parties.
2. Providing inconsistent values (s) when acting as the receiver in the Extend command of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ with 2 different honest parties.

In both instances, we are only concerned with the cases in which a dishonest party interacts with an honest one. If both parties are corrupt, $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ need not be simulated in the proof.

Difference between [HSS17] and this: In [HSS17], the adversary can use different values as inputs when acting as the receiver with different honest parties. This translates to a chosen additive error by the adversary. However, in our case the adversary inputs a seed s, from which the value $\boldsymbol{u}$ is computed as $\mathsf{Expand}(s)$. Therefore, this will not be an arbitrarily chosen additive error but limited to a subset of values over the field.

For the analysis, we continue to treat this error as an arbitrarily chosen additive error.

These attacks are modelled by defining the inputs used by a corrupt P_j, with every honest party. Let P_{i_0} be the party for which P_j uses the inputs s^{j,i_0}, and Δ^{j,i_0}, which we consider to be the *actual* inputs. As a result of using a different s with different parties, the values $\boldsymbol{r}, t$ will be different. Let the values used by P_j with P_{i_0} be r_l^{j,i_0}, t^{j,i_0} $\forall l \in [m]$. For simplicity, we omit the i_0 in the superscript for these values. Ideally P_j should use the same inputs with every other honest party. We can model the errors as:

$$\begin{aligned}
\varepsilon^{j,i_0} &= 0, \quad \varepsilon^{j,i} = \Delta^{j,i} - \Delta^j, \quad i \notin (\mathcal{A} \cup i_0)\\
\delta^{j,i_0} &= 0, \quad \delta_l^{j,i} = r_l^{j,i} - r_l^j, \quad l \in [m], i \notin (\mathcal{A} \cup i_0)\\
\hat{\delta}^{j,i_0} &= 0, \quad \hat{\delta}_l^{j,i} = t^{j,i} - t^j, \quad i \notin (\mathcal{A} \cup i_0)
\end{aligned}$$

Where $\varepsilon^{j,i}$ is the error in the global key used by P_j with P_i. This error is fixed in the $\mathsf{Initialise}$ command, whereas the error δ can be different in every instance of Extend. If P_i, P_j are both corrupt, or both honest, the errors are set to 0. Therefore, the outputs of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ between (P_j, P_i) satisfy:

$$M_i^J(r_l^{j,i}) = K_j^i(r_l^{j,i}) + r_l^{j,i} \cdot \Delta^{i,j}$$

or equivalently,

$$M_i^j(r_l^j + \varepsilon_l^{j,i}) = K_j^i(r_l^j + \varepsilon_l^{j,i}) + (r_l^j + \delta_l^{j,i}) \cdot (\Delta^j + \varepsilon^{i,j})$$

$\delta^{j,i} \neq 0$ if P_j (the receiver) cheated, and $\varepsilon^{i,j} \neq 0$ if P_i (the sender) cheated.

The first case is of a corrupt sender P_j, which uses inconsistent global keys $\Delta^{j,i}$ when acting as a sender with different honest parties P_i, $i \notin (\mathcal{A} \cup i_0)$. The inconsistency is proved impossible via:

Lemma 3. *If Π_{nVOLE} succeeds, then all the global keys $\Delta^{j,i}$ are consistent and well defined, i.e $\varepsilon^{j,i} = 0$ for every $i, j \in [1, n]$.*

Proof. We start by analysing possible deviations by $P_j \in \mathcal{A}$ in Step 4g in Fig. 7, where we want to catch inconsistent $\Delta^{j,i}$ used with different honest parties.

In Step 4e, parties broadcast their shares of C, and the corrupted parties can send the wrong shares so that $\sum_{j=1}^{n} \hat{C}^j = C + e$, where e is the additive error from P_j. Another thing the corrupted parties can do is cheat in the commitments, by committing to $\hat{Z}_j^l$ values such that $\sum_{l\in\mathcal{A}} \hat{Z}_j^l = \sum_{l\in\mathcal{A}} Z_j^l + E^j$.

Therefore, the check now becomes:

$$\begin{aligned}
0 &= \sum_{i=1}^{n} \hat{Z}_j^i \\
&= E^j + Z_j^j + \sum_{i\neq j} Z_j^i \\
&= E^j + \left[(C^j - C - e)\cdot\Delta^j - \sum_{i\neq j} K^j(C^i) \right] + \sum_{i\neq j} M_j^i(C^i) \\
&= E^j + (C^j - C - e)\cdot\Delta^j + \sum_{i\neq j}(M_j^i(C^i) - K_i^j(C^i)) \\
&= E^j + (C^j - C - e)\cdot\Delta^j + \sum_{i\neq j} C^i\cdot\Delta^{j,i} \\
&= E^j + (C^j + \sum_{i\neq j} C^i - C - e)\cdot\Delta^j + \sum_{i\neq j} C^i\cdot\varepsilon^{j,i} \\
&= E^j - e\cdot\Delta^j + \sum_{i\neq j} C^i\cdot\varepsilon^{j,i}
\end{aligned}$$

where $\varepsilon^{j,i}$ indicates the error as compared to the Δ^j used in computing Z_j^j. Using inconsistent global keys means that $\exists i' \notin (\mathcal{A}\cup i_0), \varepsilon^{j,i'} \neq 0$. Therefore the attack would require $e\cdot\Delta^j - E^j = C^{i'}\cdot\varepsilon^{j,i'}$. P_j does not know anything about the shares of C at the time of committing to $\hat{Z}_j^l$ due to using the re-randomised shares of C for reconstruction in step 4e. Therefore, the probability that the check passes with the errors is $1/\mathbb{F}$ as the adversary will have to guess the share of C.

The second case is proving that P_j as a corrupted receiver cannot input inconsistent values $\boldsymbol{e}^{j,i}$ to different honest parties.

Lemma 4. *If Π_{nVOLE} succeeds, every ordered pair (P_i, P_j) holds a secret sharing of $r_l^j \cdot \Delta^i$ for every $l \in [1, m]$. In other words, $\delta_l^{j,i} = 0$ for every i, j, l.*

Proof. We can define the MAC on C^j held by P_j with party P_i as,

$$M_i^j(C^j) = \sum_{l=1}^{m} \chi_l \cdot M_i^j(r_l^{j,i}) + M_i^j(t^{j,i})$$

and the key held by P_i as,

$$K_j^i(C^j) = \sum_{l=1}^{m} \chi_l \cdot K_j^i(r_l^{j,i}) + K_j^i(t^{j,i})$$

In step 4f of Π_{nVOLE}, a corrupted P_j can commit to incorrect MACs $\hat{Z}_i^j(C^j) = M_i^j(C^j) + E_i^j$ and $\hat{C}^j = C^j + e^j$. In order to succeed, the check $\hat{Z}_i^j = K_j^i(C^j) + \hat{C}^j \cdot \Delta^i$ from step 4g must hold. This implies,

$$M_i^j(C^j) + E_i^j = K_j^i(C^j) + (C^j + e^j) \cdot \Delta^i$$

$$\implies E_i^j - (C^j + e^j) \cdot \Delta^i = K_j^i(C^j) - M_i^j(C^j) = -\left(\sum_{l=1}^{m} \chi_l \cdot r_l^{j,i} + t^{j,i}\right) \cdot \Delta^i$$

$$\implies E_i^j = \left(C^j + e^j - \sum_{l=1}^{m} \chi_l \cdot (r_l^j + \delta_l^{j,i}) + (t^{j,i} + \hat{\delta}^{j,i})\right) \cdot \Delta^i = (e^j - \sum_{l=1}^{m} \chi_l \cdot \delta_l^{j,i} + \hat{\delta}^{j,i}) \cdot \Delta^i$$

A malicious P_j has two options to cheat, both with probability of $1/\mathbb{F}$ to succeed:

1. Setting $E_i^j = (e^j - \sum_{l=1}^{m} \chi_l \cdot \delta_l^{j,i} + \hat{\delta}^{j,i}) \cdot \Delta^i \neq 0$, which requires guessing Δ^i, known only to P_i.
2. Set $E_i^j = 0$ and $e^j = \sum_{l=1}^{m} \chi_l \cdot \delta_l^{j,i} + \hat{\delta}^{j,i}$ for every $i \notin \mathcal{A}$. Since $\delta_l^{j,i_0} = \hat{\delta}^{j,i_0} = 0$, e^j should also be 0. Therefore, for $i \notin (\mathcal{A} \cup i_0)$ it should hold that,

$$0 = \sum_{l=1}^{m} \chi_l \cdot \delta_l^{j,i} + \hat{\delta}^{j,i} = \hat{\delta}^{j,i} = -\sum_{l=1}^{m} \delta_l^{j,i} \cdot \chi_l \in \mathbb{F}_{p^r}$$

Since $\boldsymbol{\chi}$ are uniformly random values from a field, the probability that this holds is $1/\mathbb{F}$.

C.2 Security Proof

Theorem 6 (Theorem 2, restated). *Protocol Π_{nVOLE} UC-securely computes $\mathcal{F}_{\mathsf{nVOLE}}$ in the presence of a static malicious party corruption up to $n-1$ in the $(\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}})$-hybrid model.*

Proof. We construct a PPT Simulator ($\mathcal{S}$) that run the adversary ($\mathcal{A}$) as a subroutine, and is given access to $\mathcal{F}_{\mathsf{nVOLE}}$. It internally emulates the functionalities $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}, \mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{Commit}}$ and we implicitly assume that it passes all communication between $\mathcal{A}$ and the environment ($\mathcal{Z}$).

