Fluid MPC: Secure Multiparty Computation with Dynamic Participants

Arka Rai Choudhuri[1][0000−0003−0452−3426], Aarushi Goel[1], Matthew Green[1], Abhishek Jain[1], and Gabriel Kaptchuk[2]

[1] Johns Hopkins University, Baltimore, USA,
{achoud,aarushig,mgreen,abhishek}@cs.jhu.edu
[2] Boston University, Boston, USA, kaptchuk@bu.edu

Abstract. Existing approaches to secure multiparty computation (MPC) require all participants to commit to the entire duration of the protocol. As interest in MPC continues to grow, it is inevitable that there will be a desire to use it to evaluate increasingly complex functionalities, resulting in computations spanning several hours or days.

Such scenarios call for a *dynamic* participation model for MPC where participants have the flexibility to go offline as needed and (re)join when they have available computational resources. Such a model would also democratize access to privacy-preserving computation by facilitating an "MPC-as-a-service" paradigm — the deployment of MPC in volunteer-operated networks (such as blockchains, where dynamism is inherent) that perform computation on behalf of clients.

In this work, we initiate the study of *fluid MPC*, where parties can dynamically join and leave the computation. The minimum commitment required from each participant is referred to as *fluidity*, measured in the number of rounds of communication that it must stay online. Our contributions are threefold:

- We provide a formal treatment of fluid MPC, exploring various possible modeling choices.
- We construct information-theoretic fluid MPC protocols in the honest-majority setting. Our protocols achieve *maximal fluidity*, meaning that a party can exit the computation after receiving and sending messages in one round.
- We implement our protocol and test it in multiple network settings.

1 Introduction

Secure multiparty computation (MPC) [48, 32, 6, 10] allows a group of parties to jointly compute a function while preserving the confidentiality of their inputs. The increasing practicality of MPC protocols has recently spurred demand for its use in a wide variety of contexts such as studying the wage gap in Boston [37] and student success [8].

Given the increasing popularity of MPC, it is inevitable that more ambitious applications will be explored in the near future — like complex simulations on

secret initial conditions or training machine learning algorithms on *massive, distributed datasets*. Because the circuit representations of these functionalities can be extremely deep, evaluating them could take several hours or even days, even with highly efficient MPC protocols. While MPC has been studied in a variety of settings over the years, nearly all previous work considers *static* participants who must commit to participating for the entire duration of the computation. However, this requirement may not be reasonable for large, long duration computations such as above because the participants may be limited in their computational resources or in the amount of time that they can devote to the computation at a stretch. Indeed, during such a long period, it is more realistic to expect that some participants may go offline either to perform other duties (or undergo maintenance), or due to connectivity problems.

To accommodate increasingly complex applications and participation from parties with fewer computational resources, MPC protocols must be designed to support *flexibility*. In this work, we formalize the study of MPC protocols that can support *dynamic* participation – where parties can join and leave the computation without interrupting the protocol. Not only would this remove the need for parties to commit to entire long running computations, but it would also allow fresh parties to join midway through, shepherding the computation to its end. It would also reduce reliance on parties with very large computational resources, by enabling parties with low resources to contribute in long computations. This would effectively yield a *weighted*, privacy preserving, distributed computing system.

Highly dynamic computational settings have already started to appear in practice, *e.g.* Bitcoin [42], Ethereum [9], and TOR [21]. These networks are powered by volunteer nodes that are free to come and go as they please, a model that has proven to be wildly successful. Designing networks to accommodate high churn rates means that anyone can participate in the protocol, no matter their computational power or availability. Building MPC protocols that are amenable to this setting would be an important step towards replicating the success of these networks. This would allow the creation of volunteer networks capable of *private computation*, creating an "MPC-as-a-service" [3] system and democratizing access to privacy preserving computation.

Fluid MPC. To bring MPC to highly dynamic settings, we formalize the study of *fluid MPC*. Consider a group of clients that wish to compute a function on confidential inputs, but do not have the resources to conduct the full computation themselves. These clients share their inputs in a privacy preserving manner with some initial *committee* of (volunteer) servers. Once the computation begins, both the clients and the initial servers may exit the protocol execution. Additionally, other servers, even those not present during the input stage, can simply "sign-up" to join part-way through the protocol execution. The resulting protocol should still provide the security properties we expect from MPC.

We consider a model in which the computation is divided into an *input* stage, an *execution* stage, and an *output* stage. We illustrate this in Figure 1. During the input stage, a set of clients prepare their inputs for computation and hand them

over to the first committee of servers. The execution stage is further divided into a sequence of *epochs*. During each epoch, a committee of servers are responsible for doing some part of the computation, and then the intermediary state of the computation is securely transferred to a new committee. Once the full circuit has been evaluated, there is an output stage where the final results are recovered by the clients.

In order to see how well suited a particular protocol is to this dynamic setting, we introduce the notion of *fluidity* of a protocol. Fluidity captures the minimum commitment required from each server participating in the execution stage, measured in communication rounds. More specifically, fluidity is the number of communication rounds within an epoch.

A protocol with worse fluidity might require that servers remain active to send, receive, or act as passive observers on many rounds of communication. In this sense, MPC protocols designed for static participants have the worst possible fluidity — all participants must remain active throughout the lifetime of the entire protocol. In this work, we focus on protocols with only a single round of communication per epoch, which we say achieve *maximal fluidity*. Note that such protocols must have no intra-committee communication, as the communication round must be used to transfer state.

Recall that the idea of flexibility is central to the goal of Fluid MPC. Achieving maximal fluidity is ideal for fluid MPC protocols, as they give the most flexibility to the servers participating in the protocol. It allows owners of computational resources to contribute spare cycles to MPC during downtime, and a quick exit (without disrupting computation) when they are needed for another, possibly a more important task. Maximal fluidity is important to achieving this vision. Moreover, since one of our motivations behind introducing this model is evaluation of deep circuits, an important goal of this work is also to design protocol that not only achieve maximal fluidity, but also where the computation done by the servers in each epoch is independent of the size of the function/circuit.

There are several other modeling choices that can significantly impact feasibility and efficiency of a fluid MPC protocol — many of which are non-trivial and unique to this setting. For instance: when and how are the identities of the servers in the committee of a particular epoch fixed? What requirements are there on the churn rate of the system? How does the adversary's corruption model interact with the dynamism of the protocol participants? We have already seen from the extensive literature on consensus networks that different networks make different, reasonable assumptions and arrive at very different protocols.

We discuss these modeling choices and provide a formal treatment of fluid MPC in Section 3. For the constructions we give in this work, we assume that the identities of the servers in a committee are made known during the previous epoch.

Applications. We imagine that fluid MPC will be most useful for applications that involve long-running computations with deep circuits. In such a setting, being able to temporarily enlist dynamic computing resources could facilitate privacy-preserving computations that are difficult or impossible with limited

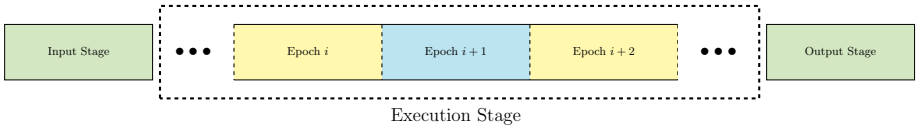

Fig. 1. Computation model of fluid MPC. A set of clients initiate the computation with the input stage. During the execution stage, servers come and go, doing small amounts of work during the compute phases and transferring state in the hand-off phase. Finally, once the entire circuit has been evaluated, the output parties recover the outputs during the output stage.

static resources. This model would be especially valuable in scientific computing, where deep circuits are common and resources can be scarce. Consider, for example, an optimization problem with many constraints over distributed medical datasets. Using a fluid MPC protocol makes it more feasible to perform such a computation with limited resources: the privacy provided by MPC can help clear important regulatory or legal impediments that would otherwise prevent stakeholders from contributing data to the analysis, and a dynamic participation model can allow stakeholders to harness computing resources as they become available.

Prior Work: Player Replaceability. In recent years, the notion of *player replaceability* has been studied in the context of Byzantine Agreement (BA) [40, 11]. These works design BA protocols where after every round, the "current" set of players can be replaced with "new" ones without disrupting the protocol. This idea has been used in the design of blockchains such as Algorand [30], where player replaceability helps mitigate targeted attacks on chosen participants after their identity is revealed.

Our work can be viewed as extending this line of research to the setting of MPC. We note that unlike BA where the parties have no private states – and hence, do not require state transfer for achieving player replaceability – the MPC setting necessitates a state transfer step to accommodate player churn. Maximal fluidity captures the best possible scenario where this process is performed in a *single* round.

1.1 Our Contributions

In this work, we initiate the study of fluid MPC. We state our contributions below.

Model. We provide a formal treatment of fluid MPC, exploring possible modeling choices in the setting of dynamic participants.

Protocols With Maximal Fluidity. We construct information-theoretic fluid MPC protocols that achieve maximal fluidity. We consider adversaries that (adaptively) corrupt any minority of the servers in each committee.

We begin by observing that the protocol by Genarro, Rabin and Rabin [28], which is an optimized version of the classical semi-honest BGW protocol [6] can be adapted to the fluid MPC setting in a surprisingly simple manner. We call this protocol Fluid-BGW. This protocol also achieves division of work, in the sense that the amount of work that each committee is required to do is independent of the depth of the circuit.

To achieve security against malicious adversaries, we extend the "additive attack" paradigm of [26] to the fluid MPC setting, showing that any malicious adversarial strategy on semi-honest fluid MPC protocols (with a specific structure and satisfying a weak notion of privacy against malicious adversaries[3]) is limited to injecting additive values on the intermediate wires of the circuit. We use this observation to build an *efficient* compiler (in a similar vein as recent works of [12, 43]) that transforms such semi-honest fluid MPC protocols into ones that achieve security with abort against malicious adversaries. Our compiler enjoys two salient properties:

- It *preserves fluidity* of the underlying semi-honest protocol.
- It incurs a *multiplicative overhead of only* 2 (for circuits over large fields) in the communication complexity of the underlying protocol.

Applying our compiler to Fluid-BGW yields a maximally fluid MPC protocol that achieves security with abort against malicious adversaries.

