Cerberus: Secure Computation for Zero-Trust Environments

Version 0.4

Nicolas Le Bel Alberto Ibarrondo Sergiu Carpov
Marius Vuille Victor Treinsoutrot Daniel Filipe Nunes Silva
Yannik Schrade* Nico Schapeler*

Abstract

Cerberus lets mutually distrustful parties jointly compute over private data without ever exposing it: inputs stay secret and outputs reach only their intended recipient, or remain encrypted, so the computing parties learn nothing, neither inputs nor results. It provides the strongest guarantees in Multi-Party Computation (MPC), staying secure in the *dishonest-majority* setting where all but *one* of the n parties may be corrupted and deviate arbitrarily. For an honest party this amounts to *zero trust*: its inputs remain private and it is never coerced into accepting an incorrect result, no matter how many others collude. Cerberus further achieves *identifiable abort*, pinpointing and excluding any misbehaving party. This is the unlock for the *decentralized, permissionless* setting: by recasting malicious behavior from an anonymous denial-of-service into an accountable act, sensitive computation can be safely outsourced to a permissionless set of mutually distrustful third parties.

This document specifies the protocol end to end. Building on the design of [BMRS24], Cerberus composes the full cryptographic stack: low-level primitives (Oblivious Transfer, Oblivious Linear Evaluation and its vector form VOLE, authenticated secret sharing, and signatures), an input-independent preprocessing phase, and an online phase that securely evaluates arbitrary arithmetic circuits with minimal communication per gate, detailing every component alongside its optimizations.

These guarantees hold over any decentralized substrate offering public, append-only communication and a basis for accountability, a role a blockchain fills naturally. Cerberus is the most secure protocol powering Arcium, where computing parties are distinct from data owners: the network computes over private inputs from users and applications outside the MPC set who trust no individual node, with a blockchain anchoring state, input & output delivery, and the economic incentives behind accountability, so anyone can outsource computation to the network.

*Corresponding authors: yannik@arcium.com, nico@arcium.com

Contents

1 Introduction

Preface: How to read this document. This specification aims to be as thorough and self-contained as possible, providing the necessary background to understand the implemented protocols, their security properties, and the design choices made.

A reader interested in the circuit/gates, the overall system and/or its components should skip the foundations and primitives, and focus on the n-party preprocessing and online phase sections. Later, if any of the protocols or primitives pique their interest, they can jump back to the corresponding sections for more details. An expert reader may skip the definitions in the first two chapters, but should still read the last paragraphs of each section, as they summarize the design and implementation choices made regarding said protocol or primitive.

1.1 Our Goals

As an encrypted computation network, Arcium aims to provide state-of-the-art MPC protocols to execute arbitrary mathematical operations while operating under the tightest security guarantees. We aim for protocols to be run by n parties for a moderately small n, since most of the computation protocols aren't fault tolerant, and we assume a fully connected network where all parties can communicate with each other.

This document specifies Cerberus, the maliciously secure protocol of the Arcium suite. Cerberus prioritizes security above all else, providing n-party computation that withstands actively malicious adversaries who may deviate arbitrarily from the protocol. Set in the dishonest-majority threat model, it relies on a single honest party among the n participants and, whenever a deviation is detected, aborts with identifiable abort, pinpointing and excluding the misbehaving party rather than silently returning an incorrect result. Cerberus complements Manticore, the suite's n-party semi-honest protocol optimized for efficiency and scalability, which targets use cases with a less stringent threat model. Manticore is out of the scope of this document.

We conducted exhaustive literature reviews to select our suite of protocols, picking among those whose security is guaranteed under minimal and/or well-tested and well-understood falsifiable assumptions, e.g., Learning Parity with Noise (LPN), Elliptic-Curve Diffie-Hellman (ECDH). Given that many instances of these protocols will run concurrently, we favor protocols that were proven secure under concurrent composition in the Universal Composability (UC) paradigm [Can01], guaranteeing that their security remains independent of any other runs of the same protocol taking place during its execution.

While we take an active best-effort approach to write Constant-Time (CT) implementations of protocols, we neither claim nor expect our implementations to be CT. We argue that a local timing attack in one of the participants of a protocol is akin to a semi-honest corruption and thus strictly weaker than a malicious corruption, already covered by our threat models. Besides, we disregard remote timing attacks due to the inherently volatile nature of communications

(latency, jitter), expecting our attempt at CT to be enough against these attacks.

In the absence of standardization, we opt for peer-reviewed protocols and primitives backed by ample community convergence. In the instances where customizations to a protocol are necessary, we carefully analyze and meticulously present them. Lastly, we submitted this document alongside our implementation to both the main author of [BMRS24] and to an independent domain-expert auditing entity, incorporating their feedback and addressing any issues raised.

The present paper limits its scope to cryptographic protocols, their required primitives and some relevant implementation choices (e.g., local connection management) around them. We purposely leave out higher level details pertaining the Arcium network and implementation language, e.g., how to establish a set of parties (*MXE*) to run these protocols, how are nodes incentivized to participate, how do nodes interact with a ledger, the subset of Rust covered by our DSL, etc. We refer the interested reader to Arcium's official documentation[1] for these details.

1.2 Cerberus in a Nutshell

Following the trends in MPC, we split our MPC executions into two phases: the *offline* (input-independent) phase and the *online* (input-dependent) phase. We leverage the offline phase to generate one-time-use *preprocessing* data that will be consumed in the execution of the online phase once the inputs are known. This approach allows us to optimize the online phase, which is the one that ultimately determines the user experience. We design our protocols to minimize the amount of interaction and communication rounds in the online phase, often at the cost of a more intensive offline phase.

The diagram in Figure 1 illustrates the full lifecycle of a secure multiparty computation (MPC) workflow. It begins with a distributed (without trusted dealer) generation of long-term secrets (e.g., shared encryption keys) and correlated randomness among a set of computing parties. These shares, denoted as s_i and r_i, are crucial resources: the long-term secrets anchor the security of the system, while the correlated randomness is consumed during computation to maintain privacy and correctness. Input providers then supply private data (x) to the network, which is split into additive shares $(x_1, x_2, \ldots, x_n)$ and distributed among the parties by communicating via the blockchain. At this point, each party holds a unique combination of the share of the long term secret, the correlated randomness, and its shares of the input (e.g., an external input submitted by a data provider, a secret state held in encrypted form in the blockchain).

Once all the necessary inputs and setup values are distributed, the parties jointly execute the MPC protocol. The protocol takes as input the vectors of shares (x_i, s_i, r_i) across all parties and computes corresponding output shares $(y_1, \ldots, y_n)$. During this process, correlated randomness is consumed to pre-

[1] https://docs.arcium.com/

serve security guarantees against leakage. After the computation concludes, the output shares are combined to reconstruct the final result $y = f(x)$, representing the agreed function evaluation over the inputs. The blockchain provides a transparent and verifiable communication layer, ensuring that inputs are reliably distributed and results can be reconstructed with accountability. Together, the diagram highlights the sequential structure of ARCIUM's MPC computation: trustless setup, distributed correlated randomness generation, input distribution via the blockchain, privacy-preserving computation, and output reconstruction.

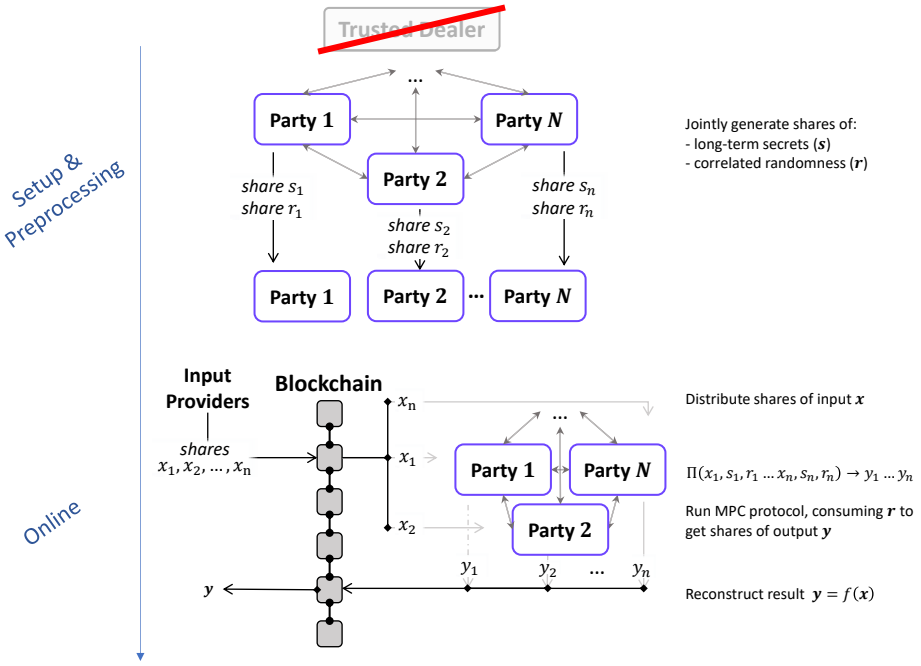

Figure 1: Overview of the execution of a computation in the ARCIUM network.

Following this layout, we devote this spec to describing the different protocols and cryptographic primitives that compose the full stack of ARCIUM's most secure protocol, CERBERUS, based on the design from BMRS24 [BMRS24]. The spec is organized following a top-down approach. We kick off with the foundations in Section 2, covering the main cryptographic and mathematical tools needed to formulate all the protocols. We then describe the cryptographic primitives (e.g., OT, VOLE) required by our main protocols in Section 3. We introduce the preprocessing protocols in Section 4, and cover the online phase circuits consuming said preprocessing in Section 5. Lastly, we present the architecture of the full protocol stack in Section 6.

Table 1: Notation, Symbols and Operators

Symbol	Description
a	An integer value $\in \mathbb{Z}$, equivalent to a bit-string $\in \{0,1\}^*$
$\|a\|$	Bit-length of bit-string a.
$\boldsymbol{a}$	A vector of values $\{a_{(1)}, a_{(2)} \dots\}$
$a_{(i)}$	ith element of vector $\boldsymbol{a}$
S	A set with elements $\{s_1, s_2, \dots\}$
$[n]$	Set of integers $\{1, 2, \dots, n\}$
$\|\mathrm{S}\|$	Number of elements in set S (its cardinality)
s_i	ith element of a set S, with $1 \leq i \leq \|\mathrm{S}\|$
$[n]^k$	Set of all k-subsets of $[n]$, that is, $[n]^k \triangleq \{\mathrm{X}\colon \mathrm{X} \subseteq [n] \wedge \|\mathrm{X}\| = k\}$
$A = (A_x, A_y)$	A group element (e.g., EC point), with coordinates A_x, A_y
$\mathbf{p}$	A polynomial of a certain finite degree d
$\mathbf{p}(x)$	Polynomial evaluated on point x
$\mathbf{p}_i$	ith coefficient of polynomial $\mathbf{p}$, s.t. $\mathbf{p}(x) = \sum_{i=0}^{d} \mathbf{p}_i x^i$
pk, sk	Public and private encryption keys
$[\![a]\!]$	Ciphertext, encryption of value a with pk
"tag"	An ascii-encoded string
$a \leftarrow 1$	Assignment operator. Set the value of a to 1
$a \xleftarrow{\$} \mathrm{S}$	Sampling operator. Sample value a uniformly from set S
$(a)_{\bullet}$	Operator to appends a to the transcript
$a \xleftarrow{\bullet} 1$	Assign value 1 to a and append a to the transcript
$a \stackrel{?}{=} 1$	Equality test operator. Returns 1 if true, 0 otherwise
$a \mod q$	Modulo operator
$a \mid\mid q$	Concatenation operator (elements, sets, bit-strings...)
$\mathsf{Func}_a(x)$	Function parametrized by a with input x
$\mathsf{gcd}(x, y)$	Greatest common divisor of x and y
$\mathsf{lcm}(x, y)$	Least common multiple of x and y
$\mathsf{quot}(x)$	Quotient function $\lfloor \frac{x-1}{n} \rfloor$
$\mathsf{sgn}(x)$	Sign function, 1 if $x > 0$, -1 if $x < 0$, 0 if $x = 0$
$\mathcal{P}_1, \mathcal{P}_2, \dots$	Computing parties in MPC protocol
$\mathcal{S}, \mathcal{R}, \dots$	Parties with a specific role (e.g., *Sender*, *Receiver*, ...)
$\mathcal{P}_k.\mathsf{Func}(x)$	Party k runs a certain function Func on input x.
$\mathcal{F}_{name}$	An ideal functionality
κ, λ	Computational security parameters (typ. 128-256 bits)
σ, λ_s	The statistical security parameters (typ. 80-128 bits)
$\mathbb{G}(q, G)$	Cyclic group of prime order q and generator G
$\mathrm{E}(\mathbb{G}, q, G, I)$	Elliptic curve with group $\mathbb{G}(q, G)$ and identity element I

1.3 Notation

We use plain low-case letters (e.g., a, k) for scalars, bold letters to denote vectors (e.g., $\boldsymbol{x}$, $\boldsymbol{y}$), upper-case non-italic letters for sets (e.g., $\mathrm{S} = \{1, 2, 3\}$) and upper-case italic letters (e.g., A) to denote points in elliptic curves. $x_{(i)}$ denotes the ith element of vector $\boldsymbol{x}$. We write a polynomial $\mathbf{p}$ of degree d as $\mathbf{p}(x) = \sum_{i=0}^{d-1} \mathbf{p}_i x^i$, where $\mathbf{p}_i$ is the ith coefficient of $\mathbf{p}$. We use $q \leftarrow 4$ to set a local variable q to 4, $a \stackrel{?}{=} b$ to check whether a is equal to b (returning a 0 or 1 as result), and $a = b$ to denote equivalence between a and b. We note U_S as the uniform random distribution in a set S, and write $r \xleftarrow{\$} S$ to indicate sampling from U_S and assigning the sample to r. We denote λ as the computational security parameter, and σ as the statistical security parameter.

In the context of MPC protocols, $\mathcal{P}_1, \mathcal{P}_2, \ldots$ denote the computing parties. We generalize behavior common to a set of n parties by resorting to $\mathcal{P}_i$ for an index $i \in \{1, 2, \ldots, n\}$, and employ $j \in \{1, 2, \ldots, n\} \setminus \{i\}$ for behavior common to all other parties $\mathcal{P}_j \neq \mathcal{P}_i$ given a certain party $\mathcal{P}_i$. Certain parties fulfilling a specific roles are denoted with that same font (e.g., $\mathcal{R}$ for receiver, $\mathcal{S}$ for sender). We employ sans-serif fonts for functions (sub-routines) and protocol rounds (e.g., $\mathsf{Round1}$ for the first round of a protocol, $\mathsf{gcd}(x, y)$ for the greatest common divisor of x and y), and denote an ideal functionality as $\mathcal{F}_{name}$. For convenience, we summarize our notation choices in Table 1.

2 Foundations

2.1 On Secure Multiparty Computation

Secure Multi-Party Computation (MPC) is a foundational branch of cryptography that enables a group of parties, each potentially holding private inputs, to jointly compute a function of their inputs while preserving two central guarantees: *privacy*, meaning that no party learns more than what can be inferred from its own input and the output; and *correctness*, meaning that the output is consistent with the prescribed function, even if some parties behave dishonestly. Formally, MPC can be defined in the framework of secure function evaluation (SFE) [Yao82, GMW87, BOGW88, CCD88], where n parties wish to compute $f(x_1, \ldots, x_n)$ for private inputs x_i.

The central question in MPC research is: what security guarantees can be achieved under different adversary models, network and setup assumptions?

Parties/Participants and the Environment. The environment defines the set of characteristics and assumptions that we make about the context in which the privacy-preserving protocols take place and where adversary operates. As part of this environment we designate a *party/participant* (can be used interchangeably) to represent an actual real-world entity, consisting of a set of one or several Interactive Turing Machines [Can01] capable of executing Probabilistic Polynomial Time (PPT) algorithms that may use randomness to produce non-deterministic results. We borrow Definition 4 from [Can01], informally extending a classical Turing Machine with the ability to interact with other machines that may or may not belong to the same party. This can be seen as a generalization of the Turing Machine pairs defined in [GMR19] for the case of multiple parties.

Adversarial Behavior. An *adversary* in MPC represents an entity attempting to break the security of the protocol by corrupting a set of parties. We distinguish between two main types of adversaries:

- **Semi-honest adversaries** (also called *passive*) follow the protocol but try to infer additional information from the transcript. Protocols secure in this model are easier to design and often efficient, but provide weaker real-world security.
- **Malicious adversaries** (also called *active*) may deviate arbitrarily, by sending false messages, aborting early, withholding certain messages etc... Security against malicious adversaries is stronger but significantly more challenging to obtain.

Honest vs. Dishonest Majority. A crucial distinction is whether a majority of the parties are assumed to follow the protocol specification:

- **Honest majority.** If strictly more than half the parties are honest, then general MPC can be achieved with unconditional (information-theoretic) security, even against malicious adversaries [BOGW88, CCD88].
- **Dishonest majority.** If half or more of the parties may be corrupted, strong notions such as fairness and guaranteed output delivery are impossible [Cle86a]. Instead, weaker guarantees such as *security with abort* must be adopted [GMW87].

Termination Guarantees. Even when correctness and privacy hold, adversaries can interfere with completion:

- **Security with abort.** Either all honest parties receive the correct output, or all learn that the protocol has aborted. However, a single malicious party can force an abort, creating a denial-of-service vulnerability.
- **Robustness.** A protocol is robust if adversaries cannot prevent honest parties from obtaining their outputs. This is closely related to classical *Byzantine fault tolerance* in distributed computing.

Identifiable Abort and Public Accountability. Since robustness is unattainable in the dishonest majority setting, refined abort notions are important:

- **Identifiable abort.** Upon abort, at least one corrupted party is identified as the cause, allowing exclusion and potential restart of the computation.
- **Publicly identifiable abort.** The identity of a misbehaving party is revealed in a way that is verifiable by external observers. This accountability enables punishment or exclusion in real-world applications.

Feasibility Bounds. Based on the categorization above, there are innate limitations on what can be achieved in MPC. Several seminal works established long ago what is feasible [Yao82, GMW87, BOGW88, CCD88, Cle86a, IOZ12]. We revisit them :

- With an honest majority, unconditional security against malicious adversaries is achievable.
- With a dishonest majority, fairness and guaranteed output delivery are impossible; instead, one settles for security with abort.
- In the two-party setting, fairness is impossible in general, but security with abort is achievable under standard assumptions.
- With additional setup assumptions, such as correlated randomness or oblivious transfer, stronger guarantees like identifiable abort become possible, though unconditional realizations remain subject to impossibility results [IOZ12].

In this line, we select a favorable setting (semi-honest security) to get as much efficiency as possible out of the protocols in MANTICORE, while we select a stronger setting (malicious security with identifiable abort) to maximize security in CERBERUS, given the infeasibility of achieving full robustness in the dishonest majority setting.

2.2 Cryptographic Assumptions

Modern cryptographic protocols rely on certain computational problems that are believed to be difficult to solve efficiently. In this section we outline the main assumptions that underpin much of public-key cryptography. These include the hardness of discrete logarithms and Diffie–Hellman problems, the concept of indistinguishability, the principle of random self-reducibility, and the use of the random oracle model for reasoning about hash functions.

2.2.1 Discrete Logarithm and Diffie–Hellman Assumptions

Let $G = \langle g \rangle$ be a cyclic group of order n. Several security assumptions are defined in this setting:

Discrete Logarithm (DL) Assumption: Given an element $h = g^x$ for a secret exponent $x \in \mathbb{Z}_n$, it is computationally hard to recover x.

Computational Diffie–Hellman (CDH) Assumption: Given g^x and g^y for random exponents $x, y \in \mathbb{Z}_n$, it is hard to compute the shared value g^{xy}.

Decisional Diffie–Hellman (DDH) Assumption: Given the triple (g^x, g^y, g^z), it is hard to decide whether $z = xy$ (i.e., whether the third component is g^{xy} or just a random element of the group).

Clearly, if one can solve the discrete logarithm problem, then CDH and DDH become easy. Thus we have the implication chain

$$\text{DL} \;\Rightarrow\; \text{CDH} \;\Rightarrow\; \text{DDH}.$$

However, many cryptographic schemes can only be proven secure under the stronger DDH assumption.

2.2.2 Indistinguishability

A key requirement in cryptography is that certain distributions of values should "look the same" to any efficient adversary. This is formalized using *statistical distance*. Depending on how close two distributions are, we distinguish three important cases:

- **Perfect indistinguishability:** The two distributions are exactly identical.

- **Statistical indistinguishability:** The statistical distance between the distributions is negligible (i.e., vanishes faster than any inverse polynomial in the security parameter).
- **Computational indistinguishability:** No efficient algorithm (a probabilistic polynomial-time distinguisher) can tell the distributions apart with more than negligible advantage.

For example, the DDH assumption can be restated as saying that the distribution of Diffie–Hellman triples (g^x, g^y, g^{xy}) is computationally indistinguishable from the distribution of random triples (g^x, g^y, g^z).

2.2.3 The Random Oracle Model (ROM)

Many cryptographic protocols use hash functions, which in practice are fixed algorithms such as SHA-256 [Dan15] or BLAKE2 [SA15]. To assist in proving security, proofs in the ROM model a hash function as an idealized *random oracle*. A random oracle is an abstract black box that, for each new query, returns a random output, but is consistent if the same input is queried again.

Working in the random oracle model allows security proofs to assume that hash outputs are completely unpredictable and independent. Of course, real hash functions are not truly random oracles, so results in this model should be seen as heuristic. Nevertheless, the random oracle model remains a powerful and widely used tool in many cryptographic proofs.

2.3 Proving security in MPC

Security in cryptography must be defined and proved formally. In the context of Multi-Party Computation, two major approaches dominate: game-based proofs and simulation-based proofs.

2.3.1 Game-Based Security Proofs

In the *game-based* paradigm, security is defined by an explicit challenge game between a challenger and an adversary. The adversary wins the game if it can break the intended security property (e.g., distinguishing a real transcript from a random one). A protocol is secure if every efficient adversary has only negligible advantage in winning.

This approach has the benefit of being concrete and easy to state. However, it often captures only specific attack scenarios and may not compose well: a protocol proven secure in one game may leak information when used as a subroutine in a larger protocol.

2.3.2 Simulation-Based Security Proofs

The *simulation paradigm*, foundational in modern MPC, defines security by comparing interactions occurring in two scenarios:

- The **ideal world**, where a trusted party computes the function f on the parties' inputs and returns outputs.
- The **real world**, where the parties execute the actual protocol in the presence of an adversary.

A protocol is secure if for every real-world adversary, there exists a *simulator* in the ideal world that can generate a view indistinguishable from the adversary's real-world view [Lin17]. Intuitively, this means the adversary learns nothing more in the real world than what it could have learned in the ideal world.

Simulation-based proofs are powerful because they yield strong security guarantees and allow protocols to compose modularly. They are, however, technically demanding: the simulator must often both extract the adversary's effective input and generate a consistent transcript, which requires delicate constructions.

Additional Considerations. Several refinements and extensions have been developed:

- **Modular composition.** Generic simulation-based proofs support mainly sequential composition, whereas Universal Composability (below) ensures concurrent composition.
- **Adaptive security.** In some settings, adversaries may decide dynamically which parties to corrupt. Achieving adaptive security requires special care, often involving erasure or forward-secure techniques.
- **Random oracle and hybrid models.** In practice, proofs sometimes rely on idealized models (e.g., random oracles or ideal functionalities) as stepping stones toward constructing real-world protocols.

2.3.3 Universal Composability (UC) Security

The *Universal Composability* framework [Can01] extends simulation-based security by requiring that security hold under arbitrary concurrent composition. In UC, protocols are modeled as interactive machines running concurrently, and the environment represents all external influences. A protocol securely realizes an ideal functionality if no environment can distinguish between interacting with the real protocol and interacting with the ideal functionality and a simulator.

UC security is considered the "gold standard" for cryptographic protocols. It ensures that protocols proven secure remain secure even when arbitrarily composed with others—a necessity for real-world deployment. However, achieving UC security often requires setup assumptions, such as a common reference string (CRS) or public-key infrastructure (PKI).

In summary, game-based proofs provide concrete but narrow guarantees; simulation-based proofs capture strong ideal/real-world equivalence; and UC security elevates these guarantees to full composability. For MPC, simulation-based and UC security are the dominant frameworks, forming the backbone of our choices of protocols.

2.3.4 Session Id and Distributed Randomness Sampling

To achieve Universally-Composability (UC) our protocols are framed in the Common Reference String (CRS) [CLOS02] model, crucially relying on the existence of a common value from an arbitrary distribution that is agreed-upon and known by all the participants. This value acts as a unique session identifier for each protocol execution, effectively separating multiple executions of the same protocol.

While this value need not be random in all CRS settings, we employ a protocol named drand to sample a uniformly random value that will serve as CRS. This protocol realizes the $\mathcal{F}^{CRS}$ ideal functionality as defined in [CLOS02]. We instantiate this protocol following stablished literature in the domain of CRS-based ZK-SNARKs [BSCG+15, BGG19, Mie19], where a pre-commitment phase is employed to ensure that the final random value is not biased in presence of at least one honest party. This protocol is presented in Protocol 2.1.

Protocol 2.1 drand

A protocol inspired by [BSCG+15] to sample a uniformly random value r among n parties. It requires a commitment scheme (e.g., Scheme 3.1), a broadcast channel $\mathcal{F}^{Broadcast}$ and a hash function H (e.g., SHA-256 [Dwo15], a $\mathsf{Transcript}$)

Players: $\mathcal{P}_1, \ldots, \mathcal{P}_i, \ldots, \mathcal{P}_n$
Outputs: r, a random value

$\mathcal{P}_i.\mathsf{Round1}() \dashrightarrow c_i$
1: Sample $r_i \xleftarrow{\$} \mathbb{Z}_{q^*}$ as the random value of $\mathcal{P}_i$.
2: Run $(c_i, w_i) \leftarrow \mathsf{Commit}(r_i)$ to get the commitment c_i and the witness w_i.
3: $\mathcal{F}^{Broadcast}(c_i)$ to distribute all commitments c_i to all parties.

$\mathcal{P}_i.\mathsf{Round2}(\boldsymbol{c} = \{c_1, c_2, ..., c_n\}) \dashrightarrow r_i, w_i$
1: $\mathcal{F}^{Broadcast}(r_i, w_i)$ to reveal r_i and w_i to all parties.

$\mathcal{P}_i.\mathsf{Round3}(\boldsymbol{c} = \{c_1, c_2, ..., c_n\}, \boldsymbol{r} = \{r_1, r_2, ..., r_n\}, \boldsymbol{w} = \{w_1, w_2, ..., w_n\}) \dashrightarrow r$
1: **for** $i \in [n]$ **do**
2: $\quad \mathsf{Open}(r_i, c_i, w_i)$, ABORT if it fails.
3: $r \leftarrow \mathsf{H}(\boldsymbol{r})$
return r

This protocol can be framed as an instance of Distributed Randomness Beacons [CMB23] where the beacon is managed by the participants themselves. We note that, in the context of an honest majority (e.g., during DKG in a 2-out-of-3 setting) this protocol can instead be instantiated with a Publicly Verifiable Secret Sharing (PVSS) scheme to ensure that the final random value is fully unbiased. However, we opt for a simpler instantiation that does not require a PVSS scheme, as the protocol is used to sample a CRS and thus does not rely on the randomness being perfectly unbiased (it just requires the CRS to be unique with high enough probability).

