Le Mans: Dynamic and Fluid MPC for Dishonest Majority

Rahul Rachuri[0000−0003−4541−3928], Peter Scholl[0000−0002−7937−8422]

Department of Computer Science, Aarhus University, Denmark
{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 $P_1, \ldots, P_n$ 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. In recent, concurrent work [BGIN22], another MPC protocol with sublinear preprocessing was given. Their preprocessing protocol also relies on PCGs, but scales with the square root of the circuit size rather than logarithmically. However, their online phase communication matches is slightly better than ours, and the communication of their preprocessing phase scales better with the number of parties.

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 per party. 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 $O(n \log|C|)$, instead of $O(|C|)$ storage and $O(n|C|)$ communication for standard SPDZ preprocessing. Note that after locally expanding the PCG seeds, the preprocessing material for our dynamic and fluid protocols has size $O(n|C|)$ per party, which is n times larger than SPDZ. However, once the online committee is known in Dynamic SPDZ, this can be compressed down to $O(|C|)$.

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

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

focus on the most challenging goal of *maximal fluidity*, where in each round, a different committee can sign up to receive one round of messages from the previous committee, before sending one round of messages 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. Since the MAC key of a committee is determined by the sum of the MAC keys of the parties in it, different committees will have different MAC keys. The issue with this 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. Hence, re-sharing state from one committee to another will lead to values that are authenticated under a different MAC key.

As a first attempt to deal with this problem, one could have the current committee, $\mathcal{P}_{\mathsf{curr}}$, securely *reshare* the 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.

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 the full version [RS21]. 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 do 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

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

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 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 the full version [RS21].

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 $M_j^i = K_i^j + \Delta^j \cdot x^i$, where P_i holds the MAC M_j^i and P_j holds the local key K_i^j 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:

$$\langle x \rangle^{\mathcal{P}_A, \mathcal{P}_B} = \left(\{x^i, \left(M_j^i\right)_{j \in \mathcal{P}_B}\}_{i \in \mathcal{P}_A}, \{\Delta^j, (K_i^j)_{i \in \mathcal{P}_A}\}_{j \in \mathcal{P}_B} \right)$$

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:

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

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. 2). 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. One drawback of this preprocessing, compared to SPDZ, is that the size of each partial triple is $O(n)$ field elements per party, due to the pairwise MACs and products. However, once the online phase committee in which the triples will be used is known, they can be compressed to standard, constant-sized SPDZ triples.

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. 3. 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. 4, 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

[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 $M_j^i = K_i^j + \Delta^j \cdot r^i \in \mathbb{F}_{p^r}$.
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 $c = a \cdot b + \sum_{i \in \mathcal{P}_H \cap \mathcal{P}_{\mathsf{curr}}} (a^i \cdot \delta_b^i + b^i \cdot \delta_a^i)$.
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. 2: Functionality for the preprocessing

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

Protocol. Given these building blocks, we use the preprocessing protocol Π_{Prep} (Fig. 5) 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 $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$

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. 3: Functionality for programmable OLE

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^j and compute $\boldsymbol{w}_j^i = \boldsymbol{u}^i \cdot \Delta^j + \boldsymbol{v}_i^j$.
4. If P_j is corrupt, receive a set I from $\mathcal{A}$. If $\mathsf{seed} \in I$, send $\mathsf{success}$ to P_j and continue. Else, send abort to both parties, output seed to P_j 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. 4: 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

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. 5: Protocol for preprocessing

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 the full version [RS21], 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.*

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 $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ [RS21], 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. 6.

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 idea 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 the full version [RS21], along with the proof for the following theorem:

Protocol Π_{nVOLE}

Parameters: Extension field $\mathbb{F}_{p^r}$, parties $P_1, \ldots, P_n$.
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 $\langle r_1 \rangle, \ldots, \langle r_m \rangle$,

1. Each party P_i samples a seed s^i.
2. Each ordered pair of parties (P_i, P_j) call $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$, with P_i sending (Extend, s^i) and P_j sending Extend. P_i receives $\{r_k^i, M_j^{i,k}\}$ and P_j receives $K_i^{j,k}$ for $k \in [1, m+1]$.
3. The outputs of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ define sharings $\langle r_1 \rangle, \ldots, \langle r_m \rangle, \langle t \rangle \in \mathbb{F}_{p^r}$, where each $r_j = \sum_{i=1}^n r_j^i$ and $t = \sum_{i=1}^n r_{m+1}^i$.
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 $\chi_1, \ldots, \chi_m \in \mathbb{F}_{p^r}$.
   (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. 6: Protocol for Consistent VOLE

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 the full version [RS21].

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}:

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

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. 7.

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 continue.

Fig. 7: Protocol to check MACs in Dynamic SPDZ

Online Protocol. $\Pi_{\mathsf{SPDZ\text{-}Online}}$ (Fig. 8) 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 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 the full version [RS21] 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. 8.