We note that, while we consider a slightly restrictive setting where the adversary is limited to corrupting a minority of servers in each committee, there is evidence that our assumption might hold in practice if we, e.g., leverage certain blockchains. The work of [7] (see also [29]) explores a similar problem of dynamism in the context of secret-sharing with a similar honest-majority assumption as in our work. They show that in certain blockchain networks, it is possible to leverage the honest-majority style assumption (which is crucial to the security of such blockchains) to elect committees of servers with an honest majority of parties. The same mechanism can also be used in our work (we discuss this in more detail in Section 3.2). Moreover, the honest majority assumption is necessary for achieving information-theoretic security (or for using assumptions weaker than oblivious transfer), for the same reasons as in standard (static) MPC.

Implementation. We implement Fluid-BGW and our malicious compiler in `C++`, building off the code-base of [16, 12]. We run our implementation across multiple network settings and give concrete measurements. Due to space constraints, we discuss our implementation and experimental results in the full version of the paper [13].

[3] It was observed in [26] that almost all known secret sharing based semi-honest protocols in the static model naturally satisfy this weak privacy property. We observe that the fluid version of BGW continues to satisfy this property. Further, we conjecture that most secret-sharing based approaches in the fluid MPC setting would also yield semi-honest protocols that achieve this property.

1.2 Related Work

Proactive Multiparty Computation. The proactive security model, first introduced in [44], aims to model the persistent corruption of parties in a distributed computation, and the continuous race between parties for corruption and recovery. To capture this, the model defines a "mobile" adversary that is not restricted in the total number of corruptions, but can corrupt a subset of parties in different time periods, and the parties periodically reboot to a clean state to mitigate the total number of corruptions. Prior works have investigated the feasibility of proactive security both in the context of secret sharing [35, 39] and general multiparty computation [44, 4, 22].

While both fluid MPC and Proactive MPC (PMPC) consider dynamic models, the motivation behind the two models are completely different. This in turn leads to different modeling choices. Indeed, the dynamic model in PMPC considers slow-moving adversaries, modeling a spreading computer virus where the set of participants are fixed through the duration of the protocol. This is in contrast to the Fluid MPC model where the dynamism is derived from participants leaving and joining the protocol execution as desired. As such, the primary objective of our work is to construct protocols that have maximal fluidity while simultaneously minimizing the computational complexity in each epoch. Neither of these goals are a consideration for protocols in the PMPC setting. Furthermore, unlike PMPC, fluid MPC captures the notion of volunteer servers that sign-up for computation proportional to the computational resources available to them.

The difference in motivation highlighted above also presents different constraints in protocol design. For instance, unlike PMPC, the size of private states of parties is a key consideration in the design of fluid MPC; we discuss this further in Section 2. We do note, however, that some ideas from the PMPC setting, such as state re-randomization are relevant in our setting as well.

Transferable MPC. In [14], Clark and Hopkinson consider a notion of Transferable MPC (T-MPC) where parties compute partial outputs of their inputs and transfer these shares to other parties to continue computation while maintaining privacy. Unlike our setting, the sequence of transfers, and the computation at each step is determined completely by the circuit structure. In the constructed protocol, each partial computation involves multiple rounds of interaction and therefore does not achieve fluidity; additionally parties cannot leave during computation sacrificing on dynamism.

Concurrent and Independent Work. Two independent and concurrent works [33, 7] also model dynamic computing environments by considering protocols that progress in discrete stages denoted as epochs, which are further divided into computation and hand-off phases. These works study and design *secret sharing* protocols in the dynamic environment. In contrast, our work focuses on the broader goal of *multi-party computation* protocols for all functionalities.

Furthermore, we focus on building protocols that achieve maximal fluidity. While this goal is not considered in [33], [7] can be seen as achieving maximal

fluidity for secret sharing. In choosing committees for each epoch, [33] consider an approach similar to ours where the committee is announced at the start of the hand-off phase of each epoch. [7] leverage properties in the blockchain to implement a committee selection procedure that ensures an honest majority in each committee.

Lastly, both of these works consider a security model incomparable to ours. Specifically, they consider security with guaranteed output delivery for secret sharing against computationally bounded adversaries, whereas we consider MPC with security with abort against computationally unbounded adversaries.

Malicious Security Compilers for MPC. There has been a recent line of exciting work [12, 43, 38, 1, 2, 41, 36, 23] in designing concretely efficient compiler that upgrade security from semi-honest to malicious in the honest majority setting. Some of these compilers rely on the additive attack paradigm introduced in [26]. We take a similar approach, but adapt and extend the additive attack paradigm to the fluid MPC setting.

2 Technical Overview

We start by briefly discussing some specifics of the model in which we will present our construction. A detailed formal description of our model is provided in Section 3.

As discussed earlier, we consider a client-server model where computation proceeds in three phases – input stage, execution stage and output stage (see Figure 1). The execution stage proceeds in epochs, where different committees of servers perform the computation. Each epoch ℓ is further divided into two phases: (1) *computation phase*, where the servers in the committee (denoted as $\mathcal{S}^\ell$) perform computation, and (2) *hand-off* phase, where the servers in $\mathcal{S}^\ell$ transfer their states to the incoming committee $\mathcal{S}^{\ell+1}$. We require that at the start of the hand-off phase of epoch ℓ, everyone is aware of committee $\mathcal{S}^{\ell+1}$. We consider security in the presence of an adversary who can corrupt a minority of servers in every committee.

For the remainder of the technical overview, we describe our ideas for the simplified case where all the committees are disjoint and the size of the committees remain the same across all epochs, denoted as n. Neither of these restrictions are necessary for our protocols, and we refer the reader to the technical sections for further details.

Main Challenges Designing protocols that are well suited to the fluid MPC setting requires overcoming challenges that are not standard in the static setting. While some of these challenges have been considered previously in isolation in other contexts, the main difficulty is in addressing them at the same time.

1. **Fluidity.** The primary focus of our work is the fluidity of protocols, a measure of how long the servers must remain online in order to contribute to

the computation. The fluidity of a protocol is the number of rounds of interaction in a single epoch, and we say that a protocol achieves maximal fluidity if there is only a *single* round in each epoch. Designing protocols with maximal fluidity means that the computation phase of an epoch must be "silent" (i.e., non-interactive), and the hand-off phase must complete in a single round.

2. **Small State Complexity.** In many classical MPC protocols, the private state held by each party is quite large, often proportional to the size of the circuit (see, e.g. [19]). We refer to this as the *state complexity* of the protocol. While state complexity is generally not considered an important measure of a protocol's efficiency, in the fluid MPC setting it takes on new importance. Because the state held by the servers must be transferred between epochs, the state complexity of a protocol contributes directly to its communication complexity. Protocols with large state complexity, say proportional to the size of the circuit, would require each committee to perform a large amount of work, undermining any advantage of fluidity. Therefore, special attention must be paid to minimize the state complexity of the protocol in the fluid MPC setting.
3. **Secure State Transfer.** As mentioned earlier, we consider adversaries that can corrupt a minority of servers in every committee. As such, state cannot be naively handed off between committees in a one-to-one manner. To illustrate why this is true, consider secret sharing based protocols where the players collectively hold a t-out-of-n secret sharing of the wire values and iteratively compute on these shares. If states were transferred by having each server in committee $\mathcal{S}^i$ choose a unique server in $\mathcal{S}^{i+1}$ (as noted, we assume for convenience that $|\mathcal{S}^i| = |\mathcal{S}^{i+1}|$) and simply sending that new server their state, the adversary would see $2t$ shares of the transferred state, t shares from $\mathcal{S}^i$ and another t shares from $\mathcal{S}^{i+1}$, thus breaking the privacy of the protocol. Fluid MPC protocols must therefore incorporate mechanisms to securely transfer the protocol state between committees.

In this work, we focus our attention on protocols that achieve maximal fluidity. Designing such protocols requires careful balancing between these three factors. In particular, the need for small state complexity makes it difficult to use many of the efficient MPC techniques known in the literature, as we will discuss in more detail below.

Adapting Optimized Semi-honest BGW [28] to Fluid MPC Despite the challenges involved in the design of fluid MPC protocols, we observe that the protocol by Gennaro et al. [28], which is an optimized version of the semi-honest BGW [6] protocol can be adapted to the fluid MPC setting in a surprisingly simple manner.

Recall that in [28], the parties collectively compute over an arithmetic circuit representation of the functionality that they wish to compute, using Shamir's secret sharing scheme. For each intermediate wire in the circuit, the following invariant is maintained: the shares held by the parties correspond to a t-of-n

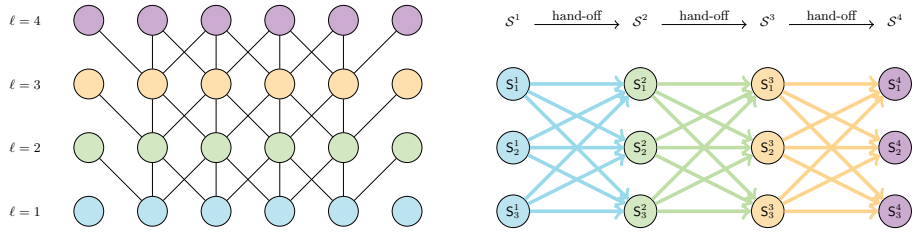

Fig. 2. Left: Part of the circuit partitioned into different layers, indicated by the different colors. **Right:** A visual representation of the flow of information during the modified version of BGW presented in Section 2, running with committees of size 3, which achieves maximal fluidity. $\mathcal{S}^\ell = \{\mathsf{S}_1^\ell, \mathsf{S}_2^\ell, \mathsf{S}_3^\ell, \}$ denotes the set of active servers in each committee corresponding to level ℓ, indicated by the same color.

secret sharing of the value induced by the inputs on that wire. Evaluating addition gates requires the parties to simply add their shares of the incoming wires, leveraging the linearity of the secret sharing scheme. For evaluating multiplication gates, the parties first locally multiply their shares of the incoming wires, resulting in a distributed degree $2t$ polynomial encoding of the value induced on the output wire of the gate. Then, each party computes a fresh t-out-of-n sharing of this degree $2t$ share and sends one of these *share-of-share* to every other party. Finally, the parties locally interpolate these received shares and as a result, all the parties hold a t-out-of-n sharing of the product. Thus, every multiplication gate requires only one round of communication.

