Extending the Security of SPDZ with Fairness

Bart Veldhuizen
The Netherlands
bartveldhuizen@outlook.com

Gabriele Spini
TNO
The Hague, The Netherlands
gabriele.spini@tno.nl

Thijs Veugen
TNO
The Hague, The Netherlands
University of Twente
Enschede, The Netherlands
thijs.veugen@tno.nl

Lisa Kohl
CWI, Cryptology Group
Amsterdam, The Netherlands
lisa.kohl@cwi.nl

ABSTRACT

SPDZ refers to a family of protocols for Secure Multi-Party Computation (MPC) that lie at the foundation of very popular software frameworks for MPC, such as SCALE-MAMBA and MP-SPDZ. SPDZ provides good efficiency while guaranteeing security even when all but one of the participants are corrupted. This seemingly optimal property comes at a price: the protocol only offers security with abort, meaning that even a single cheating participant can force the protocol to abort, leaving honest participants with no clue on what the correct output is, or who cheated. This is especially problematic since cheating participants are able to obtain the correct output of the computation, effectively 'stealing' it. We propose a *hybrid secure* adaptation to SPDZ, which retains the existing security guarantees, but in case the number of cheating players is less than half of the total, we achieve *fairness,* meaning that either all players obtain the correct output of the computation, or no player does. The 'less than half' threshold of corrupted players has been proven to be a tight bound to achieve fairness. Aside from the description of the protocol and its security proof, we also present a proof-of-concept implementation, and evaluate its practical performance, thereby demonstrating that our solution has negligible overhead compared to standard SPDZ in most application scenarios.

KEYWORDS

Secure Multi-Party Computation, SPDZ, fairness, hybrid security.

1 INTRODUCTION

In *Secure Multi-Party Computation,* or *MPC* for short, n mutually distrusting parties $P_1, \ldots, P_n$, each holding inputs $x_1, \ldots, x_n$ respectively, wish to evaluate a previously determined function f on their inputs, thereby obtaining $f(x_1, \ldots, x_n)$, without revealing any other information on their input. MPC thus simulates, with cryptographic means, an ideal model where a trusted third party receives all inputs $x_1, \ldots, x_n$, computes $f(x_1, \ldots, x_n)$ and sends the resulting value back to the parties, thus guaranteeing privacy of the inputs $x_1, \ldots, x_n$ and correctness of the output $f(x_1, \ldots, x_n)$.

Introduced by Yao [42] in the 1980s, MPC only began to be used in practical scenarios in the last decade, starting from the seminal work on agricultural-market auctions in Denmark [7], and subsequently other topics such as financial benchmarking [18], genome computation [28], and various other applications [43]. This increasing number of practical deployments of MPC is due, on one hand, to new MPC frameworks with very high efficiency, and on the other hand, the increased need for secure data sharing solutions.

Several settings for MPC have been investigated, with different assumptions regarding the number and behavior of participants, and with different security guarantees. The highly popular family of protocols based on SPDZ [19, 21, 32], which led to several widely used software frameworks [1, 30], supports any number $n \geq 2$ of participants, and guarantees security even in the presence of $n-1$ corrupted participants. This, however, has effects on the exact type of security that is achieved: while privacy of the inputs is guaranteed (under computational-hardness assumptions), only a weak form of correctness is ensured, known as *security with selective abort.* This means that no honest participant will be led to accept an incorrect result, even when all other participants cooperate by actively cheating and deviating from the instructions of the protocol. However, no 'positive' guarantee is given: cheating behavior will cause the protocol to abort, leaving the honest participants with no clue as to what the correct result is, or who cheated. This effectively allows a dishonest participant to perform a denial-of-service attack on the protocol. On top of this, an attack is described at the end of Subsection 2.3 that enables a *single* dishonest participant to obtain the correct result, while forcing the protocol to abort without any fear of getting caught, thus effectively 'stealing' the result. This can be highly problematic in various scenarios, such as auctions, where the inputs $x_1, \ldots, x_n$ are bids and the result is the highest bid or bidder: in this case, a dishonest party can learn the highest bid, while leaving the other participants with no clue on its value or who cheated, thus gaining an unfair advantage and effectively barring the use of protocols that merely guarantee security-with-abort in this setting. Similar problems occur in financial benchmarking, where one cheating party could walk away with exclusive access to important benchmarking results, and contract signing, where a single party may obtain a (possibly legally enforceable) signed contract. At its core one or more of the participants may be enticed

Proceedings on Privacy Enhancing Technologies 2024(2), 330–350

https://doi.org/10.56553/popets-2024-0053

to 'steal' the output $f(x_1, \ldots, x_n)$ in scenarios where the value of asymmetrical knowledge between the $P_1, \ldots, P_n$ may be larger than the value of the output only [13].

A possible mitigation to this problem in the context of SPDZ-based protocols was introduced in the form of cheater detection [41], where dishonest parties can still 'steal' the result, but will be identified as cheaters by the other parties (although honest parties will not necessarily identify the same cheater). More generally, the shortcomings of security with abort has inspired the study of secure computation protocols with *identifiable* abort [38]. However, this only solves part of the problem, since cheating parties are still able to obtain the result of the computation, and/or prevent other parties from doing so.

The ideal solution is captured by the notion of *fairness:* either all participants obtain the correct output of the computation, or no participant does. However, fairness cannot, in general, be guaranteed when half or more of the participants are corrupted [14], seemingly conflicting with the existing security guarantee of SPDZ, which holds in the presence of up to $n-1$ corrupted participants.

We address this apparent incompatibility with a *hybrid-secure* [11, 23] approach. Namely, we propose an adaptation to the SPDZ-based protocol(s) that guarantees different notions of security, depending on the number of corrupted participants t. If $n/2 \leq t < n$ *(dishonest majority),* we retain the security guarantees of SPDZ: no honest party will be led to accept an incorrect result. However, when $t < n/2$, we guarantee fairness, thus eliminating the possibility that cheating parties 'steal' the computation result. Privacy of the inputs is always guaranteed.

The variant of SPDZ we introduce thus has the potential of considerably increasing the scope of possible application scenarios of SPDZ-based software suites.

1.1 Contributions

Our work is an enhancement of the SPDZ-2 protocol [19], and is compatible with improvements that share the same online phase, such as MASCOT [31], LowGear and HighGear [32], and TopGear [3].

The modified version we introduce maintains the security guarantees of SPDZ, i.e. it ensures *privacy* of the parties' inputs and *security with abort*, even when all but one parties are corrupted. When only a minority of the parties is corrupted, our modified protocol achieves *fairness.* Guaranteeing fairness extends the use cases of the aforementioned protocols with those where asymmetric knowledge of the output is valuable and accepting an incorrect result is undesirable. The requirement that only a minority of parties may be corrupted has been shown to be, in general, a necessary condition to achieve fairness [14]. Furthermore, it was shown that the even stronger property of robustness cannot be guaranteed while also achieving security with abort for up to $n-1$ corruptions, for general MPC protocols based on secret sharing, such as the SPDZ-based protocols [27]. Extending the SPDZ-based protocols with fairness in the presence of an honest majority is therefore optimal. We stress the fact that the number of corrupted parties does *not* need to be given as input parameter to the protocol.

We describe our solution and formally prove that it satisfies the two aforementioned notions of security. Moreover, we present a proof-of-concept implementation and provide benchmarking results that show that the overhead of our solution, compared to 'standard' SPDZ, is negligible in both communication and computation.

A point of attention is that we only provide a security proof in the stand-alone model, as opposed to the universal-composability model [10] where the security of standard SPDZ is proved. This is due to the fact that, during the last phase of our protocol, parties need to unanimously agree on whether proceeding to attempt to reconstruct the output, or aborting; intuitively, this requires synchronous channels, and hence make a security proof in the UC model (which is inherently asynchronous) extremely complex at best [29]. We stress the fact that a lack of formal UC proof does not mean that our solution is inherently *in*secure when composed in parallel; for practical applications as those described in the previous subsection, in particular, we do not foresee this to pose limitations: for such applications, MPC solutions would arguably be run in a protected environment, with end-to-end encryption and authenticated network connections. On the one hand, parties could therefore agree to run a single execution if they deem attacks on composed protocols to be realistic; on the other hand, these environment would typically enforce measures, by e.g. direct cable connections or traffic control, that would prevent the adversary from arbitrarily delaying messages and disrupt unanimous agreement.

1.2 Related Work

The SPDZ-2 protocol [19], which forms the basis for our scheme, was introduced in 2012. The protocol improves upon the original SPDZ [21] protocol by introducing a (covert-) secure key-generation protocol, improving the overall efficiency of the preprocessing phase and introducing a new, efficient online phase. The new online phase uses a simpler MAC scheme, enabling the generation of shared randomness for multiple function evaluations using the same MAC-key.

Since the introduction of SPDZ-2, several improvements have been made with regard to the efficiency of the preprocessing phase. Keller et al. [31] introduced MASCOT, which improves the throughput of generating multiplication triples up to two orders of magnitude using symmetric key primitives in combination with oblivious transfer, instead of the previously used public-key schemes. Keller et al. [32] introduce the LowGear and HighGear protocols in the Overdrive paper, which improve the efficiency of the Zero-Knowledge Proof of Knowledge (ZKPoK) for a low and high number of participants respectively. Baum et al. [3] introduced TopGear, which leverages a new proving strategy to improve the efficiency of the ZKPoKs once more. Since all these protocols, including the SPDZ-2 protocol, use the same online phase, we refer to them as the SPDZ-based protocols.

Other work aims to extend the security that SPDZ-based protocols offer. Spini and Fehr [41] deter participants from cheating by adding *cheater detection.* Their scheme retains the original efficiency of the online phase when no cheating occurs. Baum et al. [5] extend SPDZ with *identifiable abort* and *public verifiability*, coming at the cost of increased complexity. Cunningham et al. [15] extend the protocol with three separate properties: *complete identifiable abort*, *complete identifiable auditability* and *openability.*

A very relevant work in the field of *hybrid security* is an impossibility result by Ishai et al. [27], proving that there exists no protocol based on secret sharing (as the SPDZ-based protocols do) that can guarantee security with abort for $n-1$ participants, while also guaranteeing robustness. A multitude of hybrid secure protocols have been proposed, including schemes that are hybrid in the number of corrupted participants [9], type of corruption (active/passive) [27], computational capabilities of the adversary [12] and a mix of these characteristics [26, 27].

Another relevant feasibility result was presented by Cleve [14], which shows that, in general, it is impossible to achieve fairness when a majority of the participants is corrupt. To guarantee fairness, a wide variety of techniques has been proposed. These include the use of trusted dealers [25, 34], stimulating honest behavior by linking the computation to cryptocurrencies [4, 33] or reputation systems [2], and using public bulletin boards [13]. An interesting approach to achieving fairness is that of *gradual release*, where sensitive information is released gradually. This gradual release of information ensures that the advantage of a malicious participant is bounded by a certain factor [6, 17, 26]. Notice that these works typically rely on GMW-like constructions [24], and are hence less efficient than SPDZ-based protocols, due to the use of zero-knowledge proofs.

A framework that can transform a variety of secret-sharing based semi-honest minority protocols into malicious minority protocols, guaranteeing security with abort, was introduced by Lindell and Nof [35]. They guarantee security with abort by verifying all multiplications that were performed in the computation stage, before they continue with the output-reconstruction stage. We note that they do not achieve fairness, although this is possible in the honest-majority setting.

Damgård et al. [20] present a general construction, based on verifiable secret sharing, that constructs a fair protocol from any protocol satisfying security with abort against a dishonest *minority* and having a special structure, called compute-then-open by the authors. While SPDZ seems to satisfy this condition, the authors do not make claims on a possible extension to the dishonest-majority setting.

Finally, Nordholt and Veeningen [37] continue on this work, including the addition of fairness. First, they propose a protocol that guarantees fairness in the malicious minority setting, but is only secure for a small number of participants. Second, they provide a SPDZ-based three-party computation protocol that achieves fairness in the malicious minority setting by blinding the output values with three input masks, one for every participant, before verifying them.

Table 1 provides an overall comparison of the results discussed in this section.

1.3 High-level Overview of our Construction

In this section, we provide some intuition on how our scheme works and provide some insights in its applicability to arbitrary actively-secure MPC protocols.

We begin with some intuition on the overall idea behind our construction. For simplicity, assume that the functionality f that the parties wish to compute produces a single output. Loosely speaking, our construction makes use of two components: an actively secure MPC protocol offering security with abort against $n-1$ corrupted parties, and a fair reconstruction protocol that, on input of a secret sharing of a given value, allows the parties to reconstruct that value, while guaranteeing correctness against $n-1$ corrupted parties, and fairness against $< n/2$ corrupted parties. In our case, the first protocol is given by SPDZ, while the second one is based on the conversion of an additive n-out-of-n secret sharing to a Shamir secret sharing with threshold $\lceil n/2 \rceil$, and subsequent reconstruction. The parties then use the MPC protocol to compute $f(x_1, \ldots, x_n)+b$, where b is a random 'masking' value, and to compute a secret sharing of b of the format required by the fair-reconstruction protocol. At this point, the parties simply use the latter protocol to fairly reconstruct b, then subtract this value from the output of the MPC protocol, and finally obtain $f(x_1, \ldots, x_n)$ in a fair way.

Intuitively, the security with abort against $n-1$ corrupted parties is guaranteed since both the MPC protocol and the fair-reconstruction protocol enjoy this property, while fairness is guaranteed by the reconstruction protocol: dishonest parties can only obtain $f(x_1, \ldots, x_n) + b$ for an unknown random value b through the (unfair) MPC protocol, and are thus unable to obtain any information related to the actual output $f(x_1, \ldots, x_n)$. At this point, reconstructing the actual output is equivalent to reconstructing the masking value b, and the fairness property of the reconstruction protocol assures fairness of the complete secure computation. This high-level intuition is therefore quite general in nature; in particular, it is not per se specific to the SPDZ protocol.

However, the actual construction described in this article uses particular aspects of the SPDZ protocol for improved efficiency. More precisely, recall that the fair-reconstruction protocol is given by a share conversion from an additive n-out-of-n sharing to a Shamir sharing. This conversion needs some additional security mechanisms to guarantee correctness; in our case, this is achieved by reusing security mechanisms from SPDZ. Therefore, instantiating the MPC protocol and the fair-reconstruction protocol separately would actually incur in some redundancy, since some operations that are necessary to achieve correctness would need to be executed twice. In order to gain efficiency, we instead execute them only once during the execution of SPDZ, and feed the values produced by them to the fair-reconstruction protocol. Moreover, we use another important efficiency aspect of SPDZ, namely its division into a preprocessing phase (independent of the computation inputs) and a very efficient 'online' phase, by executing some of the operations of the fair-reconstruction protocol during the preprocessing phase.

These adaptations result in a very efficient construction, which nicely adheres to the design principles of SPDZ. However, they also imply that the two components of our construction (an MPC protocol offering security with abort, and a fair-reconstruction protocol) can no longer be seen as two generic black-box components.

1.4 Organization

The paper is organized as follows. In Section 2 the required background knowledge and notation are described. Section 3 presents

Table 1: Comparison of most closely related work, excluding results based on trusted set-up assumptions.

	Fairness, $t < n/2$	Identifiable abort, $t \geq n/2$	Cheater detection, $t \geq n/2$	Security with abort, $t \geq n/2$	Any $n \geq 2$ supported	SPDZ-based
This work	✓	×	×	✓	✓	✓
SPDZ	×	×	×	✓	✓	✓
[41]	×	×	✓	✓	✓	✓
[5], [15]	×	✓	✓	✓	✓	✓
[20]	✓	×	×	×	✓	×
[37]	✓	×	×	✓	×	✓
[26]	✓	×	×	✓	✓	×

the fair SPDZ protocol by introducing a new reveal phase, accompanied with a complexity analysis. Section 4 analyses the practical performance of both the non-fair and fair protocol, by evaluating their running time and bandwidth usage. Section 5 provides the formal security statement of our construction, with complete proof found in Appendix A. Finally, conclusions are drawn in Section 6.

