MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer

Marcel Keller Emmanuela Orsini
Peter Scholl
Department of Computer Science, University of Bristol
{m.keller,emmanuela.orsini,peter.scholl}@bristol.ac.uk

Abstract

We consider the task of secure multi-party computation of *arithmetic circuits* over a finite field. Unlike Boolean circuits, arithmetic circuits allow natural computations on integers to be expressed easily and efficiently. In the strongest setting of malicious security with a dishonest majority — where any number of parties may deviate arbitrarily from the protocol — most existing protocols require expensive public-key cryptography for each multiplication in the preprocessing stage of the protocol, which leads to a high total cost.

We present a new protocol that overcomes this limitation by using *oblivious transfer* to perform secure multiplications in general finite fields with reduced communication and computation. Our protocol is based on an *arithmetic* view of oblivious transfer, with careful consistency checks and other techniques to obtain malicious security at a cost of less than 6 times that of semi-honest security. We describe a highly optimized implementation together with experimental results for up to five parties. By making extensive use of parallelism and SSE instructions, we improve upon previous runtimes for MPC over arithmetic circuits by more than 200 times.

Keywords: Multi-party computation, oblivious transfer

1 Introduction

Secure multi-party computation (MPC) allows a set of parties to jointly compute a function on their private inputs, learning only the output of the function. In the last decade, MPC has rapidly moved from purely theoretical study to an object of practical interest, with a growing interest in practical applications, and many implementations now capable of handling complex computations [25, 26].

Most MPC protocols either perform secure computation of Boolean circuits, or arithmetic circuits over a finite ring or field such as $\mathbb{F}_p$, for some prime p. Historically, the Boolean circuit approach has led to fast protocols that mostly need only symmetric cryptography, such as two-party protocols based on Yao's garbled circuits [37], or protocols based on fast oblivious transfer techniques [27, 32]. In contrast, protocols for arithmetic circuits are typically based on more expensive, public-key technology (except for special cases when a majority of the parties are honest).

Despite the need for expensive techniques, secret-sharing-based MPC protocols for arithmetic circuits have the key advantage that secure addition requires no communication and essentially come 'for free', whereas with current Boolean circuit-based 2-PC, the only 'free' operation is XOR.

The following motivating examples further highlight the practical applicability of integer-based secure computation, compared with Boolean circuits:

- Bogdanov et al. [7, 8] describe using MPC to perform secure statistical analysis of income tax records for the Estonian government. The latter work analyzed a large database with over 600000 students and 10 million tax records. The kinds of computations involved were very simple statistics, but made heavy use of the fact that secure additions are non-interactive.

- In [12], an application of MPC to *confidential benchmarking* was presented, allowing banks to jointly evaluate customers' risks whilst retaining privacy for the customers' data. They used secure linear programming, which is a highly complex task in MPC, requiring either secure floating point arithmetic or very large integer arithmetic (to emulate real numbers without overflow), both of which would be impractical using Boolean circuits.

- MPC has been suggested as a tool for helping prevent collisions between satellites, by securely performing collision detection using sensitive location and trajectory data. Kamm et al. [23] showed how to implement the relevant conjunction analysis algorithms in MPC with a protocol based on secret-sharing. This also requires secure floating point operations.

Unfortunately, all of the above case studies are somewhat limited, in either the security properties obtained, or the efficiency. The first and third examples above used the Sharemind system [1], which is restricted to semi-honest security with three parties, where at most one is corrupt. The second example used the SPDZ MPC protocol [15], which has security against any number of maliciously corrupted parties, but is much slower. They report a fairly quick evaluation time of around 20–30 s with a prototype implementation, but this does not include the costly 'preprocessing' stage required in SPDZ, which would likely take several hours.

We conclude that although these applications are practical, the MPC protocols used still fall short: in many real-world applications, semi-honest adversaries and an honest majority are not realistic assumptions, and MPC may not be cost-effective if it requires several hours of heavy computation.

Furthermore, it is the case that *all* known practical protocols for MPC with integer operations either require an honest majority, or expensive public-key techniques for every multiplication in the circuit. For example, the SPDZ protocol [14, 15] mentioned above uses a *somewhat homomorphic encryption* scheme to perform secure multiplications, whilst the BDOZ protocol [5] uses additively homomorphic encryption, and both of these require expensive zero-knowledge proofs or cut-and-choose techniques to achieve security against malicious adversaries.

These protocols mitigate this cost to an extent by restricting the expensive computation to a *preprocessing phase*, which is independent of the inputs and can be done in advance. Although this is highly effective for reducing the *latency* of the secure computation — as the online phase is indeed very efficient — the *total cost* of these protocols can still be thousands of times greater than the online phase, which may render them ineffective for many applications.

Frederiksen et al. [17] recently showed how to efficiently use oblivious transfer to generate multiplication triples — the main task of the SPDZ preprocessing — in binary fields, and estimated much improved performance, compared with previous methods. However, this does not give the benefits of general arithmetic circuits that allow integer operations.

1.1 Our contributions

In this paper, we present MASCOT: a new MPC protocol designed to overcome the above limitations of the preprocessing phase, allowing for efficient, secure computation of general arithmetic circuits using almost exclusively fast, symmetric cryptography.

Protocol	Field	Comms. (kbit)	Throughput, $n = 2$ (/s)
SPDZ (active)	$\mathbb{F}_p$, 128-bit	$215n(n-1)$	23.5
	$\mathbb{F}_{2^{40}}$	$2272n(n-1)$	3.68
SPDZ (covert, pr. 1/10)	$\mathbb{F}_p$, 128-bit	$66n(n-1)$	204
	$\mathbb{F}_{2^{40}}$	$844n(n-1)$	31.9
Ours (active)	$\mathbb{F}_p$, 128-bit	$180n(n-1)$	**4842**
	$\mathbb{F}_{2^{128}}$	$180n(n-1)$	**4827**

Table 1: Comparing the cost of n-party secure multiplication in our OT-based protocol with previous implementations of SPDZ [13, 14].

Arithmetic-circuit MPC from OT. We present the first practical protocol for secure multiparty computation of arithmetic circuits based on oblivious transfer (OT), in the dishonest majority setting. We achieve this by taking an "arithmetic" view of OT, which allows us to generalize the preprocessing protocol by Frederiksen et al. [17] to create multiplication triples in any (sufficiently large) finite field, instead of just binary fields. We achieve security against malicious adversaries using simple consistency checking and privacy amplification techniques, with the result that our maliciously secure protocol is only 6 times less efficient than a semi-honest version of the protocol. Moreover, our protocol can be based entirely on symmetric primitives, after a one-time setup phase, by using efficient OT extensions [22, 24].

Implementation. A key advantage of our approach to triple generation is that we obtain a streamlined protocol, which is highly amenable to a parallelized and pipelined implementation that interleaves computation and communication. Table 1 highlights this: the time for a single secure multiplication in a prime field is 200 times faster than the previous best actively secure implementation based on somewhat homomorphic encryption [14], in spite of a fairly small improvement in communication cost. Compared with a covertly secure implementation [1] using SHE [14], our actively secure protocol requires slightly more communication, but still runs over 20 times faster. In binary fields, where SHE is much less suited, the improvement is over 1000 times, compared to previous figures [13]. Note that the online phase of our protocol is identical to that of SPDZ, which has been previously reported to achieve very practical performance for a range of applications [25].

Our optimized implementation utilizes over 80% of the network's capacity, whereas the previous schemes based on SHE are so computation-intensive that the network cannot come close to capacity. We also describe new techniques for reducing the cost of OT extension using consumer hardware instructions, namely efficient matrix transposition using SSE instead of Eklundh's algorithm, and hashing using the Matyas–Meyer–Oseas construction from any block cipher, which

[1] For $\mathbb{F}_{2^{40}}$ in SPDZ with covert security, we could not find precise figures so the throughput in Table 1 is estimated based on other results.

allows hashing 128-bit messages with AES-NI whilst avoiding a re-key for every hash. With all of these optimizations, our protocol outperforms previous protocols by 2–3 orders of magnitude.

More general assumptions. On a theoretical level, we also improve upon previous works by allowing a much wider variety of cryptographic assumptions, since we only require a secure OT protocol, which can be built from DDH, quadratic residuosity or lattices [33]. In contrast, security of the SHE scheme used in SPDZ is based on the ring learning with errors assumption, which is still relatively poorly understood — it is possible that new attacks could surface that render the protocol totally impractical for secure parameters. So as well as increasing efficiency, we obtain much greater confidence in the security of our protocol, and it seems more likely to withstand the test of time.

Improvements over Frederiksen et al. As well as a more general protocol, we obtain several advantages compared with the previous method for MPC in binary fields [17]. We describe an optimized sacrifice procedure for checking correctness of triples, and a new, simpler security analysis, which put together, reduce communication costs by one third. Also, we present a complete MPC protocol, rather than just the method for triple generation; this leads to some technical challenges for verifying correctness of the method for secret-sharing parties' inputs, which was left open in [17]. We resolve this with an additional consistency check, requiring some careful analysis in the security proof.

1.2 Related work

Aside from the works already mentioned, many other secure computation protocols use oblivious transfer. Protocols based on GMW [2, 19] and TinyOT [9, 27, 32] use OT extensions for efficient MPC on binary circuits, and fast garbled circuit protocols use OT extensions in the input stage of the protocol [28]. Pinkas et al. [34, 35] used OT extensions to achieve a very efficient and scalable protocol for the dedicated application of private set intersection. We also note that in 1999, Gilboa used OT for secret-shared multiplication in semi-honest two-party RSA key generation [18], similarly to our high-level approach.

Comparison with recent works. Recently, two independent works have appeared that also consider generating multiplication triples in SPDZ. Baum et al. [3] described improvements to the 'sacrifice' step and the zero-knowledge proofs used with somewhat homomorphic encryption. Their sacrifice technique requires generating triples that form codewords, which does not seem straightforward with our protocol. Their zero-knowledge proofs improve upon the method by Damgård et al. [14] by roughly a factor of two, but our protocol still performs much faster.

Damgård et al. [16] consider building secure multiplication for MPC in large (e.g. 512 or more bits) finite fields out of multiplication for small fields. Their techniques are complementary to ours: our OT-based protocol can be used to perform the small field multiplications in their protocol, thus obtaining more efficient multiplication in large fields (and avoiding the quadratic scaling of our protocol in the field size).

2 Preliminaries

In this section, we describe the security model, introduce some important notation, define the oblivious transfer primitive, and give a basic overview of the SPDZ protocol.

Security model. We prove our security statements in the universal composition (UC) framework of Canetti [10]. Our protocols assume n parties from the set $\mathcal{P} = \{P_1, \ldots, P_n\}$, and we consider security against malicious, static adversaries, i.e. corruption may only take place before the protocols start, corrupting up to $n-1$ of n parties. We let λ and κ denote the computational and statistical security parameters, respectively, and use the standard notions of negligible and overwhelming probability with respect to a security parameter.

Notation. The protocols we present in this paper work in both $\mathbb{F}_p$, for prime $p = 2^k + \mu$, and $\mathbb{F}_{2^k}$, we require some new notation to unify the two finite fields. First note that if $k \geq \kappa$, for statistical security parameter κ, and $\mu \in \mathsf{poly}(k)$ then with overwhelming probability a random element of $\mathbb{F}_p$ can be represented with k bits in $\{0,1\}$, and likewise for any element of $\mathbb{F}_{2^k}$. Let $\mathbb{F}$ denote the finite field, which will be either $\mathbb{F}_p$ or $\mathbb{F}_{2^k}$, and write $\mathbb{F}_{2^k} \cong \mathbb{F}_2[X]/f(X)$ for some monic, irreducible polynomial $f(X)$ of degree k. We use lower case letters to denote finite field elements and bold lower case letters for vectors in $\mathbb{F}$, for any finite field $\mathbb{F}$. If $\mathbf{x}, \mathbf{y}$ are vectors over $\mathbb{F}$, then $\mathbf{x} * \mathbf{y}$ denotes the component-wise products of the vectors. We denote by $a \xleftarrow{\$} A$ the uniform sampling of a from a set A, and by $[d]$ the set of integers $\{1, \ldots, d\}$.

Following notation often used in lattice-based cryptography, define the 'gadget' vector $\mathbf{g}$ consisting of the powers of two (in $\mathbb{F}_p$) or powers of X (in $\mathbb{F}_{2^k}$), so that

$$\mathbf{g} = (1, g, g^2, \ldots, g^{k-1}) \in \mathbb{F}^k,$$

where $g = 2$ in $\mathbb{F}_p$ and $g = X$ in $\mathbb{F}_{2^k}$. Let $\mathbf{g}^{-1} : \mathbb{F} \to \{0,1\}^k$ be the 'bit decomposition' function that maps $x \in \mathbb{F}$ to a bit vector $\mathbf{x}_B = \mathbf{g}^{-1}(x) \in \{0,1\}^k$, such that $\mathbf{x}_B$ can be mapped back to $\mathbb{F}$ by taking the inner product $\langle \mathbf{g}, \mathbf{g}^{-1}(x) \rangle = x$. These basic tools allow us to easily switch between field elements and vectors of bits whilst remaining independent of the underlying finite field.

Oblivious Transfer. Oblivious transfer (OT) is a protocol between a sender and a receiver, where the sender transmits one of several messages to the receiver, whilst remaining oblivious to which message was sent. All known constructions of OT require public-key cryptography, but in 2003, Ishai et al. [22] introduced the concept of OT extensions, where cheap, symmetric primitives (often available in consumer hardware) are used to produce many OTs from only a few.

Recently, Keller et al. [24] presented a simple consistency check that allows maliciously secure OT extension at essentially no extra cost: the cost for a single OT on random strings is almost that of computing two hash function evaluations and sending one string.

The ideal functionality for a single 1-out-of-2 oblivious transfer on k-bit strings is specified as follows, along with the random OT variant, where the sender's messages are sampled at random:

$$\begin{aligned} \mathcal{F}_{\mathsf{OT}}^{1,k} : ((s_0, s_1), b) &\mapsto (\perp, s_b) \\ \mathcal{F}_{\mathsf{ROT}}^{1,k} : (\perp, b) &\mapsto ((r_0, r_1), r_b), \end{aligned}$$

where $r_0, r_1 \xleftarrow{\$} \{0,1\}^k$, and $b \in \{0,1\}$ is the receiver's input. We use the notation $\mathcal{F}_{\mathsf{OT}}^{m,k}, \mathcal{F}_{\mathsf{ROT}}^{m,k}$ to denote m sets of oblivious transfers on k-bit strings.

2.1 The SPDZ Protocol

The online phase of SPDZ [14, 15] uses additive secret sharing over a finite field, combined with information-theoretic MACs to ensure active security. A secret value $x \in \mathbb{F}$ is represented by

$$[\![x]\!] = (x^{(1)}, \ldots, x^{(n)}, m^{(1)}, \ldots, m^{(n)}, \Delta^{(1)}, \ldots, \Delta^{(n)}),$$

where each party P_i holds the random share $x^{(i)}$, the random MAC share $m^{(i)}$ and the fixed MAC key share $\Delta^{(i)}$, such that the MAC relation $m = x \cdot \Delta$ holds, for

$$x = \sum_i x^{(i)}, \quad m = \sum_i m^{(i)}, \quad \Delta = \sum_i \Delta^{(i)}$$

over $\mathbb{F}$.