We observe that adapting this version of semi-honest BGW to fluid MPC setting, which we will refer to as Fluid-BGW, is straightforward. The key observation is that the degree reduction procedure of this protocol *simultaneously* re-randomizes the state, so that only a *single round of communication* is required to accomplish both goals. In each epoch, the servers will evaluate all the gates in a *single layer* of the circuit, which may contain both addition and multiplication gates (see Figure 2). More specifically, for each epoch ℓ:

Computation Phase: The servers in $\mathcal{S}^\ell$ interpolate the shares-of-shares (received from the previous committee) to obtain a degree t sharing for full intermediary state (for each gate in that layer). Then, they locally evaluate each gate in layer ℓ, possibly increasing the degree of the shares that they hold. Finally, they compute a t-out-of-n secret sharing of the *entire* state they hold, including multiplied shares, added shares and any "old" values that may be needed later in the circuit.

Hand-off Phase: The servers in $\mathcal{S}^\ell$ then send one share of each sharing to each active server in $\mathcal{S}^{\ell+1}$.

The computation phase is non-interactive and the hand-off phase consists of only a single round of communication, and therefore the above protocol achieves maximal fluidity.

Recall that we consider adversaries who can corrupt a minority of t servers in *each committee*, a significant departure from the classical setting in which a *total* of t parties can be corrupted. At first glance, it may seem as though the adversary can gain significant advantage by corrupting (say) the first t parties in committee $\mathcal{S}^\ell$ and the last t parties in committee $\mathcal{S}^{\ell+1}$. However, since computing shares-of-shares essentially re-randomizes the state, at the end of the hand-off phase of epoch ℓ, the adversary is aware of the (1) nt shares-of-shares that were sent to the last t corrupt servers during the hand-off phase of epoch ℓ and (2) $(n-t)\times t$ shares-of-shares that the first t corrupt servers in $\mathcal{S}^\ell$ sent to the $(n-t)$ honest servers in $\mathcal{S}^{\ell+1}$. This is in fact no different than regular BGW. Since the partial information that the adversary has about the states of the $(n-t)$ honest servers in $\mathcal{S}^{\ell+1}$ only corresponds to t shares of their individual states, privacy is ensured.

Compiler for Malicious Security Having established the feasibility of semi-honest MPC with maximal fluidity, we now describe our ideas for transforming semi-honest fluid MPC protocols into ones that achieve security against malicious adversaries. Our goal is to achieve two salient properties: (1) *fluidity preservation*, i.e., preserve the fluidity of the underlying protocol, (2) *multiplicative overhead of 2* in the complexity of the underlying protocol.

Shortcomings of Natural Solutions. Consider a natural way of achieving malicious security: after each gate evaluation, the servers perform a check that the gate was properly evaluated, as is done in the malicious-secure version of BGW [6]. However, known techniques for implementing gate-by-gate checks rely on primitives such as verifiable secret sharing (among others) that require *additional interaction* between the parties. Such a strategy is therefore incompatible with our goal of achieving maximal fluidity, which requires a single round hand-off phase. Even computational techniques like non-interactive zero knowledge proofs do not appear to be directly applicable as they may require a committee to have access to all prior rounds of communication in order to verify that the received messages were correctly computed.

Starting Idea: Consolidated Checks. Since performing gate-by-gate checks is not well-suited to fluid MPC, we consider a *consolidated check* approach to malicious security, where the correctness of the computation (of the entire circuit) is checked *once*. This approach has previously been studied in the design of efficient MPC protocols [20, 26, 25, 43, 12, 23, 34]. In this line of work, [26] made an important observation, that *linear-based MPC protocols* (a natural class of semi-honest honest-majority MPC protocols) are secure *up to additive attacks*, meaning any strategy followed by a malicious adversary is equivalent to injecting an additive error on each wire in the circuit. They use this observation to first compile the circuit into another circuit that automatically detects errors, e.g., AMD circuits and then run a semi-honest protocol on this modified circuit to get malicious security. Many other works [25, 27] follow suit.

Assuming that the same observation caries over to the fluid MPC setting, for feasibility, one could consider running a semi-honest, maximally fluid MPC

protocol on such transformed circuits. However, transforming a circuit into an AMD circuit incurs very high overhead in practice. In order to design a more efficient compiler that only incurs an overhead of 2, we turn towards the approach taken by some of the more recent malicious security compilers [43, 12, 23, 34]. In some sense, the ideas used in these works can be viewed as a more efficient implementation of the same idea as above (without using AMD circuits).

Roughly speaking, in the approach taken by these recent compilers, for every shared wire value z in the circuit, the parties also compute a secret sharing of a MAC on z. At the end of the protocol, the parties verify validity of all the MACs in one shot. Given the observation from [26], it is easy to see that the parties can generate a single, secret MAC key r at the beginning of the protocol and compute $MAC(r, z) = rz$ for each wire z in the circuit. It holds that if the adversary injects an additive error δ on the wire value z, to surpass the check, they must inject a corresponding additive error of $\hat{\delta} = r\delta$ on the MAC. Because r is uniformly distributed and unknown to all servers, this can only happen with probability negligible in the field size. While previously, this approach has primarily been used for improving the efficiency of MPC protocols, we use it in this work for also maximizing fluidity.

Verifying the MACs requires revealing the key r, but this is only done at the *end* of the protocol, as revealing r too early would allow the adversary to forge MACs. Furthermore, to facilitate efficient MAC verification, the parties finish the protocol with the following "condensed" check: they generate random coefficients α_k and use them to compute linear combinations of the wire values and MACs as follows:

$$u = \sum_{k \in [|C|]} \alpha_k \cdot z_k \text{ and } v = \sum_{k \in [|C|]} \alpha_k \cdot rz_k.$$

Finally, they reconstruct the key r and interactively verify if $v = ru$, before revealing the output shares.

To build on this approach, we first need to show that *linear-based fluid MPC protocols* are also secure up to additive attacks against malicious adversaries. We prove this to be true in the full version of the paper and show that the semi-honest Fluid-BGW satisfies the structural requirement of linear-based fluid MPC protocols. At first glance, it would appear that we can then directly implement the above mechanism to the fluid MPC setting as follows: in the output stage, parties interactively generate shares of α_k, locally compute this linear combination, reconstruct r, and perform the equality check.

To see where this approach falls short, consider the state complexity of this protocol. To perform the consolidated check, parties in the output stage require shares of *all wires in the circuit*, namely z_k and rz_k for $k \in [|C|]$, which must have been passed along as part of the state between each consecutive pair of committees. This means that the state complexity of the protocol is proportional to the *size of the circuit*, which (as discussed earlier) would undermine the advantages of the fluid MPC model. More concretely, this approach would incur at least $|C|$ multiplicative overhead in the communication of the underlying protocol – far higher than our goal of achieving constant overhead.

Incrementally Computing Linear Combination. In order to implement the above consolidated check approach in the fluid MPC setting, we require a method for computing the aforementioned aggregated values that does not require access to the entire intermediate computation during the output stage. Towards this, we observe that the servers can *incrementally* compute u and v throughout the protocol. This can be done by having each committee incorporate the part of u and v corresponding to the *gates evaluated by the previous committee* into the partial sum. That is, committee $\mathcal{S}^\ell$ is responsible for (1) evaluating the gates on layer ℓ, (2) computing the MACs for gates on layer ℓ, and (3) computing the partial linear combination for all the gates *before* layer $\ell - 1$.

Let the output of the k^{th} gate on the i^{th} layer of the circuit be denoted as z_k^i. Apart from the shares of $z_k^{\ell-1}$ and $rz_k^{\ell-1}$ (for $k \in [w]$), the servers computing layer ℓ of the circuit $\mathcal{S}^\ell$ also receive shares of

$$u_{\ell-2} = \sum_{i \leq \ell-2} \sum_{k \in [w]} \alpha_k^i \cdot z_k^i \text{ and } v_{\ell-2} = \sum_{i \leq \ell-2} \sum_{k \in [w]} \alpha_k^i \cdot rz_k^i$$

from $\mathcal{S}^{\ell-1}$ during hand-off, where α_k^i is a random value associated with the gate outputting z_k^i. While $u_{\ell-2}$ and $v_{\ell-2}$ represent the consolidated check for all gates in the circuit *before* layer $\ell - 1$. $\mathcal{S}^\ell$ then computes shares of

$$u_{\ell-1} = u_{\ell-2} + \sum_{k \in [w]} \alpha_k^{\ell-1} \cdot z_k^{\ell-1} \text{ and } v_{\ell-1} = v_{\ell-2} + \sum_{k \in [w]} \alpha_k^{\ell-1} \cdot rz_k^{\ell-1}$$

in addition to shares of the outputs of gates on layer ℓ (z_k^ℓ and rz_k^ℓ) and transfer $u_{\ell-1}$ and $v_{\ell-1}$ to $\mathcal{S}^{\ell+1}$ during hand-off. Note that the final $u = u_d$ and $v = v_d$, where d is the depth of the circuit. This leaves the following main question: how do the servers agree upon the values of α_k^ℓ?

Notice that $|\{\alpha_k^\ell\}_{k \in [w], \ell \in [d]}| = |C|$, therefore generating shares of all the α_k^ℓ values at the beginning of the protocol and passing them forward will, again, yield a protocol that has an excessively large state complexity. Another natural solution might be to have the servers generate α_k^ℓ as and when they need them. However, because our goal is to maintain maximal fluidity, the servers in $\mathcal{S}^j$ for some fixed j cannot generate α_k^j, as this would require communication *within* $\mathcal{S}^j$.

