Earn While You Reveal: Private Set Intersection that Rewards Participants

Abstract. In Private Set Intersection protocols (PSIs), a non-empty result always reveals something about the private input sets of the parties. Moreover, in various variants of PSI, not all parties necessarily receive or are interested in the result. Nevertheless, to date, the literature has assumed that those parties who do not receive or are not interested in the result still contribute their private input sets to the PSI for free, although doing so would cost them their privacy. In this work, for the first time, we propose a multi-party PSI, called "Anesidora", that *rewards* parties who contribute their private input sets to the protocol. Anesidora is efficient; it mainly relies on symmetric key primitives and its computation and communication complexities are linear with the number of parties and set cardinality. It remains secure even if the majority of parties are corrupted by active colluding adversaries.

1 Introduction

Private Set Intersection (PSI) is a protocol which allows two parties to jointly compute the intersection of their private sets without revealing anything beyond the result. PSI has numerous applications. For instance, it has been used in Vertical Federated Learning (VFL) [32], COVID-19 contact tracing schemes [18], remote diagnostics [9], and finding leaked credentials [39].

There exist two facts about PSIs: (i) a non-empty result always reveals something about the parties' private input sets (i.e., the set elements that are in the intersection), and (ii) various variants of PSIs do not output the result to all parties, even in those PSIs that do, not all of the parties are necessarily interested in it. Given these facts, one may ask a natural question:

How can we incentivise the parties that do not receive the result or are not interested in it to participate in a PSI?

In this work, we answer the above question for the first time. We present a multi-party PSI, called "Anesidora", that allows a buyer who initiates the PSI computation (and is interested in the result) to pay other parties proportionate to the number of elements it learns about other parties' private inputs. Anesidora is efficient and mainly relies on symmetric key primitives. Its computation and communication complexities are linear with the number of parties and set cardinality. Anesidora remains secure even if the majority of parties are corrupt by active adversaries which may collude with each other.

We develop Anesidora in a modular fashion. First, we define the notion of "PSI with Fair Compensation" ($\mathcal{PSI}^{\mathcal{FC}}$) and devise the first construction, called

"Justitia", that realises the notion. $\mathcal{PSI}^{\mathcal{FC}}$ ensures that either all parties get the result or if the protocol aborts in an unfair manner (where only dishonest parties learn the result), then honest parties will receive financial compensation. Next, we upgrade $\mathcal{PSI}^{\mathcal{FC}}$ to the notion of "PSI with Fair Compensation and Reward" ($\mathcal{PSI}^{\mathcal{FCR}}$) and develop Anesidora that realises $\mathcal{PSI}^{\mathcal{FCR}}$. The latter notion ensures that honest parties (a) are rewarded regardless of whether all parties are honest, or a set of them aborts in an unfair manner and (b) are compensated in the case of an unfair abort. We formally prove the two PSIs using the simulation-based model. To devise efficient PSIs, we have developed a primitive, called "unforgeable polynomial" that might be of independent interest.

A PSI, like Anesidora, that supports more than two parties and rewards set contributors can create opportunities for much richer analytics and incentivise parties to participate. It can be used (1) by an advertiser who wants to conduct advertisements targeted at certain customers by first finding their common shopping patterns distributed across different e-commerce companies' databases [24], (2) by a malware detection service that allows a party to send a query to a collection of malware databases held by independent antivirus companies to find out whether all of them consider a certain application as malware [38], or (3) by a bank, like "WeBank", that uses VFL and PSI to gather information about certain customers from various partners (e.g., national electronic invoice and other financial institutions) to improve its risk management of loans [11]. The set contributors will be rewarded by such a PSI in all these cases.

We hope that our work initiates future research on developing reward mechanisms for participants of *generic secure Multi-Party Computation (MPC)*. Such reward mechanisms have the potential to increase MPC's real-world adoption.

2 Related Work

Since their introduction in [19], various PSIs have been designed. PSIs can be divided into *traditional* and *delegated* ones. In *traditional* PSIs, data owners interactively compute the result using their local data. Very recently, Raghuraman and Rindal [37] proposed two two-party PSIs, one secure against semi-honest/passive and the other against malicious/active adversaries. To date, these two protocols are the fastest two-party PSIs. They rely on Oblivious Key-Value Stores (OKVS) and Vector Oblivious Linear Evaluation. Their computation cost is $O(c)$, where c is a set's cardinality. They impose $O(c \log c^2 + \kappa)$ and $O(c \cdot \kappa)$ communication costs in the semi-honest and malicious security models respectively, where l is a set element's bit-size and κ is a security parameter.

Also, researchers designed PSIs that let multiple (i.e., more than two) parties efficiently compute the intersection. The multi-party PSIs in [23,31] are secure against passive adversaries while those in [5,21,41,31,35] were designed to resist active ones. To date, the protocols in [31] and [35] are the most efficient multi-party PSIs designed to be secure against passive and active adversaries respectively. The two remain secure even if the majority of parties are corrupt. The former relies on symmetric key primitives such as Programmable Pseudorandom Function (OPPRF), while the latter mainly uses OPPRF and OKVS.

The computation and communication complexities of the PSI in [31] are $O(c \cdot m^2 + c \cdot m)$ and $O(c \cdot m^2)$ respectively, where m is the number of clients. Later, to achieve efficiency, Chandran *et al.* [10] proposed a multi-party PSI that remains secure only if the minority of the parties is corrupt by a semi-honest adversary. The PSI in [35] has a parameter t that determines how many parties can collude with each other and must be set before the protocol's execution, where $t \in \{2, m\}$. The protocol divides the parties into three groups, clients: $A_1, ..A_{m-t-1}$, leader: A_{m-t}, and servers: $A_{m-t+1}, ..A_m$. Each client needs to send a set of messages to every server and the leader which jointly compute the final result. Hence, this protocol's overall computation and communication complexities are $O(c \cdot \kappa(m + t^2 - t(m + 1)))$ and $O(c \cdot m \cdot \kappa)$ respectively.