When opening a shared value $[\![x]\!]$, parties first broadcast their shares $x^{(i)}$, and then run the MAC checking protocol, Π_{MACCheck}, in Protocol 2, which verifies the MAC relation whilst keeping the global MAC key Δ secret, ensuring that values cannot be opened incorrectly except by guessing the key Δ.

The main task of the SPDZ preprocessing phase is to produce the following types of random, authenticated shared values:

> **Input** P_i**:** $([\![r]\!], i)$ a random, shared value r, such that only party P_i knows the value r.
>
> **Triple:** $([\![a]\!], [\![b]\!], [\![c]\!])$ for uniformly random a, b, with $c = a \cdot b$.

In the online phase, parties interact and use the **Input** values to create shared representations of their private inputs, and the **Triple** values to perform multiplications on secret-shared values. Note that since the $[\![\cdot]\!]$ representation is linear, additions and linear functions can be computed locally.

3 Arithmetic correlated oblivious transfer (or Correlated oblivious product evaluation (COPE))

In this section we describe an arithmetic generalization of the passively secure OT extension of Ishai et al. [22], which we call *correlated oblivious product evaluation* (COPE). This allows two parties to obtain an additive sharing of the product $x \cdot \Delta$, where one party holds $x \in \mathbb{F}$ and the other party holds $\Delta \in \mathbb{F}$. The correlation, Δ, is fixed at the start of the protocol, and then future iterations create sharings for different values of x.

Oblivious product evaluation. The key mechanism behind our protocol Π_{COPEe} (Protocol 1) is a general method for (possibly non-correlated) oblivious product evaluation, which is illustrated for $\mathbb{F}_p$ in Fig. 1, and also used in our triple generation protocol later. The two parties run k sets of OTs on k-bit strings, where in each OT the sender, P_S, inputs a random value $t_i \xleftarrow{\$} \mathbb{F}$ and the correlated value $t_i + a$, where $a \in \mathbb{F}$ is the sender's input. The receiver inputs the bit decomposition of their input, $(b_1, \ldots, b_k) \in \{0,1\}^k$, and receives back either t_i or $t_i + a$, depending on the bit b_i. Since the sender's correlation is computed over $\mathbb{F}$, we have the relation

$$q_i = t_i + b_i \cdot a,$$

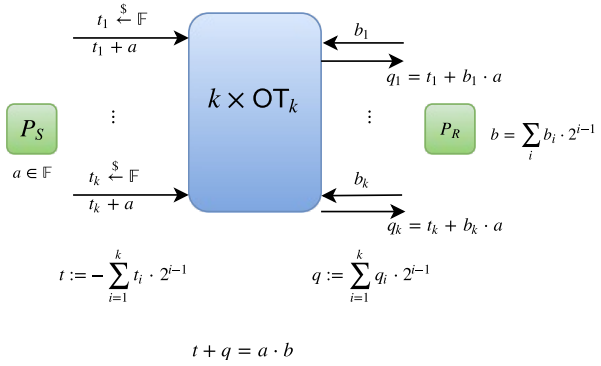

Figure 1: Two-party secret-shared multiplication in $\mathbb{F}_p$ using 1-out-of-2 OT

where q_i is the receiver's output in the i-th OT. Now both parties simply compute the inner product of their values $(q_i)_i, (-t_i)_i$ with the gadget vector $\mathbf{g}$ to obtain values q and t which form an additive sharing of the product of the inputs, so that

$$q + t = a \cdot b \in \mathbb{F}.$$

Correlated OPE. To obtain COPE, where one party's input is fixed for many protocol runs, we only need to perform the k OTs once, where the receiver, P_B, inputs their bits of $\Delta \in \mathbb{F}$ and the sender, P_A, inputs k pairs of random λ-bit seeds (recall that λ is the computational security parameter and $k = \lfloor \log |\mathbb{F}| \rfloor$). This is the **Initialize** phase of the protocol.

After initialization, on each **Extend** call the parties expand the original seeds to create k bits of fresh random OTs, with the same receiver's choice bits $\mathbf{\Delta}_B$. Party P_A now creates a correlation between the two sets of PRG outputs (steps 1–2) using their input, x. The masked correlation is sent to P_B, who uses this to adjust their PRG output accordingly; now both parties have k correlated OTs on field elements. These are then mapped into a single field element by taking the inner product of their outputs with the gadget vector $\mathbf{g}$ to obtain an additive sharing of the product $x \cdot \Delta$ over $\mathbb{F}$ in steps 5–7.

Malicious behavior. Now consider what happens in Π_{COPEe} if the parties do not follow the protocol. Party P_B fixes their input Δ at the start of the protocol, and sends no more messages thereafter, so cannot possibly cheat. On the other hand, P_A may use different values of x in each u^i that is sent in step 2 of **Extend**. Suppose a corrupt P_A uses x^i to compute u^i, for $i \in [k]$, then in step 4 we will instead have

$$\mathbf{q} = \mathbf{t} + \mathbf{x} * \mathbf{\Delta}_B,$$

where $\mathbf{x} = (x^1, \ldots, x^k)$. We do not prevent this in our protocol, but instead model this behavior in the functionality $\mathcal{F}_{\mathsf{COPEe}}$ (Fig. 2). When we use $\mathcal{F}_{\mathsf{COPEe}}$ in our authentication protocol, the main challenge is to ensure that any cheating here can be easily and securely detected by all parties.

The proof of security for our protocol, showing that it securely implements $\mathcal{F}_{\mathsf{COPEe}}$ in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model, is straightforward, since any deviations by an adversary in the protocol directly

Protocol 1 The protocol Π_{COPEe}: Oblivious correlated product evaluation with errors over the finite field $\mathbb{F}$.

The protocol uses an arbitrary output length PRG $G : \{0,1\}^\lambda \to \{0,1\}^*$. Note that after initialization, **Extend** may be called multiple times.

Initialize: This initializes the finite field $\mathbb{F}$ and P_A's correlation, Δ.

1: P_A samples k pairs of seeds, $\{(\mathbf{k}_0^i, \mathbf{k}_1^i)\}_{i=1}^k$, each in $\{0,1\}^\lambda$.
2: P_B inputs $\Delta \in \mathbb{F}$, let $\mathbf{\Delta}_B = (\Delta_0, \ldots, \Delta_{k-1}) \in \{0,1\}^k \subset \mathbb{F}^k$.
3: Both parties call $\mathcal{F}_{\mathsf{OT}}^{k,\lambda}$ with inputs $\{\mathbf{k}_0^i, \mathbf{k}_1^i\}_{i\in[k]}$ and $\mathbf{\Delta}_B$.
4: P_B receives $\mathbf{k}_{\Delta_i}^i$ for $i \in [k]$.

Extend: On input $x \in \mathbb{F}$ from P_A:

1: Expand $\mathbf{k}_0^i$ and $\mathbf{k}_1^i$ using the PRG in a stateful way, that is, $G(\mathbf{k}_0^i)$ creates fresh randomness in every call. Obtain

$$t_0^i = G(\mathbf{k}_0^i) \in \mathbb{F} \quad \text{and} \quad t_1^i = G(\mathbf{k}_1^i) \in \mathbb{F}, \quad i \in [k].$$

so P_A knows (t_0^i, t_1^i) and P_B knows $t_{\Delta_i}^i$ for $i = 1, \ldots, k$.
2: P_A computes $u^i = t_0^i - t_1^i + x \in \mathbb{F}$. for $i = 1, \ldots, k$ and sends these to P_B.
3: P_B computes

$$\begin{aligned} q^i &= \Delta_i \cdot u^i + t_{\Delta_i}^i \\ &= t_0^i + \Delta_i \cdot x \in \mathbb{F} \end{aligned}$$

for $i = 1, \ldots, k$.
4: Let $\mathbf{q} = (q^1, \ldots, q^k)$ and $\mathbf{t} = (t_0^1, \ldots, t_0^k)$. Note that

$$\mathbf{q} = \mathbf{t} + x \cdot \mathbf{\Delta}_B \in \mathbb{F}^k.$$

5: P_B sets $q = \langle \mathbf{g}, \mathbf{q} \rangle$.
6: P_A sets $t = -\langle \mathbf{g}, \mathbf{t} \rangle$.
7: Now it holds that $t + q = x \cdot \Delta \in \mathbb{F}$.

correspond to errors introduced in the functionality, and the receiver's input $\mathbf{x}$ is computationally hidden by the output of the PRG when the receiver sends the $\mathbf{u}^i$ values. We therefore omit the precise details of the simulation, noting that it is essentially the same of that given by Nielsen [31].

Complexity. The cost of a single iteration of our COPE protocol, after the base OTs in initialization, is just that of sending k field elements, for a total of k^2 bits.

4 Authenticating and opening additive shares

In this section we show how to create authenticated SPDZ shares using COPE and securely open linear combinations of these shares with a MAC checking procedure. The main challenge is to ensure that an adversary who inputs errors in our COPE protocol cannot later open an authenticated share to the incorrect value. We model these requirements in a single functionality, $\mathcal{F}_{[\![\cdot]\!]}$ (Fig. 3), which is independent of the details of the MAC scheme used and the underlying MAC keys; this leads to a

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

The functionality uses a finite field $\mathbb{F}$, of bit length k, and runs with with parties P_A, P_B and an adversary $\mathcal{S}$.

Initialize($\mathbb{F}$)**:** Upon receiving $\Delta \in \mathbb{F}$ from P_B, the functionality stores Δ.

Extend: Upon receiving $x \in \mathbb{F}$ from P_A:

1: Sample $t \xleftarrow{\$} \mathbb{F}$.
2: If P_A is corrupt then receive vectors $\mathbf{x} \in \mathbb{F}^k$ from $\mathcal{A}$ and compute q, such that

$$q + t = \langle \mathbf{x}, \mathbf{\Delta}_B \rangle$$

Otherwise, compute q such that

$$q + t = x \cdot \Delta$$

3: If P_B is corrupt then receive $q \in \mathbb{F}$ from $\mathcal{A}$ and recompute t to satisfy the above. Output t to P_A and q to P_B.

Figure 2: Correlated oblivious product evaluation with errors

Functionality $\mathcal{F}_{[\![\cdot]\!]}$

The functionality maintains a dictionary, Val, to keep track of the authenticated values. Entries of Val lie in the (fixed) finite field $\mathbb{F}$ and cannot be changed, for simplicity. $\mathcal{F}_{[\![\cdot]\!]}$ also maintains the sets Open and Cheat to record all openings and those where the adversary tried to cheat.

Input: On receiving $(\mathsf{Input}, \mathsf{id}_1, \ldots, \mathsf{id}_m, x_1, \ldots, x_m, P_j)$ from party P_j and $(\mathsf{Input}, \mathsf{id}_1, \ldots, \mathsf{id}_m, P_j)$ from all other parties, where $x \in \mathbb{F}$, set $\mathsf{Val}[\mathsf{id}_i] \leftarrow x_i$ for $i = 1, \ldots, m$.

Linear comb.: On receiving $(\mathsf{LinComb}, \overline{\mathsf{id}}, \mathsf{id}_1, \ldots, \mathsf{id}_t, c_1, \ldots, c_t, c)$ from all parties, where $(\mathsf{id}_1, \ldots, \mathsf{id}_t) \subseteq \mathsf{Val.keys}()$ and the combination coefficients $c_1, \ldots, c_t, c \in \mathbb{F}$, set $\mathsf{Val}[\overline{\mathsf{id}}] \leftarrow \sum_{i=1}^{t} \mathsf{Val}[\mathsf{id}_i] \cdot c_i + c$.

Open: On receiving $(\mathsf{Open}, \mathsf{id})$ from all parties, where $\mathsf{id} \in \mathsf{Val.keys}()$, send $\mathsf{Val}[\mathsf{id}]$, wait for x from the adversary, and output x to all parties.

Check: On receiving $(\mathsf{Check}, \mathsf{id}_1, \ldots, \mathsf{id}_t, x_1, \ldots, x_t)$ from every party P_i, wait for an input from the adversary. If it inputs OK, and $\mathsf{Val}[\mathsf{id}_j] = x_j$ for all j, return OK to all parties, otherwise return $\perp$ and terminate.

Abort: On receiving Abort from the adversary, send $\perp$ to all parties and terminate.

Figure 3: Functionality for authenticating, computing linear combinations of, and opening additively shared values

clearer exposition and greatly simplifies higher level protocols that use the functionality, compared with previous works. We first explain the mechanics of the functionality, and then describe the protocols for implementing it.

Inputs are provided to the functionality with the Input command, which takes as input a value x from one party and stores it along with an identifier, id. Linear functions can be computed on

Protocol 2 MAC checking subprotocol

On input an opened value y, a MAC share $m^{(i)}$ and a MAC key share $\Delta^{(i)}$ from party P_i, each P_i does the following:

1: Compute $\sigma^{(i)} \leftarrow m^{(i)} - y \cdot \Delta^{(i)}$ and call $\mathcal{F}_{\mathsf{Comm}}$ to commit to this and receive the handle τ_i.
2: Call $\mathcal{F}_{\mathsf{Comm}}$ with (Open, τ_i) to open the commitments.
3: If $\sigma^{(1)} + \cdots + \sigma^{(n)} \neq 0$, output $\perp$ and abort, otherwise continue.

values that have been input using the LinComb command.

The Open command lets the adversary output inconsistent or incorrect values. However, if this happened to honest parties, the Check command will reveal this.

4.1 Authentication using COPE

Using the correlated oblivious product evaluation protocol, one party holding a share $x^{(i)}$ can authenticate $x^{(i)}$ under another party's secret MAC key, $\Delta^{(j)}$. When every party has an additive share of x, this protocol can be run between every pair of parties to authenticate each share; if party P_i uses the same MAC key $\Delta^{(i)}$ in every instance then the resulting pairwise MACs can be added together to create shares of a single SPDZ MAC on the additively shared value, x.