Instead, consider a protocol in which the servers in $\mathcal{S}^{j-1}$ do the work of generating the shares of α_k^j. Each server in $\mathcal{S}^{j-1}$ generates a random value and shares it, sending one share to each server in $\mathcal{S}^j$. The servers in $\mathcal{S}^j$ then combine these shares using a Vandermonde matrix to get correct shares of α_k^j, as suggested by [5]. While this approach achieves maximal fluidity and requires a small state complexity, it incurs a multiplicative overhead of n in the complexity of the underlying semi-honest protocol.[4]

[4] In the static setting, this technique allows for batched randomness generation, by generating $O(n)$ sharings with $O(n^2)$ messages. In the fluid MPC setting, however, the number of servers *cannot* be known in advance and may not correspond to the width of the circuit. Therefore, such amortization techniques are not applicable.

Efficient Compiler. We now describe our ideas for achieving multiplicative overhead of only 2 (for circuits over large fields). In our compiler, we use the above intuition, having each committee, evaluate gates for its layer, compute MACs for the previous layer, and incrementally add to the sum. In the input stage, the clients generate a sharing of a secret random MAC key r, and secret random values $\beta, \alpha_1, \ldots, \alpha_w$. Over the course of the protocol, the servers will incrementally compute values

$$u = \sum_{\ell \in [d]} \sum_{k \in [w]} (\alpha_k (\beta)^\ell) \cdot z_k^\ell \text{ and } v = \sum_{\ell \in [d]} \sum_{k \in [w]} (\alpha_k (\beta)^\ell) \cdot r z_k^\ell$$

where z_k^ℓ is the output of the k^{th} gate on level ℓ, $(\beta)^\ell$ is β raised to the ℓ^{th} power, and $\alpha_k(\beta)^\ell$ is the "random" coefficient associated with it. At the end of the protocol, the parties verify whether $v = ru$.

Notice that at the beginning of the execution stage, the servers do not have shares of $(\alpha_k(\beta)^\ell)$ for $\ell > 0$, but they have the necessary information to compute a valid sharing of this coefficient in parallel with the normal computation, namely $\beta, \alpha_1, \ldots, \alpha_w$. To compute the coefficients, we require that the servers computing layer ℓ are given shares of $(\alpha_k(\beta)^{\ell-1})$ and β by the previous set of servers, in addition to the shares of the actual wire values. The servers in $\mathcal{S}^\ell$ then use these shares to compute shares of (1) the values z_k^ℓ on outgoing wires from the gates on layer ℓ, (2) the partial sums by adding the values computed in the previous layer $u_{\ell-1} = u_{\ell-2} + (\alpha_k(\beta)^{\ell-1}) \cdot z_k^{\ell-1}$ and $v_{\ell-1} = v_{\ell-2} + (\alpha_k(\beta)^{\ell-1}) \cdot r z_k^{\ell-1}$, and (3) the coefficients for the next layer $(\alpha_k(\beta)^\ell) = \beta \cdot \alpha_k(\beta)^{\ell-1}$. All of this information can be securely transferred to the next committee.

We give a simplified sketch to illustrate why this check is sufficient. Let $\epsilon_{z,k}^\ell$ (and $\epsilon_{rz,k}^\ell$ resp.) be the additive error introduced by the adversary on the computation of z_k^ℓ (rz_k^ℓ resp.).

As before, the check succeeds if

$$r \cdot \sum_{\ell \in [d]} \sum_{k \in [w]} (\alpha_k(\beta)^\ell)(z_k^\ell + \epsilon_{z,k}^\ell) = \sum_{\ell \in [d]} \sum_{k \in [w]} (\alpha_k(\beta)^\ell)(r z_k^\ell + \epsilon_{rz,k}^\ell)$$

Let the q^{th} gate on level m be the first gate where the adversary injects errors $\epsilon_{z,q}^m$ and $\epsilon_{rz,q}^m$. The above equality can be re-written as.

$$\alpha_q \left[\sum_{\ell=m}^{d} ((\beta)^\ell \epsilon_{rz,q}^\ell) - r \sum_{\ell=m}^{d} ((\beta)^\ell \epsilon_{z,q}^\ell) \right] =$$
$$r \cdot \sum_{\ell=m}^{d} \sum_{\substack{k \in [w] \\ k \neq q}} (\alpha_k(\beta)^\ell)(z_k^\ell + \epsilon_{z,k}^\ell) - \sum_{\ell=m}^{d} \sum_{\substack{k \in [w] \\ k \neq q}} (\alpha_k(\beta)^\ell)(r z_k^\ell + \epsilon_{rz,k}^\ell)$$

This holds only if either (1) $\sum_{\ell=m}^{d} ((\beta)^\ell \epsilon_{z,q}^\ell) = 0$ and $\sum_{\ell=m}^{d} ((\beta)^\ell \epsilon_{rz,q}^\ell) = 0$. The key point is that since these are polynomials in β with degree at most d, the probability that β is equal to one of its roots is $d/|\mathbb{F}|$. Or if (2) $r =$

$\sum_{\ell=m}^{d}((\beta)^{\ell}\epsilon_{rz,q}^{\ell})(\sum_{\ell=m}^{d}((\beta)^{\ell}\epsilon_{z,q}^{\ell}))^{-1}$. Since r is uniformly distributed, this happens only with probability $1/|\mathbb{F}|$.

This analysis is significantly simplified for clarity and the full analysis is included in the full version of the paper [13]. Note that the adversary can inject additive errors on r and β, since these values are also re-shared between sets of servers. Also, since the α values for the gates on level $\ell > 0$ are computed using a multiplication operation, the adversary can potentially inject additive errors on these values as well. However, we observe that the additive errors on the value of β and consequently on the α values associated with the gates on higher levels, does not hamper the correctness of output. But the errors on the value of r, do need to be taken into consideration. The analysis in the full version of the paper addresses how these errors can be handled, making it non-trivial and notationally complicated, but the core intuition remains the same.

We note that we are not the first to consider generating multiple random values by raising a single random value to consecutively larger powers. In particular, [20] performs consolidated checks by taking a linear combination of all wire values, the coefficients for which need to be generated securely, i.e. be randomly distributed and authenticated. But this generation is expensive, so they generate a single secure value and derive all other values by raising it to consecutively larger powers. A consequence of this technique is that once the single secure value is revealed, the exponentiations are done locally and therefore precludes any introduction of errors in this computation for the honest parties. Although this technique might seem similar to ours, our specific implementation is different and for a different purpose, namely, achieving maximal fluidity together with small constant multiplicative overhead.

Implementation Overview. Due to space constraints, discussion of our implementation does not fit in this version of this work, so we briefly discuss it here. We implement Fluid-BGW with our malicious security compiler in `C++`, using `libscapi` [16] and the code written for [12] as a starting point. We implement several minor optimizations for our implementation. For instance, we preprocess the circuit so the players always know the maximum number of random values that will be needed in future layers for the malicious security compiler. This allows the player to never pass on unnecessary information. We run our protocol both on a single large server, to benchmark its computational performance, and using the AWS C4.large instances spread between North Virginia, Germany and India, replicating the WAN deployment in [12]. We report both per-layer timing results and total runtime for between 3 and 20 servers per epoch.

3 Fluid MPC

In this section, we give a formal treatment of the fluid MPC setting. We start by describing the model of computation and then turn to the task of defining security. Our goals in this section are twofold: first, we illustrate that there are many possible modeling parameters to choose from in the fluid MPC setting. Second, we highlight the modeling choices that we make for the protocols we

describe in later sections. Before beginning, we reiterate that the functionalities considered in this setting can be represented by circuits where the depth of such circuits are large.

Model of Computation. We consider a *client-server* model of computation where a set of clients $\mathcal{C}$ want to compute a function over their private inputs. The clients delegate the computation of the function to a set of servers $\mathcal{S}$. Unlike the traditional client-server model [15, 17, 18] where every server is required to participate in the entire computation (and hence, remain online for its entire duration), we consider a dynamic model of computation where the servers can volunteer their computational resources for *part of the computation* and then potentially go offline. That is, the set of servers is not fixed in advance.

We adopt terminology from the execution model used in the context of permissionless blockchains [45, 46, 24]. The protocol execution is specified by an interactive Turing Machine (ITM) $\mathcal{E}$ referred to as the *environment*. The environment $\mathcal{E}$ represents everything that is external to the protocol execution. The environment generates inputs to all the parties, reads all the outputs and additionally can interact in an arbitrary manner with an adversary $\mathcal{A}$ during the execution of the protocol.

Protocols in this execution model proceed in rounds, where at the start of each round, the environment $\mathcal{E}$ can specify an input to the parties, and receive an output from the corresponding parties at the end of the round. We also allow the environment $\mathcal{E}$ to spawn new parties at any point during the protocol. The parties have access to point-to-point and broadcast channels. In addition, we assume fully synchronous message channels, where the adversary does not have control over the delivery of messages. This is the commonly considered setting for MPC protocols.

3.1 Modeling Dynamic Computation

In a fluid MPC protocol, computation proceeds in three stages:

Input Stage: In this stage, the environment $\mathcal{E}$ hands the input to the clients at the start of the protocol, who then pre-process their inputs and hand them off to the servers for computation.

Execution Stage: This is the main stage of computation where only the servers participate in the computation of the function.

Output Stage: This is the final stage where only the clients participate in order to reconstruct the output of the function. The output is then handed to the environment.

The clients only participate in the input and output stages of the protocol. Consequently, we require that the computational complexity of both the input and the output stages is *independent* of the depth of the functionality (when represented as a circuit) being computed by the protocol. Indeed, a primary goal of this work is to offload the computation work to the servers and a computation-intensive input/output phase would undermine this goal.

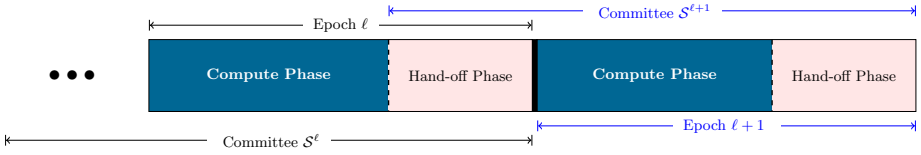

Fig. 3. Epochs ℓ and $\ell + 1$

We wish to capture dynamism for the bulk of the computation, and thus model dynamism in the *execution stage* of the protocol (rather than the input and output stages). In the following, we highlight the key modeling choices for the protocols we present in the full version of the paper by displaying them in **bold font in color**.