Dong *et al.* proposed a "fair" two-party PSI [15] that ensures either both parties receive the result or neither does, even if a malicious party aborts prematurely during the protocol's execution. It uses homomorphic encryption, zero-knowledge proofs, and polynomial representation of sets. The protocol's computation and communication complexities are $O(c^2)$ and $O(c)$ respectively. Since then, various fair two-party PSIs have been proposed, e.g., in [12,14,13]. To date, the fair PSI in [13] is the most efficient one. It uses ElGamal encryption, verifiable encryption, and zero-knowledge proofs, which often impose a significant overhead. Its computation and communication cost is $O(c)$. So far, there exists no fair *multi-party* PSI in the literature. Our Justitia is the first one.

Delegated PSIs use cloud computing for computation and/or storage, while preserving the privacy of the computation inputs and outputs from the cloud. They can be divided further into protocols that support *one-off* and *repeated* delegation of PSI computation. The former like [25,27,42] cannot reuse their outsourced encrypted data and require clients to re-encode their data locally for each computation. The most efficient such protocol is [25], which has been designed for the two-party setting and its computation and communication complexity is $O(c)$. Those protocols that support repeated PSI delegation let clients outsource the storage of their encrypted data to the cloud only once, and then execute an unlimited number of computations on the outsourced data. To date, the protocol in [1] is the most efficient PSI that supports repeated delegation in the semi-honest model. It relies on the polynomial representation of sets, pseudorandom function, and hash table. Its communication and computation complexities are $O(h \cdot d^2)$ and $O(h \cdot d)$ respectively, where h is the total number of bins in the hash table, d is a bin's capacity (often $d = 100$), and $h \cdot d$ is linear with c. Recently, a multi-party PSI that supports repeated delegation and efficient *updates* has been proposed in [2]. It lets a party efficiently update its outsourced set securely. It is also in the semi-honest model and uses a pseudorandom function, hash table, and Bloom filter. It imposes $O(h \cdot d^2 \cdot m)$ and $O(h \cdot d \cdot m)$ computation and communication costs respectively, during the PSI computation. It also imposes $O(d^2)$ computation and communication overheads, during the update phase.

3 Notations and Preliminaries

3.1 Notations

Table 1 summarises the main notations used in this paper.

Table 1: Notation Table.

Setting	Symbol	Description
Generic	CL	Set of all clients, $\{A_1, ..., A_m, D\}$
	D	Dealer client
	A_m	Buyer client
	m	Total number of clients (excluding D)
	p	Large prime number
	$\mathtt{H}$	Hash function
	$\lvert S_\cap \rvert$	Intersection size
	S_{min}	Smallest set's size
	S_{max}	Largest set's size
	$\mid$	Divisible
	$\setminus$	Set subtraction
	c	Set's cardinality
	h	Total number of bins in a hash table
	d	A bin's capacity
	λ	Security parameter
	$\mathtt{OLE}$	Oblivious Linear Evaluation
	$\mathtt{OLE}^+$	Advanced $\mathtt{OLE}$
	$\mathtt{Com}$	Commitment algorithm of commitment
	$\mathtt{Ver}$	Verification algorithm of commitment
	$\mathtt{MT.genTree}$	Tree construction algorithm of Merkle tree
	$\mathtt{MT.prove}$	Proof generation algorithm of Merkle tree
	$\mathtt{MT.verify}$	Verification algorithm of Merkle tree
	$\mathtt{CT}$	Coin tossing protocol
	$\mathtt{VOPR}$	Verifiable Oblivious Poly. Randomization
	$\mathtt{ZSPA}$	Zero-sum Pseudorandom Values Agreement
	$\mathtt{ZSPA\text{-}A}$	$\mathtt{ZSPA}$ with an External Auditor
	$\mathtt{JUS}$	Protocol that realises $\mathcal{PSI}^{\mathcal{FC}}$
	$\mathtt{ANE}$	Protocol that realises $\mathcal{PSI}^{\mathcal{FCR}}$
	$\mathtt{PRF}$	Pseudorandom function
	$\mathtt{PRP}$	Pseudorandom permutation
	gcd	Greatest common divisor
	ϵ	Negligible function

Setting	Symbol	Description
Counter Collusion Contracts	$\mathcal{SC}_{\mathrm{PC}}$	Prisoner's Contract
	$\mathcal{SC}_{\mathrm{CC}}$	Colluder's Contract
	$\mathcal{SC}_{\mathrm{TC}}$	Traitor's Contract
	$\ddot{c}$	Server's cost for computing a task
	$\ddot{ch}$	Auditor's cost for resolving disputes
	$\ddot{d}$	Deposit a server pays to get the job
	$\ddot{w}$	Amount a server receives for completing the task
	(pk, sk)	$\mathcal{SC}_{\mathtt{JUS}}$'s auditor's public-private key pair
Justitia (JUS)	$\mathcal{SC}_{\mathtt{JUS}}$	$\mathtt{JUS}$'s smart contract
	$\boldsymbol{\omega}, \boldsymbol{\omega}', \boldsymbol{\rho}$	Random poly. of degree d
	$\boldsymbol{\gamma}, \boldsymbol{\delta}$	Random poly. of degree $d+1$
	$\boldsymbol{\nu}^{(C)}$	Blinded poly. sent by each C to $\mathcal{SC}_{\mathtt{JUS}}$
	$\boldsymbol{\phi}$	Blinded poly. encoding the intersection
	$\boldsymbol{\chi}$	Poly. sent to $\mathcal{SC}_{\mathtt{JUS}}$ to identify misbehaving parties
	L	List of identified misbehaving parties
	$\ddot{y}$	A portion of a party's deposit into $\mathcal{SC}_{\mathtt{JUS}}$ transferred to honest clients if it misbehaves
	Q^{Init}	Initiation predicate
	Q^{Del}	Delivery predicate
	$Q^{\mathrm{UF\text{-}A}}$	UnFair-Abort predicate
	$Q^{\mathrm{F\text{-}A}}$	Fair-Abort predicate
Anesidora (ANE)	$\mathcal{SC}_{\mathtt{ANE}}$	$\mathtt{ANE}$'s smart contract
	$\ddot{d}'$	Extractor's deposit
	$\ddot{y}'$	Each client's deposit into $\mathcal{SC}_{\mathtt{JUS}}$
	$\ddot{l}$	Reward a client earns for an intersection element
	$\ddot{r}$	Extractor's cost for extracting an intersection element
	$\ddot{f}$	Shorthand for $\ddot{l}(m-1)$
	$\ddot{v}$	Price a buyer pays for an intersection element $\ddot{v} = m \cdot \ddot{l} + 2\ddot{r}$
	ct_{mk}	Encryption of mk under pk
	$Q_{\mathrm{R}}^{\mathrm{Del}}$	Delivery-with-Reward predicate
	$Q_{\mathrm{R}}^{\mathrm{UF\text{-}A}}$	UnFair-Abort-with-Reward predicate