2 PRELIMINARIES

This section provides the background information which is necessary for our construction. Table 2 explains the meaning of the main variables that we use.

2.1 Security Setting

Several assumptions are made about the security setting in which our scheme, including the online phase of the SPDZ protocol, operates. The set of n participants $\mathcal{P}$ is partitioned in two non-overlapping sets: the set of h *honest* participants $\mathcal{H}$ and the set of t *actively corrupt* participants $\mathcal{A}$. The *adversary* is the central entity that controls the participants in $\mathcal{A}$. Since the proposed design is hybrid-secure in the number of corrupted participants, different corruption thresholds are assumed in different settings. To be precise, we achieve security with abort for $t < n$, while we can guarantee fairness with $t < n/2$. We therefore assume that $n > 2$, since otherwise fairness cannot be achieved.

Since our design extends the SPDZ-2 protocol [19], several assumptions are inherited. First, the adversary is *computationally bounded*. Second, the adversary is *static*, i.e. the adversary determines which parties to corrupt before the protocol starts. Finally, it is assumed that the participants have access to a complete *synchronous point-to-point* communication network, i.e. the communication between participants is private and authenticated, and messages sent by an honest participant cannot be arbitrarily delayed by the adversary.

It must be noted that the different preprocessing phases for SPDZ operate in various security settings (covert/active, …). We assume that the preprocessing phase securely generates all necessary data for the online phase under the aforementioned assumptions.

2.2 Protocol building blocks

The pseudorandom generator $\mathcal{U}_s(p,t)$ is a deterministic function that extends a seed s to a list of t pseudo-random elements in the field $\mathbb{F}_p$, as defined in [19]. Furthermore, we assume parties have access to a cryptographic hash function $\mathcal{H}_1$, which is modeled as a random oracle [19].

Table 2: Notation

n	total number of participants
P_i	participant i
$\mathcal{P}$	set of all participants
h	number of honest participants
$\mathcal{H}$	set of honest participants
t	number of actively corrupted parties
$\mathcal{A}$	set of actively corrupted parties
$\mathcal{U}_s(p,t)$	generator on seed s of t pseudorandom $\mathbb{F}_p$-elements
$\mathbb{F}_p$	finite field of prime order p
$SH(s)$	Shamir secret-sharing of s
$SH(s)_i$	Shamir share of participant i
k	Shamir reconstruction threshold
$w^{(i)}$	weight of i-th share during reconstruction
$C^{(i)}$	commitment of participant i
$O^{(i)}$	opening information of participant i
m	size of hash output (for broadcasts, commitments)
$[\![x]\!]$	additive secret-sharing of x
x_i	additive share of participant i
$\gamma(x)$	MAC-tag of value x
α	secret MAC-key ($\gamma(x) = x \cdot \alpha$)
$\langle\cdot\rangle$	sharing consisting of $[\![\cdot]\!]$ and $[\![\gamma(\cdot)]\!]$
q	number of unverified partially-opened values
$\tilde{a}^{(i)}$	i-th unverified partially-opened value
$\tilde{a}$	random linear combination of the $\tilde{a}^{(i)}$-s
σ	difference between MAC-tag of $\tilde{a}$ and original $\gamma(a)$
$y^{(i)}$	i-th output value
ℓ	the number of output values
b	random value masking the output
z	blinded output value ($z = y + b$)

2.2.1 Shamir's secret-sharing scheme. Shamir's secret-sharing scheme [40] is a powerful secret-sharing scheme that operates over a finite field. We will only discuss this scheme over the finite field

$\mathbb{F}_p$, where p is prime.[1] A (k, n)-sharing of secret s indicates that the secret s is split into n shares, of which k are needed to reconstruct the secret s again. The complete Shamir sharing of s is denoted by $SH(s)$, while the Shamir share of participant $P_i \in \mathcal{P}$ is denoted by $SH(s)_i$. In order to share a secret value s, a random $(k-1)$-degree polynomial f is sampled, such that s is the constant term, and n unique non-zero points on this polynomial are computed, which act as the shares. In our scheme, the point $(i, f(i))$ is always given to participant P_i.

The secret value s can be reconstructed by interpolating the polynomial f. Following Lagrange's interpolation theory, we know that it is always possible to interpolate a $(k-1)$-degree polynomial using k points. To reconstruct from any set of k points $(x_1, f(x_1)), \ldots, (x_k, f(x_k))$ we use Lagrange's interpolation formula, albeit in a different form, which allows us to precisely formulate the required number of operations to reconstruct the polynomial f (Section 3.3). The polynomial is calculated as $f(x) = \sum_{j=1}^{k} c_0^{(j)} \cdot f(x_j) \cdot x^0 + \ldots + \sum_{j=1}^{k} c_{k-1}^{(j)} \cdot f(x_j) \cdot x^{k-1}$, which is a linear combination of the Shamir shares and reconstruction coefficients $c_0^{(1)} \ldots, c_{k-1}^{(k)}$. These coefficients are calculated as follows:

$$c_i^{(j)} = (-1)^z \cdot w^{(j)} \cdot \sum_{\substack{m_1=1 \\ m_1 \neq j}}^{k} \cdots \sum_{\substack{m_z=m_{z-1}+1 \\ m_z \neq j}}^{k} (x_{m_1} \cdot \ldots \cdot x_{m_z}),$$

where $z = k - i - 1$. The weighting terms $w^{(1)}, \ldots, w^{(k)}$ are calculated as $w^{(j)} = (\prod_{i=1\ldots k, i\neq j}(x_j - x_i))^{-1}$. Once the polynomial f is interpolated, it is verified that the remaining points $(x_{k+1}, f(x_{k+1})), \ldots, (x_n, f(x_n))$ also lie on f. Since $c_0^{(1)}, \ldots, c_{k-1}^{(k)}$ and $w^{(1)}, \ldots, w^{(k)}$ are constant for the same $x_1, \ldots, x_k$, the secret s can be efficiently constructed, when these constants have been precomputed.

We refer to the complete interpolation as Reconstruct, while only reconstructing the secret value using precomputed weighting terms and coefficients is referred to as EffReconstruct.

2.2.2 Broadcast. The Broadcast protocol is used to let participants communicate a message to all other participants, while ensuring consistency of the received messages. Notice that we only describe a situation where all participants are required to broadcast a message simultaneously, and that a weak form of broadcast, offering security with selective abort, will be enough for our purposes. We formally model the required broadcasting functionality in $\mathcal{F}_{\text{Broadcast}}$ (Funct. 3; a formal definition is provided in Appendix C). We write $\text{Broadcast}(m^{(1)}, \ldots, m^{(n)})$ to denote the parties jointly running an instance of the broadcast functionality $\mathcal{F}_{\text{Broadcast}}$, where party P_i broadcasts $m^{(i)}$.

In the random-oracle model, this functionality can be implemented as follows: Each participant $P_i \in \mathcal{P}$ begins by communicating their message $m^{(i)}$ plus a broadcast message to all other participants. Subsequently, each participant calculates the hash $\mathcal{H}_1(m)$, where $m = m^{(1)} \parallel \ldots \parallel m^{(n)}$, and communicates this hash to all other participants. Finally, each participant compares the received hashes to their own, ensuring that each participant received the same messages (or outputs $\perp$).

[1] The reason for this will become clear in Protocol 4.

2.2.3 SecureOpen. The SecureOpen protocol simulates the simultaneous revealing of elements $\xi^{(1)}, \ldots, \xi^{(n)}$, where each $\xi^{(i)}$ is held by participant P_i. We formally model the required selective opening functionality in $\mathcal{F}_{\text{SecureOpen}}$ (Funct. 4; a formal definition is provided in Appendix C). We write $\text{SecureOpen}(\xi^{(1)}, \ldots, \xi^{(n)})$ to denote the parties jointly running an instance of the secure opening functionality $\mathcal{F}_{\text{SecureOpen}}$, where party P_i opens $\xi^{(i)}$.

In the random oracle model, the functionality can be instantiated as follows. First, each participant $P_i \in \mathcal{P}$ runs $\text{Commit}\left(\xi^{(i)}\right)$, which generates an r from the output range of hash function $\mathcal{H}_1$, and returns the tuple $\left(C^{(i)}, O^{(i)}\right) = \left(\mathcal{H}_1\left(\xi^{(i)} \parallel r\right), \xi^{(i)} \parallel r\right)$. The commitments $C^{(1)}, \ldots, C^{(n)}$ are broadcast, preventing anyone from claiming an input $\tilde{\xi} \neq \xi$. Once all commitments are received, the participants broadcast the opening information $O^{(1)}, \ldots, O^{(n)}$. To verify the correctness of the commitments, the function $\text{Open}\left(C^{(i)}, O^{(i)}\right)$ is executed for $P_i \in \mathcal{P}$, i.e. it is verified whether $C^{(i)} = \mathcal{H}_1\left(O^{(i)}\right)$.

2.2.4 $\mathcal{F}_{\text{Continue}}$. We further need a protocol that allows the honest parties to agree whether to continue with the protocol execution before opening the blinding values; this functionality has to satisfy security with unanimous abort in the honest-majority case. We formalize this notion in $\mathcal{F}_{\text{Continue}}$ (Funct. 5; a formal definition is provided in Appendix C). This functionality can be instantiated with protocols for detectable byzantine agreement (which in fact satisfies a stronger notion of fairness, both in the honest and dishonest majority case). By [22] detectable byzantine agreement can be achieved unconditionally for any number of corruptions $t < n$.

2.3 An Overview of SPDZ-2

The protocol introduced in [19], referred to as SPDZ-2, is a general multi-party computation protocol that guarantees security with abort against up to $n-1$ active corrupt participants, while still providing efficient function evaluation. A key factor for achieving this efficiency is the separation of the protocol into a *preprocessing* and *online* phase. In the time-consuming preprocessing phase, the participants generate shared randomness. Subsequently, in the online phase, the output of a given function f is computed by evaluating a circuit over the inputs of the participants. In this phase, the shared randomness is consumed to efficiently perform certain operations, such as sharing input data, and multiplying and squaring intermediate (secret) values. The preprocessing phase can be run before the function f and the inputs are known (only an upper bound on the number of inputs and on the different types of gates of f, expressed as a circuit, is needed). Other protocols have been proposed to generate shared randomness more efficiently, such as the MASCOT, LowGear, HighGear and TopGear protocols [3, 31, 32].

Privacy is achieved by additively secret-sharing each value throughout the circuit. An additively secret-shared value x is denoted as $[\![x]\!]$, where each participant $P_i \in \mathcal{P}$ holds share x_i and the sum of the shares of the participants $\sum_{i=1}^{n} x_i$ is equal to x in $\mathbb{F}_p$. Correctness is guaranteed by adding an additively secret-shared MAC-tag to each value throughout the circuit. The MAC-tag of a value x, denoted as $\gamma(x)$, is equal to $x \cdot \alpha$, where α is the secret MAC-key. This MAC-key α is additively secret-shared itself. Each secret-shared value throughout the circuit therefore consists of

the additive sharings of the value and the corresponding MAC-tag, referred to as a $\langle\cdot\rangle$-sharing, where $\langle\cdot\rangle = (\llbracket\cdot\rrbracket, \llbracket\gamma(\cdot)\rrbracket)$. The participants can *partially open* a $\langle\cdot\rangle$-sharing by opening the shares of its underlying $\llbracket\cdot\rrbracket$-sharing. This process, known as PartialOpen, consists of two steps. First, the participants $P_1, \ldots, P_n$ send their additive shares to a designated participant — say, P_1. Second, P_1 calculates the sum of the respective shares and distributes these to the participants $P_2, \ldots, P_n$.

On several occasions the participants receive unverified values; to make this situation clear, we add a tilde symbol to such a value, obtaining e.g. $\tilde{a}$. The MAC-Check protocol, formalized in Protocol 1, is used to determine whether any number of q unverified partially-opened values are correctly $\langle\cdot\rangle$-shared. Assuming $q > 1$, the participants start by calculating a random linear combination over the unverified partially opened values $\tilde{a}^{(1)}, \ldots, \tilde{a}^{(q)}$ to obtain a single value $\tilde{a} = \sum_{i=1}^{q} r^{(i)} \cdot \tilde{a}^{(i)}$, given the jointly generated random values $r^{(1)}, \ldots, r^{(q)}$. A similar random linear combination is calculated over the MAC-tags, resulting in $\llbracket\gamma(a)\rrbracket = \sum_{i=1}^{q} r^{(i)} \cdot \left\llbracket \gamma\left(a^{(i)}\right)\right\rrbracket$. Subsequently, the participants recalculate the MAC-tag over the value $\tilde{a}$ and subtract it from the original MAC-tag $\gamma(a)$, resulting in $\langle\sigma\rangle$. Afterwards, the participants securely open the $\llbracket\sigma\rrbracket$-sharing, and verify whether it equals zero. If so, the original MAC-tag and the recalculated MAC-tag were equal, guaranteeing that the values $\tilde{a}^{(1)}, \ldots, \tilde{a}^{(q)}$ were correctly $\langle\cdot\rangle$-shared. If not, the protocol returns the abort message $\perp$, resulting in the fact that $\perp$ is broadcast to all participants, who then immediately abort the protocol.

At the end of the online phase, once the circuit is evaluated and the output sharings $\left\langle y^{(1)}\right\rangle, \ldots, \left\langle y^{(q)}\right\rangle$ have been calculated, the reveal phase is initialized. The standard, 'non-fair' reveal phase is formalized in Protocol 2. This reveal phase starts by verifying whether the computation has been performed correctly. This is guaranteed by verifying whether all earlier t partially opened values were correctly $\langle\cdot\rangle$-shared, using the MAC-Check protocol. Afterwards, the participants partially open the ℓ output sharings $\left\langle y^{(1)}\right\rangle, \ldots, \left\langle y^{(\ell)}\right\rangle$, which are also verified using the MAC-Check protocol.

It can now be seen that it is quite simple for the adversary to obtain the correct output(s), while preventing the honest participants from doing so. Namely, the adversary can instruct the dishonest participants to correctly execute the protocol up until the last step, meaning that all parties do obtain correct sharings $\left\langle y^{(1)}\right\rangle, \ldots, \left\langle y^{(\ell)}\right\rangle$ of the output values. At this point, these values will be partially opened, and hence the values $y^{(1)}, \ldots, y^{(\ell)}$ become public, and subsequently, the MAC-Check protocol is executed in order to confirm their correctness. Now the dishonest participants can simply cheat during the execution of MAC-Check (e.g., by sending incorrect values σ_j), hence leading the protocol to fail: this means that honest participants will *not* accept $y^{(1)}, \ldots, y^{(\ell)}$ as the correct output values, and will therefore be left with no output. On the other hand, the adversary knows that these values were actually correctly computed, and that it was only their correctness check that was improperly executed; the adversary thus obtains these (correct) values as output.

Protocol 1 MAC-Check: Verifying SPDZ sharings.

Inputs:

(1) The partially opened values $\tilde{a}^{(1)}, \ldots, \tilde{a}^{(q)}$.
(2) The sharings $\left\langle a^{(1)}\right\rangle, \ldots, \left\langle a^{(q)}\right\rangle$.

Output:

(1) Each party either obtains the values $a^{(1)}, \ldots, a^{(q)}$ or an abort message $\perp$.