Epoch. We model the progression of the execution stage in discrete steps referred to as *epochs*. In each epoch ℓ, only a subset of servers $\mathcal{S}^{\ell}$ participate in the computation. We refer to this set of servers $\mathcal{S}^{\ell}$ as the **committee** for epoch ℓ. An epoch is further divided into two phases, illustrated in Figure 3:

- **Computation Phase:** Every epoch begins with a computation phase where the servers in the committee $\mathcal{S}^{\ell}$ perform computation over their local states, possibly involving multiple rounds of interaction with each other.
- **Hand-off Phase:** The epoch then transitions to a hand-off phase where the committee $\mathcal{S}^{\ell}$ transfers the protocol state to the next committee $\mathcal{S}^{\ell+1}$. As with the computation phase, this phase may involve multiple rounds of interaction. When this phase is completed, epoch $\ell + 1$ begins.

Fluidity. We define the notion of *fluidity* to measure the minimum commitment that a server needs to make for participating in the execution stage.

Definition 1 (Fluidity). *Fluidity is defined as the number of* rounds *of interaction within an epoch.*

Clearly, the fewer the number rounds in an epoch, the more "fluid" the protocol. We say that a protocol has **maximal fluidity** when the number of rounds in an epoch is 1. We emphasize that this is only possible when the computation phase of an epoch is completely *non-interactive*, i.e., the servers only perform local computation on their states without interacting with each other. This is because the hand-off phase must consist of at least one round of communication. *In this work, we aim to design protocols with* ***maximal fluidity****.*

3.2 Committees

We now explore modeling choices for committees. We address three key aspects of a committee – its formation, size and possible overlap with other committees. Along the way, we also discuss how long a server needs to remain *online*.

Functionality $f_{\mathsf{committee}}$

Hardcoded: Sampling function $\mathsf{Sample} : \mathcal{P} \mapsto \mathcal{P}$.

1. Set $\mathcal{P} := \emptyset$
2. When party P_i sends input $\mathsf{nominate}$, $\mathcal{P} := \mathcal{P} \cup \{\mathsf{P}_i\}$.
3. When the environment $\mathcal{E}$ sends input elect, compute $\mathcal{P}' \leftarrow \mathsf{Sample}(\mathcal{P})$ and broadcast $\mathcal{P}'$ as the selected committee.

Fig. 4. Functionality for Committee Formation.

Committee Formation. From our above discussion on computation progressing in epochs, we consider two choices for *committee formation*:

Static. In the most restrictive choice, the environment determines right at the start, which servers will participate in the protocol, *and* the epoch(s) they will be participating in. This in turn determines the committee for every epoch. This means that the servers must commit to their resources ahead of time. We view this choice to be too restrictive and shall not consider it for our model.

On-the-fly. In the other choice, committees are determined dynamically such that **committee for epoch $\ell + 1$ is determined and known to everyone at the start of the hand-off phase of epoch ℓ**. We consider the functionality $f_{\mathsf{committee}}$ described in Figure 4 to capture this setting.

In an epoch ℓ, if the environment $\mathcal{E}$ provides input $\mathsf{nominate}$ to a party at the start of the round, it relays this message to $f_{\mathsf{committee}}$ to indicate that it wants to be considered in the committee for epoch $\ell + 1$. The functionality computes the committee using the sampling function Sample, from the set of parties $\mathcal{P}$ that have been "nominated." The environment $\mathcal{E}$ is also allowed a separate input elect that specifies the cut-off point for the functionality to compute the committee. The cut-off point corresponds to the start of the hand-off phase of epoch ℓ where the parties in $\mathcal{S}^\ell$ are made aware of the committee $\mathcal{S}^{\ell+1}$ via a broadcast from $f_{\mathsf{committee}}$.

We consider two possible committee choices in this dynamic setting below.

Volunteer Committees. One can view the servers as "volunteers" who sign up to participate in the execution stage whenever they have computational resources available. Essentially anyone, who wants to, can join (up until the cut-off point) in aiding with the computation. This can be implemented by simply setting the sampling function Sample in $f_{\mathsf{committee}}$ to be the identity function, i.e. a party is included in the committee for epoch $\ell + 1$ if and only if it sent a $\mathsf{nominate}$ to $f_{\mathsf{committee}}$ during the computation phase of epoch ℓ.

Elected Committees. One could envision other sampling functions that implement a *selection* process using a participation criterion such

as the cryptographic sortition [30] considered in the context of proof of stake blockchains. The work of [7] considers the function Sample that is additionally parameterized by a probability p; for each party in $\mathcal{P}$, Sample independently flips a coin that outputs 1 with probability p, and only includes the party in the final committee if the corresponding coin toss results in the value 1. To ensure that all parties are considered in the selection process, one can simply require that every party sends a nominate to $f_{\text{committee}}$ in each epoch. Committee election has also been studied in different network settings; e.g., the recent work of [47] provides methods for electing committees over TOR [21].

Both of the above choices have direct consequences on the corruption model. The former choice of volunteer committees models protocols that are accessible to anyone who wants to participate. However, an adversary could misuse this accessibility to corrupt a large fraction of (maybe even all) participants of a committee. As such, we view this as an *optimistic* model since achieving security in this model can require placing severe constraints on the global corruption threshold.

The latter choice of elected committees can, by design, be viewed as a semi-closed system since not everyone who "volunteers" their resources are selected to participate in the computation. However, by using an appropriate sampling function, this selection process can potentially ensure that the number of corruptions in each committee are kept within a desired threshold.

We envision that the choice of the specific model (i.e. the sampling function Sample) is best determined by the environment the protocol is to be deployed in and the corruption threshold one is willing to tolerate. (We discuss the latter implication in Section 3.3.) Our protocol design is agnostic to this choice and only requires that the committee $\mathcal{S}^\ell$ knows committee $\mathcal{S}^{\ell+1}$ at the start of the hand-off phase.

Participant Activity. We say that a server is *active* within an epoch if it either (a) performs some protocol computation, or (b) sends/receives protocol messages. Clearly, a server S is active during epoch ℓ only if it belongs to $\mathcal{S}^\ell \cup \mathcal{S}^{\ell+1}$. When extending this notion to a committee, we say committee $\mathcal{S}^\ell$ is active from the beginning of the hand-off phase in epoch $\ell-1$ to the end of the hand-off phase in epoch ℓ (see Figure 3).

We say that a server is "online" if it is active (in the above sense) or simply passively listening to broadcast communication. A protocol may potentially require a server to be online throughout the protocol and keep its local state up-to-date as a function of all the broadcast protocol messages (possibly for participation at a later stage). In such a case, while a server may not be performing active computation throughout the protocol, it would nevertheless have to commit to being present and listening throughout the protocol. To minimize the amount of online time of participants, ideally one would like servers to be online only when active.

Committee Sizes. In view of modeling committee members signing up as and when they have available computational resources, we allow for **variable com-**

mittee sizes in each epoch. This simply follows from allowing the environment $\mathcal{E}$ to determine how many parties it provides the nominate input. For simplicity, we describe our protocol in the technical sections for the simplified setting where the committee sizes in each epoch are equal and indicate how it extends to the variable committee size setting. An alternative choice would be to require the committee to have a fixed size, or change sizes at some prescribed rate. These choices might be more reasonable under the requirement that servers announce their committee membership at the start of the protocol.

Committee Overlap. In our envisioned applications, participants with available computational resources will sign up more often to be a part of a committee (see Remark 1). In view of this, we make **no restriction on committee overlap**, i.e., we allow a server to volunteer to be in multiple epoch committees. As we discuss below, this has some bearing on modeling security for the protocol.

Remark 1 (Weighted Computation.). We note that our model naturally allows for a form of *weighted computation*, where the amount of work performed by a participant is proportional to its available resources. This is because a participant (i.e., a server) can choose to participate in a number of epochs proportional to its available resources.

3.3 Security

As in traditional MPC, there are various choices for modeling corruption of parties to determine the number of parties that can be corrupted (i.e., honest vs dishonest majority) as well as the time of corruption (i.e., static vs adaptive corruption). The environment $\mathcal{E}$ can determine to corrupt a party, and on doing so, hands the local state of the corrupted party to the adversary $\mathcal{A}$. For a semi-honest (passive) corruption, $\mathcal{A}$ is only able to continue viewing the local state, but for a malicious (active) corruption, $\mathcal{A}$ takes full control of the party and instructs its behavior subsequently.

Corruption Threshold. We consider an *honest-majority* model for fluid MPC where we restrict $(\mathcal{A}, \mathcal{E})$ to the setting where the adversary **$\mathcal{A}$ controls any minority of the clients** as well as **any minority of servers in every committee in an epoch**.

We discuss the impact of the choice of committee formation on corruption threshold:

- **Volunteer Committee.** In the volunteer setting, ensuring honest majority in each epoch may be difficult; as such we view it as an optimistic model. In the extreme case, honest-majority per epoch can be enforced by assuming the global corruption threshold to be $N/2E$ where E is the total number of epochs and N is the total number of parties across all epochs.
- **Elected Committee.** In the elected committee model, the committee selection process may enforce an honest majority amongst the selected participants in every epoch. The work of [7] enforces this via a cryptographic

sortition process in proof-of-stake blockchains where an honest majority of stake is assumed (in fact they require a larger stake fraction to be honest for their committee selection).

An alternative model is where an adversary may control a majority of clients and additionally a majority of servers in one or more epochs. We leave the study of such a model for future work.

Corruption Timing. Given that the protocol progresses in discrete steps, and knowledge of committees may not be known in advance, it is important to model when an adversary can specify the list of corrupted parties. For clients, this is straightforward: we assume that the environment $\mathcal{E}$ specifies the list of corrupted clients at the start of the protocol, i.e. **we assume static corruption for the clients**. Since the servers perform the bulk of the computation, and their participation is already dynamic, there are various considerations for corruption timing. We consider two main aspects below: *point of corruption* and *effect on prior epochs*.