In more detail: first, each party P_i samples a random share, $\Delta^{(i)}$, of the global MAC key. Now suppose we wish to authenticate an additively shared value $x \in \mathbb{F}$, so each party holds a share $x^{(i)}$. A natural approach would then be to run an instance of $\mathcal{F}_{\mathsf{COPEe}}$ between every pair of parties (P_i, P_j): P_j inputs $\Delta^{(j)}$ as their correlation into $\mathcal{F}_{\mathsf{COPEe}}.\mathsf{Initialize}$, and then they call $\mathcal{F}_{\mathsf{COPEe}}.\mathsf{Extend}$ where P_i inputs $x^{(i)}$. Let $t^{(i,j)}$ be the $\mathcal{F}_{\mathsf{COPEe}}$ output received by P_i when playing with P_j, and let $q^{(j,i)}$ be the output received by P_j in the same instance. If they both played honestly, then

$$q^{(j,i)} + t^{(i,j)} = x^{(i)} \cdot \Delta^{(j)}.$$

After the $n(n-1)$ executions are done, each party P_i simply combines their results as follows to compute the MAC share

$$m^{(i)} = x^{(i)} \cdot \Delta^{(i)} + \sum_{j \neq i} \left(q^{(i,j)} + t^{(i,j)} \right).$$

Correctness in the semi-honest case. To see that this is correct if all parties behaved honestly, look at the sum of the MAC shares and observe that the MAC relation holds:

$$\begin{aligned} m = \sum_{i=1}^{n} m^{(i)} &= \sum_{i=1}^{n} x^{(i)} \cdot \Delta^{(i)} + \sum_{i=1}^{n} \sum_{j \neq i} \left(q^{(i,j)} + t^{(i,j)} \right) \\ &= \sum_{i=1}^{n} x^{(i)} \cdot \Delta^{(i)} + \sum_{i=1}^{n} \sum_{j \neq i} \left(q^{(j,i)} + t^{(i,j)} \right) \\ &= \sum_{i=1}^{n} x^{(i)} \cdot \Delta^{(i)} + \sum_{i=1}^{n} \sum_{j \neq i} x^{(i)} \cdot \Delta^{(j)} \\ &= x \cdot \Delta. \end{aligned}$$

Protocol 3 $\Pi_{[\![\cdot]\!]}$, creating $[\![\cdot]\!]$ elements

This protocol authenticates additively shared inputs in $\mathbb{F}$, and allows linear operations and opening to be carried out on these shares. Note that the **Initialize** procedure only needs to be called once, to set up the MAC keys.

Initialize: Each party P_i samples a MAC key share $\Delta^{(i)} \in \mathbb{F}$. Each pair of parties (P_i, P_j) (for $i \neq j$) calls $\mathcal{F}_{\mathsf{COPEe}}.\mathsf{Initialize}(\mathbb{F})$ where P_j inputs $\Delta^{(j)}$.

Input: On input $(\mathsf{Input}, \mathsf{id}_1, \ldots, \mathsf{id}_m, x_1, \ldots, x_m, P_j)$ from P_j and $(\mathsf{Input}, \mathsf{id}_1, \ldots, \mathsf{id}_m, P_j)$ from all other parties:

1: P_j samples $x_0 \xleftarrow{\$} \mathbb{F}$.
2: For $h = 0, \ldots, m$, P_j generates a random additive sharing $\sum_i x_h^{(i)} = x_h$ and sends $x_h^{(i)}$ to P_i.
3: For every $i \neq j$, P_i and P_j call $\mathcal{F}_{\mathsf{COPEe}}.\mathsf{Extend}$, where P_j inputs $(x_0, \ldots, x_m) \in \mathbb{F}^m$.
4: P_i receives $q_h^{(i,j)}$ and P_j receives $t_h^{(j,i)}$ such that

$$q_h^{(i,j)} + t_h^{(j,i)} = x_h \cdot \Delta^{(i)}, \text{ for } h = 0, \ldots, m.$$

5: Each P_i, $i \neq j$, computes the MAC shares $m_h^{(i)} = \sum_{j \neq i} q_h^{(i,j)}$, and P_j computes the MAC shares

$$m_h^{(j)} = x_h \cdot \Delta^{(j)} + \sum_{j \neq i} t_h^{(j,i)}$$

to obtain $[\![x_h]\!]$, for $h = 0, \ldots, m$.
6: The parties sample $\mathbf{r} \leftarrow \mathcal{F}_{\mathsf{Rand}}(\mathbb{F}^{m+1})$.
7: P_j computes and broadcasts $y = \sum_{h=0}^{m} r_h \cdot x_h$.
8: Each party P_i computes $m^{(i)} = \sum_{h=0}^{m} r_h \cdot m_h^{(i)}$.
9: The parties execute Π_{MACCheck} with y and $\{m^{(i)}\}_{i \in [n]}$.
10: All parties store their shares and MAC shares under the handles $\mathsf{id}_1, \ldots, \mathsf{id}_m$.

Linear comb.:

On input $(\mathsf{LinComb}, \overline{\mathsf{id}}, \mathsf{id}_1, \ldots, \mathsf{id}_t, c_1, \ldots, c_t, c)$, the parties retrieve their shares and MAC shares $\{x_j^{(i)}, m(x_j)^{(i)}\}_{j \in [t], i \in [n]}$ corresponding to $\mathsf{id}_1, \ldots, \mathsf{id}_t$, and each P_i computes:

$$y^{(i)} = \sum_{j=1}^{t} c_j \cdot x_j^{(i)} + \begin{cases} c & i = 0 \\ 0 & i \neq 0 \end{cases}$$

$$m(y)^{(i)} = \sum_{j=1}^{t} c_j \cdot m(x_j)^{(i)} + c \cdot \Delta^{(i)},$$

where They then store the new share and MAC of $[\![y]\!]$ under the handle $\overline{\mathsf{id}}$.

Active adversaries. We now examine the case of active adversaries, who may not follow the procedure as specified. A corrupted party P_j may cheat in the above protocol in one of three possible ways: they can use a different MAC key share $\Delta^{(i,j)}$ in the $\mathcal{F}_{\mathsf{COPEe}}.\mathsf{Initialize}$ step with party P_i and they could use non-monochrome input shares $\mathbf{x}^{(i,j)}$ in the $\mathcal{F}_{\mathsf{COPEe}}.\mathsf{Extend}$ calls, and

Protocol $\Pi_{[\![\cdot]\!]}$ (continued)

11: $\Pi_{[\![\cdot]\!]}$ (continued)

Open: On input $(\mathsf{Open}, \mathsf{id})$:

1: Each P_i retrieves and broadcasts their share $x^{(i)}$.

2: Parties reconstruct $x = \sum_{i=1}^{n} x_j^{(i)}$ and output it.

Check: On input $(\mathsf{Check}, \mathsf{id}_1, \ldots, \mathsf{id}_t, x_1, \ldots, x_t)$, the parties do the following:

1: Sample a public, random vector $\mathbf{r} \leftarrow \mathcal{F}_{\mathsf{Rand}}(\mathbb{F}^t)$.

2: Compute $y \leftarrow \sum_{j=1}^{t} r_j \cdot x_j$ and $m(y)^{(i)} \leftarrow \sum_{j=1}^{t} r_j \cdot m_{\mathsf{id}_j}^{(i)}$, where $m_{\mathsf{id}_j}^{(i)}$ denotes P_i's MAC share stored under id_j for all $i \in [n]$ and $j \in [t]$.

3: Execute Π_{MACCheck} with y and $m(y)^{(i)}$.

these may also differ between each instance of $\mathcal{F}_{\mathsf{COPEe}}$. We will show that using inconsistent inputs in different $\mathcal{F}_{\mathsf{COPEe}}$ instances does not help the adversary, but non-monochrome vectors allow them to open to more than one value with non-negligible probability. This is the reason for the check at the end of the input step, which forces the adversary to decide upon a value, and allows the simulator to extract this value in the security proof, to provide as input to $\mathcal{F}_{[\![\cdot]\!]}$.

The sum of the honest parties' MACs is given by

$$
\begin{aligned}
&\sum_{i \notin A} m^{(i)} = \sum_{i \notin A} x^{(i)} \cdot \Delta^{(i)} + \sum_{i \notin A} \sum_{j \neq i} \Big(q^{(i,j)} + t^{(i,j)} \Big) \\
&= \sum_{i \notin A} x^{(i)} \cdot \Delta^{(i)} + \sum_{j \notin A, i \neq j} x^{(i)} \cdot \Delta^{(j)} \\
&\quad + \sum_{i \notin A} \sum_{j \in A} \langle \mathbf{g} * \mathbf{x}^{(i,j)}, \mathbf{\Delta}_B^{(i)} \rangle - t^{(j,i)} + x^{(i)} \cdot \Delta^{(j,i)} - q^{(j,i)} \\
&= \sum_{i \notin A} x^{(i)} \cdot \Delta^{(i)} + \sum_{j \notin A, i \neq j} x^{(i)} \cdot \Delta^{(j)} \\
&\quad + \sum_{i \notin A} \sum_{j \in A} \langle \mathbf{g} * \mathbf{x}^{(i,j)}, \mathbf{\Delta}_B^{(i)} \rangle - t^{(j,i)} + x^{(i)} \cdot \Delta^{(j,i)} - q^{(j,i)} \\
&= \sum_{i \notin A} \Big(\Big\langle \mathbf{g} \cdot \sum_{j \notin A} x^{(j)} + \mathbf{g} * \sum_{j \in A} \mathbf{x}^{(i,j)}, \mathbf{\Delta}_B^{(i)} \Big\rangle \\
&\quad + \sum_{j \in A} (-t^{(j,i)} + x^{(i)} \cdot \Delta^{(j,i)} - q^{(j,i)}) \Big).
\end{aligned}
$$

Define the second summand to be R_i. If only checking the opening of this value to x, the honest party P_i broadcasts $\sigma^{(i)} = m^{(i)} - x \cdot \Delta^{(i)}$. Summing up,

$$
\begin{aligned}
&\sum_{i \notin A} m^{(i)} - x \cdot \Delta^{(i)} \\
&= \sum_{i \notin A} \Big(\Big\langle \mathbf{g} \cdot \sum_{j \notin A} x^{(j)} + \mathbf{g} * \sum_{j \in A} \mathbf{x}^{(i,j)} - \mathbf{g} \cdot x, \mathbf{\Delta}_B^{(i)} \Big\rangle + R_i \Big) \\
&= \sum_{i \notin A} \Big(\Big\langle \mathbf{g} \cdot \big(\sum_{j \notin A} x^{(j)} - x \big) + \mathbf{g} * \sum_{j \in A} \mathbf{x}^{(i,j)}, \mathbf{\Delta}_B^{(i)} \Big\rangle + R_i \Big).
\end{aligned}
\tag{1}
$$

The corrupted parties have to provide values $\{\sigma^{(i)}\}_{i\in A}$ such that $\sum_i \sigma^{(i)} = 0$ for the check to pass. Note that $\{R_i\}_{i\notin A}$ are independent of $\{\boldsymbol{\Delta}_B^{(i)}\}_{i\notin A}$ and known to the adversary because $\{x^{(i)}\}_{i\notin A}$ were revealed in the opening. The adversary therefore has to "guess" the inner product for random $\{\boldsymbol{\Delta}_B^{(i)}\}_{i\notin A}$. Clearly, success is guaranteed if $x - \sum_{j\notin A} x^{(j)} = \sum_{j\in A} \mathbf{x}^{(i,j)}$ for all $i \notin A$. This roughly corresponds to the corrupted parties following the protocol in the sense that all deviations cancel out when taking a global view. It is easy to see that once value has been opened, the same value cannot be opened to another value without guessing $\sum_i \Delta^{(i)}$, which happens with negligible probability. However, opening a linear combination only fixes the linear combination this way but not the summands of the linear combination. On the other hand, we will prove that opening a random linear combination indeed fixes every summand.

Linear combinations. Since the additive secret sharing and MAC schemes are linear, performing linear combinations on authenticated shared values is straightforward. When adding a public value y to $[\![x]\!]$, each party must locally adjust their MAC share for x by adding $\Delta^{(i)} \cdot y$ so that the resulting MAC is correct.

Checking MACs. The Check subprotocol checks a batch of t MACs using the same procedure as Damgård et al. [14]. First a public, secure random value $\mathbf{r} \in \mathbb{F}^m$ is sampled, and then this is used to compute a random linear combination, y, of the inputs. The MAC on y is then checked by having each party P_i first commit to, and then open, $\sigma^{(i)} \leftarrow m(y)^{(i)} - y \cdot \Delta^{(i)}$. The parties then check that these shares sum to zero, which holds if the MAC is correct, i.e. $m(y) = y \cdot \Delta$.

The security of our authentication and MAC checking protocols is given formally in the following theorem, which we prove in Appendix A.

Theorem 1. *The protocol $\Pi_{[\![\cdot]\!]}$ securely implements $\mathcal{F}_{[\![\cdot]\!]}$ in the $(\mathcal{F}_{\mathsf{COPEe}}, \mathcal{F}_{\mathsf{Comm}}, \mathcal{F}_{\mathsf{Rand}})$-hybrid model, with statistical security parameter $\log|\mathbb{F}| - 2\log\log|\mathbb{F}|$.*

The most intricate part of the proof is to guarantee that, once the adversary has passed the check in the input phase, they are committed to a particular value. However, the adversary has an edge (reflected by the $2\log\log|\mathbb{F}|$ subtrahend) because only a random combination of inputs can be checked (otherwise all the inputs would be revealed). This can be seen as follows: Assume that there is only one honest and one dishonest party. Denote by $x_{h,g}$ the g-th entry of the vector $\mathbf{x}_h$ input when authenticating the h-th value, and denote by $\{r_h\}_{h\in[m]}$ the random coefficients generated using $\mathcal{F}_{\mathsf{Rand}}$. For $g \neq g' \in [k]$, if $x_{h,g} \neq x_{h,g'}$, there is a $1/|\mathbb{F}|$ chance that $\sum r_h x_{h,g} = \sum r_h x_{h,g'}$. Because the check only relates to the randomly weighted sum, the adversary could therefore act as if $x_{h,g} = x_{h,g'}$ and decide later between $\{x_{h,g}\}_{h\in[m]}$ and $\{x_{h,g'}\}_{h\in[m]}$. The fact that there are $\log|\mathbb{F}|(\log|\mathbb{F}| - 1)/2$ such pairs $g \neq g'$ explains the $2\log\log|\mathbb{F}|$ subtrahend in the theorem. It is easy to see that a repeated check would suffice for security parameter $\log|\mathbb{F}|$.

5 Multiplication triples using oblivious transfer

In the previous section we showed how parties can compute linear functions on their private inputs using the authentication and MAC checking protocols. We now extend this to arbitrary functions, by showing how to create multiplication triples using $\mathcal{F}_{[\![\cdot]\!]}$ and OT.

Recall that a multiplication triple is a tuple of shared values $([\![a]\!], [\![b]\!], [\![c]\!])$ where $a, b \xleftarrow{\$} \mathbb{F}$ and $c = a \cdot b$. Given $\mathcal{F}_{[\![\cdot]\!]}$ and a protocol for preprocessing triples, the online phase of the resulting MPC

protocol is straightforward, using Beaver's method for multiplying two secret-shared values [4]. For completeness, this is detailed in Appendix 6.

Our protocol is designed to use $\mathcal{F}_{[\![\cdot]\!]}$ and an ideal OT functionality to securely implement the functionality $\mathcal{F}_{\mathsf{Triple}}$, which has all of the same features as $\mathcal{F}_{[\![\cdot]\!]}$, with the following additional command:

$\mathcal{F}_{\mathsf{Triple}}$**:** On input $(\mathsf{Triple}, \mathsf{id}_a, \mathsf{id}_b, \mathsf{id}_c)$ from all parties, sample two random values $a, b \xleftarrow{\$} \mathbb{F}$ and set $(\mathsf{Val}[\mathsf{id}_a], \mathsf{Val}[\mathsf{id}_b], \mathsf{Val}[\mathsf{id}_c]) \leftarrow (a, b, a \cdot b)$.