3.2 Security Model

In this paper, we use the simulation-based paradigm of secure computation [22] to define and prove our protocols. Since both types of active and passive adversaries are involved in our protocols, we will outline definitions for both types (and refer readers to Appendix A for more details). We consider a static adversary, we assume there is an authenticated private (off-chain) channel between the clients and we consider a standard public blockchain, e.g., Ethereum.

Two-party Computation. A two-party protocol Γ problem is captured by specifying a random process that maps pairs of inputs to pairs of outputs, one for each party. Such process is referred to as a functionality denoted by $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$, where $f := (f_1, f_2)$. For every input pair (x, y), the output pair is a random variable $(f_1(x, y), f_2(x, y))$, such that the party with input x obtains $f_1(x, y)$ while the party with input y receives $f_2(x, y)$. When f is deterministic, then $f_1 = f_2$. In the setting where f is asymmetric and only one party (say the first one) receives the result, f is defined as $f := (f_1(x, y), \perp)$.

Security in the Presence of Passive Adversaries. In this setting, a protocol is secure if whatever can be computed by a party in the protocol can be computed using its input and output only.

Definition 1. *Let f be the deterministic functionality defined above. Protocol Γ security computes f in the presence of a static passive adversary if there exist polynomial-time algorithms $(\mathsf{Sim}_1, \mathsf{Sim}_2)$ such that:*

$$\{\mathsf{Sim}_1(x, f_1(x,y))\}_{x,y} \stackrel{c}{\equiv} \{\mathsf{View}_1^{\Gamma}(x,y)\}_{x,y}, \{\mathsf{Sim}_2(x, f_2(x,y))\}_{x,y} \stackrel{c}{\equiv} \{\mathsf{View}_2^{\Gamma}(x,y)\}_{x,y}$$

where party i's view (during the execution of Γ) on input pair (x, y) is denoted by $\mathsf{View}_i^{\Gamma}(x, y)$ and equals $(w, r^i, m_1^i, ..., m_t^i)$, where $w \in \{x, y\}$ is the input of i^{th} party, r_i is the outcome of this party's internal random coin tosses, and m_j^i represents the j^{th} message this party receives.

Security in the Presence of Active Adversaries. In this adversarial model, correctness is required beyond the possibility that a corrupted party may learn more than it should. To capture the threats, a protocol's security is analyzed by comparing what an adversary can do in the real protocol to what it can do in an ideal scenario. This is formalized by considering an ideal computation involving an incorruptible Trusted Third Party (TTP) to whom the parties send their inputs and receive the output of the ideal functionality.

Definition 2. *Let f be the two-party functionality defined above and Γ be a two-party protocol that computes f. Protocol Γ securely computes f with abort in the presence of static active adversaries if for every non-uniform probabilistic polynomial time adversary $\mathcal{A}$ for the real model, there exists a non-uniform probabilistic polynomial-time adversary (or simulator)* Sim *for the ideal model, such that for every $i \in \{0, 1\}$, it holds that:* $\{\mathsf{Ideal}^{f}_{\mathsf{Sim}(z),i}(x,y)\}_{x,y,z} \stackrel{c}{\equiv} \{\mathsf{Real}^{\Gamma}_{\mathcal{A}(z),i}(x,y)\}_{x,y,z}$

where the ideal execution of f on inputs (x, y) and z is denoted by $\mathsf{Ideal}^{f}_{\mathcal{A}(z),i}(x, y)$ and is defined as the output pair of the honest party and $\mathcal{A}$ from the ideal execution. The real execution of Γ is denoted by $\mathsf{Real}^{\Gamma}_{\mathcal{A}(z),i}(x, y)$, it is defined as the joint output of the parties engaging in the real execution of Γ (on the inputs), in the presence of $\mathcal{A}$.

3.3 Smart Contracts

Cryptocurrencies, such as Bitcoin [34] and Ethereum [40], beyond offering a decentralised currency, support computations on transactions. In this setting, a certain computation logic is encoded in a computer program, called a *"smart contract"*. To date, Ethereum is the most predominant cryptocurrency framework that enables users to define arbitrary smart contracts. In this framework, contract code is stored on the blockchain and executed by all parties (i.e., miners) maintaining the cryptocurrency. The program execution's correctness is guaranteed by the security of the underlying blockchain components.

3.4 Counter Collusion Smart Contracts

To let a client efficiently delegate a computation to a couple of potentially colluding servers, Dong *et al.* [16] proposed two main smart contracts; namely, "Prisoner's Contract" ($\mathcal{SC}_{\text{PC}}$) and "Traitor's Contract" ($\mathcal{SC}_{\text{TC}}$). $\mathcal{SC}_{\text{PC}}$ is signed by the client and the servers. It tries to incentivize correct computation by using the following idea. It requires each server to pay a deposit before the computation is delegated and is equipped with an external auditor that is invoked to detect a misbehaving server only when the servers provide non-equal results.