As a side note, we remark that there is a formal separation between the low-

round distributed randomness sampling protocols described above and the high-round coin flipping protocols widely studied in the literature [Blu83, Cle86b, BOO15, WAS22]. The coin flipping protocols suffer from a fundamental impossibility result when trying to achieve fairness (all parties receiving the unbiased coin): a rushing adversary can wait until he has received all the messages in the last round, simulate the coin result and decide whether to abort if the result is not desired. The main difference comes from the adversary's ability to run as many protocols as he wants in parallel, whereas we can run our distributed randomness sampling protocol much more sporadically and deal with the aborts in the last round in a more controlled manner: we can limit the number of aborts to a low enough number before freezing and investigating the issue. That being said, we leave these control mechanisms out of the scope of the primitives, much like the total aborts in some of our signing protocols.

3 Primitives

3.1 Networking

We define a comprehensive networking model that supports various communication patterns required for secure multi-party computation. The network functionality $\mathcal{F}^{Network}$ (Functionality 3.1) provides a unified interface for different networking primitives.

Functionality 3.1 $\mathcal{F}^{Network}$

Network functionality providing authenticated point-to-point and broadcast communication between parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$.

Players: n parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$.
Inputs: Session identifier $\text{sid} \in \{0,1\}^*$.
Outputs: Authenticated message delivery with integrity and consistency guarantees.

$\mathcal{F}^{Network}$.**Send**(j, m)--→(SENT, RECEIVED)
On input $(\text{SEND}, \text{sid}, j, m)$ from party $\mathcal{P}_i$ where $j \in [n]$ and $j \neq i$: queue message m for delivery to $\mathcal{P}_j$, output $(\text{SENT}, \text{sid}, i)$ to $\mathcal{P}_i$, and when ready output $(\text{RECEIVED}, \text{sid}, i, m)$ to $\mathcal{P}_j$.

$\mathcal{F}^{Network}$.**Broadcast**(m)--→(BROADCASTED, RECEIVED)
On input $(\text{BROADCAST}, \text{sid}, m)$ from party $\mathcal{P}_i$: for each $j \in [n]$ with $j \neq i$, queue message m for delivery to $\mathcal{P}_j$, output $(\text{BROADCASTED}, \text{sid})$ to $\mathcal{P}_i$, and when ready output $(\text{RECEIVED}, \text{sid}, i, m)$ to each $\mathcal{P}_j$.

$\mathcal{F}^{Network}$.**OpenShare**$(\llbracket v \rrbracket)$--→$v$
On input $(\text{OPENSHARE}, \text{sid}, \llbracket v \rrbracket)$ from all parties: each party broadcasts their share component, collect all n share components, reconstruct v from the shares, and output $(\text{OPENED}, \text{sid}, v)$ to all parties.

3.1.1 Point-to-Point Communication

We assume that participants are connected via point-to-point authenticated channels. Sending a message through this channel is denoted by invoking $\mathcal{F}^{Send}$ functionality, which is defined in Functionality 3.2. To simplify the notation, we will call this functionality Send and use it via the convention $\mathcal{P}_i.\text{Send}(m) \to \mathcal{P}_j$

Functionality 3.2 $\mathcal{F}^{Send}$

A functionality for authenticated Peer-To-Peer (P2P) communication between a pair of parties $\mathcal{P}_i, \mathcal{P}_j$.

- On receiving (P2PSEND-IN, $j, \boldsymbol{m}$) from $\mathcal{P}_i$ with id i, $\mathcal{F}^{Send}$ sends (P2PSEND-OUT, $i, \boldsymbol{m}$) to party $\mathcal{P}_j$ with id j.

3.1.2 Broadcast Communication

We also use a broadcast functionality (Functionality 3.3), which is when players need to send the same message to all other players.

Functionality 3.3 $\mathcal{F}^{Broadcast}$

Authenticated Broadcast communication among all parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$.

- On receiving (BROADCAST-IN, m) from $\mathcal{P}_i$ with id i, $\mathcal{F}^{Broadcast}$ sends (BROADCAST-OUT,i, m) to $\mathcal{P}_j$ $\forall j \in [n]$.

$\mathcal{F}^{Broadcast}$ can be UC-realized, with security with abort in a Byzantine setting, via the two-round Echo - Broadcast protocol formalized in [GL02, Protocol 2]. In this context, we require that whenever a party aborts, they send a message to all other parties indicating that they have aborted. Honest parties should abort whenever they receive an abort message. Note that many protocols can amortize the second message of the echo broadcast and send it only at the end. Adding identifiability to this protocol is a highly non-trivial task, and as such we leave it for future work.

In the context of dishonest majority, broadcast cannot be guaranteed to terminate unless we incorporate further assumptions. We tackle this in two possible ways:

- **Default to a bigger cluster with honest majority:** If parties don't reach a consensus, they can fall back to a larger group of nodes where we can assume honest majority, given that these nodes are part of a blockchain and this honest majority assumption is already in place for a big enough subset of the network.
- **Post in the blockchain:** If parties don't reach a consensus, they can post their messages in the blockchain, which is assumed to be public and append-only. This way, all parties can read the messages from the blockchain and naturally reach a consensus.

Note that both approaches incur in significant costs, and as such they must be used with caution. Further treatment of the economic penalties incurred by these approaches is left for higher level design considerations, out of the scope of this document.

3.2 Random Generation

Most cryptographic primitives require a generating mechanism to sample random elements. These generators must satisfy:

- The *next-bit test*, that is, given the first i bits of a random sequence, there is no polynomial-time algorithm that can predict the $(k+1)-$th bit with probability of success non-negligibly better than 50% [KL07]. In other words, the output distribution is computationally undistinguishable from a true random distribution. As a byproduct, cryptographically-secure random generators should pass the *statistical randomness tests* often used to characterize the quality of Pseudo-Random Number Generators (PRNGs) (e.g., Diehard [Mar08], TestU01 [LS07]).

- *Forward secrecy*: If part or all of a generator's internal state is revealed, it should be infeasible to reconstruct the stream of random numbers prior to the revelation.

- *Backward secrecy*: If there is an entropy input while running, it should be infeasible to use knowledge of the input's state to predict future conditions of the generator's state.

Based on these properties, we classify random generators into two categories:

- **(Unkeyed) Random Generator**: A mechanism to generate local random elements without the need for a seed via non-deterministic algorithms, e.g., in the generation of cryptographic keys or other long-term secrets. Typically, a short seed is obtained from a physical source of entropy (e.g., thermal noise, radioactive decay, etc.) and then extended via a cryptographically secure PRG for faster generation. We employ the operating system's randomness API (e.g., `/dev/random` in Unix-like systems) to sample a high-entropy seed, and resort to a properly managed PRG seeded with said seed, e.g., Rust's `ThreadRng`, to generate random elements as needed.

- **Cryptographically-Secure (Keyed) Pseudo-Random Generator (PRG)**: These generators use an algorithm to generate random numbers by deterministically "extending" a high-entropy source used as seed. The output of these generators is unpredictable and statistically independent up to a certain degree (computational indistinguishability from a true random distribution). In practice we employ either `ChaCha12` (default in Rust's `rand` crate, a) or `AES-CTR` (when hardware acceleration is available) as the underlying PRG in all our protocols and constructions. We model the PRG mechanism in Functionality 3.4.

Functionality 3.4 $\mathcal{F}^{PRG}$

Pseudo-Random Generator, defined by two algorithms:

- $\mathcal{F}^{PRG}.\mathsf{Seed}(s)$ seeds a PRNG instance with seed s.
- $\mathcal{F}^{PRG}.\mathsf{Sample}(n) \to u \in \{0,1\}^n$ generates n uniformly random bits.

In cases where a prg needs to be seeded multiple times, we simplify to $x \leftarrow \mathsf{PRG}_n(s)$ to represent $\mathcal{F}^{PRG}.\mathsf{Seed}(s)$ followed by $x \leftarrow \mathcal{F}^{PRG}.\mathsf{Sample}(n)$

We employ both types of generators in our protocols. In our Cerberus protocol stack, we use Rust's default unkeyed random generator (stemming from the OS) to sample a long-term seed. This seed fuels a keyed PRG instance used to generate all the randomness required by the different protocols deterministically, allowing for reproducibility of the entire execution as required to identify misbehaving parties. In accordance with Rust crypto, we select the stream cipher ChaCha12 (a shortened version of ChaCha20 [B+08] considered sufficiently secure) as our default PRG.

From this point onwards, we denote $r \xleftarrow{\$} \mathbb{Z}_q$ to uniformly sample a value r from a set ($\mathbb{Z}_q$ in this example) by using said keyed PRG unless otherwise noted. In cases where q is not a power of two, we perform one of the following:

- *rejection sampling*: sample a bit-string $r \in \mathbb{Z}_{2^{|q|}}$ of bit-length $|q|$ as many times as needed until $r > q$.
- *constant-time modular reduction*: sample a sufficiently large number $r \xleftarrow{\$} \mathbb{Z}_{2^{|q|+\lambda}}$ and reduce it $r \leftarrow r' \mod q$ at the cost of some acceptably low bias ($\approx 2^{-\lambda}$).

We employ rejection sampling for generation of non-zero elements (e.g., in $\mathbb{Z}_q^*$), and favor the constant-time modular reduction when sampling elements in $\mathbb{Z}_q$ via pre-existing constant-time implementations.

3.3 Hashing

A cryptographic hash function $\mathsf{H}(m)$ is a one-way function mapping arbitrary inputs, typically of a variable size, to seemingly uncorrelated outputs of a defined size [KL07, Chapter 4]. We refer to the output of a hash function for a given input as its digest. A hash function holds several properties:

- *Preimage resistance*: Given a digest d, it should be difficult to find any input message m such that $d = \mathsf{H}(m)$. This property is sometimes named "Security Strength" or "One-Way" in the literature.
- *Second-preimage resistance*: for a specified input m, it is computationally infeasible to find an input m' producing the same result $\mathsf{H}(m') = \mathsf{H}(m)$.
- *Collision resistance*: It should be difficult to find a pair different messages m_1 and m_2 such that $\mathsf{H}(m_1) = \mathsf{H}(m_2)$. This pair is called a *collision*. Due

to the birthday attack, a digest size of at least $2n$ bits is required for n bits of collision resistance.

In our protocols, we make extensive use of hash functions to convert interactive protocols into its non-interactive version via several transformations, to Commit on a secret value before opening it, and in general as a method to realize Random Oracles (ROs) by employing some agreed-upon fresh random value (the session ID, but more of that on Section 2.3.4) alongside other arbitrary tags for domain separation.

Given the ubiquity and importance of hash functions in our protocols, we pick BLAKE3 [ONAZ] to benefit from its performance. BLAKE3 is based on an optimized instance of the established hash function BLAKE2 [SA15]. Additionally, we implement the hash-to-curve primitives from RFC 9380 [FHSS+23], defining mechanisms to hash arbitrary strings into both fields of any order and elliptic curve points.

TmmoHash. Following a common approach from the MPC literature, we implement a cheaper block-cipher-based hash for OTs and other equivalent bit-level hashing inside protocols using the Tweakable Matyas-Meyer-Oseas (TMMO) construction of [GKWY20, Section 7.4]. This construction benefits from hardware support of AES[2] in fixed-key block mode (ECB) to iteratively compute the digest. The main component of this hash function uses a block cipher (AES-128 in our case) as an ideal permutation π. With an input x of size a single block of π, and an an output with n blocks, the TMMO construction is defined as:

$$digest_{(i)} = \mathsf{TMMO}^{\pi}(x, i) = \pi(\pi(x) \oplus i) \oplus \pi(x) \quad \forall i \in [n] \tag{1}$$

where $\pi(x)$ is the block cipher using as key the previous output of the TMMO $digest_{(i-1)}$ as prescribed by the Matyas-Meyer-Oseas construction, and employing a fixed Initialization Vector (IV) for the first block. The detailed description is condensed in Algorithm 3.1. Note that this construction provides both *correlation robustness*[3] as proven in [GKWY20], and *preimage resistance* due to its use of AES as a one-way function.

3.4 Commitments

A commitment scheme is a cryptographic primitive that allows a sender party to commit to a chosen value / statement while keeping it hidden to others, with the ability to reveal the committed value to all the receivers later. Interactions in a commitment scheme follow two steps, abstracted in Functionality 3.5.

[2] AES-NI and equivalent CPU instruction sets.

[3] More concretely, it achieves *tweakable* (admits variable output lengths) *circular correlation robustness*, a strictly stronger notion of correlation robustness.

Algorithm 3.1 $\mathsf{TmmoHash}_n(x)$

A fix-length-input and variable-length output hash constructed following the Tweakable Matyas-Meyer-Oseas (TMMO) construction from [GKWY20], using AES-128 in ECB mode as a permutation π, with $\kappa = 128$ bits. The first block uses an arbitrary initialization vector $IV \in \mathbb{Z}_2^\kappa$.

Inputs: $x \in \mathbb{Z}_2^\kappa$ an input of length κ bits
Outputs: $\boldsymbol{d} \in \mathbb{Z}_2^{n\times\kappa}$, a digest of length $n\kappa$ bits (n blocks of π)

1: $\pi.\mathsf{SetKey}(IV)$
2: **for** $i \in [n]$ **do**
3: $\quad y \leftarrow \pi.\mathsf{Encrypt}(x)$
4: $\quad z \leftarrow \pi.\mathsf{Encrypt}(y \oplus i)$
5: $\quad d_{(i)} \leftarrow z \oplus y$
6: $\quad \pi.\mathsf{SetKey}(d_{(i)})$
return $\boldsymbol{d}$

Functionality 3.5 $\mathcal{F}^{Commitment}$

Commitment scheme, composed of two algorithms:

- $\mathsf{Commit}(m, w) \rightarrow c$: a value m is fixed, a commitment c to that value is generated using a witness w and sent to all receiving parties.
- $\mathsf{Open}(c, m, w) \rightarrow$ *valid* (a.k.a. *reveal*|*de-commitment*): committer reveals m alongside w, then the receivers validate that it matches the prior commitment c.

Commitment schemes possess two main properties: *hiding*, as the committed value m is kept secret from the receivers until the sender reveals it, and *binding*, as the sender cannot change the value m after sending the commit c. Commitment schemes are used in many cryptographic protocols, including zero-knowledge proofs, verifiable secret sharing, and secure multiparty computation. They can be instantiated from multiple assumptions (e.g., a CSPRNG, a one-way permutation, discrete log assumptions...) and they accept an optional input *sid* to achieve UC-security. We adopt two commitment schemes:

1. A folkloric *hash-based commitment scheme* detailed in Scheme 3.1.

Scheme 3.1 Commitment

A bit-level commitment scheme based on a hash function H. The commitment is implicitly broadcasted after the Commit step. Later, the sender reveals the committed value m and the receivers run the Open function to verify its validity.

Inputs: $\mathcal{P}_S : m$, an input message to commit and later open.
sid: a unique session identifier (optional, for UC-security).
Outputs: *valid* if the commitment is verified correctly.

Commit(m)--→(c, w)
1: Sample $w \xleftarrow{\$} \{0,1\}^*$, a random witness
2: $c \leftarrow \mathsf{H}(m \mid\mid w \mid\mid sid)$, a commitment to m
return (c, w)

Open(m, c, w)--→*valid*
1: $c' \leftarrow \mathsf{H}(m \mid\mid w \mid\mid sid)$
return *valid* if $c = c'$, ABORT otherwise

2. The *discrete-log-based commitment scheme* from [Ped91] over a group $\mathbb{G}$, detailed in Scheme 3.2.

Scheme 3.2 Pedersen Commitment (PC)

The commitment scheme from [Ped91, Section3] parametrized by a group $\mathbb{G}$ of prime order q with a generator G (e.g., an elliptic curve $\mathsf{E}(\mathbb{G}, q, G)$), and a second generator H chosen independently[4] from G.

Inputs: m, an input message to commit and later open.
Outputs: *valid* if the commitment is verified correctly.

Commit$(m \in \mathbb{Z}_q)$--→(C, w)
1: Sample $w \xleftarrow{\$} \mathbb{Z}_q$, a random witness
2: $C \leftarrow w \cdot G + m \cdot H$ as the commitment of m.
return (C, w)

Open(m, C, w)--→*valid*
1: $c' \leftarrow m \cdot G + w \cdot H$
return *valid* if $c \stackrel{?}{=} c'$, ABORT otherwise.

Both the broadcasting of c at the end of the Commit step and the sending of m prior to Open are implicit in our descriptions, as we delegate them to protocols using the schemes.

[4] Such that nobody knows x s.t. $H = x \cdot G$. This can effectively achieved via aggregation of commitments of independent random values following the instructions of [Ped91], or via $H \leftarrow \mathsf{Hash2Curve}(m)$ of a fixed message m (e.g., $m =$ "NOTHING UP MY SLEEVE") as we do.

3.5 Secret Sharing

Secret sharing refers to methods for distributing a secret s among multiple parties, in such a way that no individual holds any intelligible information about the secret s, but when a sufficient number of individuals combine their *shares*, s can be reconstructed unequivocally. In the setting of t-out-of-n (threshold) schemes, there is one dealer and n parties. The dealer generates a share of the secret for each of the parties (a.k.a. Split a secret), and any subset of t or more parties can together reconstruct the secret (a.k.a. Combine shares) but no group of fewer than t parties can. We use two standard sharing schemes in our protocols, both information-theoretic secure[5]:

- The *additive secret sharing* scheme (SS), a n-out-of-n sharing scheme where the secret is an element of a group, and the shares are group elements sampled uniformly at random, such that the secret is the sum of the shares evaluated in that group. Typical groups are $\mathbb{Z}_2$ (binary shares), $\mathbb{Z}_{2^k}$ (k-bit integer shares, convenient for simple CPU implementations when $k \in \{32, 64\}$), or a certain field $\mathbb{F}$ (e.g. $\mathbb{Z}_q$ for some prime q) as in ECC. SS can be naturally extended into a t-out-of-n scheme by stacking multiple SS instances[6]. We describe it formally in Scheme 3.3, stressing that it is implicitly used across many protocols and algorithms in this document.

Scheme 3.3 Additive Secret Sharing (SS)

Additive Secret Sharing Scheme. This scheme is parametrized by a group $\mathbb{G}$ in and the number of shares produced n.

$\mathit{Split}_n(x \in \mathbb{G}) \dashrightarrow (\boldsymbol{y} = \{y_{(1)}, \dots y_{(n)}\})$
1: **for** $i \in [n-1]$ **do**
2: Sample $y_{(i)} \xleftarrow{\$} \mathbb{G}$ as n - 1 random shares
3: $y_{(n)} \leftarrow x - \sum_{iins[n-1]} y_{(i)}$ as the last random share
return $\boldsymbol{y} = \{y_{(i)}\}_{i \in [n]}$ as the n shares

$\mathit{Combine}_n(\boldsymbol{y'} = \{y_{(i)} \in \mathbb{G}\}_{i \in [n]}) \dashrightarrow x$
1: $x \leftarrow \sum_{i \in [n]} y_{(i)}$
return x as the reconstructed secret

- The *Shamir secret sharing* scheme [Sha79] (SSS) sets a secret field element as the evaluation $\mathbf{p}(0)$ at the point $x = 0$ of a polynomial $\mathbf{p}$ with degree $t-1$ with randomly picked coefficients $\mathbf{p}_i$, such that the evaluations $\mathbf{p}(i)$ $\forall i \in [n]$ are the secret shares. We describe the scheme in Scheme 3.4.

[5] Meaning that a subset of $t-1$ shares does not reveal any information about the secret.

[6] It suffices to run $\binom{n}{t}$ independent t-out-of-t SS schemes, one for every possible subset of t parties. Note that this is way less efficient than SSS when the number of parties increase, specially for unbalanced ($t \approx n/2$) configurations.

Scheme 3.4 Shamir Secret Sharing (SSS)

Shamir's Secret Sharing Scheme [Sha79] based on polynomial interpolation. This scheme is parametrized by a finite field $\mathbb{F}$, the number of shares produced n, and the number of shares required to reconstruct t (threshold).

Split$_{t,n}(s \in \mathbb{F}) \dashrightarrow (\boldsymbol{y})$

1: Set $\mathbf{p}_0 \leftarrow s$ as the constant term of a polynomial $\mathbf{p}(x)$ of degree $t-1$.
2: **for** $k \in [t-1]$ **do**
3: Sample $\mathbf{p}_k \xleftarrow{\$} \mathbb{F}$ as the random coefficients of the polynomial $\mathbf{p}$.
4: **for** $i \in [n]$ **do**
5: $y_{(i)} \leftarrow \Sigma_{k=0}^{t-1}(\mathbf{p}_k \cdot i^k)$ as the evaluation of $\mathbf{p}(x)$ in $x=i$.
return $(\boldsymbol{y}=\{y_{(i)}\}_{i\in[n]})$ as the n shares.

Combine$_{t,n}(\mathrm{X} \in [n]^t,\ \boldsymbol{y'}=\{y_{(i)} \in \mathbb{F}\}_{i\in \mathrm{X}}) \dashrightarrow s$

1: **for** $i \in \mathrm{X}$ **do** *(Iterate over the indices of the t shares)*
2: Set $\mathrm{X}' \leftarrow \mathrm{X} \setminus \{i\}$
3: $\ell_i \leftarrow \Pi_{k\in \mathrm{X}'} \frac{k}{k-i}$ as the Lagrange coefficient i
4: $s \leftarrow \sum_{i\in \mathrm{X}} \ell_i \cdot y_{(i)}$ for the Lagrange interpolation of $\mathbf{p}(x)$ in $x=0$.
return s as the reconstructed secret.

For convenience, we provide algorithms to convert from additive shares to Shamir shares in Algorithm 3.2, and vice-versa in Algorithm 3.3.

Algorithm 3.2 $\mathsf{AdditiveToShamir}_{\mathsf{t,n},\mathbb{F}}(\mathsf{i}, \mathrm{X}, \mathsf{x_{(i)}}) \leftarrow \mathsf{y_{(i)}}$

Inputs: $i \in [n]$ as the party index
$\mathrm{X} \in [n]^t$ as a subset of t indeces
$x_{(i)} \in \mathbb{F}$ as a n-out-of-n additive share
Outputs: $y_{(i)} \in \mathbb{F}$ as the corresponding t-out-of-n Shamir share
1: Set $\mathrm{X}' \leftarrow \mathrm{X} \setminus \{i\}$
2: $\ell_i \leftarrow \prod_{j\in \mathrm{X}'} \frac{k}{k-i}$ as the Lagrange coefficient
return $y_{(i)} \leftarrow x_{(i)}/\ell_i$ as the Shamir share

Algorithm 3.3 $\mathsf{ShamirToAdditive}_{t,n,\mathbb{F}}(i, \mathrm{X}, y_{(i)}) \leftarrow x_{(i)}$

Inputs: $i \in [n]$ as the party index
$\mathrm{X} \in [n]^t$ as a subset of t indeces
$y_{(i)} \in \mathbb{F}$ as a t-out-of-n shamir share
Outputs: $x_{(i)} \in \mathbb{F}$ as the corresponding t-out-of-t additive share
1: Set $\mathrm{X}' \leftarrow \mathrm{X} \setminus \{i\}$
2: $\ell_i \leftarrow \prod_{j\in \mathrm{X}'} \frac{k}{k-i}$ as the Lagrange coefficient
return $x_{(i)} \leftarrow y_{(i)} \cdot \ell_i$ as the additive share

3.5.1 Authenticated Sharing

In order to detect malicious behavior from parties in an MPC protocol, we use *authenticated sharing* schemes, which extend regular secret sharing schemes with a mechanism to verify that the shares held by each party are consistent and have not been tampered with. There are two main flavors of authenticated sharing, summarized in Table 2:

- *Pairwise authenticated sharing*, as in [BDOZ11], employs additive shares over a field $\mathbb{F}_p$, alongside Message Authentication Codes (MACs) over an extension field $\mathbb{F}_{p^k}$ to ensure that the shares are consistent and have not been tampered with. Each party $\mathcal{P}_i$ holds a share $x_{(i)}$ of the secret x, and for every other party $\mathcal{P}_j$ it holds a MAC $m_{(i,j)} = \alpha_{(j)} \cdot x_{(i)} | \beta_{(j)}$, where α_j is a static MAC key (the same for all shares) and β_j is a dynamic MAC key (one per share), both held by P_j to authenticate the share $x_{(i)}$.
- *Global authenticated sharing*, as in [SPDZ12], uses additive shares over a field $\mathbb{F}_p$, but uses a single global MAC key α held in secret shares by all parties. Each party $\mathcal{P}_i$ holds a share $x_{(i)}$ of the secret x, a share of the MAC $m_{(i)}$ and a share of the MAC key $\alpha_{(i)}$, such that $\sum m_{(i)} = \sum x_{(i)} \cdot \sum \alpha_{(i)}$.

Scheme	**Global ([SPDZ12])**	**Pairwise ([BDOZ11])**
Add. Share of x	$x_{(i)} \in \mathbb{F}_p$ s.t. $\sum x_{(i)} = x$	
MACs	$m_{(i)} \in \mathbb{F}_p$	$m_{(i,j)} \in \mathbb{F}_{p^k}\ \forall j \neq i$
MAC Keys	$\alpha_{(i)} \in \mathbb{F}_p$	$\alpha_{(j)} \in \mathbb{F}_{p^k}\ \forall j \neq i$ (static) $\beta_{(i,j)} \in \mathbb{F}_{p^k}\ \forall j \neq i$
Authentication	$\sum m_{(i)} = \sum \alpha_{(i)} \sum x_{(i)}$	$m_{(i,j)} = \alpha_{(j)} x_{(i)} + \beta_{(j)}\ \forall i, j \neq i$
Auth. Share of x	$(\, x_{(i)},\ m_{(i)},\ \alpha_{(i)}\,)$	$(\, x_{(i)},\ \{m_{(i,j)}, \alpha_{(i,j)}, \beta_{(i,j)}\}_{j \neq i}\,)$
Size per share	$3\lvert\mathbb{F}\rvert$	$\lvert\mathbb{F}\rvert + 2(n-1)\lvert\mathbb{F}_{p^k}\rvert$

Table 2: Flavors of authenticated sharing for party $\mathcal{P}_i$.

To achieve identifiable abort, we use the pairwise authenticated sharing scheme from [BDOZ11]. This scheme has a higher storage and communication overhead than the global authenticated sharing scheme from [SPDZ12], but it allows each party to verify the shares held by every other party individually, which enables identifying malicious parties that provide incorrect shares during the protocol execution. In contrast, the global authenticated sharing scheme only allows to verify the shares collectively, which means that if an inconsistency is detected, it is not possible to identify which party provided incorrect shares, nor to prove that a specific party provided incorrect shares.

