MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer

Marcel Keller\textsuperscript{*} Emmanuela Orsini\textsuperscript{†} Peter Scholl\textsuperscript{‡}
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.

\textsuperscript{*}Supported by EPSRC via grant EP/M016803.
\textsuperscript{†}Supported by ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO.
\textsuperscript{‡}Supported by the DARPA Brandeis program and the US Navy under contract #N66001-15-C-4070.
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.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Field</th>
<th>Comms. (kbit)</th>
<th>Throughput, $n = 2$ (/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPDZ (active)</td>
<td>$F_p$, 128-bit</td>
<td>$215n(n - 1)$</td>
<td>23.5</td>
</tr>
<tr>
<td></td>
<td>$F_{240}$</td>
<td>$2272n(n - 1)$</td>
<td>3.68</td>
</tr>
<tr>
<td>SPDZ (covert, pr. 1/10)</td>
<td>$F_p$, 128-bit</td>
<td>$66n(n - 1)$</td>
<td>204</td>
</tr>
<tr>
<td></td>
<td>$F_{240}$</td>
<td>$844n(n - 1)$</td>
<td>31.9</td>
</tr>
<tr>
<td>Ours (active)</td>
<td>$F_p$, 128-bit</td>
<td>$180n(n - 1)$</td>
<td>4842</td>
</tr>
<tr>
<td></td>
<td>$F_{2128}$</td>
<td>$180n(n - 1)$</td>
<td>4827</td>
</tr>
</tbody>
</table>

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].

\(^1\)For $F_{240}$ in SPDZ with covert security, we could not find precise figures so the throughput in Table 1 is estimated based on other results.
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 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) frame-
work 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 \( \lambda \) and \( \kappa \) 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 \( \kappa \), and \( \mu \in \text{poly}(k) \) 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 \( x, y \) are vectors over \( \mathbb{F} \), then \( x \ast y \) denotes 
the component-wise products of the vectors. We denote by \( a \overset{\$}{\leftarrow} A \) 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 \( g \) consisting 
of the powers of two (in \( \mathbb{F}_p \)) or powers of \( X \) (in \( \mathbb{F}_{2^k} \)), so that 
\[
g = (1, g, g^2, \ldots, g^{k-1}) \in \mathbb{F}^k,
\]

where \( g = 2 \) in \( \mathbb{F}_p \) and \( g = X \) in \( \mathbb{F}_{2^k} \). Let \( g^{-1} : \{0, 1\}^k \rightarrow \{0, 1\}^k \) be the ‘bit decomposition’ function 
that maps \( x \in \mathbb{F} \) to a bit vector \( x_B = g^{-1}(x) \in \{0, 1\}^k \), such that \( x_B \) can be mapped back to \( \mathbb{F} \) by 
taking the inner product \( \langle g, g^{-1}(x) \rangle = x \). 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:
\( \mathcal{F}_{OT}^{1,k} : ((s_0, s_1), b) \mapsto (\perp, s_b) \)
\( \mathcal{F}_{ROT}^{1,k} : (\perp, b) \mapsto ((r_0, r_1), r_b) \),

where \( r_0, r_1 \overset{\$}{\leftarrow} \{0,1\}^k \), and \( b \in \{0,1\} \) is the receiver’s input. We use the notation \( \mathcal{F}_{OT}^{m,k}, \mathcal{F}_{ROT}^{m,k} \) 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, \( \Pi_{MACCheck} \), in Protocol 2, which verifies the MAC relation whilst keeping the global MAC key \( \Delta \) secret, ensuring that values cannot be opened incorrectly except by guessing the key \( \Delta \).

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, \( \Delta \), 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 \Pi_{COPE_e} (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 \( t_i \in \mathbb{F} \) 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

\[ q_i = t_i + b_i \cdot a, \]

where \( q_i \) is the receiver’s output in the \( i \)-th OT. Now both parties simply compute the inner product of their values \( (q_i), (-t_i) \) with the gadget vector \( g \) 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 \( \lambda \)-bit seeds (recall that \( \lambda \) 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 \( \Delta_B \). 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 \( g \) 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 \Pi_{COPE_e} if the parties do not follow the protocol. Party \( P_B \) fixes their input \( \Delta \) 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
Protocol 1 The protocol $\Pi_{\text{COPE}e}$: Oblivious correlated product evaluation with errors over the finite field $F$.

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

**Initialize:** This initializes the finite field $F$ and $P_A$’s correlation, $\Delta$.

1. $P_A$ samples $k$ pairs of seeds, $\{(k_0^i,k_1^i)\}_{i=1}^k$, each in $\{0,1\}^\lambda$.
2. $P_B$ inputs $\Delta \in F$, let $\Delta_B = (\Delta_0,\ldots,\Delta_{k-1}) \in \{0,1\}^k \subset F^k$.
3. Both parties call $F^k_{\text{OT}}$ with inputs $\{k_0^i,k_1^i\}_{i\in[k]}$ and $\Delta_B$.
4. $P_B$ receives $k^i_\Delta$ for $i \in [k]$.

**Extend:** On input $x \in F$ from $P_A$:

1. Expand $k_0^i$ and $k_1^i$ using the PRG in a stateful way, that is, $G(k_0^i)$ creates fresh randomness in every call. Obtain
   
   \[ t_0^i = G(k_0^i) \in F \quad \text{and} \quad t_1^i = G(k_1^i) \in F, \quad i \in [k]. \]

   so $P_A$ knows $(t_0^i,t_1^i)$ and $P_B$ knows $t^i_\Delta$, for $i = 1,\ldots,k$.
2. $P_A$ computes $u^i = t_0^i - t_1^i + x \in F$, for $i = 1,\ldots,k$ and sends these to $P_B$.
3. $P_B$ computes
   
   \[ q^i = \Delta_i \cdot u^i + t^i_\Delta, \]
   \[ = t_0^i + \Delta_i \cdot x \in F \]
   
   for $i = 1,\ldots,k$.
4. Let $q = (q^1,\ldots,q^k)$ and $t = (t_0^1,\ldots,t_0^k)$. Note that
   
   \[ q = t + x \cdot \Delta_B \in F^k. \]
5. $P_B$ sets $q = \langle g, q \rangle$.
6. $P_A$ sets $t = -\langle g, t \rangle$.
7. Now it holds that $t + q = x \cdot \Delta \in F$.

