MPC with Publicly Identifiable Abort from Pseudorandomness and Homomorphic Encryption

Marc Rivinius

Institute of Information Security, University of Stuttgart, Stuttgart, Germany
marc.rivinius@sec.uni-stuttgart.de

Abstract. Publicly identifiable abort is a critical feature for ensuring accountability in outsourced computations using secure multiparty computation (MPC). Despite its importance, no prior work has specifically addressed identifiable abort in the context of outsourced computations. In this paper, we present the first MPC protocol that supports publicly identifiable abort with minimal overhead for external clients. Our approach minimizes client-side computation by requiring only a few pseudorandom function evaluations per input. On the server side, the verification process involves lightweight linear function evaluations using homomorphic encryption. This results in verification times of a few nanoseconds per operation for servers, with client overhead being approximately two orders of magnitude lower. Additionally, the publicly verifiable nature of our protocol reduces client input/output costs compared to SPDZ-based protocols, on which we base our protocol. For example, in secure aggregation use cases, our protocol achieves over twice the efficiency during the offline phase and up to an 18 % speedup in the online phase, significantly outperforming SPDZ.

1 Introduction

Outsourced computations are of utmost importance in today's world of computing. This can easily be seen with current advances in machine learning (ML). Especially there, outsourced computation comes with many challenges, including for security and privacy. On the one hand, the increasing computational requirements – both in memory and compute resources – mean that only top-tier compute servers can handle them, necessitating outsourced computation. On the other hand, evaluating or training machine learning algorithms often requires access to sensitive information, such as personal data, confidential algorithms, or model weights. Simply sending this data obviously risks the privacy of the data. Additionally, any distributed computation raises the question of correctness: Does the other party really compute what they are supposed to? While secure multiparty computation (MPC) can be used in such settings to guarantee privacy of the inputs to the computation and correctness of the outputs, not all protocols proposed in the literature are ready to solve this in practice.

Firstly, additional security properties become relevant for outsourced computation, namely public verifiability [34,6] and identifiable abort [29,30]. *Public* *verifiability* allows parties not involved in performing the computation to get a guarantee that a computed output was correct. For outsourced computations in particular, the parties that give the inputs to the computation are not directly involved in computation [24,3] (from a classical MPC perspective), exemplifying the need for public verifiability. Additionally, even external parties can verify the computation, which can be relevant in high-stakes computations or if there was a dispute (e.g., in combination with identifiable abort described next). The other security property is *identifiable abort*. With this, the MPC protocol guarantees to find a responsible party if the computation has to abort, which happens if a party misbehaves, e.g., trying to influence the computation result or trying to break privacy. In combination with public verifiability, we get *publicly (verifiable) identifiable abort*, i.e., everyone can learn the identity of misbehaving parties. This is a strong deterrent against misbehavior because parties can be publicly identified and, for example, contractual punishments can be enforced. It, therefore, not only prevents denial-of-service attacks but also allows potential clients of outsourced computations to find trustable servers that did not misbehave in the past.

Secondly, not all MPC protocols – especially those that provide the above additional security guarantees – are ready for outsourced computation. Of the few protocols that support identifiable abort, even fewer provide publicly identifiable abort (see Sec. 1.1). As argued above, the publicly verifiable version of this property is much more useful for outsourced computation. Orthogonally, the desired efficiency gain from outsourced computation is only possible if a protocol supports lightweight clients and more capable servers. Clients are input/output parties whose amount of work should be independent of the computed function, while servers are compute parties that perform the actual computation. This is not supported in the standard paradigm for MPC protocols, where every party is involved and has to contribute to the whole computation. Therefore, we specifically need protocols that support clients for outsourced computation [3]. Moreover, it is usually not straightforward to extend protocols with identifiable abort to support such clients while still getting (publicly) identifiable abort for the whole protocol, because of the way cryptographic primitives are used for identifiable abort. Overall, we only identified one prior work that combines publicly identifiable abort with support for outsourced computations [42]. Our work improves on this by introducing a unique construction that enables publicly verifiable identifiable abort while supporting outsourced computation, while achieving performance comparable to non-identifiable MPC protocols.

Our protocol features a novel combination of homomorphic encryption (HE) and pseudorandom functions (PRFs) to achieve publicly identifiable abort. We use PRF-based message authentication codes (MACs) that can be publicly verified to check every operation done by the compute parties (i.e., servers). Input/output parties (i.e., clients) only have to check these lightweight MACs for private inputs/outputs. MAC tags are generated by the compute parties using HE. We construct homomorphically encrypted tags such that the same information to verify MAC tags – with only little extra information – is also used to verify the tag generation. This allows us to minimize the use of zero-knowledge proofs (ZKPs) and other expensive cryptographic primitives by verifying homomorphically encrypted messages at the same time as the MACs. By doing so, we shift some work that is normally done in a preprocessing phase to the verification phase. However, the overhead is relatively low and only relevant for the compute parties, not the clients.

We evaluate the practicality of our protocol by comparing it to standard (non-identifiable) actively secure MPC protocols (see Sec. 8) because prior protocols with identifiable abort lack an implementation. Our implementation[1] shows that the concrete overhead to achieve identifiable abort is relatively low. Notably, the overhead for verification is as low as a few nanoseconds per operation. Additionally, our focus on outsourced computations leads to faster protocols for inputs and outputs, allowing us to outperform even non-identifiable protocols. The main driver for this improvement is simpler correlated randomness that is used for input and outputs, where the public verifiability of our protocol avoids the need for more complex correlated randomness [24]. Next, we outline the current landscape of MPC protocols with identifiable abort and compare our protocol on a theoretical level. Section 8 contains our practical evaluation.

1.1 Related Work

While MPC with identifiable abort (IA) in its current form was first formally discussed by Ishai et al. [29,30], identifying misbehaving parties has already been part of protocols and security definitions for a long time, e.g., [28,4]. After these first mainly theoretical works, Baum et al. [8] showed that IA can be achieved efficiently. Efficiency of protocols with IA was further improved after this, with the goal of reducing the overhead compared to protocols without IA, i.e., protocols that only achieve security with abort. Our discussion in this section focuses on such efficient protocols, i.e., not on protocols that focus on feasibility or minimal requirements for IA [29,30,15,44], or minimizing the round complexity [1,20], without trying to improve concrete efficiency. We also do not discuss publicly verifiable protocols [6,31] in detail. Additionally, we focus on protocols that evaluate arithmetic circuits, as opposed to, e.g., identifiable abort for garbled circuits [10], due to systematical differences between the different types of protocols.

As all protocols with IA that we compare below are based on the SPDZ protocol [26] or its predecessor BeDOZa [12], we also include two SPDZ variants as baseline for the comparison: Overdrive LowGear [33] and TopGear [5]. These are optimized SPDZ-based protocols *without* identifiable abort (cf. Tab. 1). The overhead of protocols with IA can be better judged by comparing them to the non-identifiable protocols. Additionally, our protocol takes inspiration from both LowGear and TopGear for the offline phase, which is why we include these specific non-identifiable versions of SPDZ in the comparison. The other protocols in our comparison are Baum et al.'s aforementioned first efficient protocol with

[1] Our implementation is available online: https://github.com/sec-stuttgart/pia-mpc.

Table 1: Security Properties of Related Protocols

	SPDZ[a]	[8]	[45]	[23]	[21]	[7]	Ours
Secure with Abort	✓	✓	✓	✓	✓	✓	✓
Identifiable Abort	✗	✓	✓	✓	✓	✓	✓
Publicly Identifiable Abort	✗	[9]	✗[b]	✓	✓	✗	✓
Supports Outsourcing Computation	[24]	✗	✗	[42]	✗	✗	✓

✓: Protocol has this property ✗: Protocol does not have this property
[X]: Reference [X] describes modifications to achieve this property
[a] SPDZ-like protocols, e.g., [26,25,33,5] [b] might be without agreement on cheaters

IA (BOS [8]), the protocol of Spini and Fehr (SF [45]), the commitment-based protocol of Cunningham et al. (CFY [23]), the committed OT-based protocol by Cohen et al. (CDKS [21]), and the PCG [13]-based protocol by Baum et al. (BMRS [7]).

Security Table 1 describes the differences in security properties of these protocols. Of the protocols that support identifiable abort, only BOS [8], the commitment-based CFY [23], and CDKS [21] achieve *publicly identifiable abort* (PIA), i.e., they let the compute parties convince even external parties of the identity of malicious parties that caused an abort. As mentioned before, PIA can be essential, especially for outsourced computation. BOS [8] achieves PIA by (conceptually) running a second copy of the non-PIA protocol and publishing information from the duplicated protocol to convince external parties. This is a modification presented in [9]. CFY and CDKS achieve PIA with (publicly verifiable) homomorphic commitments and committed OT, respectively. Similarly to CFY, SF [45] can publicly identify cheaters by using encryptions from the offline phase as homomorphic commitments – however, only *in some cases* (depending on adversarial behavior). The adversary can avoid this public identification and instead cause a situation where the compute parties do not agree on the set of corrupted parties that are identified, making it unsuitable for outsourced computation where the external parties should learn the identity of cheaters and also the identity of all cheaters. Finally, BMRS [7] uses pairwise correlations between the compute parties, which does not straightforwardly support PIA.

Pairwise constructions like this are the main reason why many of the protocols [8,21,7] also do not support outsourced computation. Roughly speaking, these protocols do the following to make pairwise correlations publicly verifiable (if the respective protocol supports this [8,21]): If a party detects misbehavior and aborts, they reveal information that that party used to detect the misbehavior. Then, external parties can perform the same check. However, this cannot be easily done for external inputs or outputs, where the input/output parties always need to verify data but the protocol does not necessarily abort. Additionally, the information that would be sent to, e.g., the input parties would include the MAC keys and must stay secret for the remainder of the online phase.

Existing generic mechanisms like the one by Damgård et al. [24] could be used in all protocols, but this does not allow identification of a party that cheats towards the client: The client could only detect that some misbehavior happened, resulting in a protocol that is only secure with (non-identifiable) abort w.r.t. client inputs or outputs. The commitment-based protocol of Rivinius et al. (RRRK [42]) is a variant of CFY [23] that supports clients but via expensive HE and ZKPs. In contrast, our protocol is designed for PIA and lets clients publicly identify cheaters during the inputs or outputs without heavy cryptographic primitives.

Efficiency All protocols operate in multiple phases: (i) A setup phase that is independent of the computed function, (ii) an input-independent offline phase used to speed up the online phase, (iii) an online phase that operates on preprocessing material from the offline phase and the actual inputs, and (iv) a verification phase to assert correctness and identify malicious parties. Note that (iv) is sometimes also integrated into the online phase, e.g., in [21,7]. Tables 2 and 3 give an overview of the asymptotic communication and computation complexity of the protocols in these phases.[2] For communication complexity (Tab. 2), we analyze the asymptotic number of broadcasts and peer-to-peer (P2P) messages depending on the number of compute parties n and the size of the computed function (the number of multiplications M when represented as an arithmetic circuit). As multiple protocols, e.g., [8,23], describe ways to reduce costly broadcasts (especially in the online phase; by first sending P2P messages and later checking consistency), we count the number of *conceptual* broadcasts for the protocols to make them more comparable. Similar techniques to reduce the number of broadcasts could then be used in all protocols. E.g., our protocol could send $\mathrm{O}(Mn^2)$ P2P messages in the online phase instead of $\mathrm{O}(Mn)$ broadcasts and use techniques to check the consistency of the P2P messages at the end. For computation complexity (Tab. 3), we give the asymptotic number of operations performed by all parties depending on the number of parties n and the function size M. As outsourced computations are not supported by all protocols (cf. Tab. 1), we assume there are no external clients in our comparison. We compare the communication and computation efficiency of the protocols next.

Communication Complexity The baseline SPDZ-based protocols come in two flavors: Pairwise preprocessing protocols like Overdrive LowGear [33], which show practical efficiency for a low number of parties, and somewhat homomorphic encryption (SHE)-based protocols like TopGear [5], which have a better asymptotic complexity w.r.t. the number of parties n in the offline phase. The online complexity for both is $\mathrm{O}(Mn)$. BOS [8], CDKS [21], and BMRS [7] use

[2] [21] does not include a full description of their MPC protocol and the full version of the paper is not updated to reflect the Crypto paper at the time of writing. Therefore, we estimate the complexity based on the rough description in [21]. The PCGs in [7] can be initiated with different protocols. We assume a pairwise setup. Note that [21,7] have no verification phase as they eagerly verify.

Table 2: Communication Complexity of Related Protocols.[2] Complexity is given in O-notation depending on the number of compute parties n and circuit size M (number of multiplications).

Phase	Com.	[33]	[5]	[8]	[45]	[23]	[21]	[7]	Ours
Setup	P2P	n^2						n^2	
	BC		n	n^2	n	n	n		n
Offline	P2P	Mn^2	n					Mn^2	Mn^2
	BC	Mn	Mn	Mn^3	$Mn+n^2$	Mn	Mn^3	$Mn+n^2$	Mn
Online	P2P	n			Mn		Mn^2	Mn^2	
	BC	Mn	Mn	Mn^2	n^2	Mn			Mn
Verif.	P2P								
	BC	n	n	n^2	n	n			n

P2P: Peer-to-peer communication BC: Broadcast communication

Table 3: Computation Complexity of Related Protocols.[2] Complexity is given in O-notation depending on the number of compute parties n and circuit size M (number of multiplications).

Phase	[33]	[5]	[8]	[45]	[23]	[21]	[7]	Ours
Setup	n^2	n^2	n^3	n^2	n^2	n	n^2	n
Offline	Mn^2	Mn^2	Mn^4	Mn^2+n^3	Mn^2	Mn^3	Mn^2+n^3	Mn^2
Online	Mn^2	Mn^2	Mn^3	$Mn+n^3$	Mn^2	Mn^2	Mn^2	Mn^2
Verif.	$Mn+n^2$	$Mn+n^2$	Mn^2+n^3	n^2	$Mn+n^2$			Mn^2

pairwise MACs for their protocols. These have $O(Mn^2)$ complexity in the online phase. However, for BOS this is broadcast complexity and for the others only P2P. The offline complexity is $O(Mn^3)$, except for BMRS, which generates the pairwise correlations differently in the offline phase with $O(Mn^2)$ P2P messages and $O(Mn + n^2)$ broadcasts for verifying the correlations. SF [45] operates similarly to SPDZ but adds a verification step with $O(n^2)$ complexity both online and offline. This step is independent of the circuit size, and the remaining protocol complexity stays $O(Mn)$ as in SPDZ (online and offline). CFY [23] also models the protocol closely to SPDZ but generates homomorphic commitments that are used to identify malicious parties if the parties abort. This keeps the communication complexity similar to SPDZ but has an overhead for concrete instantiations (an overhead of around factor 4; cf. [42]). Our protocol combines parts of LowGear and TopGear in the offline phase, resulting in similar efficiency. Also, the online complexity is just like SPDZ, with different computations for verification. We discuss the computation complexity next.

Computation Complexity As shown in Tab. 3, the computation complexity is usually the same as the number of P2P messages and a factor of n larger than the broadcast complexity for all protocols in each phase. This is unsurprising as the parties have to use and check all incoming messages. Another relevant factor for practical protocol deployments is the type of cryptographic primitives used. SPDZ-based protocols [33,5] often use HE in the offline phase, as do [8,45,23]. CDKS [21] uses correlated OTs and BMRS [7] uses PCGs that avoid such public-key operations.[3] Furthermore, all protocols use cheap field operations in the online phase. SF [45] implements a multi-stage identification mechanism if misbehavior is detected. This uses HE also in the online phase if certain types of misbehavior are detected. CFY [23] uses homomorphic commitments to a similar effect, which has a significant impact on real-world performance – up to orders of magnitude overhead [42]. Our offline phase is based on LowGear and TopGear. Therefore, we also use HE. We only use cheap field operations in the online phase – and, importantly, for all client operations. In the verification phase, our protocol uses HE to verify some parts of the offline phase. However, compared to other protocols [45,23], we use HE independently of the circuit structure and can completely parallelize it. Therefore, the concrete overhead is very low (see Sec. 8).