Throughout this section, we write $[\![x]\!]$ to mean that each party holds a random, additive share of x, and the value of x is stored in the ideal box $\mathcal{F}_{[\![\cdot]\!]}$. We start by showing correctness of the protocol Π_{Triple} (Protocol 4) for semi-honest adversaries.

Each party P_i first samples their input shares $\mathbf{a}^{(i)} \xleftarrow{\$} \mathbb{F}^\tau$, $b^{(i)} \xleftarrow{\$} \mathbb{F}$. Then they run $\tau \cdot k$ sets of random OTs on k-bit strings ($\mathcal{F}_{\mathsf{ROT}}^{\tau k, k}$), where P_i plays receiver with input $\mathbf{a}_B^{(i)} = (a_1^{(i)}, \ldots, a_{\tau k}^{(i)}) \in \mathbb{F}_2^{\tau k}$, with $\mathbf{a}_B^{(i)} = \mathbf{g}^{-1}(\mathbf{a}^{(i)})$, and P_j obtains $q_{0,h}^{(j,i)}$, $q_{1,h}^{(j,i)} \in \mathbb{F}$, $h \in [\tau k]$. Then P_j sends $d_h^{(j,i)} = q_{0,h}^{(j,i)} - q_{1,h}^{(j,i)} + b^{(j)} \in \mathbb{F}$ to P_i, who computes $t_h^{(i,j)} = q_{0,h}^{(j,i)} + a_h^{(i)} \cdot b^{(j)}$ for each h. By splitting the vectors $(t_1^{(i,j)}, \ldots, t_{\tau k}^{(i,j)})$ and $(q_1^{(j,i)}, \ldots, q_{\tau k}^{(j,i)})$ into τ vectors of k components, P_i and P_j respectively obtain $\mathbf{t}^{(i,j)} = (\mathbf{t}_1^{(i,j)}, \ldots, \mathbf{t}_\tau^{(i,j)})$ and $\mathbf{q}^{(j,i)} = (\mathbf{q}_1^{(j,i)}, \ldots, \mathbf{q}_\tau^{(j,i)})$. Then they apply $\mathbf{g}$ to each component of these, getting

$$\mathbf{c}_{i,j}^{(i)} + \mathbf{c}_{i,j}^{(j)} = \mathbf{t}^{(i,j)} + \mathbf{q}^{(j,i)} = \mathbf{a}^{(i)} \cdot b^{(j)} \in \mathbb{F}^\tau.$$

After running this between every pair of parties, we have the following relation:

$$\begin{aligned}
\mathbf{c} &= \sum_{i \in [n]} \mathbf{c}^{(i)} = \sum_{i \in [n]} \mathbf{a}^{(i)} \cdot b^{(i)} + \sum_{i \in [n]} \sum_{j \neq i} \mathbf{c}_{i,j}^{(i)} + \mathbf{c}_{i,j}^{(j)} \\
&= \sum_{i \in [n]} \mathbf{a}^{(i)} \cdot b^{(i)} + \sum_{i \in [n]} \mathbf{a}^{(i)} \cdot \sum_{j \neq i} b^{(j)} \\
&= \sum_{i \in [n]} \mathbf{a}^{(i)} \cdot \sum_{i \in [n]} b^{(i)} = \big(\sum_{i \in [n]} \mathbf{a}^{(i)}\big) \cdot b = \mathbf{a} \cdot b.
\end{aligned}$$

The parties then locally **Combine** together the τ components of $\mathbf{a}^{(i)} = (a_1^{(i)}, \ldots, a_\tau^{(i)})$ and $\mathbf{c}^{(i)} = (c_1^{(i)}, \ldots, c_\tau^{(i)})$ using two random values $\mathbf{r}$ and $\hat{\mathbf{r}}$ in $\mathbb{F}^\tau$ obtained from $\mathcal{F}_{\mathsf{Rand}}$. The outputs of this step are a, c and $\hat{a}, \hat{c}$ in $\mathbb{F}$ such that $c = a \cdot b$ and $\hat{c} = \hat{a} \cdot b$. Looking ahead, this step is necessary to achieve active security.

The parties then use $\mathcal{F}_{[\![\cdot]\!]}$ to **Authenticate** their shares of $a, \hat{a}, b$, c and $\hat{c}$.

Finally, correctness of the triples $[\![a]\!], [\![b]\!], [\![c]\!]$ is checked in a **Sacrifice** phase, using $[\![\hat{a}]\!]$ and $[\![\hat{c}]\!]$. The idea of this step is similar to the corresponding step in previous works [14, 15].

Active security. It is clear that an adversary can choose to not follow the protocol in several ways, during both multiplication and authentication. Although the **Sacrifice** step ensures that the final triple is still correct, this does not prevent the adversary from forcing *leakage* on a triple. We prevent this in the **Combine** step, which ensures that the final triple is sufficiently random. To see how this works, let us first examine the possible adversarial deviations in the **Multiply** step.

Suppose P_j is corrupt. Let $\mathbf{a}^{(j,i)} \in \mathbb{F}^\tau$ and $\mathbf{b}^{(j,i)} \in \mathbb{F}^{\tau k}$ be the *actual* values used by P_j in the two executions of steps 1 and 3 with an honest P_i, instead of $\mathbf{a}^{(j)}$ and $b^{(j)}$. Define the values $\mathbf{a}^{(j)}$

Protocol 4 Triple generation protocol, Π_{Triple}

The integer parameter $\tau \geq 3$ specifies the amount of random combining to perform.

Multiply:

1. Each party samples $\mathbf{a}^{(i)} \xleftarrow{\$} \mathbb{F}^\tau, b^{(i)} \xleftarrow{\$} \mathbb{F}$.
2. Every ordered pair of parties (P_i, P_j) does the following:
   1: Both parties call $\mathcal{F}_{\mathsf{ROT}}^{\tau k,k}$ where P_i inputs $(a_1^{(i)}, \ldots, a_{\tau k}^{(i)}) = \mathbf{g}^{-1}(\mathbf{a}^{(i)}) \in \mathbb{F}_2^{\tau k}$.
   2: P_j receives $q_{0,h}^{(j,i)}, q_{1,h}^{(j,i)} \in \mathbb{F}$ and P_i receives $s_h^{(i,j)} = q_{a_h^{(i)},h}^{(j)}$, for $h = 1, \ldots, \tau k$.
   3: P_j sends $d_h^{(j,i)} = q_{0,h}^{(j,i)} - q_{1,h}^{(j,i)} + b^{(j)}, h \in [\tau k]$.
   4: P_i sets $t_h^{(i,j)} = s_h^{(i,j)} + a^{(i)} \cdot d_h^{(j,i)} = q_{0,h}^{(j,i)} + a_h^{(i)} \cdot b^{(j)}$, for $h = 1, \ldots, \tau k$. Set $q_h^{(j,i)} = q_{0,h}^{(j,i)}$.
   5: Split $(t_1^{(i,j)}, \ldots, t_{\tau k}^{(i,j)})$ and $(q_1^{(j,i)}, \ldots, q_{\tau k}^{(j,i)})$ into τ vectors of k components each, $(\mathbf{t}_1, \ldots, \mathbf{t}_\tau)$ and $(\mathbf{q}_1, \ldots, \mathbf{q}_\tau)$.
   6: P_i sets $\mathbf{c}_{i,j}^{(i)} = (\langle \mathbf{g}, \mathbf{t}_1 \rangle, \ldots, \langle \mathbf{g}, \mathbf{t}_\tau \rangle) \in \mathbb{F}^\tau$.
   7: P_j sets $\mathbf{c}_{i,j}^{(j)} = -(\langle \mathbf{g}, \mathbf{q}_1 \rangle, \ldots, \langle \mathbf{g}, \mathbf{q}_\tau \rangle) \in \mathbb{F}^\tau$.
   8: Now we have
   $$\mathbf{c}_{i,j}^{(i)} + \mathbf{c}_{i,j}^{(j)} = \mathbf{a}^{(i)} \cdot b^{(j)} \in \mathbb{F}^\tau$$
3. Each party P_i computes:
   $$\mathbf{c}^{(i)} = \mathbf{a}^{(i)} \cdot b^{(i)} + \sum_{j \neq i} (\mathbf{c}_{i,j}^{(i)} + \mathbf{c}_{j,i}^{(i)})$$

Combine:

1: Sample $\mathbf{r}, \hat{\mathbf{r}} \leftarrow \mathcal{F}_{\mathsf{Rand}}(\mathbb{F}^\tau)$.
2: Each party P_i sets
$$a^{(i)} = \langle \mathbf{a}^{(i)}, \mathbf{r} \rangle, \qquad c^{(i)} = \langle \mathbf{c}^{(i)}, \mathbf{r} \rangle \qquad \text{and}$$
$$\hat{a}^{(i)} = \langle \mathbf{a}^{(i)}, \hat{\mathbf{r}} \rangle, \qquad \hat{c}^{(i)} = \langle \mathbf{c}^{(i)}, \hat{\mathbf{r}} \rangle$$

Authenticate: Each party P_i runs $\mathcal{F}_{[\![\cdot]\!]}.\mathsf{Input}$ on their shares to obtain authenticated shares $[\![a]\!], [\![b]\!], [\![c]\!], [\![\hat{a}]\!], [\![\hat{c}]\!]$.

Sacrifice: Check correctness of the triple $([\![a]\!], [\![b]\!], [\![c]\!])$ by sacrificing $[\![\hat{a}]\!], [\![\hat{c}]\!]$.

1: Sample $s \leftarrow \mathcal{F}_{\mathsf{Rand}}(\mathbb{F})$.
2: Call $\mathcal{F}_{[\![\cdot]\!]}.\mathsf{Open}$ on input $s \cdot [\![a]\!] - [\![\hat{a}]\!]$ to obtain ρ.
3: Call $\mathcal{F}_{[\![\cdot]\!]}.\mathsf{LinComb}$ to store $s \cdot [\![c]\!] - [\![\hat{c}]\!] - [\![b]\!] \cdot \rho$ under $[\![\sigma]\!]$.
4: Run $\mathcal{F}_{[\![\cdot]\!]}.\mathsf{Check}([\![\rho]\!], [\![\sigma]\!], \rho, 0)$ and abort if $\mathcal{F}_{[\![\cdot]\!]}$ aborts.

Output: $([\![a]\!], [\![b]\!], [\![c]\!])$ as a valid triple.

and $b^{(j)}$ to be those values used in the instance with an arbitrary (e.g. lowest index) honest party P_{i_0}.

Then, for each $i \notin A$, let $\boldsymbol{\delta}_a^{(j,i)} = \mathbf{a}^{(j,i)} - \mathbf{a}^{(j)} \in \mathbb{F}^\tau$ and $\boldsymbol{\delta}_b^{(j,i)} = \mathbf{b}^{(j,i)} - (b^{(j)}, \ldots, b^{(j)}) \in \mathbb{F}^{\tau k}$ be the deviation in P_j's input with an honest P_i. Let $\boldsymbol{\delta}_a^{(i)} = \sum_{j \in A} \boldsymbol{\delta}_a^{(j,i)}$ and $\boldsymbol{\delta}_b^{(i)} = \sum_{j \in A} \boldsymbol{\delta}_b^{(j,i)}$, and consider $\boldsymbol{\delta}_b^{(i)}$ as a length τ vector with components in $\mathbb{F}^k$ (similarly to $\mathbf{t}_h, \mathbf{q}_h$ in the protocol).

Now by analyzing the possible adversarial deviations and summing up shares, we can see that

the h-th component of $\mathbf{c}$ (for $h \in [\tau]$), at the end of the **Multiply** stage, is

$$\mathbf{c}[h] = \mathbf{a}[h] \cdot b + \underbrace{\sum_{i \notin A} \langle (\mathbf{a}^{(i)}[h])_B, \boldsymbol{\delta}_b^{(i)}[h] \rangle}_{=e_{a_h}} + \underbrace{\sum_{i \notin A} b^{(i)} \cdot \boldsymbol{\delta}_a^{(i)}[h]}_{=e_{b_h}} . \tag{2}$$

Intuitively, it is easy to see that any non-zero $\boldsymbol{\delta}_a^{(i)}$ errors will be blown up by the random honest party's share $b^{(i)}$, so should result in an incorrect triple with high probability. On the other hand, the $\boldsymbol{\delta}_b^{(i)}$ errors can be chosen so that e_{a_h} only depends on single bits of the shares $\mathbf{a}^{(i)}$. This means that a corrupt party can attempt to guess a few bits (or linear combinations of bits) of $\mathbf{a}^{(i)}$. If this guess is incorrect then the resulting triple should be incorrect; however, if all guesses succeed then the triple is correct and the sacrifice step will pass, whilst the adversary learns the bits that were guessed.

This potential leakage (or *selective failure attack*) is the reason for initially using a vector for the $\mathbf{a}^{(i)}$ shares, rather than just one field element. The **Combine** step then randomly reduces $\mathbf{a}^{(i)}$ down to two field elements. The intuition here is that, to be able to guess a single bit of the final shares $a^{(i)}, \hat{a}^{(i)}$, the adversary must have guessed *many bits* from the input vector, which is very unlikely to happen. To prove this intuition, we analyze the distribution of the honest party's output shares using the Leftover Hash Lemma, and show that with suitable parameter choices, the combined output is statistically close to uniform to the adversary.

Finally, after authentication, the **Sacrifice** stage uses $[\![\hat{a}]\!]$ and $[\![\hat{c}]\!]$ to verify correctness of the output triple. In previous works [14, 15], a whole triple is wasted to check one other; however, we observe that by using two triples with the same b component, we can save the cost of authenticating one field element in this step. Note that we also save performing an additional multiplication, as we use the original triple $(\mathbf{a}, b, \mathbf{c})$ to produce *both* triples for sacrificing, by combining with two different random seeds.

The following results (proven in Appendix B) state the security of our protocol. The first requires the combining parameter set to $\tau = 4$, to obtain a a general result for any k-bit field, whilst the second shows that for 128-bit fields and 64-bit statistical security, $\tau = 3$ suffices.

Theorem 2. *The protocol Π_{Triple} (Protocol 4) securely implements $\mathcal{F}_{\mathsf{Triple}}$ in the $(\mathcal{F}_{\mathsf{ROT}}, \mathcal{F}_{[\![\cdot]\!]})$-hybrid model, with statistical security parameter k, for $\tau = 4$.*

Corollary 1. *For $k \geq 128$ and $\tau = 3$, Π_{Triple} securely implements $\mathcal{F}_{\mathsf{Triple}}$ with statistical security parameter 64.*

6 Complete preprocessing and online protocols

We now describe the complete protocols for preprocessing and the online phase of our MPC protocol.