3.6 Digital Signatures

A digital signature is an asymmetric cryptographic primitive for verifying the authenticity of digital messages or documents [GMY83, GMR88, BM88, Rom90, GB08]. A valid signature on a message gives a recipient trust in that the message was authored by a sender known to the recipient. Digital signatures are commonly used for financial transactions, software distribution, and to detect forgery or tampering in communication.

Assuming that the private key remains secret, digital signatures are designed to be inherently *resistant to forging*, making it computationally infeasible to generate a valid signature on behalf of a party without knowing that party's private key. They may also provide *non-repudiation*, thus signer cannot successfully claim they did not sign a message. Digitally signed messages may be anything that can be represented as a bit-string, e.g., cryptocurrency transactions, electronic mail, or messages sent as part of cryptographic protocols.

We generalize digital signature schemes in Functionality 3.6:

Functionality 3.6 $\mathcal{F}^{Signature}$

Signature scheme, composed of three algorithms:

- $\mathsf{KeyGen}() \rightarrow (sk, pk)$, generates a private & public key pair (sk, pk)
- $\mathsf{Sign}(sk, m) \rightarrow \sigma$ yields signature σ for message m & private key sk
- $\mathsf{Verify}(pk, m, \sigma) \rightarrow b \in \{0, 1\}$ verifies signature σ for message m and public key pk, either accepting $(b=1)$ or rejecting $(b=0)$ it[7].

The security of digital signature schemes is filled with nuances. In general, the security of a digital signature scheme is defined by a security game between a challenger and an adversary. Based on the capabilities granted to the adversary, we distinguish three basic attacks (more details in [GMY83]):

- *Key-Only Attack (KOA)*: the adversary knows only the public key of the signer, thus it can verify signatures of messages given to him.
- *Known Signature Attack (KSA)*: the adversary is given the public key of the signer and message/signature pairs from a legal signer.
- *Chosen Message Attack (CMA)*: The adversary is also given access to a signing oracle, a black-box that signs any message of the adversary's choice. Out of the three, this is the most powerful adversary and is considered the only realistic notion for many real-world scenarios.

The adversary's goal in the security game may be:

- *Existential Forgery (EF)*: succeed in forging the signature of one message, not necessarily of his choice.

[7] Equivalently, it returns *valid* $(b=1)$ for a valid signature and ABORT otherwise.

- *Selective Forgery (SF)*: succeed in forging the signature of some message of his choice.
- *Universal Forgery (UF)*: succeeds in forging the signature of any message.
- *Total Break*: succeed in computing the signer's secret key.

The signature scheme is then considered secure if the adversary's probability of winning the game for a given goal and attack is negligible. We refer the avid reader to [GB08] for more formal definitions.

There exist many different digital signature schemes, each with their own performance characteristics. In this section, we focus our attention to our chosen signature scheme: Schnorr / EdDSA [JL17]. This scheme is proven secure under the *UF-CMA*, even constituting *zero knowledge proofs of knowledge* of the secret key, the strongest combination out of the classical security notions presented above.

3.6.1 Schnorr

The Schnorr signature scheme [Sch91], known for its simplicity, is an efficient scheme that generates short signatures based on the hardness of the discrete log assumption. In fact, Schnorr is essentially a ZKPoK of the discrete log of the public key, obtained via the Fiat-Shamir transform applied to the classic Schnorr Sigma protocol. The Schnorr scheme provides a benefit over ECDSA [Por13]: it allows for native signature aggregation, enabling batch verification.

We begin by detailing the original Schnorr signature scheme [Sch91], covered in Scheme 3.5. We can distinguish the following steps:

1. *Nonce generation*: the signer generates a random nonce.
2. *Commitment*: the signer commits to the nonce.
3. *Challenge*: the signer creates a challenge via hashing a combination of the public key, the commitment, the message and other parameters.
4. *Signature composition*: the signer crafts the final signature.

Scheme 3.5 Schnorr

The Schnorr signature scheme [Sch91], parametrized by an elliptic curve $\mathrm{E}(\mathbb{G}, q, G, I)$ with identity I, and a hash function H. Following a prior KeyGen, the signer holds a private key $x \in \mathbb{Z}_q^\star$ and a public key $Q = x \cdot G$

Inputs: m, a message to sign

Sign$(m,\ x \in \mathbb{Z}_q^\star,\ Q \in \mathbb{G}) \dashrightarrow \sigma$

1: Sample $k \xleftarrow{\$} \mathbb{Z}_{q^*}$ *(Nonce generation)*
2: $R \leftarrow k \cdot G$ *(Commitment)*
3: $e \leftarrow \mathsf{H}(R \,||\, Q \,||\, m)$ *(Challenge)*
4: $s \leftarrow (x \cdot e) + k$ *(Signature composition)*
return $\sigma = \{e, s\}$ as the signature

Verify$(m, Q \in \mathbb{G},\ \sigma = \{e \in \mathbb{Z}_{q^*}, s \in \mathbb{Z}_{q^*}\}) \dashrightarrow valid$

1: $R \leftarrow s \cdot G + (-e) \cdot Q$
2: $e' \leftarrow \mathsf{H}(R \,||\, Q \,||\, m)$
3: Check if $e' \stackrel{?}{=} e$, otherwise ABORT.
return *valid*

Multiple variants of the Schnorr signature scheme have been proposed over the years, each with its own version of the steps above (e.g., BIP340/BIP341 [WNR18, WNT19], Zilliqa [Zil]). The most common one is EdDSA [JL17]. Used by blockchains like Solana, Monero, Stellar or Nano, EdDSA (Edwards Digital Signature Algorithm) is a variant that uses deterministic nonce generation via hashing. It is defined over a twist of *curve25519*, supporting very simple validation of curve points and scalars thanks to a close-to-power-of-two field order. We detail EdDSA (Scheme 3.6) below.

Scheme 3.6 EdDSA

The EdDSA signature scheme [Sch91] for $ed25519(\mathbb{G}, q, G, I)$ and hash function H (often $\mathsf{Sha512}$). Signer run holds private key $x \in \mathbb{Z}_2^{|q|}$ and public key $Q \in \mathbb{G}$.

Inputs: m, a message to sign

KeyGen$()\dashrightarrow(x, Q)$

1: Sample $a \xleftarrow{\$} \{0,1\}^{2|q|}$
2: $Q \leftarrow a \cdot G$
3: $x \leftarrow \{a_{(i+|q|)}\}_{i \in [|q|]}$
return (x, Q) as the key pair

Sign$(m,\ x \in \mathbb{Z}_q^\star,\ Q \in \mathbb{G})\dashrightarrow\sigma$

4: $k \leftarrow \mathsf{H}(x \mid\mid m)$ *(Nonce generation)*
5: $R \leftarrow k \cdot G$ *(Commitment)*
6: $e \leftarrow \mathsf{H}(R \mid\mid Q \mid\mid \mathsf{H}(m))$ *(Challenge)*
7: $s \leftarrow (x \cdot e) + k$ *(Signature composition)*
return $\sigma = \{R, s\}$ as the signature

Verify$(m, Q \in \mathbb{G},\ \sigma = \{R \in \mathbb{G}, s \in \mathbb{Z}_{q^*}\})\dashrightarrow valid$

1: $e' \leftarrow \mathsf{H}(G \mid\mid R \mid\mid Q \mid\mid \mathsf{H}(m))$
2: $R' \leftarrow s \cdot G + (-e) \cdot Q$
3: Check if $R' \stackrel{?}{=} R$, otherwise ABORT.
return *valid*

3.7 Transcript

A transcript is a data structure used to hold the public history of a protocol execution, that is, all the messages publicly exchanged among parties. This shared record of interactions is used mainly for two purposes:

1. To glue together the different phases of our protocols by making subsequent phases depend on the transcript of the previous ones, e.g., to salt/customize the hash function by employing the a succinct representation of transcript (its *state*) as an input. The transcript *state* then acts as a Common Reference String (CRS).

2. To perform the Fiat-Shamir transformation, by replacing the random challenges with a hash of the transcript (e.g., in the Schnorr ZKPoK, but more of this on Section 3.8).

In a nutshell, the Fiat-Shamir heuristic provides a way to transform a (public-coin) interactive proof into a non-interactive proof. Intuitively, the idea is to replace a verifier's random challenges with a hash of the prover's prior messages. In general, cryptographic protocols are designed and analyzed in the interactive setting, whereas their implementation is often sought to be non-interactive. However there are many details to be specified: what exactly the prover's messages are, how they are formatted, how to ensure the encoding is unambiguous, how to handle domain separators, how to link challenges between

rounds (in multi-round protocols), or how to expand challenges (in the case that a protocol requires more challenge bytes than the digest size), etc. These details open space for security vulnerabilities, such as in the Helios protocol used for IACR elections [BPW12], and in the SwissPost/Scytl e-voting system [HLPT20].

We present a Transcript structure[8] to specify all these details (Scheme 3.7). Our construction relies on a hash function H to chain inputs[9] into an internal state s, and use this state as seed for a PRNG to extract randomness when needed. In this line, the two functions provided by this structure are:

- $\mathsf{Append}(s, l, m)$: embeds a message m with a label l (a.k.a. domain separation tag) into the transcript state s via hash chaining.
- $\mathsf{Extract}(s, l, k)$: extracts k bits of randomness by seeding a PRNG with the transcript state s, using a label l as domain separation tag.

Scheme 3.7 Transcript (T)

A transcript abstraction holding a state $s \in \mathbb{Z}_2^{2\lambda}$, a condensed description of the messages exchanged in a protocol execution. It is parametrized by a hash function H (we employ Blake3 [ONAZ]), and a PRG (we use `ChaCha12`, a variant of `ChaCha20` [B+08]). The state s is initialized to zero and updated upon each Append / Extract.

T.Append$_l$$(m)$ $\dashrightarrow$
1: $\mathsf{T}.s \leftarrow \mathsf{H}(\mathsf{H}(\mathsf{T}.s \,||\, l) \,||\, m)$ with message m and label l to update s

T.Extract$_l$$(k \in \mathbb{N}) \dashrightarrow r \in \mathbb{Z}_2^k$
2: $\mathsf{T}.s \leftarrow \mathsf{H}(\mathsf{T}.s \,||\, l)$ with label l to update s
3: Set $\mathsf{PRG.Seed}(\mathsf{T}.s)$ and compute $r \leftarrow \mathsf{PRG.Get}(k)$ to get the randomness.
return r

Throughout our protocols, we append all the exchanged messages among parties[10], we set fixed formats for all the messages being appended to the transcript with unambiguous encoding, we hardcode a unique label for each message to act as domain separator tags, we pass a unified transcript object to link challenges between rounds, and we use the Extract function to expand challenges when needed. In addition to its use for Fiat-Shamir transformations, we employ it to glue protocols together with a common source of randomness. We obviate the choices of labels l across all our protocols for simplicity, and we mark the moment a message is appended to the transcript with a subscript $_\bullet$ either on

[8] We discard the use of Merlin Transcripts [dV] based on *strobe* [Ham17], as they specifically warn that their use is not recommended for production-level environments.

[9] In line with the optimisation described in [CY24, Chapter 15], we perform hash chaining instead of message concatenation to avoid a quadratic blow-up in the inputs to the hash function when applying the Fiat-Shamir transform.

[10] We thus follow the *strong* Fiat-Shamir strategy when transcribing messages (a.k.a. append all the messages). We purposely discard the *weak* Fiat-Shamir strategy where you transcribe only the commitments of PoK, as it often leads to unintended pitfalls.

the message (e.g., $(m)_{\cdot}$) or on the assign/sample operators (e.g., $m \xleftarrow{} 42$ or $m \xleftarrow{\$} \mathbb{Z}_q$).

3.8 Proofs of Knowledge

In cryptography, a Proof of Knowledge (PoK) is a protocol in which a prover can convince a verifier that it knows something [KL07]. More formally, a PoK is defined for a certain relation $R : \{(x, w) : x \in L, w \in W(x)\}$ composed of a statement x (e.g., I know a value whose hash is a certain known digest d) from a language[11] L and all the witnesses w that satisfy that statement $W(x)$ (e.g., the values m such that $\mathsf{H}(m) = d$). For such a relation, and with knowledge error κ, a PoK is a two party protocol between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ with the following two properties:

- *Completeness*: if the statement is true and thus $(x, w) \in R$, then a prover $\mathcal{P}$ who knows a witness w for x succeeds in convincing the verifier $\mathcal{V}$ of his knowledge with probability 1.
- *Soundness*: if the statement is false and thus $\mathcal{P}$ doesn't know a witness w for x, the probability that any prover $\mathcal{P}$ can convince the verifier $\mathcal{V}$ of his knowledge of x is at most κ.

Additionally, a PoK is *zero-knowledge* (ZKPoK) if the verifier $\mathcal{V}$ learns nothing about the witness w other than the fact that it satisfies the relation R[12]. The proofs presented in this section are all ZK. We require these proofs to protect against active adversaries in our protocols and abort upon detection of malicious behavior.

We are mainly interested on public-coin Interactive Proofs (IP) where, after a protocol setup $\mathsf{IP.Setup}$, the verifier $\mathcal{V}$ sends in one or multiple rounds some random challenges $e_i \leftarrow \mathsf{IP.Challenge}$ (the public coins) to the prover $\pi_i \leftarrow \mathsf{IP.Prove}$, who then responds with a proof π_i. The verifier $\mathcal{V}$ can then verify each proof $\mathsf{IP.Verify}(\pi_i, x, e_i)$ using the statement x and its challenge e_i.

3.8.1 Sigma Protocols and Non-Interactivity

A Sigma Protocol $\mathbf{\Sigma}$ is a four-move interactive PoK protocol run between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ with three distinct communication rounds:

1. $\mathsf{Commitment}$: $\mathcal{P}$ gets $A \leftarrow \mathsf{Commit}(w)$ of statement w and sends it to $\mathcal{V}$.
2. $\mathsf{Challenge}$: $\mathcal{V}$ computes and sends a challenge e to $\mathcal{P}$.
3. $\mathsf{Response}$: $\mathcal{P}$ computes and sends a response z to $\mathcal{V}$.

[11] That is, a set of statements with some common properties. $L \in NP$ for our purposes, making it computationally unfeasible to craft statements without appropriate witnesses.

[12] This property is formalized by the existence of a simulator $\mathcal{S}$ named the *knowledge extractor* that can produce a transcript of the protocol between $\mathcal{P}$ and $\mathcal{V}$ without knowing any witness w for x.

4. Verify: $\mathcal{V}$ checks if the response is *valid*, rejecting the proof otherwise.

Σ-protocols require heavy interaction, often employing more rounds of communication than the protocols in which they are used. To transform any interactive Sigma Protocol into a non-interactive proof, we can apply one of two well-studied transformations:

Scheme 3.8 Fiat-Shamir(FS)

A transformation from a public-coin Interactive-Proof (IP) IP_{R} into a Non-Interactive proof system following [FS86], parametrized by a hash function H.

Players: A prover $\mathcal{P}$ and a verifier $\mathcal{V}$.

$\mathcal{P}.\mathsf{Prove}(x, w) \dashrightarrow \pi$

1: Run all steps of the prover in $\pi \leftarrow \mathsf{IP}_{\mathrm{R}}.\mathsf{Prove}(x, w)$, replacing[13] the verifier's challenges with $e_i \leftarrow \mathsf{H}(x \mid\mid m_0 \ldots \mid\mid m_i)$ where m_i is $\mathcal{P}$ message at step i. **return** π

$\mathcal{V}.\mathsf{Verify}(x, \pi) \dashrightarrow \mathit{valid}$

1: Run all steps of the verifier in $\mathsf{IP}_{\mathrm{R}}.\mathsf{Verify}(x, \pi)$, replacing the verifier's challenges with $e_i \leftarrow \mathsf{H}(x \mid\mid m_0 \ldots \mid\mid m_i)$ where m_i is $\mathcal{P}$ message at step i.

- **Fiat-Shamir Heuristic** [FS86]: replaces the challenge e with a hash of the commitment A and the transcript of the previous rounds. This popular non-interactive transformation is used throughout our suite of protocols, such as the Schnorr signing scheme described in Scheme 3.5 or the consistency check as part of the Oblivious Transfer Extensions. Despite its simplicity, the Fiat-Shamir heuristic is not universally composable (UC) secure, as it requires forking to extract the simulator's view. This limitation is addressed by the Fischlin transform, which we describe next. A more recent take on the Fiat-Shamir heuristic can be consulted in [CY24].
- **Fischlin Transform** [Fis05]: replaces the challenge e with a set of hashes of the commitment A with random values r sampled by the prover $\mathcal{P}$. Contrary to Fiat-Shamir, the Fischlin transform is straight-line extractable without forking, allowing it to be proven UC secure. Despite this, a known limitation of the Fischlin transform is that it applies to a limited class of Sigma Protocols with a "quasi-unique response" property, which doesn't necessarily permit standard compositions for Sigma protocols (e.g. one proof OR another proof). A randomized variant of Fischlin proposed by Kondi-Shelat [KS22, Section 6.4] removes this limitation, and thus we select it for our compiler.

We employ Fiat-Shamir transform in the context of OTs, VOLEs and Singlets, allowing us to replace the otherwise interactive challenges with non-interactive ones. We leave the adaptation of the Fischlin transcript for future

[13] In practice we employ a transcript T (Section 3.7), replacing message concatenation with $\mathsf{T}.\mathsf{Append}$ and the final hashing with the hash-chaining of $\mathsf{T}.\mathsf{Extract}$.

work, as it is more computationally intensive and certain protocols do not need it to be UC secure.

Composition. Proofs of Knowledge (PoK) can be composed in the following ways:

AND($\mathbf{\Sigma}_1, \mathbf{\Sigma}_2$): Straightforwardly achieved by running both proofs in parallel.

OR($\mathbf{\Sigma}_1, \mathbf{\Sigma}_2$): Following the composition technique from [Dam02, CPS+16], it requires simulating the proof that is not used by the prover $\mathcal{P}$ while running the proof that is used in a standard fashion.

This composition forces the challenge generation in our proofs to be defined over generic bit-strings in order to generalize over different kinds of challenge spaces (i.e., $\{0,1\}^n$ or $\mathbb{Z}_q$). Additionally, all proofs must provide a *simulator* that can generate a valid transcript without knowing the secret a of $\mathcal{P}$.

3.8.2 PoK of the Discrete Log of a number

The interactive Schnorr protocol [Sch91] for proving knowledge of the discrete logarithm of a group element is a textbook example of ZKPoK, while also being a prime example of Sigma Protocols. This proof system, realizing $\mathcal{F}^{R_{DL}}$ with relationship $\mathsf{R}_{\mathsf{DL}} = \{(G, X, w) : X = G \cdot w\}$ reads as follows: given a group element X and a secret w, the prover $\mathcal{P}$ must convince the verifier $\mathcal{V}$ that it knows the discrete logarithm of X with respect to the base point G, i.e., $X = w \cdot G$. This proof system Σ_{DL} works as follows:

$$\begin{aligned} \text{Commitment:} &\quad a \xleftarrow{\$} \mathbb{Z}_q \quad A \leftarrow G \cdot a \\ \text{Challenge:} &\quad e \xleftarrow{\$} \mathbb{Z}_q \\ \text{Response:} &\quad z \leftarrow a + e \cdot w \\ \text{Verify:} &\quad X \stackrel{?}{=} G \cdot z - A \cdot e \end{aligned} \tag{2}$$

Batching Schnorr Proofs. There are instances where we need to prove the knowledge of the discrete logarithms of n group elements at once. To this end, we employ the batching technique described in [GLSY04] to aggregate multiple Schnorr proofs into a single proof. In short, given the relation $\mathsf{R}_{\mathsf{BatchDL}}(\mathbb{G}, G) = \left\{ \{X_i, w_i\}_{i \in [n]} : X_i = G \cdot w_i \;\; \forall i \in [n] \right\}$ the sigma protocol $\mathbf{\Sigma}_{\mathsf{BatchDL}}$ that constructs such proofs consists of:

$$\begin{aligned} \text{Commitment:} &\quad a \xleftarrow{\$} \mathbb{Z}_q \quad A \leftarrow G \cdot a \\ \text{Challenge:} &\quad e_i \xleftarrow{\$} \mathbb{Z}_q \; \forall i \in [n] \\ \text{Response:} &\quad z \leftarrow a + \cdot \Sigma(e_i \cdot w_i) \\ \text{Verify:} &\quad G \cdot z \stackrel{?}{=} = A + \Sigma(X_i \cdot e_i) \end{aligned} \tag{3}$$

3.8.3 PoK of Discrete Log Equality

On $\mathsf{R}_{\mathsf{DLEQ}}(\mathbb{G}, G_1, G_2) = \{X_1, X_2, w_1, w_2 : X_1 = w_1 G_1 \wedge X_2 = w_2 G_2 \wedge w_1 = w_2\}$, the relation of equality $w_1 = w_2$ of the discrete log of two group elements $X_1 = w_1 \cdot G_1$ and $X_2 = w_2 \cdot G_2$ with respect to two base points G_1, G_2, we can prove it via the sigma protocol Σ_{DLEQ} from Chaum-Pedersen [CP92]:

$$\begin{aligned} \text{Commitment:} \quad & s \xleftarrow{\$} \mathbb{Z}_q, \quad A_1 \leftarrow G_1 \cdot s, \quad A_2 \leftarrow G_2 \cdot s \\ \text{Challenge:} \quad & e \xleftarrow{\$} \mathbb{Z}_q, \\ \text{Response:} \quad & z \leftarrow s + e \cdot w \\ \text{Verify:} \quad & G_1 \cdot z \stackrel{?}{=} A_1 + X_1 \cdot e \quad \text{and} \quad G_2 \cdot z \stackrel{?}{=} A_2 + X_2 \cdot e \end{aligned} \tag{4}$$

Much like the Schnorr protocol, we use our compiler to straightforwardly transform the Sigma Protocol Σ_{DLEQ} into a non-interactive proof. We omit the resulting non interactive proof for brevity.

3.9 Learning Parity with Noise (LPN) and Variants

Another cornerstone hardness assumption in modern cryptography is the *Learning Parity with Noise* (LPN) problem. Unlike factoring or discrete logarithms, which are number-theoretic in nature, LPN belongs to the family of learning problems in computational learning theory. Its importance stems from the fact that it is conjectured to be hard even for quantum computers, making it a candidate for post-quantum cryptography.

The LPN Problem. Let $n \in \mathbb{N}$ be the dimension, and let $s \in \{0,1\}^n$ be a secret vector chosen uniformly at random. Consider a distribution of samples (a, b) where $a \in \{0,1\}^n$ is chosen uniformly, and $b = \langle a, s \rangle \oplus e$. Here $\langle a, s \rangle$ denotes the inner product modulo 2, and $e \in \{0,1\}$ is an independent noise bit that equals 1 with probability τ (the *noise rate*) and 0 otherwise.

Given access to polynomially many samples (a, b) from the distribution above, the LPN problem is to recover the secret vector s. The assumption is that no polynomial-time algorithm can solve LPN with non-negligible advantage when $\tau > 0$ is constant.

Intuitively, LPN asks us to solve a system of noisy linear equations over $\mathbb{F}_2$. Without noise ($\tau = 0$), Gaussian elimination solves the system efficiently. With noise, however, the problem becomes much harder, since standard linear algebra breaks down. Unlike factoring or discrete logarithms, LPN is not known to be solvable efficiently by quantum algorithms, making it a candidate foundation for post-quantum cryptography.

Variants of LPN. Several structured versions of LPN have been proposed, motivated by efficiency or functionality:

- **Ring-LPN.** In the Ring-LPN problem, the inner product is replaced by multiplication in a finite ring, followed by addition of noise. The ring

structure allows more compact key sizes and faster operations, while preserving conjectured hardness.

- **Sparse-LPN.** In sparse variants, the vectors a are drawn from restricted distributions (e.g., sparse Hamming weight). Such variants arise in practical lightweight protocols.
- **Connection to LWE.** Learning With Errors (LWE) generalizes LPN from the field $\mathbb{F}_2$ to larger finite fields and lattices. LWE is reducible from worst-case lattice problems, giving it stronger worst-case hardness guarantees. LPN can thus be seen as the "binary" cousin of LWE, with simpler structure and efficient applications.

Hardness of LPN. While LPN does not (yet) enjoy worst-case hardness guarantees like LWE, it is supported by decades of cryptanalytic evidence. The best known algorithms [BKW03, AG11] run in time super-polynomial or exponential in the dimension n, depending on the noise rate τ. Thus, LPN with constant noise is widely considered intractable for polynomial-time adversaries.

Application to our Protocols. LPN based Pseudorandom Correlation Generators (PCGs) are a powerful tool for efficient MPC preprocessing. They allow parties to generate correlated randomness (e.g., multiplication triples, oblivious transfers) with minimal communication and computation. The LPN assumption underpins the security of these PCGs, ensuring that an adversary cannot distinguish the generated correlations from random values. This applies to all pairwise protocols: OTs, OLEs and VOLEs.

4 Preprocessing Protocols

Our MPC framework relies heavily on the preprocessing model, where parties generate correlated randomness in an *offline* phase in preparation the actual computation. This preprocessing enables efficient and secure evaluation of arithmetic circuits in the online phase. We implement a suite of preprocessing protocols, both for two-party and n-party settings, to generate the necessary correlated randomness to support a wide range of operations in our online phase.

4.1 Two Party Protocols

4.1.1 Oblivious Transfer

An Oblivious Transfer (OT) functionality consists of a two-party interaction where a sender transfers several messages to a receiver, and the receiver chooses a subset of these messages and receives them. Both parties remain oblivious to the other party's actions: the sender is unaware of messages the receiver chose, and the receiver does not learn the content of the messages he didn't choose.

There are multiple flavors of OTs according to the number of messages, choices and the relationship established among the inputs & outputs. For our OT functionalities, we stick to senders sending two messages and receivers receiving one message. These functionalities are:

- *Standard*: In the standard $\binom{2}{1}$-OT described in Functionality 4.7, the sender inputs κ-bit messages, and the receiver inputs a choice bit. The receiver then receives the message corresponding to his choice bit.

Functionality 4.7 $\mathcal{F}^{OT}{}_{M}(\boldsymbol{m_0}, \boldsymbol{m_1}, \boldsymbol{b}) \longrightarrow (\boldsymbol{m_0}, \boldsymbol{m_1}, \boldsymbol{m_b})$

One-out-of-two Oblivious Transfer between sender $\mathcal{S}$ and receiver $\mathcal{R}$ for M instances.

Players: A sender $\mathcal{S}$, and a receiver $\mathcal{R}$.

Inputs: $\mathcal{S} \leftarrow (\boldsymbol{m_0}, \boldsymbol{m_1}) \in \mathbb{Z}_2^{2\times M\times\kappa}$, two vectors of κ-bit messages.
$\mathcal{R} \leftarrow \boldsymbol{b} \in \mathbb{Z}_2^M$, a vector of choice bits.

Outputs: $\mathcal{S} \leftarrow (\boldsymbol{m_0}, \boldsymbol{m_1})$, the sender retains both message vectors.
$\mathcal{R} \leftarrow \boldsymbol{m_b} \in \mathbb{Z}_2^{M\times\kappa}$, chosen messages s.t. $m_{b(i)} = m_{1(i)} \cdot b_{(i)} + m_{0(i)} \cdot (1 - b_{(i)})$ for all $i \in [M]$.