The parties controlled by the $\mathcal{A}$ are indicated by $\mathcal{P}_{\mathcal{A}}$ and the honest parties by $\mathcal{P}_{\mathcal{H}}$. The simulator uses a flag which is set to 1 in case $\mathcal{A}$ is caught cheating before the consistency check happens, and the simulation is carried on. The simulation proceeds as follows:

Malicious $\mathcal{P}_{\mathcal{A}}$:

Init: $\mathcal{S}$ receives a vector $\boldsymbol{\Delta}^i$ for every $i \in \mathcal{A}$, which are its inputs to $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$. $\mathcal{S}$ chooses the first one in each of these vectors and forwards them to $\mathcal{F}_{\mathsf{nVOLE}}$ with the Init command. If any of these vectors are not of the form $(\Delta^i, \ldots, \Delta^i)$, set the $\mathsf{flag} = 1$.

Random Values:

1. When $\mathcal{A}$ acts as receiver in step 2, $\mathcal{S}$ receives a vector $\boldsymbol{e}_j^i$ from every $P_i \in \mathcal{P}_{\mathcal{A}}$ and $j \in [1, n]$. It picks the first vector and forwards it to $\mathcal{F}_{\mathsf{nVOLE}}$ with the Extend command. If any of the vectors received from a P_i are inconsistent, set $\mathsf{flag} = 1$.
2. For $P_i \in \mathcal{P}_{\mathcal{A}}$ and $j \in [1, n]$, $\mathcal{S}$ records $\boldsymbol{w}_j^i$ when $\mathcal{A}$ acts as the receiver in step 2, and $\boldsymbol{v}_j^i$ when it acts as the sender.
3. Emulate the call to $\mathcal{F}_{\mathsf{Rand}}$ by sampling $\chi_1, \ldots, \chi_m$ and sending them to $\mathcal{A}$.
4. Receive zero-shares from $\mathcal{A}$ and record them. Sample a zero-share for $P_j \in \mathcal{P}_H$ and send them to $\mathcal{A}$.
5. Sample a random share of C for each honest party and send them to $\mathcal{A}$. Receive $\hat{C}^i$ for $P_i \in \mathcal{P}_{\mathcal{A}}$, reconstruct $C = \sum_{i=1}^n \hat{C}^i$.
6. Emulate $\mathcal{F}_{\mathsf{Commit}}$ by recording $\tilde{C}^i, (Z_j^{i'})_{j \neq i}, Z_i^{i'}$ from $P_i \in \mathcal{P}_{\mathcal{A}}$. $\mathcal{S}$ computes C^i as it knows $\boldsymbol{\chi}$, and shares of $\mathcal{A}$ for $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$. Using those, it sets $\langle C \rangle = \sum_{i=1}^m \chi_i \cdot \langle r_i \rangle + \langle t \rangle$ for all parties in $\mathcal{P}_{\mathcal{A}}$.
7. If $\tilde{C}^i = C^i$ and $\mathsf{flag} = 0$: for each sum, $\sum_{i=1}^n Z_j^i$, where $j \in [1, n]$, sample shares for the honest parties as follows: sample uniformly random values for all but one honest party, and pick the last share such that the sum is zero.
8. If $\tilde{C}^i = C^i$ and $\mathsf{flag} = 1$: sample random values for $\mathcal{P}_H$ for shares of Z and send them to $\mathcal{A}$, send abort to $\mathcal{F}_{\mathsf{nVOLE}}$ and abort.
9. If $\tilde{C}^i \neq C^i$, compute $\tilde{Z}_j^i - Z_j^i$, where Z_j^i is the value computed by $\mathcal{S}$ using C^i, for all $j \in \mathcal{P}_H$. For $\mathcal{A}$ to pass the check, it must have guessed the correct Δ^j for every honest P_j.
   (a) Therefore, $\mathcal{S}$ can extract $\mathcal{A}$'s guess as $\tilde{\Delta}^j = (\tilde{Z}_j^i - Z_j^i)/(\tilde{C}^i - C^i)$. Set $\tilde{\boldsymbol{\Delta}} = (\tilde{\Delta}^j, \ldots)$.
   (b) Forward $(\mathsf{guess}, \tilde{\boldsymbol{\Delta}})$ to $\mathcal{F}_{\mathsf{nVOLE}}$. If $\mathcal{F}_{\mathsf{nVOLE}}$ returns $\mathsf{success}$, send true to $\mathcal{A}$, forward $\boldsymbol{w}$ to $\mathcal{F}_{\mathsf{nVOLE}}$. Compute shares of $\mathcal{P}_H$ such that $\sum_{i=1}^n Z_j^i = 0$ for $j \in [1, n]$ and send them to $\mathcal{A}$. Output whatever $\mathcal{A}$ outputs.
   (c) Else, receive $(\mathsf{abort}, \boldsymbol{\Delta})$, where $\boldsymbol{\Delta}$ is the vector of Δ values used by $\mathcal{P}_H$. Compute shares of $\mathcal{P}_H$ using $\boldsymbol{\Delta}$, send them to $\mathcal{A}$, and abort.
10. Whenever $\mathcal{A}$ queries $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ with a set I, forward it to $\mathcal{F}_{\mathsf{nVOLE}}$.

D Realizing $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$

D.1 Chosen-input single point VOLE

We start with a standard random VOLE functionality called Base VOLE, as shown in Fig. 20. This can be realised by any of the existing protocols for VOLE [ADI+17,BCGI18,BCG+19a,WYKW21]. Using this, we build a single-point subfield VOLE (spsVOLE), where the input of the receiver is a $\boldsymbol{u}$ such that $\boldsymbol{u}[\alpha] = \beta$ and is 0 everywhere else. Wolverine [WYKW21] has a construction for random spsVOLE, where the sender's global key Δ and the receiver's input $\boldsymbol{u}$ are randomly picked. Since, in our setting, we want parties to be able to influence the randomness used to derive their inputs, we give a modified version of this protocol that supports chosen-inputs, in Fig. 22.

Functionality $\mathcal{F}_{\mathsf{sVOLE}}$

Parameters: An extension field $\mathbb{F}_{p^r}$, length m, and party identifiers P_A, P_B.
Initialise: On receiving Init from P_A, and (Init, Δ) from P_B, store the global key Δ, and ignore all subsequent Init commands.
Extend: This procedure can be run multiple times. On receiving (Extend, l) from P_A, P_B, do:

1. If P_B is honest, sample $\mathsf{K}[x] \leftarrow \mathbb{F}_{p^r}^l$. Else, receive $\mathsf{K}[x] \in \mathbb{F}_{p^r}^l$ from $\mathcal{A}$.
2. If P_A is honest, sample $x \leftarrow \mathbb{F}_p^l$ and compute $\mathsf{M}[x] = \mathsf{K}[x] + \Delta \cdot x \in \mathbb{F}_{p^r}^l$. Else, receive $x \in \mathbb{F}_p^l$ and $\mathsf{M}[x] \in \mathbb{F}_{p^r}^l$ from $\mathcal{A}$ and recompute $\mathsf{K}[x] = \mathsf{M}[x] - \Delta \cdot x \in \mathbb{F}_{p^r}^l$.
3. Send $(x, \mathsf{M}[x])$ to P_A and $\mathsf{K}[x]$ to P_B.

Global key query: If P_A is corrupted, receive $(\mathsf{guess}, \Delta')$ from $\mathcal{A}$ with $\Delta' \in \mathbb{F}_{p^r}$. If $\Delta' = \Delta$, send $\mathsf{success}$ to P_A and ignore any subsequent global key query. Else, send abort to both parties and abort.

Fig. 20: Functionality for a subfield VOLE (Base VOLE)

To reflect the chosen-input protocol, we need to slightly modify the spsVOLE functionality, $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$, in Fig. 21. First, we let the party P_A choose α and β, which determine the special point in the vector $\boldsymbol{u}$, which is nonzero only at $\boldsymbol{u}[\alpha] = \beta$. The second tweak is to make $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ reveal the secret index α used by an honest P_A, in case of an abort. Previously, this was not needed, since α was always sampled at random and not a private input.

The protocol $\Pi^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ uses $\mathcal{F}_{\mathsf{OT}}$, a standard OT functionality, and $\mathcal{F}_{\mathsf{EQ}}$, a functionality to check equality, which reveals an honest P_A's input to P_B. The protocol can be split into two parts, with the first part being a semi-honest VOLE protocol, and the second part involving a consistency check.