1.2 Contributions

– We present the first MPC protocol with publicly identifiable abort and support for outsourced computation that is practically relevant for clients. While one existing protocol [42] technically supports clients, they do so by utilizing heavy cryptographic primitives (homomorphic encryption together with zero-knowledge proofs) on the client side. In our protocol, clients perform only a few cheap field operations and PRF evaluations for inputs and outputs.
– Our implementation[1] is one of the few available for identifiable protocols and the first to support outsourced computations with publicly identifiable abort.
– Our protocol uses novel verification techniques with little overhead in the online phase. This overhead for servers is as low as 19 ns (for 64 bit operations) or 34 ns (at 128 bit) per input and three times as much per multiplication. Verification time for clients is almost two orders of magnitude less at around 384 ps (64 bit) or 1 ns (128 bit) per input. Overall, our protocol is around 2 to 4 $\times$ slower than non-identifiable protocols when performing multiplications in the online phase, compared to previous protocols that are 4 to 18 $\times$ slower [23,42].
– Our protocol directly supports outsourced computation, unlike SPDZ [26] which uses Damgård et al.'s input/output subprotocols [24]. In use cases such as secure aggregation, our protocol can achieve 2.3 $\times$ faster preprocessing and 18 % faster online computation than this protocol without identifiable abort.

[3] State-of-the-art PCG-based protocols also have the prospect of requiring sublinear communication in the circuit size. However, [7] cannot benefit from this for improved offline complexity because the protocol adds checks that require communication linear in the circuit size.

2 Background

Before we present our protocol in Secs. 3 to 6, we discuss the required concepts used in SPDZ and related protocols. This includes secret-sharing, the general protocol structure, HE, and MACs. We use $\mathcal{P}$ for the set of compute parties.

2.1 Maliciously Secure Multiparty Computation: SPDZ

Secret-Sharing In our work, we use additive secret-sharing. For party $P_i \in \mathcal{P}$, let $[x]_i$ denote P_i's share of a secret x. We assume that $[x]_i$ and x are elements of a finite field $\mathbb{F}$. We use $\mathbb{F} = \mathbb{Z}_p \coloneqq \mathbb{Z}/p\mathbb{Z}$ in this work. Shares are chosen such that one can reconstruct the secret by summing up all $n = |\mathcal{P}|$ shares:

$$\mathsf{Rec}([x]) \coloneqq \mathsf{Rec}([x]_0, \ldots, [x]_{n-1}) \coloneqq \sum_{i=0}^{n-1} [x]_i = x. \tag{1}$$

When we talk about shared values in general, but not specific shares $[x]_i$, we use the notation $[x]$ as above in Eq. (1).

This secret-sharing scheme allows for reconstruction, only if all shares are known. Furthermore, we use the well-known fact that if all shares $[x]_i$ are selected uniformly at random, the overall secret is uniformly random as well. This is true, even if only one party chose their share uniformly at random (the other parties could be malicious and colluding), which is necessary in the security setting that we discuss.

The scheme is also linearly homomorphic:

$$[x+y]_i \coloneqq [x]_i + [y]_i, \qquad [c \cdot x]_i \coloneqq c \cdot [x]_i, \qquad [x+c]_i \coloneqq [x]_i + c \cdot \delta_i \tag{2}$$

for shared values x, y, and public constant $c \in \mathbb{F}$ (with δ_i being the Kronecker delta). This allows parties to perform linear operations on shares non-interactively.

The SPDZ Protocol SPDZ [26] is one of the first maliciously secure MPC protocols with practical efficiency. Many later protocols follow its general protocol structure and improve upon it (in efficiency or by adding additional capabilities). As mentioned before, our protocol and all protocols we compare to in Sec. 1.1 are based on SPDZ. The general structure is based on the above secret-sharing scheme and works as follows: Parties (i) transform secret inputs to shares, (ii) perform operations on shares – linear operations as in Eq. (2) and multiplication with so-called Beaver multiplication (cf. Sec. 4.2), (iii) publish the shares of the output, and (iv) verify that the operations on shares were done correctly. Like this, arbitrary functions (representable as arithmetic circuits) can be computed.

For inputs, multiplications, and outputs (steps (i) to (iii)), auxiliary data in the form of *preprocessing material* is generated to speed up the protocol. While steps (i) to (iv) happen in a so-called *online phase*, the preprocessing happens in a more demanding *offline phase* before that. This kind of separation is applicable for many use cases as the offline phase only requires knowing an upper bound on the operations in the online phase, but the actual data does not have to be known at that time. The verification (iv) is done with so-called authenticated secret-sharing based on MACs.

SPDZ MACs SPDZ uses a MAC scheme to detect misbehavior. The tag for message x is computed as $\alpha \cdot x$ with MAC key α. In the protocol, the parties compute $[\alpha \cdot x]$, i.e., no party knows the MAC tag – only shares of it (and shares $[\alpha]$). The MAC can be checked without revealing the MAC key [25]. With this scheme, only the overall values are tagged, not the individual shares. Several protocols [8,21,7] adapt this and generate MACs for the individual shares, which allows the parties to detect misbehavior on a per-party basis (see Sec. 2.3).

2.2 Homomorphic Encryption: BGV

In the offline phase of SPDZ [26] (cf. Sec. 2.1), homomorphic encryption (HE) is used to generate the preprocessing material. HE allows operations on encrypted data, e.g., we can add up two ciphertexts and obtain a ciphertext that contains the sum of the encrypted summands. In SPDZ and later works, the BGV encryption scheme [14] is used.

While we describe the basic concepts here, more technical details can be found in [14,26,25]. Let $R \coloneqq \mathbb{Z}[X]/\Phi_m(X)$ be the integer polynomials modulo the m-th cyclotomic polynomial $\Phi_m(X) \coloneqq X^{m/2} + 1$ for $m > 2$ a power of two. Let $N \coloneqq m/2$. Let $R_p \coloneqq \mathbb{Z}_p[X]/\Phi_m(X) = R/pR$ for a prime $p \equiv 1 \bmod m$.

For plaintext modulus p and ciphertext modulus $q > p$, the BGV scheme consists of a public key / private key pair $(\mathsf{pk}, \mathsf{sk}) \in R_q^2 \times R_q$, the encryption function $\mathsf{Enc}_{\mathsf{pk}} : R_p \times R_q^3 \to R_q^2$ (that uses randomness sampled from R_q^3), and the decryption function $\mathsf{Dec}_{\mathsf{sk}} : R_q^2 \to R_p$. We can embed plaintext vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{Z}_p^N$ into R_p such that (note that we denote vectors in bold, e.g., $\boldsymbol{x}$)

$$\mathsf{Dec}_{\mathsf{sk}}(\mathsf{Enc}_{\mathsf{pk}}(\boldsymbol{x}, \boldsymbol{r}) + \mathsf{Enc}_{\mathsf{pk}}(\boldsymbol{y}, \boldsymbol{r}')) = \boldsymbol{x} + \boldsymbol{y} \tag{3a}$$

$$\mathsf{Dec}_{\mathsf{sk}}(\boldsymbol{x} \cdot \mathsf{Enc}_{\mathsf{pk}}(\boldsymbol{y}, \boldsymbol{r}')) = \boldsymbol{x} \odot \boldsymbol{y} \tag{3b}$$

for (suitable) randomness $\boldsymbol{r}, \boldsymbol{r}'$, i.e., BGV allows linear homomorphic operations on ciphertexts. We use this linear version of BGV, as well as a somewhat homomorphic version, where we can also multiply two encrypted values (instead of one encrypted and one unencrypted value as in Eq. (3b)). We use a linear BGV key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$ per compute party $P_i \in \mathcal{P}$, as well as a somewhat homomorphic BGV key pk known to all parties, but the private key is secret-shared between the parties. This way, encryptions with pk can only be decrypted if all parties participate in a distributed decryption subprotocol DistDec. We use the notation $\langle x \rangle_i$ for encryptions of x under key pk_i and $\langle\!\langle x \rangle\!\rangle$ for encryptions under pk. Additionally, we use so-called drowning encryption $\mathsf{EncDrown}$, which hides the noise in homomorphically computed ciphertexts [33,41].

2.3 MAC Scheme

We use a MAC scheme that is conceptually similar to the ones used in other protocols with identifiable abort [8,21,7]. There, and in this work, the MAC is of the form $\alpha \cdot x + \rho$ for message x, MAC key α, and per-message randomness ρ. Like Baum et al. (BMRS [7]), we use pseudorandomness for the per-message randomization and define the tag for message x as

$$(\!|x|\!)^{\alpha,\mathsf{fk}}_{\mathsf{ctx}} \coloneqq \mathsf{MAC}(\alpha, \mathsf{fk}, x, \mathsf{ctx}) \coloneqq \alpha \cdot x + \mathsf{PRF}(\mathsf{fk}, \mathsf{ctx}) \tag{4}$$

where ctx is a unique context for x, e.g., an increasing counter, and fk is a PRF key. This is equivalent to using the MAC scheme presented by Catalano and Fiore [18] (see Appendix A for a proof of equivalence of the representations). The main difference to prior work is how we use the MAC scheme. Previously, the MAC scheme was instantiated once for every party and applied to every share, i.e., the protocols would compute $\alpha_j \cdot [x]_i + \rho_{ij}$ for every pair of parties $(P_i, P_j) \in \mathcal{P}^2$ (with $P_i \neq P_j$). This allows all parties to check the shares of all other parties. Our protocol uses a global (not party-dependent) MAC key instead. This has two main benefits: (i) Now there is only a single tag per share, reducing the complexity by a factor of n compared to, e.g., BOS [8]; and (ii) all parties – including external parties – verify all tags with the same MAC key, giving us a straightforward way to achieve public verifiability. The challenge lies in generating and handing tags in a way that does not reveal the MAC key to any party before the verification phase and also allows for verifying the tag generation itself. Our subprotocols for tag generation and verification are designed to handle these challenges. Before we describe our protocols in Secs. 3 to 6, we describe MAC-related notation and the intuition for tag generation below.

Notation If the context and keys are clear, we use the simplified notation $(\!|x|\!)$ instead of $(\!|x|\!)^{\alpha,\mathsf{fk}}_{\mathsf{ctx}}$. If we want to highlight the use of a specific key, we use the shorthand $(\!|x|\!)^{\alpha}$, where we omit the PRF key and assume it is implicitly associated with the MAC key. In particular, the notation $(\!|x|\!)^{\omega}$ implies that another PRF key fk' associated with ω is used. We define *tag randomness* $\rho_x \coloneqq \mathsf{PRF}(\mathsf{fk}, \mathsf{ctx})$ used for $(\!|x|\!)$. With this, we define linear operations on tags and randomness (subtraction works analogously):

$$(\!|x+y|\!) \coloneqq (\!|x|\!) + (\!|y|\!), \qquad (\!|c \cdot x|\!) \coloneqq c \cdot (\!|x|\!), \qquad (\!|x+c|\!) \coloneqq (\!|x|\!), \tag{5}$$

$$\rho_{x+y} \coloneqq \rho_x + \rho_y, \qquad \rho_{c \cdot x} \coloneqq c \cdot \rho_x, \qquad \rho_{x+c} \coloneqq \rho_x - \alpha \cdot c \tag{6}$$

for messages x, y and public constants c. Verification is simply defined as

$$\mathsf{Check}(\alpha, z, \rho_z, (\!|z|\!)) = 1 \qquad \Leftrightarrow \qquad (\!|z|\!) = \alpha \cdot z + \rho_z. \tag{7}$$

As mentioned above, we want to use MACs for each of the shares. This gives us an *authenticated secret-sharing* scheme. For a share $[x]_i$, we define the corresponding authenticated share as $[\![x]\!]_i \coloneqq ([x]_i, (\!|[x]_i|\!))$ with linear operations derived from combining Eqs. (2) and (5):

$$[\![x+y]\!]_i \coloneqq [\![x]\!]_i + [\![y]\!]_i, \quad [\![c \cdot x]\!]_i \coloneqq c \cdot [\![x]\!]_i, \quad [\![x+c]\!]_i \coloneqq [\![x]\!]_i + (c \cdot \delta_i, 0). \tag{8}$$

Multiparty MACs A core part of our protocols is the generation of MAC tags for every party's shares obliviously, without revealing the share to other parties or the MAC key to any party. We can achieve this by secret-sharing the MAC key α into shares $[\alpha]_i$ and selecting a PRF that allows simple distributed evaluation. For this, we define

$$\mathsf{PRF}(\mathsf{fk}, \mathsf{ctx}) \coloneqq \sum_{i=0}^{n-1} \mathsf{PRF}(\mathsf{fk}_i, \mathsf{ctx}), \qquad \mathsf{fk} \coloneqq (\mathsf{fk}_i)_{i=0}^{n-1}, \tag{9}$$

(abusing the notation) to get a PRF that each party can partially evaluate without leaking the overall PRF key fk. Concretely, the parties compute

$$(\!|x|\!)_{\mathsf{ctx}}^{\alpha,\mathsf{fk}} = \sum_{i=0}^{n-1} [\alpha]_i \cdot x + \mathsf{PRF}(\mathsf{fk}_i, \mathsf{ctx}), \tag{10}$$

where party P_i computes the i-th summand. Note that the PRF in Eq. (9) is pseudorandom if at least one party is honest in computing their summand. Consequently, Eq. (10) is equivalent to Eq. (4), i.e., a valid MAC, but every party P_i can only compute one summand. This summand is also linear in x, so this can be easily done with linear HE, as we do in Sec. 5.1. This protects the privacy of x, which will be replaced by a share of each party in our protocol (recall that we want to generate tags for all shares).

3 Our Protocol

With the required background on secret-sharing, HE, and our MAC scheme covered in Sec. 2, we describe our protocol setting (security and communication model) next. Then, we give an overview of our protocol before we describe it in more detail in Secs. 4 to 6. We then prove its security in Sec. 7.

Security and Communication Model Our protocols are designed to be maliciously secure even in the presence of a dishonest majority, where all but at most one compute party may be malicious. Any number of clients can be malicious – even all clients. We assume that external auditor parties are honest as these parties cannot influence the computation [6]. We use $\mathcal{P}, \mathcal{I}, \mathcal{O}$ for the set of compute parties, input parties, and output parties, respectively. Note that the latter two types of parties can be clients in a client-server setting, while the compute parties are the servers. However, input/output parties could be compute parties as well.