6.1 Preprocessing

As well as multiplication triples, we also want the preprocessing to produce random, shared values known by a single party (called input tuples), to allow that party to provide inputs in the online phase. This is easy to do with Protocol 5: the relevant party simply inputs a random value to $\mathcal{F}_{[\![\cdot]\!]}$. In the online phase, they broadcast the difference of this and their actual input, so that the shared

Protocol 5 Preprocessing input tuples, $\Pi_{\mathsf{InputTuple}}$

Input: On input (Input, P_j) from all parties, do the following:

1: P_j samples $r \xleftarrow{\$} \mathbb{F}$, and calls $\mathcal{F}_{[\![\cdot]\!]}$ with (Input, r, P_j).
2: All parties output $[\![r]\!]$ and P_j outputs r.

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

$\mathcal{F}_{\mathsf{Prep}}$ has all of the same features as $\mathcal{F}_{[\![\cdot]\!]}$, with the following additional commands:

Input Tuple: On input $(\mathsf{InputTuple}, P_j, \mathsf{id})$ from all parties, sample $\mathsf{Val}[\mathsf{id}] \xleftarrow{\$} \mathbb{F}$, and output it to P_j.

Triple: On input $(\mathsf{Triple}, \mathsf{id}_a, \mathsf{id}_b, \mathsf{id}_c)$ from all parties, sample two random values $a, b \xleftarrow{\$} \mathbb{F}$ and set $(\mathsf{Val}[\mathsf{id}_a], \mathsf{Val}[\mathsf{id}_b], \mathsf{Val}[\mathsf{id}_c]) \leftarrow (a, b, a \cdot b)$.

Figure 4: Ideal functionality for the SPDZ preprocessing phase.

random value can then be adjusted to the correct value by all parties. Note that this method avoids having to use the Input command of $\mathcal{F}_{\mathsf{Prep}}$ (and hence of $\Pi_{[\![\cdot]\!]}$ in the actual protocol) in the online phase, by instead offloading this cost to the preprocessing.

The requirements for input tuple and triple generation are specified in the functionality $\mathcal{F}_{\mathsf{Prep}}$ (Fig. 4), which also contains all features from $\mathcal{F}_{[\![\cdot]\!]}$ (like $\mathcal{F}_{\mathsf{Triple}}$). Given this and the proof of Theorem 2, it is straightforward to show that the triple generation and input tuple generation protocols securely implement $\mathcal{F}_{\mathsf{Prep}}$.

Theorem 3. *The protocols Π_{Triple} and $\Pi_{\mathsf{InputTuple}}$ together securely realize the functionality $\mathcal{F}_{\mathsf{Prep}}$, in the $(\mathcal{F}_{[\![\cdot]\!]}, \mathcal{F}_{\mathsf{ROT}}, \mathcal{F}_{\mathsf{Rand}})$-hybrid model.*

6.2 Online phase

Given the preprocessing data from $\mathcal{F}_{\mathsf{Prep}}$, the online phase is quite straightforward, essentially the same as in SPDZ [15], and shown in Protocol 6. Note that all of the linear computations on $[\![\cdot]\!]$-shared data are performed by calling the relevant command of $\mathcal{F}_{[\![\cdot]\!]}$.

To share an input x_i by party P_i, they take a preprocessed random value $[\![r]\!]$ and broadcast the value $x_i - r$. Since r is uniformly random in $\mathbb{F}$ and unknown to all other parties, it acts as a one-time pad to perfectly hide x_i. All parties can then locally compute $[\![r]\!] + (x_i - r)$ to obtain $[\![x_i]\!]$.

Multiplication of two shared values $[\![x]\!]$ and $[\![y]\!]$ uses Beaver's circuit randomization technique. Given a multiplication triple $[\![a]\!], [\![b]\!], [\![c]\!]$, first the values $x - a$ and $y - b$ are opened; again, the triple values perfectly mask the inputs, so this appears uniformly random to an adversary. Given this, a sharing of the product $x \cdot y$ can be locally computed by all parties using the triple.

The final functionality that the online phase implements is the arithmetic black box, shown in Fig. 5. The following theorem proves UC security of the protocol.

Theorem 4. *The protocol Π_{Online} (Protocol 6) securely implements the functionality $\mathcal{F}_{\mathsf{ABB}}$ (Fig. 5) against a static, active adversary corrupting up to $n-1$ parties in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model.*

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

Initialize: On input $(\mathsf{Init}, \mathbb{F})$ from all parties, store $\mathbb{F}$.

Input: On input $(\mathsf{Input}, P_i, \mathsf{id}, x)$ from P_i and $(\mathsf{Input}, P_i, \mathsf{id})$ from all other parties, with id a fresh identifier and $x \in \mathbb{F}$, store (id, x).

Add: On command $(\mathsf{Add}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from all parties (where $\mathsf{id}_1, \mathsf{id}_2$ are present in memory), retrieve (id_1, x), (id_2, y) and store $(\mathsf{id}_3, x + y)$.

Multiply: On input $(\mathsf{Mult}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from all parties (where $\mathsf{id}_1, \mathsf{id}_2$ are present in memory), retrieve (id_1, x), (id_2, y) and store $(\mathsf{id}_3, x \cdot y)$.

Output: On input $(\mathsf{Output}, \mathsf{id})$ from all honest parties (where id is present in memory), retrieve (id, y) and output it to the adversary. Wait for an input from the adversary; if this is $\mathsf{Deliver}$ then output y to all parties, otherwise output Abort.

Figure 5: The ideal functionality for the MPC arithmetic black box.

Protocol 6 Operations for Secure Function Evaluation, Π_{Online}

Initialize: The parties call $\mathcal{F}_{\mathsf{Prep}}$ for the handles of a number of multiplication triples $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$ and mask values $(r_i, \llbracket r_i \rrbracket)$ as needed for the function being evaluated. If $\mathcal{F}_{\mathsf{Prep}}$ aborts then the parties output $\perp$ and abort.

Input: To share an input x_i, party P_i takes an available mask value $(r_i, \llbracket r_i \rrbracket)$ and does the following:

1: Broadcast $\epsilon \leftarrow x_i - r_i$.
2: The parties compute $\llbracket x_i \rrbracket \leftarrow \llbracket r_i \rrbracket + \epsilon$.

Add: On input $(\llbracket x \rrbracket, \llbracket y \rrbracket)$, locally compute $\llbracket x + y \rrbracket \leftarrow \llbracket x \rrbracket + \llbracket y \rrbracket$.

Multiply: On input $(\llbracket x \rrbracket, \llbracket y \rrbracket)$, the parties do the following:

1: Take one multiplication triple $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$, compute $\llbracket \epsilon \rrbracket \leftarrow \llbracket x \rrbracket - \llbracket a \rrbracket$, $\llbracket \rho \rrbracket \leftarrow \llbracket y \rrbracket - \llbracket b \rrbracket$ and call $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Open}$ on these shares to get ϵ, ρ respectively.
2: Set $\llbracket z \rrbracket \leftarrow \llbracket c \rrbracket + \epsilon \cdot \llbracket b \rrbracket + \rho \cdot \llbracket a \rrbracket + \epsilon \cdot \rho$

Output: To output a share $\llbracket y \rrbracket$, do the following:[2]

1: Call $\mathcal{F}_{\mathsf{Prep}}.\mathsf{Check}$ with input all opened values so far. If it fails, output $\perp$ and abort.
2: Call $\mathcal{F}_{\mathsf{Prep}}$ with commands Open and then Check to open and verify $\llbracket y \rrbracket$. If the check fails, output $\perp$ and abort, otherwise accept y as a valid output.

Proof. Because most of the protocol purely consists of interaction with $\mathcal{F}_{\mathsf{Prep}}$ based on value identifiers, the simulation in Fig. 6 is straightforward. The only values sent in the protocol are masked openings for multiplications and outputs.

Since each multiplication triple is only used once, the two values opened during a multiplication call are uniformly random in both worlds. This means that up until the output stage, the two views of the environment in both worlds are identically distributed and so indistinguishable.

The output stage is straightforward to simulate because the Check aborts exactly if the adversary deviated. □

Simulator $\mathcal{S}_{\mathbf{Online}}$

Initialize: The simulation of this procedure is performed by running a local copy of $\mathcal{F}_{\mathsf{Prep}}$. It is straightforward because the InputTuple and Triple commands of $\mathcal{F}_{\mathsf{Prep}}$ only involve value identifiers.

Input: Simulate according to the following two cases:

- For inputs from an honest party, broadcast a random value.
- For inputs from a corrupt party P_i, wait for P_i to broadcast the (possibly incorrect) value ϵ', compute $x_i' \leftarrow r_i + \epsilon'$ and use x_i' as input to $\mathcal{F}_{\mathsf{ABB}}$.

Add: This local procedure requires no simulation.

Multiply: Send random values for ϵ and ρ to the adversary and wait for it to input ϵ' and ρ'. If $(\epsilon, \rho) \neq (\epsilon', \rho')$, set Fail.

Output: Simulate the output stage as follows:

1. If Fail is set, abort in the first call of $\mathcal{F}_{\mathsf{Prep}}$.Check.
2. Receive the output y from $\mathcal{F}_{\mathsf{ABB}}$ and forward it to the adversary in $\mathcal{F}_{\mathsf{Prep}}$. If it returns $y' \neq y$, input Abort to $\mathcal{F}_{\mathsf{ABB}}$ and abort in the simulation of the second Check call, otherwise input Deliver to $\mathcal{F}_{\mathsf{ABB}}$ and continue.

Figure 6: Simulator for the online phase.

7 Performance and implementation

We first analyse the complexity of our preprocessing protocol, and then describe our implementation and experiments.

7.1 Complexity

We measure the communication complexity of our protocol in terms of the *total* amount of data sent across the network. Note that the number of rounds of communication is constant, so is unlikely to heavily impact performance when generating large amounts of preprocessing data. Throughout this section, we exclude the cost of the base OTs in the initialization stages, as this is a one-time setup cost that takes around a second.

Input tuple generation. The main cost of authenticating one party's field element in a k-bit field with $\Pi_{[\![.]\!]}$ is the $n-1$ calls to Π_{COPEe}, each of which sends k^2 bits, plus sending $n-1$ shares of k bits, for a total of $(n-1)(k^2+k)$ bits. We ignore the cost of authenticating one extra value and performing the MAC check, as this is amortized away when creating a large batch of input tuples.

Triple generation. To generate a triple, each pair of parties makes τk calls to $\mathcal{F}_{\mathsf{ROT}}$, followed by sending a further τk^2 bits in step 3 and then 5 calls to Π_{COPEe} for authentication (ignoring $\mathcal{F}_{\mathsf{Rand}}$ and sending the input shares as these are negligible). Since each call to $\mathcal{F}_{\mathsf{ROT}}$ requires communicating λ bits, and Π_{COPEe} requires k^2 bits, this gives a total of $n(n-1)(\tau\lambda k + (\tau+5)k^2)$ bits sent across the network.

Table 2 shows these complexities for a few choices of field size, with $\lambda = 128$ and τ chosen to achieve at least 64 bit statistical security. We observe that as k increases, the cost of inputs scales almost exactly quadratically. For triples, $k = 64$ is slightly less efficient as we require $\tau = 4$ (instead of 3), whilst for larger k the cost reduces slightly as k becomes much larger than λ. Note also that the cost of an input is much lower than a triple, as the input protocol does not require any of the expensive sacrificing or combining that we use to obtain active security with triples. This is in contrast to the SPDZ protocol [14, 15], where creating input tuples requires complex zero-knowledge or cut-and-choose techniques.

Comparison with a passive protocol. A passively secure (or semi-honest) version of our protocol can be constructed by setting $\tau = 1$ and removing the authentication step, saving 5 calls to Π_{COPEe} for every pair of parties. The communication cost of a single triple is then $n(n-1)(\lambda k+k^2)$ bits. For triples where $k \geq 128$, and 64-bit statistical security, the actively secure protocol achieves $\tau = 3$, so is just 5.5 times the cost of the passive variant.

Field bit length	Input cost (kbit)	Triple cost (kbit)
64	$4.16(n-1)$	$53.25n(n-1)$
128	$16.51(n-1)$	$180.22n(n-1)$
256	$65.79(n-1)$	$622.59n(n-1)$
512	$262.66(n-1)$	$2293.76n(n-1)$

Table 2: Communication cost of our protocols for various field sizes, with n parties.

7.2 Implementation

As part of our implementation, we have used the optimizations described below. The first two apply to the OT extension by Keller et al. [24].

Bit matrix transposition. Asharov et al. [2] mention the bit matrix transposition as the most expensive part of the computation for their OT extension. They propose Eklundh's algorithm to reduce the number of cache misses. Instead of transposing a matrix bit by bit, the matrix is transposed with respect to increasingly small blocks while leaving the blocks internally intact. Keller et al. also use this algorithm.

However, for security parameter λ, the OT extension requires the transposition of a $n \times \lambda$-matrix. We store this matrix as list of $\lambda \times \lambda$-blocks, and thus, we only have to transpose those blocks. For $\lambda = 128$, one such block is 2 KiB, which easily fits into the L1 cache of most modern processors.

Furthermore, we use the PMOVMSKB instruction from SSE2. It outputs a byte consisting of the most significant bits of 16 bytes in a 128-bit register. Together with a left shift (PSLLQ), this allows a 16×8-matrix to be transposed [30] with only 24 instructions (eight of PMOVSKB, PSLLQ, and MOV each).

Pseudorandom generator and hashing. Keller et al. [24] used AES-128 in counter mode to implement the PRG needed for the OT extension. This allows to use the AES-NI extension

provided by modern processors. We have also implemented the hash function using AES-128 by means of the Matyas–Meyer–Oseas construction [29], which was proven secure by Black et al. [6]. This construction uses the compression function $h_i = E_{g(h_{i-1})}(m_i) \oplus m_i$, where m_i denotes the i-th message block, h_i is the state after the i-th compression, and g denotes a conversion function. In our case, the input is only one block long (as many bits as the computational security parameter of the OT extension), and g is the identity. This gives a hash function $H(m) = E_{IV}(m) \oplus m$ for some initialization vector IV, which allows to precompute the key schedule. This precomputation in turn allows to easily take advantage of the pipelining capabilities of AES-NI in modern Intel processors: While the latency of the AESENC instruction is seven clock cycles, the throughput is one per clock cycle [21]. This means that the processor is capable of computing seven encryptions in parallel.

Inner product computation. Both Π_{COPEe} and Π_{Triple} involve the computation of $\langle \mathbf{g}, \mathbf{x} \rangle$ for $\mathbf{x} \in \mathbb{F}^{\log|\mathbb{F}|}$. Elements of both $\mathbb{F}_{2^k}$ and $\mathbb{F}_p$ are commonly represented as elements of larger rings ($\mathbb{F}_2[X]$ and $\mathbb{Z}$, respectively), and some operations involves a modular reduction (modulo an irreducible polynomial or p). When computing, we defer this reduction until after computing the sum. Furthermore, we use the `mpn_*` functions of MPIR [36] for the large integer operations for $\mathbb{F}_p$. For $\mathbb{F}_{2^k}$ on the other hand, the computation before the modular reduction is straightforward because addition in $\mathbb{F}_2[X]$ corresponds to XOR.