Parties P_A, P_B start by generating $\langle \beta \rangle \in \mathbb{F}_p$, where β is an input of P_A. Doing so is straightforward and involves one call to $\mathcal{F}_{\mathsf{sVOLE}}$. P_A then defines the single-point vector $\boldsymbol{u} \in \mathbb{F}_p^m$ such that $\boldsymbol{u}[\alpha] = \beta$, where $\alpha \in [0, n)$ is also its input. Next we need P_B to generate $\boldsymbol{v} \in \mathbb{F}_p^m$ in such a way that P_A learns all $\boldsymbol{v}[i]$ values except $\boldsymbol{v}[\alpha]$. Towards this, parties run the GGM subroutine, starting with P_B sampling $s \leftarrow \{0,1\}^\kappa$ and computing all the nodes in the GGM tree of depth h with s as the root node. The j-th node in the tree at the i-th level is denoted by s^i_j. P_B defines $s^0_0 = s$ as the root, and computes $\left(s^i_{2j}, s^i_{2j+1}\right) = G(s^{i-1}_j)$, for $i \in [1, h)$, $j \in [0, 2^{j-1})$, where $G : \{0,1\}^\kappa \to \{0,1\}^{2k}$ is a PRG. The leaf nodes are computed as $(\boldsymbol{v}[2j], \boldsymbol{v}[2j+1]) = G'(s^{h-1}_j)$ for $j \in [0, 2^{h-1})$, where $G' : \{0,1\}^\kappa \to \mathbb{F}^2_{p^r}$ is a PRG. The $\mathsf{GGM}(1^n, s)$ output can be written as, $\left(\{v_j\}_{j \in [0,n)}, \{(K^i_0, K^i_1)\}_{i \in [h]}\right)$, where (K^i_0, K^i_1) are the XOR of the values at the even and odd nodes at level i respectively. For the leaf nodes, instead of XOR, addition over $\mathbb{F}_{p^r}$ is computed. Then, parties run h instances of $\mathcal{F}_{\mathsf{OT}}$ with P_A sending $\bar{\alpha}^i$ for $i \in [1, h]$ and P_B sending (K^i_0, K^i_1) as the input. The outputs from $\mathcal{F}_{\mathsf{OT}}$ give P_A $(\boldsymbol{w}[i])_{i \neq \alpha}$ as

$\boldsymbol{w}[i] = \boldsymbol{v}[i]$ for $i \neq \alpha$. The only thing that remains is to obtain $\boldsymbol{w}[\alpha] = \boldsymbol{v}[\alpha] + \Delta \cdot \beta$. Recall that parties already have $\langle \alpha \rangle$. Therefore, P_B can send $\mathsf{K}[\beta] - \sum_{i=1}^{m} \boldsymbol{v}[i]$ to P_A, which can compute $\boldsymbol{w}[\alpha]$ as:

$$\boldsymbol{w}[\alpha] = \mathsf{M}[\beta] - (\mathsf{K}[\beta] - \sum_{i=1}^{m} \boldsymbol{v}[i]) - \sum_{i \neq \alpha} \boldsymbol{v}[i] = \mathsf{M}[\beta] - \mathsf{K}[\beta] + \boldsymbol{v}[\alpha] = \boldsymbol{v}[\alpha] + \Delta \cdot \beta$$

To check for malicious behaviour, we run the consistency check from Wolverine, which is described here for completeness. The idea is for parties to samples uniformly random values $\chi_0, \dots, \chi_{n-1} \in \mathbb{F}_{p^r}$ and checking the randomised version of the VOLE as:

$$\sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{w}[i] = \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{v}[i] + \Delta \cdot \beta \cdot \chi_\alpha$$

P_B cannot compute this however, as it does not know α, β. Therefore, parties can use $\mathcal{F}_{\mathsf{sVOLE}}$ to generate $Z, Y \in \mathbb{F}_{p^r}$ such that $Z = Y + \Delta \cdot \beta \cdot \chi_\alpha$. Since $\beta \cdot \chi_\alpha$ lies in $\mathbb{F}_{p^r}$ as opposed to $\mathbb{F}_p$, we cannot directly use $\mathcal{F}_{\mathsf{sVOLE}}$. Instead, χ_α can be viewed as $(\chi_{\alpha,0}, \dots, \chi_{\alpha,r-1} \in \mathbb{F}_p^r$. They can then use r calls to $\mathcal{F}_{\mathsf{sVOLE}}$ to which gives P_A $\boldsymbol{z}$ and P_B $\boldsymbol{y}$ such that $\boldsymbol{z} = \boldsymbol{y} + \Delta \cdot \beta \cdot \chi_\alpha$. Let $Z = \sum_{i=0}^{r-1} \boldsymbol{z}[i] \cdot \mathsf{X}^i$ and $Y = \sum_{i=0}^{r-1} \boldsymbol{y}[i] \cdot \mathsf{X}^i$. This means P_A can compute $V_A = \sum_{i=0}^{n-1} \chi_i \boldsymbol{w}[i] - Z$ and P_B can compute $V_B = \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{v}[i] - Y$. The final step is to call $\mathcal{F}_{\mathsf{EQ}}$ with V_A, V_B, which returns either success or abort.

Functionality $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$

Parameters: An extension field $\mathbb{F}_{p^r}$, length m, and party identifiers P_A, P_B.
Initialise: On receiving Init from P_A, and (Init, Δ) from P_B, store the global key $\Delta \in \mathbb{F}_{p^r}$, and ignore all subsequent Init commands.
Extend: On receiving $(\mathsf{Extend}, m, \alpha, \beta)$ from P_A and Extend from P_B, where $m = 2^h$, do:

1. If P_B is honest, sample $\boldsymbol{v} \leftarrow \mathbb{F}_{p^r}^m$. Else, receive $\boldsymbol{v}$ from $\mathcal{A}$.
2. Set $\boldsymbol{u} \in \mathbb{F}_p^m$ such that $\{\boldsymbol{u}[i]\}_{i \neq \alpha} = 0$ and $\boldsymbol{u}[\alpha] = \beta$. Compute $\boldsymbol{w} = \boldsymbol{v} + \Delta \cdot \boldsymbol{u} \in \mathbb{F}_p^m$.
3. If P_B is corrupt, receive a set $I \subseteq [0, m)$ from $\mathcal{A}$. If $\alpha \in I$, send success to P_B and continue. Else, send abort to both parties, output α to P_B and abort.
4. Output $(\boldsymbol{u}, \boldsymbol{w})$ to P_A and $\boldsymbol{v}$ to P_B.

Global-key query: If P_A is corrupted, receive $(\mathsf{guess}, \Delta')$ from $\mathcal{A}$ with $\Delta' \in \mathbb{F}_{p^r}$. If $\Delta' = \Delta$, send success to P_A and ignore any subsequent global-key query. Else, send abort to both parties and abort.

Fig. 21: Functionality for a chosen-input sVOLE

Protocol $\Pi^{\mathsf{ci}}_{\mathsf{spsVOLE}}$

Parameters: An extension field $\mathbb{F}_{p^r}$, party identifiers P_A, P_B.
Initialise: Executed only once between a pair of parties. P_A sends Init and P_B sends (Init, Δ) to $\mathcal{F}_{\mathsf{sVOLE}}$.
Extend: Can be executed multiple times. P_A has input (α, β), where $\alpha \in [0, n)$, $\beta \in \mathbb{F}_p^*$.

1. P_A and P_B send Extend to $\mathcal{F}_{\mathsf{sVOLE}}$, which returns $(a, c) \in \mathbb{F}_p \times \mathbb{F}_{p^r}$ to P_A and $b \in \mathbb{F}_{p^r}$ to P_B such that $c = \Delta \cdot a + b$.
2. P_A sets $\delta = c$ and sends $a' = \beta - a \in \mathbb{F}_p$ to P_B which computes $\gamma = b - \Delta \cdot a'$, forming $\langle\beta\rangle$. P_A defines $\boldsymbol{u} \in \mathbb{F}_p^m$ as the single-point vector such that $\boldsymbol{u}[\alpha] = \beta$.
3. P_B samples $s \leftarrow \{0,1\}^k$, runs $\mathsf{GGM}(1^m, s)$ to get $(\{v_j\}_{j\in[0,m)}, \{(K_0^i, K_1^i)\}_{i\in[1,h]})$ and sets $\boldsymbol{v}[j] = v_j$ for $j \in [0, m)$. P_A lets $\bar{\alpha_i}$ be the compliment of the ith bit of the binary representation of α. For $i \in [1, h]$, P_A sends $\bar{\alpha_i} \in \{0,1\}$ to $\mathcal{F}_{\mathsf{OT}}$ and P_B sends (K_0^i, K_1^i) to $\mathcal{F}_{\mathsf{OT}}$. P_A receives $K^i_{\bar{\alpha_i}}$, which then runs $\{v_j\}_{j\neq\alpha} = \mathsf{GGM}'(\alpha, \{K^i_{\bar{\alpha_i}}\}_{i\in[1,h]})$.
4. P_B sends $d = \gamma - \sum_{i\in[0,m)} \boldsymbol{v}[i] \in \mathbb{F}_p^r$ to P_A. Then, P_A defines $\boldsymbol{w} \in \mathbb{F}_{p^r}^m$ as the vector with $\boldsymbol{w}[i] = v_i$ for $i \neq \alpha$ and $\boldsymbol{w}[\alpha] = \delta - \Big(d + \sum_{i\neq\alpha} \boldsymbol{w}[i]\Big)$. Note that $\boldsymbol{w} = \Delta \cdot \boldsymbol{u} + \boldsymbol{v}$.