$u^i$ 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

\[ q = t + x \cdot \Delta_B, \]

where $x = (x^1,\ldots,x^k)$. We do not prevent this in our protocol, but instead model this behavior in the functionality $F_{\text{COPE}e}$ (Fig. 2). When we use $F_{\text{COPE}e}$ 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 $F_{\text{COPE}e}$ in the $F_{\text{OT}}$-hybrid model, is straightforward, since any deviations by an adversary in the protocol directly correspond to errors introduced in the functionality, and the receiver’s input $x$ is computationally hidden by the output of the PRG when the receiver sends the $u^i$ values. We therefore omit the precise details of the simulation, noting that it is essentially the same of that given by Nielsen [31].
**Functionality $\mathcal{F}_{\text{COPE}_e}$**

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

**Initialize($\mathbb{F}$):** Upon receiving $\Delta \in \mathbb{F}$ from $P_B$, the functionality stores $\Delta$. Let $\Delta_B = g^{-1}(\Delta)$.

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

1. If $P_A$ is corrupt then receive $t \in \mathbb{F}$ and $x \in \mathbb{F}^k$ from $A$, and compute $q$ such that $q + t = \langle x, \Delta_B \rangle$.

2. Otherwise, sample $t \stackrel{\$}{\leftarrow} \mathbb{F}$ and compute $q$ such that $q + t = x \cdot \Delta$.

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

**Functionality $\mathcal{F}_{\text{JK}}$**

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

**Input:** On receiving $(\text{Input}, id_1, \ldots, id_m, x_1, \ldots, x_m, P_j)$ from party $P_j$ and $(\text{Input}, id_1, \ldots, id_m, P_j)$ from all other parties, where $x \in \mathbb{F}$, set $\text{Val}[id_i] \leftarrow x_i$ for $i = 1, \ldots, m$.

**Linear comb.:** On receiving $(\text{LinComb}, \text{id}, id_1, \ldots, id_t, c_1, \ldots, c_t, c)$ from all parties, where $(id_1, \ldots, id_t) \subseteq \text{Val.keys}()$ and the combination coefficients $c_1, \ldots, c_t, c \in \mathbb{F}$, set $\text{Val}[\text{id}] \leftarrow \sum_{i=1}^t \text{Val}[id_i] \cdot c_i + c$.

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

**Check:** On receiving $(\text{Check}, id_1, \ldots, id_t, x_1, \ldots, x_t)$ from every party $P_i$, wait for an input from the adversary. If it inputs OK, and $\text{Val}[id_j] = x_j$ for all $j$, return OK to all parties, otherwise return $\perp$ and terminate.

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

**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.
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 $\mathcal{F}_{\text{Comm}}$ to commit to this and receive the handle $\tau_i$.
2: Call $\mathcal{F}_{\text{Comm}}$ with $(\text{Open}, \tau_i)$ to open the commitments.
3: If $\sigma^{(1)} + \cdots + \sigma^{(n)} \neq 0$, output $\bot$ and abort, otherwise continue.

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, $\mathcal{F}_J$ (Fig. 3), which is independent of the details of the MAC scheme used and the underlying MAC keys; this leads to a 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 $\text{Input}$ 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 values that have been input using the $\text{LinComb}$ command.

The $\text{Open}$ command lets the adversary output inconsistent or incorrect values. However, if this happened to honest parties, the $\text{Check}$ 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 $\mathcal{F}_{\text{COPE}_e}$ between every pair of parties $(P_i, P_j)$: $P_j$ inputs $\Delta^{(j)}$ as their correlation into $\mathcal{F}_{\text{COPE}_e}\text{.Initialize}$, and then they call $\mathcal{F}_{\text{COPE}_e}\text{.Extend}$ where $P_i$ inputs $x^{(i)}$. Let $t^{(i,j)}$ be the $\mathcal{F}_{\text{COPE}_e}$ 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

$$m^{(i)} = x^{(i)} \cdot \Delta^{(i)} + \sum_{j \neq i} \left( q^{(i,j)} + t^{(i,j)} \right).$$
Protocol 3 $\Pi_{\ell,1}$, creating $[.]$ 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 $\mathcal{F}_{\text{COPE}e}$-Initialize($\mathbb{F}$) where $P_j$ inputs $\Delta^{(j)}$.

Input: On input $(\text{Input}, id_1, \ldots, id_m, x_1, \ldots, x_m, P_j)$ from $P_j$ and $(\text{Input}, id_1, \ldots, id_m, P_j)$ from all other parties:

1. $P_j$ samples $x_0 \overset{\$}{\leftarrow} \mathbb{F}$.
2. For $h = 0, \ldots, m$, $P_j$ generates a random additive sharing $\sum_i x^{(i)}_h = x_h$ and sends $x^{(i)}_h$ to $P_i$.
3. For every $i \neq j$, $P_i$ and $P_j$ call $\mathcal{F}_{\text{COPE}e}$-Extend, where $P_j$ inputs $(x_0, \ldots, x_m) \in \mathbb{F}^m$.
4. $P_i$ receives $q^{(i,j)}_{h}$ and $P_j$ receives $t^{(j,i)}_{h}$ such that

\[ q^{(i,j)}_{h} + t^{(j,i)}_{h} = x_h \cdot \Delta^{(i)}, \text{ for } h = 0, \ldots, m. \]

5. Each $P_i$, $i \neq j$, computes the MAC shares $m^{(i)}_h = \sum_{j \neq i} q^{(i,j)}_{h}$, and $P_j$ computes the MAC shares

\[ m^{(j)}_h = x_h \cdot \Delta^{(j)} + \sum_{j \neq i} t^{(j,i)}_{h} \]

to obtain $[x_h]$, for $h = 0, \ldots, m$.
6. The parties sample $r \leftarrow \mathcal{F}_{\text{Rand}}([\mathbb{F}^{m+1}])$.
7. $P_j$ computes and broadcasts $y = \sum_{h=0}^{m} r_h \cdot x_h$.
8. Each party $P_i$ computes $m^{(i)} = \sum_{h=0}^{m} r_h \cdot m^{(i)}_h$.
9. The parties execute $\Pi_{\text{MACCheck}}$ with $y$ and $\{m^{(i)}\}_{i \in [n]}$.
10. All parties store their shares and MAC shares under the handles $id_1, \ldots, id_m$.