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 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. 9, 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 $\Pi_{\mathsf{SPDZ\text{-}Online}}$

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

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

Addition: To perform addition, $[\![z]\!] = [\![x]\!] + [\![y]\!]$, each $P_i \in \mathcal{P}_{\mathsf{curr}}$ 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 $P_i \in \mathcal{P}_{\mathsf{curr}}$ 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 $P_i \in \mathcal{P}_{\mathsf{curr}}$:

1. Calls $\mathcal{F}_{\mathsf{Prep}}$ twice with inputs $(\mathsf{Trip}, \mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{curr}}, \mathsf{count})$, 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 $(\mathsf{Rand}, \mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{curr}}, \mathsf{rcount})$ twice to receive $\langle l \rangle, \langle l' \rangle$. Increment rcount after each call.
3. Applies Π_{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 Π_{Open} on $[e] = [x - a], [d] = [y - b], [e'] = [rx - a']$ and $[d'] = [y - b'])$.
5. Runs Π_{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 $\Pi_{\mathsf{SPDZ\text{-}Verify}}$ to check the multiplications. Then, to output $[\![z]\!]$, run Π_{Open} on $[z]$, then use $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to check its MAC.

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

Fig. 8: Protocol for the online phase of 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 the full version [RS21].

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. 9: Protocol for the verification phase in Dynamic SPDZ

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 the full version [RS21], 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. 10. 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. 10: 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. 11. 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 the full version [RS21].

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.*

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. 11: Protocol to switch MAC keys

Fluid MAC Check: The MAC check protocol from SPDZ (Fig. 7) is designed to check a large batch of MACs at the end of the computation. The protocol involves computing an additively shared $[\sigma]$, which is derived from a random linear combination of all the opened values and the corresponding MACs. We call σ the *MAC check state*. If there was no cheating, σ, when opened, should be

zero. In the fluid setting, however, deferring the MAC check 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 run the full MAC Check protocol on values as soon as they are opened over the course of the computation. Instantiating this in a maximally fluid way would run over 4 epochs. Instead, we propose an incremental version of the check that updates the MAC check state in every epoch, using a fresh random challenge to serve as the next linear combination coefficient. This essentially compresses the number of things to be checked down to a constant size. Another advantage of the incremental check is that it only runs over 2 epochs.

$\Pi_{\mathsf{Fluid\text{-}MAC}}$, detailed in Fig. 12, has two subprotocols. During the online computation, parties run **Compute State** to incrementally update the MAC check state, the shared value $[\sigma]$ (which is initially zero). At the end of the computation, the final committee runs **Check State** to check that the $[\sigma]$ is still zero. 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 $[\![u]\!] - r \cdot [\![w]\!] = 0$. The underlying technique is similar to the one used in [CGG+21], and the protocol appears in the full version [RS21].

Fluid Online: We now describe how the online phase works. $\Pi_{\mathsf{Fluid\text{-}Online}}$ begins the same way as $\Pi_{\mathsf{SPDZ\text{-}Online}}$ with a set of parties $\mathcal{P}_{\mathsf{curr}} \subseteq \mathcal{P}_{\mathsf{main}}$, 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 $\Pi_{\mathsf{Fluid\text{-}Mult}}$. The first is computing output shares of

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

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

Compute State: Compute the MAC check state $[\sigma]$:

Committee i:

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

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 State: (**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 $\mathcal{F}_{\mathsf{Commit}}$ to commit to σ^j.
6. Open all commitments, and if they are consistent, Accept if $\sum_{j \in \mathcal{P}_{i+2}} \sigma^j = 0$. Else, Reject.

Fig. 12: MAC check protocol for a fluid committee

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 $\mathcal{P}_{\mathsf{curr}-1}, \mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{curr}+1}$.

The full online phase is given in Fig. 13. Below, we focus on describing the multiplication protocol, shown in the full version [RS21].

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

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

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

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

Addition: To perform addition, $[\![z]\!] = [\![x]\!] + [\![y]\!]$, each $P_i \in \mathcal{P}_{\mathsf{curr}}$ 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 $P_j \in \mathcal{P}_{\mathsf{curr}}$) 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 $P_i \in \mathcal{P}_{\mathsf{curr}}$ 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 $\mathcal{P}_{\mathsf{curr}}$, run $\Pi_{\mathsf{Fluid\text{-}Mult}}$ among $(\mathcal{P}_{\mathsf{curr\text{-}1}}, \mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{curr+1}})$.

Verify and Reconstruct:

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

Fig. 13: Protocol for a maximally fluid online phase

Security of the Online Protocol. We now briefly discuss security of the online protocol, $\Pi_{\mathsf{Fluid\text{-}Online}}$. As argued in the full version [RS21], 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.

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

Protocol	Online comm.	Preproc. comm.	Storage
SPDZ [KPR18,KOS16]	$2\|C\|$	$O(n\|C\|)$	$O(\|C\|)$
[BGIN22]	$2\|C\|$	$O(n\sqrt{\|C\|})$	$O(\sqrt{\|C\|})$
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	$O(n_c\|C\|)$	$O(n\log(\|C\|)$	$O(n\log(\|C\|)$

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

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.

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. Compared with the recent work of [BGIN22], our preprocessing scales asymptotically better with the circuit size, although its storage costs scale worse with the number of parties, and our online phase is slightly less 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 the full version [RS21], 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 **Compute State** 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 **Compute State** 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 **Compute State** 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

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.

BGIN22. Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Secure multiparty computation with sublinear preprocessing. In *EUROCRYPT 2022*. Springer, Heidelberg, May 2022.

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.

RS21. Rahul Rachuri and Peter Scholl. Le mans: Dynamic and fluid MPC for dishonest majority. Cryptology ePrint Archive, Report 2021/1579, 2021. `https://eprint.iacr.org/2021/1579`.

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.