Consistency check:

1. Both parties send (Extend, r) to $\mathcal{F}_{\mathsf{sVOLE}}$, which returns $(\boldsymbol{x}, \boldsymbol{z}) \in \mathbb{F}_p^r \times \mathbb{F}_{p^r}^r$ to P_A and $\boldsymbol{y}^* \in \mathbb{F}_{p^r}^r$ to P_B such that $\boldsymbol{z} = \Delta \cdot \boldsymbol{x} + \boldsymbol{y}^*$.
2. P_A samples $\chi_i \leftarrow \mathbb{F}_{p^r}$ for $i \in [0, m)$ and writes $\chi_\alpha = \sum_{i=0}^{r-1} \chi_{\alpha,i} \cdot \mathsf{X}^i$. Let $\chi_\alpha = (\chi_{\alpha,0}, \ldots, \chi_{\alpha,r-1}) \in \mathbb{F}_p^r$. P_A then computes $\boldsymbol{x}^* = \beta \cdot \boldsymbol{\chi}_\alpha - \boldsymbol{x} \in \mathbb{F}_p^r$ and sends $(\{\chi_i\}_{i\in[0,m)}, \boldsymbol{x}^*)$ to P_B, which computes $\boldsymbol{y} = \boldsymbol{y}^* - \Delta \cdot \boldsymbol{x}^* \in \mathbb{F}_{p^r}^r$.
3. P_A computes $\mathcal{Z} = \sum_{i=0}^{r-1} \boldsymbol{z}[i] \cdot \mathsf{X}^i \in \mathbb{F}_{p^r}$ and $V_A = \sum_{i=0}^{m-1} \chi_i \cdot \boldsymbol{w}[i] - \mathcal{Z} \in \mathbb{F}_{p^r}$, while P_B computes $\mathcal{Y} = \sum_{i=1}^{r-1} \boldsymbol{y}[i] \cdot \mathsf{X}^i \in \mathbb{F}_{p^r}$ and $V_B = \sum_{i=0}^{m-1} \chi_i \cdot \boldsymbol{v}[i] - \mathcal{Y} \in \mathbb{F}_{p^r}$. Then P_A sends V_A to $\mathcal{F}_{\mathsf{EQ}}$, and P_B sends V_B to $\mathcal{F}_{\mathsf{EQ}}$. If either party receives false or abort from $\mathcal{F}_{\mathsf{EQ}}$, it aborts.
4. P_A outputs $(\boldsymbol{u}, \boldsymbol{w})$ and P_B outputs $\boldsymbol{v}$.

Fig. 22: Protocol for single-point sVOLE

Theorem 7. *If G and G' are pseudorandom generators, then $\Pi^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ UC-realises $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ in the $(\mathcal{F}_{\mathsf{sVOLE}}, \mathcal{F}_{\mathsf{EQ}}, \mathcal{F}_{\mathsf{OT}})$-hybrid model. In particular, no PPT environment $\mathcal{Z}$ can distinguish the real-world execution from the ideal-world one except with probability at most $1/p^r + \mathsf{negl}(k)$.*

Proof. The first part deals with the case of a malicious P_A and the second one with a malicious P_B. In each case we construct a PPT simulator $\mathcal{S}$ which is given access to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ that runs the $\mathcal{A}$ as a subroutine and emulates the functionalities $\mathcal{F}_{\mathsf{sVOLE}}, \mathcal{F}_{\mathsf{EQ}}, \mathcal{F}_{\mathsf{OT}}$. We implicitly assume that the simulator $\mathcal{S}$ passes all the communication between the $\mathcal{A}$ and the environment $\mathcal{Z}$.

The $\mathcal{S}$ for a malicious P_A behaves exactly the same as it does in Wolverine [WYKW21]. The interesting case is when P_B is malicious.

Malicious P_A: Every time the extend procedure is run with inputs (m, α, β), $\mathcal{S}$ interacts with $\mathcal{A}$ as follows:

1. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{sVOLE}}$ and records the values (a, c) that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{sVOLE}}$. When $\mathcal{A}$ sends the message $a' \in \mathbb{F}_p$, then $\mathcal{S}$ sets $\beta = a' + a \in \mathbb{F}_p$ and $\delta = c$.
2. For $i \in [1, h)$, $\mathcal{S}$ samples $K^i \leftarrow \{0,1\}^\kappa$; it also samples $K^h \leftarrow \mathbb{F}_{p^r}$. Then for $i \in [1, h]$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{OT}}$ by receiving $\bar{\alpha}_i \in \{0,1\}$ from $\mathcal{A}$, and returning $K^i_{\bar{\alpha}_i} = K^i$ to $\mathcal{A}$. It sets $\alpha = \alpha_1 \dots \alpha_h$ and defines $\boldsymbol{u} \in \mathbb{F}_p^m$ as the vector that is 0 everywhere except that $\boldsymbol{u}[\alpha] = \beta$. Next, $\mathcal{S}$ computes $\{v_j\}_{j \neq \alpha} = \mathsf{GGM}'(\alpha, \{K^i_{\bar{\alpha}_i}\}_{i \in [1,h]})$.
3. $\mathcal{S}$ picks $d \leftarrow \mathbb{F}_{p^r}$ and sends it to $\mathcal{A}$. Then $\mathcal{S}$ defines $\boldsymbol{w}$ as the vector of length m with $\boldsymbol{w}[i] = v_i$ for $i \neq \alpha$ and $\boldsymbol{w}[\alpha] = \delta - (d + \sum_{i \neq \alpha} \boldsymbol{w}[i])$.
4. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{sVOLE}}$ by recording $(\boldsymbol{x}, \boldsymbol{z})$ from $\mathcal{A}$.
5. $\mathcal{S}$ receives $\{\chi_i\}_{i \in [0,n)}$ and $\boldsymbol{x}^* \in \mathbb{F}_p^r$ from $\mathcal{A}$, and sets $\boldsymbol{x}' = \boldsymbol{x}^* + \boldsymbol{x} \in \mathbb{F}_p^r$ and $x' = \sum_{i=0}^{r-1} \boldsymbol{x}'[i] \cdot \mathsf{X}^i$.
6. $\mathcal{S}$ records $V_A \in \mathbb{F}_p^r$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{EQ}}$. It then computes $V'_A = \sum_{i=0}^{m-1} \chi_i \cdot \boldsymbol{w}[i] - \sum_{i=0}^{r-1} \boldsymbol{z}[i] \cdot \mathsf{X}^i \in \mathbb{F}_p^r$ and does:
   (a) If $x' = \beta \cdot \chi_\alpha$, then $\mathcal{S}$ checks whether $V_A = V'_A$. If so, $\mathcal{S}$ sends true to $\mathcal{A}$, and sends $\boldsymbol{u}, \boldsymbol{w}$ to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$. Else, $\mathcal{S}$ sends abort to $\mathcal{A}$ and aborts.
   (b) Else, $\mathcal{S}$ computes $\Delta' = (V'_A - V_A)/(\beta \cdot \chi_\alpha - x') \in \mathbb{F}_p^r$ and sends a global-key query $(\mathsf{guess}, \Delta')$ to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$. If $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ returns $\mathsf{success}$, $\mathcal{S}$ sends true to $\mathcal{A}$, and sends $\boldsymbol{u}, \boldsymbol{w}$ to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$. Else, $\mathcal{S}$ sends abort to $\mathcal{A}$ and aborts.
7. Whenever $\mathcal{A}$ sends a global-key query to $(\mathsf{guess}, \Delta')$ to the functionality $\mathcal{F}_{\mathsf{sVOLE}}$, $\mathcal{S}$ forwards the query to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ and returns the answer to $\mathcal{A}$. If the answer is abort, $\mathcal{S}$ aborts.

Malicious P_B: The simulator $\mathcal{S}$ interacts with $\mathcal{A}$ as follows. First, it simulates the initialisation step by recording the global-key $\Delta \in \mathbb{F}_{p^r}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{sVOLE}}$. Then, every time (Extend, m) is called, $\mathcal{S}$ does:

1. $\mathcal{S}$ records $b \in \mathbb{F}_{p^r}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{sVOLE}}$. Then $\mathcal{S}$ samples $a' \leftarrow \mathbb{F}_p$ and sends it to $\mathcal{A}$. Next, $\mathcal{S}$ computes $\gamma = b - \Delta \cdot a'$, and samples $\beta \in \mathbb{F}_p^*$ and sets $\delta = \gamma + \Delta \cdot \beta$.
2. $\mathcal{S}$ records the values $\{(K^i_0, K^i_1)\}_{i \in [1,h]}$ sent to $\mathcal{F}_{\mathsf{OT}}$ by $\mathcal{A}$.
3. $\mathcal{S}$ receives $d \in \mathbb{F}_{p^r}$ from $\mathcal{A}$. Then, for each $\alpha \in [0, n)$, it computes a vector $\boldsymbol{w}_\alpha$ as follows:
   (a) Execute $\{v^\alpha_j\}_{j \neq \alpha} = \mathsf{GGM}'(\alpha, \{K^i_{\bar{\alpha}_i}\}_{i \in [1,h]})$ and set $\boldsymbol{w}_\alpha[i] = v^\alpha_i$ for $i \neq \alpha$.
   (b) Compute $\boldsymbol{w}_\alpha[\alpha] = \delta - (d + \sum_{i \neq \alpha} \boldsymbol{w}_\alpha[i])$.
4. $\mathcal{S}$ records the vector $\boldsymbol{y}^*$ sent to $\mathcal{F}_{\mathsf{sVOLE}}$ by $\mathcal{A}$.
5. $\mathcal{S}$ samples $\chi_i \leftarrow \mathbb{F}_{p^r}$ for $i \in [0, n)$ and $\boldsymbol{x}^* \leftarrow \mathbb{F}_{p^r}$, and sends them to $\mathcal{A}$. Then $\mathcal{S}$ computes $\boldsymbol{y} = \boldsymbol{y}^* - \Delta \cdot x^*$.
6. $\mathcal{S}$ computes $Y = \sum_{i=0}^{r-1} \boldsymbol{y}[i] \cdot \mathsf{X}^i$. It then records V_B sent to $\mathcal{F}_{\mathsf{EQ}}$ by $\mathcal{A}$. Then, $\mathcal{S}$ computes a set $I \subseteq [0, n)$ as follows:
   (a) For $\alpha \in [0, n)$, compute $V^\alpha_A = \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{w}_\alpha[i] - \Delta \cdot \beta \cdot \chi_\alpha - Y$.