If a server behaves honestly, it can withdraw its deposit. But, if a cheating server is detected by the auditor, then (a portion) of its deposit is transferred to

the client. If one of the servers is honest and the other one cheats, then the honest server receives a reward taken from the cheating server's deposit. However, the dilemma, created by $\mathcal{SC}_{\text{PC}}$ between the two servers, can be addressed if they can make an enforceable promise, say via a "Colluder's Contract" ($\mathcal{SC}_{\text{CC}}$), in which one party, called "ringleader", would pay its counterparty a bribe if both provide an incorrect computation. To counter $\mathcal{SC}_{\text{CC}}$, Dong *et al.* proposed $\mathcal{SC}_{\text{TC}}$, which incentivises a colluding server to betray the other one and report the collusion without being penalised by $\mathcal{SC}_{\text{PC}}$. In this work, we slightly adjust and use these contracts. We state these tree contracts' main parameters in Table 1. We refer readers to Appendix M for the full description of the parameters and contracts.

3.5 Pseudorandom Function and Permutation

Informally, a pseudorandom function is a deterministic function that takes a key of length λ and an input; and outputs a value indistinguishable from that of a truly random function. In this paper, we use pseudorandom functions: $\texttt{PRF} : \{0,1\}^{\lambda} \times \{0,1\}^{*} \rightarrow \mathbb{F}_p$, where $|p| = \lambda$ is the security parameter [26].

The definition of a pseudorandom permutation, $\texttt{PRP} : \{0,1\}^{\lambda} \times \{0,1\}^{*} \rightarrow \mathbb{F}_p$, is very similar to that of a pseudorandom function, with a difference; namely, it is required the keyed function $\texttt{PRP}(k,.)$ to be indistinguishable from a uniform permutation, instead of a uniform function. In cryptographic schemes that involve PRP, sometimes honest parties may require to compute the inverse of pseudorandom permutation, i.e., $\texttt{PRP}^{-1}(k,.)$, as well. In this case, it would require that $\texttt{PRP}(k,.)$ be indistinguishable from a uniform permutation even if the distinguisher is additionally given oracle access to the inverse of the permutation.

3.6 Commitment Scheme

A commitment scheme involves a *sender* and a *receiver*. It also involves two phases; namely, *commit* and *open* [22]. In the commit phase, the sender commits to a message: x as $\texttt{Com}(x,r) = com$, that involves a secret value: $r \xleftarrow{\$} \{0,1\}^{\lambda}$. At the end of the commit phase, the commitment com is sent to the receiver. In the open phase, the sender sends the opening $\hat{x} := (x,r)$ to the receiver who verifies its correctness: $\texttt{Ver}(com,\hat{x}) \stackrel{?}{=} 1$ and accepts if the output is 1. A commitment scheme must satisfy two properties: (a) *hiding*: it is infeasible for an adversary (i.e., the receiver) to learn any information about the committed message x, until the commitment com is opened, and (b) *binding*: it is infeasible for an adversary (i.e., the sender) to open a commitment com to different values $\hat{x}' := (x',r')$ than that was used in the commit phase, i.e., infeasible to find $\hat{x}'$, *s.t.* $\texttt{Ver}(com,\hat{x}) = \texttt{Ver}(com,\hat{x}') = 1$, where $\hat{x} \neq \hat{x}'$.

3.7 Hash Tables

A hash table is an array of bins each of which can hold a set of elements. It comes with a hash function. To insert an element, we first compute the element's hash, and then store the element in the bin whose index is the element's hash. Given the maximum number of elements c and the bin's maximum size d, we can determine the number of bins, h, by analysing hash tables under the balls into the bins model [6]. Appendix B explains how the hash table parameters are set.

3.8 Merkle Tree

A Merkle tree is a data structure that supports a compact commitment of a set of values/blocks. It involves two parties, a prover and a verifier. The Merkle tree scheme includes three algorithms; namely, `MT.genTree`, `MT.prove`, and `MT.verify`. Briefly, the first algorithm constructs a Merkle tree on file blocks, the second generates a proof of a block's (or set of blocks') membership, and the third verifies the proof. Appendix C provides more details.

3.9 Polynomial Representation of Sets

Using a polynomial to represent a set's elements was proposed by Freedman *et al.* in [19]. In this representation, set elements $S = \{s_1, ..., s_d\}$ are defined over $\mathbb{F}_p$ and set S is represented as a polynomial of form: $\mathbf{p}(x) = \prod_{i=1}^{d}(x - s_i)$, where $\mathbf{p}(x) \in \mathbb{F}_p[X]$ and $\mathbb{F}_p[X]$ is a polynomial ring. Often a polynomial, $\mathbf{p}(x)$, of degree d is represented in the "coefficient form" as follows: $\mathbf{p}(x) = a_0 + a_1 \cdot x + ... + a_d \cdot x^d$. As shown in [8,30], for two sets $S^{(A)}$ and $S^{(B)}$ represented by polynomials $\mathbf{p}_A$ and $\mathbf{p}_B$ respectively, their product: $\mathbf{p}_A \cdot \mathbf{p}_B$ represents the set union, while their greatest common divisor: $gcd(\mathbf{p}_A, \mathbf{p}_B)$ represents the set intersection. For two polynomials $\mathbf{p}_A$ and $\mathbf{p}_B$ of degree d, and two random polynomials $\boldsymbol{\gamma}_A$ and $\boldsymbol{\gamma}_B$ of degree d, it is proven in [8,30] that: $\boldsymbol{\theta} = \boldsymbol{\gamma}_A \cdot \mathbf{p}_A + \boldsymbol{\gamma}_B \cdot \mathbf{p}_B = \boldsymbol{\mu} \cdot gcd(\mathbf{p}_A, \mathbf{p}_B)$, where $\boldsymbol{\mu}$ is a uniformly random polynomial, and polynomial $\boldsymbol{\theta}$ contains only information about the elements in $S^{(A)} \cap S^{(B)}$, and contains no information about other elements in $S^{(A)}$ or $S^{(B)}$.