The protocol:

(1) **Sampling random elements**
  (a) If $q > 1$:
    (i) Each participant $P_i \in \mathcal{P}$ samples a random element s_i.
    (ii) The participants run SecureOpen$(s_1, \ldots, s_n)$.
    (iii) The participants set seed $s = s_1 + \cdots + s_n$.
    (iv) The participants sample q random elements $r^{(1)}, \ldots, r^{(q)} = \mathcal{U}_s(p, q)$.
  (b) If $q = 1$:
    (i) The participants set $r^{(1)} = 1$.
(2) **Recalculating MAC values**
  (a) The participants compute the linear combination $\tilde{a} = \sum_{i=1}^{q} r^{(i)} \cdot \tilde{a}^{(i)}$.
  (b) The participants calculate $\llbracket\gamma(a)\rrbracket = \sum_{i=1}^{q} r^{(i)} \cdot \left\llbracket \gamma\left(a^{(i)}\right)\right\rrbracket$.
  (c) The participants calculate $\llbracket\sigma\rrbracket = \llbracket\gamma(a)\rrbracket - \llbracket\alpha\rrbracket \cdot \tilde{a}$.
(3) **Verifying recalculated MAC values**
  (a) The participants run SecureOpen$(\sigma_1, \ldots, \sigma_n)$.
  (b) The participants calculate $\sigma = \sum_{i=1}^{n} \sigma_i$. If $\sigma \neq 0$, the overall protocol is aborted.

3 PROPOSED SCHEME

3.1 Overview

Our scheme extends all SPDZ-based protocols with the fairness security guarantee in the presence of an honest majority. This is achieved by the introduction of a novel reveal phase, which is consistent between SPDZ-based protocols, and fairly opens the output values.

To guarantee fairness, our scheme transforms the (n, n)-sharings of the output values into $(\lceil n/2\rceil, n)$-sharings. Such $(\lceil n/2\rceil, n)$-sharings guarantee fairness in an honest majority setting (where the number of honest participants is equal to, or larger than, the reconstruction threshold).

The $(\lceil n/2\rceil, n)$-sharings use Shamir's secret-sharing scheme, since this scheme has a variable reconstruction threshold, operates on prime fields, and allows for linear operations. The latter two characteristics are needed to efficiently transform the SPDZ-sharings to Shamir sharings (Section 3.2.1) and to prove consistency between these sharings (Section 3.2.2), respectively.

Consistency between the $(\lceil n/2\rceil, n)$-Shamir sharings and the (n, n)-additive sharings from which they originate, is guaranteed by generating validation data. These $(\lceil n/2\rceil, n)$-Shamir sharings and their validation data are independent from the actual values

Protocol 2 StandardReveal: Standard Reveal Protocol.

Inputs:

(1) The partially opened values $\tilde{a}^{(1)}, \ldots, \tilde{a}^{(t)}$.
(2) The sharings $\left\langle a^{(1)} \right\rangle, \ldots, \left\langle a^{(t)} \right\rangle$.
(3) The sharings $\left\langle y^{(1)} \right\rangle, \ldots, \left\langle y^{(\ell)} \right\rangle$.

Output:

(1) Each party either obtains the values $y^{(1)}, \ldots, y^{(\ell)}$ or an abort message $\perp$.

The protocol:

(1) **Verifying computation**
   (a) The participants run $\textit{MAC-Check}\left(\left(\tilde{a}^{(1)}, \ldots, \tilde{a}^{(t)}\right), \left(\left\langle a^{(1)} \right\rangle, \ldots, \left\langle a^{(t)} \right\rangle\right)\right)$.
(2) **Verifying output values**
   (a) The participants obtain $\tilde{y}^{(1)}, \ldots, \tilde{y}^{(\ell)}$ by executing $\textit{PartialOpen}\left(\left\langle y^{(1)} \right\rangle, \ldots, \left\langle y^{(\ell)} \right\rangle\right)$.
   (b) The participants run $\textit{MAC-Check}\left(\left(\tilde{y}^{(1)}, \ldots, \tilde{y}^{(\ell)}\right), \left(\left\langle y^{(1)} \right\rangle, \ldots, \left\langle y^{(\ell)} \right\rangle\right)\right)$.

of the output, and only depend on the number of output values. As a result, these data do not reveal any information on the reconstructed output values, except its count. Once the Shamir shares are opened, the validation data enable a participant to determine whether a Shamir sharing is correct and consistent, without requiring additional communication.

To optimize the practical performance of the scheme, the transformation from additive sharings to Shamir sharings, and the generation of validation data, are moved to the preprocessing phase. This approach minimizes the increase in running time of the low-latency online phase, whose efficiency is one of the main advantages of SPDZ-based protocols.

The resulting reveal phase, formalized in Protocol 3, operates as follows. First, all partially opened values are verified (Step 1a), just as in the non-fair reveal phase (Protocol 2). Second, *blinding sharings*, i.e. random SPDZ sharings, are added to the output SPDZ sharings (Step 2a). These blinded output SPDZ-sharings are then opened and verified to be correct (Steps 2b-2c); notice that this check remains secure even in the presence of a dishonest majority, i.e., it will fail if the value is incorrect. After this step, the MAC-check plays no further role: once the hidden output SPDZ-sharings are verified, the actual output values can be calculated by subtracting the aforementioned random SPDZ-sharings from the hidden output SPDZ-sharings. Since these random SPDZ-sharings have been transformed in the preprocessing phase to Shamir sharings, these values can be reconstructed by publishing the Shamir sharings; since the $(\lceil n/2 \rceil, n)$ Shamir sharings can be fairly opened in the presence of an honest majority, the output values are fairly opened in this setting. Therefore, the protocol ends by reconstructing the random secrets from the Shamir sharings (Step 3a) and subtracting these from the verified hidden output values (Step 3b). The results are the output values, given that the reconstructed random secrets from the Shamir sharings are consistent with the secrets in the random SPDZ-sharings.

Protocol 3 FairOutputs: Fair opening of output values.

Inputs:

(1) The partially opened values $\tilde{a}^{(1)}, \ldots, \tilde{a}^{(t)}$.
(2) The sharings $\left\langle a^{(1)} \right\rangle, \ldots, \left\langle a^{(t)} \right\rangle$.
(3) The sharings $\left\langle y^{(1)} \right\rangle, \ldots, \left\langle y^{(\ell)} \right\rangle$.
(4) The blinding sharings $\left\langle b^{(1)} \right\rangle, \ldots, \left\langle b^{(\ell)} \right\rangle$.
(5) The validation data ValData.

Output:

(1) The values $y^{(1)}, \ldots, y^{(\ell)}$, or an abort message $\perp$.

The protocol:

(1) **Verifying Computation**
   (a) The participants run MAC-Check $\left(\left(\tilde{a}^{(1)}, \ldots, \tilde{a}^{(t)}\right), \left(\left\langle a^{(1)} \right\rangle, \ldots, \left\langle a^{(t)} \right\rangle\right)\right)$.
(2) **Blinding outputs**
   (a) The participants calculate $\left\langle z^{(1)} \right\rangle, \ldots, \left\langle z^{(\ell)} \right\rangle$, where $\left\langle z^{(i)} \right\rangle = \left\langle y^{(i)} \right\rangle + \left\langle b^{(i)} \right\rangle$, for all $1 \leq i \leq \ell$.
   (b) The participants obtain $\tilde{z}^{(1)}, \ldots, \tilde{z}^{(\ell)}$ by running $\textsc{PartialOpen}\left(\left\langle z^{(1)} \right\rangle, \ldots, \left\langle z^{(\ell)} \right\rangle\right)$.
   (c) The participants run MAC-Check $\left(\left(\tilde{z}^{(1)}, \ldots, \tilde{z}^{(\ell)}\right), \left(\left\langle z^{(1)} \right\rangle, \ldots, \left\langle z^{(\ell)} \right\rangle\right)\right)$.
(3) **Fair output reveal**
   (a) The participants fairly obtain the *blinding* values $b^{(1)}, \ldots, b^{(\ell)}$ by running FairBlinds(ValData).
   (b) The participants calculate $y^{(1)}, \ldots, y^{(\ell)}$, where $y^{(i)} = z^{(i)} - b^{(i)}$, for all $1 \leq i \leq \ell$.

3.2 Design of the Proposed Scheme

This section formalizes the protocols that transform SPDZ-sharings into Shamir sharings, generate the corresponding validation data, and verify opened Shamir shares based on the validation data.

3.2.1 Transformation of Sharings. SPDZ-sharings can efficiently be transformed to Shamir sharings by leveraging the linearity of Shamir's secret-sharing scheme. Protocol 4 formalizes the transformation of $\ell + 1$ sharings.[2] Transforming a single SPDZ-sharing consists of two major operations. First, each participant creates a $(\lceil n/2 \rceil, n)$ Shamir sharing of their additive SPDZ-share and distributes this among the participants (Step 1). Second, each participant sums their n received Shamir shares (Step 2). The resulting Shamir shares can be used to reconstruct the same secret as their corresponding SPDZ-shares when all participants follow the protocol.

[2]One additional sharing is generated to be sacrificed further in the Protocol. This process is discussed later.

Protocol 4 ShareShamir: Transforming SPDZ-sharings to Shamir sharings.

Inputs:

(1) The blinding sharings $\left\langle b^{(1)} \right\rangle, \ldots, \left\langle b^{(\ell+1)} \right\rangle$.

Output:

(1) The blinding sharings $SH\left(\tilde{b}^{(1)}\right), \ldots, SH\left(\tilde{b}^{(\ell+1)}\right)$.//In the honest case $SH(\tilde{b}^{(i)})$ corresponds to a Shamir sharing of $b^{(i)}$.

The protocol:

For $\langle b \rangle \in \left(\left\langle b^{(1)} \right\rangle, \ldots, \left\langle b^{(\ell+1)} \right\rangle\right)$:

(1) **Shamir sharing:**
Each participant $P_i \in \mathcal{P}$ generates a $\left(\lceil \frac{n}{2} \rceil, n\right)$ Shamir sharing of $\tilde{b}_i$, and distributes the shares.

(2) **Combining Shamir shares**
The participants calculate the Shamir sharing $SH\left(\tilde{b}\right) = \sum_{i=1}^{n} SH\left(\tilde{b}_i\right)$.

3.2.2 Generating Validation Data. The validation data are generated using a Commit-Challenge-Response approach. The complete process is formalized in Protocol 5. First, each participant commits to their unverified Shamir shares by broadcasting a commitment (Step 1a-1b). Second, the participants securely generate a random seed, using the SecureOpen protocol (Step 2a-2c). This seed is used to generate the challenge (Step 2d), i.e. a list of random elements, from the pseudo-random generator $\mathcal{U}_s(p, \ell+1)$ (Section 2.2). Third, each participant calculates their response, which equals the random linear combination of their Shamir shares, where the coefficients are the aforementioned list of random elements (Step 3a). Subsequently, the resulting Shamir sharing of the secret value c is securely opened and c is reconstructed (Step 3b-3c).

The MAC-Check protocol is used to verify that c is consistent between the Shamir sharing and the SPDZ-sharing (Step 3d-3f).[3]

The validation data consist of the commitments to the Shamir shares, the list of random elements, and the published Shamir sharing of c. Intuitively, the commitments and random elements reveal no information about the random secret values, which is important as they are used to hide the output values in the reveal phase. Moreover, one of the $\ell + 1$ secret values, used to calculated c, is 'sacrificed', i.e. unused. This is necessary to prevent information on the blinding values to be prematurely leaked, while checking the consistency of the SPDZ- and Shamir-shares.

3.2.3 Verifying Shamir sharings. The opening of the Shamir sharings is formalized in Protocol 6. Once the sharings are opened, the generated validation data are used to determine whether they are consistent with the SPDZ-sharings they originate from. Whenever published Shamir shares of a participant are inconsistent with the validation data, the participant is identified as a cheater and their shares are discarded. This process repeats until the protocol is either finished, or until there are insufficient 'non-cheating' shares remaining.

[3] This trick enables us to verify a Shamir sharing with the MAC-data of a SPDZ-sharing.

Protocol 5 GenValData: Generating Validation Data.

Inputs:

(1) The blinding sharings $\left\langle b^{(1)} \right\rangle, \ldots, \left\langle b^{(\ell+1)} \right\rangle$.

(2) The blinding sharings $SH\left(\tilde{b}^{(1)}\right), \ldots, SH\left(\tilde{b}^{(\ell+1)}\right)$.

Outputs:

Each party either receives the validation data ValData, consisting of the following public data:

(1) The commitments over the Shamir shares $C^{(1)}, \ldots, C^{(n)}$. //In the honest case, $C^{(i)}$ corresponds to a commitment to the i-th share of $SH(b^{(1)}) \| \ldots \| SH(b^{(\ell+1)})$

(2) The opening data $O^{(1)}, \ldots, O^{(n)}$ for the above commitments.

(3) The random values $r^{(1)}, \ldots, r^{(\ell+1)}$.

(4) The Shamir sharing of the linear combination of blinding values $SH(c)$. //In the honest case, $SH(c) = \sum_{i=1}^{\ell+1} r^{(i)} \cdot SH(b^{(i)})$.

or an abort message $\perp$.

The protocol:

(1) **Commit phase**

(a) Each participant $P_i \in \mathcal{P}$ generates the commitment data over their Shamir shares $\left(C^{(i)}, O^{(i)}\right) = \text{Commit}\left(SH\left(\tilde{b}^{(1)}\right)_i \| \ldots \| SH\left(\tilde{b}^{(\ell+1)}\right)_i\right)$.

(b) The participants obtain $C^{(1)}, \ldots, C^{(n)}$ by running Broadcast $\left(C^{(1)}, \ldots, C^{(n)}\right)$.

(2) **Challenge phase**

(a) Each participant $P_i \in \mathcal{P}$ generates a random seed value s_i.

(b) The participants run SecureOpen $(s_1, \ldots, s_n)$.

(c) The participants set seed $s = \sum_{i=1}^{n} s_i$.

(d) The participants obtain the random values $r^{(1)}, \ldots, r^{(\ell+1)} = \mathcal{U}_s(p, \ell+1)$.

(3) **Response phase**

(a) The participants calculate the random linear combination of their Shamir shares $SH(c) = \sum_{i=1}^{\ell+1} r^{(i)} \cdot SH\left(\tilde{b}^{(i)}\right)$.

(b) The participants run SecureOpen $(SH(c)_1, \ldots, SH(c)_n)$.

(c) Each party locally calculates $c =$ Reconstruct $(SH(c)_1, \ldots, SH(c)_n)$.

(d) The participants calculate the random linear combination of blinding values $\langle c \rangle = \sum_{i=1}^{\ell+1} r^{(i)} \cdot \left\langle b^{(i)} \right\rangle$.

(e) The participants run MAC-Check $(c, \langle c \rangle)$.

(f) If any of the MAC-Check $(c, \langle c \rangle)$ fails, the corresponding party broadcasts $\perp$ and the protocol execution is aborted.

The verification process consists of two steps. First, the participants open and verify the commitments to the Shamir shares (Step 1-2b). Second, the participants recalculate the linear combination of the opened Shamir shares, and compare these to the earlier published Shamir shares (Step 2c). If these checks succeed, and there are sufficient 'non-cheating' Shamir shares, the remaining Shamir shares are used to reconstruct the secret values (Step 3).

Protocol 6 FairBlinds: Fair Opening of Blinding Values.

Inputs:

(1) The commitments over the Shamir shares $C^{(1)}, \ldots, C^{(n)}$.
(2) The opening data $O^{(1)}, \ldots, O^{(n)}$.
(3) The random values $r^{(1)}, \ldots, r^{(\ell+1)}$.
(4) The Shamir sharing of the random linear combination of blinding values $SH(c)$.

Output:

(1) Each party obtains the blinding values $b^{(1)}, \ldots, b^{(\ell)}$, or an abort message $\perp$.