(b) Define $I = \{\alpha \in [0, n) | V_A^\alpha = V_B\}$.
$\mathcal{S}$ sends I to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$. If it returns $(\mathsf{abort}, \alpha^*)$, where α^* was the value used by an honest P_A, $\mathcal{S}$ uses α^* to compute the correct $V_A^{\alpha^*}$ and sends $(\mathsf{false}, V_A^{\alpha^*})$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{EQ}}$, and then aborts. Else, $\mathcal{S}$ sends (true, V_B) to $\mathcal{A}$.

7. $\mathcal{S}$ chooses an arbitrary $\alpha \in I$ and computes a vector $\boldsymbol{v}$ as follows:
(a) Set $\boldsymbol{v}[i] = \boldsymbol{w}_\alpha[i]$ for $i \in [0, n)_{i \neq \alpha}$.
(b) Set $\boldsymbol{v}[\alpha] = \gamma - d - \sum_{i \neq \alpha} \boldsymbol{v}[i]$.
$\mathcal{S}$ sends $\boldsymbol{v}$ to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ and outputs whatever $\mathcal{A}$ outputs.

We first consider the view of the adversary $\mathcal{A}$ in the ideal-world execution and the real-world execution. The values a' and $\boldsymbol{x}^*$ simulated by $\mathcal{S}$ have the same distribution as the real values, which are masked by a uniformly random element/vector output by $\mathcal{F}_{\mathsf{sVOLE}}$. The set I extracted by $\mathcal{S}$ corresponds to a selective failure attack on the output index α^* of P_A. If $\mathcal{S}$ receives an abort from $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$, it means $\alpha^* \notin I$. In the real protocol, P_A aborts if $V_A^{\alpha^*} \neq V_B$. Therefore, $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ only aborts if the real-world protocol aborts.

Since α is given as input by P_A instead of being chosen at random, $\mathcal{S}$ cannot pick a random $\alpha \in [0, n) \backslash I$, as it does in [WYKW21]. It needs to send the V_A that corresponds to the V_A that an honest P_A would have sent in the real-world. In order to facilitate this, the $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ functionality is designed to return the α^* that was used in the real protocol, in the case of an abort. This means the distribution of V_A sent by the $\mathcal{S}$ is indistinguishable from the real world distribution.

D.2 From $\Pi^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ to $\Pi^{\mathsf{prog}}_{\mathsf{VOLE}}$

The final step is to go from single-point VOLE to standard (programmable) VOLE. Here, we will realize $\mathcal{F}^{\mathsf{prog}}_{\mathsf{VOLE}}$ instantiated with a particular expansion function $\mathsf{Expand} : S \to \mathbb{F}_p^m$, based on a variant of the LPN assumption.

t-regular vector: A t-regular vector $\boldsymbol{e}$ is defined as a set of t vectors $\boldsymbol{e}_1, \dots, \boldsymbol{e}_t$ concatenated, wherein each $\boldsymbol{e}_i$ is a sparse vector with Hamming weight one.

We use the dual form of LPN over $\mathbb{F}_p$, with a regular error distribution. This has also been considered in previous works [BCG+19a,WYKW21].

Definition 1 (Regular Dual-LPN assumption). *Let $H \in \mathbb{F}_p^{k \times m}$, and consider the following game $G_b(\kappa)$ with a PPT adversary $\mathcal{A}$, parameterised by a bit b and the security parameter κ:*

1. *Sample a random, t-regular vector $\boldsymbol{e} \in \mathbb{F}_p^k$.*
2. *If $b = 1$, let $\boldsymbol{y} = H \cdot \boldsymbol{e}$, else sample $\boldsymbol{y} \leftarrow \mathbb{F}_p^m$.*
3. *Send $\boldsymbol{y}$ to $\mathcal{A}$, which then outputs a bit b' (in case of abort, define the output of $\mathcal{A}$ to be $\perp$).*

The assumption states that $\mathcal{A}$ has negligible advantage in distinguishing $G_0(\kappa)$ and $G_1(\kappa)$.

Expansion function: Fix a dual-LPN matrix $H \in \mathbb{F}_p^{m \times k}$. We consider a seed space $S \subset \mathbb{F}_p^k$ consisting of t-regular vectors in $\mathbb{F}_p^k$. We define the LPN-based expand function

$$\mathsf{Expand}^{\mathsf{LPN}} : S \to \mathbb{F}_p^m, \qquad \mathsf{Expand}^{\mathsf{LPN}}(\boldsymbol{e}) = H \cdot \boldsymbol{e}$$

Functionality $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$

Parameters: Finite field $\mathbb{F}_{p^r}$, and expansion function $\mathsf{Expand} : S \to \mathbb{F}_p^m$ with seed space S and output length m.
The functionality runs between parties P_A and P_B.
Initialise: On receiving Init from P_A, and (Init, Δ) from P_B, store Δ, and ignore all subsequent Init commands.
Extend: On receiving Extend from P_B and $(\mathsf{Extend}, \mathsf{seed})$ from P_A, where $\mathsf{seed} \in S$:

1. Compute $\boldsymbol{u} = \mathsf{Expand}(\mathsf{seed})$.
2. Sample $\boldsymbol{v} \leftarrow \mathbb{F}_{p^r}^m$ and compute $\boldsymbol{w} = \boldsymbol{u} \cdot \Delta + \boldsymbol{v}$.
3. If P_B is corrupt, receive a set I from $\mathcal{A}$. If $\mathsf{seed} \in I$, send $\mathsf{success}$ to P_B and continue. Else, send abort to both parties, output seed to P_B and abort.
4. Output $(\boldsymbol{u}, \boldsymbol{w})$ to P_A and $\boldsymbol{v}$ to P_B.

Corrupt parties: If P_B is corrupt, $\boldsymbol{v}$ may be chosen by $\mathcal{A}$. For a corrupt P_A, $\mathcal{A}$ can choose $\boldsymbol{w}$ (and then $\boldsymbol{v}$ is recomputed accordingly).
Global key query: If P_A is corrupted, receive $(\mathsf{guess}, \Delta')$ from $\mathcal{A}$ with $\Delta' \in \mathbb{F}_{p^r}$. If $\Delta' = \Delta$, send $\mathsf{success}$ to P_A and ignore any subsequent global key query. Else, send abort to both parties and abort.

Fig. 23: Functionality for programmable VOLE

Overview of $\Pi_{\mathsf{VOLE}}^{\mathsf{prog}}$: The first step is to execute $\mathcal{F}_{\mathsf{sVOLE}}$, which gives Δ to P_B on Init and gives $\langle \boldsymbol{u} \rangle \in \mathbb{F}_p^k$. In addition, they run $\mathcal{F}_{\mathsf{spsVOLE}}^{\mathsf{ci}}$ t times, to get vectors of authenticated values. Vectors are denoted by $\boldsymbol{e}_i$, each of them is of length m/t and has exactly one nonzero entry. Parties use the public matrix H to convert these to a vector of authenticated values of length m.

Under the regular dual-LPN assumption, the values appear pseudorandom to P_B, if the seed S was sampled at random. Note, however, that the protocol perfectly realizes $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ without relying on dual-LPN, because $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ itself is defined in terms of the expansion function. Therefore, it is only when using $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ to instantiate our preprocessing protocol, where LPN comes into play.