Point of corruption: When the committee $\mathcal{S}^\ell$ is determined at the start of hand-off phase of epoch $\ell - 1$, the adversary can specify the corrupted servers from $\mathcal{S}^\ell$ in either:

1. a *static* manner, where the environment $\mathcal{E}$ is only allowed to list the set of corrupted servers when the committee $\mathcal{S}^\ell$ is determined; or
2. an *adaptive* manner, where the environment $\mathcal{E}$ can corrupt servers in $\mathcal{S}^\ell$ adaptively up until the end of epoch ℓ, i.e. while they are active.

Effect on prior epochs: We consider the effect of the adversary corrupting parties during epoch ℓ on prior epochs.

1. *No retroactive effect:* In this setting, the corruption of servers during epoch ℓ has no bearing on any epoch $j < \ell$, i.e. the adversary does not learn any additional information about epoch j at epoch ℓ. This model can be achieved in two ways:

   Erasure of states: If servers in $\mathcal{S}^j$ *erase* their respective local states at the end of epoch j, then even if the server were to participate in epoch ℓ (i.e. $\mathcal{S}^j \cap \mathcal{S}^\ell \neq \emptyset$), the adversary would not gain any additional information when the environment $\mathcal{E}$ hands over the local state.

   Disjoint committees: If the sets of servers in each epoch are disjoint, by corrupting servers in epoch ℓ, the adversary cannot learn anything about prior epochs.

   We note that for any protocol that is oblivious to the real identities of the servers (i.e. the protocol doesn't assume any prior state from the servers), the two methods of achieving *no retroactive effect*, i.e. erasures and disjoint committees are equivalent. This follows from the fact that servers do not have to keep state in order to rejoin computation, and therefore from the point of view of the protocol and for all purposes, are equivalent to new servers.[5]

[5] We would like to point out that if one were to implement point-to-point channels via a PKI, this equivalence may not hold.

2. *Retroactive effect:* In this setting, the adversary is allowed limited information from prior epochs. Specifically, when corrupting a server $\mathsf{S} \in \mathcal{S}^\ell$ in epoch ℓ, the adversary learns private states of the server in all prior epochs (if the server has been in a committee before). Therefore, the S is then assumed to have been (passively) corrupt in every epoch $j < \ell$. In order to prevent the adversary from arbitrarily learning information about prior epochs, the adversary is limited to corrupting servers in epoch ℓ as long as corrupting a server S and its retroactive effect of considering S to be corrupted in all prior epochs does not cross the corruption threshold in *any* epoch.

One could consider models with various combinations of the aforementioned aspects. We will narrow further discussion to two models of the adversary:

Definition 2 (R-adaptive Adversary). *We say that the $(\mathcal{A}, \mathcal{E})$ results in an R-adaptive adversary $\mathcal{A}$ if the environment $\mathcal{E}$ can statically corrupt a set T of the clients (at the start of the protocol) and corrupt the servers in an adaptive manner with retroactive effect. Specifically, in epoch ℓ, the environment $\mathcal{E}$ can adaptively choose to corrupt a set of servers $T^\ell \subset [n_\ell]$ from the set $\mathcal{S}^\ell$, where T^ℓ corresponds to a canonical mapping based on the ordering of servers in $\mathcal{S}^\ell$. On $\mathcal{E}$ corrupting the server, $\mathcal{A}$ learns its entire past state and can send messages on its behalf in epoch ℓ. The set of servers that $\mathcal{E}$ can corrupt, and its corresponding retroactive effect, will be determined by the corruption threshold τ specifying that $\forall \ell,\ |T^\ell| < \tau \cdot n_\ell$.*

Definition 3 (NR-adaptive Adversary). *We say that the $(\mathcal{A}, \mathcal{E})$ results in an NR-adaptive adversary $\mathcal{A}$ if the environment $\mathcal{E}$ can statically corrupt a set T of the clients (at the start of the protocol) and corrupt the servers in an adaptive manner with* no *retroactive effect. The corruption process is similar to the case of R-adaptive adversaries, except that the environment $\mathcal{E}$ can corrupt any server in epoch ℓ as long as the number of corrupted servers in epoch ℓ are within the corruption threshold. As mentioned earlier, any protocol that achieves security against such an adversary necessarily requires either (a) erasure of state, or (b) disjoint committees.*

While our security definition will be general, and encompass both adversarial models, we will consider protocols in the model with **R-adaptive adversary**.

In the above discussions, we have considered corruptions only when servers are *active*. One could also consider a seemingly stronger model where the adversary can corrupt servers when they are *offline*, i.e. no longer *active*. We remark below that our model already captures offline corruption.

Remark 2 (Offline Corruption). If servers are *offline* once they are no longer *active* i.e. they are not passively listening to protocol messages, then offline corruptions in the retroactive effect model is the same as adaptive corruptions during (and until the end of) the epoch due to the fact that the server's protocol state has not changed since the last time it was active. Going forward, since honest parties do go offline when they are no longer active, we do not specify offline corruptions as they are already captured by our model.

Remark 3 (Un-corrupting parties). It might be desirable to consider a model in which a server is initially corrupted by the adversary, but then the adversary eventually decided to "un-corrupt" that server, returning it to honest status. This kind of "mobile adversary" has been studied in some prior works [31]. We note that this can be captured in our model by just having the adversary "un-corrupt" a server by making that server leave the computation at the end of the epoch and rely on the natural churn of the network to replace that server.

Defining Security. We consider a network of m-clients and N-servers $\mathcal{S}$ and denote by $(\overrightarrow{n} = (n_1, \ldots, n_E), E)$ the partitioning of the servers into E tuples (corresponding to epochs) where the ℓ-th tuple has n_ℓ parties (corresponding to committee in the ℓ-th epoch), i.e. $\mathcal{S}^\ell \subset \mathcal{S}$ such that $\forall \ell \in [E]$, $|\mathcal{S}^\ell| = n_\ell$.

Similar to the **client-server** setting, defined in [15, 17, 18], only the m clients have an input (and receive output), computing a function $f : X_1 \times \cdots \times X_m \to Y_1 \times \cdots \times Y_m$, where for each $i \in [m]$, X_i and Y_i are the input and output domains of the i-th client.

We provide a definition of fluid MPC that corresponds to the classical security notion in the MPC literature called **security with abort**, but note that other commonly studied security notions can also be defined in this setting in a straightforward manner. The security of a protocol (with respect to a functionality f) is defined by comparing the real-world execution of the protocol with an ideal-world evaluation of f by a trusted party. More concretely, it is required that for every adversary $\mathcal{A}$, which attacks the real execution of the protocol, there exist an adversary Sim, also referred to as a simulator in the ideal-world such that no environment $\mathcal{E}$ can tell whether it is interacting with $\mathcal{A}$ and parties running the protocol or with Sim and parties interacting with f. As mentioned earlier, the environment $\mathcal{E}$ (i) determines the inputs to the parties running the protocol in each round; (ii) sees the outputs to the protocol; and (iii) interacts in an arbitrary manner with the adversary $\mathcal{A}$. In this context, one can view the environment $\mathcal{E}$ as an interactive distinguisher.

It should be noted that it is only the clients that have inputs to the protocol π. While the servers have no input, the environment $\mathcal{E}$, in any round, can provide it with the input $\mathsf{nominate}$ upon which the server relays this message to the ideal functionality to indicate it is volunteering for the committee in the subsequent epoch. These servers have no output, so do not relay any information back to $\mathcal{E}$.

In the **real execution** of the $(\overrightarrow{n}, E)$-party protocol π for computing f in the presence of $f_{\mathsf{committee}}$ proceeds first with the environment passing the inputs to all the clients, who then pre-process their inputs and hand it off to the servers in $\mathcal{S}^1$. The protocol then proceeds in epochs as described earlier in the presence of an adversary $\mathcal{A}$ and environment $\mathcal{E}$. $\mathcal{E}$ at the start of the protocol chooses a subset of clients $T \subset [m]$ to corrupt and hands their local states to $\mathcal{A}$. As discussed, the corruption of the clients is static, and thus fixed for the duration of the protocol. The honest parties follow the instructions of π. Depending on whether $\mathcal{A}$ is $\mathsf{R\text{-}adaptive}$ or $\mathsf{NR\text{-}adaptive}$, $\mathcal{E}$ proceeds with adaptively corrupting servers and handing over their states to $\mathcal{A}$ who then sends messages on their behalf.

The execution of the above protocol defines $\mathsf{REAL}_{\pi,\mathcal{A},T,\mathcal{E},f_{\mathsf{committee}}}(z)$, a random variable whose value is determined by the coin tosses of the adversary and the honest players. This random variable contains (a) the output of the adversary (which may be an arbitrary function of its view); (b) the outputs of the uncorrupted clients; and (c) list of all the corrupted servers $\left\{T^{\ell}\right\}_{\ell\in[E]}$.

The **ideal world execution** is defined similarly to prior works. We formally define the ideal execution for the case of retroactive adaptive security, and the analogous definition for non-retroactive adaptive security can be obtained by appropriate modifications. Roughly, in the ideal world execution, the participants have access to a trusted party who computes the desired functionality f. The participants send their inputs to this trusted party who computes the function and returns the output to the participants.

More formally, an ideal world execution for a function f in the presence of $f_{\mathsf{committee}}$ with adversary Sim proceeds as follows:

- **Clients send inputs to the trusted party:** The clients send their inputs to the trusted party, and we let x_i' denote the value sent by client C_i. The adversary Sim sends inputs on behalf of the corrupted clients.
- **Corruption Phase of servers:** The trusted party initializes $\ell = 1$. Until Sim indicates the end of the current phase (see below), the following steps are executed:
  1. Trusted party sends ℓ to Sim and initializes an *append-only* list $\mathsf{Corrupt}^{\ell}$ to be $\emptyset$.
  2. Sim then sends pairs of the form (j, i) where j denotes epoch number and i denotes the *index of the corrupted server* in epoch $j \leq \ell$. Upon receiving this, the trusted party appends i to the list $\mathsf{Corrupt}^{j}$. This step can be repeated multiple times.
  3. Sim sends $\mathsf{continue}$ to the trusted party, and the trusted party increments ℓ by 1.

  Sim may also send an abort message to the trusted party in this phase in which case the trusted party sends $\perp$ to all honest clients and stops. Else, Sim sends $\mathsf{next\ phase}$ to the trusted party to indicate the end of the current phase.