Linear comb.: On input $(\text{LinComb}, \overline{id}, id_1, \ldots, id_t, c, c_t, c)$, the parties retrieve their shares and MAC shares $\{x^{(i)}_j, m^{(i)}(x^{(i)}_j)\}_{j \in [t], i \in [n]}$ corresponding to $id_1, \ldots, id_t$, and each $P_i$ computes:

\[ y^{(i)} = \sum_{j=1}^{t} c_j \cdot x^{(i)}_j + \begin{cases} c & i = 0 \\ 0 & i \neq 0 \end{cases} \]
\[ m(y^{(i)}) = \sum_{j=1}^{t} c_j \cdot m^{(i)}(x^{(i)}_j) + c \cdot \Delta^{(i)}, \]

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

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:

\[ m = \sum_{i=1}^{n} m^{(i)} = \sum_{i=1}^{n} x^{(i)} \cdot \Delta^{(i)} + \sum_{i=1}^{n} \sum_{j \neq i} (q^{(i,j)} + t^{(i,j)}) \]
Protocol $\Pi_{[\|]}$ (continued)

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

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

1. Each $P_i$ retrieves and broadcasts their share $x^{(i)}$.
2. Parties reconstruct $x = \sum_{i=1}^{n} x^{(i)}$ and output it.

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

1. Sample a public, random vector $r \leftarrow \mathcal{F}_{\text{Rand}}(\mathbb{F})$.
2. Compute $y \leftarrow \sum_{j=1}^{t} r_j \cdot x_j$ and $m(y)^{(i)} \leftarrow \sum_{j=1}^{t} r_j \cdot m_{\text{id}_j}^{(i)}$, where $m_{\text{id}_j}^{(i)}$ denotes $P_i$’s MAC share stored under $\text{id}_j$ for all $i \in [n]$ and $j \in [t]$.
3. Execute $\Pi_{\text{MACCheck}}$ with $y$ and $m(y)^{(i)}$.

\[
\begin{align*}
\sum_{i \not\in A} m^{(i)} &= \sum_{i \not\in A} x^{(i)} \cdot \Delta^{(i)} + \sum_{i \not\in A} \sum_{j \not\in A, i \neq j} \left(q^{(i,j)} + t^{(i,j)}\right) \\
&= \sum_{i \not\in A} x^{(i)} \cdot \Delta^{(i)} + \sum_{j \in A} x^{(i)} \cdot \Delta^{(j)} \\
&\quad + \sum_{i \not\in A} \sum_{j \in A} \left( x^{(i)} - t^{(j,i)} \right) + x^{(i)} \cdot \Delta^{(j,i)} - q^{(j,i)} \\
&= \sum_{i \not\in A} x^{(i)} \cdot \Delta^{(i)} + \sum_{j \not\in A, i \neq j} x^{(i)} \cdot \Delta^{(j)} \\
&\quad + \sum_{i \not\in A} \sum_{j \in A} \left( x^{(i)} - t^{(j,i)} \right) + x^{(i)} \cdot \Delta^{(j,i)} - q^{(j,i)} \\
&= \sum_{i \not\in A} \left( x^{(i)} + \sum_{j \not\in A} x^{(j)} + g \cdot \sum_{j \in A} x^{(i,j)} \cdot \Delta^{(i)} \right)
\end{align*}
\]

**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 $\mathcal{F}_{\text{COPEe}}$\textunderscore\text{Initialize} step with party $P_i$ and they could use non-monochrome input shares $x^{(i,j)}$ in the $\mathcal{F}_{\text{COPEe}}$\textunderscore\text{Extend} calls, and these may also differ between each instance of $\mathcal{F}_{\text{COPEe}}$. We will show that using inconsistent inputs in different $\mathcal{F}_{\text{COPEe}}$ 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 $\mathcal{F}_{[\|]}$. The sum of the honest parties’ MACs is given by

\[
\begin{align*}
\sum_{i \not\in A} m^{(i)} &= \sum_{i \not\in A} x^{(i)} \cdot \Delta^{(i)} + \sum_{i \not\in A} \sum_{j \not\in A, i \neq j} \left(q^{(i,j)} + t^{(i,j)}\right) \\
&= \sum_{i \not\in A} x^{(i)} \cdot \Delta^{(i)} + \sum_{j \in A} x^{(i)} \cdot \Delta^{(j)} \\
&\quad + \sum_{i \not\in A} \sum_{j \in A} \left( x^{(i)} - t^{(j,i)} \right) + x^{(i)} \cdot \Delta^{(j,i)} - q^{(j,i)} \\
&= \sum_{i \not\in A} x^{(i)} \cdot \Delta^{(i)} + \sum_{j \not\in A, i \neq j} x^{(i)} \cdot \Delta^{(j)} \\
&\quad + \sum_{i \not\in A} \sum_{j \in A} \left( x^{(i)} - t^{(j,i)} \right) + x^{(i)} \cdot \Delta^{(j,i)} - q^{(j,i)} \\
&= \sum_{i \not\in A} \left( x^{(i)} + \sum_{j \not\in A} x^{(j)} + g \cdot \sum_{j \in A} x^{(i,j)} \cdot \Delta^{(i)} \right)
\end{align*}
\]
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,

\[
\sum_{i \not\in A} m^{(i)} - x \cdot \Delta^{(i)} = \sum_{i \not\in A} \left( \left\langle g \cdot \sum_{j \not\in A} x^{(j)} + g \cdot \sum_{j \in A} x^{(i,j)} - g \cdot x, \Delta_B^{(i)} \right\rangle + R_i \right) = \sum_{i \not\in A} \left( \left\langle g \cdot \left( \sum_{j \not\in A} x^{(j)} - x \right) + g \cdot \sum_{j \in A} x^{(i,j)} \cdot \Delta_B^{(i)} \right\rangle + R_i \right).
\]