$\mathcal{F}^{OT}.\textsf{Transfer}(\boldsymbol{m_0}, \boldsymbol{m_1}, \boldsymbol{b}) \dashrightarrow \boldsymbol{m_b}$
Receive $(\boldsymbol{m_0}, \boldsymbol{m_1})$ from $\mathcal{S}$ and $\boldsymbol{b}$ from $\mathcal{R}$, send $m_{b(i)} = m_{1(i)} \cdot b_{(i)} + m_{0(i)} \cdot (1 - b_{(i)})$ to $\mathcal{R}$ for all $i \in [M]$.

- *Random*: In the random $\binom{2}{1}$-ROT described in Functionality 4.8, the sender doesn't input any messages. Instead, his messages are randomly sampled by the functionality. The relationship between the inputs and outputs is the same as in the standard OT.

Functionality 4.8 $\mathcal{F}^{ROT}{}_{M} \longrightarrow (\boldsymbol{m_0}, \boldsymbol{m_1}, \boldsymbol{b}, \boldsymbol{m_b})$

One-out-of-two Random Oblivious Transfer between sender $\mathcal{S}$ and receiver $\mathcal{R}$ for M instances.

Players: A sender $\mathcal{S}$, and a receiver $\mathcal{R}$.
Inputs: None (all values are randomly sampled).
Outputs: $\mathcal{S} \leftarrow (\boldsymbol{m_0}, \boldsymbol{m_1}) \in \mathbb{Z}_2^{2\times M\times\kappa}$, two random message vectors.
$\mathcal{R} \leftarrow (\boldsymbol{b} \in \mathbb{Z}_2^M, \boldsymbol{m_b} \in \mathbb{Z}_2^{M\times\kappa})$, random choice bits and chosen messages s.t. $m_{b(i)} = m_{1(i)} \cdot b_{(i)} + m_{0(i)} \cdot (1 - b_{(i)})$ for all $i \in [M]$.

$\mathcal{F}^{ROT}.\textsf{SampleMessages}() \dashrightarrow \boldsymbol{m_0}, \boldsymbol{m_1}$
Sample and send $(\boldsymbol{m_0}, \boldsymbol{m_1}) \xleftarrow{\$} \mathbb{Z}_2^{2\times M\times\kappa}$ to $\mathcal{S}$.
$\mathcal{F}^{ROT}.\textsf{SampleChoices}() \dashrightarrow \boldsymbol{b}, \boldsymbol{m_b}$
Sample $\boldsymbol{b} \xleftarrow{\$} \mathbb{Z}_2^M$, compute $m_{b(i)} = m_{1(i)} \cdot b_{(i)} + m_{0(i)} \cdot (1 - b_{(i)})$ for all $i \in [M]$, send $(\boldsymbol{b}, \boldsymbol{m_b})$ to $\mathcal{R}$.

- *Correlated*: In the correlated $\binom{2}{1}$-COT described in Functionality 4.9, a random OT is modified to enforce a mathematical relationship (over a group operation for group $\mathbb{G}$) between the inputs and outputs of the protocol, such that $z_a + z_B = a \cdot x$.

Functionality 4.9 $\mathcal{F}^{COT}{}_{M}(\boldsymbol{u}) \longrightarrow (\boldsymbol{v}, \boldsymbol{u}, \boldsymbol{b}, \boldsymbol{w})$

One-out-of-two Correlated Oblivious Transfer between sender $\mathcal{S}$ and receiver $\mathcal{R}$ for M instances with individual correlations.

Players: A sender $\mathcal{S}$, and a receiver $\mathcal{R}$.
Inputs: $\mathcal{S} \leftarrow \boldsymbol{u} \in \mathbb{F}^M$, a vector of correlation values.
Outputs: $\mathcal{S} \leftarrow (\boldsymbol{v}, \boldsymbol{u}) \in \mathbb{F}^{2\times M}$, random base values and correlations.
$\mathcal{R} \leftarrow (\boldsymbol{b} \in \mathbb{Z}_2^M, \boldsymbol{w} \in \mathbb{F}^M)$, choice bits and messages s.t. $w_{(i)} = u_{(i)} \cdot b_{(i)} + v_{(i)}$ for all $i \in [M]$.

$\mathcal{F}^{COT}.\textsf{Sample}() \dashrightarrow \boldsymbol{v}, \boldsymbol{b}$
Sample $\boldsymbol{v} \xleftarrow{\$} \mathbb{F}^M$ and send to $\mathcal{S}$, sample $\boldsymbol{b} \xleftarrow{\$} \mathbb{Z}_2^M$ and send to $\mathcal{R}$.
$\mathcal{F}^{COT}.\textsf{Correlate}(\boldsymbol{u}) \dashrightarrow \boldsymbol{w}$
Receive $\boldsymbol{u}$ from $\mathcal{S}$, compute $w_{(i)} = u_{(i)} \cdot b_{(i)} + v_{(i)}$ for all $i \in [M]$, send $\boldsymbol{w}$ to $\mathcal{R}$.

These functionalities are tightly related. In fact, both the standard OT and the Correlated OTs are commonly constructed from a Random OT [MR19]. To achieve standard OT, the sender uses the ROT output messages to one-time-pad encrypt (XOR) his two input messages, and the receiver can only decrypt one of the two encrypted messages. Similarly, a Correlated OT sender maps the ROT output messages into a group $\mathbb{G}$ and uses them to create and send a correlating

mask, and the receiver can apply this mask to his ROT output to establish the desired correlation. Based on the techniques from [MR19], we specify the $\mathsf{ROT} \rightarrowtail \mathsf{OT}$ composition in Protocol 4.2, and the $\mathsf{ROT} \rightarrowtail \mathsf{COT}$ composition in Protocol 4.3. Consequently, the remaining protocols in this section focus only on the ROT functionality. All protocols are generalized to run a batch of ξ OTs in parallel (with ξ choice bits), and all ROT/OT/COT messages are composed of ℓ elements, where each element is either a κ-bit string (ROT/OT) or an element of a group $\mathbb{G}$ (COT).

Protocol 4.2 $\mathsf{OT}_{\xi,\ell}$

(Standard) OT protocol from any ROT protocol and one-time-pad encryption

Players: a sender $\mathcal{S}$, and receiver $\mathcal{R}$

Inputs: $\mathcal{R}\colon \boldsymbol{x} \in \mathbb{Z}_2^{\xi}$, the input choice bits

$\mathcal{S}\colon \boldsymbol{m_0}, \boldsymbol{m_1} \in \mathbb{Z}_2^{\xi\times\ell\times\kappa}$, pairs of messages

Outputs: $\mathcal{R} \leftarrow \boldsymbol{m_x} \in \mathbb{Z}_2^{\xi\times\ell\times\kappa}$, the chosen messages

$\mathcal{S}\&\mathcal{R}.\mathbf{ROT}(b) \dashrightarrow \mathcal{S}\colon (\boldsymbol{r_0}, \boldsymbol{r_1});\ \mathcal{R}\colon \boldsymbol{r_b}$

1: $\mathcal{S}$ runs $\mathsf{ROT}_{\xi,\ell}$ as sender, obtaining $\boldsymbol{r_0}, \boldsymbol{r_1} \in \mathbb{Z}_2^{\xi\times\ell\times\kappa}$

2: $\mathcal{R}$ runs $\mathsf{ROT}_{\xi,\ell}$ as receiver, obtaining $\boldsymbol{r_b} \in \mathbb{Z}_2^{\xi\times\ell\times\kappa}$

$\mathcal{S}.\mathbf{Encrypt}(\boldsymbol{r_0}, \boldsymbol{r_1}, \boldsymbol{m_0}, \boldsymbol{m_1}) \dashrightarrow (\boldsymbol{\tau_0}, \boldsymbol{\tau_1})$

1: $(\boldsymbol{\tau_0}, \boldsymbol{\tau_1}) \leftarrow \left\{\left\{r_{0(i,j)} \oplus m_{0(i,j)}\right\}_{j\in[\ell\times\kappa]}\right\}_{i\in[\xi]}$

2: $\mathsf{Send}(\boldsymbol{\tau_0}, \boldsymbol{\tau_1}) \rightarrow \mathcal{R}$

$\mathcal{R}.\mathbf{Decrypt}(\boldsymbol{\tau_0}, \boldsymbol{\tau_1}) \dashrightarrow \boldsymbol{m_x}$

$$\textbf{return } \boldsymbol{m_x} \leftarrow \left\{ \left\{ \begin{cases} \tau_{0(i,j)} \oplus r_{b(i,j)} & \text{if } x_{(i)} = 0 \\ \tau_{1(i,j)} \oplus r_{b(i,j)} & \text{if } x_{(i)} = 1 \end{cases} \right\}_{j\in[\ell\times\kappa]} \right\}_{i\in[\xi]}$$

Protocol 4.3 $\mathsf{COT}_{\mathbb{G},\xi,\ell}$

Correlated OT protocol from any ROT protocol and a uniform mapper $\mathsf{H}\colon \mathbb{Z}_2^\kappa \mapsto \mathbb{G}$ (e.g., Hash2Field [FHSS+23] for EC), parametrized by a group $\mathbb{G}$

Players: a sender $\mathcal{S}$, and receiver $\mathcal{R}$
Inputs: $\mathcal{S}\colon \boldsymbol{a} \in \mathbb{G}^{\xi\times\ell}$, group elements
$\mathcal{R}\colon \boldsymbol{x} \in \mathbb{Z}_2^\xi$, the choice bits
Outputs: $\mathcal{S} \leftarrow \boldsymbol{z_A} \in \mathbb{G}^{\xi\times\ell}$
$\mathcal{R} \leftarrow \boldsymbol{z_B} \in \mathbb{G}^{\xi\times\ell}$ s.t. $z_{A(i,j)} + z_{B(i,j)} = a_{(i,j)} \cdot x_{(i)} \quad \forall i\in[\xi]\ \forall j\in[\ell]$

$\mathcal{S}\&\mathcal{R}.\textbf{RunROT}(b) \dashrightarrow \mathcal{S}\colon (\boldsymbol{r_0}, \boldsymbol{r_1});\ \mathcal{R}\colon \boldsymbol{r_b}$
1: $\mathcal{S}$ runs $\mathsf{ROT}_{\xi,\ell}$ as sender, obtaining $\boldsymbol{r_0}, \boldsymbol{r_1} \in \mathbb{Z}_2^{\xi\times\ell\times\kappa}$
2: $\mathcal{R}$ runs $\mathsf{ROT}_{\xi,\ell}$ as receiver, obtaining $\boldsymbol{r_b} \in \mathbb{Z}_2^{\xi\times\ell\times\kappa}$

$\mathcal{S}.\textbf{CreateCorrelation}(\boldsymbol{r_0}, \boldsymbol{r_1}, \boldsymbol{a}) \dashrightarrow (\boldsymbol{\tau})$
1: $\boldsymbol{z_A} \leftarrow \{\{\mathsf{H}(r_{0(i,j)})\}_{j\in[\ell]}\}_{i\in[\xi]}$
2: $\boldsymbol{\tau} \leftarrow \{\{\mathsf{H}(r_{1(i,j)}) - z_{A(i,j)} + a_{(i,j)}\}_{j\in[\ell]}\}_{i\in[\xi]}$
3: $\mathsf{Send}(\boldsymbol{\tau}) \to \mathcal{R}$
return $\boldsymbol{z_A}$

$\mathcal{R}.\textbf{ApplyCorrelation}(\boldsymbol{\tau}) \dashrightarrow \boldsymbol{z_B}$
return $\boldsymbol{z_B} \leftarrow \{\{\tau_{(i,j)} \cdot b_{(i)} - \mathsf{H}(r_{b(i,j)})\}_{j\in[\ell]}\}_{i\in[\xi]}$

We also define the vectorized Correlated OT functionality (VCOT) for reference:

Functionality 4.10 $\mathcal{F}^{VCOT}{}_M(\Delta) \longrightarrow (\boldsymbol{v}, \Delta, \boldsymbol{b}, \boldsymbol{w})$

One-out-of-two Vector Correlated Oblivious Transfer between sender $\mathcal{S}$ and receiver $\mathcal{R}$ for M instances with a single global correlation Δ.

Players: A sender $\mathcal{S}$, and a receiver $\mathcal{R}$.
Inputs: $\mathcal{S} \leftarrow \Delta \in \mathbb{F}$, a global correlation value.
Outputs: $\mathcal{S} \leftarrow (\boldsymbol{v}, \Delta) \in \mathbb{F}^{M+1}$, random base values and the correlation.
$\mathcal{R} \leftarrow (\boldsymbol{b} \in \mathbb{Z}_2^M, \boldsymbol{w} \in \mathbb{F}^M)$, choice bits and messages s.t. $w_{(i)} = \Delta \cdot b_{(i)} + v_{(i)}$ for all $i\in[M]$.

$\mathcal{F}^{VCOT}.\textbf{Sample}() \dashrightarrow \boldsymbol{v}, \boldsymbol{b}$
Sample $\boldsymbol{v} \xleftarrow{\$} \mathbb{F}^M$ and send to $\mathcal{S}$, sample $\boldsymbol{b} \xleftarrow{\$} \mathbb{Z}_2^M$ and send to $\mathcal{R}$.

$\mathcal{F}^{VCOT}.\textbf{Correlate}(\Delta) \dashrightarrow \boldsymbol{w}$
Receive Δ from $\mathcal{S}$, compute $w_{(i)} = \Delta \cdot b_{(i)} + v_{(i)}$ for all $i\in[M]$, send $\boldsymbol{w}$ to $\mathcal{R}$.

OT protocols can realize these functionalities with or without relying on preprocessing generated in a a setup phase, yielding two classes of OT protocols:

- *Base OTs* do not rely on a setup phase, yet they require public-key cryptography and thus incur in higher computation costs, higher per-bit com-

munication costs and more rounds to achieve. As such, they are used mostly as a building block in the one-time setup of OT extensions.

- *OT Extensions* rely on the prior execution of a few Base OTs in a setup phase, and *extend* each Base OT by seeding a PRNG with the Base OT outputs and generating a pseudo-random sequence that holds the OT relationship. Semi-honest security is achieved cheaply without communication, whereas malicious security requires some communication.

We implement multiple OT protocols in our suite, including base OTs and various OT extension protocols with different security and efficiency tradeoffs.

Base ROT Protocols. We implement two Base $\binom{2}{1}$-ROT protocols:

Batched Base OT. We implement Figure 3 of [MRR21a] as a Base OT protocol achieving endemic security[14]. We detail this three-round protocol in Protocol 4.4. The notation for our protocol follows closely that of an analogous description of a $\binom{2}{1}$-ROT from [MRR21a, Figure 3]. To realize BBOT we select:

- the Key Agreement (KA) protocol from [MRR21a, Figure 8], employing Hash2Curve (RFC9380 [FHSS+23]) as a random oracle that gives outputs in the curve.
- the Programmable-Once Pseudo-random Function (POPF) from [MRR21a, Figure 6], making use of two random oracles H_0 and H_1.

[14] That is, while a maliciously corrupted party may influence his output, the resulting protocol outputs preserve the desired relationship (selection / correlation). More info in [MR19].

Protocol 4.4 $\mathsf{BBOT}_{\xi,\ell,\mathbb{G}}$

Endemically secure Batched Base ROT protocol from [MRR21a, Figure 3] and a hash function H (for two ROs $\mathsf{H}_0(x)$ and $\mathsf{H}_1(x)$), parametrized by a batch of ξ messages with $\ell \times \kappa$ bits each, a group $\mathbb{G}$ of prime order q with generator G.

Players: a sender $\mathcal{S}$, and receiver $\mathcal{R}$.
Inputs: $\mathcal{R}$: $\boldsymbol{b} \in \mathbb{Z}_2^{\xi}$, the input choice bits.
Outputs: $\mathcal{S} \leftarrow \boldsymbol{m_0}, \boldsymbol{m_1} \in \mathbb{Z}_2^{\xi \times \ell \times \kappa}$, two random messages.
$\mathcal{R} \leftarrow \boldsymbol{m_b} \in \mathbb{Z}_2^{\xi \times \ell \times \kappa}$, the chosen message.

$\mathcal{S}$.**Round1**()$\dashrightarrow(A)$
1: Sample $a \xleftarrow{\$} \mathbb{Z}_q$ $(KA.R)$
2: $A \leftarrow a \cdot G$ $(KA.msg_1)$
3: $\mathsf{Send}(A) \rightarrow \mathcal{R}$

$\mathcal{R}$.**Round2**$(A \in \mathbb{G}, \boldsymbol{b}) \dashrightarrow (\boldsymbol{m_x}, \boldsymbol{\phi})$
1: **for** $\forall i \in [\xi]$ $\forall l \in [\ell]$ **do**
2: Sample $\beta \xleftarrow{\$} \mathbb{Z}_q$ $(KA.R)$
3: $m_R \leftarrow \beta \cdot G$ $(KA.msg_2)$
4: $m_{b(i,l)} \leftarrow \mathsf{H}(\beta \cdot A, i \mid\mid b_{(i)})$ $(KA.key_2)$
5: Sample $\phi_{i,l,1-b_{(i)}} \xleftarrow{\$} \mathbb{G}$ $(POPF.Program)$
6: $\phi_{i,l,b_{(i)}} \leftarrow m_R - \mathsf{H}_{b_{(i)}}(\phi_{i,l,1-b_{(i)}})$ $(POPF.Program)$
7: $\mathsf{Send}(\boldsymbol{\phi_0} = \{\{\phi_{i,l,0}\}_{l\in[\ell]}\}_{i\in[\xi]},\ \boldsymbol{\phi_1} = \{\{\phi_{i,l,1}\}_{l\in[\ell]}\}_{i\in[\xi]}) \rightarrow \mathcal{S}$
return $(\boldsymbol{m_b} = \{\{m_{b(i,l)}\}_{l\in[\ell]}\}_{i\in[\xi]})$

$\mathcal{S}$.**Round3**$(\boldsymbol{\phi_0}, \boldsymbol{\phi_1} \in \mathbb{G}^{\xi\times\ell}) \dashrightarrow \boldsymbol{m_0}, \boldsymbol{m_1}$
1: **for** $\forall i \in [\xi]$ $\forall l \in [\ell]$ $\forall j \in \{0,1\}$ **do**
2: $P \leftarrow \phi_{j(i,l)} + \mathsf{H}_j(\phi_{1-j(i,l)})$ $(POPF.Eval)$
3: $m_{j(i,l)} \leftarrow \mathsf{H}(a \cdot P, i \mid\mid j)$ $(KA.key_1)$
return $\boldsymbol{m_0} = \{\{m_{0(i,l)}\}_{l\in[\ell]}\}_{i\in[\xi]},\ \boldsymbol{m_1} = \{\{m_{1(i,l)}\}_{l\in[\ell]}\}_{i\in[\xi]}$

MRR21 Base ROT. We also implement the base ROT protocol from [MRR21b] which provides an alternative construction with different performance characteristics.

Protocol 4.5 $\mathsf{BaseROT}_\kappa$ - [MRR21] Batched Base ROT

A two-party protocol realizing the $\mathcal{F}^{ROT}_{\kappa,\kappa}$ functionality for κ random OT instances using elliptic curve cryptography. Based on [MRR21] with Masny-Rindal POPF [MR19] and ECDHKA.

Players: Sender $\mathcal{S}$, and Receiver $\mathcal{R}$
Inputs: $\mathcal{P}_A \& \mathcal{P}_B \to \text{sid} \in \{0,1\}^*$ as session id
Outputs: $\mathcal{S} \leftarrow (\boldsymbol{m}^0, \boldsymbol{m}^1) \in \mathbb{Z}_2^{\kappa \times \kappa \times 2}$ $\mathcal{R} \leftarrow (\boldsymbol{b} \in \mathbb{Z}_2^\kappa, \boldsymbol{m^b} \in \mathbb{Z}_2^{\kappa \times \kappa})$

$\mathcal{S}.\mathsf{Round1}() \dashrightarrow \boldsymbol{A}$
1: Sample $\boldsymbol{a} \xleftarrow{\$} \mathbb{Z}_q^\kappa$ as random scalars
2: Compute $\boldsymbol{A} \leftarrow \{a_{(i)} \cdot G\}_{i \in [\kappa]}$ as curve points
3: $\mathsf{Send}(\boldsymbol{A}) \to \mathcal{R}$

$\mathcal{R}.\mathsf{Round2}(\boldsymbol{A}) \dashrightarrow (\boldsymbol{B}, \boldsymbol{b}, \boldsymbol{m^b})$
1: Sample $\boldsymbol{b} \xleftarrow{\$} \mathbb{Z}_2^\kappa$ as choice bits
2: Sample $\boldsymbol{y} \xleftarrow{\$} \mathbb{Z}_q^\kappa$ as random scalars
3: Compute $\boldsymbol{B} \leftarrow \{y_{(i)} \cdot G + b_{(i)} \cdot A_{(i)}\}_{i \in [\kappa]}$
4: Compute $\boldsymbol{k}^b \leftarrow \{\mathsf{H}(y_{(i)} \cdot A_{(i)}, i, \text{sid})\}_{i \in [\kappa]}$ as receiver keys
5: Set $\boldsymbol{m^b} \leftarrow \boldsymbol{k}^b$
6: $\mathsf{Send}(\boldsymbol{B}) \to \mathcal{S}$
return $(\boldsymbol{b}, \boldsymbol{m^b})$

$\mathcal{S}.\mathsf{Round3}(\boldsymbol{B}) \dashrightarrow (\boldsymbol{m}^0, \boldsymbol{m}^1)$
1: Compute $\boldsymbol{k}^0 \leftarrow \{\mathsf{H}(a_{(i)} \cdot B_{(i)}, i, \text{sid})\}_{i \in [\kappa]}$ as sender keys for 0
2: Compute $\boldsymbol{k}^1 \leftarrow \{\mathsf{H}(a_{(i)} \cdot (B_{(i)} - A_{(i)}), i, \text{sid})\}_{i \in [\kappa]}$ as sender keys for 1
3: Set $\boldsymbol{m}^0 \leftarrow \boldsymbol{k}^0$ and $\boldsymbol{m}^1 \leftarrow \boldsymbol{k}^1$
return $(\boldsymbol{m}^0, \boldsymbol{m}^1)$

ROT Extension Protocols. We implement several $\binom{2}{1}$-ROT Extension protocols with varying security and performance properties:

SoftspokenOT Extension. We implement the $\binom{2}{1}$-ROT Extension from SoftspokenOT [Roy22]. The protocol is depicted in Protocol 4.6. The protocol comprises three rounds, and requires a one-time setup of κ batched Base OTs where the Sender/Receiver roles are reversed w.r.t. the ROT extension. To realize this protocol, we the parameters of [Roy22, Figure 12] to $p = q_{softspoken} = 2$, $k = 1$, $\mathcal{C} = Rep(\mathbb{F}_2^n)$. The notation for our protocol follows closely that of an analogous description of a $\binom{2}{1}$-ROT from [KOS15, Figure 10]. We perform several modifications with respect to [KOS15]:

1. Following the definitions from SoftspokenOT [Roy22], we apply the Fiat-Shamir transformation [FS86] to convert the interactive coin-flipping exchange for the challenge generation into a non-interactive computation based on the currently exchanged transcript. This maintains UC security as long as the ROM is salted properly. As a clarifying note, the consistency check of the protocol can be framed as an interactive proof IP

without a commitment phase (therefore it is not a sigma protocol) and with a single challenge generation. Consequently, it suffices to query the random oracle only once in the Random Oracle Model (ROM), an therefore the *salt* that Fiat-Shamir transformation requires [CY24, Section 14] is immediately achieved by appending the Session Id to the transcript at the beginning of the protocol.

2. We set the statistical security parameter $\sigma = \lambda$ to align the statistical security with the computational security given that we are using the Fiat-Shamir transformation.

3. We generalize the behavior of the protocol to run a batch of ξ instances of the protocol in parallel, each with messages of ℓ blocks of λ bits each.

Overall, the protocol relies on a one-time setup of λ Base OTs whose results are used as persistent seeds for all executions of the protocol. A first *expansion* phase (step 3 of $\mathcal{R}$.Round1 and step 1 of $\mathcal{S}$.Round2) extends these seeds and establishes the correlation with the receiver's choice bits x (step 4 of $\mathcal{R}$.Round1 and step 2 of $\mathcal{S}$.Round2). To induce the same choice inside all bits of each message, the receiver repeats his choice bits ℓ times ($\boldsymbol{x}_{\text{rep}}$ in step 1 of $\mathcal{R}$.Round1) and then concatenates σ additional randomz choice bits ($\boldsymbol{x}_{\boldsymbol{\sigma}}$) to be consumed as part of the consistency check. A subsequent *consistency check* phase, made non-interactive via the Fiat-Shamir transformation, provides security against a malicious receiver[15] (steps 5-6 of $\mathcal{R}$.Round1 and steps 3-4 of $\mathcal{S}$.Round2). The protocol concludes with a *re-randomization* phase, where both parties break the correlation of their output messages with the Base OT choice bits while maintaining the correlation with the ROTe receiver's choice bits (steps 7-8 of $\mathcal{R}$.Round1 and steps 5-6 of $\mathcal{S}$.Round2).