The protocol

(1) **Opening Shamir sharings**
Party P_i sends opening $O^{(i)}$ to all other parties $P_j \neq P_i$.
(2) **Verifying Shamir shares**
(a) Each party initializes an empty local set of cheating participants $\Omega_i = \{\}$.
(b) Each party locally verifies whether the opening data is correct by running Open $\left(C^{(i)}, O^{(i)}\right)$, for all $1 \leq i \leq n$. If not, participant P_i is added to the set of cheating participants Ω_i.
(c) Each party locally determines whether $SH(c)_i = \sum_{j=1}^{\ell+1} r^{(j)} \cdot SH\left(b^{(j)}\right)_i$, for all participants $1 \leq i \leq n$. If not, participant P_i is added to the local set of cheating participants Ω_i.
(d) If $|\Omega_i| \geq n/2$ for party P_i, party P_i outputs $\perp$.
(3) **Opening blinding values**
If $|\Omega_i| < n/2$, P_i reconstructs the blinding values $b^{(1)}, \ldots, b^{(\ell)}$ using $b^{(\iota)} =$ EffReconstruct $\left(SH\left(b^{(\iota)}\right)_j\right)$, $j \notin \Omega_i$, for all $1 \leq \iota \leq \ell$.

3.3 Complexity of the Proposed Scheme

In this section the communication and computational complexities of the proposed scheme are analyzed. The difference in complexity compared to 'standard', non-fair SPDZ is discussed, in order to provide a thorough understanding of the overhead associated with guaranteeing fairness. Furthermore, a distinction is made between the complexities of the online phase and the preprocessing phase.

The complexities are formalized for the number of participants n, the reconstruction threshold $k = \lceil n/2 \rceil$, the number of output values ℓ, the prime p (recall that operations are performed on $\mathbb{F}_p$), the hash size m and the number of partially opened values q. The additional complexity introduced by our scheme does not depend on the depth of the circuit, number of inputs, nor the amount of consumed randomness. All complexities are formalized for a single participant.

Throughout this section, it is assumed that the participants have access to sufficiently many random SPDZ-sharings.[4]

3.3.1 Communication Complexity. The communication complexity is analyzed in terms of the number of communication rounds and the average number of bits that are sent by a participant. To simplify this analysis, only the case where the number of output values is larger than one will be discussed; furthermore, we also exclude the cost of the functionality $\mathcal{F}_{\text{Continue}}$ for simplicity.

The communication complexity of the 'standard', non-fair reveal phase is formalized in Table 3. The majority of the communication rounds are the result of performing the MAC-Check protocol (Step 1a and Step 2b). The majority of the number of sent bits is a result of partially opening the output values (Step 2a), when the number of output values ℓ is large.

Table 3: Communication complexity of the non-fair reveal phase (excluding $\mathcal{F}_{\text{Continue}}$).

Step	Rounds	Sent bits
Online:		
Protocol 2		
Step 1a	4	$2 \cdot (n-1) \cdot (\lceil \log_2(p) \rceil + 2 \cdot m)$
Step 2a	2	$\frac{1}{n} \cdot (2 \cdot (n-1) \cdot \ell \cdot \lceil \log_2(p) \rceil)$
Step 2b	4	$2 \cdot (n-1) \cdot (\lceil \log_2(p) \rceil + 2 \cdot m)$
Total	10	$(n-1)((2\ell/n+4)\lceil \log_2(p) \rceil + 8m)$

The communication complexity of the fair design is formalized in Table 4. The total increase in the number of communication rounds, compared to the 'standard', non-fair reveal phase, is primarily due to generating the validation data (Protocol 5). This protocol performs Broadcast once (Step 1b) and SecureOpen three times (Steps 2b, 3b and 3f), either directly or indirectly, all requiring two communication rounds. Besides this, our design only adds one communication round in Protocol 4, and two communication rounds in Protocol 6. The majority of the increase in sent bits originates from broadcasting $\ell + 1$ elements (Protocol 4, Step 1 and Protocol 6, Step 1), for a large number of output values.

When focusing on the online phase, the increase in communication complexity is relatively modest. The number of communication rounds increases from 10 to 12, while the number of sent bits increases from roughly $2/n \cdot \theta$ to $(1 + 2/n) \cdot \theta$, where $\theta = (n-1) \cdot \ell \cdot \lceil \log_2(p) \rceil$, for a large number of output values.

3.3.2 Computational Complexity. The computational complexity is analyzed in terms of the number of values to be hashed, random elements to be generated, and field operations to be performed. The different input and/or output sizes of such hashes and random elements are ignored, effectively assuming a constant cost. Moreover, for simplicity sake, only the case where the number of partially opened values and output values are larger than one is discussed, and we exclude the cost of the functionality $\mathcal{F}_{\text{Continue}}$.

[4] The complexities of generating random SPDZ-sharings are not analyzed, since they depend on the underlying preprocessing phase.

Table 4: Communication complexity of the proposed design (excluding $\mathcal{F}_{\text{CONTINUE}}$).

Step	Rounds	Sent bits
Preprocessing:		
Protocol 4		
Step 1	1	$(n-1)\cdot(\ell+1)\cdot\lceil\log_2(p)\rceil$
Protocol 5		
Step 1b	2	$2\cdot(n-1)\cdot m$
Step 2b	2	$(n-1)\cdot(\lceil\log_2(p)\rceil+2\cdot m)$
Step 3b	2	$(n-1)\cdot(\lceil\log_2(p)\rceil+2\cdot m)$
Step 3f	2	$(n-1)\cdot(\lceil\log_2(p)\rceil+2\cdot m)$
Total	9	$(n-1)\cdot((\ell+4)\cdot\lceil\log_2(p)\rceil+8m)$
Online:		
Protocol 3		
Step 1a	4	$2\cdot(n-1)\cdot(\lceil\log_2(p)\rceil+2\cdot m)$
Step 2b	2	$\frac{1}{n}\cdot(2\cdot(n-1)\cdot\ell\cdot\lceil\log_2(p)\rceil)$
Step 2c	4	$2\cdot(n-1)\cdot(\lceil\log_2(p)\rceil+2\cdot m)$
Protocol 6		
Step 1	2	$(n-1)\cdot((\ell+1)\cdot\lceil\log_2(p)\rceil+2\cdot m)$
Total	12	$(n-1)((2\ell/n+5)\lceil\log_2(p)\rceil+10m)$

The computational complexity of the non-fair reveal phase is shown in Table 5. Since both Step 1a and Step 2b are calls to the MAC-Check protocol, they result in a similar complexity. Within each call to the MAC-Check protocol, the SecureOpen protocol is run twice, requiring $2n$ hashes. The generated random values in these steps are used to calculate linear combinations over the partially opened values. The number of additions in Step 2a is multiplied by $1/n$, since only one of the n participants sums the opened output values, which are then redistributed.

In Table 6 the computational complexity of our scheme is formalized. The hashes in Protocol 5, Steps 2b, 3b, 3f, and Protocol 3, Steps 1a and 2c, are computed during calls to the SecureOpen protocol, either directly or indirectly. The hashes computed in Steps 1 and 2b are used in the Broadcast and Open functions respectively. The additional random elements are primarily generated for the random coefficients when transforming the SPDZ-sharings to Shamir sharings (Protocol 4, Step 1) and the random elements for calculating the random linear combinations of the secret values. When focusing on the online phase, the number of hashes increases from $4n$ to $5n$, while the number of random values remains the same.

The transformation of SPDZ-sharings to Shamir sharings is an intensive operation with regard to the number of field operations (Protocol 4, Step 1). Each of the $n\cdot(\ell+1)$ generated shares requires $k-1$ additions and multiplications. Moreover, the values $i^2,\dots,i^{k-1}$ need to be computed for $2\le i\le n$, requiring $(n-1)\cdot(k-2)$ multiplications. Later, in Step 2, each participant must sum n shares for each of the $\ell+1$ secret values, requiring $(n-1)\cdot(\ell+1)$ additions.

Another noteworthy operation is the Reconstruct function (Protocol 5, Step 3c). First, the k weighting terms $w^{(1)},\dots,w^{(k)}$ are calculated, each requiring $k-1$ subtractions, $k-2$ multiplications and one division. Second, the k^2 base coefficients $c_i^{(j)}$, $0\le i<k$, $1\le j\le k$, are calculated. To calculate all k possible values of z, $k+1$ subtractions are required. The first term requires $k-1$ multiplications, since z has k possible values. The last term can be efficiently computed by pre-computing the 2^k-k-1 (singletons and empty set excluded) multiplicative subsets of $x_1,\dots,x_k$, which can be realized in 2^k-k-1 multiplications. Adding the respective multiplicative subsets requires an additional 2^k-k-1 additions. To combine the three terms, each base coefficient requires two multiplications for a total of $2k^2$ multiplications.

Once the base coefficients have been calculated, any polynomial can be efficiently reconstructed from a Shamir sharing with $(k+1)\cdot(k-1)$ additions and k^2 multiplications. After the polynomial is interpolated, it is determined whether the other $\lfloor n/2\rfloor$ points also lie on this polynomial, requiring $k-1$ additions and k multiplications each, assuming that $i^2,\dots,i^{k-1}$, for $2\le i\le n$, have been stored.

In the online phase, each reconstruction only costs k multiplications and $k-1$ additions (Protocol 6, Step 3), since only the constant term needs to be reconstructed, and the base coefficients have already been pre-computed.

When $q\gg\ell$ and $q\gg n$, i.e., the number of operations that consume shared randomness is much greater than the number of outputs values and the number of participants,[5] both the fair and the non-fair reveal phase require roughly q random values, $2q$ additions and $2q$ multiplications.

4 PERFORMANCE AND IMPLEMENTATION

In this section we aim to analyze the overhead of guaranteeing fairness for SPDZ-based protocols by benchmarking, and comparing the fair and non-fair reveal phase. To provide an honest comparison between the two, both are implemented in Python.[6] Moreover, a practical function is evaluated using the SPDZ protocol, providing insight in the total overhead introduced by our scheme. The benchmarks are based on the existing *MobileNet V1 0.25_128* neural network implementation [16], where one participant inputs an image and another participant inputs the trained network. Our implementation uses the MP-SPDZ framework [30] with the LowGear preprocessing phase and TopGear zero-knowledge proof of knowledge.[7] Due to space constraints, only the results on the total performance overhead are presented here; for an in-depth analysis of the overhead of the reveal phase, the reader can refer to Appendix D.

All benchmarks are performed on a 6-core AMD Ryzen 5 3600 with 12 threads. Each participant is run in a separate docker container with access to two threads, providing consistent computing power for varying numbers of participants.[8] We define a LAN setting, where the traffic between the participants is unhindered, and the WAN setting, where all traffic is delayed by 50 ms and the bandwidth of the participants is restricted to 50 Mbit/s, as used before in [31, 32]. Finally, we differentiate between the running data of the offline phase, i.e. preprocessing phase, and the online phase. All data are the result of averaging a hundred protocol runs, unless stated otherwise.

To analyze the total overhead introduced by guaranteeing fairness, the *MobileNet V1 0.25_128* neural network is evaluated using

[5] Intuitively, most non-trivial function evaluations follow this expectation.
[6] https://gitlab.com/Bart.V/fairness-benchmarking
[7] https://gitlab.com/Bart.V/mp-spdz-container
[8] https://www.docker.com

Table 5: Required additional field operations for the original reveal phase (excluding $\mathcal{F}_{\text{CONTINUE}}$).

Position	Hashes	Random Value	Addition	Subtraction	Multiplication	Division
Online:						
Protocol 2						
Step 1a	$2 \cdot n$	$q+3$	$2 \cdot ((q-1)+(n-1))$	1	$2 \cdot t+1$	-
Step 2a	-	-	$1/n \cdot \ell \cdot (n-1)$	-	-	-
Step 2b	$2 \cdot n$	$\ell+3$	$2 \cdot ((\ell-1)+(n-1))$	1	$2 \cdot \ell+1$	-
Total	$4 \cdot n$	$q+\ell+6$	$2 \cdot q+2 \cdot \ell-4+(2+1/n+\ell) \cdot (n-1)$	2	$2 \cdot (q+\ell+1)$	-

Table 6: Required additional field operations for the proposed design (excluding $\mathcal{F}_{\text{CONTINUE}}$).

Position	Hashes	Random Value	Addition	Subtraction	Multiplication	Division
Preprocessing:						
Protocol 4	-	-	-	-	-	-
Step 1	-	$(k-1) \cdot (\ell+1)$	$n \cdot (\ell+1) \cdot (k-1)$	-	$n \cdot (\ell+1) \cdot (k-1)$ $+(n-1) \cdot (k-2)$	-
Step 2	-	-	$(n-1) \cdot (\ell+1)$	-	-	-
Protocol 5						
Step 1a	1	1	-	-	-	-
Step 1b	1	-	-	-	-	-
Step 2a	-	1	-	-	-	-
Step 2b	n	1	-	-	-	-
Step 2c	-	-	$n-1$	-	-	-
Step 2d	-	$\ell+1$	-	-	-	-
Step 3a	-	-	ℓ	-	$\ell+1$	-
Step 3b	n	1	-	-	-	-
Step 3c	-	-	$2^k+n \cdot k-n-2$	$k \cdot k+1$	$2^k+3k^2-(n-2)k-2$	k
Step 3d	-	-	$2 \cdot \ell$	-	$2 \cdot (\ell+1)$	-
Step 3f	n	1	$n-1$	1	1	-
Total	$3 \cdot n+2$	$k \cdot (\ell+1)+5$	$(n \cdot k-1) \cdot (\ell+1)+2^k$ $+3 \cdot \ell+(k+1) \cdot n-4$	$k \cdot k+2$	$(n \cdot k-n+3) \cdot (\ell+1)$ $+2^k+3k^2+k-2n+1$	k
Online:						
Protocol 3						
Step 1a	$2 \cdot n$	$q+3$	$2 \cdot ((q-1)+(n-1))$	1	$2 \cdot q+1$	-
Step 2a	-	-	$2 \cdot \ell$	-	-	-
Step 2b	-	-	$1/n \cdot \ell \cdot (n-1)$	-	-	-
Step 2c	$2 \cdot n$	$\ell+3$	$2 \cdot ((\ell-1)+(n-1))$	1	$2 \cdot \ell+1$	-
Step 3b	-	-	-	ℓ	-	-
Protocol 6						
Step 1	1	-	-	-	-	-
Step 2b	$n-1$	-	-	-	-	-
Step 2c	-	-	$(n-1) \cdot \ell$	-	$(n-1) \cdot (\ell+1)$	-
Step 3	-	-	$\ell \cdot (k-1)$	-	$\ell \cdot k$	-
Total	$5 \cdot n$	$q+\ell+6$	$\ell(k+n-1/n+3)+4n+2q-8$	$\ell+2$	$(n+k-1)\ell+2(q+\ell)+n+1$	-

both the standard and fair reveal phase, presented in Table 7. The standard run-time data are obtained by evaluating the neural network using the MP-SPDZ framework. The fair run-time data are equal to the standard run-time data, increased with the cost of generating the required number of random SPDZ-sharings for our scheme and the overhead introduced by fairly opening the output values. The run-time data of the offline phase are obtained from one function evaluation, due to its long running time.

Table 7 clearly indicates that the amount of sent data increases at a negligible rate ($< 0.005\%$), for both the offline and online phase and for varying numbers of participants. Moreover, the running time of the offline phase only increases marginally ($< 1\%$), primarily due to the cost of generating additional random SPDZ-sharings.

Table 7: Run-time data of evaluating the MobileNet V1 0.25_128 neural network, using the MP-SPDZ framework.

		Offline			Online		
	n	Standard	Fair	Inc.	Standard	Fair	Inc.
Sec	3	21163	21187	0.12%	35.76	35.77	0.02%
	4	29881	30157	0.92%	40.90	40.91	0.02%
Mb	3	364386	364400	0.00%	2389.91	2389.99	0.00%
	4	541103	541105	0.00%	2688.69	2688.81	0.00%

Most importantly, the latency of the online phase, which is one of the main advantages of SPDZ-based protocols, only increases at a negligible rate (0.025%).