Our security proofs are done in the universal composability (UC) model [17]. This means we prove that our protocol is equivalent to an ideal functionality that models our security properties: correctness of the computation, privacy of the inputs, independence of inputs, and publicly identifiable abort. We present the ideal functionality in Fig. 1 and our full protocol in Fig. 2. Our formal security proof can be found in Sec. 7, and additional (standard) functionalities that we use can be found in Appendix B. In our protocol and functionality notation, we omit explicitly sending session IDs and security parameters for a more concise presentation. Similarly, we omit explicitly prefixing messages with unique strings that specify the context of the message, e.g., our functionality implicitly sends $(\texttt{comp-result}, \texttt{sessionID}, \boldsymbol{y}_{\text{pub}}, \boldsymbol{y}_j)$ in Line 11 of Fig. 1 to the adversary $\mathcal{A}$.

```
    In the following, process the init, prep, and comp phases at most once and only
    in order. Additionally, only accept audit messages after the init phase.
1:  On input (init) from all $\mathcal{P}$:
2:      Store $\mathcal{M} \leftarrow \perp$ and $\boldsymbol{z} \leftarrow \perp$; receive the set of statically corrupted parties $\mathcal{C}$ from
        the adversary $\mathcal{A}$
3:  On input $(\texttt{prep}, I, M, O)$ from all $\mathcal{P}$:
4:      Receive and store the initial set of misbehaving parties $\mathcal{M} \subseteq \mathcal{C} \cap \mathcal{P}$ from $\mathcal{A}$
5:      Below, only accept circuits with at most $I$ inputs, $M$ multiplications, $O$ outputs
6:  On input $(\texttt{comp}, f)$ from all $\mathcal{P} \cup \mathcal{O}$ and $(\texttt{comp}, f, \boldsymbol{x}_j)$ from all $P_j \in \mathcal{I}$:
7:      if $\mathcal{M} = \emptyset$ then // i.e., no misbehavior in preprocessing
8:          Receive $\boldsymbol{x}'_j$ from $\mathcal{A}$ for each $P_j \in \mathcal{C} \cap \mathcal{I}$ and use $\boldsymbol{x}'_j$ instead of $\boldsymbol{x}_j$
9:          Evaluate the function on all inputs, i.e., compute $\boldsymbol{y} := f((\boldsymbol{x}_j)_{P_j \in \mathcal{I}})$
10:         Let $\boldsymbol{y}_{\text{pub}}$ be all public outputs and $\boldsymbol{y}_j$ be the private outputs for $P_j \in \mathcal{O}$
11:         Send $\boldsymbol{y}_{\text{pub}}$ and $\boldsymbol{y}_j$ for $P_j \in \mathcal{C} \cap \mathcal{O}$ to $\mathcal{A}$
12:         Receive and store $\mathcal{M} \subseteq \mathcal{C}$ from $\mathcal{A}$; if $\mathcal{M} = \emptyset$, store $\boldsymbol{z} \leftarrow \boldsymbol{y}$
13:         Send $(\boldsymbol{z}_{\text{pub}}, \mathcal{M})$ to all parties and also $\boldsymbol{z}_j$ to all $P_j \in \mathcal{O}$ // $z = \perp$ if $\mathcal{M} \neq \emptyset$
14: On input (audit) from $P_{\text{audit}}$:
15:     Send $(\mathcal{M}, \boldsymbol{z}_{\text{pub}})$ to $P_{\text{audit}}$ and also $\boldsymbol{z}_j$ for any $P_j \in \mathcal{O}$ with $P_{\text{audit}} = P_j$.
        Note that $\boldsymbol{z}$ can be $\perp$; then, we define $\boldsymbol{z}_{\text{pub}}$ and $\boldsymbol{z}_j$ as $\perp$ as well.
```

Fig. 1: Functionality $\mathcal{F}_{\text{MPC}}$ with Publicly Identifiable Abort

We assume that the parties can communicate via secret authenticated channels. We also require an authenticated broadcast functionality. In our protocols, this is expressed by a Broadcast instruction. The use of broadcasts, or alternatively a public bulletin board, is standard in protocols that try to achieve properties that can be publicly checked (e.g., public verifiability [6], publicly identifiable abort [8,23], or public accountability [42]). With a bulletin board, an external auditor does not have to exist during the run of the protocol, but instead, a transcript of the public messages can be obtained from the bulletin board [6]. Additionally, parties can Forward messages in our protocol. This means that they broadcast messages that were received over private channels and all other parties can see that this message came from the original sender. In practice, we would implement this with signatures. In general, we assume then that all parties implicitly ignore messages without valid signatures.

Protocol Overview Our protocol is shown in Fig. 2, with more details in Figs. 3 and 4. We structure it in a setup, offline, online, and verification phase. The setup

```
The parties process the following phases only in order and at most once (except for
auditing). Parties can abort at any step where they verify data from other parties
and misbehaving parties are identified, e.g., Line 14 in Fig. 4.
1: Setup:
2:     Each $P_i \in \mathcal{P}$ does the following:
3:         Run $\mathcal{F}_{\text{setup}}$ to obtain BGV keys and setup the commitment functionality
4:         Sample symmetric keys $\mathsf{k}_i, \mathsf{k}_{ij}, \mathsf{k}'_i, \mathsf{k}'_{ik}$ for $P_j \in \mathcal{I}, P_k \in \mathcal{O}$
5:         Sample MAC key shares and PRF/PRG keys $[\alpha]_i, [\omega]_i, \mathsf{fk}_i, \mathsf{gk}_i, \mathsf{fk}'_i, \mathsf{gk}'_i$
6:         Publicly commit to the keys
7: Preprocessing: // Parties agreed on circuit size: $I$ inputs, $M$ muls., $O$ outputs
8:     // Offline phase (see Fig. 4):
9:     Each $P_i \in \mathcal{P}$ does the following:
10:        Run AuthInputShare($I$) for inputs, AuthTripleShare($M$) for multi-
           plications, and AuthOutputShare($O$) for outputs (including verifying all
           ZKPs; note that Auth is verified after the offline phase)
11:        Broadcast prep-ok
12: Computation: // Parties agreed on circuit $f$
13:    // Online phase (input, circuit evaluation, output; see Fig. 3):
14:    Each $P_i \in \mathcal{P}$ and $P_j \in \mathcal{I}$ does the following:
15:        Run Input for each input
16:    Each $P_i \in \mathcal{P}$ does the following:
17:        Traverse $f$ in topological order and evaluate each arithmetic gate in $f$ with
           Add, AddConst, MulConst, or Mul
18:    Each $P_i \in \mathcal{P}$ does the following:
19:        Run Output for each public or private output
20:    // Verification phase (see Fig. 3):
21:    Each $P_i \in \mathcal{P}$ does the following:
22:        Decommit $[\alpha]_i, \mathsf{fk}_i, \mathsf{gk}_i$ and verify all calls to Auth that used these keys
23:        Broadcast auth-ok and decommit $\mathsf{k}_i, \mathsf{k}_{ij}$
24:    Each $P_i \in \mathcal{P}$ and $P_j \in \mathcal{I}$ does the following:
25:        Verify all calls to Input
26:        Broadcast input-ok
27:    Each $P_i \in \mathcal{P}$ does the following:
28:        Verify the remaining arithmetic gates from online phase and Output
29:        Broadcast online-ok
30:    Each $P_i \in \mathcal{P}$ and $P_j \in \mathcal{O}$ does the following:
31:        Run FinOutput for each public and FinOutputTo for each private output
32:    Each $P_i \in \mathcal{P}$ does the following:
33:        Decommit $[\omega]_i, \mathsf{fk}'_i, \mathsf{gk}'_i$ and verify all calls to Auth that used these keys
34:        Broadcast auth-ok' and decommit $\mathsf{k}'_i, \mathsf{k}'_{ij}$
35:    Each $P_i \in \mathcal{P}$ and $P_j \in \mathcal{O}$ does the following:
36:        Verify all calls to FinOutput and calls to FinOutputTo
37:        Broadcast output-ok
38: Audit:
39:    If any party did not send prep-ok, verify the ZKPs
40:    Verify any message that was Forwarded during the protocol
```

Fig. 2: Protocol Π_{MPC} with Publicly Identifiable Abort

simply generates the necessary keys for encryption, commitments, and MACs. Then, the offline phase produces authenticated shares as preprocessing material used in the online phase. The preprocessing material is similar to SPDZ-like protocols: authenticated masks for inputs and outputs, as well as authenticated multiplication triples for multiplications. In the online phase, we use authenticated secret-sharing to perform computations on private data. We use the homomorphic property of the secret-sharing scheme, as well as Beaver's trick [11] to multiply shares. We give specialized input and output subprotocols that directly use our authenticated shares. As input and output parties can verify shares they receive themselves, we avoid overhead from protocols such as Damgård et al.'s input/output protocols [24]. For verification, we release verification data (keys for MACs and encryption) step-by-step to verify the computation locally with relatively low cost. Only after verifying the protocol are the outputs revealed, followed by an additional (final) verification step to also verify the outputs. In particular, our protocol proceeds as follows after the setup (see also Fig. 2).

(i) The authenticated preprocessing material is generated using HE. However, we only verify the correlations of the preprocessing material (e.g., for multiplication triples) in the offline phase, *not* correct authentication of shares (see Eq. (10) and Sec. 5.1). This avoids heavy cryptographic mechanisms like additional ZKPs and comes almost for free: The information needed to verify the authentication is revealed later anyway for MAC checks, so we defer it to the verification phase and perform a few more operations there.

(ii) The online phase proceeds as in any SPDZ-like protocol, utilizing the homomorphic properties of the secret-sharing scheme and the MACs. However, we only proceed until just before the outputs of the protocol and publicly open masked versions of the outputs (these can be verified in the next steps). Another caveat of our protocol is that we cannot reveal the MAC tags just yet as the tag generation was not checked, and faulty tags might leak information (see Sec. 6.1). Therefore, we only reveal symmetrically encrypted tags in the online phase. The parties commit to the corresponding symmetric keys in a setup phase, which are decommitted before verification of the tags in (iv), allowing everyone to decrypt the tags. Compared to homomorphic encryption in the offline phase, symmetric encryption has virtually no overhead (next to none for communication and very little computationally).

(iii) As the first step of the verification, the MAC and PRF keys are revealed. With this (and some auxiliary information in form of PRG keys used to deterministically derive randomness in for encryptions such that they can be recomputed in the verification), the parties can verify the authentication locally. Unlike protocols that use homomorphic commitments for identifiable abort [45,23,42], verifying the public-key primitives in our authentication does not depend on the circuit structure. Therefore, this is trivially parallelizable and fast to verify in a few nanoseconds per operation on modern hardware (see Sec. 8.2).

(iv) After the parties ensure that the tags are correctly authenticated in the offline phase, they can reveal the symmetric keys used to hide the MAC tags in the online phase. Then, the online phase can be verified with simple MAC checks. This can also be done very efficiently (see Sec. 8.2).

(v) Finally, the parties can reveal the output masks (used to publicly reveal masked outputs at the end of the online phase). To verify these, we proceed just as in (iii) and (iv) – with another caveat: The MAC and PRF keys are already public, so we cannot rely on MACs for verification for anything that is sent at this point in the protocol. Therefore, we switch to a second MAC and PRF key for the outputs and can then perform the verification of the authentication (like in (iii)) and the MACs (like in (iv)) for the output masks. Note that the offline phase with the second MAC and PRF key is also performed in step (i) together with the preprocessing using the primary MAC key.

In the following, we describe the online phase first (Sec. 4) so it is more clear what the offline phase (Sec. 5) has to generate. Finally, we describe how the parties verify the computation (Sec. 6). In the online and offline protocols (Figs. 3 and 4), we highlight steps that belong to the verification phase *like this.* Note that parties broadcast certain messages to synchronize the protocol into well-defined phases (see, for example, Line 11 in Fig. 2). Whenever a party identifies cheaters (e.g., Line 14 in Fig. 4), they finish verifying the remaining phase and then abort. Then, they do not participate in the next broadcast to end the current phase. This is done implicitly in all protocols.

4 The Online Protocol

In the online protocol, the compute parties $\mathcal{P}$ evaluate an agreed-upon function f (an arithmetic circuit) on private inputs from the input parties $\mathcal{I}$. The result can be partially public and partially private. In the latter case, there is also a set of output parties $\mathcal{O}$. As with standard SPDZ, all that is necessary for this are four main components: (i) Securely transforming private inputs into (authenticated) secret shares, (ii) performing linear operations and multiplications on shared values, (iii) securely sending the result to the parties that should obtain the result, and (iv) verifying the computation. Note that linear operations on shared values can be performed directly without interaction with the used linear secret-sharing scheme (cf. Eq. (8)). Therefore, we focus on multiplication for (ii). Additionally, we present the verification (iv) later in Sec. 6.2. The following Secs. 4.1 to 4.3 describe how we handle steps (i) to (iii) in more detail.

4.1 Input

In the client-server setting, we have dedicated input parties $\mathcal{I}$ that are allowed to give inputs to the multiparty computation (for $\mathcal{I} \subseteq \mathcal{P}$, see an optimization in Appendix C). With our publicly verifiable MAC scheme, the input parties can follow an input procedure that is very similar to existing input subprotocols where only compute parties can give inputs. The Input subprotocol (depicted in Fig. 3) works as follows. First, the compute parties obtain a fresh, uniformly random, and authenticated (shared) mask $[\![r]\!]$. This mask is computed in the

```
procedure OPEN(Secret $[\![x]\!]_i$)
    BROADCAST $[x]_i$ and $\mathsf{enc}(\mathsf{k}_i, (\![\![x]\!]_i\!))$ $\boxed{\textit{If } \mathsf{Check}(\alpha, [x]_i, \rho_{[x]_i}, (\![\![x]\!]_i\!)) \neq 1, \textit{ identify } P_i}$
    return $x \coloneqq \mathsf{Rec}([x])$
procedure OPENTO(Recipient $P_j$, Secret $[\![x]\!]_i$)
    Send $[x]_i$ and $\mathsf{enc}(\mathsf{k}_{ij}, (\![\![x]\!]_i\!))$ to $P_j$
    if partyID $= j$ then // partyID is the ID of the party running/verifying this
        $\boxed{\textit{If } \mathsf{Check}(\alpha, [x]_i, \rho_{[x]_i}, (\![\![x]\!]_i\!)) \neq 1, \textit{ identify } P_i \textit{ and } \text{FORWARD } \mathsf{enc}(\mathsf{k}_{ij}, (\![\![x]\!]_i\!))}$
        return $x \coloneqq \mathsf{Rec}([x])$
    $\boxed{\textit{If } P_j \text{ FORWARD}\textit{ed a message and if } \mathsf{Check}(\alpha, [x]_i, \rho_{[x]_i}, (\![\![x]\!]_i\!)) \neq 1, \textit{ identify } P_i}$
procedure INPUT(Recipient $P_j$, Input $x$ at $P_j$)
    Obtain a mask $[\![r]\!]_i$ from preprocessing
    OPENTO($P_j, [\![r]\!]_i$) $\boxed{\textit{Verify } \text{OPENTO}(P_j, [\![r]\!]_i) \textit{ with tag randomness } \rho_{[r]_i}}$
    if partyID $= j$ then // $P_j$ receives $r$ with OPENTO
        BROADCAST $u \coloneqq x - r$
    return $[\![x]\!]_i \coloneqq [\![r]\!]_i + (u \cdot \delta_i, 0)$ $\boxed{\textit{Store } \rho_{[r]_i} - \alpha u \delta_i \textit{ as tag randomness of } [\![x]\!]_i}$
procedure ADD(Secret $[\![x]\!]_i$, Secret $[\![y]\!]_i$)
    return $[\![z]\!]_i \coloneqq [\![x]\!]_i + [\![y]\!]_i$ $\boxed{\textit{Store } \rho_{[x]_i} + \rho_{[y]_i} \textit{ as tag randomness of } [\![z]\!]_i}$
procedure ADDCONST(Secret $[\![x]\!]_i$, Constant $c$)
    return $[\![z]\!]_i \coloneqq [\![x]\!]_i + (c \cdot \delta_i, 0)$ $\boxed{\textit{Store } \rho_{[x]_i} - \alpha c \delta_i \textit{ as tag randomness of } [\![z]\!]_i}$
procedure MULCONST(Constant $c$, Secret $[\![x]\!]_i$)
    return $[\![z]\!]_i \coloneqq c \cdot [\![x]\!]_i$ $\boxed{\textit{Store } c \cdot \rho_{[x]_i} \textit{ as tag randomness of } [\![z]\!]_i}$
procedure MUL(Secret $[\![x]\!]_i$, Secret $[\![y]\!]_i$)
    Obtain a triple $([\![a]\!]_i, [\![b]\!]_i, [\![c]\!]_i)$ from preprocessing
    $u \coloneqq$ OPEN($[\![x]\!]_i - [\![a]\!]_i$); $v \coloneqq$ OPEN($[\![y]\!]_i - [\![b]\!]_i$) $\boxed{\textit{Verify both calls to } \text{OPEN}}$
    return $[\![z]\!]_i \coloneqq [\![c]\!]_i + [\![a]\!]_i \cdot v + u \cdot [\![b]\!]_i + (u \cdot v \cdot \delta_i, 0)$
    $\boxed{\textit{Store } \rho_{[c]_i} + \rho_{[a]_i} \cdot v + u \cdot \rho_{[b]_i} - \alpha \cdot u \cdot v \cdot \delta_i \textit{ as tag randomness of } [\![z]\!]_i}$
procedure OUTPUT(Secret $[\![x]\!]_i^\alpha$)
    Obtain a double-authenticated mask $([\![r]\!]_i^\alpha, [\![r]\!]_i^\omega)$ from preprocessing
    $u \coloneqq$ OPEN($[\![x]\!]_i^\alpha - [\![r]\!]_i^\alpha$) $\boxed{\textit{Verify } \text{OPEN} \textit{ with tag randomness } \rho^\alpha_{[x]_i} - \rho^\alpha_{[r]_i}}$
procedure FINOUTPUT(Secret $[\![x]\!]_i^\alpha$) // use $\mathsf{k}'_i$ instead of $\mathsf{k}_i$ in OPEN below
    Let $u$ and $[\![r]\!]_i^\omega$ be as in OUTPUT($[\![x]\!]_i^\alpha$)
    $r \coloneqq$ OPEN($[\![r]\!]_i^\omega$) $\boxed{\textit{Verify } \text{OPEN}([\![r]\!]_i^\omega) \textit{ with tag randomness } \rho^\omega_{[r]_i} \textit{ using } \omega, \mathsf{fk}'}$
    return $x \coloneqq u + r$
procedure FINOUTPUTTO(Recipient $P_j$, Secret $[\![x]\!]_i^\alpha$) // use $\mathsf{k}'_{ij}$ instead of $\mathsf{k}_{ij}$
    Let $u$ and $[\![r]\!]_i^\omega$ be as in OUTPUT($[\![x]\!]_i^\alpha$)
    OPENTO($P_j, [\![r]\!]_i^\omega$) $\boxed{\textit{Verify } \text{OPENTO}(P_j, [\![r]\!]_i^\omega) \textit{ with } \rho^\omega_{[r]_i} \textit{ using } \omega, \mathsf{fk}'}$
    if partyID $= j$ then // $P_j$ receives $r$ with OPENTO
        return $x \coloneqq u + r$
```

Fig. 3: Online Protocol at Party P_i. See Sec. 4 for details on the online phase and Sec. 6.2 for the verification. The verification phase is marked $\boxed{\textit{like this}}$.

preprocessing (see Sec. 5.1). The compute parties send their shares and the MAC tags for their shares (for later verification, see Sec. 6.2) to an input party $P_j \in \mathcal{I}$ that holds an input x. Note that the tags are symmetrically encrypted to avoid leaking information via incorrectly tagged shares; we describe the reason for this in more detail in Sec. 6.1. The mask r can be reconstructed from the shares $[r]$, and P_j broadcasts $x - r$. As r is uniformly random, this does not leak information about x. Furthermore, the linear secret-sharing scheme allows the compute parties to compute $[\![x]\!] = [\![r + (x - r)]\!]$, which is then used to perform the remaining computations in the online phase.