Given a polynomial $\boldsymbol{\theta}$ that encodes sets intersection, one can find the set elements in the intersection via one of the following approaches. First, via polynomial evaluation: the party who already has one of the original input sets, say $\mathbf{p}_A$, evaluates $\boldsymbol{\theta}$ at every element s_i of $\mathbf{p}_A$ and considers s_i in the intersection if $\mathbf{p}_A(s_i) = 0$. Second, via polynomial root extraction: the party who does not have one of the original input sets, extracts the roots of $\boldsymbol{\theta}$, which contain the roots of (i) random polynomial $\boldsymbol{\mu}$ and (ii) the polynomial that represents the intersection, i.e., $gcd(\mathbf{p}_A, \mathbf{p}_B)$. In this approach, to distinguish errors (i.e., roots of $\boldsymbol{\mu}$) from the intersection, PSIs in [1,30] use the *"hash-based padding technique"*. In this technique, every element u_i in the set universe $\mathcal{U}$, becomes $s_i = u_i || \mathtt{H}(u_i)$, where $\mathtt{H}$ is a cryptographic hash function with a sufficiently large output size. Given a field's arbitrary element, $s \in \mathbb{F}_p$ and $\mathtt{H}$'s output size $|\mathtt{H}(.)|$, we can parse s into x_1 and x_2, such that $s = x_1 || x_2$ and $|x_2| = |\mathtt{H}(.)|$. In a PSI that uses polynomial representation and this padding technique, after we extract each root of $\boldsymbol{\theta}$, say s, we parse it into (x_1, x_2) and check $x_2 \stackrel{?}{=} \mathtt{H}(x_1)$. If the equation holds, then we consider s as an element of the intersection.

3.10 Horner's Method

Horner's method [17] allows for efficiently evaluating polynomials at a given point. Specifically, given a polynomial of the form: $\boldsymbol{\tau}(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + ... + a_n \cdot x^n$ and a point: x_0, one can efficiently evaluate $\boldsymbol{\tau}(x)$ at x_0 iteratively, in the following fashion: $\boldsymbol{\tau}(x_0) = a_0 + x_0(a_1 + x_0(a_2 + ... + x_0(a_{n-1} + x_0 \cdot a_n)...)))$.

Evaluating a polynomial of degree n naively requires n additions and $\frac{(n^2+n)}{2}$ multiplications. However, using Horner's method the evaluation requires only n additions and n multiplications. We use this method in this paper.

3.11 Oblivious Linear Function Evaluation

Oblivious Linear function Evaluation (OLE) is a two-party protocol that involves a sender and receiver. In OLE, the sender has two inputs $a, b \in \mathbb{F}_p$ and the receiver has a single input, $c \in \mathbb{F}_p$. The protocol allows the receiver to learn only $s = a \cdot c + b \in \mathbb{F}_p$, while the sender learns nothing. Ghosh *et al.* [20] proposed an efficient OLE that has $O(1)$ overhead and involves mainly symmetric key operations. Later, in [21] an enhanced OLE, called OLE^+, was proposed. The latter ensures that the receiver cannot learn anything about the sender's inputs, even if it sets its input to 0. In this paper, we use OLE^+. We refer readers to Appendix D, for its construction.

3.12 Coin-Tossing Protocol

A Coin-Tossing protocol, CT, allows two mutually distrustful parties, say A and B, to jointly generate a single random bit. Formally, CT computes the functionality $f_{\text{CT}}(in_A, in_B) \rightarrow (out_A, out_B)$, which takes in_A and in_B as inputs of A and B respectively and outputs out_A to A and out_B to B, where $out_A = out_B$. A basic security requirement of a CT is that the resulting bit is indistinguishable from a truly random bit. Blum proposed a simple CT in [7] that works as follows. Party A picks a random bit $in_A \xleftarrow{\$} \{0,1\}$, commits to it and sends the commitment to B which sends its choice of random input, $in_B \xleftarrow{\$} \{0,1\}$, to A which sends the commitment opening (including in_A) to B, which checks if the commitment matches its opening. If so, each party computes the final random bit as $in_A \oplus in_B$.

There also exist *fair* coin-tossing protocols, e.g., in [33], that ensure either both parties learn the result or nobody does. They can be generalised to *multi-party* coin-tossing protocols to generate a *random string*, e.g., see [4,28]. The complexities of (fair) multi-party coin-tossing protocols are often linear with the number of participants. In this paper, any multi-party CT that generates a random string can be used. For simplicity, we let a multi-party f_{CT} take m inputs and output a single value, i.e., $f_{\text{CT}}(in_1, ..., in_m) \rightarrow out$.

4 Definition of Multi-party PSI with Fair Compensation

In this section, we present the notion of multi-party PSI with Fair Compensation ($\mathcal{PSI}^{\mathcal{FC}}$) which allows either all clients to get the result or the honest parties to be financially compensated if the protocol aborts in an unfair manner, where only dishonest parties learn the result.