E Details for Dynamic SPDZ

Protocol Variants. If supporting a dynamic committee for the online phase is not a requirement, we could modify our scheme by shifting the verification of

Protocol $\Pi^{\mathsf{prog}}_{\mathsf{VOLE}}$

Parameters: Extension field $\mathbb{F}_{p^r}$, length m, noise weight t, LPN dimension n and matrix $H \in \mathbb{F}^{m\times k}$, and party identifiers P_A, P_B. $q = k/t$.
Intialise: Executed only once between two parties. P_A, P_B send Init, (Init, Δ) respectively to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$.
Extend: On input seed from P_A, where seed describes a t-regular vector $\boldsymbol{e} \in \mathbb{F}_p^k$:

1. For $i \in [1, t]$, P_A and P_B send (Extend, q) to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$, which returns $(\boldsymbol{e}_i, \boldsymbol{c}_i)$ to P_A and $\boldsymbol{b}_i$ to P_B such that $\boldsymbol{c}_i = \Delta \cdot \boldsymbol{e}_i + \boldsymbol{b}_i \in \mathbb{F}_{p^r}^q$ and $\boldsymbol{e}_i \in \mathbb{F}_p^q$ has exactly one nonzero entry. If either party receives abort from $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ in any of these executions, it aborts.
2. P_A defines $\boldsymbol{e} = (\boldsymbol{e}_1, \ldots, \boldsymbol{e}_t) \in \mathbb{F}_p^k$ and $\boldsymbol{c} = (\boldsymbol{c}_1, \ldots, \boldsymbol{c}_t) \in \mathbb{F}_{p^r}^k$. Then P_A computes $\boldsymbol{x} = H \cdot \boldsymbol{e}$ and $\boldsymbol{z} = H \cdot \boldsymbol{c}$. P_B defines $\boldsymbol{b} = (\boldsymbol{b}_1, \ldots, \boldsymbol{b}_t) \in \mathbb{F}_{p^r}^k$ and computes $\boldsymbol{y} = H \cdot \boldsymbol{b} \in \mathbb{F}_{p^r}^m$.
3. P_A outputs $(\boldsymbol{s}, \mathsf{M}[\boldsymbol{s}]) = (\boldsymbol{x}, \boldsymbol{z}) \in \mathbb{F}_p^m \times \mathbb{F}_{p^r}^m$. P_B updates $\boldsymbol{v}$ by setting $\boldsymbol{v} = \boldsymbol{y} \in \mathbb{F}_{p^r}^m$, and outputs $\mathsf{K}[\boldsymbol{s}] = \boldsymbol{y} \in \mathbb{F}_{p^r}^m$.

Fig. 24: Protocol to extend $\mathsf{spsVOLE}$

multiplication triples to the preprocessing. This reduces the overhead of the online phase, and is essentially a regular SPDZ protocol run with our preprocessing. We simply authenticate all the c, c' components of the triples during the preprocessing phase, and then run a standard pairwise verification procedure [DPSZ12] to check one triple using another. This effectively moves the 4 extra openings in our online phase to the preprocessing, leading to an online phase with the same cost as SPDZ, although now the preprocessing has $O(N)$ complexity.

Of course, if the entire preprocessing committee $\mathcal{P}_{\mathsf{main}}$ does this, this introduces a lot more interaction from parties who may not have been involved in the online phase. Another option is to run this verification in the online committee $\mathcal{P}_{\mathsf{curr}}$ at the *start* of the online phase, after $\mathcal{P}_{\mathsf{curr}}$ has been elected, but possibly before the desired computation has been determined.

E.1 Security Analysis

Lemma 5 (Lemma 1, restated). *Suppose $\mathcal{A}$ introduces additive errors of the form $\delta_a^{j,i}, \delta_b^{j,i} \neq 0$, for malicious parties P_j and honest P_i in $\mathcal{F}_{\mathsf{Prep}}$, and in $\Pi_{\mathsf{SPDZ\text{-}Online}}$ additive errors $\delta_c, \delta_{c'} \neq 0$ when authenticating triples a, b, c and a', b', c' respectively. If any errors are non-zero, then the Verification phase in $\Pi_{\mathsf{SPDZ\text{-}Online}}$ fails with probability less than $2/p$.*

Proof. Consider a multiplication gate at layer k, wherein the multiplications carried out are $z_k = x_k \cdot y_k$, and $rz_k = rx_k \cdot y_k$. Note that rx, ry will have errors from the layer $k-1$. $\mathcal{A}$ can insert an additive error when c, c' are authenticated and these are denoted by $\delta_c, \delta_{c'}$ respectively. The errors $\delta_c, \delta_{c'}$ are going to be consistent with the MACs as well, due to the way c and c' are authenticated.

They will not get caught during the MAC Check, which is why we compute the randomised circuit in addition to using MACs.

$\mathcal{A}$ can insert an additive error in the output of a multiplication, and the error is indexed by ε^k for layer k. Let the error introduced by $\mathcal{A}$ in computing $[\![r]\!] \cdot [\![x]\!]$ be denoted by ε_1, ignoring the superscript for simplicity. The errors in computing $[\![x]\!] \cdot [\![y]\!]$ and $[\![r]\!] \cdot [\![v]\!]$, where $[\![v]\!]$ is the input, are indicated by ε_2, and ε_4 respectively. Finally, computing $[\![rz]\!]$ is done by computing $[\![rx]\!] \cdot [\![y]\!]$, and the error introduced is ε_3.

In addition, we need to account for the errors in the triples used to carry out these multiplications. Parties receive a triple of the form $[\![a]\!], [\![b]\!], [c]$ from $\mathcal{F}_{\mathsf{Prep}}$ in the online phase. The $[c]$ part of the triple has additive errors due to using an inconsistent $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$, as explained in Section 3.2. These errors can be viewed as $[\hat{c}] = [c] + \{\delta_a^{j,i} \cdot b^i + \delta_b^{j,i} \cdot a^i\}$, for $j \in \mathcal{P}_A, i \in \mathcal{P}_H$. On top of this, parties authenticate $[c]$ in the online phase before processing the multiplication gates, wherein $\mathcal{A}$ can introduce another additive error, denoted by δ_c. We let $\hat{\varepsilon} = \varepsilon + \{\delta_a^{j,i} \cdot b^i + \delta_b^{j,i} \cdot a^i\} + \delta_c$, for $j \in \mathcal{P}_{\mathcal{A}}, i \in \mathcal{P}_H$ and a multiplication that used a triple $[\![a]\!], [\![b]\!], [\![c]\!]$. Therefore, the values computed will be:

1. $[\![rx]\!] = [\![r]\!] \cdot [\![x]\!] \to [\![rx + \hat{\varepsilon}_1]\!]$ (layer $k-1$)
2. $[\![z]\!] = [\![x]\!] \cdot [\![y]\!] \to [\![xy + \hat{\varepsilon}_2]\!]$
3. $[\![rz]\!] = [\![rx]\!] \cdot [\![y]\!] \to ([\![rx + \hat{\varepsilon}_1]\!] \cdot [\![y]\!]) + \hat{\varepsilon}_3$
4. $[\![rv]\!] = [\![r]\!] \cdot [\![v]\!] \to [\![rv + \hat{\varepsilon}_4]\!]$

Note: The MACs on these values have been checked for consistency by this point (and we ignore here the case that an invalid MAC was successfully forged).

Parties sample random values $\alpha_1, \ldots, \alpha_N$ and $\beta_1, \ldots, \beta_M$ to compute a random linear combination on the actual values and their randomised variants. This is computed for all the inputs to the circuit, and the outputs of every multiplication gate. The random linear combination of the actual values is denoted by $[\![w]\!]$ and the randomised one is denoted by $[\![u]\!]$. The idea is that parties will then open $[\![r]\!]$, and compute $[\![u]\!] - r \cdot [\![w]\!]$. Ideally, this value would be equal to 0. We calculate and show that the probability that $\mathcal{A}$ injects errors as detailed earlier, and does not get caught in the check is upper bounded by $2/p$.

Parties start by computing $[\![u]\!], [\![w]\!]$ as,

$$[\![u]\!] = \sum_{i=1}^{N} \alpha_i \cdot \left((rx + \hat{\varepsilon}_1^i) \cdot y + \hat{\varepsilon}_3^i\right) + \sum_{i=1}^{M} \beta_i \cdot (rv + \hat{\varepsilon}_4^i)$$

$$[\![w]\!] = \sum_{i=1}^{N} \alpha_i \cdot (x \cdot y + \hat{\varepsilon}_2^i) + \sum_{i=1}^{M} \beta_i \cdot v$$

$$\begin{aligned}
[\![u]\!] - r \cdot [\![w]\!] &= \sum_{i=1}^{N} \alpha_i \cdot \left((rx + \hat{\varepsilon}_1^i) \cdot y + \hat{\varepsilon}_3^i\right) + \sum_{i=1}^{M} \beta_i \cdot (rv + \hat{\varepsilon}_4^i) \\
&- r \cdot \left(\sum_{i=1}^{N} \alpha_i \cdot (x \cdot y + \hat{\varepsilon}_2^i) + \sum_{i=1}^{M} \beta_i \cdot v \right) \\
&= \sum_{i=1}^{N} \alpha_i (\hat{\varepsilon}_1^i \cdot y + \hat{\varepsilon}_3^i - r \cdot \hat{\varepsilon}_2^i) + \sum_{i=1}^{M} \beta_i \cdot \hat{\varepsilon}_4^i
\end{aligned}$$

The analysis, below, is similar to [CGH+18]. The intuition about why the additional errors introduced in the triples do not give the adversary any additional advantage is as follows. Errors in $[c]$ received from $\mathcal{F}_{\mathsf{Prep}}$ are of the form $\delta_a^{j,i} \cdot b^i$, for when a corrupt P_j interacts with an honest P_i. Since the adversary does not know the honest P_i's share b^i, this is going to be a random additive error that is not known to $\mathcal{A}$. At this point, if the triple was authenticated in the same round as the computing the multiplication, in other words opening $x - a, y - b$ along with opening $l + c$, $\mathcal{A}$ can wait until it receives $x - a, y - b$ in the clear. Using these values, it can choose a δ_c such that this results in an error of the form $x \cdot \delta_b^{j,i}$, a selective failure attack.