[15]The protocol is secure against a malicious sender by design, as the random messages $(\boldsymbol{m_0}, \boldsymbol{m_1})$ arise from an expansion of the baseOT seeds $(\boldsymbol{k_0}, \boldsymbol{k_1})$

Protocol 4.6 $\mathsf{ROTe}_{\xi,\ell}(\boldsymbol{x}) \to (\boldsymbol{m_0}, \boldsymbol{m_1}, \boldsymbol{m_x})$

Maliciously secure ROT extension protocol from SoftSpokenOT [Roy22] for a batch with ξ messages of $\ell \times \kappa$ bits each, setting $\eta = \xi \times \ell \times \kappa$ and $\mu = \eta/\sigma$. It uses a pseudo-random generator $\mathsf{PRG}\colon \mathbb{Z}_2^\kappa \mapsto \mathbb{Z}_2^{\eta'}$ for $\eta' = \eta + \sigma$ (e.g., $\mathsf{TmmoHash}_{\eta'}$), a transcript T (e.g., Scheme 3.7), a base OT (BBOT) and a hash $\mathsf{H}\colon \mathbb{Z}_2^\kappa \mapsto \mathbb{Z}_2^\kappa$

Players: sender $\mathcal{S}$, and receiver $\mathcal{R}$
Inputs: $\mathcal{R}: \boldsymbol{x} \in \mathbb{Z}_2^\xi$, the choice bits
Outputs: $\mathcal{S}: \boldsymbol{m_0}, \boldsymbol{m_1} \in \mathbb{Z}_2^\eta$, pairs of random messages
$\mathcal{R}: \boldsymbol{m_x} \in \mathbb{Z}_2^\eta$, chosen messages such that
$m_{x(i,j)} = m_{0(i,j)} x_{(i)} \oplus m_{1(i,j)}(1 - x_{(i)}) \quad \forall i \in [\xi] \;\; \forall j \in [\ell]$

$\mathcal{S}\&\mathcal{R}.\mathsf{Setup}() \dashrightarrow \mathcal{S}: (\boldsymbol{k_0}, \boldsymbol{k_1});\ \mathcal{R}: \boldsymbol{k_b}$
1: $\mathcal{S}$ samples $\boldsymbol{b} \xleftarrow{\$} \mathbb{Z}_2^\kappa$ as the base OT choice bits
2: $\mathcal{R}$ runs $\mathsf{BBOT}_\kappa()$ as the base OT sender, obtaining $\boldsymbol{k_0}, \boldsymbol{k_1} \in \mathbb{Z}_2^{\kappa \times \kappa}$
3: $\mathcal{S}$ runs $\mathsf{BBOT}_\kappa(\boldsymbol{b})$ as the base OT receiver, receiving $\boldsymbol{k_b} \in \mathbb{Z}_2^{\kappa \times \kappa}$

$\mathcal{R}.\mathsf{Round1}(\boldsymbol{x} \in \mathbb{Z}_2^\xi) \dashrightarrow \boldsymbol{u}, \dot{\boldsymbol{x}}, \dot{\boldsymbol{t}}, \boldsymbol{m_x}$
1: Set $\boldsymbol{x}_{\text{rep}} \leftarrow \{\{x_{(i)}, x_{(i)}, \ldots, x_{(i)}\}_\ell\}_{i \in [\xi]}$ by repeating ℓ times $\boldsymbol{x}$
2: Sample $\boldsymbol{x_\sigma} \xleftarrow{\$} \mathbb{Z}_2^\sigma$ and concatenate $\boldsymbol{x'} \leftarrow \boldsymbol{x}_{\text{rep}} \,||\, \boldsymbol{x_\sigma}$
3: Extend $(\boldsymbol{t_0}, \boldsymbol{t_1}) \leftarrow \{\mathsf{PRG}(k_{0(i)}), \mathsf{PRG}(k_{1(i)})\}_{i \in [\kappa]}$ with $\boldsymbol{t_0}, \boldsymbol{t_1} \in \mathbb{Z}_2^{\kappa \times \eta'}$
4: $\boldsymbol{u} \leftarrow \{\{t_{0(i,j)} \oplus t_{1(i,j)} \oplus x'_{(j)}\}_{j \in [\eta']}\}_{i \in [\kappa]}$ with $\boldsymbol{u} \in \mathbb{Z}_2^{\kappa \times \eta'}$
5: Run $\mathsf{T}.\mathsf{Append}(\boldsymbol{u})$ and $\boldsymbol{\chi} \leftarrow \mathsf{T}.\mathsf{Extract}(\eta)$ to get the challenge $\boldsymbol{\chi} \in \mathbb{Z}_2^{\mu \times \sigma}$
6: Compute the challenge response $(\dot{\boldsymbol{x}}, \dot{\boldsymbol{t}})$ as:
6.a: $\dot{\boldsymbol{x}} \leftarrow \{x_{\sigma(k)} \oplus \bigoplus_{m=1}^{\mu} \chi_{(m,k)} \cdot x_{\text{rep}(\sigma m + k)}\}_{k \in [\sigma]}$ with $\dot{\boldsymbol{x}} \in \mathbb{Z}_2^\sigma$
6.b: $\dot{\boldsymbol{t}} \leftarrow \{\{t_{0(i,\eta+k)} \oplus \bigoplus_{m=1}^{\mu} \chi_{(m,k)} \cdot t_{0(i,\sigma m+k)}\}_{k \in [\sigma]}\}_{i \in [\kappa]}$ with $\dot{\boldsymbol{t}} \in \mathbb{Z}_2^{\kappa \times \sigma}$
7: Transpose $\boldsymbol{t'_0} \leftarrow \{\{t_{0(i,j)}\}_{i \in [\kappa]}\}_{j \in [\eta']}$ with $\boldsymbol{t'_0} \in \mathbb{Z}_2^{\eta' \times \kappa}$
8: $\mathsf{Send}(\boldsymbol{u}, \dot{\boldsymbol{x}}, \dot{\boldsymbol{t}}) \to \mathcal{S}$
9: $\boldsymbol{m_x} \leftarrow \{\{\mathsf{H}(j \,||\, t'_{0(j\ell+l)})\}_{l \in [\ell]}\}_{j \in [\eta]}$
return $\boldsymbol{m_x}$

$\mathcal{S}.\mathsf{Round2}(\boldsymbol{u}, \dot{\boldsymbol{x}}, \dot{\boldsymbol{t}}) \dashrightarrow (\boldsymbol{m_0}, \boldsymbol{m_1})$
1: Extend $\boldsymbol{t_b} \leftarrow \{\mathsf{PRG}(k_{b(i)})\}_{i \in [\kappa]}$ with $\boldsymbol{t_b} \in \mathbb{Z}_2^{\kappa \times \eta'}$
2: $\boldsymbol{q} \leftarrow \{\{b_{(i)} \cdot u_{(i,j)} \oplus t_{b(i,j)}\}_{j \in [\eta']}\}_{i \in [\kappa]}$ with $\boldsymbol{q} \in \mathbb{Z}_2^{\kappa \times \eta'}$
3: Run $\mathsf{T}.\mathsf{Append}(\boldsymbol{u})$ and $\boldsymbol{\chi} \leftarrow \mathsf{T}.\mathsf{Extract}(\eta)$ to get the challenge $\boldsymbol{\chi} \in \mathbb{Z}_2^{\mu \times \sigma}$
4: Verify the challenge response:
4.a: $\dot{\boldsymbol{q}} \leftarrow \{\{q_{(i,\eta+k)} \oplus \bigoplus_{m=1}^{\mu} \chi_{(m,k)} \cdot q_{(i,\sigma m+k)}\}_{k \in [\sigma]}\}_{i \in [\kappa]}$ with $\dot{\boldsymbol{q}} \in \mathbb{Z}_2^{\kappa \times \sigma}$
4.b: Check $q_{(i,k)} \stackrel{?}{=} \dot{q}_{(i,k)} \quad \forall i \in [\kappa] \;\; \forall k \in [\eta']$; otherwise ABORT
5: Transpose $\boldsymbol{q'} \leftarrow \{\{q_{(i,j)}\}_{i \in [\kappa]}\}_{j \in [\eta']}$ with $\boldsymbol{q'} \in \mathbb{Z}_2^{\eta' \times \kappa}$
6: $(\boldsymbol{m_0}, \boldsymbol{m_1}) \leftarrow \{\{\mathsf{H}(j \,||\, q'_{(j)}), \mathsf{H}(j \,||\, (q_{(j\ell+l)} \oplus b_{(j\ell+l)}))\}_{l \in [\ell]}\}_{j \in [\xi]}$
return $(\boldsymbol{m_0}, \boldsymbol{m_1})$

KOS ROT Extension. We implement the KOS (Keller-Orsini-Scholl) ROT extension protocol [KOS15], which provides efficient malicious security through a consistency check mechanism.

Protocol 4.7 $\mathsf{ROTe}_{\mathsf{KOS}}$

A two-party protocol realizing the $\mathcal{F}^{VCOT}_{\mathbb{F},L}$ functionality for L vector correlated OT instances over $\mathbb{F} = \mathbb{F}_{2^\lambda}$, using κ base OTs as seeds. Achieves malicious security with statistical parameter S.

Players: Sender $\mathcal{S}$, and Receiver $\mathcal{R}$

Inputs: $\mathcal{S} \to (\boldsymbol{k}^0, \boldsymbol{k}^1) \in \mathbb{Z}_2^{\kappa\times\kappa\times 2}$, base OT sender seeds
$\mathcal{R} \to (\Delta \in \mathbb{F}, \boldsymbol{k}^\Delta \in \mathbb{Z}_2^{\kappa\times\kappa})$, global correlation and base OT receiver seeds
$\mathcal{S}\&\mathcal{R} \to \mathrm{sid} \in \{0,1\}^*$ as session id

Outputs: $\mathcal{S} \leftarrow \boldsymbol{v} \in \mathbb{F}^L$ $\mathcal{R} \leftarrow \boldsymbol{w} \in \mathbb{F}^L$ s.t. $w_{(i)} = \Delta \cdot b_{(i)} + v_{(i)}$ $\forall i \in [L]$

$\mathcal{R}.\mathsf{Round1}() \dashrightarrow \boldsymbol{U}$

1: Sample $\boldsymbol{b} \xleftarrow{\$} \mathbb{Z}_2^L$ as choice bits for the extended OTs
2: For each $j \in [\kappa]$, expand $\boldsymbol{u}_j \leftarrow \mathrm{PRG}(k^\Delta{}_{(j)})$ to length L'
3: Form matrix $\boldsymbol{U} \in \mathbb{Z}_2^{\kappa\times L'}$ from rows $\boldsymbol{u}_j$
4: XOR choice bits: for each $i \in [L]$, $U_{(:,i)} \leftarrow U_{(:,i)} \oplus (b_{(i)} \cdot \Delta)$
5: $\mathsf{Send}(\boldsymbol{U}) \to \mathcal{S}$

$\mathcal{S}.\mathsf{Round2}(\boldsymbol{U}) \dashrightarrow \boldsymbol{v}$

1: For each $j \in [\kappa]$, expand $\boldsymbol{q}_j^0 \leftarrow \mathrm{PRG}(k^0{}_{(j)})$ and $\boldsymbol{q}_j^1 \leftarrow \mathrm{PRG}(k^1{}_{(j)})$ to length L'
2: Form matrices $\boldsymbol{Q}^0, \boldsymbol{Q}^1 \in \mathbb{Z}_2^{\kappa\times L'}$ from rows $\boldsymbol{q}_j^0, \boldsymbol{q}_j^1$
3: Compute $\boldsymbol{T} \leftarrow \boldsymbol{Q}^0 \oplus \boldsymbol{Q}^1 \oplus \boldsymbol{U}$
4: For each $i \in [L]$, compute $v_{(i)} \leftarrow \mathsf{H}(\boldsymbol{Q}^0_{:,i}, i, \mathrm{sid})$
5: Check consistency using random linear combination over first S columns
return $\boldsymbol{v}$

$\mathcal{R}.\mathsf{Finalize}() \dashrightarrow \boldsymbol{w}$

1: For each $i \in [L]$, compute $w_{(i)} \leftarrow \mathsf{H}(\boldsymbol{U}_{:,i} \oplus b_{(i)} \cdot \boldsymbol{T}_{:,i}, i, \mathrm{sid})$
return $\boldsymbol{w}$

BCGI ROT Extension. We also implement the BCGI (Boyle-Couteau-Gilboa-Ishai) ROT extension [BCG+19], which offers different performance characteristics and security guarantees.

Protocol 4.8 $\mathsf{ROTe}_{\mathsf{BCGI}}$ - ROT Extension

A two-party protocol realizing the $\mathcal{F}^{VCOT}{}_{\mathbb{F},L}$ functionality for arbitrary L vector correlated OT instances over $\mathbb{F} = \mathbb{F}_{2^\lambda}$, using programmable sVOLE seeds. Based on [BCG+19a] silent OT extension with pseudorandom correlation generators.

Players: Sender $\mathcal{S}$, and Receiver $\mathcal{R}$
Inputs: $\mathcal{S} \to$ sVOLE sender seeds from sVOLE
$\mathcal{R} \to (\Delta \in \mathbb{F}$, sVOLE receiver seeds from $\mathsf{sVOLE})$
$\mathcal{S}\&\mathcal{R} \to \mathrm{sid} \in \{0,1\}^*$ as session id
Outputs: $\mathcal{S} \leftarrow \boldsymbol{v} \in \mathbb{F}^L \quad \mathcal{R} \leftarrow \boldsymbol{w} \in \mathbb{F}^L$ s.t. $w_{(i)} = \Delta \cdot b_{(i)} + v_{(i)} \ \ \forall i \in [L]$

$\mathcal{S}\&\mathcal{R}.\mathsf{Setup}()\dashrightarrow$
0: $\mathcal{S}\&\mathcal{R}$ run sVOLE initialization to generate LPN seed sVOLEs
1: Both parties locally expand seeds using LPN to generate long sVOLE correlations

$\mathcal{R}.\mathsf{Extend}()\dashrightarrow \boldsymbol{U}$
1: Sample $\boldsymbol{b} \xleftarrow{\$} \mathbb{Z}_2^L$ as choice bits
2: Use expanded sVOLE receiver shares $\boldsymbol{v}_R$ to mask choice bits
3: Compute $\boldsymbol{U} \leftarrow \boldsymbol{v}_R \oplus (\boldsymbol{b} \cdot \Delta)$
4: $\mathsf{Send}(\boldsymbol{U}) \to \mathcal{S}$

$\mathcal{S}.\mathsf{Extend}(\boldsymbol{U})\dashrightarrow \boldsymbol{v}$
1: Use expanded sVOLE sender shares $\boldsymbol{w}_S$ and correction $\boldsymbol{U}$
2: For each $i \in [L]$, derive $v_{(i)} \leftarrow \mathsf{H}(w_{S(i)}, i, \mathrm{sid})$
return $\boldsymbol{v}$

$\mathcal{R}.\mathsf{Finalize}()\dashrightarrow \boldsymbol{w}$
1: For each $i \in [L]$, derive $w_{(i)} \leftarrow \mathsf{H}(v_{R(i)} \oplus b_{(i)} \cdot U_{(i)}, i, \mathrm{sid})$
return $\boldsymbol{w}$

4.1.2 Vector Oblivious Linear Evaluation

Vector Oblivious Linear Evaluation (VOLE) is a fundamental cryptographic primitive that allows two parties to generate correlated randomness over vectors. We implement multiple VOLE protocols with different security and efficiency characteristics.

VOLE Functionalities. We define several VOLE-related functionalities:

Base sVOLE. The base subfield VOLE functionality provides the foundation for more complex VOLE protocols.

Functionality 4.11 $\mathcal{F}^{BaseSVOLE}_{\mathbb{F},M}(\boldsymbol{u}) \longrightarrow (\Delta, \boldsymbol{w}, \boldsymbol{v})$

Random Base Subfield Vector Oblivious Linear Evaluation interacting with two parties Alice $\mathcal{P}_A$ and Bob $\mathcal{P}_B$.

Players: A sender $\mathcal{P}_A$, and a receiver $\mathcal{P}_B$.
Inputs: $\mathcal{P}_A \leftarrow \boldsymbol{u} \in \mathbb{F}_0^M$, a vector of subfield multiplicative shares.
Outputs: $\mathcal{P}_A \leftarrow \boldsymbol{w} \in \mathbb{F}^M$, a vector of additive shares.
$\mathcal{P}_B \leftarrow (\Delta \in \mathbb{F}, \boldsymbol{v} \in \mathbb{F}^M)$, a field element and the additive shares.

$\mathcal{F}^{BaseSVOLE}$.**Sampling**()$\dashrightarrow \Delta, \boldsymbol{w}$
Sample and send $\Delta \xleftarrow{\$} \mathbb{F}$ to $\mathcal{P}_B$, sample and send $\boldsymbol{w} \xleftarrow{\$} \mathbb{F}^M$ to $\mathcal{P}_A$.
$\mathcal{F}^{BaseSVOLE}$.**Multiplication**($\boldsymbol{u}$)$\dashrightarrow \boldsymbol{v}$
Receive $\boldsymbol{u}$ from $\mathcal{P}_A$, send $\boldsymbol{v}$ to $\mathcal{P}_B$ s.t. $w_{(i)} = u_{(i)} \cdot \Delta + v_{(i)} \ \ \forall i \in [M]$.

sVOLE. The standard subfield VOLE functionality.

Functionality 4.12 $\mathcal{F}^{sVOLE}_{\mathbb{F},M}(\boldsymbol{u}) \longrightarrow (\Delta, \boldsymbol{w}, \boldsymbol{v})$

Random Subfield Vector Oblivious Linear Evaluation interacting with two parties, sender $\mathcal{S}$ and receiver $\mathcal{R}$, over a field extension $\mathbb{F}/\mathbb{F}_0$ for M instances.

Players: A sender $\mathcal{S}$, and a receiver $\mathcal{R}$.
Inputs: $\mathcal{S} \leftarrow \boldsymbol{u} \in \mathbb{F}_0^M$, a vector of subfield multiplicative shares.
Outputs: $\mathcal{S} \leftarrow \boldsymbol{w} \in \mathbb{F}^M$, a vector of additive shares.
$\mathcal{R} \leftarrow (\Delta \in \mathbb{F}, \boldsymbol{v} \in \mathbb{F}^M)$, a field element and additive shares s.t. $w_{(i)} = u_{(i)} \cdot \Delta + v_{(i)}$ for all $i \in [M]$.

$\mathcal{F}^{sVOLE}$.**Sampling**()$\dashrightarrow \Delta, \boldsymbol{w}$
Sample and send $\Delta \xleftarrow{\$} \mathbb{F}$ to $\mathcal{R}$, sample and send $\boldsymbol{w} \xleftarrow{\$} \mathbb{F}^M$ to $\mathcal{S}$.
$\mathcal{F}^{sVOLE}$.**Multiplication**($\boldsymbol{u}$)$\dashrightarrow \boldsymbol{v}$
Receive $\boldsymbol{u}$ from $\mathcal{S}$, send $\boldsymbol{v}$ to $\mathcal{R}$ s.t. $w_{(i)} = u_{(i)} \cdot \Delta + v_{(i)}$ for all $i \in [M]$.

VOLE Protocols. We implement several VOLE protocols:

Base sVOLE Protocol. The base subfield VOLE protocol provides correlated randomness generation without requiring preprocessing.

Protocol 4.9 $\mathsf{BaseSVOLE}_{\mathbb{F},M}$

A two-party protocol realizing the $\mathcal{F}^{BaseSVOLE}_{\mathbb{F},M}$ random subfield multiplicative to additive share generation with malicious security in a field extension $\mathbb{F}/\mathbb{F}_0$ of degree d, based on a $\mathsf{COPEe}_{\mathbb{F}}$ subprotocol and a hash function $\mathsf{H}_{\mathbb{F}}\colon \{0,1\}^* \mapsto \mathbb{F}$.

Players: Alice $\mathcal{P}_A$, and Bob $\mathcal{P}_B$

Inputs: $\mathcal{P}_A \to \boldsymbol{u} \in \mathbb{F}_0^M$; $\mathcal{P}_B \to \Delta \in \mathbb{F}$; $\mathcal{P}_A\&\mathcal{P}_B \to \mathrm{sid} \in \{0,1\}^*$ as session id

Outputs: $\mathcal{P}_A \leftarrow \boldsymbol{w} \in \mathbb{F}^M$ $\mathcal{P}_B \leftarrow \boldsymbol{v} \in \mathbb{F}^M$ s.t. $w_{(i)} = u_{(i)} \cdot \Delta + v_{(i)}$ $\forall i \in [M]$

$\mathcal{P}_A\&\mathcal{P}_B.\mathsf{Setup}()\dashrightarrow$

0: $\mathcal{P}_A\&\mathcal{P}_B$ run $\mathsf{ROTe.Setup}()$ to generate $|\mathbb{F}|$ Base OT seeds for the $\binom{2}{1}$-ROTe functionality with $\mathcal{P}_A$ as sender $\mathcal{S}$ and $\mathcal{P}_B$ as receiver $\mathcal{R}$

1: Set a gadget vector $\boldsymbol{g} = \{1, G, G^2, \ldots, G^{d-1}\} \in \mathbb{F}^d$ where G is the generator of $\mathbb{F}$

$\mathcal{P}_A.\mathsf{Round1}(\boldsymbol{u})\dashrightarrow(\boldsymbol{s},\boldsymbol{t})$

1: Sample $\boldsymbol{a} \xleftarrow{\$} \mathbb{F}_0^d$ as random subfield coefficients

2: Run $(\boldsymbol{s},\boldsymbol{w}) \leftarrow \mathsf{COPEe.Send}(\boldsymbol{u})$ to obtain sender shares

3: Run $(\boldsymbol{t},\boldsymbol{c}) \leftarrow \mathsf{COPEe.Send}(\boldsymbol{a})$ to obtain consistency check shares

4: $\mathsf{Send}(\boldsymbol{s},\boldsymbol{t}) \to \mathcal{P}_B$

return $\boldsymbol{w}$ as the sender's intermediate output

$\mathcal{P}_B.\mathsf{Round2}(\boldsymbol{s},\boldsymbol{t})\dashrightarrow(\boldsymbol{v},\boldsymbol{b})$

1: Run $\boldsymbol{v} \leftarrow \mathsf{COPEe.Receive}(\boldsymbol{s})$ to obtain receiver shares

2: Run $\boldsymbol{b} \leftarrow \mathsf{COPEe.Receive}(\boldsymbol{t})$ to obtain consistency check shares

return $\boldsymbol{v}$ as the receiver's intermediate output

$\mathcal{P}_A.\mathsf{Round3}(\boldsymbol{u},\boldsymbol{w},\boldsymbol{a},\boldsymbol{c})\dashrightarrow(x,z)$

1: Derive challenge $\boldsymbol{\chi} \leftarrow \mathsf{H}_{\mathbb{F}}(\mathrm{sid} \mid\mid \text{“Chi”})^M$ as a vector of M random field elements

2: Compute $x \leftarrow \sum_{i=1}^{M} \chi_{(i)} \cdot u_{(i)} + \sum_{h=1}^{d} g_{(h)} \cdot a_{(h)}$ as the combined input

3: Compute $z \leftarrow \sum_{i=1}^{M} \chi_{(i)} \cdot w_{(i)} + \sum_{h=1}^{d} g_{(h)} \cdot c_{(h)}$ as the combined output

4: $\mathsf{Send}(x,z) \to \mathcal{P}_B$

return $\boldsymbol{w}$ as the sender's final output

$\mathcal{P}_B.\mathsf{Round4}(x,z,\boldsymbol{v},\boldsymbol{b})\dashrightarrow\boldsymbol{v}$

1: Derive challenge $\boldsymbol{\chi} \leftarrow \mathsf{H}_{\mathbb{F}}(\mathrm{sid} \mid\mid \text{“Chi”})^M$ as a vector of M random field elements

2: Compute $y \leftarrow \sum_{i=1}^{M} \chi_{(i)} \cdot v_{(i)} + \sum_{h=1}^{d} g_{(h)} \cdot b_{(h)}$ as the combined receiver shares

3: Check if $z \stackrel{?}{=} y + x \cdot \Delta$, ABORT otherwise

return $\boldsymbol{v}$ as the receiver's final output

Programmable sVOLE. This protocol allows for programmable correlation patterns in the VOLE outputs.

Protocol 4.10 sVOLE

A two-party protocol realizing the $\mathcal{F}^{sVOLE}_{\mathbb{F},N}$ functionality for arbitrarily large N using LPN-based pseudorandom correlation generators. Reference: Figure 8, [WYKW21].

Players: Sender $\mathcal{S}$, and Receiver $\mathcal{R}$

Inputs: $\mathcal{S} \rightarrow$ sVOLE seeds from $\mathsf{SpsSVOLE}$
$\mathcal{R} \rightarrow (\Delta \in \mathbb{F}$, sVOLE seeds from $\mathsf{SpsSVOLE})$
$\mathcal{S}\&\mathcal{R} \rightarrow \mathrm{sid} \in \{0,1\}^*$ as session id

Outputs: $\mathcal{S} \leftarrow (\boldsymbol{u}, \boldsymbol{w}) \in \mathbb{F}_0^N \times \mathbb{F}^N$ $\mathcal{R} \leftarrow \boldsymbol{v} \in \mathbb{F}^N$ s.t. $w_{(i)} = u_{(i)} \cdot \Delta + v_{(i)}$ $\forall i \in [N]$

$\mathcal{S}\&\mathcal{R}.\mathsf{Setup}()\dashrightarrow$
0: Run $\mathsf{SpsSVOLE}$ to generate:
1: - Seed sVOLEs: $(\boldsymbol{u}_{seed}, \boldsymbol{w}_{seed})$ and $\boldsymbol{v}_{seed}$ of size S
2: - Expansion sVOLEs: $(\boldsymbol{u}_{exp}, \boldsymbol{w}_{exp})$ and $\boldsymbol{v}_{exp}$ of size K
3: Both parties seed LPN matrix generator PRG(sid) with session ID

$\mathcal{S}.\mathsf{Round1}()\dashrightarrow \boldsymbol{u}, \boldsymbol{w}$
1: Generate sparse LPN matrix $\boldsymbol{H} \in \mathbb{F}_0^{N \times K}$ using PRG(sid)
2: For each column $j \in [K]$: get sparse indices from $\boldsymbol{u}_{seed}$
3: Compute $\boldsymbol{u} \leftarrow$ sparse positions from $\boldsymbol{u}_{seed}$
4: Compute $\boldsymbol{w} \leftarrow \boldsymbol{H} \cdot \boldsymbol{w}_{exp} + \boldsymbol{w}_{seed}$ via sparse multiplication
return $(\boldsymbol{u}, \boldsymbol{w})$

$\mathcal{R}.\mathsf{Round2}()\dashrightarrow \boldsymbol{v}$
1: Generate sparse LPN matrix $\boldsymbol{H} \in \mathbb{F}_0^{N \times K}$ using PRG(sid)
2: Compute $\boldsymbol{v} \leftarrow \boldsymbol{H} \cdot \boldsymbol{v}_{exp} + \boldsymbol{v}_{seed}$ via sparse multiplication
return $\boldsymbol{v}$

Single-Point sVOLE. An optimized protocol for generating VOLE correlations at a single point.

Protocol 4.11 SpsSVOLE - [WYKW21] Single-Point Subfield VOLE

A two-party protocol realizing sparse subfield VOLE correlations with malicious security. Generates N sVOLE instances where only $M \ll N$ of the sender's inputs are non-zero. Reference: Figure 7, [WYKW21].

Players: Sender $\mathcal{S}$, and Receiver $\mathcal{R}$

Inputs: $\mathcal{S} \to$ seed sVOLEs $(\boldsymbol{u}, \boldsymbol{w})$ from BaseSVOLE
$\mathcal{R} \to (\Delta \in \mathbb{F}$, seed sVOLEs $\boldsymbol{v}$ from BaseSVOLE$)$
$\mathcal{S}\&\mathcal{R} \to$ sid$\in \{0,1\}^*$ as session id

Outputs: $\mathcal{S} \leftarrow (\boldsymbol{u}_{sparse}, \boldsymbol{w}_{ext})$ where $|\boldsymbol{u}_{sparse}| = M, |\boldsymbol{w}_{ext}| = N$
$\mathcal{R} \leftarrow \boldsymbol{v}_{ext} \in \mathbb{F}^N$ s.t. $w_{ext(i)} = u_{sparse_{(i)}} \cdot \Delta + v_{ext(i)}$ for non-zero positions

$\mathcal{S}\&\mathcal{R}$.Setup()-->
0: Both parties construct a GGM tree of depth d over the seed sVOLEs
1: Generate check sVOLEs $(\boldsymbol{x}, \boldsymbol{z})$ and $\boldsymbol{y}$ for consistency

$\mathcal{S}$.Extend()-->$\boldsymbol{U}$
1: Sample M random leaf positions $\{i_1, \ldots, i_M\}$ where $i_j \in [2^d]$
2: For each leaf i_j: expand seed to leaf using GGM PRG along tree path
3: At each leaf i_j: assign sparse value $u_{sparse_{(j)}}$ from expanded seed
4: Compute correction values $\boldsymbol{U}$ to fix GGM expansion
5: Send$(\boldsymbol{U}) \to \mathcal{R}$

$\mathcal{R}$.Extend($\boldsymbol{U}$)-->
1: Expand full GGM tree from root seed to all 2^d leaves
2: Apply corrections $\boldsymbol{U}$ to obtain dense receiver output $\boldsymbol{v}_{ext}$

$\mathcal{S}\&\mathcal{R}$.Check()-->
1: $\mathcal{S}$ sends check value χ_S computed from $\boldsymbol{x}, \boldsymbol{z}$
2: $\mathcal{R}$ computes check value χ_R from $\boldsymbol{y}$
3: Both parties run WeakEquality(χ_S, χ_R) to verify consistency
4: ABORT if equality check fails

Single-Point sVOLE with Equality Check. An enhanced version of single-point sVOLE that includes equality checking capabilities.