In a $\mathcal{PSI}^{\mathcal{MFC}}$, three types of parties are involved; namely, (1) a set of clients $\{A_1, ..., A_m\}$ potentially active adversaries and all but one may collude with each other, (2) a non-colluding dealer, D, potentially passive adversary and has an input set, and (3) an auditor Aud potentially active adversary, where all parties except Aud have input set. For simplicity, we assume that given an address, one can determine whether it belongs to Aud. The basic functionality that a

multi-party PSI computes is defined as $f^{\text{PSI}}(S_1, ..., S_{m+1}) \rightarrow \underbrace{(S_\cap, ..., S_\cap)}_{m+1}$, where $S_\cap = S_1 \cap S_2, ..., \cap S_{m+1}$. To formally define a $\mathcal{PSI}^{\mathcal{FC}}$, we equip f^{PSI} with four predicates, $Q := (Q^{\text{Init}}, Q^{\text{Del}}, Q^{\text{UF-A}}, Q^{\text{F-A}})$, which ensure that certain financial conditions are met. We borrow three of these predicates (i.e., $Q^{\text{Init}}, Q^{\text{Del}}, Q^{\text{UF-A}}$) from [29]; nevertheless, we will (i) introduce an additional predicate $Q^{\text{F-A}}$ and (ii) provide more formal accurate definitions of these predicates.

Predicate Q^{Init} specifies under which condition a protocol that realises $\mathcal{PSI}^{\mathcal{FC}}$ should start executing, i.e., when all set owners have enough deposit. Predicate Q^{Del} determines in which situation parties receive their output, i.e., when honest parties receive their deposit back. Predicate $Q^{\text{UF-A}}$ specifies under which condition the simulator can force parties to abort if the adversary learns the output, i.e., when an honest party receives its deposit back plus a predefined amount of compensation. Predicate $Q^{\text{F-A}}$ specifies under which condition the simulator can force parties to abort if the adversary receives no output, i.e., when honest parties receive their deposits back. Intuitively, by requiring any protocol that realises $\mathcal{PSI}^{\mathcal{FC}}$ to implement a wrapped version of f^{PSI} that includes Q, we will ensure that an honest set owner only aborts in an unfair manner if $Q^{\text{UF-A}}$ returns 1, it only aborts in a fair manner if $Q^{\text{F-A}}$ returns 1, and outputs a valid value if Q^{Del} returns 1. Now, we formally define each of these predicates.

Definition 3 (Q^{Init}: Initiation predicate). *Let $\mathcal{G}$ be a stable ledger, adr_{sc} be smart contract sc's address, Adr be a set of $m+1$ distinct addresses, and $\ddot{x}$ be a fixed amount of coins. Then, predicate $Q^{Init}(\mathcal{G}, adr_{sc}, m+1, Adr, \ddot{x})$ returns 1 if every address in Adr has at least $\ddot{x}$ coins in sc; otherwise, it returns 0.*

Definition 4 (Q^{Del}: Delivery predicate). *Let $pram := (\mathcal{G}, adr_{sc}, \ddot{x})$ be the parameters defined above, and $adr_i \in Adr$ be the address of an honest party. Then, predicate $Q^{Del}(pram, adr_i)$ returns 1 if adr_i has sent $\ddot{x}$ amount to sc and received $\ddot{x}$ amount from it; thus, its balance in sc is 0. Otherwise, it returns 0.*

Definition 5 ($Q^{\text{UF-A}}$: UnFair-Abort predicate). *Let $pram := (\mathcal{G}, adr_{sc}, \ddot{x})$ be the parameters defined above, and $Adr' \subset Adr$ be a set containing honest parties' addresses, $m' = |Adr'|$, and $adr_i \in Adr'$. Let also G be a compensation function that takes as input three parameters $(\ddot{deps}, adr_i, m')$, where $\ddot{deps}$ is the amount of coins that all $m+1$ parties deposit. It returns the amount of compensation each honest party must receive, i.e., $G(\ddot{deps}, ard_i, m') \rightarrow \ddot{x}_i$. Then, predicate $Q^{UnF\text{-}Abt}$ is defined as $Q^{UF\text{-}A}(pram, G, \ddot{deps}, m', adr_i) \rightarrow (a, b)$, where $a = 1$ if adr_i is an honest party's address and adr_i has sent $\ddot{x}$ amount to sc and received $\ddot{x} + \ddot{x}_i$ from it, and $b = 1$ if adr_i is Aud's address and adr_i received $\ddot{x}_i$ from sc. Otherwise, $a = b = 0$.*

Definition 6 ($Q^{\text{F-A}}$: Fair-Abort predicate). *Let $pram := (\mathcal{G}, adr_{sc}, \ddot{x})$ be the parameters defined above, and $Adr' \subset Adr$ be a set containing honest parties' addresses, $m' = |Adr'|$, $adr_i \in Adr'$, and adr_j be Aud's address. Let G be the compensation function, defined above and let $G(deps, ard_j, m') \rightarrow \ddot{x}_j$ be the compensation that the auditor must receive. Then, predicate $Q^{F\text{-}A}(pram, G, \ddot{deps}, m', adr_i, adr_j)$ returns 1, if adr_i (s.t. $adr_i \neq adr_j$) has sent $\ddot{x}$ amount to sc and received $\ddot{x}$ from it, and adr_j received $\ddot{x}_j$ from sc. Otherwise, it returns 0.*

Definition 7 ($\mathcal{PSI}^{\mathcal{FC}}$). *Let f^{PSI} be the multi-party PSI functionality defined above. We say protocol Γ realises f^{PSI} with Q-fairness in the presence of $m-1$ static active-adversary clients (i.e., A_js) or a static passive dealer D or passive auditor Aud, if for every non-uniform probabilistic polynomial time adversary $\mathcal{A}$ for the real model, there exists a non-uniform probabilistic polynomial-time adversary (or simulator)* Sim *for the ideal model, such that for every $I \in \{A_1, ..., A_m, D, Aud\}$, it holds that:*

$\{\mathsf{Ideal}^{\mathcal{W}(f^{PSI},Q)}_{\mathsf{Sim}(z),I}(S_1, ..., S_{m+1})\}_{S_1,...,S_{m+1},z} \stackrel{c}{\equiv} \{\mathsf{Real}^{\Gamma}_{\mathcal{A}(z),I}(S_1, ..., S_{m+1})\}_{S_1,...,S_{m+1},z}$

where z is an auxiliary input given to $\mathcal{A}$ and $\mathcal{W}(f^{PSI}, Q)$ is a functionality that wraps f^{PSI} with predicates $Q := (Q^{Init}, Q^{Del}, Q^{UF\text{-}A}, Q^{F\text{-}A})$.