5 FULL PROTOCOL AND SIMULATION-BASED SECURITY

In this section we provide the full protocol description (Protocol 7) and give the outline of the formal security guarantees that our protocol satisfies: security with selective abort against up to $n-1$ malicious participants, and fairness against $< n/2$ malicious participants. We provide a stand-alone simulation-based proof in the synchronous model, namely we show that our scheme is indistinguishable from the ideal functionality provided by $\mathcal{F}_{\text{Online}}$ (Functionality 1) assuming synchronous point-to-point communication channels.

In order to abstract out the underlying functionality provided by SPDZ, we introduce the functionality $\mathcal{F}_{\text{BlindOnline}}$ (Functionality 2). This functionality is a straightforward extension of the functionality implemented by SPDZ, which in the preprocessing phase provides parties with an additive secret sharing of the MAC value α, together with authenticated additive shares of random blinding values $b^{(1)},\dots,b^{(\ell)}$ (where malicious parties can determine their own shares). In the computation phase, the protocol then proceeds as SPDZ, but instead of outputting the output values $(y^{(1)},\dots,y^{(\ell)})$ directly, it outputs the blinded values $(z^{(1)},\dots,z^{(\ell)})$, where $z^{(j)} = y^{(j)} + b^{(j)}$, offering security with abort. Note that it is straightforward to obtain a secure implementation of this functionality with SPDZ, since the blinding values $b^{(1)},\dots,b^{(\ell)}$ can simply be viewed as additional (random) input to the circuit $C'(x_1,\dots,x_m;b^{(1)},\dots,b^{(\ell)}) = C(x_1,\dots,x_m) + (b^{(1)},\dots,b^{(\ell)})$.

With this, the major part of the online protocol (Protocol 7) is a procedure to fairly open the blinding values $b^{(1)},\dots,b^{(\ell)}$, given *authenticated* blinding values $b^{(1)},\dots,b^{(\ell+1)}$ (where the additional value $b^{(\ell+1)}$ is sacrificed in Protocol 5 to ensure that the conversion to Shamir shares was performed correctly).

In Appendix A.4 we prove the main security theorem.

Theorem 5.1. *If $COM = (\textsc{Setup}, \textsc{Com}, \textsc{Open}, \textsc{TCom}_1, \textsc{TCom}_2)$ is a trapdoor commitment scheme, $\mathbb{F} = \mathbb{F}_p$ is a super-polynomially sized field, and $\mathcal{U}_s(p, \ell+1)$ is a secure pseudorandom generator and assuming pairwise authenticated synchronous point-to-point channels between all parties, then Π_{Online} (Prot. 7) implements $\mathcal{F}_{\text{Online}}$ (Funct. 1) in the $(\mathcal{F}_{\text{Broadcast}}, \mathcal{F}_{\text{SecureOpen}}, \mathcal{F}_{\text{BlindOnline}})$-hybrid model with computational security against any static malicious adversary corrupting up to $n-1$ parties.*

Protocol 7 Π_{Online}.

Parameters. Let p be a super-polynomial prime and $\mathbb{F} = \mathbb{F}_p$ be the finite field of size p. Let C be an arithmetic circuit over $\mathbb{F}$ with m input gates $\text{id}_1,\dots,\text{id}_m$ and ℓ output gates. Let further each input gate be uniquely associated with a participant.

Preprocessing phase:

- Each party P_i inputs Preprocess to $\mathcal{F}_{\text{BlindOnline}}$, which outputs
$$\left(\alpha_i, \left(b_i^{(j)}, \gamma(b^{(j)})_i\right)_{j=1}^{\ell+1}\right)$$
to P_i for $i = 1,\dots,n$.
- The parties run the protocol ShareShamir (Prot. 4) with input $\langle b^{(1)}\rangle,\dots,\langle b^{(\ell+1)}\rangle$ to obtain shares $SH(\tilde{b}^{(1)}),\dots,SH(\tilde{b}^{(\ell+1)})$. //In the honest case $SH(\tilde{b}^{(i)})$ corresponds to a Shamir sharing of $b^{(i)}$.
- The parties run the protocol GenValData (Prot. 5) to obtain the following public values: the commitments $C^{(1)},\dots,C^{(n)}$, the opening data $O^{(1)},\dots,O^{(n)}$, the random values $r^{(1)},\dots,r^{(\ell+1)}$, and a Shamir sharing $SH(c)$. //In the honest case $C^{(i)}$ corresponds to a commitment to the i-th share of $SH(b^{(1)})\|\dots\|SH(b^{(\ell+1)})$ with opening information $O^{(i)}$, and $SH(c) = \sum_{i=1}^{\ell+1} r^{(i)} \cdot SH(b^{(i)})$.

Input phase:

- For each input x for input gate id_j hold by P_i, the party inputs (Input, P_i, id_j, x) to $\mathcal{F}_{\text{BlindOnline}}$.

Computation phase:

- Each party inputs Compute to $\mathcal{F}_{\text{BlindOnline}}$ and receives output $z = \left(z^{(1)},\dots,z^{(\ell)}\right)$ or $\perp$. //In the honest case it holds $z^{(i)} = y^{(i)} + b^{(i)}$, where $\left(y^{(1)},\dots,y^{(\ell)}\right) = C(x_1,\dots,x_m)$.
- The parties call functionality $\mathcal{F}_{\text{Continue}}$, where P_i inputs ok if it received the output in the previous step, and abort otherwise. If party P_i received $\perp$ in the last step or $\mathcal{F}_{\text{Continue}}$ outputs $\perp$ to P_i, the party aborts the protocol execution and outputs $\perp$. Otherwise, it continues to the next step.
- The parties run protocol FairBlinds (Prot. 6) with commitments $C^{(1)},\dots,C^{(n)}$, opening data $O^{(1)},\dots,O^{(n)}$, random values $r^{(1)},\dots,r^{(\ell+1)}$ and the Shamir sharing of linear combination of blinding values $SH(c)$ as input, and obtain blinding values $\tilde{b}^{(1)},\dots,\tilde{b}^{(\ell)}$ or $\perp$ as output. If the parties obtain $\perp$, the protocol aborts. //In the honest case the parties obtain $\tilde{b}^{(i)} = b^{(i)}$ for all $i \in [\ell]$.
- Each party sets $y^{(j)} = z^{(j)} - \tilde{b}^{(j)}$ for all $j = 1,\dots,\ell$ and outputs $\left(y^{(1)},\dots,y^{(\ell)}\right)$.

Functionality 1 $\mathcal{F}_{\text{Online}}$.

Parameters. Let p be a super-polynomial prime and $\mathbb{F} = \mathbb{F}_p$ be the finite field of size p. Let C be an arithmetic circuit over $\mathbb{F}$ with m input gates $\text{id}_1, \ldots, \text{id}_m$ and ℓ output gates. Let further each input gate id_j be uniquely associated with a party P_i.
Corrupted parties. Let $\mathcal{A}$ denote the set of parties corrupted by the adversary.
Input phase: On input $(\text{Input}, P_i, \text{id}_j, x)$ from party P_i (potentially controlled by the adversary) proceed as follows:

(1) If P_i is not associated with input gate id_j, if a value x for input gate id_j has already been registered, if $x \notin \mathbb{F}$, or if the query has the wrong format, ignore the input.
(2) Else, register $x^{(j)} = x$ as the input value corresponding to gate id_j.

Computation phase: Initialize $S := \emptyset$. On input Compute from party P_i, set $S := S \cup \{P_i\}$. If $S = \{P_1, \ldots, P_n\}$, proceed as follows:

(1) If no input has been registered for an input gate of C, return $\perp$ to all parties and to the adversary and exit.
(2) Else, compute $\left(y^{(1)}, \ldots, y^{(\ell)}\right) = C\left(x^{(1)}, \ldots, x^{(m)}\right)$, where the inputs $x^{(1)}, \ldots, x^{(m)}$ are the values registered (i.e., associated to the input gates) during the input phase.
(3) If $|\mathcal{A}| < n/2$ send DeliverMessage to the adversary and receive ToAll or ToNone message. If ToAll, communicate $y^{(1)}, \ldots, y^{(\ell)}$ to all honest parties and to the adversary. If ToNone, communicate $\perp$ to all honest parties and to the adversary.
(4) If $|\mathcal{A}| \geq n/2$ send $(\text{DeliverMessage}, y^{(1)}, \ldots, y^{(\ell)})$ to the adversary and receive a set (ToSet, S) for $S \subseteq \{P_1, \ldots, P_n\} \backslash \mathcal{A}$. Communicate $y^{(1)}, \ldots, y^{(\ell)}$ to all parties in S and $\perp$ to all other parties.

By default, the functionality ignores any unexpected input, and aborts sending $\perp$ to all parties if the adversary fails to respond on any of the queries.

Functionality 2 $\mathcal{F}_{\text{BlindOnline}}$.

Parameters. Let p be a super-polynomial prime and $\mathbb{F} = \mathbb{F}_p$ be the finite field of size p. Let C be an arithmetic circuit over $\mathbb{F}$ with m input gates $\text{id}_1, \ldots, \text{id}_m$ and ℓ output gates. Let further each input gate id_j be uniquely associated with a party P_i.
Corrupted parties. Let $\mathcal{A}$ denote the set of parties corrupted by the adversary.
Preprocessing phase: Initialize $S := \emptyset$. On input Preprocess from party P_i, set $S := S \cup \{P_i\}$. If $S = \{P_1, \ldots, P_n\}$, proceed as follows: The functionality sets up a MAC key α and blinding values $b^{(1)}, \ldots, b^{(\ell+1)}$ as follows:

(1) for each $P_i \in \mathcal{A}$, send (Mac, P_i) to the adversary and receive input $\alpha_i \in \mathbb{F}$;
(2) for each $P_i \notin \mathcal{A}$, sample $\alpha_i \in \mathbb{F}$ uniformly at random;
(3) for each $P_i \in \mathcal{A}$, send $(\text{BlindingShares}, P_i)$ to the adversary to receive $b_i^{(1)}, \ldots, b_i^{(\ell+1)} \in \mathbb{F}$;
(4) for each $P_i \notin \mathcal{A}$, sample $b_i^{(1)}, \ldots, b_i^{(\ell+1)} \in \mathbb{F}$ uniformly at random;
(5) define the MAC key α as $\alpha = \sum_{i=1}^n \alpha_i$;
(6) define the blinding value $b^{(j)}$ as $b^{(j)} = \sum_{i=1}^n b_i^{(j)}$ for $j = 1, \ldots, \ell + 1$;
(7) generate random shares $\gamma(b^{(j)})_1, \ldots, \gamma(b^{(j)})_n$ conditioned on $\sum_{i=1}^n \gamma(b^{(j)})_i = \alpha \cdot b^{(j)}$ for $j = 1, \ldots, \ell + 1$;
(8) output $\left(\alpha_i, \left(b_i^{(j)}, \gamma(b^{(j)})_i\right)_{j=1}^{\ell+1}\right)$ to P_i for $i = 1, \ldots, n$.

Input phase: On input $(\text{Input}, P_i, \text{id}_j, x)$ from party P_i (potentially controlled by the adversary) proceed as follows:

(1) If P_i is not associated with input gate id_j, if a value x for input gate id_j has already been registered, if $x \notin \mathbb{F}$, or if the query has the wrong format, ignore the input.
(2) Else, register $x^{(j)} = x$.

Computation phase: Initialize $S := \emptyset$. On input Compute from party P_i, set $S := S \cup \{P_i\}$. If $S = \{P_1, \ldots, P_n\}$, proceed as follows:

(1) If no input has been registered for an input gate of C, return $\perp$ to all parties and to the adversary and exit.
(2) Else, compute:
   - $\left(y^{(1)}, \ldots, y^{(\ell)}\right) = C\left(x^{(1)}, \ldots, x^{(m)}\right)$, where the inputs $x^{(i)}$ are the values registered during the input phase;
   - $z^{(j)} = y^{(j)} + b^{(j)}$ for all $j = 1, \ldots, \ell$.
(3) Send $(\text{DeliverMessage}, z^{(1)}, \ldots, z^{(\ell)})$ to the adversary and receive a message (ToSet, S) for $S \subseteq \{P_1, \ldots, P_n\} \backslash \mathcal{A}$. Communicate $z^{(1)}, \ldots, z^{(\ell)}$ to all parties in S and $\perp$ to all other parties.

By default, the functionality ignores any unexpected input, and aborts sending $\perp$ to all parties if the adversary fails to respond on any of the queries.

6 CONCLUSION AND FUTURE WORK

We introduced an adaptation of SPDZ and similar MPC platforms to overcome the lack of fairness, where we additionally blinded the MPC outputs, and fairly revealed the blinding values.

We implemented the adaptation and benchmarked practical performance, to find that the run-time for guaranteeing fairness takes less than one percent in the preprocessing phase, and roughly 0.02% in the online phase. W.r.t. communication, the increase in the amount of sent data is less than 0.005% for both the preprocessing and online phase. Furthermore, the additional work is less than linear in the number of parties and in the number of output values.

To conclude, we found a way to make SPDZ, and implementations thereof, such as MP-SPDZ and SCALE-MAMBA, fair at a negligible cost. For future work, other MPC platforms building on authenticated secret sharing could similarly be made fair, and the scheme might be extended with cheater detection by leveraging the formalized method to detect invalid Shamir shares. Moreover, although we only provide a security proof in the stand-alone model, this does not directly imply that our solution becomes insecure when concurrently executed: we actually expect the resulting protocol to be secure, although a dedicated security proof would be needed. An interesting direction for future work would be providing such a proof.

ACKNOWLEDGMENTS

This paper is based on the Master Thesis "Making SPDZ Fair" by Bart Veldhuizen (2020) as a requirement for the master Computing Science at the Radboud University, Nijmegen, NL. All research related to this thesis was performed at the Cyber Security and Robustness department, TNO, The Hague, NL.

This work was further supported by the TNO Early Research Programme "Next Generation Cryptography".

The work of L. Kohl was carried out in the CWI Cryptology group, Amsterdam, supported by the NWO Talent Programme Veni (VI.Veni.222.348) and the NWO Gravitation Project QSC.

REFERENCES

[1] Abdelrahaman Aly, Karl Cong, Daniele Cozzo, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Oliver Scherer, Peter Scholl, Nigel P Smart, Titouan Tanguy, et al. 2021. SCALE–MAMBA v1. 14: Documentation. https://homes.esat.kuleuven.be/~nsmart/SCALE/.

[2] Gilad Asharov, Yehuda Lindell, and Hila Zarosim. 2013. Fair and Efficient Secure Multiparty Computation with Reputation Systems. In *Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 8270)*, Kazue Sako and Palash Sarkar (Eds.). Springer, 201–220. https://doi.org/10.1007/978-3-642-42045-0_11

[3] Carsten Baum, Daniele Cozzo, and Nigel P. Smart. 2019. Using TopGear in Overdrive: A More Efficient ZKPoK for SPDZ. In *Selected Areas in Cryptography - SAC 2019 - 26th International Conference, Waterloo, ON, Canada, August 12-16, 2019, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11959)*, Kenneth G. Paterson and Douglas Stebila (Eds.). Springer, 274–302. https://doi.org/10.1007/978-3-030-38471-5_12

[4] Carsten Baum, Bernardo David, and Rafael Dowsley. 2018. Insured MPC: Efficient Secure Multiparty Computation with Punishable Abort. *IACR Cryptology ePrint Archive* 2018 (2018), 942. https://eprint.iacr.org/2018/942

[5] Carsten Baum, Emmanuela Orsini, and Peter Scholl. 2016. Efficient Secure Multiparty Computation with Identifiable Abort. In *Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9985)*, Martin Hirt and Adam D. Smith (Eds.). 461–490. https://doi.org/10.1007/978-3-662-53641-4_18

[6] Donald Beaver and Shafi Goldwasser. 1989. Multiparty Computation with Faulty Majority, See [8], 589–590. https://doi.org/10.1007/0-387-34805-0_51

[7] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. 2009. Secure Multiparty Computation Goes Live. In *Financial Cryptography and Data Security, 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5628)*, Roger Dingledine and Philippe Golle (Eds.). Springer, 325–343. https://doi.org/10.1007/978-3-642-03549-4_20

[8] Gilles Brassard (Ed.). 1990. *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*. Lecture Notes in Computer Science, Vol. 435. Springer. https://doi.org/10.1007/0-387-34805-0

[9] Anne Broadbent, Stacey Jeffery, Samuel Ranellucci, and Alain Tapp. 2012. Trading Robustness for Correctness and Privacy in Certain Multiparty Computations, beyond an Honest Majority. In *Information Theoretic Security - 6th International Conference, ICITS 2012, Montreal, QC, Canada, August 15-17, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7412)*, Adam D. Smith (Ed.). Springer, 14–36. https://doi.org/10.1007/978-3-642-32284-6_2

[10] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 136–145.