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 \( \{R_i\}_{i \not\in A} \) are independent of \( \{\Delta_B^{(i)}\}_{i \not\in A} \) and known to the adversary because \( \{x^{(i)}\}_{i \not\in A} \) were revealed in the opening. The adversary therefore has to “guess” the inner product for random \( \{\Delta_B^{(i)}\}_{i \not\in A} \). Clearly, success is guaranteed if \( x - \sum_{j \not\in A} x^{(j)} = \sum_{j \in A} x^{(i,j)} \) for all \( i \not\in A \). 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 all \( \sum_i \Delta^{(i)} \), 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 \( \lceil x \rceil \), 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 \( r \in \mathbb{F}^m \) 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, \( \sigma^{(i)} \leftarrow m^{(i)} - y \cdot \Delta^{(i)} \). 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 \( \Pi_{\text{CP}} \) securely implements \( \mathcal{F}_{\Pi} \) in the \( (\mathcal{F}_{\text{COPEe}}, \mathcal{F}_{\text{Comm}}, \mathcal{F}_{\text{Rand}}) \)-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 \( x_{h} \) input when authenticating the \( h \)-th value, and denote by \( \{r_h\}_{h \in [m]} \) the random coefficients generated.
using $F_{\text{Rand}}$. For $g \neq g' \in [k]$, if $x_{h,g} \neq x_{h,g'}$, there is a $1/|F|$ chance that $\sum_{h} r_{h} x_{h,g} = \sum_{h} r_{h} x_{h,g'}$. 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 $\log |F| (\log |F| - 1)/2$ such pairs $g \neq g'$ explains the $2 \log \log |F|$ subtrahend in the theorem. It is easy to see that a repeated check would suffice for security parameter $\log |F|$.

## 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 $F_4$ and OT.

Recall that a multiplication triple is a tuple of shared values $([a], [b], [c])$ where $a, b \overset{\$}{\leftarrow} F$ and $c = a \cdot b$. Given $F_4$ 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 $F_4$ and an ideal OT functionality to securely implement the functionality $F_{\text{Triple}}$, which has all of the same features as $F_{\Pi_4}$, with the following additional command:

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

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 $F_4$. We start by showing correctness of the protocol $\Pi_{\text{Triple}}$ (Protocol 4) for semi-honest adversaries.

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

\[ c^{(i,j)} + e^{(i,j)} = t^{(i,j)} + q^{(j)} = a^{(i)} \cdot b^{(j)} \in F^\tau. \]

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

\[ c = \sum_{i \in [n]} c^{(i)} = \sum_{i \in [n]} a^{(i)} \cdot b^{(i)} + \sum_{i \in [n]} c^{(i)}_{i,j} + c^{(j)}_{i,j} = \sum_{i \in [n]} a^{(i)} \cdot b^{(i)} + \sum_{i \in [n]} a^{(i)} \cdot c^{(j)}_{i,j} = \sum_{i \in [n]} a^{(i)} \cdot \sum_{i \in [n]} b^{(i)} = (\sum_{i \in [n]} a^{(i)}) \cdot b = a \cdot b. \]

The parties then locally Combine together the $\tau$ components of $a^{(i)} = (a_1^{(i)}, \ldots, a_{\tau k}^{(i)})$ and $c^{(i)} = (c_1^{(i)}, \ldots, c_{\tau k}^{(i)})$ using two random values $r$ and $\hat{r}$ in $F^\tau$ obtained from $F_{\text{Rand}}$. The outputs of this step
Protocol 4 Triple generation protocol, $Π_{\text{Triple}}$

The integer parameter $τ ≥ 3$ specifies the amount of random combining to perform.

**Multiply:**

1. Each party samples $a^{(i)} \overset{\$}{\leftarrow} \mathbb{F}^{τ}, b^{(i)} \overset{\$}{\leftarrow} \mathbb{F}$.
2. Every ordered pair of parties $(P_i, P_j)$ does the following:
   1. Both parties call $\mathcal{F}_{\text{ROT}}^{τ,k}$ where $P_i$ inputs $(a^{(i)}_1, \ldots, a^{(i)}_k) = g^{-1}(a^{(i)}) \in \mathbb{F}_2^{τ}.$
   2. $P_j$ receives $q^{(j,i)}_0, q^{(j,i)}_1 \in \mathbb{F}$ and $P_i$ receives $s^{(i,j)}_h = q^{(i,j)}_h a^{(i)}_h$, for $h = 1, \ldots, τk$.
   3. $P_j$ sends $a^{(j,i)}_h = q^{(j,i)}_0 - q^{(j,i)}_1 - b^{(j)}, h \in [τk]$.
   4. $P_i$ sets $t^{(i,j)}_h = s^{(i,j)}_h + a^{(i)}_h, d^{(i,j)}_h = q^{(i,j)}_0 + a^{(i)}_h \cdot b^{(j)}$, for $h = 1, \ldots, τk$. Set $q^{(j,i)}_h = q^{(j,i)}_0$.
   5. Split $(t^{(i,j)}_1, \ldots, t^{(i,j)}_{τk})$ and $(q^{(j,i)}_1, \ldots, q^{(j,i)}_{τk})$ into $τ$ vectors of $k$ components each, $(t_1, \ldots, t_τ)$ and $(q_1, \ldots, q_τ)$.
   6. $P_i$ sets $c^{(i)}_{i,j} = (\langle \mathbf{g}, t_1 \rangle, \ldots, \langle \mathbf{g}, t_τ \rangle) \in \mathbb{F}^τ$.
   7. $P_j$ sets $c^{(j)}_{i,j} = - (\langle \mathbf{g}, q_1 \rangle, \ldots, \langle \mathbf{g}, q_τ \rangle) \in \mathbb{F}^τ$.
   8. Now we have $c^{(i)}_{i,j} + c^{(j)}_{i,j} = a^{(i)} \cdot b^{(j)} \in \mathbb{F}^τ$
3. Each party $P_i$ computes:
   $$c^{(i)} = a^{(i)} \cdot b^{(i)} + \sum_{j \neq i} (c^{(i)}_{i,j} + c^{(j)}_{i,j})$$

**Combine:**

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

**Authenticate:** Each party $P_i$ runs $\mathcal{F}_{\text{[3]}}$.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 $s \leftarrow \mathcal{F}_{\text{Rand}}(\mathbb{F})$.
2. Call $\mathcal{F}_{\text{[3]}}$.Open on input $s \cdot [a] - [\hat{a}]$ to obtain $ρ$.
3. Call $\mathcal{F}_{\text{[3]}}$.LinComb to store $s \cdot [c] - [\hat{c}] - [b] \cdot ρ$ under $[σ]$.
4. Run $\mathcal{F}_{\text{[3]}}$.Check($[ρ], [σ], ρ, 0$) and abort if $\mathcal{F}_{\text{[3]}}$ aborts.

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

In this scheme, $a, c$ 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 $\mathcal{F}_{\text{[3]}}$ to Authenticate their shares of $a, \hat{a}, b, c$ and $\hat{c}$.