When we later authenticate the triple, $\mathcal{A}$ has still learnt no information about a or b (since we haven't yet opened $x - a, y - b$), so any error δ_c that $\mathcal{A}$ injects will also be an independent, additive error.

The analysis can be split into two cases:

Case 1: There exists some index i such that $\hat{\varepsilon}_4^i \neq 0$. Let m be the smallest one for which it holds. $[\![u]\!] - r \cdot [\![w]\!] = 0$ if and only if:

$$\beta_m = \left(- \sum_{i=1}^{N} \alpha_i \left(\hat{\varepsilon}_1^i \cdot y + \hat{\varepsilon}_3^i - r \cdot \hat{\varepsilon}_2^i\right) + \sum_{i \neq m}^{M} \beta_i \cdot \hat{\varepsilon}_4^i \right) \cdot (\hat{\varepsilon}_4^m)^{-1} \tag{1}$$

Since β_m is chosen independently and is uniformly distributed over $\mathbb{F}$, this holds with probability at most $1/p$.

Case 2: All $\hat{\varepsilon}_4^i = 0$, meaning there was no cheating in the triple used to compute $[\![rv]\!]$ or in the output of the multiplication. Assuming the multiplication wires in the succeeding layers were tampered, $\hat{\varepsilon}_2^i \neq 0$ and/or $\hat{\varepsilon}_3^i \neq 0$. Let k be the wire for this, and it holds that $\hat{\varepsilon}_1^k = 0$ for the wire as no input was tampered with before this point. Therefore, $[\![u]\!] - r \cdot [\![w]\!] = 0$ if and only if,

$$\alpha_k \cdot (\hat{\varepsilon}_3^k - r \cdot \hat{\varepsilon}_2^k) = - \sum_{i=1}^{N} \alpha_i \left(\hat{\varepsilon}_1^i \cdot y + \hat{\varepsilon}_3^i - r \cdot \hat{\varepsilon}_2^i\right) \tag{2}$$

There are two scenarios, one in which $(\hat{\varepsilon}_3^i - r \cdot \hat{\varepsilon}_2^i) = 0$. The probability of this happening is $1/p$ as r is sampled independently and $\mathcal{A}$ does not know r at the time of injecting errors into the triple or even to the output of the multiplication gate. The other scenario is when $(\hat{\varepsilon}_3^i - r \cdot \hat{\varepsilon}_2^i) \neq 0$. Since α_k is chosen independently and not known to $\mathcal{A}$, the probability of this holding is $(1 - 1/p) \cdot 1/p$. Therefore the total probability of the adversary passing the check in Case 2 is bounded by $2/p$.

Theorem 8. *Protocol* $\Pi_{\mathsf{SPDZ\text{-}Online}}$ *UC-securely computes* $\mathcal{F}_{\mathsf{ABB}}$ *in the presence of a static malicious adversary corrupting up to all-but-one of the parties in* $\mathcal{P}_{\mathsf{curr}}$*, in the* $(\mathcal{F}_{\mathsf{Prep}}, \mathcal{F}_{\mathsf{Coin}})$*-hybrid model.*

Proof. We construct a PPT Simulator $(\mathcal{S})$ that run the adversary $(\mathcal{A})$ as a subroutine, and is given access to $\mathcal{F}_{\mathsf{DABB}}$. It internally emulates the functionalities $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}, \mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{Commit}}$ and we implicitly assume that it passes all communication between $\mathcal{A}$ and the environment $(\mathcal{Z})$.

The parties controlled by the $\mathcal{A}$ are indicated by $\mathcal{P}_{\mathcal{A}}$ and the honest parties by $\mathcal{P}_{\mathcal{H}}$. The simulator uses a flag which is set to 1 in case $\mathcal{A}$ is caught cheating before the consistency check happens, and the simulation is carried on. The simulation proceeds as follows:

Malicious $\mathcal{P}_A$:

Init: Receive $(\mathsf{Init}, m_T, m_R)$ from $P_i \in \mathcal{P}_A$ sent to $\mathcal{F}_{\mathsf{Prep}}$, sample a random Δ^i and send it back. Receive $(\mathsf{Rand}, P_i, \mathcal{P}_{\mathsf{curr}}, \mathsf{rcount})$ from each $P_i \in \mathcal{P}_{\mathcal{A}}$, abort if $\mathcal{P}_{\mathsf{curr}}$ is not consistent across calls. Receive $\mathcal{A}$'s shares for $[\![r]\!]$ and store them. Sample random shares for inputs of $\mathcal{P}_H$ and send them to $\mathcal{A}$.

Input:

1. Receive $(\mathsf{Rand}, P_i, \mathcal{P}_{\mathsf{curr}}, \mathsf{rcount})$ from each $P_i \in \mathcal{P}_{\mathcal{A}}$, abort if $\mathcal{P}_{\mathsf{curr}}$ is not consistent across calls. Receive $\mathcal{A}$'s shares for $\langle t \rangle$ and store them. For the honest parties' calls to $\mathcal{F}_{\mathsf{Prep}}$, let $\mathcal{A}$ choose its shares and sample the honest parties' shares at random.
2. Simulate the multiplication step as described below.

Addition, Multiplication by constant: Need not be simulated as they are local operations.

Multiplication:

3. Receive the Trip calls to $\mathcal{F}_{\mathsf{Prep}}$, sample random values for $\mathcal{A}$'s shares of the triples and send them. Receive $\{\delta_a^j\}_{j \in \mathcal{P}_{\mathcal{A}}}, \{\delta_b^j\}_{j \in \mathcal{P}_{\mathcal{A}}}$ from $\mathcal{A}$ and if either $\sum_{j \in \mathcal{P}_A} \delta_a^j \neq 0$ or $\sum_{j \in \mathcal{P}_A} \delta_b^j \neq 0$, set $\mathsf{flag} = 1$.
4. On receiving the Rand call to $\mathcal{F}_{\mathsf{Prep}}$, sample random values for the shares $[\![l]\!], [\![l']\!]$ and send them to $\mathcal{A}$.
5. Receive shares of $[\![x{-}a]\!], [\![y{-}b]\!], [\![rx{-}a']\!], [\![y{-}b']\!]$ and $[l{+}c], [l'{+}c']$. $\mathcal{S}$ computes the correct shares $\mathcal{A}$ was supposed to send, and if they are inconsistent, sets $\mathsf{flag} = 1$. Send random values for shares of $\mathcal{P}_H$.
6. At this point, one of the following things can happen:
   (a) Case 1: The $\mathsf{flag} = 1$ because $\mathcal{A}$ cheated in one of the openings by sending inconsistent values. In this case, $\mathcal{S}$ sends random values on behalf of the honest parties in $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ and aborts at the end of it.

(b) Case 2: The $\mathsf{flag} = 1$ because $\mathcal{A}$ cheated in one of the calls to Trip during a multiplication but not in the openings. $\mathcal{S}$ proceeds with $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ by simulating the Rand call to $\mathcal{F}_{\mathsf{Prep}}$. It then records $\{\sigma^i\}_{\mathcal{P}_{\mathcal{A}}}$ sent to $\mathcal{F}_{\mathsf{Commit}}$ during $\Pi_{\mathsf{SPDZ\text{-}MAC}}$. If $\mathcal{A}$ sent the correct value, it samples shares for the honest parties such that $\sum_{i=1}^{n} \sigma^i = 0$ and sends them to $\mathcal{A}$. It then simulates $\Pi_{\mathsf{SPDZ\text{-}Verify}}$ by sending random values for the honest party shares and aborts at the end of it.

(c) Case 3: The $\mathsf{flag} = 0$, but $\mathcal{A}$ cheats in $\Pi_{\mathsf{SPDZ\text{-}MAC}}$. $\mathcal{S}$ aborts at the end of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$.

(d) Case 4: The $\mathsf{flag} = 0$ and there was no cheating in the MACs, so $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ does not abort, but the $\mathcal{A}$ causes an inconsistency in the randomised circuit computation. This could be in one of four places:

    i. Opening of $[\![r]\!]$.
    ii. $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ on r.
    iii. Opening of $[\![u]\!] - r \cdot [\![w]\!]$.
    iv. $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ on $u - rw$.

    In this case, $\mathcal{S}$ records $\{\sigma^i\}_{\mathcal{P}_{\mathcal{A}}}$ sent to $\mathcal{F}_{\mathsf{Commit}}$ during $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, and samples shares for the honest parties such that $\sum_{i=1}^{n} \sigma^i = 0$ and sends them to $\mathcal{A}$. In $\Pi_{\mathsf{SPDZ\text{-}Verify}}$, send random values for $[\![r]\!]$, and $[\![u]\!] - r \cdot [\![w]\!]$. Abort at the end of the protocol.

(e) Case 5: There was no cheating. $\mathcal{S}$ simulates $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ as in the previous cases when there was no cheating. In $\Pi_{\mathsf{SPDZ\text{-}Verify}}$, it opens a random $[\![r]\!]$ to $\mathcal{A}$ by sending random shares on behalf of $\mathcal{P}_H$. It receives shares of $[\![u]\!] - r \cdot [\![w]\!]$ from $\mathcal{A}$, and samples shares such that $u - r \cdot w = 0$. To compute the outputs, $\mathcal{S}$ sends $\mathcal{A}$'s inputs to $\mathcal{F}_{\mathsf{ABB}}$ using the relevant commands and forwards the output it receives from $\mathcal{F}_{\mathsf{ABB}}$ to $\mathcal{A}$. If $\mathcal{A}$ outputs abort, forward it to $\mathcal{F}_{\mathsf{ABB}}$ and abort.