4.2 Multiplication

Multiplication of shares is done with a standard technique: Beaver triples [11]. For this, authenticated triples $([\![a]\!], [\![b]\!], [\![c]\!])$ with uniform random a, b, and $c = a \cdot b$ are required. These triples can be precomputed in the offline phase (see Sec. 5.2). In the online phase, the triples are used as shown in Fig. 3: The values a and b mask the values x and y that should be multiplied. As the masks are uniformly random, we can open masked values $u \coloneqq x - a$ and $v \coloneqq y - b$ without leaking any information. Finally, we can use the opened values together with the authenticated triple to get a share of the product $x \cdot y$. More concretely, the classical Beaver multiplication is

$$x \cdot y = a \cdot b + a \cdot (y - b) + (x - a) \cdot b + (x - a) \cdot (y - b) \qquad (11)$$

$$\Rightarrow \qquad [\![x \cdot y]\!] \coloneqq [\![c]\!] + [\![a]\!] \cdot v + u \cdot [\![b]\!] + u \cdot v \qquad (12)$$

for scalar multiplication. This technique can be extended to any bilinear operation, such as matrix multiplication or convolutions [37,19,39,41]. Then, multiplication is replaced by the corresponding bilinear operation (matrix multiplication/convolution), and a, b, c are shared matrices or tensors. For the latter, the sharing scheme is (trivially) extended element-wise to matrices or tensors. As the above only uses linear operations, the operations can be verified with the authenticated secret-sharing scheme described in Sec. 2.3. We describe the verification in more detail in Sec. 6.2. Also note that, analogously to inputs (Sec. 4.1), only symmetrically encrypted tags are opened in the online phase (to avoid information leakage; see Sec. 6.1). During verification, the corresponding keys are decommitted to allow access to the tags via decryption.

4.3 Output

After evaluating all arithmetic gates, the parties hold authenticated shares of the outputs. The final outputs of the protocol are revealed in a multi-stage procedure, as shown in Figs. 2 and 3. This prevents an adversary from obtaining more information than only the output, similar to what is done in SPDZ [26].

The following example makes clear why this is necessary. Assume the MPC parties want to compute $f(x, y, c) \coloneqq x + c \cdot (y - x) \eqqcolon z$ for inputs $x, y \in \mathbb{F}$ and $c \in \{0, 1\} \subset \mathbb{F}$, i.e., $z = x$ for $c = 0$ and $z = y$ for $c = 1$. This uses only one multiplication and two linear addition/subtraction gates. Let x, y be secret inputs by honest parties. Additionally, we assume that the adversary knows or chooses $c := 0$ and controls at least one compute party. Then, a single malicious party P_j can use $[c]_j + 1$ instead of $[c]_j$ in the multiplication protocol, which results in the final computation result $z' = y$ instead of $z = x$. In the security proof for our protocol (cf. Sec. 7), a simulator for the protocol would only obtain the real output $z = x$ and has no way to provide any value related to y, so an adversary could easily distinguish the simulation from the real protocol.

Our proposed multi-stage output in Fig. 3 reveals a masked value (in the Output subprotocol) before the computation is verified. Only if the computation was performed without error, the final result is revealed using FinOutput for public outputs and FinOutputTo for private outputs. While an adversary can still introduce errors in the final result at this stage, this cannot be related to intermediate values in the computation.

For the multi-stage output, we use double-authenticated masks, i.e., shares of a mask authenticated with two different MAC (and PRF) keys. As the mask is uniformly random, we can publicly open masked values without risking privacy. Then, the computation can be publicly verified until the public opening of the masked value. Finally, the mask is opened – publicly for public outputs or privately for private outputs. The opened mask is then verified with the second key pair (cf. Sec. 6.2), and the mask can be removed locally to obtain the output. Note that, as for the rest of the online phase, all opened tags are first only published in an encrypted form to allow more controlled step-by-step verification (cf. Sec. 6.1).

Note that we can skip the multi-stage output if the function to be computed in MPC is linear, i.e., we can use Open/OpenTo instead of Output and FinOutput/FinOutputTo (Fig. 3). This is because linear functions are computed locally by the compute parties, without the influence of the adversary. Introducing errors in Open/OpenTo is then equivalent to introducing errors in the final stage of the multi-stage output procedure above.

5 The Offline Protocol

To perform computations in the online phase as described in Sec. 4, we need to generate the necessary authenticated shares: authenticated masks for inputs (used in Sec. 4.1), authenticated triples for multiplications (used in Sec. 4.2), and double-authenticated masks for outputs (used in Sec. 4.3). The base for these authenticated shares will be our share authentication subprotocol presented in Sec. 5.1, on which we build a triple generation subprotocol (Sec. 5.2) and a subprotocol to generate double-authenticated masks (Sec. 5.3).

5.1 Authentication

Our authentication subprotocol Auth is shown in Fig. 4. With it, each compute party P_i authenticates their share $[\boldsymbol{r}]_i$ and helps all other compute parties authenticate their share as well. We assume all parties hold their share $[\boldsymbol{r}]_i$, their share of the MAC key $[\alpha]_i$, their PRF key fk_i, and a PRG key gk_i. Additionally, all parties hold encryptions $\langle[\boldsymbol{r}]_j\rangle_j$ of $[\boldsymbol{r}]_j$ (encrypted under P_j's public key pk_j) for all $P_j \in \mathcal{P}$.

The parties proceed then in a pairwise manner, similar to Overdrive's Low-Gear protocol [33]: Every other party's encrypted share is multiplied with the own $[\alpha]_i$ and masked with a (pseudo)random (drowned) encryption. The pairwise results are then summed up to obtain a tag for the input share (instead of a share of a MAC as in Overdrive; see Eq. (10)). Using fixed pseudorandomness for the masks $\boldsymbol{s}_{i,j}$ and the drowning encryption allows us to delay verification of the authentication. Like this, we can reveal the PRF/PRG keys later in the protocol, and parties locally recompute what was sent to them while keeping the shares $[\boldsymbol{r}]_i$ still secret (see Sec. 6.1).

All that is left to generate authenticated masks in the AuthInputShare subprotocol (used for inputs; see Sec. 4.1) is to obtain the encrypted shares of all parties. This can be done with standard zero-knowledge proofs as in SPDZ [26] or with more recent optimizations such as (variants of the) TopGear proofs [5].[4] We use a ZKP subprotocol Enc-ZK for this, which we describe in Appendix B.

5.2 Multiplication

To multiply shared values in the online phase, we require Beaver triples (see Sec. 4.2). Our triple generation subprotocol AuthTripleShare is shown in Fig. 4. Its high-level goal is to generate encrypted shares for all components of the triple $(a, b, c = a \cdot b)$ and then re-use the authentication from Sec. 5.1. This approach is similar to SPDZ [26] and related protocols like [6,8]. However, we use a different protocol for authentication. Additionally, we need to ensure PIA. The straightforward way to achieve this is to use somewhat homomorphic encryption (as the mentioned related work) and (verifiable) distributed decryption. However, all parties' shares need to be encrypted with the respective party's public key for our authentication subprotocol. This prevents us from homomorphically combining different parties' encryptions or from using standard distributed decryption. Therefore, we require the parties to additionally encrypt their shares with a common public key pk, so all parties' shares can be homomorphically summed and multiplied. For this, we use the PubEnc-ZK subprotocol, which is similar to Enc-ZK (used in Fig. 4; cf. Sec. 5.1 and Appendix B). It outputs not only the encryptions of each party's share but also an encryption of the sum of all parties' shares under the shared public key pk. This can be achieved with a ZKP that combines the one from Enc-ZK with a TopGear proof [5] to also get the encrypted sum of all shares.[5]

[4] Opposed to [5], we do not require n-party proofs but 1-party proofs. Furthermore, TopGear ZKPs require the shares and ciphertexts to be multiplied by 2 as these proofs only guarantee small noise for encryptions of $2 \cdot \boldsymbol{r}$. For simpler presentation, this is omitted in Fig. 4.

[5] As mentioned in Footnote 4, we have to multiply the individual parties' shares and ciphertexts by 2 if we use TopGear ZKPs. Again, this is omitted in Fig. 4 for simpler presentation.

```
1: procedure $\textsc{Auth}([\alpha]_i, \mathsf{fk}_i, \mathsf{gk}_i, [\boldsymbol{r}]_i, (\langle[\boldsymbol{r}]_j\rangle_j)_{j=0}^{n-1})$
2:     Let $\mathsf{ctx}$ be a string describing the context of this call, e.g., triple-a for Line 29
3:     for $j$ from $0$ to $n-1$ do
4:         $\boldsymbol{s}_{i,j} := \mathsf{PRF}(\mathsf{fk}_i, (\mathsf{ctx}, j))$
5:         if partyID $= j$ then
6:             $\boldsymbol{t}_{i,i} := [\alpha]_i \cdot [\boldsymbol{r}]_i + \boldsymbol{s}_{i,i}$
7:         else
8:             // access PRG output at index determined by $(\mathsf{ctx}, j)$
9:             $\langle\boldsymbol{s}_{i,j}\rangle_j := \mathsf{EncDrown}(\mathsf{pk}_j, \boldsymbol{s}_{i,j}, \mathsf{PRG}(\mathsf{gk}_i)[\mathsf{ctx}, j])$
10:            $\langle\boldsymbol{t}_{i,j}\rangle_j := [\alpha]_i \cdot \langle[\boldsymbol{r}]_j\rangle_j + \langle\boldsymbol{s}_{i,j}\rangle_j$
11:            Send $\langle\boldsymbol{t}_{i,j}\rangle_j$ to $P_j$ and receive $\langle\boldsymbol{t}_{j,i}\rangle_i$ from $P_j$
12:            Compute $\langle\boldsymbol{t}_{j,i}\rangle_i$ analogously to above; let the computed value be $\langle\boldsymbol{t}'_{j,i}\rangle_i$
13:            If partyID $= i$ and if $\langle\boldsymbol{t}_{j,i}\rangle_i \neq \langle\boldsymbol{t}'_{j,i}\rangle_i$, identify $P_j$ and $\textsc{Forward}$ $\langle\boldsymbol{t}_{j,i}\rangle_i$
14:            If $P_i$ $\textsc{Forward}$ed $\langle\boldsymbol{t}_{j,i}\rangle_i$ and if $\langle\boldsymbol{t}_{j,i}\rangle_i \neq \langle\boldsymbol{t}'_{j,i}\rangle_i$, identify $P_j$ as cheater
15:            $\boldsymbol{t}_{j,i} := \mathsf{Dec}(\mathsf{sk}_i, \langle\boldsymbol{t}_{j,i}\rangle_i)$
16:    return $[\![\boldsymbol{r}]\!]_i := ([\boldsymbol{r}]_i, (\![\boldsymbol{r}]_i]\!)) := ([\boldsymbol{r}]_i, \sum_{j=0}^{n-1} \boldsymbol{t}_{j,i})$
17: procedure $\textsc{AuthInputShare}(\text{Count } m)$
18:    $[\boldsymbol{r}]_i \xleftarrow{\$} \mathbb{F}^m$ // sample random shares
19:    $(\langle[\boldsymbol{r}]_j\rangle_j)_{j=0}^{n-1} \xleftarrow{\$} \textsc{Enc-ZK}([\boldsymbol{r}]_i)$ // includes verification via ZKPs
20:    return $\textsc{Auth}([\alpha]_i, \mathsf{fk}_i, \mathsf{gk}_i, [\boldsymbol{r}]_i, (\langle[\boldsymbol{r}]_j\rangle_j)_{j=0}^{n-1})$ $\boxed{\textit{Verify } \textsc{Auth} \textit{ as above}}$
21: procedure $\textsc{AuthTripleShare}(\text{Count } m)$
22:    $([\boldsymbol{a}]_i, [\boldsymbol{b}]_i, [\boldsymbol{r}]_i) \xleftarrow{\$} \mathbb{F}^{3\times m}$ // sample random shares
23:    $((\langle[\boldsymbol{a}]_j\rangle_j)_{j=0}^{n-1}, \langle\!\langle\boldsymbol{a}\rangle\!\rangle) \xleftarrow{\$} \textsc{PubEnc-ZK}([\boldsymbol{a}]_i)$ // includes verification via ZKPs
24:    $((\langle[\boldsymbol{b}]_j\rangle_j)_{j=0}^{n-1}, \langle\!\langle\boldsymbol{b}\rangle\!\rangle) \xleftarrow{\$} \textsc{PubEnc-ZK}([\boldsymbol{b}]_i)$ // includes verification via ZKPs
25:    $((\langle[\boldsymbol{r}]_j\rangle_j)_{j=0}^{n-1}, \langle\!\langle\boldsymbol{r}\rangle\!\rangle) \xleftarrow{\$} \textsc{PubEnc-ZK}([\boldsymbol{r}]_i)$ // includes verification via ZKPs
26:    $\boldsymbol{m} \xleftarrow{\$} \textsc{DistDec}(\langle\!\langle\boldsymbol{a}\rangle\!\rangle \cdot \langle\!\langle\boldsymbol{b}\rangle\!\rangle - \langle\!\langle\boldsymbol{r}\rangle\!\rangle)$ // includes verification via ZKPs
27:    $[\boldsymbol{c}]_i := [\boldsymbol{r}]_i + \delta_i \cdot \boldsymbol{m}$
28:    $(\langle[\boldsymbol{c}]_j\rangle_j)_{j=0}^{n-1} := (\langle[\boldsymbol{r}]_j\rangle_j + \delta_j \cdot \boldsymbol{m})_{j=0}^{n-1}$
29:    $[\![\boldsymbol{a}]\!]_i := \textsc{Auth}([\alpha]_i, \mathsf{fk}_i, \mathsf{gk}_i, [\boldsymbol{a}]_i, (\langle[\boldsymbol{a}]_j\rangle_j)_{j=0}^{n-1})$ $\boxed{\textit{Verify } \textsc{Auth} \textit{ as above}}$
30:    $[\![\boldsymbol{b}]\!]_i := \textsc{Auth}([\alpha]_i, \mathsf{fk}_i, \mathsf{gk}_i, [\boldsymbol{b}]_i, (\langle[\boldsymbol{b}]_j\rangle_j)_{j=0}^{n-1})$ $\boxed{\textit{Verify } \textsc{Auth} \textit{ as above}}$
31:    $[\![\boldsymbol{c}]\!]_i := \textsc{Auth}([\alpha]_i, \mathsf{fk}_i, \mathsf{gk}_i, [\boldsymbol{c}]_i, (\langle[\boldsymbol{c}]_j\rangle_j)_{j=0}^{n-1})$ $\boxed{\textit{Verify } \textsc{Auth} \textit{ as above}}$
32:    return $([\![\boldsymbol{a}]\!]_i, [\![\boldsymbol{b}]\!]_i, [\![\boldsymbol{c}]\!]_i)$
33: procedure $\textsc{AuthOutputShare}(\text{Count } m)$
34:    $[\boldsymbol{r}]_i \xleftarrow{\$} \mathbb{F}^m$ // sample random shares
35:    $(\langle[\boldsymbol{r}]_j\rangle_j)_{j=0}^{n-1} \xleftarrow{\$} \textsc{Enc-ZK}([\boldsymbol{r}]_i)$ // includes verification via ZKPs
36:    $[\![\boldsymbol{r}]\!]_i^\alpha := \textsc{Auth}([\alpha]_i, \mathsf{fk}_i, \mathsf{gk}_i, [\boldsymbol{r}]_i, (\langle[\boldsymbol{r}]_j\rangle_j)_{j=0}^{n-1})$ $\boxed{\textit{Verify } \textsc{Auth} \textit{ as above}}$
37:    $[\![\boldsymbol{r}]\!]_i^\omega := \textsc{Auth}([\omega]_i, \mathsf{fk}'_i, \mathsf{gk}'_i, [\boldsymbol{r}]_i, (\langle[\boldsymbol{r}]_j\rangle_j)_{j=0}^{n-1})$ $\boxed{\textit{Verify } \textsc{Auth} \textit{ with } \omega, \mathsf{fk}', \mathsf{gk}'}$
38:    return $([\![\boldsymbol{r}]\!]_i^\alpha, [\![\boldsymbol{r}]\!]_i^\omega)$
```

Fig. 4: Offline Protocol at Party P_i. See Sec. 5 for details on the offline phase and Sec. 6.1 for the verification. The verification phase is marked $\boxed{\textit{like this}}$.