Protocol 4.12 $\mathsf{WeakEquality}_{\mathbb{F}}$ - [WYKW21] Weak Equality Check

A two-party subprotocol for checking equality of field elements with abort. Uses hash-based commitments to prevent selective failure attacks.

Players: Sender $\mathcal{S}$, and Receiver $\mathcal{R}$
Inputs: $\mathcal{S} \to x \in \mathbb{F}$, sender's value
$\mathcal{R} \to y \in \mathbb{F}$, receiver's value
$\mathcal{S}\&\mathcal{R} \to \text{sid} \in \{0,1\}^*$ as session id
Outputs: Both parties accept if $x = y$, otherwise ABORT

$\mathcal{R}.\mathsf{Commit}(\text{y})\text{--}\!\!\rightarrow\text{c}$
1: Sample $r \xleftarrow{\$} \{0,1\}^\kappa$ as randomness
2: Compute $c \leftarrow \mathsf{H}(y \mid\mid r \mid\mid \text{sid})$ as commitment
3: $\mathsf{Send}(c) \to \mathcal{S}$

$\mathcal{S}.\mathsf{Send}(\text{x, c})\text{--}\!\!\rightarrow\text{x}$
1: $\mathsf{Send}(x) \to \mathcal{R}$

$\mathcal{R}.\mathsf{Check}(\text{x})\text{--}\!\!\rightarrow (r, y)$ or $\perp$
1: Check if $x \stackrel{?}{=} y$
2: If equal: $\mathsf{Send}((r, y)) \to \mathcal{S}$
3: Otherwise: $\mathsf{Send}(\perp) \to \mathcal{S}$ and ABORT

$\mathcal{S}.\mathsf{Verify}(\text{r, y})\text{--}\!\!\rightarrow$
1: Verify $c \stackrel{?}{=} \mathsf{H}(y \mid\mid r \mid\mid \text{sid})$
2: Verify $x \stackrel{?}{=} y$
3: ABORT if any check fails

4.1.3 Oblivious Linear Evaluation

Oblivious Linear Evaluation (OLE) is a two-party cryptographic primitive where one party (the sender) inputs a linear function and the other party (the receiver) inputs a point, and the receiver learns the evaluation of the function at that point without the sender learning the point or the receiver learning the function.

OLE Functionalities. We define the OLE functionality:

Functionality 4.13 $\mathcal{F}^{OLE}_{\mathbb{F},M}(\boldsymbol{u}) \longrightarrow (\boldsymbol{w}, \boldsymbol{x}, \boldsymbol{v})$

Oblivious Linear Evaluation interacting with two parties, sender $\mathcal{S}$ and receiver $\mathcal{R}$, over a field $\mathbb{F}$ for M instances.

Players: A sender $\mathcal{S}$, and a receiver $\mathcal{R}$.
Inputs: $\mathcal{S} \leftarrow \boldsymbol{u} \in \mathbb{F}^M$, a vector of slope values.
Outputs: $\mathcal{S} \leftarrow \boldsymbol{w} \in \mathbb{F}^M$, random output values.
$\mathcal{R} \leftarrow (\boldsymbol{x} \in \mathbb{F}^M, \boldsymbol{v} \in \mathbb{F}^M)$, random input values and outputs s.t. $w_{(i)} = u_{(i)} \cdot x_{(i)} + v_{(i)}$ for all $i \in [M]$.

$\mathcal{F}^{OLE}.\mathsf{Sample}() \dashrightarrow \boldsymbol{w}, \boldsymbol{x}$
Sample $\boldsymbol{w} \xleftarrow{\$} \mathbb{F}^M$ and send to $\mathcal{S}$, sample $\boldsymbol{x} \xleftarrow{\$} \mathbb{F}^M$ and send to $\mathcal{R}$.

$\mathcal{F}^{OLE}.\mathsf{LinearEval}(\boldsymbol{u}) \dashrightarrow \boldsymbol{v}$
Receive $\boldsymbol{u}$ from $\mathcal{S}$, compute $v_{(i)} = w_{(i)} - u_{(i)} \cdot x_{(i)}$ for all $i \in [M]$, send $\boldsymbol{v}$ to $\mathcal{R}$.

OLE Protocols. We implement several OLE protocols with different characteristics:

Binary OLE. An OLE protocol optimized for binary fields, providing efficient operations over $\mathbb{F}_2$.

Protocol 4.13 $\mathsf{BinaryOLE}_M$ - Binary Field OLE

A two-party protocol realizing the $\mathcal{F}^{OLE}{}_{\mathrm{Gf2},M}$ functionality over the binary field $\mathrm{Gf2} = \{0,1\}$ using random oblivious transfer. Efficient special case for binary arithmetic.

Players: Sender $\mathcal{S}$, and Receiver $\mathcal{R}$
Inputs: $\mathcal{S} \to \boldsymbol{u} \in \mathrm{Gf2}^M$, sender's multiplicative inputs
$\mathcal{R} \to \boldsymbol{x} \in \mathrm{Gf2}^M$, receiver's multiplicative inputs
$\mathcal{S}\&\mathcal{R} \to \mathrm{sid} \in \{0,1\}^*$ as session id
Outputs: $\mathcal{S} \leftarrow \boldsymbol{v} \in \mathrm{Gf2}^M$ $\mathcal{R} \leftarrow \boldsymbol{w} \in \mathrm{Gf2}^M$ s.t. $w_{(i)} = x_{(i)} \cdot u_{(i)} + v_{(i)} \ \ \forall i \in [M]$

$\mathcal{S}\&\mathcal{R}.\mathsf{ROT}()$--→ROT outputs
1: Run $\mathcal{F}^{ROT}{}_{\lambda,M}$ with $\mathcal{S}$ as ROT sender and $\mathcal{R}$ as ROT receiver
2: $\mathcal{S}$ obtains $(\boldsymbol{m}^0, \boldsymbol{m}^1)$, $\mathcal{R}$ obtains $(\boldsymbol{b}, \boldsymbol{m}^{\boldsymbol{b}})$

$\mathcal{R}.\mathsf{Round1}(\boldsymbol{x})$--→$\boldsymbol{w}$
1: Set choice bits $\boldsymbol{b} \leftarrow \boldsymbol{x}$
2: For each $i \in [M]$: $w_{(i)} \leftarrow \mathsf{H}(m^{\boldsymbol{b}}{}_{(i)}) \oplus x_{(i)}$
return $\boldsymbol{w}$

$\mathcal{S}.\mathsf{Round2}(\boldsymbol{u})$--→$\boldsymbol{v}$
1: For each $i \in [M]$:
2: Compute $t_0 = \mathsf{H}(m^0{}_{(i)})$ and $t_1 = \mathsf{H}(m^1{}_{(i)})$
3: Set $v_{(i)} \leftarrow t_0 \oplus (t_0 \oplus t_1 \oplus 1) \cdot u_{(i)}$
return $\boldsymbol{v}$

DKLS OLE. The DKLS (Doerner-Kondi-Lee-Shelat) OLE protocol provides efficient OLE with malicious security guarantees.

Protocol 4.14 $\mathsf{OLE}_{\mathbb{F},M}$ - OT-based Malicious OLE

A two-party protocol realizing the $\mathcal{F}^{OLE}_{\mathbb{F},M}$ functionality with malicious security over field $\mathbb{F}$. Based on Protocol 1 from [DKLS19], using oblivious transfer and hash functions.

Players: Sender $\mathcal{S}$, and Receiver $\mathcal{R}$
Inputs: $\mathcal{S} \to \boldsymbol{u} \in \mathbb{F}^M$, sender's multiplicative inputs
$\mathcal{R} \to \boldsymbol{x} \in \mathbb{F}^M$, receiver's multiplicative inputs
$\mathcal{S}\&\mathcal{R} \to \text{sid} \in \{0,1\}^*$ as session id
Outputs: $\mathcal{S} \leftarrow \boldsymbol{v} \in \mathbb{F}^M$ $\mathcal{R} \leftarrow \boldsymbol{w} \in \mathbb{F}^M$ s.t. $w_{(i)} = x_{(i)} \cdot u_{(i)} + v_{(i)}$ $\forall i \in [M]$

$\mathcal{R}.\mathsf{Round1}(\boldsymbol{x})$--→OT choices
1: Decompose each $x_{(i)}$ into bits: $x_{(i)} = \sum_j x_{(i,j)} \cdot 2^j$
2: Prepare OT receiver with choice bits from all $x_{(i,j)}$
3: Run $M \cdot |\mathbb{F}|$ instances of OT as receiver

$\mathcal{S}.\mathsf{Round2}(\boldsymbol{u})$--→OT messages
1: For each $i \in [M]$ and bit position j:
2: Sample $r_{i,j} \xleftarrow{\$} \mathbb{F}$ as random mask
3: Prepare OT messages: $m_0 = r_{i,j}$, $m_1 = r_{i,j} + 2^j \cdot u_{(i)}$
4: Run $M \cdot |\mathbb{F}|$ instances of OT as sender with prepared messages

$\mathcal{S}.\mathsf{Finalize}()$--→$\boldsymbol{v}$
1: For each $i \in [M]$: $v_{(i)} \leftarrow -\sum_j r_{i,j}$
return $\boldsymbol{v}$

$\mathcal{R}.\mathsf{Finalize}()$--→$\boldsymbol{w}$
1: For each $i \in [M]$: $w_{(i)} \leftarrow \sum_j \text{OT\_output}_{i,j}$
return $\boldsymbol{w}$

PCG-based OLE . We employ recent advances in PCGs to generate our OLEs in an amortized setting. This allows us to efficiently produce large batches of OLEs with reduced communication and computational overhead. Our main reference is [LLX+26]. We perform a careful bootstrapping of the PCG-based OLEs, starting with a small number of OLEs generated using the DKLS protocol, and then using those OLEs to seed the PCG and generate a much larger batch of OLEs efficiently.

4.2 N-party Protocols

4.2.1 Singlets: Authenticated Random Masks

A **Singlet** is an authenticated share of a random field element, used as a mask in secure multi-party computation (MPC) protocols. Singlets are essential for input masking, as already authenticated randomness, and as building blocks for other preprocessing resources.

Mathematical Formulation.

- Let $a \in \mathbb{F}$ be a random field element.
- Each party $\mathcal{P}_i$ holds a share $a_{(i)}$ such that $a = \sum_i a_{(i)}$.
- The shares are authenticated, ensuring integrity and correctness.

Table 3: Singlet Structure

Component	Domain	Description
r	$\mathbb{F}$	Random field element
$r_{(i)}$	Share	Party i's authenticated share of r

Functionalities. We define the singlets functionality:

Functionality 4.14 $\mathcal{F}^{Singlets}_{\mathbb{F},M} \longrightarrow \langle\langle \boldsymbol{r} \rangle\rangle$

Authenticated random share generation (Singlets) for M values over field $\mathbb{F}$ among parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_N\}$.

Players: N parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_N\}$.
Inputs: None (all values are randomly sampled).
Outputs: Each party $\mathcal{P}_i$ receives authenticated shares $\langle\langle \boldsymbol{r}_i \rangle\rangle$ of random values s.t. $\sum_{j=1}^{N} r_{(k)_j}$ is uniformly random in $\mathbb{F}$ for all $k \in [M]$, and all MAC relations hold.

$\mathcal{F}^{Singlets}$.**Generate**()$\dashrightarrow \langle\langle \boldsymbol{r} \rangle\rangle$
Sample $\boldsymbol{r} \xleftarrow{\$} \mathbb{F}^M$, generate authenticated additive shares $\{\langle\langle \boldsymbol{r}_i \rangle\rangle\}_{i=1}^{N}$ and send $\langle\langle \boldsymbol{r}_i \rangle\rangle$ to party $\mathcal{P}_i$.

Usage. Singlets are used to mask inputs to the online phase, provide randomness, and serve as foundational elements for constructing more complex correlated resources (e.g., triples, DaBits).

Generation. Singlets are generated by stacking instances of pairwise VOLEs (each based on bidirectional OTs) between pairs of parties, as described in BMRS24 [BMRS24]. The VOLEs are combined and a small subset is sacrificed (opened) to verify the correctness of the remaining singlets. The protocol stack is illustrated in Figure 2.

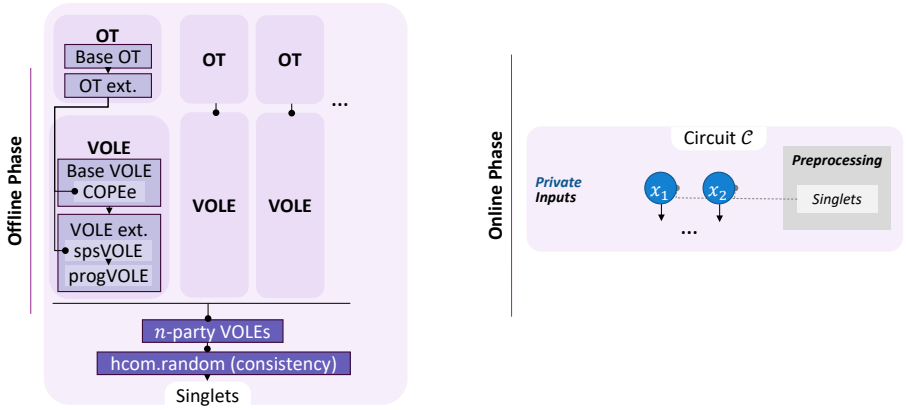

Figure 2: Singlet generation and usage.

BMRS24 Singlets Protocol. We implement the singlets generation protocol from BMRS24 [BMRS24], which provides malicious security through a sacrifice-based verification mechanism.

Protocol 4.15 $\mathsf{SingletsBMRS24}_N$ - [BMRS24] Authenticated Random Singlets

---

An N-party protocol realizing the $\mathcal{F}^{Singlets}_{\mathbb{F},M}$ functionality with malicious security. Generates authenticated random field elements (singlets) using homomorphic commitments. Based on [BMRS24].

Players: N parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_N\}$
Inputs: All parties $\rightarrow \mathrm{sid} \in \{0,1\}^*$ as session id
Outputs: All parties $\leftarrow$ authenticated random singlets $[\![\boldsymbol{r}]\!] = \{[\![r_{(j)}]\!]\}_{j\in[M]}$

$\mathcal{P}_i.\mathsf{Generate}() \dashrightarrow [\![\boldsymbol{r}]\!]$

1: Run $\mathsf{HCom.Random}()$ to generate authenticated random shares
2: Each party $\mathcal{P}_i$ obtains share and MACs for random value $[\![\boldsymbol{r}]\!]$
return $[\![\boldsymbol{r}]\!]$

4.2.2 Triples: Authenticated Beaver Multiplication

A **Triple** (Beaver triple [Bea91]) is a fundamental cryptographic resource in secure multi-party computation (MPC), enabling efficient and secure multiplication of secret-shared values. Each triple consists of three random field elements (a, b, c), with $c = a \cdot b$, and all values are secret-shared among the parties.

Mathematical Formulation.

- Each party $\mathcal{P}_i$ holds shares $(a_{(i)}, b_{(i)}, c_{(i)})$.
- The values are shared additively: $a = \sum_i a_{(i)}$, $b = \sum_i b_{(i)}$, $c = \sum_i c_{(i)}$.

- The shares are correlated such that $c = a \cdot b$.
- For authenticated triples, additional MAC relations are enforced to guarantee integrity.

Table 4: Triple Structure

Component	Domain	Description
a	$\mathbb{F}$	Random field element
b	$\mathbb{F}$	Random field element
c	$\mathbb{F}$	Product $c = a \cdot b$
$(a_{(i)}, b_{(i)}, c_{(i)})$	Shares	Party i's shares of the triple

Functionalities. We define both authenticated and unauthenticated triples functionalities:

Functionality 4.15 $\mathcal{F}^{AuthTriples}_{\mathbb{F},M} \longrightarrow (\langle\!\langle \boldsymbol{a} \rangle\!\rangle, \langle\!\langle \boldsymbol{b} \rangle\!\rangle, \langle\!\langle \boldsymbol{c} \rangle\!\rangle)$

Authenticated Beaver Triple generation for M triples over field $\mathbb{F}$ among parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_N\}$.

Players: N parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_N\}$.

Inputs: None (all values are randomly sampled).

Outputs: Each party $\mathcal{P}_i$ receives authenticated shares $(\langle\!\langle \boldsymbol{a}_i \rangle\!\rangle, \langle\!\langle \boldsymbol{b}_i \rangle\!\rangle, \langle\!\langle \boldsymbol{c}_i \rangle\!\rangle)$ s.t. for all $j \in [M]$:
$\sum_{i=1}^{N} a_{(j)_i} \cdot \sum_{i=1}^{N} b_{(j)_i} = \sum_{i=1}^{N} c_{(j)_i}$, and
all MAC relations hold: $m(x_{(j)})_i = \alpha_i \cdot x_{(j)}$ for $x_{(j)} \in \{a_{(j)}, b_{(j)}, c_{(j)}\}$.

$\mathcal{F}^{AuthTriples}.\mathsf{SampleMAC}() \dashrightarrow \boldsymbol{\alpha}$
Sample global MAC key shares $\{\alpha_i\}_{i=1}^{N}$ s.t. $\sum_i \alpha_i = \alpha$ for random $\alpha \xleftarrow{\$} \mathbb{F}$.

$\mathcal{F}^{AuthTriples}.\mathsf{GenerateTriples}(\boldsymbol{\alpha}) \dashrightarrow \langle\!\langle \boldsymbol{a} \rangle\!\rangle, \langle\!\langle \boldsymbol{b} \rangle\!\rangle, \langle\!\langle \boldsymbol{c} \rangle\!\rangle$
For each $j \in [M]$, sample $a_{(j)}, b_{(j)} \xleftarrow{\$} \mathbb{F}^2$, compute $c_{(j)} = a_{(j)} \cdot b_{(j)}$, generate authenticated additive shares for each value and send $(\langle\!\langle \boldsymbol{a}_i \rangle\!\rangle, \langle\!\langle \boldsymbol{b}_i \rangle\!\rangle, \langle\!\langle \boldsymbol{c}_i \rangle\!\rangle)$ to party $\mathcal{P}_i$.

Functionality 4.16 $\mathcal{F}^{UnauthTriples}_{\mathbb{F},M} \longrightarrow (\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$

Unauthenticated Beaver Triple generation for M triples over field $\mathbb{F}$ among parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_N\}$.

Players: N parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_N\}$.

Inputs: None (all values are randomly sampled).

Outputs: Each party $\mathcal{P}_i$ receives $(\boldsymbol{a}_i, \boldsymbol{b}_i, \boldsymbol{c}_i) \in \mathbb{F}^{3\times M}$ s.t. $\sum_{i=1}^{N} a_{(j)_i} \cdot \sum_{i=1}^{N} b_{(j)_i} = \sum_{i=1}^{N} c_{(j)_i}$ for all $j \in [M]$.

$\mathcal{F}^{UnauthTriples}$.**SampleTriples**()$\dashrightarrow \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$

For each $j \in [M]$, sample $a_{(j)}, b_{(j)} \xleftarrow{\$} \mathbb{F}^2$, compute $c_{(j)} = a_{(j)} \cdot b_{(j)}$.

$\mathcal{F}^{UnauthTriples}$.**Distribute**$(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})\dashrightarrow$

For each triple $(a_{(j)}, b_{(j)}, c_{(j)})$, generate additive shares $\{a_{(j)_i}\}_{i=1}^{N}, \{b_{(j)_i}\}_{i=1}^{N}, \{c_{(j)_i}\}_{i=1}^{N}$ s.t. $\sum_i a_{(j)_i} = a_{(j)}$, $\sum_i b_{(j)_i} = b_{(j)}$, $\sum_i c_{(j)_i} = c_{(j)}$. Send $(\boldsymbol{a}_i, \boldsymbol{b}_i, \boldsymbol{c}_i)$ to party $\mathcal{P}_i$.

Usage. Triples are used to securely compute the product of two secret-shared values in the online phase, without interaction. They allow parties to mask their local computations and reveal only the necessary information for correctness and security.

Generation. Triples are generated using a stack of pairwise OLEs (Oblivious Linear Evaluations), which can be built from bidirectional OTs (Oblivious Transfers) or from modern PCGs. We employ the former, but we'll seek to implement the latter in the future. The OLEs generate unauthenticated triples, which are then authenticated using singlets, as well as a sacrifice step to verify correctness, following the approach in [BMRS24].

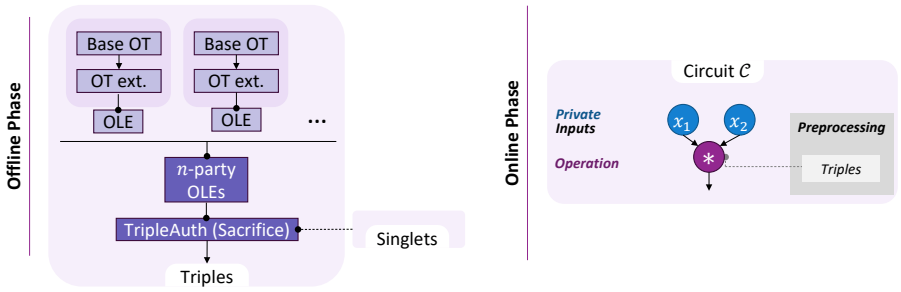

Figure 3: Authenticated Beaver triple generation and usage.

Triple Generation Protocols. We implement multiple protocols for generating authenticated and unauthenticated triples:

Authenticated Triples. The BDOZ-authenticated triples protocol from [BMRS24] provides malicious security through sacrifice-based verification.

Protocol 4.16 AuthTriples

An N-party protocol realizing the $\mathcal{F}^{AuthTriples}_{\mathbb{F},M}$ functionality with malicious security, generating M authenticated Beaver triples. Based on [BMRS24] using homomorphic commitments and sacrificing.

Players: N parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_N\}$
Inputs: All parties $\rightarrow$ sid$\in\{0,1\}^*$ as session id
Outputs: Each $\mathcal{P}_i \leftarrow$ authenticated triple $([\![a]\!]_i, [\![b]\!]_i, [\![c]\!]_i)$ s.t. $[\![c]\!] = [\![a]\!] \cdot [\![b]\!]$ with MACs

$\mathcal{P}_i$.Generate()--→candidate triples
1: Run HCom.Random to generate authenticated random shares $[\![a]\!]$ and $[\![b]\!]$
2: Compute authenticated products using distributed multiplication
3: Each party obtains candidate authenticated triple $([\![a]\!], [\![b]\!], [\![c]\!])$

$\mathcal{P}_i$.Sacrifice()--→verified triples
1: Generate sacrificial triples using same procedure
2: Open linear combinations to check correctness:
3: Sample $\rho \xleftarrow{\$} \mathbb{F}$ via coin-tossing
4: Open $[\![e]\!] = [\![a]\!] - \rho \cdot [\![a']\!]$ and $[\![f]\!] = [\![b]\!] - \rho \cdot [\![b']\!]$
5: Verify $[\![c]\!] - \rho \cdot [\![c']\!] = e \cdot [\![b]\!] + f \cdot [\![a]\!] - e \cdot f$
6: ABORT if verification fails
return verified authenticated triples

YWZ20 Authenticated Triples. An alternative authenticated triples protocol from [YWZ20] provide the same functionality for $\mathbb{F}_2$, since the security of the [BMRS24] protocol depended on the field size and made it unsuitable for binary fields.

Protocol 4.17 $\mathsf{AuthTriplesYWZ20}_N$ - [YWZ20] Authenticated Binary Triples

An N-party protocol realizing the $\mathcal{F}^{AuthTriples}_{\mathbb{F}_{2^\lambda},M}$ functionality over binary field with malicious security. Based on [YWZ20] using PRF-based multiplication and authentication.

Players: N parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_N\}$
Inputs: All parties $\rightarrow \mathrm{sid} \in \{0,1\}^*$ as session id
Outputs: Each $\mathcal{P}_i \leftarrow$ authenticated binary triple $([\![a]\!]_i, [\![b]\!]_i, [\![c]\!]_i)$ over $\mathbb{F}_{2^\lambda}$ s.t. $[\![c]\!] = [\![a]\!] \cdot [\![b]\!]$

$\mathcal{P}_i.\mathsf{Setup}()$--→PRF keys
1: Each party $\mathcal{P}_i$ samples PRF key $k_i \xleftarrow{\$} \{0,1\}^\kappa$
2: Parties engage in authenticated key distribution

$\mathcal{P}_i.\mathsf{Generate}()$--→triples
1: $\phi_i \xleftarrow{\$} \mathbb{F}_{2^\lambda}^M$
2: Compute $u_i \leftarrow \sum_{j \neq i} \mathsf{PRF}_{k_j}(\phi_i)$ via distributed PRF evaluation
3: Set authenticated shares using PRF-MAC technique
4: Compute products using binary field multiplication

$\mathcal{P}_i.\mathsf{Check}()$--→
1: Perform consistency checks on MACs via distributed verification
2: ABORT if any check fails
return authenticated binary triples

Unauthenticated Triples. We build the unauthenticated triples by composing pairwise OLEs, following a standard approach in the literature.

Protocol 4.18 $\mathsf{UnauthTriples}$

An N-party protocol realizing the $\mathcal{F}^{UnauthTriples}_{\mathbb{F},M}$ functionality, generating M unauthenticated Beaver triples (a, b, c) where $a \cdot b = c$. Based on [BMRS24] using OLE and random sampling.

Players: N parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_N\}$
Inputs: All parties $\rightarrow \mathrm{sid} \in \{0,1\}^*$ as session id
Outputs: Each $\mathcal{P}_i \leftarrow (a_i, b_i, c_i) \in \mathbb{F}^M \times \mathbb{F}^M \times \mathbb{F}^M$ s.t. $\sum_i c_i = (\sum_i a_i) \cdot (\sum_i b_i)$

$\mathcal{P}_i.\mathsf{Sample}()$--→$a_i$, b_i
1: Sample $(a_i, b_i) \xleftarrow{\$} (\mathbb{F}^M, \mathbb{F}^M)$

$\mathcal{P}_i.\mathsf{OLE}()$--→$c_i$
1: For each pair $(\mathcal{P}_i, \mathcal{P}_j)$ where $i < j$:
2: Run $\mathsf{OLE}_{\mathbb{F},M}$ with $\mathcal{P}_i$ as sender (input a_i) and $\mathcal{P}_j$ as receiver (input b_j)
3: $\mathcal{P}_i$ obtains $v_{i,j}$, $\mathcal{P}_j$ obtains $w_{i,j}$ where $w_{i,j} = a_i \cdot b_j + v_{i,j}$
4: Each $\mathcal{P}_i$ computes $c_i \leftarrow a_i \cdot b_i + \sum_{j \neq i}(\text{sent } v_{i,j} + \text{received } w_{j,i})$
return (a_i, b_i, c_i)

4.2.3 DaBits: Double Authenticated Random Bits

A **DaBit** (*Double Authenticated Bit*) is a cryptographic primitive used in secure multi-party computation (MPC) to bridge arithmetic and boolean circuits. It consists of a random bit $x \in \mathbb{F}_2$ and its corresponding value $\hat{x} \in \mathbb{F}$ in a finite field, such that $\hat{x} = x$ when both are opened. Each DaBit is secret-shared among the parties in both the binary and arithmetic domains.