Finally, correctness of the triples $[a], [\hat{a}], [c]$ 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 $a^{(j,i)} \in \mathbb{F}^r$ and $b^{(j,i)} \in \mathbb{F}^r$ be the actual values used by $P_j$ in the two executions of steps 1 and 3 with an honest $P_i$, instead of $a^{(j)}$ and $b^{(j)}$. Define the values $a^{(j)}$ and $b^{(j)}$ to be those values used in the instance with an arbitrary (e.g. lowest index) honest party $P_0$.

Then, for each $i \in A$, let $\delta_a^{(j,i)} = a^{(j,i)} - a^{(j)} \in \mathbb{F}^r$ and $\delta_b^{(j,i)} = b^{(j,i)} - (b^{(j)}, \ldots, b^{(j)}) \in \mathbb{F}^r$ be the deviation in $P_j$’s input with an honest $P_i$. Let $\delta_a^{(i)} = \sum_{j \in A} \delta_a^{(j,i)}$ and $\delta_b^{(i)} = \sum_{j \in A} \delta_b^{(j,i)}$, and consider $\delta_b^{(i)}$ as a length $\tau$ vector with components in $\mathbb{F}^k$ (similarly to $t_h, q_h$ in the protocol).

Now by analyzing the possible adversarial deviations and summing up shares, we can see that the $h$-th component of $c$ (for $h \in [\tau]$), at the end of the Multiply stage, is

$$c[h] = a[h] \cdot b + \sum_{i \in A} \langle (a^{(i)}[h])_B, \delta_a^{(i)}[h] \rangle + \sum_{i \in A} b^{(i)} \cdot \delta_a^{(i)}[h].$$ (2)

Intuitively, it is easy to see that any non-zero $\delta_a^{(i)}$ 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 $\delta_b^{(i)}$ errors can be chosen so that $e_{ah}$ only depends on single bits of the shares $a^{(i)}$. This means that a corrupt party can attempt to guess a few bits (or linear combinations of bits) of $a^{(i)}$. 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 $a^{(i)}$ shares, rather than just one field element. The Combine step then randomly reduces $a^{(i)}$ down to two field elements. The intuition here is that, to be able to guess a single bit of the final shares $\hat{a}^{(i)}, \hat{a}^{(i)}$, 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 $(a, b, c)$ 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 $\Pi_{\text{Triple}}$ (Protocol 4) securely implements $\mathcal{F}_{\text{Triple}}$ in the $(\mathcal{F}_{\text{ROT}}, \mathcal{F}_1)$-hybrid model, with statistical security parameter $k$, for $\tau = 4$. 
Protocol 5 Preprocessing input tuples, Π_{InputTuple}

Input: On input \((\text{Input}, P_j)\) from all parties, do the following:

1: \(P_j\) samples \(r \xleftarrow{\$} F\), and calls \(F_{\Pi}^{J \cdot K}\) with \((\text{Input}, r, P_j)\).
2: All parties output \([r]\) and \(P_j\) outputs \(r\).

Functionality \(F_{\text{Prep}}\)

\(F_{\text{Prep}}\) has all of the same features as \(F_{\Pi}^{J \cdot K}\), with the following additional commands:

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

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

Figure 4: Ideal functionality for the SPDZ preprocessing phase.

**Corollary 1.** For \(k \geq 128\) and \(\tau = 3\), \(\Pi_{\text{Triple}}\) securely implements \(F_{\text{Triple}}\) 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 \(F_{\Pi}^{J \cdot K}\). In the online phase, they broadcast the difference of this and their actual input, so that the shared random value can then be adjusted to the correct value by all parties. Note that this method avoids having to use the \textbf{Input} command of \(F_{\text{Prep}}\) (and hence of \(F_{\Pi}^{J \cdot K}\) 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 \(F_{\text{Prep}}\) (Fig. 4), which also contains all features from \(F_{\Pi}^{J \cdot K}\) (like \(F_{\text{Triple}}\)). Given this and the proof of Theorem 2, it is straightforward to show that the triple generation and input tuple generation protocols securely implement \(F_{\text{Prep}}\).

**Theorem 3.** The protocols \(\Pi_{\text{Triple}}\) and \(\Pi_{\text{InputTuple}}\) together securely realize the functionality \(F_{\text{Prep}}\), in the \((F_{\Pi}^{J \cdot K}, F_{\text{ROT}}, F_{\text{Rand}})\)-hybrid model.

6.2 Online phase

Given the preprocessing data from \(F_{\text{Prep}}\), 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 \(F_{\Pi}^{J \cdot K}\).
Theorem 4. The protocol $Π_{\text{Online}}$ (Protocol 6) securely implements the functionality $F_{\text{ABB}}$ (Fig. 5) against a static, active adversary corrupting up to $n - 1$ parties in the $F_{\text{Prep}}$-hybrid model.
Simulator $S_{\text{Online}}$

**Initialize:** The simulation of this procedure is performed by running a local copy of $F_{\text{prep}}$. It is straightforward because the InputTuple and Triple commands of $F_{\text{prep}}$ 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 $\epsilon'$, compute $x'_i \leftarrow r_i + \epsilon'$ and use $x'_i$ as input to $F_{\text{ABB}}$.

**Add:** This local procedure requires no simulation.

**Multiply:** Send random values for $\epsilon$ and $\rho$ to the adversary and wait for it to input $\epsilon'$ and $\rho'$. 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 $F_{\text{prep}, \text{Check}}$.
2. Receive the output $y$ from $F_{\text{ABB}}$ and forward it to the adversary in $F_{\text{prep}}$. If it returns $y' \neq y$, input Abort to $F_{\text{ABB}}$ and abort in the simulation of the second Check call, otherwise input Deliver to $F_{\text{ABB}}$ and continue.

![Figure 6: Simulator for the online phase.](image)

**Proof.** Because most of the protocol purely consists of interaction with $F_{\text{prep}}$ 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.

\[\square\]

## 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 $\Pi_{\|\cdot\|}$ is the $n - 1$ calls to $\Pi_{\text{COPEe}}$, 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 $\tau k$ calls to $\mathcal{F}_{\text{ROT}}$, followed by sending a further $\tau k^2$ bits in step 3 and then 5 calls to $\Pi_{\text{COPEe}}$ for authentication (ignoring $\mathcal{F}_{\text{Rand}}$ and sending the input shares as these are negligible). Since each call to $\mathcal{F}_{\text{ROT}}$ requires communicating $\lambda$ bits, and $\Pi_{\text{COPEe}}$ 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 $\tau$ 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 $\lambda$. 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 $\Pi_{\text{COPEe}}$ 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.