With the encryptions of $\boldsymbol{a}$ and $\boldsymbol{b}$, everyone can compute a ciphertext that encrypts the component-wise product $\boldsymbol{a} \odot \boldsymbol{b}$. The parties can use an encryption of $\boldsymbol{r}$ to mask this ciphertext and decrypt a masked version of the product with the distributed decryption subprotocol DistDec. For the latter, ZKPs (as in [8,23,42]) give us PIA. Now, parties can homomorphically compute their own share $[\boldsymbol{c}] = [\boldsymbol{a} \odot \boldsymbol{b}]$ from $[\boldsymbol{r}]$ and encrypted shares for all parties from encryptions of $[\boldsymbol{r}]$, respectively. Finally, the authenticated shares can be computed with Auth as in Sec. 5.1. Note that one can easily modify our protocol to generate, e.g., convolution triples instead [41].

5.3 Output

As mentioned in Sec. 4.3, we require double-authenticated masks to prevent leaking intermediate values through outputs in the online phase. These masks are similar to those used for inputs (cf. Sec. 5.1) but need to be authenticated with two independent MAC and PRF keys. Therefore, AuthOutputShare (see Fig. 4) follows the same process as AuthInputShare, but we call Auth once for each set of keys.

6 The Verification Protocol

The final step of our protocol is verifying the computation. This happens after the online phase (or could be seen as the final step of the online phase). Verification is mainly done by the compute parties, which verify the remaining parts of the offline phase (Sec. 6.1) and the online phase (Sec. 6.2). Verification that depends on private data is done by input parties (Sec. 6.2) and output parties that receive private outputs (Sec. 6.2). If cheating is detected by compute parties, input parties, or output parties, the (private) messages that failed verification are published and can be verified by every party, including external auditors. The only part that involves expensive cryptographic primitives is verifying the offline phase, done by the compute parties. Therefore, our protocol has relatively low resource requirements for clients, i.e., input and/or output parties.

Note that once the verification phase is reached, no more messages that depend on the MAC key α and PRF key fk are sent. Therefore, the compute parties can reveal these, which allows for a more efficient verification than if we tried to keep the keys secret for the whole protocol. We discuss the verification of the offline phase next, followed by the verification of the online phase (see Sec. 6.2).

6.1 Verifying the Offline Phase

After ZKPs for the well-formedness of ciphertexts and distributed decryption are already verified in the offline phase, only verification of the Auth subprotocol is left. As mentioned before, this is deferred to the verification phase to improve the overall efficiency of our protocol, as we avoid more expensive verification mechanisms for this part of the protocol. We now describe the verification for this subprotocol in more detail.

Authentication In the offline phase (see Sec. 5.1 and Fig. 4), every compute party P_i sends an encryption $\langle \boldsymbol{t}_{i,j} \rangle_j$ to every other compute party P_j. This value is used to construct P_j's MAC tag for their share $[\boldsymbol{r}]_j$, and conversely, each P_j sends a ciphertext to P_i to construct the tag for their share. If any party P_j cheated in the construction of this encryption, P_i's share would later fail a MAC check, making it look like P_i was cheating. Therefore, it is critical to prevent this from happening in our protocol. Luckily, verifying the encryption comes almost for free with our protocol, as the MAC and PRF keys are published anyway to perform the MAC check. Therefore, the encryptions can simply be recomputed during verification to see which party cheated in the offline phase. The only missing information to recompute the encryption is the randomness used for the drowning encryption. We let the parties use PRGs to compute this randomness, meaning everyone can deterministically recompute the encryptions once the PRG keys are revealed (together with the MAC key shares and PRF keys). Note that revealing this randomness does not impact the privacy of any values in our protocol, as the drowning encryption is only used to hide P_i's $[\alpha]_i$ and $\boldsymbol{s}_{i,j}$ from other parties P_j. These values are already public at the time of the verification, so hiding the encryption randomness is no longer necessary.

If P_i detects that P_j cheated during the authentication step by sending a different ciphertext than the one P_i locally computed, P_i can publish $\langle \boldsymbol{t}_{j,i} \rangle_i$ that was sent via P2P communication in the offline phase. Now, everyone can see if this ciphertext matches the expected one. A malicious P_i cannot falsely blame an honest P_j because the $\langle \boldsymbol{t}_{j,i} \rangle_i$ is accompanied by a signature that P_i would have to forge in order to convince others that P_j sent a wrong value. As mentioned before, all parties simply ignore messages that do not have valid signatures.

For double-authenticated masks, we have a second set of MAC and PRF keys. Calls to AUTH that used these keys will be verified as soon the second set of keys is published (after verifying the computation with the first set of keys).

On Verifying the Authentication Before the Online Phase As can be seen in Figs. 2 and 3, the tags for messages are not simply revealed during openings but only encrypted versions thereof. Only after successful verification of the offline phase (including calls to AUTH) are the encryption keys decommitted to reveal the MAC tags via decryption. (The decryption is not shown in Fig. 2 but happens implicitly after decommiting the keys.) Directly revealing the tags would make our protocol vulnerable to attacks via manipulated tags. For this, take the example of opening a share $[\![x - r]\!]_i$ for honest P_i.[6] If P_j would cheat in AUTH by sending an encryption of $([\alpha]_j + 1) \cdot [r]_i + \mathsf{PRF}(\mathsf{fk}_j, i)$, the MAC tag for $[x - r]_i$ would be off by $[r]_i$. With public knowledge of the MAC and PRF keys (as we are now in the verification phase), P_j could take the difference between the tag published by P_i and the (locally computed) expected tag to

[6] During multiplication or for each output, values like this are opened where r is supposed to be uniformly random and unknown to all parties.

find out $[r]_i$.[7] In case P_i is the only honest party, this is enough information to infer r and thus also x. P_j would be identified when verifying Auth, but private information has already been leaked at that point. This is why we reveal the tags only after successful verification of Auth (indirectly by first sending only encryptions and later revealing the key; this acts as a commitment, but we only have a one-time cost for decommiting symmetric encryption keys).

6.2 Verifying the Online Phase

After the preprocessing in the offline phase and evaluating the circuit in the online phase, messages sent in the online phase have to be verified (see Fig. 3). This can be done by verifying tagged messages. Note that the offline phase is now fully verified, and the encryption keys to decrypt the MAC tags can be published (as discussed above). Next, we discuss all types of verifications for the online phase.

Input As seen in Fig. 3, input parties receive authenticated masks for each input (see Sec. 4.1). This includes shares from each compute party and tags for each share. To verify these, the input parties simply recompute the tag randomness, i.e., the PRF evaluation, and perform a MAC check for each authenticated share that they received. If any check fails, the corresponding share and tag can be published, and others can verify that the input party really received faulty shares. Note that publishing authenticated shares of the mask that failed verification does not risk the privacy of inputs. Firstly, these shares had to come from malicious parties as honest parties do not send wrong messages, so these shares are already known to an adversary attacking the protocol. Secondly, we assume that the offline phase was verified, i.e., shares of honest parties are ensured to be correctly tagged and thus not revealed. Because at least one compute party is honest, at least one share is not revealed, keeping the input secret.

Multiplication In each multiplication, two values are publicly opened (see Sec. 4.2 and Fig. 3). Verification of these messages works similarly to the input verification with the following differences. Firstly, every (compute) party verifies the opened values instead of individual designated input parties. Secondly, the tag randomness is not directly determined by the offline phase but is a linear combination of tag randomness: partly from the tuple entries a, b and partly from the inputs to Mul. (For the input masks, the tag randomness for the verifications is exactly what is used in the offline phase to authenticate the masks.) Therefore, the parties have to traverse the circuit again to compute the tag randomness for each multiplication (see Eq. (5) and Fig. 3). By computing the tag randomness as in Fig. 3 for the output of every gate (e.g., Input or Add gates), we maintain the invariant that the tag randomness of the input of every

[7] This can be generalized to let P_i act as a decryption oracle for any ciphertext encrypted under P_i's public key.

gate is known using Eqs. (5) and (6). This way, the MAC checks (cf. Eq. (7)) can be performed for all authenticated shares opened during multiplications.

Output Also for outputs, parties have to verify tags. The tag and its tag randomness for the publicly opened value in Output (cf. Fig. 3) depend on intermediate values of the circuit, similar to the opened values for multiplications. Therefore, they can be computed and checked in the same way. In contrast, the values that need to be verified in FinOutput and FinOutputTo only depend on masks from the preprocessing. This means that output parties do not have to traverse the whole circuit to check the tags they received, but they can perform the MAC check directly, as for inputs (see above).

As seen in Fig. 2, the whole computation except FinOutput/FinOutputTo is verified using the first key pair α, fk. Only after this is done, is the second key pair ω, fk' revealed, and the verification proceeds analogously to before: First, Auth is verified, and only then, are the keys to decrypt the final tags revealed. These tags are then checked to verify FinOutput/FinOutputTo.

6.3 Auditing the Protocol

As seen above, the verification of the protocol is done mainly by compute parties, and only very little is done by input/output parties. An external auditor also has to perform only few operations for the verification. In fact, if at least one party is honest, receiving the final `output-ok` message from all parties implies that the whole computation was successful. If a party cheated, on the other hand, either a public message did not verify or at least one honest party would Forward a private message that identifies a cheater. For the former, a public message is a ZKP during the offline phase or publicly opened tags during the online phase. For the latter, a private message is a ciphertext sent in Auth or an authenticated share for inputs/outputs. Verifying this only requires a single check of the Auth subprotocol or a MAC check.

7 Security of the Protocol

With our protocol described in Secs. 3 to 6, we can now prove that it is UC-secure. We describe our UC simulator in the proof of the theorem below. The detailed simulator can be found in Appendix B (Fig. 7).

Theorem 1. *Our protocol Π_{MPC} (see Fig. 2) securely realizes $\mathcal{F}_{MPC}$ (see Fig. 1) in the $(\mathcal{F}_{comm}, \mathcal{F}_{setup}, \mathcal{F}_{rand})$-hybrid model. We assume static corruption of up to $n-1$ of the n compute parties $\mathcal{P}$, as well as static corruption of any number of input parties $\mathcal{I}$ and output parties $\mathcal{O}$. Furthermore, we assume HomUF-CMA security of the used MAC scheme [18], the meaningless keys property of BGV [26], and enhanced CPA-security for BGV [33].*

Proof. Correctness of the Offline Phase. We start with the correctness of generating random authenticated shares in the offline phase (Fig. 4). The shares $[\boldsymbol{r}]_i$ are uniformly random shares of a uniformly random value as required by our protocols. The tags for these shares are

$$\begin{aligned}\sum_{j=0}^{n-1} \boldsymbol{t}_{j,i} &= \sum_{j=0}^{n-1} [\alpha]_j \cdot [\boldsymbol{r}]_i + \boldsymbol{s}_{j,i} \\ &= \sum_{j=0}^{n-1} [\alpha]_j \cdot [\boldsymbol{r}]_i + \mathsf{PRF}(\mathsf{fk}_j, i) \\ &= \alpha \cdot [\boldsymbol{r}]_i + \sum_{j=0}^{n-1} \mathsf{PRF}(\mathsf{fk}_j, i) \\ &= \alpha \cdot [\boldsymbol{r}]_i + \mathsf{PRF}(\mathsf{fk}, i) \\ &= (\!|[\boldsymbol{r}]_i|\!) \end{aligned} \tag{13}$$

as required. The correctness of the generated output shares (Fig. 4) follows trivially from this. For triple shares (Fig. 4), we note that $\boldsymbol{a}$ and $\boldsymbol{b}$ are correctly shared and authenticated by the above as well. The masked value $\boldsymbol{m} = \boldsymbol{a} \odot \boldsymbol{b} - \boldsymbol{r}$ implies that

$$\boldsymbol{c} = \sum_{j=0}^{n-1} [\boldsymbol{c}]_j = \boldsymbol{m} + \sum_{j=0}^{n-1} [\boldsymbol{r}]_j = \boldsymbol{a} \odot \boldsymbol{b},$$

i.e., the shares of $\boldsymbol{c}$ are correct, and similarly, $\langle [\boldsymbol{c}]_j \rangle_j$ are correct encryptions of these shares. Finally, the correct authentication for shares of $\boldsymbol{c}$ follows from the correctness of AUTH.

Correctness of the Online Phase. The correctness of the online phase (Fig. 3) follows from correctness of the offline phase, the linearity of the authenticated secret-sharing scheme, and Eq. (11).

Correctness of the Verification. With a correct offline phase, all compute parties have valid tags for their preprocessing material. Together with the linearity of shares and tags, i.e., Eqs. (2), (5) and (8), the MACs of opened values verify correctly. Additionally, the verification of AUTH only (correctly) recomputes what is done in the offline phase.

In the remainder of the proof, we describe why the interaction of an adversary with the real protocol is indistinguishable from an interaction with our simulator (see Fig. 7) and the ideal functionality (see Fig. 1). Again, we structure the proof by the phases of our protocol (offline, online, and verification phase).

Simulator. Before arguing our protocol's simulation security, we have to define our simulator. Similar to other SPDZ-like protocols, the simulator internally emulates the real protocol where the simulated honest input parties set their inputs to zero. This results in the simulated output z. Additionally, the simulator gets the computation result from the ideal functionality y (if the interaction does not already abort in the offline phase). With this, it can modify the share of one honest compute party such that the output in the real world and the ideal world are the same (by adding the difference $y - z$ to one share). Because the simulator fully controls the simulation, it can generate a valid MAC tag for this modified share. Next, we argue why the simulation and a real protocol run are indistinguishable, phase by phase.

Setup. The simulator can obtain all MAC and PRF keys through the commitment functionality (or a trapdoor if we instantiate it with an extractable commitment scheme). Similarly, it obtains the private BGV keys through $\mathcal{F}_{\text{setup}}$.

Offline Phase. The distributions of the real and ideal world are indistinguishable in the offline phase as the simulator simply runs the protocol. The simulation aborts if some parties fail to provide valid ZKPs, e.g., as part of the distributed decryption. Then, the simulator forwards the identified parties to the functionality and the ideal world also aborts. The simulator correctly detects all invalid values that could lead to wrong results later (due to correctness and soundness of the ZKPs). If all proofs are valid, the simulator obtains all shares of corrupted parties by decrypting the corresponding ciphertexts. With this and the obtained MAC and PRF keys from the setup, it is able to generate all expected values that a non-misbehaving (corrupted) party would compute in the online phase.