  The following steps are only executed if the Sim has not already sent an abort message to the trusted.
- **Trusted party sends output to the adversary:** The trusted party computes $f(x_1', \ldots, x_m') = (y_1, \ldots, y_m)$ and sends $\{y_i\}_{i\in T}$ to the adversary Sim.
- **Adversary instructs trust party to abort or continue:** This is formalized by having the adversary send either a $\mathsf{continue}$ or abort message to the trusted party. In the latter case, the trusted party sends to each uncorrupted client C_i its output value y_i. In the former case, the trusted party sends the special symbol $\perp$ to each uncorrupted client.
- **Outputs:** Sim outputs an arbitrary function of its view, and the honest parties output the values obtained from the trusted party.

Sim also interacts with the environment $\mathcal{E}$ in an identical manner to the real execution interaction between $\mathcal{E}$ and $\mathcal{A}$. In particular this means, Sim cannot

rewind $\mathcal{E}$ or look at its internal state. The above ideal execution defines a random variable $\mathsf{IDEAL}_{\pi,\mathsf{Sim},T,\mathcal{E},f_{\mathsf{committee}}}(z)$ whose value is determined by the coin tosses of the adversary and the honest players. This random variable containing the (a) output of the ideal adversary Sim; (b) output of the honest parties after an ideal execution with the trusted party computing f where Sim has control over the adversary's input to f; and (c) the lists $\left\{\mathsf{Corrupt}^{\ell}\right\}_{\ell}$ of corrupted servers output by the trusted party. If Sim sends abort in the *corruption phase of the server*, the trusted party outputs the lists that have been updated until the point the abort message was received from Sim.

Having described the real and the ideal worlds, we now define security.

Definition 4. *Let $f : X_1 \times \cdots \times X_m \to Y_1 \times \cdots \times Y_m$ be a functionality and let π be a fluid MPC protocol for computing f with m clients, N servers and E epochs. We say that π achieves (τ, μ) retroactive adaptive security (resp. non-retroactive adaptive security) if for every probabilistic adversary $\mathcal{A}$ in the real world there exists a probabilistic simulator Sim in the ideal world such that for every probabilistic environment $\mathcal{E}$ if $\mathcal{A}$ is R-adaptive (resp. NR-adaptive) controlling a subset of servers $T^{\ell} \subseteq \mathcal{S}^{\ell}$, $\forall \ell \in [E]$ s.t. $|T^{\ell}| < \tau \cdot n_{\ell}$ and less than $\tau \cdot m$ clients, it holds that for all auxiliary input $z \in \{0,1\}^*$*

$$\mathsf{SD}\left(\mathsf{IDEAL}_{f,\mathsf{Sim},T,\mathcal{E},f_{\mathsf{committee}}}(z), \mathsf{REAL}_{\pi,\mathcal{A},T,\mathcal{E},f_{\mathsf{committee}}}(z)\right) \leq \mu$$

where $\mathsf{SD}(X, Y)$ is the statistical distance between distributions X and Y.

When μ is a negligible function of some security parameter λ, we say that the protocol π is τ-secure.

Remark 4. We note that the above definitions do not explicitly state whether the adversary behaves in (a) a semi-honest manner, where the messages that it sends on behalf of the parties are computed as per protocol specification; or (b) a malicious manner, where it can deviate from the protocol specification. Our intention is to give a general definition independent of the type of adversary. In the subsequent description, we will appropriately prefix the adversary with semi-honest/malicious to indicate the power of the adversary.

This Work. We summarize the fluid MPC model that we focus on in the full version of this paper [13], in the definition below.

Definition 5 (Maximally-Fluid MPC with R-Adaptive Security). *We say that a Fluid MPC protocol π is a* **Maximally-Fluid MPC with R-Adaptive Security** *if it additionally satisfies the following properties:*

- ***Fluidity:*** *It has maximal fluidity.*
- ***Volunteer Based Sign-up Model:*** *Committee for epoch $\ell+1$ is determined and known to everyone at the start of the hand-off phase of epoch ℓ where the sampling function for $f_{\mathsf{committee}}$ is the identity function. Each epoch can have variable committee sizes, and the committees themselves can arbitrarily overlap. A server is only required to be online during epochs where it is active.*

– ***Malicious R-Adaptive Security:*** *It achieves security as per Definition 4 against malicious R-adaptive adversaries who control any minority ($\tau < 1/2$) of clients and any minority of servers in every committee in an epoch.*

As we have just shown, there are many interesting, reasonable modeling choices that can be made in the study of fluid MPC. While our specific model name may be heavy-handed, we want to ensure that our modeling choices are clear throughout this work. Additionally, we hope to emphasize that our work is an initial foray in the study of fluid MPC and much is to be done to fully understand this setting.

4 Results in Full Version of the Paper

In the full version of this work [13], we construct a Maximally-Fluid MPC with R-Adaptive Security (see Definition 5). In this section, we outline the sequence of steps used for obtaining this result, and include the main theorems we prove for completeness.

1. We start by adapting the additive attack paradigm of [26] to the fluid MPC setting. In particular, we formally define a class of secret sharing based fluid MPC protocols, called "linear-based fluid MPC protocols". We then focus on "weakly private" linear-based fluid MPC protocols, which are semi-honest protocols that additionally achieve a weak notion of privacy against a malicious R-adaptive (see Definition 2) adversary. We show that such weakly private protocols are also secure against a malicious R-adaptive adversary up to "additive attacks". Formally, we prove the following theorem:

   Theorem 1. *Let Π be a Fluid MPC protocol computing a (possibly randomized) m-client circuit $C : \left(\mathbb{F}^{\mathsf{in}}\right)^m \to \mathbb{F}^{\mathsf{out}}$ using N servers that is a linear-based Fluid MPC with respect to a t-out-of-n secret sharing scheme, and is weakly-private against malicious R-adaptive adversaries controlling at most $t_\ell < n_\ell/2$ servers in committee $\mathcal{S}_\ell$ (for each $\ell \in [d]$) and $t < m/2$ clients, where d is the depth of the circuit C and n_ℓ are the number of servers in epoch ℓ. Then, Π is a $1/2$-secure Fluid MPC with R-Adaptive Security with d epochs for computing the additively corruptible version $\widetilde{f}_C$ of C.*

2. Next, we present a general compiler that can transform any linear based fluid MPC protocol that is secure against a malicious R-adaptive adversary up to additive attacks, into a protocol that achieves security with abort against a malicious R-adaptive adversary. Our resulting protocol only incurs a constant multiplicative overhead in the communication complexity of the original protocol and also preserves its fluidity. Formally, we prove the following theorem:

   Theorem 2. *Let $C : \left(\mathbb{F}^{\mathsf{in}}\right)^m \to \mathbb{F}^{\mathsf{out}}$ be a (possibly randomized) m-client circuit. Let $\tilde{C}$ be the robust circuit corresponding to C. Let Π be a Fluid*

MPC protocol computing $\tilde{C}$ using N servers that is linear-based with respect to a t-out-of-n secret sharing scheme, and is weakly-private against malicious R-adaptive adversaries controlling at most $t_\ell < n_\ell/2$ servers in committee $\mathcal{S}_\ell$ (for each $\ell \in [d+1]$) and $t < m/2$ clients, where d is the depth of the circuit C and n_ℓ is the number of servers in epoch ℓ. Then, the there exists a protocol that is a $1/2$-secure Fluid MPC with R-Adaptive Security with $d+1$ epochs for computing C. Moreover, this protocol preserves the fluidity of Π and only adds a constant multiplicative overhead to the communication complexity of Π.

3. Finally, we adapt the semi-honest protocol of Genarro, Rabin and Rabin [28], which is an optimized version of the classical semi-honest BGW protocol [6], to the fluid MPC setting and show that this protocol is both linear-based and weakly private against a malicious R-adaptive adversary, and achieves maximal fluidity. Using Theorem 1, we establish that this linear-based weakly private protocol is also secure against a malicious R-adaptive adversary up to additive attacks. Finally, we apply the compiler from Theorem 2 to this protocol to obtain a maximally fluid MPC protocol secure against malicious R-adaptive adversaries. Concretely, the following corollary holds directly from the two theorems above:

   Corollary 1. *There exists an information-theoretically secure Maximally-Fluid MPC with R-Adaptive Security (See Definition 5) for any $f \in P/Poly$.*

5 Acknowledgements

The fourth author would like to thank Amit Sahai and Sunoo Park for insightful discussions on dynamism in MPC. The fifth author would like to thank Shaanan Cohney for early discussions around blockchains and MPC.

Arka Rai Choudhuri, Aarushi Goel and Abhishek Jain were supported in part by DARPA/ARL Safeware Grant W911NF-15-C-0213, NSF CNS-1814919, NSF CAREER 1942789, Samsung Global Research Outreach award and Johns Hopkins University Catalyst award. Arka Rai Choudhuri is also supported by NSF Grants CNS-1908181 and Office of Naval Research Grant N00014-19-1-2294. Matthew Green is supported by NSF under awards CNS-1653110 and CNS-1801479, the Office of Naval Research under contract N00014-19-1-2292, DARPA under Contract No. HR001120C0084, and a Security and Privacy research award from Google. Abhishek Jain was additionally supported in part by an Office of Naval Research grant N00014-19-1-2294. Gabriel Kaptchuk is supported by the National Science Foundation under Grant #2030859 to the Computing Research Association for the CIFellows Project. Significant portions of this work were done while Gabriel Kaptchuk was at Johns Hopkins University and supported by NSF CNS-1329737. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA.