Mathematical Formulation.

- Each party $\mathcal{P}_i$ holds shares $(x_{(i)}, \hat{x}_{(i)})$.
- The bit x is shared additively modulo 2: $x = \sum_i x_{(i)} \mod 2$
- The field value $\hat{x}$ is shared additively in $\mathbb{F}$: $\hat{x} = \sum_i \hat{x}_{(i)}$
- The shares are correlated so that $\hat{x} = x$

Table 5: DaBits

Component	Domain	Description
x	$\mathbb{F}_2$	Random bit, shared among parties
$\hat{x}$	$\mathbb{F}$	Field representation, correlated with x
$(x_{(i)}, \hat{x}_{(i)})$	Shares	Party i's shares in both domains

Functionalities. We define the DaBits functionality:

Functionality 4.17 $\mathcal{F}^{DaBits}_{\mathbb{F},M} \longrightarrow (\langle\!\langle \boldsymbol{b}_{\mathbb{F}_2} \rangle\!\rangle, \langle\!\langle \boldsymbol{b}_{\mathbb{F}} \rangle\!\rangle)$

Double-authenticated Bit generation (daBits) for M bits over field $\mathbb{F}$ and binary field $\mathbb{F}_2$ among parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_N\}$. Each daBit is authenticated both in the binary and arithmetic backends.

Players: N parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_N\}$.
Inputs: None (all values are randomly sampled).
Outputs: Each party $\mathcal{P}_i$ receives authenticated bit shares $\langle\!\langle \boldsymbol{b}_{\mathbb{F}_2 i} \rangle\!\rangle$ and authenticated field shares $\langle\!\langle \boldsymbol{b}_{\mathbb{F} i} \rangle\!\rangle$ s.t. for all $k \in [M]$:
$\sum_{j=1}^{N} (b_{(k)})_{\mathbb{F}_2 j} = \sum_{j=1}^{N} (b_{(k)})_{\mathbb{F} j} \in \{0, 1\}$, and
all MAC relations hold in both backends.

$\mathcal{F}^{DaBits}.\mathsf{Generate}() \dashrightarrow (\langle\!\langle \boldsymbol{b}_{\mathbb{F}_2} \rangle\!\rangle, \langle\!\langle \boldsymbol{b}_{\mathbb{F}} \rangle\!\rangle)$
Sample $\boldsymbol{b} \xleftarrow{\$} \{0,1\}^M$, generate authenticated additive shares in both $\mathbb{F}_2$ and $\mathbb{F}$ for each bit, send $(\langle\!\langle \boldsymbol{b}_{\mathbb{F}_2 i} \rangle\!\rangle, \langle\!\langle \boldsymbol{b}_{\mathbb{F} i} \rangle\!\rangle)$ to party $\mathcal{P}_i$.

Usage. DaBits enable efficient conversion between arithmetic and boolean operations in MPC protocols. They are essential for mixed-protocol computations, such as securely evaluating circuits that combine field operations and bitwise logic, or to move from one field to another (e.g., form a curve's base field to its scalar field).

Generation. DaBits are generated using a correlated randomness generation protocol adapted from [RW19].

RST19 DaBits Protocol. We implement the DaBits generation protocol from Rotaru-Soria-Tanguy 2019 [RW19], which efficiently generates double authenticated bits through a combination of authenticated triples and boolean operations.

Protocol 4.19 $\mathsf{DaBitsRST19}_N$ - [RST19] Malicious DaBit Generation

---

An N-party protocol realizing the $\mathcal{F}^{DaBits}_{\mathbb{F},M}$ functionality with malicious security. Generates double-authenticated bits (daBits) that are authenticated both in binary $\mathbb{F}_{2^\lambda}$ and arithmetic $\mathbb{F}$. Based on and improving upon [RST19].

Players: N parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_N\}$
Inputs: All parties $\rightarrow$ sid $\in \{0,1\}^*$ as session id
Outputs: All parties $\leftarrow$ daBits $\{\text{DaBit}_j\}_{j\in[M]}$ where each $\text{DaBit}_j = (\llbracket b_j \rrbracket_{\text{Gf2}}, \llbracket b_j \rrbracket_{\mathbb{F}})$

$\mathcal{P}_i$.GenerateTriples()--→triples
1: Generate authenticated triples in both $\mathbb{F}_{2^\lambda}$ and $\mathbb{F}$ using AuthTriples
2: Obtain binary triples $\llbracket (a_b, b_b, c_b) \rrbracket$ and arithmetic triples $\llbracket (a_a, b_a, c_a) \rrbracket$

$\mathcal{P}_i$.GenerateCandidates()--→candidate daBits
1: Each $\mathcal{P}_i$ samples random bits $\boldsymbol{r}_i \xleftarrow{\$} \{0,1\}^M$
2: Input bits into HCom in both fields to get $\llbracket \boldsymbol{r} \rrbracket_{\text{Gf2}}$ and $\llbracket \boldsymbol{r} \rrbracket_{\mathbb{F}}$
3: Generate candidate daBits from authenticated random bits

$\mathcal{P}_i$.BucketCutChoose()--→verified daBits
1: Use bucket verification with cut-and-choose:
2: Partition candidates into buckets of size B
3: Sample challenges $\xleftarrow{\$} \{0,1\}^*$ via coin-tossing
4: Open challenged daBits and verify consistency
5: Combine remaining daBits in each bucket via XOR
6: ABORT if any verification fails
return verified daBits

4.3 Additional Preprocessing Primitives

4.3.1 Homomorphic Commitments

Homomorphic Commitments (HCom) provide a way for parties to commit to values while maintaining the ability to perform homomorphic operations on the committed values. This primitive is essential for authenticated secret sharing schemes where parties need to verify the correctness of shared values without revealing the underlying secrets.

Functionalities. We define the homomorphic commitment functionality:

Functionality 4.18 $\mathcal{F}^{HCom}_{\mathbb{F},M}(\boldsymbol{x}) \longrightarrow \langle\!\langle \boldsymbol{x} \rangle\!\rangle$

Homomorphic Commitment functionality for M values over field $\mathbb{F}$ among parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_N\}$. Supports both random value generation and committing to chosen inputs.

Players: N parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_N\}$.
Inputs: $\mathcal{P}_i \leftarrow \boldsymbol{x}_i \in \mathbb{F}_0^M$ (optional), party $\mathcal{P}_i$'s input values.
Outputs: Each party $\mathcal{P}_j$ receives authenticated shares $\langle\!\langle \boldsymbol{x}_j \rangle\!\rangle$ of the committed values s.t. $\sum_{j=1}^{N} x_{(k)_j} = x_{(k)}$ for all $k \in [M]$, and all MAC relations hold.

$\mathcal{F}^{HCom}$.**Random**()$\dashrightarrow\langle\!\langle \boldsymbol{r} \rangle\!\rangle$
Sample $\boldsymbol{r} \xleftarrow{\$} \mathbb{F}^M$, generate authenticated additive shares and send $\langle\!\langle \boldsymbol{r}_i \rangle\!\rangle$ to each party $\mathcal{P}_i$.

$\mathcal{F}^{HCom}$.**Input**$(\boldsymbol{x}_i)\dashrightarrow\langle\!\langle \boldsymbol{x} \rangle\!\rangle$
Receive $\boldsymbol{x}_i$ from party $\mathcal{P}_i$, compute $\boldsymbol{x} = \boldsymbol{x}_i$, generate authenticated additive shares and send $\langle\!\langle \boldsymbol{x}_j \rangle\!\rangle$ to each party $\mathcal{P}_j$.

$\mathcal{F}^{HCom}$.**Open**$(\langle\!\langle \boldsymbol{y} \rangle\!\rangle)\dashrightarrow\boldsymbol{y}$
Receive authenticated shares $\langle\!\langle \boldsymbol{y}_i \rangle\!\rangle$ from each party $\mathcal{P}_i$, verify MAC relations, reconstruct $y_{(k)} = \sum_{i=1}^{N} y_{(k)_i}$ for all $k \in [M]$, send $\boldsymbol{y}$ to all parties.

Protocols.

BMRS24 Homomorphic Commitment. We implement the homomorphic commitment protocol from BMRS24 [BMRS24], which uses sVOLE-based MACs to provide authenticated secret sharing with homomorphic properties and malicious security.

Protocol 4.20 HCom - Homomorphic N-Party Commitment

An N-party protocol realizing the $\mathcal{F}^{HCom}_{\mathbb{F},M}$ functionality with malicious security. Provides authenticated secret sharing with homomorphic properties using sVOLE-based MACs. Based on [BMRS24].

Players: N parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_N\}$
Inputs: All parties $\rightarrow \mathrm{sid} \in \{0,1\}^*$ as session id
For Input: $\mathcal{P}_i \rightarrow \boldsymbol{x}_i \in \mathbb{F}_0^M$, input values
Outputs: All parties $\leftarrow$ authenticated shares $[\![\boldsymbol{x}]\!] = \{[\![x_{(j)}]\!]\}_{j\in[M]}$ with MACs

$\mathcal{P}_i$.Setup()$\dashrightarrow$MAC keys
0: Each party $\mathcal{P}_i$ samples global MAC key $\Delta_i \xleftarrow{\$} \mathbb{F}$
1: Run sVOLE between each pair $(\mathcal{P}_i, \mathcal{P}_j)$ to establish MAC correlation

$\mathcal{P}_i$.Input$(x_{(i)})\dashrightarrow[\![\boldsymbol{x}]\!]$
1: $\mathcal{P}_i$ secret-shares input: $\boldsymbol{x}_i \leftarrow \sum_j \boldsymbol{x}_j$
2: For each $\mathcal{P}_j$: $\mathcal{P}_i$ sends $\boldsymbol{x}_j$ to $\mathcal{P}_j$
3: Each $\mathcal{P}_j$ computes MAC: $\mathrm{mac}_j \leftarrow \Delta_j \cdot \boldsymbol{x}_j + \mathrm{sVOLE\_share}$
4: All parties store authenticated share $[\![\boldsymbol{x}]\!] = (\{\boldsymbol{x}_j\}_j, \{\mathrm{mac}_j\}_j)$
return $[\![\boldsymbol{x}]\!]$

$\mathcal{P}_i$.Random()$\dashrightarrow[\![\boldsymbol{r}]\!]$
1: Each $\mathcal{P}_i$ samples $\boldsymbol{r}_i \xleftarrow{\$} \mathbb{F}^M$
2: Compute MACs using sVOLE shares: $\mathrm{mac}_j \leftarrow \Delta_j \cdot \boldsymbol{r}_j + \mathrm{sVOLE\_share}$
3: All parties obtain $[\![\boldsymbol{r}]\!] = (\{\boldsymbol{r}_j\}_j, \{\mathrm{mac}_j\}_j)$
return $[\![\boldsymbol{r}]\!]$

$\mathcal{P}_i$.Open$([\![\boldsymbol{x}]\!])\dashrightarrow\boldsymbol{x}$
1: Each $\mathcal{P}_i$ broadcasts share $\boldsymbol{x}_i$
2: Each $\mathcal{P}_j$ verifies MAC: Check $\sum_i \mathrm{mac}_i \stackrel{?}{=} \Delta_j \cdot \sum_i \boldsymbol{x}_i$
3: ABORT if verification fails
4: Compute $\boldsymbol{x} \leftarrow \sum_i \boldsymbol{x}_i$
return $\boldsymbol{x}$

4.3.2 Correlated Oblivious Product Evaluation with Errors

Correlated Oblivious Product Evaluation with Errors (COPEe) is a two-party protocol that allows one party (the sender) to input a vector of subfield elements and another party (the receiver) to input a field element, such that the parties obtain correlated outputs satisfying a multiplicative relationship. This primitive is useful for generating authenticated shares and implementing efficient secure computation protocols over field extensions.

Functionalities. We define the COPEe functionality:

Functionality 4.19 $\mathcal{F}^{COPEe}_{\mathbb{F},M}(\boldsymbol{x}) \longrightarrow (\Delta, \boldsymbol{t}, \boldsymbol{q})$

Correlated Oblivious Product Evaluation with Errors functionality interacting with two parties Alice $\mathcal{P}_A$ and Bob $\mathcal{P}_B$.

Players: A sender $\mathcal{P}_A$, and a receiver $\mathcal{P}_B$.
Inputs: $\mathcal{P}_A \leftarrow \boldsymbol{x} \in \mathbb{F}_0^M$, a vector of subfield elements.
Outputs: $\mathcal{P}_A \leftarrow \boldsymbol{t} \in \mathbb{F}^M$, a vector of additive shares.
$\mathcal{P}_B \leftarrow (\Delta \in \mathbb{F}, \boldsymbol{q} \in \mathbb{F}^M)$, a field element and the additive shares.

$\mathcal{F}^{COPEe}$.**Sampling**()$\dashrightarrow \Delta, \boldsymbol{t}$
Sample and send $\Delta \xleftarrow{\$} \mathbb{F}$ to $\mathcal{P}_B$, sample and send $\boldsymbol{t} \xleftarrow{\$} \mathbb{F}^M$ to $\mathcal{P}_A$.

$\mathcal{F}^{COPEe}$.**Correlation**$(\boldsymbol{x})\dashrightarrow \boldsymbol{q}$
Receive $\boldsymbol{x}$ from $\mathcal{P}_A$, send $\boldsymbol{q}$ to $\mathcal{P}_B$ s.t. $t_{(i)} = x_{(i)} \cdot \Delta + q_{(i)} \quad \forall i \in [M]$.

Protocols.

COPEe Protocol. We implement the COPEe protocol based on Random OT Extension, which provides selective-failure security in field extensions. The protocol is particularly efficient when the subfield size provides sufficient statistical security.

Protocol 4.21 $\mathsf{COPEe}_{\mathbb{F},M}$

A two-party subfield Correlated Oblivious Product Evaluation with Errors protocol realizing the $\mathcal{F}^{COPEe}_{\mathbb{F},M}$ functionality with selective-failure security in a field extension $\mathbb{F}/\mathbb{F}_0$ where $|\mathbb{F}_0| > \kappa$ (statistical security parameter), based on a $\mathsf{ROTe}_{|\mathbb{F}|}$ (Random OT Extension) with batch-size $|\mathbb{F}|$ (bit-length of field elements).

Players: Alice $\mathcal{P}_A$, and Bob $\mathcal{P}_B$

Inputs: $\mathcal{P}_A \to \boldsymbol{x} \in \mathbb{F}_0^M$; $\mathcal{P}_B \to \Delta \in \mathbb{F}$; $\mathcal{P}_A\&\mathcal{P}_B \to \text{sid} \in \{0,1\}^*$ as session id

Outputs: $\mathcal{P}_A \leftarrow \boldsymbol{t} \in \mathbb{F}^M$ $\mathcal{P}_B \leftarrow \boldsymbol{q} \in \mathbb{F}^M$ s.t. $t_{(i)} = x_{(i)} \cdot \Delta + q_{(i)}$ $\forall i \in [M]$

$\mathcal{P}_A\&\mathcal{P}_B.\mathsf{Setup}()\dashrightarrow$

0: $\mathcal{P}_A\&\mathcal{P}_B$ run $\mathsf{ROTe.Setup}()$ to generate $|\mathbb{F}|$ Base OT seeds with $\mathcal{P}_A$ as sender $\mathcal{S}$ and $\mathcal{P}_B$ as receiver $\mathcal{R}$
1: $\mathcal{P}_B$ decomposes Δ into bits: $\boldsymbol{\beta} \leftarrow \{\beta_1, \beta_2, \ldots, \beta_{|\mathbb{F}|}\}$ where $\Delta = \sum_{j=1}^{|\mathbb{F}|} \beta_j \cdot 2^{j-1}$
2: $\mathcal{P}_B$ uses $\boldsymbol{\beta}$ as choice bits in ROTe
3: Set gadget vector $\boldsymbol{g} = \{1, G, G^2, \ldots, G^{|\mathbb{F}|-1}\} \in \mathbb{F}^{|\mathbb{F}|}$ where G is the generator of $\mathbb{F}$

$\mathcal{P}_A.\mathsf{Send}(\boldsymbol{x})\dashrightarrow(\boldsymbol{\tau}, \boldsymbol{t})$

1: Initialize $\boldsymbol{\tau} \leftarrow \{\}$ and $\boldsymbol{t} \leftarrow \boldsymbol{0}$
2: **for** $j \in [|\mathbb{F}|]$ **do**
3: Run ROTe to obtain random messages $(\boldsymbol{w_0^{(j)}}, \boldsymbol{w_1^{(j)}}) \leftarrow \text{PRG}(\text{sid}, k_j^0, k_j^1)$ where each $\boldsymbol{w_b^{(j)}} \in \mathbb{F}_0^M$
4: **for** $i \in [M]$ **do**
5: $\tau_{(j,i)} \leftarrow {w_0^{(j)}}_{(i)} - {w_1^{(j)}}_{(i)} - x_{(i)}$
6: $t_{(i)} \leftarrow t_{(i)} + g_{(j)} \cdot {w_0^{(j)}}_{(i)}$
7: $\mathsf{Send}(\boldsymbol{\tau}) \to \mathcal{P}_B$
return $\boldsymbol{t}$ as the sender output

$\mathcal{P}_B.\mathsf{Receive}(\boldsymbol{\tau})\dashrightarrow \boldsymbol{q}$

1: Initialize $\boldsymbol{q} \leftarrow \boldsymbol{0}$
2: **for** $j \in [|\mathbb{F}|]$ **do**
3: Run ROTe to obtain $\boldsymbol{w_{\beta_j}^{(j)}} \leftarrow \text{PRG}(\text{sid}, k_j^{\beta_j})$ where $\boldsymbol{w_{\beta_j}^{(j)}} \in \mathbb{F}_0^M$
4: **for** $i \in [M]$ **do**
5: $v_{j,i} \leftarrow {w_{\beta_j}^{(j)}}_{(i)} + \beta_j \cdot \tau_{(j,i)}$
6: $q_{(i)} \leftarrow q_{(i)} + g_{(j)} \cdot v_{j,i}$
return $\boldsymbol{q}$ as the receiver output

5 Online Phase Circuits

The online phase of Cerberus begins with the acquisition of private inputs. Leveraging the preprocessing phase to offload most of the computational load, it securely evaluates an arbitrarily long arithmetic circuit over a finite field $\mathbb{F}_p$ with as much as one round of communication per gate (certain operations can be performed locally without interaction). The online phase concludes with the reconstruction of the output. Figure 4 illustrates the main steps of the online phase.

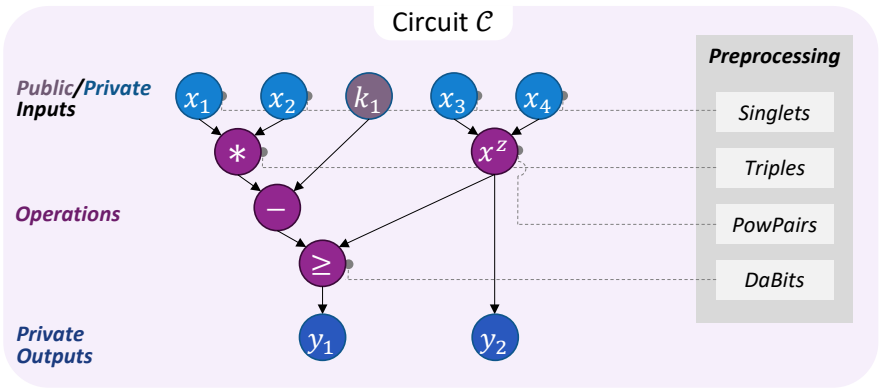

Figure 4: Online phase of Cerberus.

Supported backends. We currently support four different finite fields as backends for our online phase circuits:

- The prime field $\mathbb{F}_{2^{255}-19}$, corresponding to the base field of Curve25519 and used in the Ed25519 signature scheme [BDL+12], and supported by the Dalek cryptography library [dcp25].
- The prime field $\mathbb{F}_{2^{252}-277\ldots}$, the scalar field stemming from the cyclic subgroup of Curve25519.
- The binary field $\mathbb{F}_2$, used in conjunction with all other fields for bitwise and logical operations (e.g., bit decomposition, truncation).
- The faster Mersenne-107 field $\mathbb{F}_{2^{107}-1}$, used for fast arithmetic operations preserving full 64-bit precision with an extra leeway of 40 bits for statistical masking.

In all big prime cases, the pairwise MACs and keys are defined directly over the field $\mathbb{F}_p$. In the binary case, we define the pairwise MACs and keys over the extension field $\mathbb{F}_{2^{64}}$ to leverage efficient CPU operations on 64-bit words while still providing a sufficiently large statistical security parameter $\sigma = 64$.

The online phase is currently implemented by wrapping each circuit gate into its own asynchronous task, which are then scheduled and executed by a runtime (e.g., `Tokio`). This design allows for parallel execution of independent gates, maximizing resource utilization and minimizing latency. Future iterations may explore batching multiple gates into single tasks and composing them in layers of a graph to further optimize performance.

5.1 Supported Gates

We support a variety of arithmetic and logical gates to construct the circuits evaluated during the online phase, summarized in Table 6:

- **Local addition:** Parties can add their shares without interaction.
- **Local multiplication by public constant:** Multiplying a shared secret by a known field element is fully local.
- **Multiplication:** Secure multiplication of two shared secrets uses one round of communication and consumes one Beaver triple.
- **Zero testing:** A secret can be tested for zero, with the result revealed publicly, by consuming one singlet and one triple.
- **Multiplicative inverse:** Inversion is performed via a combination of singlets, triples, and zero testing.
- **Exponentiation (pow):** Repeated multiplication is computed via *Pow-Pairs*, a preprocessed resource optimized for exponentiation.
- **State encryption/decryption (scalar field only):** Shared values can be encrypted or decrypted using ElGamal over the scalar field.
- **Local multiplication by public point (scalar field only):** A scalar share can be locally multiplied by a known curve point.
- **Multiplication against point shares (scalar field only):** Multiplying a scalar share with a shared point requires one Beaver triple over the corresponding group.

5.2 Online Phase Functionality

The online phase is orchestrated by the $\mathcal{F}^{OnlinePhase}$ functionality, which manages the secure evaluation of arithmetic circuits:

Gate / Operation	$\mathbb{F}_{2^{255}-19}$	$\mathbb{F}_{2^{252}-277\ldots}$	$\mathbb{F}_2$	$\mathbb{F}_{2^{107}-1}$
Local addition	✓	✓	✓	✓
Local multiplication by public constant	✓	✓	✓	✓
Multiplication (1 round, consumes triple)	✓	✓	✓	✓
Zero testing (public reveal, singlets+triples)	✓	✓	✓	✓
Multiplicative inverse (singlets+triples+zero test)	✓	✓	✓	✓
Exponentiation (via powpairs)	✓	✓	✓	✓
State encryption / decryption (ElGamal)	–	✓	–	–
Local multiplication by public point	–	✓	–	–
Multiplication with point shares (consumes triple)	–	✓	–	–

Table 6: Gate support of CERBERUS over pairwise authenticated shares.

Functionality 5.20 $\mathcal{F}^{OnlinePhase}_{\mathbb{F},\mathbb{G}}$

Online phase functionality for secure computation over field $\mathbb{F}$ and elliptic curve group $\mathbb{G}$ among parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$. Maintains authenticated shares for field elements and curve points.

Players: n parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$.
Inputs: Session identifier $\text{sid} \in \{0,1\}^*$.
Outputs: Authenticated computation with integrity guarantees.

$\mathcal{F}^{OnlinePhase}$.**Input**$(x_1, \ldots, x_m)$--→$[\![x_i]\!]$
On input $(\text{INPUT}, \text{sid}, x_1, \ldots, x_m)$ from party $\mathcal{P}_j$ where $x_i \in \mathbb{F}$: generate authenticated shares $[\![x_i]\!]$ for each x_i, store $[\![x_i]\!]$ and output $(\text{SHARED}, \text{sid}, j, i)$ to all parties.

$\mathcal{F}^{OnlinePhase}$.**Add**$([\![x]\!], [\![y]\!])$--→$[\![z]\!]$
On input $(\text{ADD}, \text{sid}, [\![x]\!], [\![y]\!])$ from all parties: compute $z = x + y$, store $[\![z]\!]$ and output $(\text{ADDED}, \text{sid}, [\![z]\!])$ to all parties.

$\mathcal{F}^{OnlinePhase}$.**AddPlaintext**$([\![x]\!], c)$--→$[\![z]\!]$
On input $(\text{ADDPLAINTEXT}, \text{sid}, [\![x]\!], c)$ from all parties where $c \in \mathbb{F}$ is public: compute $z = x + c$, store $[\![z]\!]$ and output $(\text{ADDEDPLAINTEXT}, \text{sid}, [\![z]\!])$ to all parties.

$\mathcal{F}^{OnlinePhase}$.**Multiply**$([\![x]\!], [\![y]\!])$--→$[\![z]\!]$
On input $(\text{MULTIPLY}, \text{sid}, [\![x]\!], [\![y]\!])$ from all parties: compute $z = x \cdot y$, store $[\![z]\!]$ and output $(\text{MULTIPLIED}, \text{sid}, [\![z]\!])$ to all parties.

$\mathcal{F}^{OnlinePhase}$.**ScalarMul**$([\![x]\!], [\![P]\!])$--→$[\![Q]\!]$
On input $(\text{SCALARMUL}, \text{sid}, [\![x]\!], [\![P]\!])$ from all parties where $x \in \mathbb{F}$ and $P \in \mathbb{G}$: compute $Q = x \cdot P$, store $[\![Q]\!]$ and output $(\text{SCALARMULTIPLIED}, \text{sid}, [\![Q]\!])$ to all parties.

$\mathcal{F}^{OnlinePhase}$.**AddPoints**$([\![P]\!], [\![Q]\!])$--→$[\![R]\!]$
On input $(\text{ADDPOINTS}, \text{sid}, [\![P]\!], [\![Q]\!])$ from all parties where $P, Q \in \mathbb{G}$: compute $R = P + Q$, store $[\![R]\!]$ and output $(\text{POINTSADDED}, \text{sid}, [\![R]\!])$ to all parties.

$\mathcal{F}^{OnlinePhase}$.**Open**$([\![x]\!])$--→$x$
On input $(\text{OPEN}, \text{sid}, [\![x]\!])$ from all parties: reconstruct x from $[\![x]\!]$ and output $(\text{OPENED}, \text{sid}, x)$ to all parties.