Multithreading. In order to make optimal use of resources, we have organized the triple generation as follows: There are several threads independently generating triples, and every such thread controls $n-1$ threads for the OTs with the $n-1$ other players. Operations independent of OT instances, such as amplification and sacrificing, are performed by the triple generation threads. We found that performance is optimal if the number of generator threads is much larger than the number of processor cores. This is an indication that the communication is the main bottleneck.

7.3 Experiments

We have tested our implementation for up to five parties on off-the-shelf machines (eight-core i7 3.1 GHz CPU, 32 GB RAM) in a local network. Fig. 7 shows our results.

We could generate up to 4800 and 1000 $\mathbb{F}_{2^{128}}$ triples per second with two and five parties, respectively. For $\mathbb{F}_p$ with p a 128-bit prime, the figures are the same. These figures come close to the maximum possible throughput of the correlation steps, which is 5500 and 1400, respectively. The maximum figures are computed from the analysis below, with $\tau = 3$ and $k = \lambda = 128$. Assuming a 1 Gbit/s link per party and unlimited routing capacity gives the desired result.

By increasing the bandwidth to 2 Gbit/s, we could increase the throughput to 9500 and 1600 triples per seconds for two and five parties, respectively. This confirms the observation that the communication is the main bottleneck. Fig. 8 shows the throughput for two parties in various network environments. The WAN environment was simulated over a LAN by restricting bandwidth to 50 Mbit/s and a round-trip latency of 100 ms.

7.3.1 Vickrey Auction

To highlight the practicality of our protocol, we have implemented the Vickrey second-price auction. Figure 9 shows the results for the offline and online phase run between two parties on a local network.

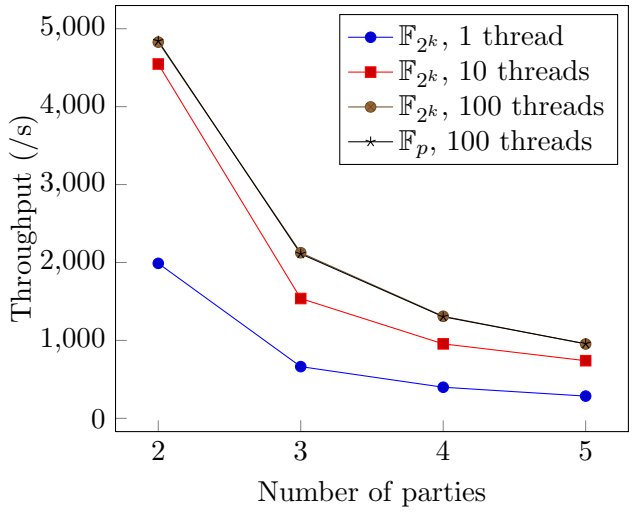

Figure 7: Triple generation throughput for 128-bit fields.

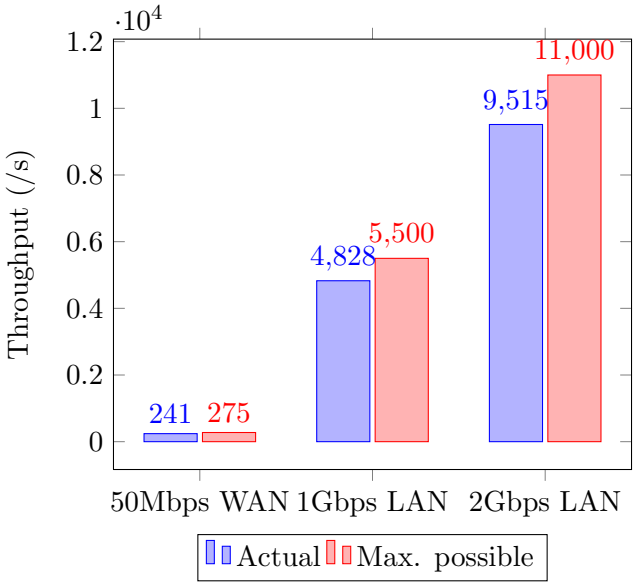

Figure 8: Throughput and maximum possible throughput for different networks with two parties

References

[1] The Sharemind project. http://sharemind.cs.ut.ee, 2007.

[2] Asharov, G., Lindell, Y., Schneider, T., and Zohner, M. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 535–548.

[3] Baum, C., Damgård, I., Toft, T., and Zakarias, R. Better preprocessing for secure multiparty computation. *IACR Cryptology ePrint Archive* (2016).

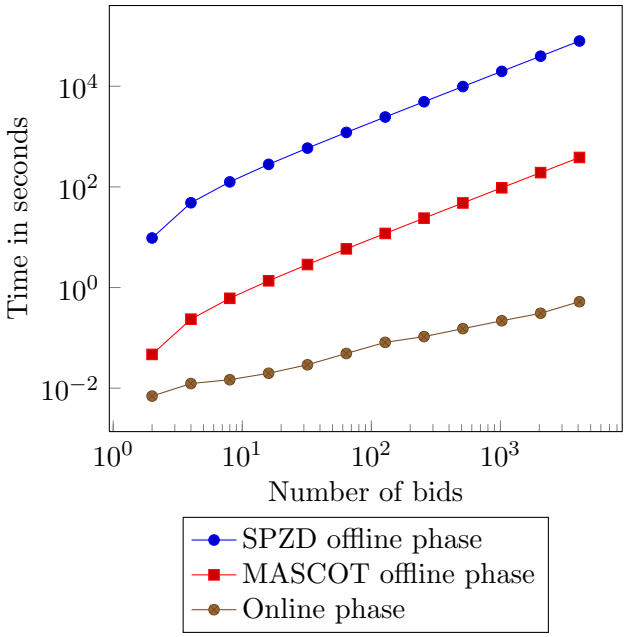

Figure 9: Vickrey auction run by two parties.

[4] Beaver, D. Efficient multiparty protocols using circuit randomization. *Advances in Cryptology - CRYPTO 1991* (1992).

[5] Bendlin, R., Damgård, I., Orlandi, C., and Zakarias, S. Semi-homomorphic encryption and multiparty computation. In *Advances in Cryptology - EUROCRYPT 2011* (2011), pp. 169–188.

[6] Black, J., Rogaway, P., Shrimpton, T., and Stam, M. An analysis of the blockcipher-based hash functions from PGV. *J. Cryptology 23*, 4 (2010), 519–545.

[7] Bogdanov, D., Jõemets, M., Siim, S., and Vaht, M. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers* (2015), pp. 227–234.

[8] Bogdanov, D., Kamm, L., Kubo, B., Rebane, R., Sokk, V., and Talviste, R. Students and taxes: a privacy-preserving social study using secure computation. *IACR Cryptology ePrint Archive* (2015).

[9] Burra, S. S., Larraia, E., Nielsen, J. B., Nordholt, P. S., Orlandi, C., Orsini, E., Scholl, P., and Smart, N. P. High performance multi-party computation for binary circuits based on oblivious transfer. Cryptology ePrint Archive, Report 2015/472, 2015. http://eprint.iacr.org/.

[10] Canetti, R. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA* (2001), pp. 136–145.

[11] Carter, L., and Wegman, M. N. Universal classes of hash functions. *J. Comput. Syst. Sci. 18*, 2 (1979), 143–154.

[12] Damgård, I., Damgård, K., Nielsen, K., Nordholt, P. S., and Toft, T. Confidential benchmarking based on multiparty computation. In *Financial Cryptography* (2016).

[13] Damgård, I., Keller, M., Larraia, E., Miles, C., and Smart, N. P. Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In *SCN* (2012), I. Visconti and R. D. Prisco, Eds., vol. 7485 of *Lecture Notes in Computer Science*, Springer, pp. 241–263.

[14] Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., and Smart, N. P. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS* (2013), J. Crampton, S. Jajodia, and K. Mayes, Eds., vol. 8134 of *Lecture Notes in Computer Science*, Springer, pp. 1–18.

[15] Damgård, I., Pastro, V., Smart, N. P., and Zakarias, S. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology – CRYPTO 2012* (2012), R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *Lecture Notes in Computer Science*, Springer, pp. 643–662.

[16] Damgård, I., Toft, T., and Zakarias, R. Fast multiparty multiplications from shared bits. *IACR Cryptology ePrint Archive* (2016).

[17] Frederiksen, T. K., Orsini, E., Keller, M., and Scholl, P. A unified approach to MPC with preprocessing using OT. In *Advances in Cryptology - ASIACRYPT 2015* (2015).

[18] Gilboa, N. Two party RSA key generation. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings* (1999), pp. 116–129.

[19] Goldreich, O., Micali, S., and Wigderson, A. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA* (1987), pp. 218–229.

[20] Impagliazzo, R., Levin, L. A., and Luby, M. Pseudo-random generation from one-way functions (extended abstracts). In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA* (1989), pp. 12–24.

[21] Intel. Intel intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide, 2016. Online; accessed February 2016.

[22] Ishai, Y., Kilian, J., Nissim, K., and Petrank, E. Extending oblivious transfers efficiently. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings* (2003), pp. 145–161.

[23] Kamm, L., and Willemson, J. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security 14*, 6 (2015), 531–548.

[24] Keller, M., Orsini, E., and Scholl, P. Actively secure OT extension with optimal overhead. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I* (2015), pp. 724–741.

[25] Keller, M., Scholl, P., and Smart, N. P. An architecture for practical actively secure MPC with dishonest majority. In *ACM Conference on Computer and Communications Security* (2013), pp. 549–560.

[26] Kreuter, B., Shelat, A., and Shen, C. Billion-gate secure computation with malicious adversaries. In *Proceedings of the 21st USENIX conference on Security symposium* (2012), USENIX Association, pp. 14–14.

[27] Larraia, E., Orsini, E., and Smart, N. P. Dishonest majority multi-party computation for binary circuits. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II* (2014), pp. 495–512.

[28] Lindell, Y., and Riva, B. Blazing fast 2pc in the offline/online setting with security for malicious adversaries. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015* (2015), pp. 579–590.

[29] Matyas, S. M., Meyer, C. H., and Oseas, J. Generating strong one-way functions with cryptographic algorithm. *IBM Technical Disclosure Bulletin 27*, 10A (1985), 5658–5659.

[30] mischasan. What is sse !@# good for? transposing a bit matrix. https://mischasan.wordpress.com/2011/07/24/what-is-sse-good-for-transposing-a-bit-matrix, 2011. Online; accessed February 2016.

[31] Nielsen, J. B. Extending oblivious transfers efficiently - how to get robustness almost for free. *IACR Cryptology ePrint Archive 2007* (2007), 215.

[32] Nielsen, J. B., Nordholt, P. S., Orlandi, C., and Burra, S. S. A new approach to practical active-secure two-party computation. In *Advances in Cryptology–CRYPTO 2012*. Springer, 2012, pp. 681–700.

[33] Peikert, C., Vaikuntanathan, V., and Waters, B. A framework for efficient and composable oblivious transfer. In *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings* (2008), pp. 554–571.

[34] Pinkas, B., Schneider, T., Segev, G., and Zohner, M. Phasing: Private set intersection using permutation-based hashing. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.* (2015), pp. 515–530.

[35] Pinkas, B., Schneider, T., and Zohner, M. Faster private set intersection based on OT extension. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.* (2014), pp. 797–812.

[36] The MPIR team. Multiple precision integers and rationals. https://www.mpir.org, 2016. Online; accessed February 2016.

[37] Yao, A. C. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986* (1986).

A Authentication and MAC checking security proof

We now prove security of the authentication and MAC checking protocol.

Theorem 5 (Theorem 1, restated)**.** *The protocol* $\Pi_{[\![\cdot]\!]}$ *securely implements* $\mathcal{F}_{[\![\cdot]\!]}$ *in the* $(\mathcal{F}_{\mathsf{COPEe}}, \mathcal{F}_{\mathsf{Comm}}, \mathcal{F}_{\mathsf{Rand}})$*-hybrid model, with statistical security parameter* $\log|\mathbb{F}| - 2\log\log|\mathbb{F}|$.

Proof. We describe a simulator, $\mathcal{S}$, that interacts with $\mathcal{F}_{[\![\cdot]\!]}$, such that no environment $\mathcal{Z}$ can distinguish between an interaction with $\mathcal{S}$ and $\mathcal{F}_{[\![\cdot]\!]}$ and an interaction with the real adversary $\mathcal{A}$ and $\Pi_{[\![\cdot]\!]}$. After describing the simulator we then argue indistinguishability of the real and ideal worlds.

$\mathcal{S}$ maintains several databases, LC for linear combinations, HS for simulated shares of honest parties in the open and check phases, and CS for the sums of shares of corrupted parties.

Simulating the Initialize phase. Emulating $\mathcal{F}_{\mathsf{COPEe}}$ instances between every pair of corrupt party P_i and honest party P_j, receive $\Delta^{(i,j)}$ input by P_i to the respective instance. Furthermore, sample $\Delta^{(i)} \xleftarrow{\$} \mathbb{F}$ for all $i \notin A$.

Simulating the Input phase.

- If P_j is corrupted, emulate the communication channels and instances of $\mathcal{F}_{\mathsf{COPEe}}$ with honest parties P_i, $i \notin A$. For the $\mathcal{F}_{\mathsf{COPEe}}$ instance with P_i, receive $\mathbf{x}_h^{(i,j)}$ from the adversary and set it to the input of P_j and output random $t_h^{(j,i)}$ to P_j for $i = 1, \ldots, n$ and $h = 0, \ldots, m$. For the checking, sample $\mathbf{r} \xleftarrow{\$} \mathbb{F}^{m+1}$ to emulate $\mathcal{F}_{\mathsf{Rand}}$, receive y from P_j, and receive $\{\sigma^{(i)}\}_{i\in A}$ emulating $\mathcal{F}_{\mathsf{Comm}}$. Compute $\sigma^{(i)} = \sum_{h=0}^{m} r_h \cdot (\langle \mathbf{g} * \mathbf{x}_h^{(i,j)}, \mathbf{\Delta}_B^{(i)} \rangle - t^{(i,j)})$ for all $i \notin A$. If $\sum_{i=1}^{n} \sigma_i \neq 0$, abort. Otherwise, solve

$$\sum_{i\notin A} \left\langle \mathbf{g} \cdot y - \mathbf{g} * \sum_{h=0}^{m} r_h \cdot \sum_{j\in A} \mathbf{x}_h^{(i,j)}, \tilde{\mathbf{\Delta}}_B^{(i)} \right\rangle = 0 \tag{3}$$

  for $\{\tilde{\mathbf{\Delta}}_B^{(i)}\}_{i\notin A}$ in $\mathbb{F}$. If $\sum_{i\notin A} \tilde{\Delta}^{(i)} = 0$ for all solutions, abort. Otherwise, for some $\{\tilde{\Delta}^{(i)}\}_{i\notin A}$ such that $\sum_{i\notin A} \tilde{\Delta}^{(i)} \neq 0$ and for each $h \in [m]$, compute

$$x_h = \Big(\sum_{i\notin A} \langle \mathbf{g}, \tilde{\mathbf{\Delta}}_B^{(i)} \rangle \Big)^{-1} \cdot \sum_{i\notin A} \sum_{j\in A} \langle \mathbf{g} * \mathbf{x}_h^{(i,j)}, \tilde{\mathbf{\Delta}}_B^{(i)} \rangle, \tag{4}$$

  and input $\{x_h\}_{h\in[m]}$ to $\mathcal{F}_{[\![\cdot]\!]}$ on behalf of P_j. We will show that, with overwhelming probability, x_h is unique over all possible $\tilde{\mathbf{\Delta}}^{(i)}$.