[11] David Chaum. 1989. The Spymasters Double-Agent Problem: Multiparty Computations Secure Unconditionally from Minorities and Cryptographically from Majorities. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings (Lecture Notes in Computer Science, Vol. 435)*, Gilles Brassard (Ed.). Springer, 591–602. https://doi.org/10.1007/0-387-34805-0_52

[12] David Chaum. 1989. The Spymasters Double-Agent Problem: Multiparty Computations Secure Unconditionally from Minorities and Cryptographically from Majorities, See [8], 591–602. https://doi.org/10.1007/0-387-34805-0_52

[13] Arka Rai Choudhuri, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers. 2017. Fairness in an Unfair World: Fair Multiparty Computation from Public Bulletin Boards. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 719–728. https://doi.org/10.1145/3133956.3134092

[14] Richard Cleve. 1986. Limits on the Security of Coin Flips when Half the Processors Are Faulty (Extended Abstract). In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, Juris Hartmanis (Ed.). ACM, 364–369. https://doi.org/10.1145/12130.12168

[15] Robert K. Cunningham, Benjamin Fuller, and Sophia Yakoubov. 2017. Catching MPC Cheaters: Identification and Openability. In *Information Theoretic Security - 10th International Conference, ICITS 2017, Hong Kong, China, November 29 - December 2, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10681)*, Junji Shikata (Ed.). Springer, 110–134. https://doi.org/10.1007/978-3-319-72089-0_7

[16] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. 2020. Secure Evaluation of Quantized Neural Networks. *Proc. Priv. Enhancing Technol.* 2020, 4 (2020), 355–375. https://doi.org/10.2478/POPETS-2020-0077

[17] Ivan Damgård. 1995. Practical and Provably Secure Release of a Secret and Exchange of Signatures. *J. Cryptology* 8, 4 (1995), 201–222. https://doi.org/10.1007/BF00191356

[18] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. 2016. Confidential Benchmarking Based on Multiparty Computation. In *Financial Cryptography and Data Security - 20th International Conference, FC 2016, Christ Church, Barbados, February 22-26, 2016, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 9603)*, Jens Grossklags and Bart Preneel (Eds.). Springer, 169–187. https://doi.org/10.1007/978-3-662-54970-4_10

[19] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. 2013. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8134)*, Jason Crampton, Sushil Jajodia, and Keith Mayes (Eds.). Springer, 1–18. https://doi.org/10.1007/978-3-642-40203-6_1

[20] Ivan Damgård, Claudio Orlandi, and Mark Simkin. 2018. Yet Another Compiler for Active Security or: Efficient MPC Over Arbitrary Rings. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10992)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, 799–829. https://doi.org/10.1007/978-3-319-96881-0_27

[21] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7417)*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Springer, 643–662. https://doi.org/10.1007/978-3-642-32009-5_38

[22] Matthias Fitzi, Daniel Gottesman, Martin Hirt, Thomas Holenstein, and Adam Smith. 2002. Detectable Byzantine agreement secure against faulty majorities. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*. 118–126.

[23] Matthias Fitzi, Thomas Holenstein, and Jürg Wullschleger. 2004. Multi-party Computation with Hybrid Security. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3027)*, Christian Cachin and Jan Camenisch (Eds.). Springer, 419–438. https://doi.org/10.1007/978-3-540-24676-3_25

[24] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, Alfred V. Aho (Ed.). ACM, 218–229. https://doi.org/10.1145/28395.28420

[25] Lein Harn, Changlu Lin, and Yong Li. 2015. Fair secret reconstruction in (t, n) secret sharing. *J. Inf. Sec. Appl.* 23 (2015), 1–7. https://doi.org/10.1016/j.jisa.2015.07.001

[26] Martin Hirt, Christoph Lucas, and Ueli Maurer. 2013. A Dynamic Tradeoff between Active and Passive Corruptions in Secure Multi-Party Computation. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II (Lecture Notes in Computer Science, Vol. 8043)*, Ran Canetti and Juan A. Garay (Eds.). Springer, 203–219. https://doi.org/10.1007/978-3-642-40084-1_12

[27] Yuval Ishai, Eyal Kushilevitz, Yehuda Lindell, and Erez Petrank. 2006. On Combining Privacy with Guaranteed Output Delivery in Secure Multiparty Computation. In *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4117)*, Cynthia Dwork (Ed.). Springer, 483–500. https://doi.org/10.1007/11818175_29

[28] Somesh Jha, Louis Kruger, and Vitaly Shmatikov. 2008. Towards Practical Privacy for Genomic Computation. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*. IEEE Computer Society, 216–230. https://doi.org/10.1109/SP.2008.34

[29] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. 2013. Universally Composable Synchronous Computation. In *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7785)*, Amit Sahai (Ed.). Springer, 477–498. https://doi.org/10.1007/978-3-642-36594-2_27

[30] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 1575–1590. https://doi.org/10.1145/3372297.3417872

[31] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 830–842. https://doi.org/10.1145/2976749.2978357

[32] Marcel Keller, Valerio Pastro, and Dragos Rotaru. 2018. Overdrive: Making SPDZ Great Again. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III (Lecture Notes in Computer Science, Vol. 10822)*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer, 158–189. https://doi.org/10.1007/978-3-319-78372-7_6

[33] Ranjit Kumaresan and Iddo Bentov. 2014. How to Use Bitcoin to Incentivize Correct Computations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, 30–41. https://doi.org/10.1145/2660267.2660380

[34] Hung-Yu Lin and Lein Harn. 1995. Fair Reconstruction of a Secret. *Inf. Process. Lett.* 55, 1 (1995), 45–47. https://doi.org/10.1016/0020-0190(95)00045-E

[35] Yehuda Lindell and Ariel Nof. 2017. A Framework for Constructing Fast MPC over Arithmetic Circuits with Malicious Adversaries and an Honest-Majority. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3133956.3133999

[36] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. 1996. *Handbook of Applied Cryptography*. CRC Press. https://doi.org/10.1201/9781439821916

[37] Peter Sebastian Nordholt and Meilof Veeningen. 2018. Minimising Communication in Honest-Majority MPC by Batchwise Multiplication Verification. In *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10892)*, Bart Preneel and Frederik Vercauteren (Eds.). Springer, 321–339. https://doi.org/10.1007/978-3-319-93387-0_17

[38] Emmanuela Orsini. 2021. Efficient, Actively Secure MPC with a Dishonest Majority: A Survey. In *Arithmetic of Finite Fields*, Jean Claude Bajard and Alev Topuzoğlu (Eds.). Springer International Publishing, Cham, 42–71.

[39] Torben Pryds Pedersen. 1992. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *Advances in Cryptology — CRYPTO '91*, Joan Feigenbaum (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 129–140.

[40] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (1979), 612–613. https://doi.org/10.1145/359168.359176

[41] Gabriele Spini and Serge Fehr. 2016. Cheater Detection in SPDZ Multiparty Computation. In *Information Theoretic Security - 9th International Conference, ICITS 2016, Tacoma, WA, USA, August 9-12, 2016, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10015)*, Anderson C. A. Nascimento and Paulo S. L. M. Barreto (Eds.). 151–176. https://doi.org/10.1007/978-3-319-49175-2_8

[42] Andrew Chi-Chih Yao. 1982. Protocols for Secure Computations (Extended Abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*. IEEE Computer Society, 160–164. https://doi.org/10.1109/SFCS.1982.38

[43] Chuan Zhao, Shengnan Zhao, Minghao Zhao, Zhenxiang Chen, Chong-Zhi Gao, Hongwei Li, and Yu-an Tan. 2019. Secure Multi-Party Computation: Theory, practice and applications. *Inf. Sci.* 476 (2019), 357–372. https://doi.org/10.1016/j.ins.2018.10.024

A SECURITY PROOF

In this section we provide the simulators and formal proofs of security. The proof that Π_{Online} correctly implements $\mathcal{F}_{\text{Online}}$ in the $(\mathcal{F}_{\text{Broadcast}}, \mathcal{F}_{\text{SecureOpen}}, \mathcal{F}_{\text{BlindOnline}})$-hybrid model is obtained by constructing two simulators: one for the honest-majority setting (Simulator $\mathcal{S}_{\text{Online-HonestMaj}}$, Section A.1), where fairness can be achieved, and one for the dishonest-majority setting (Simulator $\mathcal{S}_{\text{Online-General}}$, Section A.2), where no such guarantee can be given. The simulators follow a relatively standard approach, where a simulated version of the protocol is run with the input of dishonest parties and with dummy inputs for honest parties (in this case, 0) in order to obtain values with the same format as an execution of the real protocol.

In the honest-majority case, the simulator makes use of the trapdoor commitment scheme, which essentially allows the simulator to produce 'generic' commitments, and only later on decide on which values they will open. This is necessary, since the simulator has to commit to the blinding shares of the honest parties *before* knowing the real output provided by the functionality. The trapdoor commitment scheme together with the fact that the simulator has control over the majority of the Shamir shares in the honest-majority case allows the simulator to adapt shares and commitments to explain the real output before opening. For a formal definition and on a discussion on how to realize such a trapdoor commitment scheme, we refer to Section B.1 in the appendix.

In the dishonest majority case, on the other hand, the simulator obtains the real output of the functionality *before* having to commit to its own inputs, and can thus form Shamir shares and commitments using the correct output values. We start by giving the simulator $\mathcal{S}_{\text{Online-HonestMaj}}$ (Section A.1) for the honest-majority case and the simulator $\mathcal{S}_{\text{Online-General}}$ Section A.2 for the case that there is no honest majority. We then show that the MAC Check provides some consistence guarantee of the inputs of the adversary (Section A.3). Finally, we give the full proof of our protocol (Section A.4).

A.1 Simulator $\mathcal{S}_{\text{Online-HonestMaj}}$

Simulator 1 $\mathcal{S}_{\text{Online-HonestMaj}}$

We assume that the set $\mathcal{A}$ of corrupted parties has cardinality $< n/2$.

In order to simplify the notation, in the following we describe the simulator for $\ell = 1$, and write y, z instead of $y^{(1)}, z^{(1)}$, as well as $b, \hat{b}$ and $r, \hat{r}$ instead of $b^{(1)}, b^{(2)}$ and $r^{(1)}, r^{(2)}$ in the following. It is straightforward to extend the simulator to arbitrary ℓ.

Preprocessing phase:

(1) The simulator runs a copy of the preprocessing phase of the functionality Func. 2 ($\mathcal{F}_{\text{BlindOnline}}$) as a subroutine: for each $P_i \in \mathcal{A}$ it sends (Mac, P_i) and $(\text{BlindingShares}, P_i)$ to the adversary to receive MAC share $\alpha_i \in \mathbb{F}$ and shares of blinding values $b_i, \hat{b}_i \in \mathbb{F}$; and randomly samples the values $\alpha_j, b_j, b'_j \in \mathbb{F}$ for the simulated honest parties $P_j \notin \mathcal{A}$; next, it samples $\gamma(b)_i, \gamma(\hat{b})_i$ at random for all $i \in \mathcal{A}$ and outputs $\left(\alpha_i, (b_i, \gamma(b)_i), \left(\hat{b}_i, \gamma(\hat{b})_i\right)\right)$ to each $P_i \in \mathcal{A}$.
(2) The next step consists of the conversion of SPDZ-sharings to Shamir sharings, executed through Prot. 4 (ShareShamir). Once again, the simulator runs a copy of this protocol:
It computes random Shamir sharings $SH(b_j), SH(\hat{b}_j)$ for each $P_j \notin \mathcal{A}$ and sends the shares corresponding to dishonest parties to the adversary, and it waits for the Shamir shares for honest parties of some value $\tilde{b}_i$ from the adversary for every $P_i \in \mathcal{A}$.
(3) Subsequently, we move to Prot. 5 (GenValData). The simulator runs a copy of this protocol as well:

Recall that Prot. 5 takes as input the SPDZ and Shamir sharings of the blinding values $b := \sum_{i=1}^{n} b_i$, $\hat{b} := \sum_{i=1}^{m} \hat{b}_i$, produces commitments to the Shamir shares of these value, computes a random linear combination $SH(c)$ of the Shamir sharings, and checks that this value c satisfies the MAC-check with tags provided by the SPDZ-sharings of the values $b, \hat{b}$.

For the commitment-creation phase, instead of properly simulating the commitments of honest parties, the simulator calls $\text{TCom}_1(\text{TD})$ to produce a commitment $C^{(i)}$ and corresponding opening key $\text{OK}^{(i)}$ for each honest party P_i. No opening information is produced for the honest parties at this point.

The other steps are the commit and challenge phase are executed by simulating the behavior of honest parties and by waiting for the inputs of dishonest parties from the adversary.

We then focus on step 3, where the linear combination $SH(c)$ of the Shamir sharings is computed and checked for consistency with the SPDZ-sharings.

If at 3b the adversary submits invalid Shamir shares (i.e. such that the n shares of both honest and dishonest parties do not lie on a polynomial of degree at most $\lfloor n/2 \rfloor$), then the simulator executes 3c, sends ToNone to the ideal functionality and quits.

If the adversary submits valid, but incorrect shares that correspond to a value $\tilde{b} \neq b$ for any $P_i \in \mathcal{A}$ (which would be detected by a failed MAC check in step 3f in an honest execution), the simulator proceeds until conclusion of the simulated Prot. 5, then sends ToNone to the ideal functionality and quits.

Input phase:

(1) The simulator runs a local copy of the Input phase of Functionality $\mathcal{F}_{\text{BlindOnline}}$: for each input gate with input identifier id_j associated with a party $P_i \in \mathcal{A}$, on input $(\text{Input}, P_i, \text{id}_j, x)$ from $P_i \in \mathcal{A}$, it forwards $(\text{Input}, P_i, \text{id}_j, x)$ to $\mathcal{F}_{\text{Online}}$.

Computation phase:

(1) The simulator locally invokes the computation phase of Functionality $\mathcal{F}_{\text{BlindOnline}}$: on input Compute by party $P_i \in \mathcal{A}$, it forwards Compute to $\mathcal{F}_{\text{Online}}$. If the functionality sends $\perp$, forward $\perp$ to the adversary and exit.

(2) On input DeliverMessage from $\mathcal{F}_{\text{Online}}$, the simulator samples $z \leftarrow \mathbb{F}$ at random and forwards (DeliverMessage, z) to the adversary to receive (ToSet, S).

(3) Next, the simulator runs a local copy of $\mathcal{F}_{\text{Continue}}$ and sets $m_j = \text{ok}$ for all honest parties $P_j \in S$ and $m_j = \text{abort}$ for all honest parties $P_j \notin S$. If the adversary sends ToNone in $\mathcal{F}_{\text{Continue}}$, or any of the broadcast messages by the honest parties is abort, the simulator forwards ToNoneto the ideal functionality $\mathcal{F}_{\text{Online}}$ and quits.