$\mathcal{F}^{OnlinePhase}$.**ZeroTest**$([\![x]\!])$--→$b$
On input $(\text{ZEROTEST}, \text{sid}, [\![x]\!])$ from all parties: compute $b = 1$ if $x = 0$, otherwise $b = 0$, and output $(\text{ISZERO}, \text{sid}, b)$ to all parties.

$\mathcal{F}^{OnlinePhase}$.**Sum**$([\![x_1]\!], \ldots, [\![x_m]\!])$--→$[\![z]\!]$
On input $(\text{SUM}, \text{sid}, [\![x_1]\!], \ldots, [\![x_m]\!])$ from all parties: compute $z = \sum_{i=1}^{m} x_i$, store $[\![z]\!]$ and output $(\text{SUMMED}, \text{sid}, [\![z]\!])$ to all parties.

5.3 Circuit Operations

We now detail the specific circuit operations supported in our online phase:

5.3.1 Input Acquisition

Private inputs are securely acquired from parties and converted into authenticated shares for computation:

Protocol 5.22 Secret Input Distribution

Protocol realizing $\mathcal{F}^{OnlinePhase}$ input operation where one party distributes secret inputs as authenticated shares using singlets for masking.

Players: n parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, with inputter $\mathcal{P}_\ell$
Inputs: $\mathcal{P}_\ell \to x_1, \ldots, x_m \in \mathbb{F}$; All parties $\to$ singlets $\{[\![r_j]\!]\}_{j=1}^m$ from $\mathcal{F}^{Singlets}$
Outputs: All parties $\leftarrow \{[\![x_j]\!]\}_{j=1}^m$

$\mathcal{P}_i$ (receiver).ReceiveMask()$\dashrightarrow\{o_j\}_{j=1}^m$

1: **for** $j \in [m]$ **do**
2: Compute opening component $o_j \leftarrow$ component of $[\![r_j]\!]$
3: Send$(\{o_j\}_{j=1}^m) \to \mathcal{P}_\ell$

$\mathcal{P}_\ell$ (inputter).BroadcastMasked$(\{x_j\}_{j=1}^m)\dashrightarrow\{\epsilon_j\}_{j=1}^m$

1: Receive opening components $\{o_j^{(i)}\}$ from each $\mathcal{P}_{i\neq\ell}$
2: **for** $j \in [m]$ **do**
3: $r_j \leftarrow \text{Reconstruct}([\![r_j]\!], \{o_j^{(i)}\}_{i\neq\ell})$
4: $\epsilon_j \leftarrow x_j - r_j$
5: Broadcast $\{\epsilon_j\}_{j=1}^m$ to all parties

All parties.ComputeShares$(\{\epsilon_j\}_{j=1}^m)\dashrightarrow\{[\![x_j]\!]\}_{j=1}^m$

1: **for** $j \in [m]$ **do**
2: **If** $\mathcal{P}_i$ is the first party: $[\![x_j]\!] \leftarrow [\![r_j]\!] + \epsilon_j$
3: **Else**: $[\![x_j]\!] \leftarrow [\![r_j]\!]$
return $\{[\![x_j]\!]\}_{j=1}^m$

5.3.2 Addition Gate

Addition is performed locally by each party on their shares:

Protocol 5.23 Addition Operations - Field and Curve Point Addition

Local addition operations realizing $\mathcal{F}^{OnlinePhase}$ addition for field elements and elliptic curve points. All operations are non-interactive.

Players: n parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$
Inputs: Each party holds authenticated shares of inputs
Outputs: Authenticated shares of sum/result

FieldAdd$([\![x]\!], [\![y]\!]) \dashrightarrow [\![z]\!]$
1: Each party locally computes $[\![z]\!] \leftarrow [\![x]\!] + [\![y]\!]$
return $[\![z]\!]$ where $z = x + y$

AddPlaintext$([\![x]\!], c) \dashrightarrow [\![z]\!]$
2: **If** $\mathcal{P}_i$ is the first party: $[\![z]\!] \leftarrow [\![x]\!] + c$
3: **Else**: $[\![z]\!] \leftarrow [\![x]\!]$
return $[\![z]\!]$ where $z = x + c$

Sum$([\![x_1]\!], \ldots, [\![x_m]\!]) \dashrightarrow [\![z]\!]$
4: Each party locally computes $[\![z]\!] \leftarrow \sum_{j=1}^{m} [\![x_j]\!]$
return $[\![z]\!]$ where $z = \sum_{j=1}^{m} x_j$

CurveAdd$([\![P]\!], [\![Q]\!]) \dashrightarrow [\![R]\!]$
5: Each party locally computes $[\![R]\!] \leftarrow [\![P]\!] + [\![Q]\!]$ using curve addition
return $[\![R]\!]$ where $R = P + Q$

AddPlaintextPoint$([\![P]\!], Q) \dashrightarrow [\![R]\!]$
6: **If** $\mathcal{P}_i$ is the first party: $[\![R]\!] \leftarrow [\![P]\!] + Q$
7: **Else**: $[\![R]\!] \leftarrow [\![P]\!]$
return $[\![R]\!]$ where $R = P + Q$

5.3.3 Field Multiplication Gate

Secure multiplication over finite fields uses Beaver triples for efficiency:

Protocol 5.24 Field Multiplication - Beaver Triple-based Multiplication

Protocol realizing $\mathcal{F}^{OnlinePhase}$ field multiplication using Beaver triples. Requires one triple and two openings per multiplication.

Players: n parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$
Inputs: All parties $\rightarrow [\![x]\!], [\![y]\!]$; Triple $([\![a]\!], [\![b]\!], [\![c]\!])$ from $\mathcal{F}^{AuthTriples}$ where $c = a \cdot b$
Outputs: All parties $\leftarrow [\![z]\!]$ where $z = x \cdot y$

All parties.Multiply$([\![x]\!], [\![y]\!])$--→$[\![z]\!]$
1: Obtain fresh triple $([\![a]\!], [\![b]\!], [\![c]\!])$ from $\mathcal{F}^{AuthTriples}$
2: Locally compute $[\![\epsilon]\!] \leftarrow [\![x]\!] - [\![a]\!]$
3: Locally compute $[\![\delta]\!] \leftarrow [\![y]\!] - [\![b]\!]$
4: Jointly open $[\![\epsilon]\!]$ to obtain $\epsilon = x - a$
5: Jointly open $[\![\delta]\!]$ to obtain $\delta = y - b$
6: Locally compute $[\![z]\!] \leftarrow [\![c]\!] + \delta \cdot [\![x]\!] + \epsilon \cdot [\![b]\!] + \epsilon \cdot \delta$
return $[\![z]\!]$ where $z = x \cdot y$

5.3.4 Curve Point Multiplication Gate

For scalar field computations, we support multiplication against elliptic curve points:

Protocol 5.25 Scalar-Point Multiplication - Curve Point Multiplication via Adapted Triples

Protocol realizing $\mathcal{F}^{OnlinePhase}$ scalar-point multiplication using adapted Beaver triples. Requires one field triple adapted to curve points.

Players: n parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$
Inputs: All parties $\rightarrow [\![x]\!] \in \mathbb{F}, [\![P]\!] \in \mathbb{G}$; Triple $([\![a]\!], [\![b]\!], [\![c]\!])$ from $\mathcal{F}^{AuthTriples}$ where $c = a \cdot b$; Generator $G \in \mathbb{G}$
Outputs: All parties $\leftarrow [\![Q]\!]$ where $Q = x \cdot P$

All parties.ScalarMul$([\![x]\!], [\![P]\!])$--→$[\![Q]\!]$
1: Obtain fresh triple $([\![a]\!], [\![b]\!], [\![c]\!])$ from $\mathcal{F}^{AuthTriples}$
2: Locally compute $[\![U]\!] \leftarrow [\![b]\!] \cdot G$ (adapt to curve)
3: Locally compute $[\![V]\!] \leftarrow [\![c]\!] \cdot G$ (adapt to curve)
4: Locally compute $[\![s]\!] \leftarrow [\![x]\!] - [\![a]\!]$ (scalar mask)
5: Locally compute $[\![T]\!] \leftarrow [\![P]\!] - [\![U]\!]$ (point mask)
6: Jointly open $[\![s]\!]$ to obtain $s = x - a$
7: Jointly open $[\![T]\!]$ to obtain $T = P - U$
8: Locally compute $[\![Q]\!] \leftarrow [\![V]\!] + s \cdot [\![P]\!] + [\![a]\!] \cdot T + s \cdot T$
return $[\![Q]\!]$ where $Q = x \cdot P$

5.3.5 Opening/Reconstruction Gate

Authenticated shares are opened to reveal the computed result:

Protocol 5.26 Open - Opening Secret Shares

Protocol realizing $\mathcal{F}^{OnlinePhase}$ open operation to reconstruct shared values.

Players: n parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$
Inputs: All parties $\rightarrow [\![x]\!]$ (field element $x \in \mathbb{F}$ or curve point $P \in \mathbb{G}$)
Outputs: All parties $\leftarrow x$ (reconstructed plaintext value)

All parties.$\mathsf{Open}([\![x]\!])\text{--}\!\rightarrow x$

1: Each party sends their share component to all other parties
2: Each party receives share components from all other parties
3: Locally reconstruct $x \leftarrow$ combine all n share components
return x

Correctness: The reconstruction algorithm guarantees that if all parties hold valid shares $[\![x]\!]$ of the same secret, combining all shares yields the original secret x.

Batch Operation: For batches of m shared values $\{[\![x_i]\!]\}_{i=1}^{m}$, parties send all m share components in a single message, receive from all others, and reconstruct all m values in parallel.

Security: Opening reveals the plaintext value to all parties. This operation should only be used when the protocol explicitly requires revealing the value (e.g., for output, intermediate checks, or computing masked values in secure multiplication).

5.3.6 Zero Test Gate

Testing whether a shared value is zero without revealing its actual value:

Protocol 5.27 Zero Test - Probabilistic Zero Test

Protocol realizing $\mathcal{F}^{OnlinePhase}$ zero test operation to determine if shared field elements are zero, with output revealed to all parties.

Players: n parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$
Inputs: All parties $\rightarrow [\![x]\!] \in \mathbb{F}$; Singlet $[\![r]\!]$ from $\mathcal{F}^{Singlets}$; Triple $([\![a]\!], [\![b]\!], [\![c]\!])$ from $\mathcal{F}^{AuthTriples}$
Outputs: All parties $\leftarrow b \in \{0, 1\}$ where $b = 1$ if $x = 0$, else $b = 0$

All parties.$\mathsf{ZeroTest}([\![x]\!])\text{--}\!\rightarrow b$

1: Obtain fresh singlet $[\![r]\!]$ from $\mathcal{F}^{Singlets}$ where $r \in_R \mathbb{F}$
2: Obtain fresh triple $([\![a]\!], [\![b]\!], [\![c]\!])$ from $\mathcal{F}^{AuthTriples}$
3: Run secure multiplication to compute $[\![z]\!] \leftarrow [\![x]\!] \cdot [\![r]\!]$
4: Jointly open $[\![z]\!]$ to obtain $z = x \cdot r$
5: Locally compute $b \leftarrow 1$ if $z = 0$, else $b \leftarrow 0$
return b

Correctness: If $x = 0$, then $z = x \cdot r = 0$ regardless of r, so the test correctly outputs 1. If $x \neq 0$ and r is uniformly random in $\mathbb{F}$, then $z = x \cdot r$ is uniformly random and non-zero except with probability $1/|\mathbb{F}|$.

Security: The value x remains hidden since only $z = x \cdot r$ is revealed, where r is a uniformly random mask. The output b reveals whether x is zero or not.

Failure Probability: False positive probability of $1/|\mathbb{F}|$ per element tested (negligible for cryptographic field sizes).

Batch Operation: For batches of m zero tests $\{[\![x_i]\!]\}_{i=1}^m$, obtain m singlets and triples, compute all m products in parallel, open concurrently, and output $\{b_i\}_{i=1}^m$.

6 Architecture of Cerberus

We now provide an overview of the architecture of Cerberus, illustrated in Figure 5, displaying the interaction between the preprocessing and online phases. The preprocessing phase is responsible for generating the correlated randomness required for the efficient execution of the online phase, which securely evaluates arithmetic circuits over a finite field $\mathbb{F}_p$ with minimal communication per gate. The online phase begins with the acquisition of private inputs, followed by the secure evaluation of the circuit, and concludes with the reconstruction of the output.

The preprocessing phase is executed independently for each supported backend, and its output is stored in a local database for later retrieval during the online phase. The online phase can then be executed employing all of the supported backends at once, consuming the appropriate preprocessed data for each gate in the circuit.

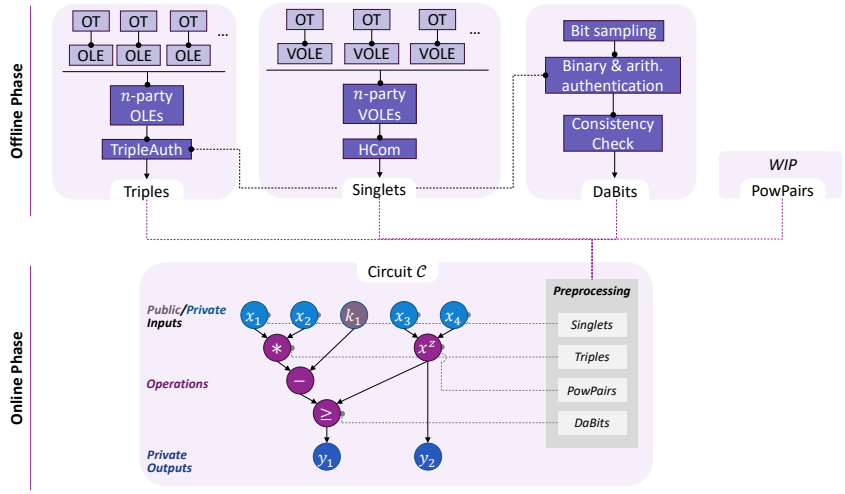

Figure 5: Offline & Online phase of Cerberus.

References

[AG11] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In *International Colloquium on Automata, Languages, and Programming*, pages 403–415. Springer, 2011.

[B+08] Daniel J Bernstein et al. Chacha, a variant of salsa20. In *Workshop record of SASC*, volume 8, pages 3–5. Lausanne, Switzerland, 2008.

[BCG+19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent ot extension and more. In *Annual International Cryptology Conference*, pages 489–518. Springer, 2019.

[BDL+12] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of cryptographic engineering*, 2:77–89, 2012.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 169–188. Springer, 2011.

[Bea91] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Annual international cryptology conference*, pages 420–432. Springer, 1991.

[BGG19] Sean Bowe, Ariel Gabizon, and Matthew D Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. In *Financial Cryptography and Data Security: FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curaçao, March 2, 2018, Revised Selected Papers 22*, pages 64–77. Springer, 2019.

[BKW03] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *Journal of the ACM (JACM)*, 50(4):506–519, 2003.

[Blu83] Manuel Blum. Coin flipping by telephone a protocol for solving impossible problems. *ACM SIGACT News*, 15(1):23–27, 1983.

[BM88] Mihir Bellare and Silvio Micali. How to sign given any trapdoor function. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 32–42, 1988.

[BMRS24] Carsten Baum, Nikolas Melissaris, Rahul Rachuri, and Peter Scholl. Cheater identification on a budget: Mpc with identifiable abort from pairwise macs. In *Annual International Cryptology Conference*, pages 454–488. Springer, 2024.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *20th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–10. ACM, 1988.

[BOO15] Amos Beimel, Eran Omri, and Ilan Orlov. Protocols for multiparty coin toss with a dishonest majority. *Journal of Cryptology*, 28(3):551–600, 2015.

[BPW12] David Bernhard, Olivier Pereira, and Bogdan Warinschi. How not to prove yourself: Pitfalls of the fiat-shamir heuristic and applications to helios. In *Advances in Cryptology–ASIACRYPT 2012: 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings 18*, pages 626–643. Springer, 2012.

[BSCG+15] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *2015 IEEE Symposium on Security and Privacy*, pages 287–304. IEEE, 2015.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols. In *20th Annual ACM Symposium on Theory of Computing (STOC)*, pages 11–19. ACM, 1988.

[Cle86a] Richard Cleve. Limits on the security of coin flips when half the processors are faulty. In *18th Annual ACM Symposium on Theory of Computing (STOC)*, pages 364–369. ACM, 1986.

[Cle86b] Richard Cleve. Limits on the security of coin flips when half the processors are faulty. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 364–369, 1986.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 494–503, 2002.

[CMB23] Kevin Choi, Aathira Manoj, and Joseph Bonneau. Sok: Distributed randomness beacons. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 75–92. IEEE, 2023.

[CP92] David Chaum and Torben Pryds Pedersen. Wallet databases with observers. In *Annual international cryptology conference*, pages 89–105. Springer, 1992.

[CPS+16] Michele Ciampi, Giuseppe Persiano, Alessandra Scafuro, Luisa Siniscalchi, and Ivan Visconti. Improved or-composition of sigma-protocols. In *Theory of Cryptography: 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II 13*, pages 112–141. Springer, 2016.

[CY24] Alessandro Chiesa and Eylon Yogev. *Building Cryptographic Proofs from Hash Functions*. Self-published, 2024.

[Dam02] Ivan Damgård. On σ-protocols. *Lecture Notes, University of Aarhus, Department for Computer Science*, 84, 2002.

[Dan15] Quynh Dang. Secure hash standard, 2015-08-04 2015.

[dcp25] The dalek-cryptography project. curve25519-dalek: A pure-rust implementation of group operations on ristretto and curve25519. https://github.com/dalek-cryptography/curve25519-dalek, 2025.

[dV] Henry de Valence. Merlin transcripts.

[Dwo15] Morris Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions, 2015-08-04 2015.

[FHSS+23] Armando Faz-Hernandez, Sam Scott, Nick Sullivan, Riad S. Wahby, and Christopher A. Wood. Hashing to Elliptic Curves. RFC 9380, August 2023.

[Fis05] Marc Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In *Annual International Cryptology Conference*, pages 152–168. Springer, 2005.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*, pages 186–194. Springer, 1986.

[GB08] Shafi Goldwasser and Mihir Bellare. Lecture notes on cryptography. *Summer course "Cryptography and computer security" at MIT*, 2008.

[GKWY20] Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. Efficient and secure multiparty computation from fixed-key block ciphers. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 825–841. IEEE, 2020.

[GL02] Shafi Goldwasser and Yehuda Lindell. Secure computation without agreement. Cryptology ePrint Archive, Paper 2002/040, 2002. https://eprint.iacr.org/2002/040.

[GLSY04] Rosario Gennaro, Darren Leigh, Ravi Sundaram, and William Yerazunis. Batching schnorr identification scheme with applications to privacy-preserving authorization and low-bandwidth communication devices. In *Advances in Cryptology-ASIACRYPT 2004: 10th International Conference on the Theory and Application of Cryptology and Information Security, Jeju Island, Korea, December 5-9, 2004. Proceedings 10*, pages 276–292. Springer, 2004.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on computing*, 17(2):281–308, 1988.

[GMR19] Shafi Goldwasser, Silvio Micali, and Chales Rackoff. The knowledge complexity of interactive proof-systems, 2019.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229. ACM, 1987.

[GMY83] Shafi Goldwasser, Silvio Micali, and Andy Yao. Strong signature schemes. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 431–439, 1983.

[Ham17] Mike Hamburg. The strobe protocol framework. Cryptology ePrint Archive, Paper 2017/003, 2017. https://eprint.iacr.org/2017/003.

[HLPT20] Thomas Haines, Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. How not to prove your election outcome. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 644–660. IEEE, 2020.

[IOZ12] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Identifying cheaters without an honest majority. In *9th Theory of Cryptography Conference (TCC)*, volume 7194 of *LNCS*, pages 21–38. Springer, 2012.

[JL17] S. Josefsson and I. Liusvaara. Edwards-curve digital signature algorithm (eddsa). RFC 8032, RFC Editor, 2017. https://www.rfc-editor.org/rfc/rfc8032.

[KL07] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography: principles and protocols*. Chapman and hall/CRC, 2007.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure ot extension with optimal overhead. In *Annual Cryptology Conference*, pages 724–741. Springer, 2015.

[KS22] Yashvanth Kondi and Abhi Shelat. Improved straight-line extraction in the random oracle model with applications to signature aggregation. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 279–309. Springer, 2022.

[Lin17] Yehuda Lindell. How to simulate it–a tutorial on the simulation proof technique. *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, pages 277–346, 2017.

[LLX+26] Zhe Li, Hongqing Liu, Chaoping Xing, Yizhou Yao, and Chen Yuan. Faster pseudorandom correlation generators via walsh-hadamard transform. *Cryptology ePrint Archive*, 2026.

[LS07] Pierre L'ecuyer and Richard Simard. Testu01: Ac library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)*, 33(4):1–40, 2007.

[Mar08] George Marsaglia. The marsaglia random number cdrom including the diehard battery of tests of randomness. *http://www. stat. fsu. edu/pub/diehard/*, 2008.

[Mie19] Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model, 2019.

[MR19] Daniel Mansy and Peter Rindal. Endemic oblivious transfer. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 309–326, 2019.

[MRR21a] Ian McQuoid, Mike Rosulek, and Lawrence Roy. Batching base oblivious transfers. In *Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part III 27*, pages 281–310. Springer, 2021.

[MRR21b] Ian McQuoid, Mike Rosulek, and Lawrence Roy. Batching base oblivious transfers. In *Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part III 27*, pages 281–310. Springer, 2021.

[NJ16] Yoav Nir and Simon Josefsson. Curve25519 and Curve448 for the Internet Key Exchange Protocol Version 2 (IKEv2) Key Agreement. RFC 8031, December 2016.

[ONAZ] Jack OConnor, Samuel Neves, Jean-Philippe Aumasson, and Zooko. Blake3: one function, fast everywhere.

[Ped91] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Annual international cryptology conference*, pages 129–140. Springer, 1991.

[Por13] Thomas Pornin. Rfc 6979: Deterministic usage of the digital signature algorithm (dsa) and elliptic curve digital signature algorithm (ecdsa), 2013.

[Rom90] John Rompel. One-way functions are necessary and sufficient for secure signatures. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 387–394, 1990.

[Roy22] Lawrence Roy. Softspokenot: Quieter ot extension from small-field silent vole in the minicrypt model. In *Advances in Cryptology–CRYPTO 2022: 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15–18, 2022, Proceedings, Part I*, pages 657–687. Springer, 2022.

[RW19] Dragos Rotaru and Tim Wood. Marbled circuits: Mixing arithmetic and boolean circuits with active security. In *International Conference on Cryptology in India*, pages 227–249. Springer, 2019.

[SA15] Markku-Juhani O. Saarinen and Jean-Philippe Aumasson. The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC), November 2015.

[Sch91] C. P. Schnorr. Efficient signature generation by smart cards. In *Journal of Cryptology*, 1991.

[Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[SPDZ12] Nigel Smart, Valerio Pastro, Ivan Damgård, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual cryptology conference*, pages 643–662. Springer, 2012.

[WAS22] Ke Wu, Gilad Asharov, and Elaine Shi. A complete characterization of game-theoretically fair, multi-party coin toss. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 120–149. Springer, 2022.

[WNR18] Pieter Wuille, Jonas Nick, and Tim Ruffing. Schnorr signatures for secp256k1, 2018.

[WNT19] Pieter Wuille, Jonas Nick, and Anthony Towns. Taproot: Segwit version 1 spending rules, 2019.

[Yao82] Andrew C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 160–164. IEEE, 1982.

[YWZ20] Kang Yang, Xiao Wang, and Jiang Zhang. More efficient mpc from improved triple generation and authenticated garbling. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1627–1646, 2020.

[Zil] Zilliqa. The zilliqa technical whitepaper.

A Appendix

A.1 An overview on Elliptic Curves

An elliptic curve $\mathsf{E}(\mathbb{G}, G, q)$ is a non-singular[16] projective[17] algebraic curve by the solutions to an equation in two variables (x and y) and a field $\mathbb{F}_{p^k}$. An elliptic curve equation in the context of Elliptic Curve Cryptography (ECC) takes one of several standard forms defined by two non-zero parameters. The most common forms[18] are:

- *Weierstrass*: $y^2 = x^3 + ax + b$, for parameters a and b where $4a^3 + 27b^2 \neq 0$.
- *Montgomery*: $by^2 = x^3 + ax^2 + x$ for parameters A and B where $(a^2 - 4)/b^2 \neq 0$ and non-square in $\mathbb{F}$.
- *Edwards*: $x^2 + y^2 = 1 + dx^2y^2$ for parameter d where $d \notin \{0, 1\}$.
- *Twisted Edwards*: $ax^2 + y^2 = 1 + dx^2y^2$ for parameters a and d where $a \neq 0$ and $d \neq 0$.

For ECC, $\mathbb{F}$ is a finite field $GF(p^k)$ of prime characteristic[19] $p > 3$. In most cases, $\mathbb{F}$ is a prime field ($k = 1$). Otherwise, $\mathbb{F}$ is an extension field ($k > 1$). A curve $\mathsf{E}(\mathbb{G}, G, q)$ induces an algebraic group of order q, defined with:

- A set of q distinct elements, where each element is a curve point satisfying the curve equation with affine coordinates (x, y) with $x, y \in \mathbb{F}$.
- A group operation $+ : \mathbb{G} \times \mathbb{G} \to \mathbb{G}$, represented as addition without loss of generality, taking two group elements and producing a group element. Indirectly, it forms a scalar multiplication operation $\cdot : \mathbb{Z}_q \times \mathbb{G} \to \mathbb{G}$ defined as a repeated application of the group operation on an element.
- A distinguished element I, called the identity point, which acts as the identity element for the group operation ($I + P = P \ \ \forall\, P \in \mathbb{G}$).

For security reasons, cryptographic applications of elliptic curves generally require using a sub-group $\mathbb{G}$ of prime order q', where $q = c \cdot q'$. In this equation, c is an integer called the *cofactor*. An algorithm that takes as input an arbitrary point on the curve E and produces as output a point in the subgroup $\mathbb{G}$ of E is said to *clear the cofactor* (a.k.a. an injective mapping to the prime-order sub-group). We stress that careful consideration should be made to exchange curves for another, as not all curves are isomorphic[20] to each other.

[16] Smooth, contains no *singular* points without properly defined derivatives.
[17] A curve that can be defined over projections on higher dimension spaces.
[18] it is often possible to map one form to another
[19] You obtain the additive identity (e.g., 0) when you add p times the multiplicative identity (e.g., 1).
[20] There exists bijective mapping between two elliptic curves that preserves the group structure. A *twist* is a special case of these mappings, applicable between curves such as *curve25519* [NJ16] and *ed25519* [JL17].

Supported curves. Our MPC implementation targets *curve25519* [JL17], a 255-bit EC with a twist, designed to be efficient[21] at the cost of employing a group of composite order ($c \neq 1$), hence requiring cofactor clearing to operate in the prime subgroup. Not as widely used as K256 or P256, it is the reference in the EdDSA signature scheme.

[21] Its group order is very close to a power of two, allowing seamless conversions between uniform bit-strings and group elements with very low bias.