- Otherwise, emulate communication channels and instances of $\mathcal{F}_{\mathsf{COPEe}}$ with corrupted parties P_i, $i \in A$. For the channel with P_i, send a random share $x^{(i)}$, and for the $\mathcal{F}_{\mathsf{COPEe}}$ instance, send $q^{(i,j)}$ to $\mathcal{A}$, for $h = 0, \ldots, m$. Store $\sum_{i \in A} x_h^{(i)}$ and $\sum_{i \in A} m_h^{(i)} = \sum_{i \in A} x^{(i)} \cdot \Delta^{(i)} + q^{(i,j)}$ in CS. Sample $\mathbf{r} \xleftarrow{\$} \mathbb{F}^t$ to emulate $\mathcal{F}_{\mathsf{Rand}}$, sample $y \xleftarrow{\$} \mathbb{F}$, and send both to the corrupted parties. Compute $\sum_{i \in A} \sigma^{(i)} = \sum_{j=0}^{m} r_h \cdot \sum_{i \in A} m_h^{(i)} - y \cdot \Delta^{(i)}$. Then sample $\{\sigma^{(i)}\}_{i \notin A}$ in $\mathbb{F}$ such that $\sum_{i=1}^{n} \sigma^{(i)} = 0$ and use $\{\sigma^{(i)}\}_{i \notin A}$ to emulate $\mathcal{F}_{\mathsf{Comm}}$ just as in the Check phase below.

Simulating the Linear comb. phase. Store the linear combination in LC under $\overline{\mathsf{id}}$.

Simulating the Open phase. Receive the constraints on x from $\mathcal{F}_{[\![\cdot]\!]}$ and check if the value to be opened and the previously opened values form a linearly dependent set in the input values w.r.t. LC. If so, compute the shares of the honest parties accordingly from the entries in HS. Otherwise, sample random shares $\{x^{(i)}\}_{i \notin A}$ and MAC shares $\{m^{(i)}\}_{i \notin A}$ such that $\sum_i x^{(i)} = x$ and $\sum_i m^{(i)} = x \cdot \Delta$ with $\sum_{i \in A} x^{(i)}$ and $\sum_{i \in A} m^{(i)}$ taken from the relevant linear combination of values from CS. Emulating the broadcast channel send $\{x^{(i)}\}_{i \notin A}$ and receive $\{x^{(j)}\}_{j \in A}$. Compute $x = \sum_i x^{(i)}$ and input it to $\mathcal{F}_{[\![\cdot]\!]}$.

Simulating the Check phase. Sample $\mathbf{r} \xleftarrow{\$} \mathbb{F}^t$ to emulate $\mathcal{F}_{\mathsf{Rand}}$ and send it to the corrupted parties. Emulating $\mathcal{F}_{\mathsf{Comm}}$, receive σ_i from corrupted party P_i for $i \in A$.

For $j \in [t]$, compute x'_j and m'_j as the respective linear combination from values in CS. Furthermore, look up $m_j^{(i)}$ in HS for all $i \notin A$. For $i \notin A$, compute $\sigma_i = \sum_{j=1}^{t} r_j \cdot (m_j^{(i)} - \Delta^{(i)} \cdot x_j)$ and complete the emulation of $\mathcal{F}_{\mathsf{Comm}}$. If $\sum_i \sigma_i = 0$, input OK to $\mathcal{F}_{[\![\cdot]\!]}$ and $\perp$ otherwise.

Indistinguishability. Now we argue indistinguishability. The $\mathsf{LinComb}$ command does not require communication, and the $\mathsf{Initialize}$ command only involve sending random shares and using $\mathcal{F}_{\mathsf{COPEe}}$, which only outputs random information from the point of view of a single party. Therefore, the simulation of these commands is easily seen to indistinguishable. It remains to discuss Input, Open, and Check.

If P_j in the Input phase is not corrupted, it is easy to see that adversary only learns random information. $\mathcal{F}_{\mathsf{COPEe}}$ only outputs random shares, y has $r_0 \cdot x_0$ as summand for random single-use x_0, and Π_{MACCheck} only reveals a random secret sharing of zero because it also contains one-time randomness in the MAC of x_0. The simulation therefore simply generates the required randomness.

More intricate is the simulation for a corrupted P_j. While it is easy to simulate all information sent to the adversary, the simulation aborts if there is no adequate solution of (3) to be used in (4). A solution is inadequate if $\sum_{i \notin A} \tilde{\Delta}^{(i)} = 0$. Clearly, there exist $|\mathbb{F}|^{n-|A|-1}$ inadequate solutions. Since every such solution corresponds to a choice of $\{\Delta^{(i)}\}_{i \notin A}$ where the MAC check succeeds, the inexistence of an adequate solution means that the success probability of the adversary is at most $2^{-\log |\mathbb{F}|}$ because there are $|\mathbb{F}|^{n-|A|}$ possible choices of $\{\Delta^{(i)}\}_{i \notin A}$.

In the Open procedure, corrupted parties learn the honest parties shares. Using HS ensures that all the openings are consistent. Furthermore, the sampling constraints ensure that the simulated honest parties' shares and the correct corrupted parties' shares sum up the correct value. Finally, $\mathcal{F}_{[\![\cdot]\!]}$ allows the adversary to determine the honest parties' outputs, which the simulator uses with values computed as in the real protocol.

The most intricate phase is the Check procedure. While it is straightforward to simulate σ_i sent by an honest party P_i, the indistinguishability of the abort behavior requires further discussion. The idea of our proof is that, once a corrupted player has passed the MAC check in the Input phase, they only can pass the MAC check for a specific value for each of their inputs, namely x_h computed in (4). In the following, we will focus on $\mathbb{F} = \mathbb{F}_{2^k}$. Later we will discuss $\mathbb{F} = \mathbb{F}_p$. Adapting (1) to the input phase gives that, in order to pass the MAC check, the adversary has to send $\{\sigma^{(i)}\}_{i\in A}$ such that

$$\begin{aligned} -\sum_{i\in A}\sigma^{(i)} &= \sum_{i\notin A}\sigma^{(i)} \\ &= \sum_{i\notin A} m^{(i)} - y\cdot\Delta^{(i)} \\ &= \sum_{i\notin A}\Big(\Big\langle \mathbf{g}\cdot y + \mathbf{g} * \sum_{j\in A}\sum_{h=0}^{m} r_h\cdot \mathbf{x}_h^{(i,j)}, \boldsymbol{\Delta}_B^{(i)}\Big\rangle + R_i\Big), \end{aligned} \tag{5}$$

where R_i is computed as the $\{r_h\}_{h=0}^m$-weighted sum from the equivalent rest terms in (1). Assuming that the above equality is satisfied for a different $\{\bar{\Delta}^{(i)}\}_{i\notin A}$, we get (3) for $\tilde{\Delta}^{(i)} = \Delta^{(i)} - \bar{\Delta}^{(i)}$ for all $i \notin A$. This proves that the set S_Δ of $\{\Delta^{(i)}\}_{i\notin A}$ fulfilling (5) is an affine subspace of $\mathbb{F}_2^{n-|A|}$.

Clearly, (4) provides a solution for $y = \sum_{h=0}^m r_h x_h$ such that

$$\Big\langle \mathbf{g}\cdot x_h - \mathbf{g} * \sum_{j\in A}\mathbf{x}_h^{(i,j)}, \tilde{\boldsymbol{\Delta}}_B^{(i)}\Big\rangle = 0 \tag{6}$$

holds for all $h \in [m]$ and some $\{\tilde{\Delta}^{(i)}\}_{i\notin A} \in \tilde{S}_\Delta$, where the latter denotes the linear space parallel to S_Δ. We have to prove that this is the only solution for a sufficiently large subspace of $\tilde{S}_\Delta$, otherwise the adversary has two sets of $\{x_h\}_{h\in[m]}$ to choose from later. Assume now that, for every $f \in [l]$ for some $l \in \mathbb{N}$, there is a different set $\{x_{f,h}\}_{h\in[m]}$ with $\sum_{i=1}^m r_h \cdot x_{f,h} = y$ and

$$\Big\langle \mathbf{g}\cdot x_{f,h} - \mathbf{g} * \sum_{j\in A}\mathbf{x}_h^{(i,j)}, \tilde{\boldsymbol{\Delta}}_{f,B}^{(i)}\Big\rangle = 0 \tag{7}$$

for all $\{\tilde{\Delta}_f^{(i)}\}_{i\notin A} \in \tilde{S}_f \subset \tilde{S}_\Delta$ such that $|\tilde{S}_f| > 2^{(n-|A|-1)\log|\mathbb{F}|}$. The latter condition is required for the adversary to be successful with probability more than $2^{-\log|\mathbb{F}|}$ at the later opening. Since $\tilde{S}_f$ clearly is a linear space for all $f \in [l]$, and $\tilde{S}_f \cap \tilde{S}_{f'} = \{0\}$ by definition, and $|\tilde{S}_\Delta| \le 2^{(n-|A|)\log|\mathbb{F}|}$ by definition, $l \le \log|\mathbb{F}|$.

Let $f \neq f' \in [l]$. Then,

$$\sum_{i\notin A}\Big\langle \mathbf{g}\cdot y - \mathbf{g} * \sum_{h=0}^{m} r_h \cdot \sum_{j\in A}\mathbf{x}_h^{(i,j)}, \tilde{\boldsymbol{\Delta}}_B^{(i)}\Big\rangle = 0$$

for all $\{\tilde{\Delta}^{(i)}\}_{i\notin A}$ implies that

$$\sum_{h=0}^{m} r_h \cdot \sum_{i\notin A}\Big\langle \mathbf{g} * \sum_{j\in A}\mathbf{x}_h^{(i,j)}, \tilde{\boldsymbol{\Delta}}_{f,B}^{(i)} - \tilde{\boldsymbol{\Delta}}_{f',B}^{(i)}\Big\rangle = 0$$

for all $\{\Delta_f^{(i)}\}_{i\notin A} \in \tilde{S}_f$ and $\{\Delta_{f'}^{(i)}\}_{i\notin A} \in \tilde{S}_{f'}$. Using (7), we get

$$\sum_{h=0}^{m} r_h \cdot \Big(x_{f,h} \cdot \sum_{i\notin A} \tilde{\Delta}_f^{(i)} - x_{f',h} \cdot \sum_{i\notin A} \tilde{\Delta}_{f'}^{(i)} \Big) = 0. \tag{8}$$

By definition, there exists $\bar{h} \in \{0,\dots,m\}$ such that $x_{f,h} \neq x_{f',h}$. Furthermore, a simple counting argument shows that $\sum_{i\notin A} \tilde{\Delta}_f^{(i)}$ and $\sum_{i\notin A} \tilde{\Delta}_{f'}^{(i)}$ each have at least two results for $\{\tilde{\Delta}_f^{(i)}\}_{i\notin A} \in \tilde{S}_f$ and $\{\tilde{\Delta}_{f'}^{(i)}\}_{i\notin A} \in \tilde{S}_{f'}$. It follows that

$$\Big(x_{f,h} \cdot \sum_{i\notin A} \tilde{\Delta}_f^{(i)} - x_{f',h} \cdot \sum_{i\notin A} \tilde{\Delta}_{f'}^{(i)} \Big) \neq 0$$

for some $\{\tilde{\Delta}_f^{(i)}\}_{i\notin A} \in \tilde{S}_f$ and $\{\tilde{\Delta}_{f'}^{(i)}\}_{i\notin A} \in \tilde{S}_{f'}$. Therefore, by applying the principle of deferred decisions, the probability of (8) is $2^{-\log|\mathbb{F}|}$ over the choice of $\{r_h\}_{h=0}^m$. Given that there are less than $(\log|\mathbb{F}|)^2$ pairs $f \neq f' \in [l]$, the overall probability is at most $(\log|\mathbb{F}|)^2 \cdot 2^{-\log|\mathbb{F}|} = 2^{-\log|\mathbb{F}|+2\log\log|\mathbb{F}|}$.

We have established that, for every $h \in [m]$, there exists a unique x_h where the adversary can compute

$$\sum_{i\notin A} \langle \mathbf{g} \cdot x_h - \mathbf{g} * \sum_{j\in A} \mathbf{x}_h^{(i,j)}, \mathbf{\Delta}_B^{(i)} \rangle.$$

(1) shows that this term is viable to passing the MAC check. Furthermore, if the computing this term for a different x'_h is equivalent to guessing $\sum_{i\notin A} \Delta^{(i)}$ because the difference between the terms is

$$\langle \mathbf{g} \cdot x_h - \mathbf{g} \cdot x'_h, \mathbf{\Delta}_B^{(i)} \rangle = (x_h - x'_h) \cdot \sum_{i\notin A} \Delta^{(i)}.$$

Since $\sum_{i\notin A} \Delta^{(i)}$ is uniformly random the probability of this happening is $2^{-\log|\mathbb{F}|}$. □

We now turn to the discussion of $\mathbb{F} = \mathbb{F}_p$. The main difference to the case of $\mathbb{F} = \mathbb{F}_{2^k}$ is that there is no bijection between $\Delta \in \mathbb{F} = \mathbb{F}_p$ and $\mathbf{\Delta} \in \mathbb{F}_p^{\log p}$. While there are canonical maps both ways, bit decomposition from $\mathbb{F}_p$ to $\mathbb{F}_p^{\log p}$ and $\langle \mathbf{g}, \mathbf{\Delta} \rangle$ from $\mathbb{F}_p^{\log p}$ to $\mathbb{F}_p$, the former is not surjective and the latter not injective. This implies that the solutions of (5) or (6) are not necessarily vectors of bits rather than elements of $\mathbb{F}_p$. Nevertheless, the lemma below proves that, if $\tilde{S}_f$ contains at least $2^{(n-|A|-1)\log|\mathbb{F}|}$ vectors consisting only of bits (which is necessary for an adversary to pass the MAC check), then it has dimension at least $(n-|A|-1)\log|\mathbb{F}|$ for all f. Together with the fact that $\tilde{S}_\Delta$ has dimension at most $(n-|A|)\log|\mathbb{F}|$ by definition and $\tilde{S}_f \cap \tilde{S}_{f'} = \{0\}$ for $f \neq f' \in [l]$, it follows that $l \leq \log|\mathbb{F}|$ as above. □

Lemma 1. *Let V be subspace of $\mathbb{F}_p^k$ containing 2^l elements of $\{0,1\}^k$. Then the dimension of V is at least l.*

Proof. We prove that if the dimension of V is l, it cannot contain more than 2^l elements of $\{0,1\}^k$. There exists a basis $\mathbf{v}_1,\dots,\mathbf{v}_l$ such that $\mathbf{v}_{i+1}$ starts with more zeroes than $\mathbf{v}_i$ for all $i = 1,\dots,l-1$. Such a basis can be constructed from any basis using Gaussian elimination. Every element of V has the form $\sum_{i=1}^l a_i \mathbf{v}_i$. Now consider the following algorithm for generating an element of $V \cap \{0,1\}^k$. For $i \in [l]$, assume that $\{a_j\}_{1\leq j<i}$ have already been chosen. Furthermore let g such that the g-th element of $\mathbf{v}_i$ is not zero but the g-th element of $\mathbf{v}_{i+1},\dots,\mathbf{v}_l$ is. This exists by definition. Then, there exist only two choices of a_i in order to let the g-th element of $\sum_{i=1}^l a_i \mathbf{v}_i$ be in $\{0,1\}$. Iterating over $a_1,\dots,a_l$, this proves that the size of $V \cap \{0,1\}^k$ is at most 2^l. □

□

B Triple generation security proof

Here we give a proof of security of Theorem 2. Before we need to recall basic facts and definition about entropy.