We now briefly argue that $\mathcal{A}$ cannot distinguish between interacting with the $\mathcal{S}$ and $\mathcal{F}_{\mathsf{Prep}}$, and $\mathcal{F}_{\mathsf{ABB}}$. In the input phase the adversary in both the simulation and the real world, only sees uniformly random values sent by the honest parties since they are masked by a random value not known to $\mathcal{A}$. Addition, addition by a constant, and multiplication by a constant are local operations. In all the calls to $\mathcal{F}_{\mathsf{Prep}}$ using $\mathsf{Trip}, \mathsf{Rand}$, $\mathcal{A}$ is allowed to choose its own share therefore the distribution of the MAC shares on these values between the real world and the simulation is perfectly indistinguishable. Furthermore, the values opened during the multiplication are uniformly random values in the real world, as is the case with the simulation. At the end of the computation, parties run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ on all the values that were opened. In the real world $\mathcal{A}$ is able to cheat with probability at most $2/p$. The check is the one as the one proved in [DKL$^+$13], so we refer the reader to it for a detailed analysis of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$. As shown in Lemma 1, the probability that $\mathcal{A}$ cheats in the calls to $\mathcal{F}_{\mathsf{Prep}}$ and passes the check is $2/p$. Therefore, the overall probability of $\mathcal{A}$ cheating is negligible in p.

F Details for Fluid SPDZ

Protocol Variants: Similar to the variants considered for the Dynamic SPDZ protocol, we can shift some of the costs involved in $\Pi_{\mathsf{Fluid\text{-}Online}}$ to a post-preprocessing phase. We can make the model slightly more restrictive by having the parties communicate the epochs of the online phase in which they would be active, at the end of the preprocessing phase. The committees are now known, which means parties can communicate within their committees to authenticate triples before the function to be computed is determined. Since the triples are authenticated by the time the online computation starts, we do not need $\mathcal{P}_{\mathsf{curr}-1}$ to send the triple to $\mathcal{P}_{\mathsf{curr}}$, saving in terms of communication.

F.1 Security Analysis

We argue security of $\Pi_{\mathsf{Key\text{-}Switch}}$ using the following lemma,

Lemma 6 (Lemma 2 restated). *. If parties in $\mathcal{P}_{\mathsf{curr}}$ follow the protocol, $\Pi_{\mathsf{Key\text{-}Switch}}$ leads to a consistent sharing of $[\![x]\!]^{\mathcal{P}_{\mathsf{curr}}}$, and its transcript is simulatable by random values.*

Proof. Consider a committee $\mathcal{P}_{\mathsf{curr}}$ running $\Pi_{\mathsf{Key\text{-}Switch}}$ on a $[\![\cdot]\!]$-shared value x. They begin by calling $\mathcal{F}_{\mathsf{Prep}}$ to receive a $\langle\cdot\rangle$-shared random t. $\mathcal{P}_{\mathsf{curr}}$ then locally applies Π_{Convert} to get $[\![t]\!]$. Note that,

$$M_j^i = \Delta^j \cdot t + K_i^j, \forall j \in \mathcal{P}_{\mathsf{next}},$$
$$\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t = \sum_{i \in \mathcal{P}_{\mathsf{curr}}} \sum_{j \in \mathcal{P}_{\mathsf{next}}} M_j^i - K_i^j$$
$$(\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t)^j = [M] - [K], \text{ where } M = \sum_{j \in \mathcal{P}_{\mathsf{next}}} M_j^i, K = \sum_{i \in \mathcal{P}_{\mathsf{curr}}} K_i^j$$

Each $P_i \in \mathcal{P}_{\mathsf{curr}}$ can compute a share of M by adding all the MACs it has with parties in $\mathcal{P}_{\mathsf{next}}$. Therefore, by resharing $[M]$, $\mathcal{P}_{\mathsf{next}}$ can compute $[\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t]$. In parallel, $\mathcal{P}_{\mathsf{curr}}$ opens $[\![x+t]\!]$ to $\mathcal{P}_{\mathsf{next}}$, which $\mathcal{P}_{\mathsf{next}}$ uses to compute MAC shares on x under the key $\Delta_{\mathcal{P}_{\mathsf{next}}}$. This is still secure as the adversary does not know t in the clear so $x+t$ is uniformly random. Finally, $\mathcal{P}_{\mathsf{curr}}$ reshares $[x]$ to $\mathcal{P}_{\mathsf{next}}$.

An adversary could cheat in the opening of $[\![x+t]\!]$ or during the resharing of $[M]$ and $[x]$. In the first scenario, since we are opening an authenticated sharing, if the adversary cheats by injecting an additive error, it will get caught in the $\Pi_{\mathsf{Fluid\text{-}MAC}}$ that is run as part of Π_{Open} except with probability $2/p$.

Let the additive error by the adversary during the resharing of $[M]$ be ϵ_M and resharing of $[x]$ be ϵ_x. We show that if $\epsilon_M, \epsilon_x \neq 0$, it will result in an inconsistent MAC on x except with negligible probability. Observe that $\mathcal{P}_{\mathsf{next}}$ will compute,

$$[\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t] = [M] - [K] + \epsilon_M,$$
$$[\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot x] = [\Delta_{\mathcal{P}_{\mathsf{next}}}] \cdot (x+t) - [\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t] + \epsilon_M$$
$$[x] = [x] + \epsilon_x$$

At this point, one of two things can happen with $[\![x]\!]$. The first is, $\mathcal{P}_{\mathsf{next}}$ uses $[\![x]\!]$ to evaluate a multiplication gate. In this case, $[\![x-a]\!]$ will be opened using a triple (a,b,c) by running Π_{Open}, which runs a MAC Check so the adversary will get caught. The other thing that could happen is $[\![x]\!]$ is reconstructed as an output, where before accepting x, a MAC Check on the opened value is run. Therefore, the probability of the adversary cheating in $\Pi_{\mathsf{Key\text{-}Switch}}$ depends on guessing $\Delta_{\mathcal{P}_{\mathsf{next}}}$ to make $\epsilon_M = \Delta_{\mathcal{P}_{\mathsf{next}}} \cdot \epsilon_x$ to cheat in the MAC Check. Since the MAC Check has a probability of $2/p$ of failing, we conclude that the adversary gets caught in $\Pi_{\mathsf{Key\text{-}Switch}}$ except with negligible probability.

F.2 Online Phase Protocol

In Fig. 25, we present the verification protocol, which was described in Section 5.

Protocol $\Pi_{\mathsf{Fluid\text{-}Verify}}$

Usage: Parties in $\mathcal{P}_{i+1}$ want to verify the output wires of multiplication gates of layer l, denoted by $\{z_j, rz_j\}_{j=1}^N$. We assume that $\mathcal{P}_{i+1}$ gets the state $[\![u']\!], [\![w']\!]$ from a previous run of $\Pi_{\mathsf{Fluid\text{-}Verify}}$.

Incremental Verification:

Committee i:

1. Each $P_j \in \mathcal{P}_i$ calls $\mathcal{F}_{\mathsf{Prep}}$ with $(\mathsf{Rand}, \mathcal{P}_i, \mathcal{P}_{i+1}, \mathsf{rcount})$ to receive $\langle s \rangle$.
2. Hand-off: P_j sends the share s^j and MAC M_k^j to each $P_k \in \mathcal{P}_{i+1}$, and runs $\Pi_{\mathsf{Key\text{-}Switch}}$ on $[\![u']\!], [\![w']\!]$.

Committee $i+1$:

3. P_k locally checks $M_k^j = s^j \cdot \Delta^k + K_j^k$ for all $j \in \mathcal{P}_i$, and aborts if any fail. Let $s = \sum_{j \in \mathcal{P}_i} s^j$. Using s as a seed for PRG, generate pseudorandom $\alpha_1, \ldots, \alpha_N \in \mathbb{F}_p$.
4. Each P_k locally computes $[\![u]\!] = [\![u']\!] + \sum_{i=1}^N \alpha_i \cdot [\![rz_i]\!]$ and $[\![w]\!] = [\![w']\!] + \sum_{i=1}^N \alpha_i \cdot [\![z_i]\!]$.

Final Check:

Committee $i+2$:

5. Parties in $\mathcal{P}_{i+2}$ start by running $\Pi_{\mathsf{Key\text{-}Switch}}$ with $\mathcal{P}_{i+1}$ to receive $[\![u]\!], [\![w]\!]$ under $\Delta_{\mathcal{P}_{i+2}}$.
6. Then they run the **Check MACs** phase of $\Pi_{\mathsf{Fluid\text{-}MAC}}$. If $\Pi_{\mathsf{Fluid\text{-}MAC}}$ fails, Reject, else continue.
7. They execute Π_{Open} on $[\![r]\!]$ to receive r, and check its MAC with $\Pi_{\mathsf{Fluid\text{-}MAC}}$.
8. Parties compute $\Pi_{\mathsf{Open}}([\![u]\!] - r[\![w]\!])$, then check the MAC. If the opened value is 0, parties Accept and go to reconstruction, else Reject.

Fig. 25: Verification phase for a fluid computation