Online Phase. Note that we assume now that the protocol did not abort in the offline phase. As mentioned above, honest inputs are set to zero and corrupted inputs are recovered using the published masked values in the Input protocol and the masks extracted from the offline phase. The simulator forwards the inputs of corrupted parties to the functionality and obtains the public results and results for corrupted parties. With these values, the simulator can adapt the masked outputs (in the Output subprotocol) to make the simulated protocol output the same result as the functionality. For this, the simulator adjusts the shares of one honest compute party as described above. It can also compute corresponding MAC tags that would pass the MAC check. As there is exactly one MAC tag for each possible value of a share, once the MAC key and PRF are fixed, the tag is indistinguishable iff. the share is indistinguishable. Overall, all values in the online phase are distributed uniformly at random because they are all masked as in the family of SPDZ protocols, and thus, the simulation is indistinguishable. This does not include the encrypted MAC tags, but those are simply encryptions of indistinguishable values and are thus also indistinguishable.

Verification and Final Output. After publishing the MAC keys, the parties verify the computation. Verification of Auth is done first. This does not reveal information about the honest shares $[r]_i$ as these are still encrypted, and now we just compute linear functions of these ciphertexts. The simulation is also indistinguishable as we do not change the simulated behavior in the offline phase. If the verification of Auth identified some cheaters, the parties in the simulation abort now. Note that missing a wrong tag generation is impossible as there is exactly one value that the honest parties expect. If there was no abort, the simulation continues as follows. After verifying the tag generation in the offline phase as above, the opened tags are checked. As argued before, the revealed shares are indistinguishable as they are randomly masked, and the tags are also indistinguishable. For the final tags, detecting that $[z]_j - [r]_j + (y - z)$ is used and tagged instead of $[z]_j - [r]_j$ is equivalent to distinguishing encryptions of $[r]_j - (y - z)$ and $[r]_j$ (for the authentication), therefore these are indistinguishable (following a reduction to CPA-security similarly to Overdrive [33]). Also other uses of ciphertext, namely SHE-BGV ciphertexts used for triple generation, are indistinguishable as in SPDZ [26] or BOS [8]: via a reduction to the meaningless keys property of BGV. Note that, similar to BOS, our simulator does not need to extract private values from ZKPs to generate authenticated shares with indistinguishable distributions. This is because we can also use a setup functionality, which enables the simulator to decrypt any value (and a commitment functionality for the initial MAC keys), to do extract the necessary values. Finally, the final steps in FinOutput/FinOutputTo are the same in the real and ideal world and thus indistinguishable as well. □

8 Evaluation

Now that we have presented our protocol and proved it secure (in Secs. 3 to 7), we describe its practicality in more detail. We focus on the theoretical aspects (see Sec. 8.1) as well as real-world efficiency (see Sec. 8.2). For the latter, we perform multiple benchmarks and consider the example use case of secure aggregation. Note that we mostly compare our protocol to SPDZ-like protocols without identifiable abort and protocols with identifiable abort outlined in Sec. 1.1. All protocols with identifiable abort discussed before only provide an asymptotic analysis, i.e., no concrete parameters or benchmarks, except RRRK [42], which compares their protocol to SPDZ [26] and CFY [23]. However, they only microbenchmark the protocols for use cases that are not in the client-server setting, which is our focus. Moreover, their results show that both identifiable protocols have an overhead of factor 3 to 20 compared to SPDZ, while our protocol is much closer to SPDZ – partially even faster (see Sec. 8.2).

8.1 Theoretical Evaluation

A comparison of asymptotic protocol complexities can be found in Sec. 1.1 (Tabs. 2 and 3). There, we also compare the main cryptographic primitives used in the different protocols. As mentioned above, most identifiable protocols do not provide more than their asymptotic complexity. In the remainder of this section, we focus on how our protocol efficiency differs from SPDZ on a theoretical level, while Sec. 8.2 shows the results of our experimental comparison to SPDZ.

For the offline phase, notice that our protocol is very similar to a hybrid of LowGear [33] for authentication and TopGear [5] for generating multiplication triples. Our main overhead is using verifiable distributed decryption once per multiplication instead of non-verifiable distributed decryption. On the positive side, this avoids the need for sacrificing (see [8]: SPDZ has to over-produce triples, which is not necessary if decryption is verifiable), but that can be avoided in come cases, as shown recently [39]. Also, our ciphertext parameters are essentially the same (see Appendix D). A main difference to SPDZ is that we can use simpler correlated randomness for inputs and outputs. Concretely, SPDZ requires two *correlated* Beaver triples or alternatively a 5-tuple $(y, r, v, w = y \cdot r, u = v \cdot r)$ per input, and two Beaver triples per private output [24]. Our protocol requires no triples for inputs or outputs. Instead, we only require a single authenticated share per input and a double-authenticated share per output. These are cheaper to generate in the offline phase.

In the online phase, our protocol has a similar advantage over SPDZ-like protocols for inputs and outputs, as our protocol supports public verification. This means the clients can verify the random masks they receive for inputs themselves (see Secs. 4.1 and 6.2). Therefore, in our protocol, each compute party sends only two field elements to a client. In contrast, for SPDZ-like protocols, each compute party sends correlated Beaver triples (five field elements) to a client [24]. After the clients receive their elements, both our protocol and [24] require the clients to broadcast one element. Similarly, for private outputs [24], a Beaver multiplication (exchanging two field elements between compute parties) and sending five field elements to clients is required for SPDZ. Our protocol requires exchanging two field elements between compute parties (Open in Output) and sending two elements to clients (OpenTo in FinOutputTo).

For evaluating an arithmetic circuit in the online phase, compute parties have to perform the same work per linear operation as in SPDZ: two field operations (one for the share and one for the MAC tag). Multiplications require twice as much communication with our protocol because parties send two field elements instead of one per opening. Additionally, our protocol performs HE operations for verification. We show below that the overhead for these is very low.

In addition to the aforementioned communication cost, our protocol comes with a more involved verification. The MAC tags for all shares have to be checked after the circuit evaluation (Sec. 6.2) and also some computation of the offline phase has to be verified (Sec. 6.1). Fortunately, MACs are very cheap to verify and checks of the Auth subprotocol are relatively cheap as well. We demonstrate this next.

8.2 Real-World Evaluation

We implemented[1] our protocol to confirm the theoretical observations from Sec. 8.1 and to show the concrete practicality of our protocol. This allows us to benchmark the additional overhead over SPDZ in the time-critical online phase – our MAC checks and checks of the Auth subprotocol – and enables comparisons of the overall runtime for concrete use cases. Our implementation uses GPU acceleration and/or multi-threading to speed up the computationally intensive local operations in the offline phase and the verification phase. We also built a LowGear-based [33] version of the SPDZ protocol using the same underlying implementation for a fair comparison. For consistent results, we ran the following experiments on a single machine (unless stated otherwise: Intel Core i9-9940X CPU, 14 cores, 3.3 GHz; Nvidia Titan RTX GPU), emulating commonly

Table 4: Runtime for Verifying Authentication and MACs. Times are given in nanoseconds and per party and verified element, i.e., the time per ciphertext slot for computing Lines 12 to 13 of Fig. 4 (i.e., verifying AUTH) and Eq. (7) (i.e., a MAC check), respectively.

			Device			
Verif.	$\log_2 p$	sec-stat	Laptop[a]	HPC-Node[b]	GPU-T[c]	GPU-A[d]
AUTH	64	64	1309.288	211.440	106.545	18.366
AUTH	128	80	3758.054	408.305	283.509	33.712
MAC	64	64	20.760	1.047	0.443	0.192
MAC	128	80	297.831	15.836	2.788	0.515

[a] Intel Core i7-8565U CPU, 4 cores, 1.8 GHz [c] Nvidia Titan RTX GPU
[b] Intel Xeon Gold 6230 CPU, 40 cores, 2.1 GHz [d] Nvidia A100 80 GB GPU

used network setups [33,32,39,41] between the parties: LAN with 10 ms network delay and maximum bandwidth of 1 Gbit/s; and WAN (50 ms delay, 50 Mbit/s bandwidth). Appendix E contains additional figures for our evaluation.

Benchmark: Verifying Authentication and MACs In the online phase, the computational overhead of our protocol compared to SPDZ consists of verifying the AUTH subprotocol and checking MACs. In Tab. 4, we show the runtime for verifying a single drowned ciphertext in the AUTH subprotocol (see Fig. 4) and the runtime for performing a MAC check (see Eq. (7)). As can be seen by the results, the MAC checks are approximately two orders of magnitude faster to verify than the authentication. This means the overhead for clients, which only perform MAC checks if there is no abort, is very low compared to the server overhead. Additionally, all types of verification can be performed (again, up to two orders of magnitude) faster with more capable hardware (faster CPUs or GPUs), leading to only a few nanoseconds of verification time. Combined with the lower communication overhead for inputs/outputs (see Sec. 8.1), our protocol can even outperform SPDZ. We show this after presenting our results for benchmarking general multiplications.

Benchmark: Multiplication Throughput For a more complete picture, we also measured the throughput for multiplications in the online phase, including verification of the AUTH subprotocol and checking MACs (the secure aggregation use case in the next section also considers inputs/outputs). Our protocol achieves up to 595080 multiplications per second in the LAN setting and 133662 multiplications per second in the WAN setting. Our SPDZ implementation achieves 2507579 and 314709 multiplications per second in these settings. See Fig. 9 (in Appendix E) for more details. Our protocol is thus $4.21\times$ and $2.35\times$ slower, respectively. This approaches the communication overhead of $2\times$ that our protocol has over plain non-identifiable SPDZ. Additionally, our overhead is lower than what was observed in prior available benchmarks for protocols with PIA [42]: The overhead of CFY [23] compared to SPDZ is at least $4\times$ and the overhead of RRRK [42] is around 15 to $18\times$. This shows that our protocol compares favorably to protocols that use commitments to achieve IA.

Use Case: Secure Aggregation To demonstrate the practicality of our approach, we implemented[1] a secure aggregation [35] use case with our protocol and SPDZ, which could be used, e.g., in federated learning. For SPDZ, we implemented Damgård et al.'s input protocol [24].[8] As mentioned above, this requires each server to send five field elements per input to each client. The 5-tuple can be computed entirely in the offline phase (as opposed to a naive implementation of [24] that might perform two multiplications in the online phase). Additionally, we only authenticate one component of the tuple in the offline phase and do not check the consistency of the tuple, i.e., we do not check that the multiplicative relations of the 5-tuple are upheld. This gives a lower bound on the runtime for SPDZ, and more checks might not be required as the clients check the 5-tuple. (However, [24] does not include this optimization, and we do not prove security of this optimization here in this paper.) Figure 10 (in Appendix E) shows the detailed results of our experiments.

In the LAN setting, the computational overhead of our protocol leads to a 4 to 22 % slower online phase (overhead in server time). On the other hand, the communication overhead of SPDZ leads to an 18 % slower online phase for SPDZ servers in the WAN setting compared to our protocol, where communication becomes the bottleneck (see Fig. 10b). Both protocols show little difference between the runtime for clients and servers. For SPDZ, we can see that the final MAC check (where the clients do not participate) makes a small but visible difference. For our protocol, the clients have to wait for servers to finish verifying the authentication before the clients can check their MACs. This leads to virtually the same runtime for clients and servers in our protocol, where clients are idle most of the time. When running additional experiments with twice the number of clients, the overall runtime increased by slightly less than a factor of two, confirming that both protocols scale linearly in the number of clients and the number of inputs.

In the offline phase (see Figs. 10c and 10d), generating a (partially authenticated) 5-tuple in SPDZ is more demanding than generating an authenticated share in our protocol. Concretely, SPDZ is 2.27 to $2.30\times$ slower in the LAN setting and 2.30 to $2.33\times$ slower in the WAN setting. Overall, this shows that our protocol is a good candidate for secure outsourced computations. Not only does our protocol achieve stronger security guarantees, namely *publicly identifiable abort*, but we also require fewer resources to prepare inputs in the offline phase, the online overhead is quite small, and our protocol can be even faster than SPDZ in the online phase for communication-bound settings.

[8] used in practice, e.g., in Carbyne Stack (https://github.com/carbynestack/amphora) or MP-SPDZ (https://github.com/data61/MP-SPDZ)

Acknowledgments. This research was partially supported by Advantest as part of the Graduate School "Intelligent Methods for Test and Reliability" (GS-IMTR) at the University of Stuttgart. The authors acknowledge support by the state of Baden-Württemberg through bwHPC. We also thank Andrés Bruhn and Azin Jahedi from the Institute for Visualization and Interactive Systems at the University of Stuttgart for providing the computational resources and assistance with running our experiments. A condensed version of this article is published elsewhere [40].

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Alon, B., Chung, H., Chung, K., Huang, M., Lee, Y., Shen, Y.: Round efficient secure multiparty quantum computation with identifiable abort. In: Crypto 2021. pp. 436–466 (2021). https://doi.org/10.1007/978-3-030-84242-0_16
2. Aranha, D.F., Baum, C., Gjøsteen, K., Silde, T.: Verifiable mix-nets and distributed decryption for voting from lattice-based assumptions. In: CCS 2023. pp. 1467–1481 (2023). https://doi.org/10.1145/3576915.3616683
3. Archer, D.W., Bogdanov, D., Lindell, Y., Kamm, L., Nielsen, K., Pagter, J.I., Smart, N.P., Wright, R.N.: From keys to databases - real-world applications of secure multi-party computation. Comput. J. **61**(12), 1749–1771 (2018). https://doi.org/10.1093/COMJNL/BXY090
4. Aumann, Y., Lindell, Y.: Security against covert adversaries: Efficient protocols for realistic adversaries. J. Cryptol. **23**(2), 281–343 (2010). https://doi.org/10.1007/S00145-009-9040-7
5. Baum, C., Cozzo, D., Smart, N.P.: Using TopGear in Overdrive: A more efficient ZKPoK for SPDZ. In: SAC 2019. pp. 274–302 (2019). https://doi.org/10.1007/978-3-030-38471-5_12
6. Baum, C., Damgård, I., Orlandi, C.: Publicly auditable secure multi-party computation. In: SCN 2014. pp. 175–196 (2014). https://doi.org/10.1007/978-3-319-10879-7_11
7. Baum, C., Melissaris, N., Rachuri, R., Scholl, P.: Cheater identification on a budget: MPC with identifiable abort from pairwise MACs. In: Crypto 2024. pp. 454–488 (2024). https://doi.org/10.1007/978-3-031-68397-8_14
8. Baum, C., Orsini, E., Scholl, P.: Efficient secure multiparty computation with identifiable abort. In: TCC 2016-B. pp. 461–490 (2016). https://doi.org/10.1007/978-3-662-53641-4_18
9. Baum, C., Orsini, E., Scholl, P.: Efficient secure multiparty computation with identifiable abort. Cryptology ePrint Archive, Paper 2016/187 (2016), https://eprint.iacr.org/2016/187
10. Baum, C., Orsini, E., Scholl, P., Soria-Vazquez, E.: Efficient constant-round MPC with identifiable abort and public verifiability. In: Crypto 2020. pp. 562–592 (2020). https://doi.org/10.1007/978-3-030-56880-1_20
11. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Crypto 1991. pp. 420–432 (1991). https://doi.org/10.1007/3-540-46766-1_34
12. Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic encryption and multiparty computation. In: Eurocrypt 2011. pp. 169–188 (2011). https://doi.org/10.1007/978-3-642-20465-4_11

13. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators: Silent OT extension and more. In: Crypto 2019. pp. 489–518 (2019). https://doi.org/10.1007/978-3-030-26954-8_16
14. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: ITCS 2012. pp. 309–325 (2012). https://doi.org/10.1145/2090236.2090262
15. Brandt, N., Maier, S., Müller, T., Müller-Quade, J.: Constructing secure multiparty computation with identifiable abort. Cryptology ePrint Archive, Paper 2020/153 (2020), https://eprint.iacr.org/2020/153
16. Brunetta, C., Tsaloli, G., Liang, B., Banegas, G., Mitrokotsa, A.: Non-interactive, secure verifiable aggregation for decentralized, privacy-preserving learning. In: ACISP 2021. pp. 510–528 (2021). https://doi.org/10.1007/978-3-030-90567-5_26
17. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS 2001. pp. 136–145 (2001). https://doi.org/10.1109/SFCS.2001.959888
18. Catalano, D., Fiore, D.: Practical homomorphic macs for arithmetic circuits. In: Eurocrypt 2013. pp. 336–352 (2013). https://doi.org/10.1007/978-3-642-38348-9_21
19. Chen, H., Kim, M., Razenshteyn, I.P., Rotaru, D., Song, Y., Wagh, S.: Maliciously secure matrix multiplication with applications to private deep learning. In: Asiacrypt 2020. pp. 31–59 (2020). https://doi.org/10.1007/978-3-030-64840-4_2
20. Ciampi, M., Ravi, D., Siniscalchi, L., Waldner, H.: Round-optimal multi-party computation with identifiable abort. In: Eurocrypt 2022. pp. 335–364 (2022). https://doi.org/10.1007/978-3-031-06944-4_12
21. Cohen, R., Doerner, J., Kondi, Y., Shelat, A.: Secure multiparty computation with identifiable abort via vindicating release. In: Crypto 2024. pp. 36–73 (2024). https://doi.org/10.1007/978-3-031-68397-8_2
22. Cramer, R., Damgård, I.: On the amortized complexity of zero-knowledge protocols. In: Crypto 2009. pp. 177–191 (2009). https://doi.org/10.1007/978-3-642-03356-8_11
23. Cunningham, R.K., Fuller, B., Yakoubov, S.: Catching MPC cheaters: Identification and openability. In: ICITS 2017. pp. 110–134 (2017). https://doi.org/10.1007/978-3-319-72089-0_7
24. Damgård, I., Damgård, K., Nielsen, K., Nordholt, P.S., Toft, T.: Confidential benchmarking based on multiparty computation. In: FC 2016. pp. 169–187 (2016). https://doi.org/10.1007/978-3-662-54970-4_10
25. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In: ESORICS 2013. pp. 1–18 (2013). https://doi.org/10.1007/978-3-642-40203-6_1
26. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Crypto 2012. pp. 643–662 (2012). https://doi.org/10.1007/978-3-642-32009-5_38
27. Gehlhar, T., Marx, F., Schneider, T., Suresh, A., Wehrle, T., Yalame, H.: SafeFL: MPC-friendly framework for private and robust federated learning. In: SPW 2023. pp. 69–76 (2023). https://doi.org/10.1109/SPW59333.2023.00012
28. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: STOC 1987. pp. 218–229 (1987). https://doi.org/10.1145/28395.28420
29. Ishai, Y., Ostrovsky, R., Seyalioglu, H.: Identifying cheaters without an honest majority. In: TCC 2012. pp. 21–38 (2012). https://doi.org/10.1007/978-3-642-28914-9_2

30. Ishai, Y., Ostrovsky, R., Zikas, V.: Secure multi-party computation with identifiable abort. In: Crypto 2014. pp. 369–386 (2014). https://doi.org/10.1007/978-3-662-44381-1_21
31. Kanjalkar, S., Zhang, Y., Gandlur, S., Miller, A.: Publicly auditable mpc-as-a-service with succinct verification and universal setup. In: EuroS&P Workshops 2021. pp. 386–411 (2021). https://doi.org/10.1109/EUROSPW54576.2021.00048
32. Keller, M., Orsini, E., Scholl, P.: MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In: CCS 2016. pp. 830–842 (2016). https://doi.org/10.1145/2976749.2978357
33. Keller, M., Pastro, V., Rotaru, D.: Overdrive: Making SPDZ great again. In: Eurocrypt 2018. pp. 158–189 (2018). https://doi.org/10.1007/978-3-319-78372-7_6
34. Küsters, R., Truderung, T., Vogt, A.: Accountability: definition and relationship to verifiability. In: CCS 2010. pp. 526–535 (2010). https://doi.org/10.1145/1866307.1866366
35. Mansouri, M., Önen, M., Jaballah, W.B., Conti, M.: SoK: Secure aggregation based on cryptographic schemes for federated learning. Proc. Priv. Enhancing Technol. **2023**(1), 140–157 (2023). https://doi.org/10.56553/POPETS-2023-0009
36. McMahan, B., Moore, E., Ramage, D., Hampson, S., y Arcas, B.A.: Communication-efficient learning of deep networks from decentralized data. In: AISTATS 2017. pp. 1273–1282 (2017)
37. Mohassel, P., Zhang, Y.: SecureML: A system for scalable privacy-preserving machine learning. In: SP 2017. pp. 19–38 (2017). https://doi.org/10.1109/SP.2017.12
38. Rathee, M., Shen, C., Wagh, S., Popa, R.A.: ELSA: secure aggregation for federated learning with malicious actors. In: SP 2023. pp. 1961–1979 (2023). https://doi.org/10.1109/SP46215.2023.10179468
39. Reisert, P., Rivinius, M., Krips, T., Küsters, R.: Overdrive LowGear 2.0: Reduced-bandwidth MPC without sacrifice. In: ASIA CCS 2023. pp. 372–386 (2023). https://doi.org/10.1145/3579856.3582809
40. Rivinius, M.: MPC with publicly identifiable abort from pseudorandomness and homomorphic encryption. In: Eurocrypt 2025 (2025), (To appear)
41. Rivinius, M., Reisert, P., Hasler, S., Küsters, R.: Convolutions in Overdrive: Maliciously secure convolutions for MPC. Proc. Priv. Enhancing Technol. **2023**(3), 321–353 (2023). https://doi.org/10.56553/POPETS-2023-0084
42. Rivinius, M., Reisert, P., Rausch, D., Küsters, R.: Publicly accountable robust multi-party computation. In: SP 2022. pp. 2430–2449 (2022). https://doi.org/10.1109/SP46214.2022.9833608
43. Silde, T.: Short paper: Verifiable decryption for BGV. In: FC 2022. pp. 381–390 (2022). https://doi.org/10.1007/978-3-031-32415-4_26
44. Simkin, M., Siniscalchi, L., Yakoubov, S.: On sufficient oracles for secure computation with identifiable abort. In: SCN 2022. pp. 494–515 (2022). https://doi.org/10.1007/978-3-031-14791-3_22
45. Spini, G., Fehr, S.: Cheater detection in SPDZ multiparty computation. In: ICITS 2016. pp. 151–176 (2016). https://doi.org/10.1007/978-3-319-49175-2_8
46. Tsaloli, G., Liang, B., Brunetta, C., Banegas, G., Mitrokotsa, A.: DEVA: decentralized, verifiable secure aggregation for privacy-preserving learning. In: ISC 2021. pp. 296–319 (2021). https://doi.org/10.1007/978-3-030-91356-4_16

A MAC Scheme

Here, we show that the MAC scheme presented in Sec. 2.3 is equivalent to the scheme proven secure by Catalano and Fiore [18]. They define the tag as

$$(\!|x|\!)_{\mathsf{ctx}}^{\mathsf{orig},\alpha,\mathsf{fk}} := (x, (\mathsf{PRF}(\mathsf{fk}, \mathsf{ctx}) - x) \cdot \alpha^{-1}) \tag{14}$$

and verification as

$$\begin{aligned} & \mathsf{MACCheck}(\alpha, \mathsf{fk}, x, \mathsf{ctx}, (\!|x|\!)^{\mathsf{orig}}) = 1 \\ \Leftrightarrow\quad & (\!|x|\!)^{\mathsf{orig}}[0] + (\!|x|\!)^{\mathsf{orig}}[1] \cdot \alpha = \mathsf{PRF}(\mathsf{fk}, \mathsf{ctx}) \wedge (\!|x|\!)^{\mathsf{orig}}[0] = x \end{aligned}$$

i.e., $(\!|x|\!)^{\mathsf{orig}}$ can be interpreted as coefficients of a degree-1 polynomial p with $p(0) = x$ and $p(\alpha) = \mathsf{PRF}(\mathsf{fk}, \mathsf{ctx})$. Despite superficial differences, our construction is equivalent to this.

Theorem 2. *Our MAC scheme from Sec. 2.3 is equivalent to the scheme of Catalano and Fiore [18].*

Proof. First, note that omitting the message part from the MAC is a trivial change and does not change anything as our definition of verification always requires the message in addition to the MAC tag.

Second, as we require the MAC key α to be invertible, our scheme is equivalent to the above scheme with MAC key $\beta := -\alpha^{-1}$ and PRF $\mathsf{PRF}'(\mathsf{fk}, \mathsf{ctx}) := \beta \cdot \mathsf{PRF}(\mathsf{fk}, \mathsf{ctx})$. Let $(\!|x|\!)_{\mathsf{ctx}}^{\mathsf{orig}',\beta,\mathsf{fk}}$ be the MAC as in Eq. (14) but with PRF' instead of PRF and with keys β, fk. Then,

$$\begin{aligned} (\!|x|\!)_{\mathsf{ctx}}^{\mathsf{orig}',\beta,\mathsf{fk}}[1] &= (\mathsf{PRF}'(\mathsf{fk}, \mathsf{ctx}) - x) \cdot \beta^{-1} \\ &= \mathsf{PRF}'(\mathsf{fk}, \mathsf{ctx}) \cdot \beta^{-1} - x \cdot \beta^{-1} \\ &= \mathsf{PRF}(\mathsf{fk}, \mathsf{ctx}) + x \cdot \alpha \\ &= (\!|x|\!)_{\mathsf{ctx}}^{\alpha,\mathsf{fk}}. \end{aligned}$$

Therefore, our MAC scheme is equivalent to the above scheme with a different key β and a slightly altered PRF PRF' (which is still pseudorandom if the original PRF is a pseudorandom function).

Another difference in the definitions is that Catalano and Fiore define evaluation of whole (linear) functions on tags, compared to our step-by-step evaluation of linear functions via Eq. (5). This change can be trivially shown to be equivalent as well. $\square$

B Protocols and Functionalities

In addition to a functionality for communication $\mathcal{F}_{\text{comm}}$ that we handle transparently (see Sec. 3), our protocol uses a few (standard) functionalities and subprotocols that we omitted in the main body of this paper and that we describe in more detail now. Firstly, we assume there is a setup functionality $\mathcal{F}_{\text{setup}}$

that generates valid BGV keys and distributes these among the parties. This includes the public key / private key pairs $(\mathsf{pk}_i, \mathsf{sk}_i)$, as well as the shared public key pk for which all parties have a share $[\mathsf{sk}]_i$ of the private key. This is used in the distributed decryption subprotocol DistDec (not shown here). The latter simply runs a verifiable distributed decryption algorithm from the literature, where parties perform their part of the decryption and prove in zero-knowledge that they performed it correctly [42,43,2]. This is a subprotocol with identifiable abort and standard for several protocols with identifiable abort [8,23,42]. Another component of $\mathcal{F}_{\text{setup}}$ is that it allows parties to commit and decommit to values. (This could also be handled by a separate commitment functionality.)

```
1: procedure Enc-ZK($[\boldsymbol{r}]_i$)
2:     $\langle[\boldsymbol{r}]_i\rangle_i \xleftarrow{\$} \mathsf{Enc}(\mathsf{pk}_i, [\boldsymbol{r}]_i)$
3:     Generate a zero-knowledge proof that $[\boldsymbol{r}]_i$ is correctly encrypted and that the
       norm of the plaintext and encryption randomness is *short* (as defined by the
       encryption scheme and the concrete ZKP instantiation). Use either a random
       oracle to produce a non-interactive proof or interactively generate the challenge
       used in the proof with $\mathcal{F}_{\text{rand}}$ (see, e.g., Fig. 6). Let $\pi_i$ be an encoding of the
       ZKP (that includes $\langle[\boldsymbol{r}]_i\rangle_i$).
4:     Broadcast $\pi_i$
5:     // Verification (can also be done by external parties):
6:     for $j$ from $0$ to $n-1$ do
7:         Verify $\pi_j$ and identify $P_j$ as cheater if the verification failed
8:     return $(\langle[\boldsymbol{r}]_j\rangle_j)_{j=0}^{n-1}$
9: procedure PubEnc-ZK($[\boldsymbol{r}]_i$)
10:    $\langle[\boldsymbol{r}]_i\rangle_i \xleftarrow{\$} \mathsf{Enc}(\mathsf{pk}_i, [\boldsymbol{r}]_i)$
11:    $\langle\!\langle[\boldsymbol{r}]_i\rangle\!\rangle \xleftarrow{\$} \mathsf{Enc}(\mathsf{pk}, [\boldsymbol{r}]_i)$
12:    Generate a zero-knowledge proof that $[\boldsymbol{r}]_i$ is correctly encrypted and that the
       norm of the plaintext and encryption randomness is *short* (as above); addition-
       ally prove that $\langle\!\langle\boldsymbol{r}\rangle\!\rangle \coloneqq \sum_{j=0}^{n-1}\langle\!\langle[\boldsymbol{r}]_j\rangle\!\rangle$ is a valid ciphertext with short norm (note
       that the latter is not an $n$-party proof as in HighGear [33] or in TopGear [5] be-
       cause we perform checks for each party; see also Fig. 6). Let $\pi_i$ be an encoding
       of the ZKP (that includes $\langle[\boldsymbol{r}]_i\rangle_i$ and $\langle\!\langle[\boldsymbol{r}]_i\rangle\!\rangle$).
13:    Broadcast $\pi_i$
14:    // Verification (can also be done by external parties):
15:    for $j$ from $0$ to $n-1$ do
16:        Verify $\pi_j$ and identify $P_j$ as cheater if the verification fails
17:    return $(\langle[\boldsymbol{r}]_j\rangle_j)_{j=0}^{n-1}, \langle\!\langle\boldsymbol{r}\rangle\!\rangle$
```

Fig. 5: Zero-Knowledge Protocol at Party P_i

In the offline phase, we let parties prove in zero-knowledge that they correctly encrypted their shares, prove knowledge of these shares, and prove that the ciphertext was generated with small plaintexts/randomness, which implies that the noise of the ciphertext is bounded. Protocols for this are shown in Fig. 5.

A TopGear-style (interactive) instantiation is shown in Fig. 6. These protocols might use the (standard) functionality to generate public randomness $\mathcal{F}_{\text{rand}}$, which is used for interactive zero-knowledge proofs. $\mathcal{F}_{\text{rand}}$ samples the required amount of random bits (or field/ring elements, depending on the usage) and outputs these to all parties. Finally, the detailed simulator for our security proof (Sec. 7) can be found in Fig. 7.

C Possible Protocol Modifications

If we want to deploy the protocol in a setup where no inputs from external clients are required, i.e., only compute parties give inputs to the protocol, one can modify the offline phase and online phase as in Fig. 8. In the modified share authentication AuthInputShareFrom, the compute party P_j that is supposed to give the input later choses the shares for all parties. This way, in the online phase (InputFrom), P_j already knows the mask that will be used for the input, eliminating some communication that would be necessary with the client-server centric design presented before. Note that this approach can also be used if only *some* inputs come from compute parties, i.e., it can be combined with Input and AuthInputShare from Figs. 3 and 4.

D Parameter Estimation

Table 5 shows the parameters for our linear BGV instantiation, used for pairwise operations in the Auth subprotocol (see Sec. 5.1). The parameters are similar to the ones used for Overdrive LowGear [33], where the difference in parameters is due to our use of TopGear ZKPs [5]. The latter requires fewer auxiliary ciphertexts per proven statement than older amortization techniques [22,26] used in LowGear (see Tab. 5). This means we send around half as many ciphertexts per amortized proof.

Table 5: Ciphertext Parameters. Ciphertext modulus q is given depending on plaintext modulus p, number of ciphertext slots N,* and statistical/ computational security parameters. U, V are ZKP parameters: We prove a statement for U ciphertexts simultaneously, using V auxiliary ciphertexts. We use the following security parameters for zero-knowledge and soundness: $\textsf{sec-zk} := \textsf{sec-stat}, \textsf{sec-so} := \textsf{sec-comp}$.