<table>
<thead>
<tr>
<th>Field bit length</th>
<th>Input cost (kbit)</th>
<th>Triple cost (kbit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>4.16(n - 1)</td>
<td>53.25n(n - 1)</td>
</tr>
<tr>
<td>128</td>
<td>16.51(n - 1)</td>
<td>180.22n(n - 1)</td>
</tr>
<tr>
<td>256</td>
<td>65.79(n - 1)</td>
<td>622.59n(n - 1)</td>
</tr>
<tr>
<td>512</td>
<td>262.66(n - 1)</td>
<td>2293.76n(n - 1)</td>
</tr>
</tbody>
</table>

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 $\lambda$, 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 \( 16 \times 8 \)-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 \( \Pi_{\text{COPE}e} \) and \( \Pi_{\text{Triple}} \) involve the computation of \( \langle g, x \rangle \) for \( x \in \mathbb{F}_{\log |\mathbb{F}|} \). 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 \texttt{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 second 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. This clearly illustrates the 200-fold performance improvement of our protocol, compared with (actively secure) SPDZ. Now the preprocessing phase is within 2–3 orders of magnitude of the online phase.

![Figure 9: Vickrey auction run by two parties.](image)

**References**


[7] Bogdanov, D., Jõemets, M., Shim, S., and Vaht, M. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In


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 $\Pi_{J,K}$ securely implements $F_{J,K}$ in the $(F_{COPEe}, F_{Comm}, F_{Rand})$-hybrid model, with statistical security parameter $\log |F| - 2 \log \log |F|$. 

Proof. We describe a simulator, $S$, that interacts with $F_{J,K}$, such that no environment $Z$ can distinguish between an interaction with $S$ and $F_{J,K}$ and an interaction with the real adversary $A$ and $\Pi_{J,K}$. After describing the simulator we then argue indistinguishability of the real and ideal worlds.

$S$ 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 $F_{COPEe}$ 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 $\Delta^{(i)} \overset{\$}{\leftarrow} F$ for all $i \notin A$.

Simulating the Input phase.

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

$$\sum_{i \notin A} \left\langle g \cdot y - g \ast \sum_{h=0}^{m} r_h \cdot \sum_{j \in A} x^{(i,j)}_h, \Delta^{(i)}_B \right\rangle = 0 \quad (3)$$
for \( \{\Delta_B^{(i)}\}_{i \notin A} \) in \( \mathbb{F} \). The solutions represent the set of all possible MAC keys of honest parties that let the MAC check pass, not just the concrete MAC keys chosen by the simulator. If \( \sum_{i \notin A} \Delta_B^{(i)} = 0 \) for all solutions, abort. Otherwise, for some \( \{\Delta_B^{(i)}\}_{i \notin A} \) such that \( \sum_{i \notin A} \Delta_B^{(i)} \neq 0 \) and for each \( h \in [m] \), compute
\[
x_h = \left( \sum_{i \notin A} \langle g, \Delta_B^{(i)} \rangle \right)^{-1} \cdot \sum_{i \notin A} \sum_{j \in A} \langle g \cdot x_{h}^{(i,j)}, \Delta_B^{(i)} \rangle,
\]
and input \( \{x_h\}_{h \in [m]} \) to \( \mathcal{F}_{[1]} \) on behalf of \( P_j \). We will show that, with overwhelming probability, \( x_h \) is unique over all possible \( \Delta_B^{(i)} \).

- Otherwise, emulate communication channels and instances of \( \mathcal{F}_{\text{COPE}e} \) with corrupted parties \( P_i, i \in A \). For the channel with \( P_i \), send a random share \( x^{(i)} \), and for the \( \mathcal{F}_{\text{COPE}e} \) instance, receive \( q^{(i,j)} \) from \( \mathcal{A} \), for \( h = 0, \ldots, m \). Store \( \sum_{i \notin A} x_{h}^{(i)} \) and \( \sum_{i \notin A} m_{h}^{(i)} = \sum_{i \notin A} x^{(i)} \cdot \Delta^{(i)} + q^{(i,j)} \)
in CS. Sample \( r \stackrel{\$}{\leftarrow} \mathbb{F}^t \) to emulate \( \mathcal{F}_{\text{Rand}} \), sample \( y \stackrel{\$}{\leftarrow} \mathbb{F} \), and send both to the corrupted parties. Compute \( \sum_{i \notin A} \sigma^{(i)} = \sum_{i \notin A} r_h \cdot \sum_{i \notin A} m_{h}^{(i)} - y \cdot \Delta^{(i)} \). 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 \( \mathcal{F}_{\text{Comm}} \) just as in the Check phase below.

**Simulating the Linear comb. phase.** Store the linear combination in LC under \( \overline{\sigma} \).

**Simulating the Open phase.** Receive the constraints on \( x \) from \( \mathcal{F}_{[\|]} \) 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 \( \sum_{i \notin A} x^{(i)} \) and \( \sum_{i \notin A} m^{(i)} \) 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 \( \mathcal{F}_{[1]} \).

**Simulating the Check phase.** Sample \( r \stackrel{\$}{\leftarrow} \mathbb{F}^t \) to emulate \( \mathcal{F}_{\text{Rand}} \) and send it to the corrupted parties. Emulating \( \mathcal{F}_{\text{Comm}} \), receive \( \sigma_i \) 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 \( m'_{j} \) 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 \( \mathcal{F}_{\text{Comm}} \). If \( \sum_{i} \sigma_i = 0 \), input OK to \( \mathcal{F}_{[1]} \) and \( \perp \) otherwise.

**Indistinguishability.** Now we argue indistinguishability. The LinComb command does not require communication, and the Initialize command only involve sending random shares and using \( \mathcal{F}_{\text{COPE}e} \), 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. \( \mathcal{F}_{\text{COPE}e} \) only outputs random shares, \( y \) has \( r_0 \cdot x_0 \) as summand for random single-use \( x_0 \), and \( \Pi_{\text{MACCheck}} \) 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 \( |F|^{n-|A|-1} \) 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 \( 2^{-\log |F|} \) because there are \( |F|^{n-|A|} \) 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, \( J_1 \) 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 \( \sigma_i \) 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 \( x_i \). 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 \( x_i \).