References

1. Araki, T., Barak, A., Furukawa, J., Lichter, T., Lindell, Y., Nof, A., Ohara, K., Watzman, A., Weinstein, O.: Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In: 2017 IEEE Symposium on Security and Privacy. pp. 843–862. IEEE Computer Society Press, San Jose, CA, USA (May 22–26, 2017)
2. Araki, T., Furukawa, J., Lindell, Y., Nof, A., Ohara, K.: High-throughput semi-honest secure three-party computation with an honest majority. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016. pp. 805–817. ACM Press, Vienna, Austria (Oct 24–28, 2016)
3. Barak, A., Hirt, M., Koskas, L., Lindell, Y.: An end-to-end system for large scale p2p mpc-as-a-service and low-bandwidth mpc for weak participants. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 695–712. CCS '18, ACM, New York, NY, USA (2018), http://doi.acm.org/10.1145/3243734.3243801
4. Baron, J., El Defrawy, K., Lampkins, J., Ostrovsky, R.: How to withstand mobile virus attacks, revisited. In: Halldórsson, M.M., Dolev, S. (eds.) 33rd ACM PODC. pp. 293–302. ACM, Paris, France (Jul 15–18, 2014)
5. Beerliová-Trubíniová, Z., Hirt, M.: Efficient multi-party computation with dispute control. In: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876, pp. 305–328. Springer, Heidelberg, Germany, New York, NY, USA (Mar 4–7, 2006)
6. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In: 20th ACM STOC. pp. 1–10. ACM Press, Chicago, IL, USA (May 2–4, 1988)
7. Benhamouda, F., Gentry, C., Gorbunov, S., Halevi, S., Krawczyk, H., Lin, C., Rabin, T., Reyzin, L.: Can a blockchain keep a secret? Cryptology ePrint Archive, Report 2020/464 (2020), https://eprint.iacr.org/2020/464
8. Bogdanov, D., Kamm, L., Kubo, B., Rebane, R., Sokk, V., Talviste, R.: Students and taxes: a privacy-preserving study using secure computation. Proceedings on Privacy Enhancing Technologies 2016(3), 117–135 (2016)
9. Buterin, V., et al.: A next-generation smart contract and decentralized application platform. white paper 3(37) (2014)
10. Chaum, D., Crépeau, C., Damgård, I.: Multiparty unconditionally secure protocols (abstract) (informal contribution). In: Pomerance, C. (ed.) CRYPTO'87. LNCS, vol. 293, p. 462. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 16–20, 1988)
11. Chen, J., Micali, S.: Algorand: A secure and efficient distributed ledger. Theor. Comput. Sci. 777, 155–183 (2019)
12. Chida, K., Genkin, D., Hamada, K., Ikarashi, D., Kikuchi, R., Lindell, Y., Nof, A.: Fast large-scale honest-majority MPC for malicious adversaries. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part III. LNCS, vol. 10993, pp. 34–64. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 19–23, 2018)
13. Choudhuri, A.R., Goel, A., Green, M., Jain, A., Kaptchuk, G.: Fluid mpc: Secure multiparty computation with dynamic participants. Cryptology ePrint Archive, Report 2020/754 (2020), https://eprint.iacr.org/2020/754
14. Clark, M.R., Hopkinson, K.M.: Transferable multiparty computation with applications to the smart grid. IEEE Trans. Inf. Forensics Secur. 9(9), 1356–1366 (2014)
15. Cramer, R., Damgård, I., Ishai, Y.: Share conversion, pseudorandom secret-sharing and applications to secure computation. In: Kilian, J. (ed.) TCC 2005. LNCS, vol.

3378, pp. 342–362. Springer, Heidelberg, Germany, Cambridge, MA, USA (Feb 10–12, 2005)
16. Cryptobiu: cryptobiu/libscapi (May 2019), https://github.com/cryptobiu/libscapi
17. Damgård, I., Ishai, Y.: Constant-round multiparty computation using a black-box pseudorandom generator. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 378–394. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 14–18, 2005)
18. Damgård, I., Ishai, Y.: Scalable secure multiparty computation. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 501–520. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 20–24, 2006)
19. Damgård, I., Nielsen, J.B.: Scalable and unconditionally secure multiparty computation. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 572–590. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 19–23, 2007)
20. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 19–23, 2012)
21. Dingledine, R., Mathewson, N., Syverson, P.: Tor: The second-generation onion router. In: Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13. pp. 21–21. SSYM'04, USENIX Association, Berkeley, CA, USA (2004), http://dl.acm.org/citation.cfm?id=1251375.1251396
22. Eldefrawy, K., Ostrovsky, R., Park, S., Yung, M.: Proactive secure multiparty computation with a dishonest majority. In: Catalano, D., De Prisco, R. (eds.) SCN 18. LNCS, vol. 11035, pp. 200–215. Springer, Heidelberg, Germany, Amalfi, Italy (Sep 5–7, 2018)
23. Furukawa, J., Lindell, Y.: Two-thirds honest-majority MPC for malicious adversaries at almost the cost of semi-honest. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 1557–1571. ACM Press (Nov 11–15, 2019)
24. Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part II. LNCS, vol. 9057, pp. 281–310. Springer, Heidelberg, Germany, Sofia, Bulgaria (Apr 26–30, 2015)
25. Genkin, D., Ishai, Y., Polychroniadou, A.: Efficient multi-party computation: From passive to active security via secure SIMD circuits. In: Gennaro, R., Robshaw, M.J.B. (eds.) CRYPTO 2015, Part II. LNCS, vol. 9216, pp. 721–741. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 16–20, 2015)
26. Genkin, D., Ishai, Y., Prabhakaran, M., Sahai, A., Tromer, E.: Circuits resilient to additive attacks with applications to secure computation. In: Shmoys, D.B. (ed.) 46th ACM STOC. pp. 495–504. ACM Press, New York, NY, USA (May 31 – Jun 3, 2014)
27. Genkin, D., Ishai, Y., Weiss, M.: Binary AMD circuits from secure multiparty computation. In: Hirt, M., Smith, A.D. (eds.) TCC 2016-B, Part I. LNCS, vol. 9985, pp. 336–366. Springer, Heidelberg, Germany, Beijing, China (Oct 31 – Nov 3, 2016)
28. Gennaro, R., Rabin, M.O., Rabin, T.: Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In: Coan, B.A., Afek, Y. (eds.) 17th ACM PODC. pp. 101–111. ACM, Puerto Vallarta, Mexico (Jun 28 – Jul 2, 1998)

29. Gentry, C., Halevi, S., Magri, B., Nielsen, J.B., Yakoubov, S.: Random-index pir and applications. Cryptology ePrint Archive, Report 2020/1248 (2020), https://eprint.iacr.org/2020/1248
30. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. In: Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017. pp. 51–68 (2017)
31. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. Cryptology ePrint Archive, Report 2017/454 (2017), http://eprint.iacr.org/2017/454
32. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: Aho, A. (ed.) 19th ACM STOC. pp. 218–229. ACM Press, New York City, NY, USA (May 25–27, 1987)
33. Goyal, V., Kothapalli, A., Masserova, E., Parno, B., Song, Y.: Storing and retrieving secrets on a blockchain. Cryptology ePrint Archive, Report 2020/504 (2020), https://eprint.iacr.org/2020/504
34. Goyal, V., Song, Y., Zhu, C.: Guaranteed output delivery comes free in honest majority MPC. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part II. LNCS, vol. 12171, pp. 618–646. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 17–21, 2020)
35. Herzberg, A., Jarecki, S., Krawczyk, H., Yung, M.: Proactive secret sharing or: How to cope with perpetual leakage. In: Coppersmith, D. (ed.) CRYPTO'95. LNCS, vol. 963, pp. 339–352. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 27–31, 1995)
36. Ikarashi, D., Kikuchi, R., Hamada, K., Chida, K.: Actively private and correct MPC scheme in $t < n/2$ from passively secure schemes with small overhead. Cryptology ePrint Archive, Report 2014/304 (2014), http://eprint.iacr.org/2014/304
37. Lapets, A., Volgushev, N., Bestavros, A., Jansen, F., Varia, M.: Secure mpc for analytics as a web application. In: 2016 IEEE Cybersecurity Development (SecDev). pp. 73–74. IEEE (2016)
38. Lindell, Y., Nof, A.: A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 259–276. ACM Press, Dallas, TX, USA (Oct 31 – Nov 2, 2017)
39. Maram, S.K.D., Zhang, F., Wang, L., Low, A., Zhang, Y., Juels, A., Song, D.: CHURP: dynamic-committee proactive secret sharing. In: ACM Conference on Computer and Communications Security. pp. 2369–2386. ACM (2019)
40. Micali, S.: Very simple and efficient byzantine agreement. In: Papadimitriou, C.H. (ed.) ITCS 2017. vol. 4266, pp. 6:1–6:1. LIPIcs, Berkeley, CA, USA (Jan 9–11, 2017)
41. Mohassel, P., Rosulek, M., Zhang, Y.: Fast and secure three-party computation: The garbled circuit approach. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 2015. pp. 591–602. ACM Press, Denver, CO, USA (Oct 12–16, 2015)
42. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system, 2008 (2008), http://www.bitcoin.org/bitcoin.pdf
43. Nordholt, P.S., Veeningen, M.: Minimising communication in honest-majority MPC by batchwise multiplication verification. In: Preneel, B., Vercauteren, F. (eds.) ACNS 18. LNCS, vol. 10892, pp. 321–339. Springer, Heidelberg, Germany, Leuven, Belgium (Jul 2–4, 2018)

44. Ostrovsky, R., Yung, M.: How to withstand mobile virus attacks (extended abstract). In: Logrippo, L. (ed.) 10th ACM PODC. pp. 51–59. ACM, Montreal, QC, Canada (Aug 19–21, 1991)
45. Pass, R., Seeman, L., shelat, a.: Analysis of the blockchain protocol in asynchronous networks. In: Coron, J.S., Nielsen, J.B. (eds.) EUROCRYPT 2017, Part II. LNCS, vol. 10211, pp. 643–673. Springer, Heidelberg, Germany, Paris, France (Apr 30 – May 4, 2017)
46. Pass, R., Shi, E.: FruitChains: A fair blockchain. In: Schiller, E.M., Schwarzmann, A.A. (eds.) 36th ACM PODC. pp. 315–324. ACM, Washington, DC, USA (Jul 25–27, 2017)
47. Wails, R., Johnson, A., Starin, D., Yerukhimovich, A., Gordon, S.D.: Stormy: Statistics in tor by measuring securely. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 615–632. ACM Press (Nov 11–15, 2019)
48. Yao, A.C.C.: How to generate and exchange secrets (extended abstract). In: 27th FOCS. pp. 162–167. IEEE Computer Society Press, Toronto, Ontario, Canada (Oct 27–29, 1986)