(4) Else, the simulator sends ToAll to the ideal functionality and receives the output value y. The simulator then runs a local copy of Prot. 6 (FairBlinds), taking care to ensure that the simulated values are consistent with y and with the values previously sent to the adversary:

The first step consists in broadcasting the opening information for the commitments produced during the simulation of Prot. 5. Now the simulator cannot just broadcast the correct opening information, because the adversary would then reconstruct the masking value b and, therefore, obtain as output a random value $z - b$ instead of y. Instead, the simulator adjusts the shares of the honest players of b so that they instead reconstruct to a different value, and makes use of the trapdoor-functionality of the commitment scheme to produce valid opening information for these adjusted shares. More precisely, let $b' := z - y$. The simulator computes a Shamir sharing $SH(b')$ such that $SH(b')_i = SH(b)_i$ for all $P_i \in \mathcal{A}$, i.e., such that the shares corresponding to corrupt players are the same as before; since there are at most $\lceil n/2 \rceil - 1$ corrupted players and since the Shamir sharing is done with polynomials of degree up to $\lceil n/2 \rceil$, this is always possible. The simulator then sets the share of each honest player $P_j \notin \mathcal{A}$ to $SH(b')_j$; combined with the shares $SH(b)_i$ of corrupted players, this forms a valid Shamir sharing of $SH(b')$.

Moreover, the adversary also saw a linear combination $SH(c)_j = r \cdot SH(b)_j + \hat{r} \cdot SH(\hat{b})_j$ for every honest party P_j, where r and $\hat{r}$ are random values and $\hat{b}$ has been produced by Functionality 2 ($\mathcal{F}_{\text{BlindOnline}}$) together with b. The simulator then adjust each share $SH(\hat{b})_j$ to $SH(\hat{b}')_j := SH(\hat{b})_j - r/\hat{r} \cdot SH(b' - b)_j$; once again, together with the shares of dishonest players of $SH(\hat{b})$, this forms a valid Shamir sharing, and we have that

$$\begin{aligned} & r \cdot SH(b')_j + \hat{r} \cdot (SH(\hat{b}')_j) \\ =& r \cdot \left(SH(b)_j + SH(b' - b)_j\right) \\ & + \hat{r} \cdot \left(SH(\hat{b})_j - r/\hat{r} \cdot SH(b' - b)_j\right) \\ =& r \cdot SH(b)_j + \hat{r} \cdot SH(\hat{b})_j = SH(c)_j. \end{aligned}$$

As recalled above, the adversary also got a commitment $C^{(j)}$ to both $SH(b)_j$ and $SH(\hat{b})_j$ for each honest player P_j. Since this commitment was created by using the trapdoor-functionality of the commitment scheme, the simulator can call $\text{TCom}_2\left(C^{(j)}, \text{OK}^{(j)}, SH(b')_j \| SH(\hat{b}')_j\right)$ to obtain valid opening information $O^{(k)}$ for $C^{(k)}, SH(b')_j \| SH(\hat{b}')_j$. The simulator hence runs the commitment-opening phase with these adjusted values for each P_j; the adversary submits the opening values of corrupted parties.

If the opening information of any (dishonest) player is incorrect, that player is added to the list Ω of cheating participants. The same happens if values $SH(b)_i$ of some dishonest player P_i do not match their linear combination $SH(c)_i$. Finally, the simulator sends Ω to the adversary and quits.

A.2 Simulator $\mathcal{S}_{\text{Online-General}}$

Simulator 2 $\mathcal{S}_{\text{Online-General}}$

We now assume that the set $\mathcal{A}$ of corrupted parties has cardinality $\geq n/2$.

In order to simplify the notation, in the following we describe the simulator for $\ell = 1$, and write y, z instead of $y^{(1)}, z^{(1)}$, as well as $b, \hat{b}$ and $r, \hat{r}$ instead of $b^{(1)}, b^{(2)}$ and $r^{(1)}, r^{(2)}$ in the following. It is straightforward to extend the simulator to arbitrary ℓ.

Preprocessing phase: The simulator proceeds as in the honest majority case, but generates honest commitments during GenValData instead of trapdoor commitments:

(1) The simulator runs a copy of the preprocessing phase of the functionality Func. 2 ($\mathcal{F}_{\text{BlindOnline}}$) as a subroutine, where it follows the description of the functionality (i.e., waits for the inputs of the adversary, and adversary samples all values uniformly at random).
(2) The simulator simulates the behavior of the honest parties in ShareShamir (Prot. 4) by following the real protocol instructions.
(3) The simulator simulates the behavior of the honest parties in GenValData (Prot. 5) by following the real protocol instructions.
Again, if at 3b the adversary submits invalid Shamir shares (i.e. such that the n shares of both honest and dishonest parties do not lie on a polynomial of degree at most $\lfloor n/2 \rfloor$), then the simulator executes 3c, sends ToNone to the ideal functionality and quits.
If the adversary submits valid, but incorrect shares that correspond to a value $\tilde{b} \neq b$ for any $P_i \in \mathcal{A}$ (which would be detected by a failed MAC check in step 3f in an honest execution), the simulator proceeds until conclusion of the simulated Prot. 5, then sends ToNone to the ideal functionality and quits.

Input phase:

(1) The simulator runs a local copy of the Input phase of Functionality $\mathcal{F}_{\text{BlindOnline}}$: for each input gate with input identifier id_j associated with a party $P_i \in \mathcal{A}$, on input $(\text{Input}, P_i, \text{id}_j, x)$ from $P_i \in \mathcal{A}$, it forwards $(\text{Input}, P_i, \text{id}_j, x)$ to $\mathcal{F}_{\text{Online}}$.

Computation phase:

(1) The simulator locally invokes the computation phase of Functionality $\mathcal{F}_{\text{BlindOnline}}$: on input Compute by party $P_i \in \mathcal{A}$, it forwards Compute to $\mathcal{F}_{\text{Online}}$. If the functionality sends $\perp$, forward $\perp$ to the adversary and exit.
(2) On input $(\text{DeliverMessage}, y)$ from $\mathcal{F}_{\text{Online}}$, the simulator forwards $(\text{DeliverMessage}, y + b)$ to the adversary, and waits for message (ToSet, S) by the adversary.
(3) Next, the simulator runs a local copy of $\mathcal{F}_{\text{Continue}}$ and sets $m_j = \text{ok}$ for all honest parties $P_j \in S$ and $m_j = \text{abort}$ for all honest parties $P_j \notin S$. It forwards $(P_i, m_i)_{i \notin \mathcal{A}}$ to the adversary and receives $(\text{ToSet}, \hat{S})$. If $S \cap \hat{S} = \emptyset$, the simulator sends $(\text{ToSet}, \emptyset)$ to the ideal functionality $\mathcal{F}_{\text{Online}}$ and quits.
(4) Else, the simulator then invokes the protocol FairBlinds (Prot. 6), where it follows the protocol for the honest parties $P_i \in S \cap \hat{S}$. It sets $S' := \{P_i \mid P_i \in S \cap \hat{S} \wedge |\Omega_i| < n/2\}$, sends (ToSet, S') to the functionality $\mathcal{F}_{\text{Online}}$ and quits.

A.3 Security of the MAC Check

Before we proceed with stating and proving the security of our scheme, we need to address one last point: proving the security of the MAC-Check protocol in an explicit way.

Note that for the purpose of the proof we assume that $\mathcal{U}_s(p, \ell+1)$ as used in GenValData (Protocol 5) returns perfect randomness. We further assume that the trapdoor commitment scheme is implemented using a random oracle as described in Section B. Note that it is straightforward to generalize the following lemma to the more general case by adapting the upper bound on the success probability of the adversary accordingly.

Before stating the main lemma recall that if $\mathcal{H}_1$ is modeled as a random oracle with R possible outputs, then the probability that the adversary can find two distinct values x, x' such that $\mathcal{H}_1(x) = \mathcal{H}_1(x')$ (and therefore break the binding property of the commitment scheme) by querying the oracle ω times, is denoted by $P_{col}(\omega)$ and is equal to $1 - \frac{R!}{(R-\omega)! \cdot R^\omega}$ [36].

We now prove that the MAC-Check protocol is sound, i.e., that it rejects all incorrectly authenticated values. Notice that a similar proof can be found in the original SPDZ articles; however, we require an explicit formulation of the error probability, and thus state it here in a slightly different form that better matches our purposes.

Lemma A.1 (Soundness of MAC-Check). *Let COM be instantiated using a (programmable) random oracle (as outlined in Section B) and let $\mathcal{U}_s(p, \ell + 1)$ be modelled as a perfect randomness generator. Then, the MAC-Check protocol (Prot. 1) is sound, i.e. it rejects values $(\tilde{a}^{(1)}, \ldots, \tilde{a}^{(\ell)}) \neq (a^{(1)}, \ldots, a^{(\ell)})$, except with a maximum probability of $2 \cdot \max(1/p, P_{col}(\omega))$, where p is the field size and $P_{col}(\omega)$ is as defined above.*

Proof. For simplicity, we focus on the case where $\ell > 1$.

The goal of the adversary is to force the honest participants into accepting incorrect values $(\tilde{a}^{(1)}, \ldots, \tilde{a}^{(\ell)}) \neq (a^{(1)}, \ldots, a^{(\ell)})$. To accomplish this, the adversary must ensure that the corresponding linear combination $\tilde{a}$ of the incorrect values (Step 2a) is accepted. This only occurs when σ is equal to 0, where $\sigma = \gamma(a) - \alpha \cdot \tilde{a}$ (Step 3b). The adversary must thus add a correcting value δ to σ, resulting in $\sigma + \delta = \gamma(a) - \alpha \cdot \tilde{a} + \delta = \alpha \cdot (a - \tilde{a}) + \delta$, such that $\sigma + \delta = 0$. This only holds if $\delta = -\alpha \cdot (a - \tilde{a})$; notice that the value $a - \tilde{a}$ is known to the adversary, but the value α is not.

By inspecting the protocol, it can be seen that the adversary has three possibilities to achieve this.

(1) The linear combination $\tilde{a} = \sum_{i=1}^{\ell} r^{(i)} \cdot \tilde{a}^{(i)}$ of incorrect values is equal to the linear combination $a = \sum_{i=1}^{\ell} r^{(i)} \cdot a^{(i)}$ of correct values. In this case, since $\tilde{a} = a$, the adversary can simply set $\delta = 0$.
(2) The adversary guesses the MAC-key α and calculates $\delta = -\alpha \cdot (a - \tilde{a})$ accordingly.
(3) In the SecureOpen protocol (Step 3a), the adversary commits to its shares of σ, but opens a commitment to different shares such that $\sum_{i=1}^{n} \sigma = 0$ after the honest participants have published their shares of σ. In this case, there is no need for the adversary to calculate a δ as described above.

The success probability of the second attack is determined by the knowledge of the MAC-key α. Since it is assumed that α is secret,

the adversary can only guess the MAC-key. The probability that this guess is correct is equal to $1/p$.[9] The success probability of the third attack is upper bounded by the probability $P_{col}(\omega)$ of finding a collision that breaks the security of the commitment scheme. Notice that these attacks are mutually exclusive, since the adversary must choose between committing to a collision, or introducing a specific δ.

For what concerns the first attack, since both $a = \sum_{i=1}^{\ell} r^{(i)} \cdot a^{(i)}$ and $\tilde{a} = \sum_{i=1}^{\ell} r^{(i)} \cdot \tilde{a}^{(i)}$ are random elements, they have a $1/p$-probability of being equal. When $\tilde{a} = a$, the participants verify $\tilde{a}$ to be correct and incorrectly conclude that $(\tilde{a}^{(1)}, \dots, \tilde{a}^{(\ell)})$ are correct. The adversary can influence whether $\tilde{a} = a$ with the seed he presents (Step 1(a)ii). The adversary can increase the probability of opening a seed such that $\tilde{a} = a$, by committing to a collision, which he finds with a probability of $P_{col}(\omega)$. Since the adversary can either commit to a random seed or to a collision, the success probability of this attack is upper bounded by $\max(1/p, P_{col}(\omega))$. As the adversary can first try forcing $\tilde{a} = a$ before trying to force $\sigma = 0$, the MAC-Check protocol is sound, except with a probability of $2 \cdot \max(1/p, P_{col}(\omega))$. □

A.4 Main security theorem

We are now ready to restate and prove the main theorem that describes the security of our scheme. Again, note that it is straightforward to adapt the below proof for arbitrary trapdoor commitment schemes and standard pseudorandom generators, by replacing $P_{col}(\omega)$ by the collision probability and adding an additional term $\text{NEGL}_{\mathcal{U}}(\lambda)$, corresponding to the advantage of an adversary in distinguishing the PRG from random.

Theorem A.2. *If $COM = (\textsc{Setup}, \textsc{Com}, \textsc{Open}, \textsc{TCom}_1, \textsc{TCom}_2)$ is a trapdoor commitment scheme, $\mathbb{F} = \mathbb{F}_p$ is a super-polynomially sized field, and $\mathcal{U}_s(p, \ell + 1)$ is a secure pseudorandom generator and assuming pairwise authenticated synchronous point-to-point channels between all parties, then $\Pi_{\textsc{Online}}$ (Prot. 7) implements $\mathcal{F}_{\textsc{Online}}$ (Funct. 1) in the $(\mathcal{F}_{\textsc{Broadcast}}, \mathcal{F}_{\textsc{SecureOpen}}, \mathcal{F}_{\textsc{BlindOnline}})$-hybrid model with computational security against any static malicious adversary corrupting up to $n - 1$ parties, except with probability at most $3/p + P_{col}(\omega) + n/2 \cdot \text{NEGL}(\lambda)$.*

Proof. In the following we give the proof for $\ell = 1$ to simplify notation. It is straightforward to adapt the simulators and proof to arbitrary ℓ.

First, note that it is straightforward to see that the real protocol execution yields the correct output if no cheating occurs, as in this case all parties first receive $z = y + b$, where b is the blinding value generated in the preprocessing phase from $\mathcal{F}_{\text{BlindOnline}}$. During the execution of FairBlinds (Prot. 6), the parties then recover $\tilde{b} = b$ (in the honest case), and thus output $z - \tilde{b} = z - b = y$ as required.

We divide the remaining proof in two cases: the honest-majority case and the general one. We provide two simulators, namely $\mathcal{S}_{\text{Online-HonestMaj}}$ (Simulator 1) and $\mathcal{S}_{\text{Online-General}}$ (Simulator 2), for these two cases.

[9] Should the adversary guess the correct MAC-key α, soundness would then be broken for all subsequent calls to the MAC-Check protocol.

Honest-majority case. We run through and analyze the steps of Prot. 7 (Π_{Online}).

Preprocessing phase:

(1) First, both the ideal functionality and the ideal one with simulator invoke the preprocessing phase of Func. 2 ($\mathcal{F}_{\text{BlindOnline}}$). Since the simulator follows the functionality, both executions are perfectly indistinguishable.
(2) Subsequently, Prot. 4 (ShareShamir) is invoked. Since the adversary obtains less than $\lceil n/2 \rceil$ Shamir shares of $b_j, \hat{b}_j$, the two executions are perfectly indistinguishable from the values received by the adversary; hence the simulation is immediately seen to be perfectly indistinguishable in case no cheating takes place. Notice that the adversary has two possibilities of cheating here: it can produce valid, but incorrect shares (i.e. not reconstructing to the SPDZ shares), or it can produce invalid Shamir shares (i.e. not lying on a polynomial of degree $\lceil n/2 \rceil - 1$).
Both cases will be detected in Prot. 5 (GenValData), as argued below.
(3) Subsequently, both the real execution and the ideal functionality with simulator invoke Prot. 5 (GenValData). If the adversary produced valid, but incorrect shares in Prot. 4 (ShareShamir), these will reconstruct to a value different from c in step 3c, except with probability $1/p$ over the sampling of the random coefficients $r^{(i)}$. The two values being different, the MAC-check of step 3f will fail except with probability $\max(1/p, P_{col}(\omega))$ (cf. Theorem A.1).
If it produced invalid shares, the reconstruction will fail at step 3c and the overall protocol will abort.