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

\[
- \sum_{i \in A} \sigma^{(i)} = \sum_{i \notin A} \sigma^{(i)} = \sum_{i \notin A} m^{(i)} - y \cdot \Delta^{(i)} = \sum_{i \notin A} \left( \left\langle g \cdot y + g \ast \sum_{j \in A} \sum_{h=0}^{m} r_h \cdot x^{(i,j)}_h, \Delta^{(i)}_B \right\rangle + R_i \right),
\]

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 \( \{\tilde{\Delta}^{(i)}\}_{i \notin A} \), we get (3) for \( \tilde{\Delta}^{(i)} = \Delta^{(i)} - \Delta^{(i)} \) for all \( i \notin A \). This proves that the set \( S_\Delta \) 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

\[
\left\langle g \cdot x_h - g \ast \sum_{j \in A} x^{(i,j)}_h, \tilde{\Delta}^{(i)}_B \right\rangle = 0
\]

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 \( S_\Delta \). 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

\[
\left\langle g \cdot x_{f,h} - g \ast \sum_{j \in A} x^{(i,j)}_h, \tilde{\Delta}^{(i)}_B \right\rangle = 0
\]

for all \( \{\tilde{\Delta}^{(i)}_f\}_{i \notin A} \in \tilde{S}_{f} \subset \tilde{S}_\Delta \) such that \( |\tilde{S}_{f}| > 2^{(n-|A|-1)\log |F|} \). The latter condition is required for the adversary to be successful with probability more than \( 2^{-\log |F|} \) 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 \( |\tilde{S}_\Delta| \leq 2^{(n-|A|)\log |F|} \) by definition, \( l \leq \log |F| \).

28
Let \( f \neq f' \in [l] \). Then,

\[
\sum_{i \notin A} \left( \langle g \cdot y - g \cdot \sum_{j=0}^{m} r_{h} \cdot \sum_{j \in A} x_{h}^{(ij)}, \tilde{\Delta}_{B}^{(i)} \rangle \right) = 0
\]

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

\[
\sum_{h=0}^{m} r_{h} \cdot \sum_{i \in A} \left( \langle g \cdot \sum_{j \in A} x_{h}^{(ij)}, \tilde{\Delta}_{f,B}^{(i)} - \tilde{\Delta}_{f',B}^{(i)} \rangle \right) = 0
\]

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

\[
\sum_{h=0}^{m} r_{h} \cdot \left( x_{f,h} \cdot \sum_{i \in A} \tilde{\Delta}_{f}^{(i)} - x_{f',h} \cdot \sum_{i \in A} \tilde{\Delta}_{f'}^{(i)} \right) = 0. \tag{8}
\]

By definition, there exists \( \tilde{h} \in \{0, \ldots, m\} \) 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 \( \{\tilde{\Delta}^{(i)}\}_{i \notin A} \in \tilde{S}_{f} \) and \( \{\tilde{\Delta}^{(i)}\}_{i \notin A} \in \tilde{S}_{f'} \). It follows that

\[
\left( x_{f,h} \cdot \sum_{i \in A} \tilde{\Delta}_{f}^{(i)} - x_{f',h} \cdot \sum_{i \in A} \tilde{\Delta}_{f'}^{(i)} \right) \neq 0
\]

for some \( \{\tilde{\Delta}_{f}^{(i)}\}_{i \in A} \in \tilde{S}_{f} \) 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 \( 2^{-\log |F|} \) over the choice of \( \{r_{h}\}_{h=0}^{m} \). Given that there are less than \( (\log |F|)^2 \) pairs \( f \neq f' \in [l] \), the overall probability is at most \( (\log |F|)^2 \cdot 2^{-\log |F|} = 2^{-\log |F|+2\log \log |F|} \).

We have established that, for every \( h \in [m] \), there exists a unique \( x_{h} \) where the adversary can compute

\[
\sum_{i \notin A} \left( \langle g \cdot x_{h} - g \cdot \sum_{j \in A} x_{h}^{(ij)}, \Delta_{B}^{(i)} \rangle \right).
\]

(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

\[
\langle g \cdot x_{h} - g \cdot x'_{h}, \Delta_{B}^{(i)} \rangle = (x_{h} - x'_{h}) \cdot \sum_{i \notin A} \Delta^{(i)}.
\]

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

We now turn to the discussion of \( F = F_{p} \). The main difference to the case of \( F = F_{pk} \) is that there is no bijection between \( \Delta \in F = F_{p} \) and \( \Delta \in F_{p}^{\log p} \). While there are canonical maps both ways, bit decomposition from \( F_{p} \) to \( F_{p}^{\log p} \) and \( \langle g, \Delta \rangle \) from \( F_{p}^{\log p} \) to \( F_{p} \), the former is not surjective and the latter is not injective. This implies that the solutions of (5) or (6) are not necessarily vectors of bits rather than elements of \( F_{p} \). Nevertheless, the lemma below proves that, if \( \tilde{S}_{f} \) contains at least \( 2^{(n-|A|-1)\log |F|} \) vectors consisting only of bits (which is necessary for an adversary to pass the MAC check), then it has dimension at least \( (n - |A| - 1) \log |F| \) for all \( f \). Together with the fact that \( \tilde{S}_{\Delta} \) has dimension at most \( (n - |A|) \log |F| \) by definition and \( \tilde{S}_{f} \cap \tilde{S}_{f'} = \emptyset \) for \( f \neq f' \in [l] \), it follows that \( l \leq \log |F| \) as above. \( \square \)
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 $v_1, \ldots, v_l$ such that $v_{i+1}$ starts with more zeroes than $v_i$ for all $i = 1, \ldots, l-1$. Such a basis can be constructed from any basis using Gaussian elimination. Every element of $V$ has the form $\sum_{i=1}^l a_i v_i$. 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 $v_i$ is not zero but the $g$-th element of $v_{i+1}, \ldots, v_l$ is. This exists by definition. Then, there exist only two choices of $a_i$ in order to let the $g$-th element of $\sum_{i=1}^l a_i v_i$ be in $\{0, 1\}$. Iterating over $a_1, \ldots, a_l$, this proves that the size of $V \cap \{0, 1\}^k$ is at most $2^l$. \qed

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

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

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 $h_t : \{0,1\}^n \rightarrow \{0,1\}^k$. 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 $\mathcal{H} = \{h_t\}_{t \in T}$, $h_t : S \rightarrow \mathbb{F}$, 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 $t \overset{\$}{\leftarrow} T$ (independent of $X$), we have

$$(h_t(X), U_t) \approx (U_S, U_t)$$

for statistical security parameter $\kappa$.