5 Subroutines Used in Justitia

In this section, we present three subroutines and a primitive that will be used in the instantiation of $\mathcal{PSI}^{\mathcal{FC}}$, i.e., Justitia.

5.1 Verifiable Oblivious Polynomial Randomisation (VOPR)

In the VOPR, two parties are involved, (i) a sender which is potentially a passive adversary and (ii) a receiver that is potentially an active adversary. The protocol allows the receiver with input polynomial $\boldsymbol{\beta}$ (of degree e') and the sender with input random polynomials $\boldsymbol{\psi}$ (of degree e) and $\boldsymbol{\alpha}$ (of degree $e + e'$) to compute: $\boldsymbol{\theta} = \boldsymbol{\psi} \cdot \boldsymbol{\beta} + \boldsymbol{\alpha}$, such that (a) the receiver learns only $\boldsymbol{\theta}$ and nothing about the sender's input even if it sets $\boldsymbol{\beta} = 0$, (b) the sender learns nothing, and (c) the receiver's misbehaviour is detected in the protocol. Thus, the functionality that VOPR computes is defined as $f^{\text{VOPR}}((\boldsymbol{\psi}, \boldsymbol{\alpha}), \boldsymbol{\beta}) \rightarrow (\perp, \boldsymbol{\psi} \cdot \boldsymbol{\beta} + \boldsymbol{\alpha})$. We will use VOPR in Justitia for two main reasons: (a) to let a party re-randomise its counterparty's polynomial (representing its set) and (b) to impose a MAC-like structure to the randomised polynomial; such a structure will allow a verifier to detect if VOPR's output has been modified.

Now, we outline how we design VOPR without using any (expensive) zero-knowledge proofs.[1] In the setup phase, both parties represent their input polynomials in the regular coefficient form; therefore, the sender's polynomials are defined as $\boldsymbol{\psi} = \sum_{i=0}^{e} g_i \cdot x^i$ and $\boldsymbol{\alpha} = \sum_{j=0}^{e+e'} a_j \cdot x^j$ and the receiver's polynomial is defined as $\boldsymbol{\beta} = \sum_{i=0}^{e'} b_i \cdot x^i$. However, the sender computes each coefficient a_j (of polynomial $\boldsymbol{\alpha}$) as follows, $a_j = \sum_{t,k=0}^{\substack{k=e' \\ t=e}} a_{t,k}$, where $t + k = j$ and each $a_{t,k}$ is a random value. For instance, if $e = 4$ and $e' = 3$, then $a_3 = a_{0,3} + a_{3,0} + a_{1,2} + a_{2,1}$. Shortly, we explain why polynomial $\boldsymbol{\alpha}$ is constructed this way.

In the computation phase, to compute polynomial $\boldsymbol{\theta}$, the two parties interactively multiply and add the related coefficients in a secure way using OLE^+

[1] Previously, Ghosh *et al.* [21] designed a protocol called Oblivious Polynomial Addition (OPA) to meet similar security requirements that we laid out above. But, as shown in [3], OPA is susceptible to several serious attacks.

(presented in Section 3.11). Specifically, for every j (where $0 \leq j \leq e'$) the sender sends g_i and $a_{i,j}$ to an instance of $\texttt{OLE}^+$, while the receiver sends b_j to the same instance, which returns $c_{i,j} = g_i \cdot b_j + a_{i,j}$ to the receiver. This process is repeated for every i, where $0 \leq i \leq e$. Then, the receiver uses $c_{i,j}$ values to construct the resulting polynomial, $\boldsymbol{\theta}$.

The reason that the sender imposes the above structure to (the coefficients of) $\boldsymbol{\alpha}$ in the setup, is to let the parties securely compute $\boldsymbol{\theta}$ via $\texttt{OLE}^+$. Specifically, by imposing this structure (1) the sender can blind each product $g_i \cdot b_j$ with random value $a_{i,j}$ which is a component of $\boldsymbol{\alpha}$'s coefficient and (2) the receiver can construct a result polynomial of the form $\boldsymbol{\theta} = \boldsymbol{\psi} \cdot \boldsymbol{\beta} + \boldsymbol{\alpha}$.

Now, we outline how the verification works. To check the result's correctness, the sender picks and sends a random value z to the receiver which computes $\boldsymbol{\theta}(z)$ and $\boldsymbol{\beta}(z)$ and sends these two values to the sender. The sender computes $\boldsymbol{\psi}(z)$ and $\boldsymbol{\alpha}(z)$ and then checks if equation $\boldsymbol{\theta}(z) = \boldsymbol{\psi}(z) \cdot \boldsymbol{\beta}(z) + \boldsymbol{\alpha}(z)$ holds. It accepts the result if the check passes. Figure 1 describes $\texttt{VOPR}$ in detail.

- *Input.*
  - *Public Parameters*: upper bound on input polynomials' degree: e and e'.
  - *Sender Input*: random polynomials: $\boldsymbol{\psi} = \sum_{i=0}^{e} g_i \cdot x^i$ and $\boldsymbol{\alpha} = \sum_{j=0}^{e+e'} a_j \cdot x^j$, where $g_i \xleftarrow{\$} \mathbb{F}_p$.