B.1 Entropy definitions.

Given a probability distribution X over a sample space S, we denote by P_X the probability distribution of X.

Definition 1. *Let X be a discrete probability distribution. The* min-entropy *of X is defined as*

$$H_\infty(X) = -\log\left(\max_x \Pr\left[X = x\right]\right)$$

Intuitively, the min-entropy of a distribution is a measure of how predictable the distribution is. We now state some basic properties of min-entropy that easily follow from the definition:

Proposition 1.

1. *If U is the uniform distribution over a sample space S, then*
$$H_\infty(U) = \log|S|$$
2. *If X is the joint distribution of $X_1, \ldots, X_n$, then there exists $i \in [n]$ such that*
$$H_\infty(X_i) \geq H_\infty(X)/n$$
3. *Let X and Y be independent distributions over a finite field $\mathbb{F}$. Then*
$$H_\infty(X + Y) \geq \max(H_\infty(X), H_\infty(Y))$$

We will also use the concept of *universal hashing*, due to Carter and Wegman [11].

Definition 2. *Let T be a set and $\mathcal{H} = \{h_t\}_{t \in t}$ be a family of keyed hash function $h_t : \{0,1\}^n \to \{0,1\}^k$. Then $\{h_t\}_{t \in T}$ is a* 2-universal hash function *family, if for every $x, y \in \{0,1\}^n$ such that $x \neq y$, we have that*

$$\Pr_{t \in T}[h_t(x) = h_t(y)] \leq 2^{-k}.$$

The following is a version of the Leftover Hash Lemma, phrased over finite fields.

Lemma 2 (Leftover Hash Lemma [20])**.** *Let S and T be two sets, and $\mathbb{F}$ a finite field. Let X be a random variable over S and $\mathcal{H} = \{h_t\}_{t \in T}$, $h_t : S \to \mathbb{F}$, a 2-universal hash function. Let U_S and U_T be the uniform distribution over S and T, respectively. If*

$$H_\infty(X) \geq 2\kappa + \log_2|\mathbb{F}|$$

then for $t \xleftarrow{\$} T$ (independent of X), we have

$$(h_t(X), U_t) \overset{s}{\approx} (U_s, U_t)$$

for statistical security parameter κ.

B.2 Proof of Theorem 2

Theorem 6 (Theorem 2, restated)**.** *The protocol* Π_{Triple} *(Protocol 4) securely implements* $\mathcal{F}_{\mathsf{Triple}}$ *in the* $(\mathcal{F}_{\mathsf{ROT}}, \mathcal{F}_{[\![\cdot]\!]})$*-hybrid model, with statistical security parameter* $\log_2 |\mathbb{F}|$*, for* $\tau = 4$*.*

Proof. Let $\mathcal{A}$ be a real world adversary corrupting up to $n-1$ parties and $A \subset \mathcal{P}$ the set of corrupt parties. We describe a simulator $\mathcal{S}$ for $\mathcal{A}$ who interacts with $\mathcal{F}_{\mathsf{Prep}}$ and simulates each received message of $\mathcal{A}$ in the protocol Π_{Triple} from the honest parties and from the other functionalities, stage by stage.

Simulating the Multiply phase. The simulator emulates $\mathcal{F}_{\mathsf{ROT}}^{\tau k,k}$ and sends $\mathbf{q}_0^{(j,i)}, \mathbf{q}_1^{(j,i)}, j \in A$ to $\mathcal{A}$. Then for each $j \in A$, $\mathcal{S}$ receives $d^{(j,i)}$ by $\mathcal{A}$, for each $j \neq i$, sets $b^{(j,i)} = d^{(j,i)} - q_{0,h}^{(j,i)} + q_{1,h}^{(j,i)}$, $h \in [\tau \cdot k]$, and sends random $d_h^{(i,j)}$, $i \notin A$ to $\mathcal{A}$.
If P_j, $j \in A$, gives any inconsistent $b^{(j,i)}$, then $\mathcal{S}$ computes $\boldsymbol{\delta}_b[h]^{(j,i)}$ and $\sum_{j \in A} \boldsymbol{\delta}_b[h]^{(j,i)} = \boldsymbol{\delta}_b[h]^{(i)}$.
If some P_j inputs inconsistent values $\mathbf{a}^{(j)}$, when playing with P_i, $i \notin A$, then $\mathcal{S}$ computes $\boldsymbol{\delta}_a[h]^{(j,i)}$ and $\sum_{j \in A} \boldsymbol{\delta}_a^{(j,i)} = \boldsymbol{\delta}_a[h]^{(i)}$.

Simulating the Combining phase. All the computations are local, so $\mathcal{S}$ just emulates $\mathcal{F}_{\mathsf{Rand}}$ and proceeds according to the protocol.

Simulating the Authentication phase. Now $\mathcal{S}$ emulates $\mathcal{F}_{[\![\cdot]\!]}$ with inputs from the corrupt parties provided by $\mathcal{A}$. So if some inputs are inconsistent with previous computation, $\mathcal{S}$ computes $e_{Auth}, \hat{e}_{Auth}$, i.e. the deviation introduced by $\mathcal{A}$ in this step. Note that here $e_{Auth}, \hat{e}_{Auth} \neq 0$ essentially means that the adversary authenticates values different from those computed in the previous phases. If $\mathcal{F}_{[\![\cdot]\!]}$ aborts, then $\mathcal{S}$ sends Abort to $\mathcal{F}_{\mathsf{Prep}}$.

Simulating the Sacrifice step. The simulator emulates the functionalities $\mathcal{F}_{\mathsf{Rand}}$ and $\mathcal{F}_{[\![\cdot]\!]}.\mathsf{Open}$ honestly. Emulating $\mathcal{F}_{[\![\cdot]\!]}.\mathsf{Check}$, $\mathcal{S}$ aborts randomly depending on how many errors there are.

Indistinguishability. Now we argue indistinguishability. During the **Multiply** command, in both the simulated and the hybrid model, $\mathcal{Z}$ can see the mask $d_h^{(i,j)}$, for each $i \notin A$, but they look perfectly random as the values $q_{1,h}^{(i,j)}$ are uniformly random and never revealed to $\mathcal{Z}$.
Then the **Amplify/Combine** command do not require communication and in the **Authenticate** command the simulator honestly runs $\mathcal{F}_{[\![\cdot]\!]}$, so the view of $\mathcal{Z}$ up to the point where the values ρ and σ are partially opened in the **Sacrifice** step, has exactly the same distribution in both the execution. In the partial openings, we need to prove that the values produced in the real world and the simulated random values are indistinguishable.

Let us consider $a = \langle \mathbf{a}, \mathbf{r} \rangle$ and $\hat{a} = \langle \mathbf{a}, \hat{\mathbf{r}} \rangle$, and let X be the joint distribution of $(a_1, \ldots, a_\tau)$. Applying the Leftover Hash Lemma (Lemma 2), with $S = \mathbb{F}^\tau, T = \mathbb{F}^\tau$ and $h_{r,\hat{r}} : \mathbb{F}^\tau \to \mathbb{F}^2$ be defined by

$$h_{r,\hat{r}}(a_1, \ldots, a_\tau) = \left(\sum_{i=1}^{\tau} r_i \cdot a_i, \sum_{i=1}^{\tau} \hat{r}_i \cdot a_i \right),$$

we have that the output of h is statistically close to uniform, for statistical security parameter κ, provided that

$$H_\infty(X) \geq 2\kappa + 2\log|\mathbb{F}|. \tag{9}$$

In this way, if (9) is satisfied, it is easy to see that the partially opened value ρ in the real protocol is statistically indistinguishable from the uniformly random value used in the simulation.

Now we consider the probability of passing the sacrificing step. We recall that, from Equation (2), after authentication parties obtain values $[\![b]\!], [\![a]\!], [\![c]\!], [\![\hat{a}]\!], [\![\hat{c}]\!]$, which can be seen as follows:

$$c = a \cdot b + e_a + e_b + e_{Auth}$$

and

$$\hat{c} = \hat{a} \cdot b + \hat{e}_b + \hat{e}_a + \hat{e}_{Auth}.$$

First of all, if no Abort occurs, we obtain a correct triple in both the worlds with high probability, as stated by the following claim which shows that after sacrificing the additive error in a triple must be zero.

Claim 1. *Let A be the set of the corrupt parties and $B = \mathcal{P} \setminus A$. If the sacrificing step passes then*

$$e = e_a + e_b + e_{Auth} = 0$$

and

$$\hat{e} = \hat{e}_a + \hat{e}_b + \hat{e}_{Auth} = 0$$

with high probability.

Proof. This is easy to see, following the same argument used for triple generation with SHE in [15]. In particular, rewriting the value σ in the second opening of sacrificing as $r \cdot (c - a \cdot b) - (\hat{c} - \hat{a} \cdot b)$, and assuming that $e, \hat{e} \neq 0$, then the probability of satisfying the check is $\log|\mathbb{F}|$, since there is only one random challenge $r \in \mathbb{F}$ for which σ would be zero. □

We have shown that if the sacrifice test passes then $e = 0$ and the output triple is correct. However this could happen even if e_a, e_b and e_{Auth} are not (all) zero.

Claim 2. *If the sacrificing step passes then $\boldsymbol{\delta}_a[h]^{(i)} = 0$, for all $i \notin A$ and $h \in [\tau]$, with high probability.*

Proof. Suppose that $\{\boldsymbol{\delta}_a^{(i)}[h]\}_{h,i}$ are not all zero. If the sacrificing passes, then by Claim 1,

$$-e_{Auth} = \sum_{h \in \tau} r[h] \cdot \Big(\sum_{i \notin A} b^{(i)} \cdot \boldsymbol{\delta}_a^{(i)}[h] + \sum_{i \notin A} \langle \mathbf{a}_B^{(i)}[h], \boldsymbol{\delta}_b[h]^{(i)} \rangle\Big),$$

where $\{b^{(i)}\}_{i \notin A}$ are uniformly random in $\mathbb{F}$ and $\{\mathbf{a}_B[h]^{(i)}\}_i$ and e_{Auth} are independent of $\{b^{(i)}\}_i$. So the probability of passing the check is the same as the probability of guessing $b^{(i)}$, i.e. $1/|\mathbb{F}|$. □

Consider now the error

$$e_a = \sum_{h \in \tau} r_h \cdot \sum_{i \notin A} \langle \mathbf{a}_B^{(i)}[h], \boldsymbol{\delta}_b[h]^{(i)} \rangle,$$

and let $m = n - |A|$ be the number of honest parties, and S the set of all possible honest shares $(\mathbf{a}_B^{(i)})_{i \notin A}$ of $\mathbf{a}_B$, which are determined by the adversarial errors, and for which the sacrifice would

The Functionality $\mathcal{F}_{\mathsf{Comm}}$

Commit: On input $(\mathsf{Comm}, v, i, \tau_v)$ by P_i or the adversary on his behalf (if P_i is corrupt), where v is either in a specific domain or $\perp$, it stores (v, i, τ_v) on a list and outputs (i, τ_v) to all parties and adversary.

Open: On input $(\mathsf{Open}, i, \tau_v)$ by P_i or the adversary on his behalf (if P_i is corrupt), the ideal functionality outputs (v, i, τ_v) to all parties and adversary. If $(\mathsf{NoOpen}, i, \tau_v)$ is given by the adversary, and P_i is corrupt, the functionality outputs $(\perp, i, \tau_v)$ to all parties.

Figure 10: Ideal Commitments

pass. The value $(\mathbf{a}_B^{(i)})_{i \notin A}$ is uniformly distributed in S, and so its min-entropy is $\log|S|$. This means that there exists an i such that $\mathbf{a}_B^{(i)}$ has min-entropy at least $\log|S_|/m$. Since at least one $\mathbf{a}_B^{(i)}$ has min-entropy $\log|S|/m$, and each $\mathbf{a}_B^{(i)}$ is independent, it follows that the shared value $\mathbf{a}_B = \sum_{i \in [n]} \mathbf{a}_B^{(i)}$ has min-entropy at least $\log|S|/m$.

Also, let β be the probability of passing the sacrifice, so $\beta := \frac{|S|}{2^{mk\tau}}$, since $(\mathbf{a}_B^{(i)})_{i \notin A}$ is chosen at random from a set of size $2^{mk\tau}$. Writing $\beta = 2^{-c}$ for some $c \geq 0$, we get

$$\begin{aligned} H_\infty(\mathbf{a}) &\geq \frac{\log|S|}{m} = \frac{\log(\beta \cdot 2^{mk\tau})}{m} \\ &= k \cdot \tau - \frac{c}{m} \geq k \cdot \tau - c \end{aligned} \tag{10}$$

Noting that $k = \lceil \log|\mathbb{F}| \rceil$, and using (10) and (9), we obtain $\kappa = (k(\tau - 2) - c)/2$. Now the overall distinguishing probability of the environment (ignoring the failure events in the previous claims that occur with negligible probability) is obtained by multiplying the probability of passing the sacrifice check and the probability of distinguishing the output distribution from random (given that the sacrifice passed), so this is given by

$$\beta \cdot 2^{-\kappa} = 2^{-\kappa - c} = 2^{-k(\tau-2)/2 - c/2}$$

For this to be no more than 2^{-k} (for any $c \geq 0$) it suffices to set the number of triples to combine to $\tau = 4$. Note that if the field size is much larger than the statistical security parameter, say $k = 2\kappa$, then we only need distinguishing probability $\leq 2^{-k/2}$, so could combine just $\tau = 3$ triples to ensure security. □

The final analysis in this proof also gives the following special case as a corollary.

Corollary 2 (Corollary 1, restated)**.** *For $k \geq 128$ and $\tau = 3$, Π_{Triple} securely implements $\mathcal{F}_{\mathsf{Triple}}$ with statistical security parameter 64.*

C Other functionalities

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

Random sample: Upon receiving $(rand; u)$ from all parties, it samples a uniform $r \in \mathbb{F}$ and outputs $(rand, r)$ to all parties.

Figure 11: Functionality $\mathcal{F}_{\mathsf{Rand}}^{\mathbb{F}}$