Input phase.

(1) Both the real execution and the ideal functionality with simulator invoke the input phase of Func. 2 ($\mathcal{F}_{\text{BlindOnline}}$). Since this phase communicates no information on the input of honest parties to the adversary, the two executions remain indistinguishable.

Computation phase.

(1) The next step consists in invoking the computation phase of Func. 2 ($\mathcal{F}_{\text{BlindOnline}}$). The behavior of the two in case of missing input is exactly the same; if no abort occurs, the difference is that the simulator samples the output value z uniformly at random, instead of returning $z = y+b$ (as it does not know y at this point). However, only z is communicated at this point to the adversary, and since b is a uniformly random value, this version of z is indistinguishable from the one obtained in a real execution.
(2) Since the simulator perfectly simulates the behavior of $\mathcal{F}_{\text{Continue}}$, both steps are indistinguishable. It is crucial for the last step that at this point either *all* honest parties received the output z in the previous step, or unanimously aborted the protocol execution.
(3) The last step consists in the execution/simulation of Prot. 6 (FairBlinds). The crucial point here is that the adversary gets to see the real output y of the computation *and* the blinding value b', while it was given a random value z at the previous step; these three values therefore must be consistent with each other. This is realized by the simulator

adjusting the shares of b of honest players, and using the trapdoor-functionality of the commitment scheme to ensure that this is not detected by the adversary. Notice that the adversary also got to see a linear combination c of the blinding value b and of another random value $\hat{b}$, which is "sacrificed" (i.e., not used to blind output values), namely $c = r \cdot b + \hat{r} \cdot \hat{b}$. The simulator also accounts for this, but cannot do that in case $\hat{r} = 0$; this, however, only happens with probability $1/p$. The only way in which the adversary can cheat at this stage is by submitting incorrect opening information for the commitments to the shares of b. The real and the simulated execution handle this case in the same way, unless the adversary is able to provide valid opening information for a value which is not the one it had committed to; by the binding property of the commitment scheme, and given that the adversary can attempt this on the commitment of each dishonest party, this happens with probability at most $n/2 \cdot \text{NEGL}(\lambda)$.

Summarizing the above, we have that the two executions are indistinguishable, except with the following probability:

$$1/p + \max(1/p, P_{col}(\omega)) + 1/p + n/2 \cdot \text{NEGL}(\lambda)$$

$$\leq 3/p + P_{col}(\omega) + n/2 \cdot \text{NEGL}(\lambda).$$

General (dishonest-majority) case. Again, we run through and analyze the steps of Π_{ONLINE}.

Preprocessing phase.

(1) First, both the ideal functionality and the ideal one with simulator invoke the initialize and preprocessing phase of Func. 2 ($\mathcal{F}_{\text{BLINDONLINE}}$). Just as in the honest-majority case, since only random values are used as input in this phase, the two executions are indistinguishable.
(2) Subsequently, Prot. 4 (SHARESHAMIR) is invoked. Once again, only random values are used in this phase; hence the simulation is immediately seen to be correct in case no cheating takes place. Similar to the honest-majority case, cheating by the adversary in the ideal and in the simulated world are indistinguishable (except with negligible probability — see below), since they are handled in the same way in the subsequent step.
(3) Subsequently, both the real execution and the ideal functionality with simulator invoke Prot. 5 (GENVALDATA). Similar to the honest-majority case, cheating by the adversary in the ideal and in the simulated world are indistinguishable, except with probability $1/p + \max(1/p, P_{col}(\omega))$.

Input phase.

(1) Both the real execution and the ideal functionality with simulator invoke the input phase of Func. 2 ($\mathcal{F}_{\text{BLINDONLINE}}$). Again, since this phase communicates no information on the input of honest parties to the adversary, the two executions remain indistinguishable.

Computation phase.

(1) The next step consists of invoking the computation phase of Func. 2 ($\mathcal{F}_{\text{BLINDONLINE}}$). Both in the simulated and in the real protocol execution, the adversary receives (DELIVERMESSAGE, $z = y + b$). The simulation is thus perfectly indistinguishable from the real protocol execution.
(2) The last step consists of the execution/ simulation of Prot. 6 (FAIRBLINDS). As argued for the honest-majority case, the simulator is able to provide consistent values to the adversary; in this case, the simulator doesn't even need to adjust the committed values, since it can directly correct the masked value z. Note that the set $S' := \{P_i \mid P_i \in S \cap \hat{S} \wedge |\Omega_i| < n/2\}$ corresponds exactly to the honest parties which receive the output y, as all other parties abort at some previous step of the protocol. Since the simulator simulates the real protocol execution perfectly otherwise, we obtain the desired behavior, except with probability $n/2 \cdot \text{NEGL}(\lambda)$ given by the possibility of the adversary providing valid opening information for a value which is not the one it had committed to.

Summarizing the above, we have that the two executions are indistinguishable, except with the following probability:

$$1/p + \max(1/p, P_{col}(\omega)) + n/2 \cdot \text{NEGL}(\lambda)$$

$$\leq 2/p + P_{col}(\omega) + n/2 \cdot \text{NEGL}(\lambda).$$

□

B ADDITIONS

B.1 Trapdoor Commitment Scheme

In order to be able to prove simulation-based security in the honest majority case, the simulator must be able to correct the blinding values of the honest parties after committing to them. This is enabled by building on a trapdoor commitment scheme, as defined in the following.

Definition B.1 (Trapdoor commitment scheme). Let M be a set and $\text{COM} = (\text{SETUP}, \text{COM}, \text{OPEN}, \text{TCOM}_1, \text{TCOM}_2)$ be a tuple of PPT algorithms of the following form:

- SETUP on input 1^λ outputs public parameters PP and a trapdoor TD.
- COM on input of public parameters PP and a message m outputs a commitment C together with opening information O.
- OPEN is a deterministic algorithm that on input of public parameters PP, a commitment C, opening information O and message $m \in M$ outputs a bit $b \in \{0, 1\}$.
- TCOM_1 on input of the trapdoor TD outputs a commitment C and an opening key OK.
- TCOM_2 on input of a commitment C, an opening key OK and a message $m \in M$ outputs opening information O.

We say COM is a *trapdoor commitment scheme* for message space M, if the following properties hold:

- *Correctness:* For all $\lambda \in \mathbb{N}$, for all $m \in M$, for all $(\text{PP}, \text{TD}) \leftarrow \text{SETUP}(1^\lambda)$, for all $(C, O) \leftarrow \text{COM}_{\text{PP}}(m)$ it holds

$$\text{OPEN}_{\text{PP}}(C, O, m) = 1.$$

- *Perfect hiding:* For all $\lambda \in \mathbb{N}$ and for all adversaries $\mathcal{A}$, it holds:
$$\Pr\left[\mathcal{A}(\text{state}, C) = b \;\middle|\; \begin{array}{c}(\text{PP}, \text{TD}) \leftarrow \text{SETUP}(1^\lambda) \\ (m_0, m_1, \text{state}) \leftarrow \mathcal{A}(\text{PP}) \\ b \xleftarrow{\$} \{0,1\} \\ (C, O) \leftarrow \text{COM}_{\text{PP}}(m_b)\end{array}\right] = \frac{1}{2},$$
where we require that $\mathcal{A}$ returns valid messages $m_0, m_1 \in M$.
- *Computational binding:* For all PPT adversaries $\mathcal{A}$, there exists a negligible function $\text{NEGL}\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$, such that for all $\lambda \in \mathbb{N}$:
$$\Pr\left[\begin{array}{c}m_0 \neq m_1 \\ \wedge \text{OPEN}_{\text{PP}}(C, O_0, m_0) = 1 \\ \wedge \text{OPEN}_{\text{PP}}(C, O_1, m_1) = 1\end{array} \;\middle|\; \begin{array}{c}(\text{PP}, \text{TD}) \leftarrow \text{SETUP}(1^\lambda) \\ (C, \{O_b\}, \{m_b\}) \leftarrow \mathcal{A}(\text{PP})\end{array}\right] \leq \text{NEGL}(\lambda),$$
where we require that $\mathcal{A}$ returns valid messages $m_0, m_1 \in M$.
- *Perfect trapdoor:* For all $\lambda \in \mathbb{N}$ and for all adversaries $\mathcal{A}$ it holds:
$$\Pr\left[\mathcal{A}(\text{state}, C_b, O_b) = b \;\middle|\; \begin{array}{c}(\text{PP}, \text{TD}) \leftarrow \text{SETUP}(1^\lambda) \\ (m, \text{state}) \leftarrow \mathcal{A}(\text{PP}) \\ (C_0, O_0) \leftarrow \text{COM}_{\text{PP}}(m) \\ (C_1, \text{OK}) \leftarrow \text{TCOM}_1(\text{TD}) \\ O_1 \leftarrow \text{TCOM}_2(C_1, \text{OK}, m) \\ b \xleftarrow{\$} \{0,1\}\end{array}\right] = \frac{1}{2},$$
where we require that $\mathcal{A}$ returns valid messages $m_0, m_1 \in M$.

Note that one can instantiate the above trapdoor commitment schemes via Pedersen commitments [39]. Alternatively, one can obtain the required trapdoor commitment schemes in the (programmable) random oracle model: Let $H\colon \{0,1\}^* \to \{0,1\}^\lambda$ be modeled as a random oracle. Then, a commitment scheme can be defined as follows:

- COM on input of a message $m \in \{0,1\}^*$ draws $r \xleftarrow{\$} \{0,1\}^\lambda$, and outputs $C = H(m\|r)$ and $O = r$.
- OPEN on input of a message $m \in \{0,1\}$, a commitment C and opening information O, outputs $b = 1$ if and only if $C = H(m\|O)$.

Instead of a trapdoor, the trapdoor algorithm TCOM_2 gets the ability to program the random oracle:

- To generate a commitment, TCOM_1 chooses $C \xleftarrow{\$} \{0,1\}^\lambda$ at random.
- To produce opening information of the commitment C to a message $m \in \{0,1\}^*$, TCOM_2 chooses $O \xleftarrow{\$} \{0,1\}^\lambda$ at random, programs $H(m\|O) := C$ and outputs the opening information O[10]

C IDEAL FUNCTIONALITIES USED IN OUR CONSTRUCTION

This section provides a formal description of some of the building blocks of our construction: namely, the (weak) broadcast functionality, the simultaneous revealing of elements held by participants, and the agreement functionality used by participants in order to decide whether to open the blinding values or not.

We begin with the broadcast functionality (Funct. 3), which provides consistency, but with selective abort.

Functionality 3 $\mathcal{F}_{\text{BROADCAST}}$.

Corrupted parties. Let $\mathcal{A}$ denote the set of parties corrupted by the adversary.
Broadcast phase:

(1) For all $P_i \notin \mathcal{A}$ receive $(\text{BROADCAST}, m^{(i)})$ from party P_i.
(2) Send $(\text{BROADCAST}, (P_i, m^{(i)})_{i \notin \mathcal{A}})$ to the adversary and receive $(\text{TOSET}, S, (m^{(i)})_{i \in \mathcal{A}})$ for $S \subseteq \{P_1, \ldots, P_n\} \backslash \mathcal{A}$. Communicate $m^{(1)}, \ldots, m^{(\ell)}$ to all parties in S and $\perp$ to all other parties.

We then present the secure-opening functionality (Funct. 4), that allows participants to simultaneously reveal given values, hence preventing the adversary from adjusting their values based on the ones communicated by honest participants.

Functionality 4 $\mathcal{F}_{\text{SECUREOPEN}}$.

Corrupted parties. Let $\mathcal{A}$ denote the set of parties corrupted by the adversary.
Input phase:

(1) For $i = 1, \ldots, n$, receive $(\text{SECUREOPEN}, \xi^{(i)})$ from party P_i.
(2) Send $(\text{SECUREOPEN}, \xi^{(1)}, \ldots, \xi^{(n)})$ to the adversary and receive (TOSET, S) for $S \subseteq \{P_1, \ldots, P_n\} \backslash \mathcal{A}$. Communicate $\xi^{(1)}, \ldots, \xi^{(n)}$ to all parties in S and $\perp$ to all other parties.

Finally, we present the functionality (Funct. 5) that allows (honest) participants to agree on whether to proceed with a given step of the protocol (in our case, revealing the blinding values) or not. This functionality has to satisfy security with unanimous abort in the honest majority case, as otherwise the adversary can force some honest parties to abort, while it can use the information from the remaining parties to reconstruct the result, thereby violating fairness.

D IN-DEPTH ANALYSIS OF THE IMPLEMENTATION PERFORMANCE OF THE REVEAL PHASE

In Table 8 the running times for both the standard and fair reveal phases are presented in the LAN and WAN setting, where applicable. To more accurately analyze the overhead introduced by our scheme the computation verification is excluded from the online phase, as it is the same for both protocols.

The increase in running time is less than one-to-one with respect to the number of output values, since e.g. the run times for 1000

[10] Note that H is undefined on $m\|O$ except with negligible probability. If $H(m\|O)$ happens to be already defined, TCOM_2 can resample O until the random oracle is yet undefined.

Functionality 5 $\mathcal{F}_{\text{Continue}}$.

Corrupted parties. Let $\mathcal{A}$ denote the set of parties corrupted by the adversary.

Input phase:

(1) For all $P_i \notin \mathcal{A}$ receive $(\text{Continue}, m_i)$ for $m_i \in \{\text{ok}, \text{abort}\}$ from party P_i and send $(\text{Continue}, (P_i, m_i)_{i \notin \mathcal{A}})$ to the adversary.
(2) If $t < n/2$: Receive ToAll or ToNone from the adversary. If ToAll, and all messages are ok, send ok to all parties. Else, send $\perp$ to all parties.
(3) If $t \geq n/2$: Receive (ToSet, S) for $S \subseteq \{P_1, \ldots, P_n\} \backslash \mathcal{A}$. Communicate ok to all parties in S and $\perp$ to all other parties.

output values are less than double of the respective times for 500 output values.

The offline times are roughly equal to the corresponding online times for our fair design. This shows that the separation between the preprocessing and online phase significantly reduces the running time of the online phase. As a result, the online phase only requires a couple of milliseconds of overhead when fairly opening a low number of output values.

Our design is significantly faster in the LAN setting than the WAN setting, as expected. Since the offline and online phase require 9 and 8 rounds, excluding the computation verification, respectively, their running times increase with around 450ms and 400ms, respectively. The additional increase in run-time is due to the bandwidth restriction in the WAN setting. This additional delay increases for both a larger number of output values and participants, as both increase the required amount of sent data.

Table 8: Running times of the different reveal phases excluding the computation verification in milliseconds.

			LAN		WAN	
	n	ℓ	Offline	Online	Offline	Online
Fair	3	1	4.6	3.9	454.3	403.8
		500	11.2	11.8	466.7	425.0
		1000	15.8	18.5	478.7	447.2
	4	1	5.8	5.2	455.5	404.7
		500	12.9	13.7	471.6	433.2
		1000	19.2	21.7	487.5	462.7
	5	1	7.5	6.6	456.9	406.0
		500	17.0	16.2	478.5	441.5
		1000	25.4	26.5	498.9	479.1
Non-Fair	3	1	-	2.8	-	302.8
		500	-	6.7	-	317.1
		1000	-	10.5	-	330.1
	4	1	-	3.8	-	303.5
		500	-	8.0	-	321.2
		1000	-	12.1	-	338.0
	5	1	-	4.9	-	304.3
		500	-	9.2	-	325.6
		1000	-	14.0	-	345.9