B.2 Proof of Theorem 2

Theorem 6 (Theorem 2, restated). The protocol $\Pi_{\text{Triple}}$ (Protocol 4) securely implements $\mathcal{F}_{\text{Triple}}$ in the $(\mathcal{F}_{\text{ROT}}, \mathcal{F}_{\text{[1]}})$-hybrid model, with statistical security parameter $\log_2 |\mathbb{F}|$, for $\tau = 4$.

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

Simulating the Multiply phase. The simulator emulates $\mathcal{F}_{\text{ROT}}^{k,k}$ and sends $d_{i,j}^{(j,i)}$, $q_i^{(j,i)}, j \in A$ to $A$. Then for each $j \in A$, $S$ receives $d_{i,j}^{(j,i)}$ by $A$, for each $j \neq i$, sets $b_{i,j}^{(j,i)} = d_{i,j}^{(j,i)} - q_i^{(j,i)} + q_{j,h}^{(j,i)}$, $h \in [\tau \cdot k]$, and sends random $d_{i,j}^{(j,i)}$, $i \notin A$ to $A$.

If $P_j$, $j \in A$, gives any inconsistent $b_{i,j}^{(j,i)}$, then $S$ computes $\delta_b[h]^{(j,i)}$ and $\sum_{j \in A} \delta_b[h]^{(j,i)} = \delta_b[h]^{(j,i)}$. If some $P_j$ inputs inconsistent values $a^{(j)}$, when playing with $P_i$, $i \notin A$, then $S$ computes $\delta_a[h]^{(j,i)}$ and $\sum_{j \in A} \delta_a[h]^{(j,i)} = \delta_a[h]^{(j,i)}$.

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

Simulating the Authentication phase. Now $S$ emulates $\mathcal{F}_{\text{[1]}}$ with inputs from the corrupt parties provided by $A$. So if some inputs are inconsistent with previous computation, $S$ computes $e_{\text{Auth}}$, $\hat{e}_{\text{Auth}}$, i.e. the deviation introduced by $A$ in this step. Note that here $e_{\text{Auth}}, $\hat{e}_{\text{Auth}} \neq 0$ essentially means that the adversary authenticates values different from those computed in the previous phases. If $\mathcal{F}_{\text{[1]}}$ aborts, then $S$ sends $\text{Abort}$ to $\mathcal{F}_{\text{Prep}}$.

Simulating the Sacrifice step. The simulator emulates the functionalities $\mathcal{F}_{\text{Rand}}$ and $\mathcal{F}_{\text{[1]}}$. Open honestly. Emulating $\mathcal{F}_{\text{[1]}}$.Check, $S$ 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, $Z$ can see the mask $d^{(i,j)}_h$, for each $i \notin A$, but they look perfectly random as the values $q^{(i,j)}_1$ are uniformly random and never revealed to $Z$.

Then the Amplify/Combine command do not require communication and in the Authenticate command the simulator honestly runs $F_J \cdot K$, so the view of $Z$ up to the point where the values $\rho$ and $\sigma$ are opened in the Sacrifice step, has exactly the same distribution in both the execution. In the openings, we need to prove that the values produced in the real world and the simulated random values are indistinguishable.

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

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

In this way, if (9) is satisfied, it is easy to see that the opened value $\rho$ 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 $B = P \setminus A$. 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 $\sigma$ 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 $1/|F|$, since there is only one random challenge $r \in F$ for which $\sigma$ would be zero. \qed

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 \( \delta_a[h]^{(i)} = 0 \), for all \( i \notin A \) and \( h \in [\tau] \), with high probability.

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

\[
-e_{Auth} = \sum_{h \in \tau} r[h] \cdot \left( \sum_{i \notin A} b^{(i)} \cdot \delta_a^{(i)}[h] + \sum_{i \in A} \langle a_B^{(i)}[h], \delta_b[h]^{(i)} \rangle \right),
\]

where \( \{b^{(i)}\}_{i \notin A} \) are uniformly random in \( \mathbb{F} \) and \( \{a_B[h]^{(i)}\}_{i} \) 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

\[
e_a = \sum_{h \in \tau} r_h \cdot \sum_{i \notin A} \langle a_B^{(i)}[h], \delta_b[h]^{(i)} \rangle,
\]

and let \( m = n - |A| \) be the number of honest parties, and \( S \) the set of all possible honest shares \( (a_B^{(i)})_{i \notin A} \) of \( a_B \), which are determined by the adversarial errors, and for which the sacrifice would pass. The value \( (a_B^{(i)})_{i \notin A} \) is uniformly distributed in \( S \), and so its min-entropy is \( \log|S| \). This means that there exists an \( i \) such that \( a_B^{(i)} \) has min-entropy at least \( \log |S|/m \). Since at least one \( a_B^{(i)} \) has min-entropy \( \log |S|/m \), and each \( a_B^{(i)} \) is independent, it follows that the shared value \( a_B = \sum_{i \in [n]} a_B^{(i)} \) has min-entropy at least \( \log |S|/m \).

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

\[
H_\infty(a) \geq \frac{\log |S|}{m} = \frac{\log (\beta \cdot 2^{mk\tau})}{m} = k \cdot \tau - \frac{c}{m} \geq k \cdot \tau - c
\]

(10)

Noting that \( k = \lceil \log |\mathbb{F}| \rceil \), (10) implies that (9) is satisfied for \( \kappa = (k(\tau - 2) - c)/2 \). 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

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

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 \( \tau = 3 \), \( \Pi_{\text{Triple}} \) securely implements \( \mathcal{F}_{\text{Triple}} \) with statistical security parameter 64.

C Other functionalities

33
The Functionality \( \mathcal{F}_{\text{Comm}} \)

**Commit:** On input \((\text{Comm, } v, i, \tau_v)\) 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 \((v, i, \tau_v)\) on a list and outputs \((i, \tau_v)\) to all parties and adversary.

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

Figure 10: Ideal Commitments

---

**Functionality \( \mathcal{F}_{\text{Rand}} \)**

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

Figure 11: Functionality \( \mathcal{F}_{\text{Rand}} \)