$\log_2 p$	$\log_2 N$	sec-stat	sec-comp	U	V	$\log_2 q$
64	16	64	128	16	8	279
128	16	80	128	16	8	440
128	16	128	128	16	8	536

* N is chosen as in Overdrive [33]; TopGear [5] uses smaller values

Below, we highlight parts that are only required for PubEnc-ZK with $\boxed{\text{boxes}}$. Additionally, we only describe the interactive variant of the proof. Let $C \coloneqq \{0\} \cup \{X^k \mid 0 \leq k < 2N = m\}$ be the polynomials of the challenge space.

```
1: procedure ZK($\boldsymbol{x}_i \in R^U$, $\langle \boldsymbol{x}_i \rangle_i$, $\boxed{\langle\!\langle \boldsymbol{x}_i \rangle\!\rangle}$)
2:     Broadcast $\langle \boldsymbol{x}_i \rangle_i$ $\boxed{\text{and } \langle\!\langle \boldsymbol{x}_i \rangle\!\rangle}$
3:     // Commitment phase:
4:     Let $\mathbf{R}_i \in R^{U \times 3}$ be the randomness used for $\langle \boldsymbol{x}_i \rangle_i$
5:     $\boxed{\text{Let } \mathbf{R}'_i \in R^{U \times 3} \text{ be the randomness used for } \langle\!\langle \boldsymbol{x}_i \rangle\!\rangle}$
6:     Sample $\boldsymbol{y}_i \in R^V$ and $\mathbf{S}_i \in R^{V \times 3}$ with norms that are $2^{\mathsf{sec\text{-}zk}}$ larger than the
       upper bounds for $\boldsymbol{x}_i$ and $\mathbf{R}_i$, respectively
7:     Compute and broadcast $\mathbf{A}_i \coloneqq \mathsf{Enc}(\mathsf{pk}_i, \boldsymbol{y}_i, \mathbf{S}_i)$
8:     Sample $\boldsymbol{y}'_i \in R^V$ and $\mathbf{S}'_i \in R^{V \times 3}$ with norms that are $2^{\mathsf{sec\text{-}zk}}$ larger than the
       upper bounds for $\boldsymbol{x}_i$ and $\mathbf{R}'_i$, respectively
9:     Compute and broadcast $\mathbf{A}'_i \coloneqq \mathsf{Enc}(\mathsf{pk}, \boldsymbol{y}'_i, \mathbf{S}'_i)$
10:    // Challenge phase:
11:    Use $\mathcal{F}_{\text{rand}}$ to sample $\mathbf{W}_j \in C^{V \times U}$ for all $P_j \in \mathcal{P}$
       $\boxed{\text{and additionally a common } \mathbf{W} \in C^{V \times U}}$
12:    // Response phase:
13:    Compute and broadcast $\boldsymbol{z}_i \coloneqq \boldsymbol{y}_i + \mathbf{W}_i \boldsymbol{x}_i$ and $\mathbf{T}_i \coloneqq \mathbf{S}_i + \mathbf{W}_i \mathbf{R}_i$
14:    $\boxed{\text{Compute and broadcast } \boldsymbol{z}'_i \coloneqq \boldsymbol{y}'_i + \mathbf{W} \boldsymbol{x}_i \text{ and } \mathbf{T}'_i \coloneqq \mathbf{S}'_i + \mathbf{W} \mathbf{R}'_i}$
15:    // Verification phase:
16:    for $j$ from $0$ to $n-1$ do
17:        Compute $\mathbf{D}_j \coloneqq \mathsf{Enc}(\mathsf{pk}_j, \boldsymbol{z}_j, \mathbf{T}_j)$
18:        Check that $\mathbf{D}_j = \mathbf{A}_j + \mathbf{W}_j \langle \boldsymbol{x}_j \rangle_j$
19:        Check that the norms of $\boldsymbol{z}_j$ and $\mathbf{T}_j$ are at most $2 \cdot 2^{\mathsf{sec\text{-}zk}}$ larger than the
           upper bounds for $\boldsymbol{x}_i$ and $\mathbf{R}_i$, respectively
20:        Compute $\mathbf{D}'_j \coloneqq \mathsf{Enc}(\mathsf{pk}, \boldsymbol{z}'_j, \mathbf{T}'_j)$
21:        Check that $\mathbf{D}'_j = \mathbf{A}'_j + \mathbf{W} \langle\!\langle \boldsymbol{x}_j \rangle\!\rangle$
22:        Check that the norms of $\boldsymbol{z}'_j$ and $\mathbf{T}'_j$ are at most $2 \cdot 2^{\mathsf{sec\text{-}zk}}$ larger than the
           upper bounds for $\boldsymbol{x}_i$ and $\mathbf{R}'_i$, respectively
23:        If any check failed, identify $P_j$ as cheater
24:    $\boxed{\text{Let } \langle\!\langle \boldsymbol{x} \rangle\!\rangle \coloneqq \sum_{j=0}^{n-1} \langle\!\langle \boldsymbol{x}_j \rangle\!\rangle}$
```

Fig. 6: TopGear [5] ZKP Protocol at Party P_i

```
In the following, our simulator $\mathcal{S}$ interacts with adversary-controlled parties $\mathcal{C}$. For
each honest $P_{\text{honest}} \in (\mathcal{P} \cup \mathcal{I} \cup \mathcal{O}) \setminus \mathcal{C}$, a simulated instance is generated that acts
just like in the protocol, except with the differences noted below.
 1: Setup:
 2:     Run the setup phase of the protocol but with a simulator-controlled instance
        of $\mathcal{F}_{\text{setup}}$ that gives $\mathcal{S}$ access to the private keys for all parties.
 3:     Extract $[\alpha]_i$, $\mathsf{fk}_i$, $\mathsf{gk}_i$ for all $P_i \in \mathcal{P} \cap \mathcal{C}$ using controlled commitment function-
        ality. Do the same for $[\omega]_i$, $\mathsf{fk}'_i$, $\mathsf{gk}'_i$.
 4: Preprocessing:
 5:     Simulate the offline phase of the protocol and add each $P_i \in \mathcal{P}$ to a set $\mathcal{M}$ for
        which the verification of some instance of ENC-ZK, PUBENC-ZK, or DISTDEC
        fails.
 6:     Obtain and store all shares of parties by decrypting $(\langle [r]_i \rangle_i)_{P_i \in \mathcal{P}}$ (and analo-
        gously for encrypted shares that will be used for Beaver triples or outputs).
 7:     Send $\mathcal{M}$ to $\mathcal{F}_{\text{MPC}}$.
 8: Computation:
 9:     If the protocol aborted above ($\mathcal{M} \neq \emptyset$ in the previous phase), skip the following
        steps.
10:     Simulate the input phase for honest parties $P_{\text{honest}} \in \mathcal{I} \setminus \mathcal{C}$ but set their input
        values to zero.
11:     For every input $x$ of a corrupted $P_j \in \mathcal{I} \cap \mathcal{C}$, retrieve the stored shares for the
        mask $r$ that is designated to be used for $P_j$'s input and collect $x' := u + r$
        (where $u$ is the value broadcasted by $P_i$ during INPUT) into a vector $\boldsymbol{x}'_j$.
12:     Send the collected $\boldsymbol{x}'_j$ for all $P_j \in \mathcal{C} \cap \mathcal{I}$ to $\mathcal{F}_{\text{MPC}}$.
13:     Receive $\boldsymbol{y}_{\text{pub}}$ and $\boldsymbol{y}_j$ for all $P_j \in \mathcal{C} \cap \mathcal{O}$ from $\mathcal{F}_{\text{MPC}}$.
14:     Simulate the rest of the online phase by following the protocol but do the
        following for outputs:
        For every public output $y$ (obtained previously from $\mathcal{F}_{\text{MPC}}$ and to be computed
        from $[\![z]\!]^\alpha$ using mask $[\![r]\!]^\alpha$), (i) compute the corresponding output $z$ in the
        simulation (note that the simulator knows the corrupted inputs $\boldsymbol{x}'_j$ and the
        inputs of honest parties in the simulation, which are set to zero), (ii) adjust the
        share $[u]_i$ of one honest $P_i \in \mathcal{P} \setminus \mathcal{C}$ in OUTPUT such that $u = y - r$, i.e., use
        $[z]_i - [r]_i + y - z$ instead of $[z]_i - [r]_i$, (iii) compute $(\!|[u]_i|\!)^\alpha$ using the extracted
        key from the setup and use this when opening $u$, and (iv) continue the rest of
        the protocol normally. Analogously adjust $u$ for private outputs to corrupted
        parties $P_j \in \mathcal{C} \cap \mathcal{O}$ using the $\boldsymbol{y}_j$ obtained from $\mathcal{F}_{\text{MPC}}$.
15:     Simulate the verification phase as in the protocol.
16:     Collect all parties that failed verification in a set $\mathcal{M}$ and send it to $\mathcal{F}_{\text{MPC}}$.
17: Audit:
18:     Do nothing (operations for honest parties are not observable except for the
        output, which is given directly by $\mathcal{F}_{\text{MPC}}$; operations for corrupted parties cannot
        be influenced by the simulator).
```

Fig. 7: Simulator for Our Protocol

```
1: procedure AuthInputShareFrom(Recipient $P_j$, Count $m$)
2:     if partyID $= j$ then
3:         $([\boldsymbol{r}]_k)_{k=0}^{n-1} \xleftarrow{\$} \mathbb{F}^{m \times n}$ // sample shares for all parties
4:         $\boldsymbol{r} := \mathsf{Rec}([\boldsymbol{r}])$ // store for use in online phase
5:         $((\langle[\boldsymbol{r}]_k\rangle_k)_{k=0}^{n-1} \xleftarrow{\$}$ Enc-ZK'$([\boldsymbol{r}]_i)$ // this uses a subprotocol analogous to Fig. 5
           but a single party generates all proofs and all other parties verify them
6:     return Auth$([\alpha]_i, \mathsf{fk}_i, \mathsf{gk}_i, [\boldsymbol{r}]_i, (\langle[\boldsymbol{r}]_k\rangle_k)_{k=0}^{n-1})$ $\boxed{\textit{Verify Auth as in Fig. 4}}$
7: procedure InputFrom(Recipient $P_j$, Input $x$ at $P_j$)
8:     Obtain a mask $[\![r]\!]_i$ from preprocessing
9:     if partyID $= j$ then
10:        Broadcast $u := x - r$ // $P_j$ knows $r$ from preprocessing
11:    return $[\![x]\!]_i := [\![r]\!]_i + (u \cdot \delta_i, 0)$ $\boxed{\textit{Store } \rho_{[r]_i} - \alpha u \delta_i \textit{ as tag randomness of } [\![x]\!]_i}$
```

Fig. 8: Protocol for Compute Party Inputs at Party P_i

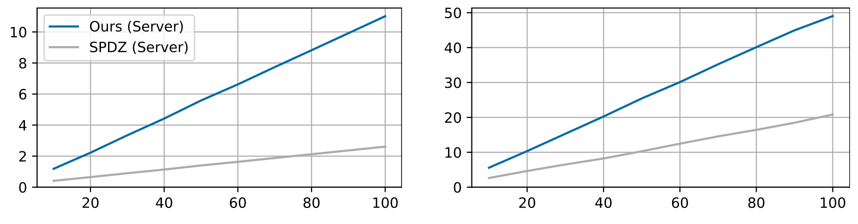
To generate multiplication triples, we use a second instantiation of BGV that is somewhat homomorphic (see Sec. 5.2). The parameters for this depend on the number of compute parties n, as well as the concrete subprotocol used for publicly identifiable distributed decryption. If the identifiable distributed decryption subprotocol does not influence the BGV parameters, we can use the same parameters as in the SPDZ-like TopGear protocol [5], because we perform the same operations on ciphertexts: We compute $\langle\!\langle a \rangle\!\rangle \cdot \langle\!\langle b \rangle\!\rangle - \langle\!\langle r \rangle\!\rangle$ where $\langle\!\langle a \rangle\!\rangle, \langle\!\langle b \rangle\!\rangle, \langle\!\langle r \rangle\!\rangle$ are sums of n ciphertext, each. Otherwise, the ciphertext modulus is at most $\mathsf{sec\text{-}stat}$ bit larger than with TopGear (e.g., via [42,43,2]).

E Evaluation

First note that all results presented in Sec. 8 and here were obtained by running ten experiments, each, and averaging the runtime results. Additionally, we average the runtime over all clients and servers, respectively. Our implementation is uploaded via the Eurocrypt submission server as accompanying code. We plan to publish our implementation on Github once we publish the ePrint version of this paper. Below, we give more detailed results for our practical evaluation from Sec. 8.2.

Benchmark: Multiplication Throughput Figure 9 shows the runtime for our protocol and SPDZ when evaluation many multiplications in the online phase. With SPDZ, we do not perform a MAC check, as this could be amortized with the remaining computations that the parties want to perform in MPC. With our protocol, we perform MAC checks and verify theAuth subprotocol as these operations scale with the number of multiplications. For the largest tested number of multiplications (6 553 600), we achieve the throughput presented in Sec. 8.2: 595 080 and 133 662 multiplications per second (LAN and

WAN, respectively) for our protocol; as well as 2 507 579 and 314 709 multiplications per second (LAN/WAN) for SPDZ.

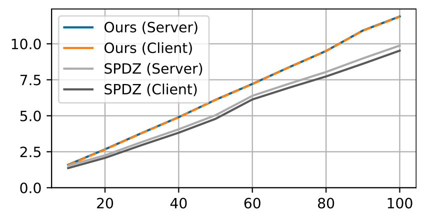


(a) Online, LAN (b) Online, WAN

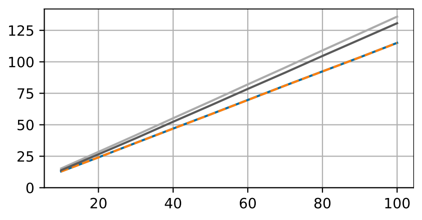
Fig. 9: Runtime for the Multiplication Benchmark in the Online Phase. See also Sec. 8.2. The plots show the runtime in seconds. The number servers is fixed to two. We vary the number of 64 bit multiplications from $10N$ to $100N$ ($N = 65536$).

Use Case: Secure Aggregation In Sec. 8.2, we apply our protocol to secure aggregation, as used, e.g., in federated learning (FL) [36]. FL is a technique to train ML models in a distributed way without performing the entire training with MPC or HE. Instead, clients train ML models locally and aggregate their models after several local training iterations. While constructing a full maliciously secure FL protocol with identifiable abort is out of scope for this work, using our protocol for secure aggregation could be used as the basis of such a protocol. Note that only a few MPC protocols for FL are maliciously secure [35] (exceptions include [16,46,38,27]), let alone secure with identifiable abort. Embedding our MPC protocol in an FL protocol would lead to the first maliciously secure FL protocol with publicly identifiable abort. However, our evaluation only includes running the secure aggregation in MPC, i.e., without training a machine learning locally. Still, our results show that using our protocol can be even more efficient than using standard (non-identifiable) MPC for this purpose.

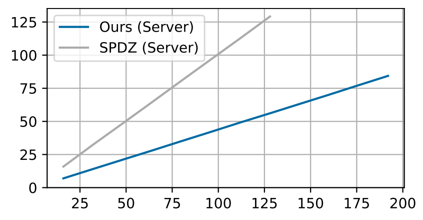
Figure 10 shows the runtime for running our protocol and SPDZ in different network settings and with different problem sizes. We report the runtime for both the online phase and the offline phase. In the online phase, our protocol performs similarly to SPDZ in the LAN setting (Fig. 10a) and even outperforms SPDZ in the WAN setting (Fig. 10b). This is due to our input subprotocol that requires the parties to send less data. In the offline phase, our protocol is faster than SPDZ in both network settings (Figs. 10c and 10d). Here, our protocol outperforms SPDZ because our correlated randomness for inputs is simpler to generate than what is used in input protocols for SPDZ [24].



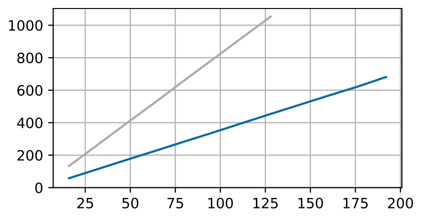
(a) Online, LAN



(b) Online, WAN



(c) Offline, LAN



(d) Offline, WAN

Fig. 10: Runtime for Secure Aggregation. The plots show the runtime in seconds. The number of clients and servers is fixed to two, each. For the online phase, we vary the number of 64 bit inputs at each client from $10N$ to $100N$ ($N = 65536$). Note, the client and server time is virtually the same for our protocol (dashed lines for better visibility). In the offline phase, we vary the number of ciphertexts from 16 to 196, each generating preprocessing material for $N = 65536$ inputs.