    Each a_j has the form: $a_j = \sum_{\substack{t,k=0 \\ t=e}}^{k=e'} a_{t,k}$, such that $t + k = j$ and $a_{t,k} \xleftarrow{\$} \mathbb{F}_p$.
  - *Receiver Input*: polynomial $\boldsymbol{\beta} = \boldsymbol{\beta}_1 \cdot \boldsymbol{\beta}_2 = \sum_{i=0}^{e'} b_i \cdot x^i$, where $\boldsymbol{\beta}_1$ is a random polynomial of degree 1 and $\boldsymbol{\beta}_2$ is an arbitrary polynomial of degree $e' - 1$.
- *Output.* The receiver gets $\boldsymbol{\theta} = \boldsymbol{\psi} \cdot \boldsymbol{\beta} + \boldsymbol{\alpha}$.

1. **Computation:**
   (a) Sender and receiver together for every j, $0 \leq j \leq e'$, invoke $e + 1$ instances of $\texttt{OLE}^+$. In particular, $\forall j, 0 \leq j \leq e'$: sender sends g_i and $a_{i,j}$ while the receiver sends b_j to $\texttt{OLE}^+$ that returns: $c_{i,j} = g_i \cdot b_j + a_{i,j}$ to the receiver ($\forall i, 0 \leq i \leq e$).
   (b) The receiver sums component-wise values $c_{i,j}$ that results in polynomial:

$$\boldsymbol{\theta} = \boldsymbol{\psi} \cdot \boldsymbol{\beta} + \boldsymbol{\alpha} = \sum_{\substack{i,j=0 \\ j=e'}}^{i=e} c_{i,j} \cdot x^{i+j}$$

2. **Verification:**
   (a) Sender: picks a random value z and sends it to the receiver.
   (b) Receiver: sends $\theta_z = \boldsymbol{\theta}(z)$ and $\beta_z = \boldsymbol{\beta}(z)$ to the sender.
   (c) Sender: computes $\psi_z = \boldsymbol{\psi}(z)$ and $\alpha_z = \boldsymbol{\alpha}(z)$ and checks if equation $\theta_z = \psi_z \cdot \beta_z + \alpha_z$ holds. If the equation holds, it concludes that the computation was performed correctly. Otherwise, it aborts.

Fig. 1: Verifiable Oblivious Polynomial Randomization ($\texttt{VOPR}$)

Theorem 1. *Let $f^{\texttt{VOPR}}$ be the functionality defined above. If the enhanced* $\texttt{OLE}$ *(i.e.,* $\texttt{OLE}^+$*) is secure against malicious (or active) adversaries, then the Verifiable Oblivious Polynomial Randomisation (*$\texttt{VOPR}$*), presented in Figure 1, securely*

computes f^{VOPR} in the presence of (i) a semi-honest sender and honest receiver or (ii) a malicious receiver and honest sender.

We refer readers to Appendix E for the proof of Theorem 1.

5.2 Zero-sum Pseudorandom Values Agreement Protocol (ZSPA)

The ZSPA allows m parties (the majority of which is potentially malicious) to efficiently agree on (a set of vectors, where each vector has) m pseudorandom values such that their sum equals zero. At a high level, the parties first sign a smart contract and then run a coin-tossing protocol CT to agree on a key: k. Next, one of the parties generates $m-1$ pseudorandom values z_j (where $1 \leq j \leq m-1$) using key k and PRF. It sets the last value as the additive inverse of the sum of the values generated, i.e. $z_m = -\sum_{j=1}^{m-1} z_j$. Then, it constructs a Merkel tree on top of the pseudorandom values and stores only the tree's root g and the key's hash value q in the smart contract. Then, each party (using the key) locally checks if the values (on the contract) have been constructed correctly; if so, then it sends a signed "approved" message to the contract. Hence, the functionality that ZSPA computes is defined as $f^{\text{ZSPA}}(\underbrace{\perp, ..., \perp}_{m}) \rightarrow \underbrace{((k, g, q), ..., (k, g, q))}_{m}$, where g is the Markle tree's root built on the pseudorandom values $z_{i,j}$, q is the hash value of the key used to generate the pseudorandom values, and $m \geq 2$. Figure 2 presents ZSPA in detail.

Briefly, ZSPA will be used in Justitia to allow clients $\{A_1, ..., A_m\}$ to provably agree on a set of pseudorandom polynomials whose sum is zero. Each of these polynomials will be used by a client to blind/encrypt the messages it sends to the smart contract, to protect the privacy of the plaintext message (from *Aud*, D, and the public). To compute the sum of the plaintext messages, one can easily sum the blinded messages, which removes the blinding polynomials.

Theorem 2. *Let f^{ZSPA} be the functionality defined above. If* CT *is secure against a malicious adversary and the correctness of* PRF, H, *and Merkle tree holds, then* ZSPA, *in Figure 2, securely computes f^{ZSPA} in the presence of $m-1$ malicious adversaries.*

We refer readers to Appendix F, for the proof of Theorem 2.

5.3 ZSPA's Extension: ZSPA with an External Auditor (ZSPA-A)

In this section, we present an extension of ZSPA, called ZSPA-A which lets a (trusted) third-party auditor, *Aud*, help identify misbehaving clients in the ZSPA and generate a vector of random polynomials. Informally, ZSPA-A requires that misbehaving parties are always detected, except with a negligible probability. *Aud* of this protocol will be invoked by Justitia when Justitia's smart contract detects that a combination of the messages sent by the clients is not well-formed. Later, in Justitia's proof, we will show that even a *semi-honest Aud* who observes all messages that clients send to Justitia's smart contracts, cannot learn anything about their set elements. We present ZSPA-A in Figure 3.

- Parties. A set of clients $\{A_1, ..., A_m\}$.
- Input. m: the total number of participants, adr: a deployed smart contract's address, and b: the total number of vectors. Let $b' = b - 1$.
- Output. k: a secret key that generates b vectors $[z_{0,1}, ..., z_{0,m}], ..., [z_{b',1}, ..., z_{b',m}]$ of pseudo-random values, h: hash of the key, g: a Merkle tree's root, and a vector of signed messages.

1. **Coin-tossing.** $\texttt{CT}(in_1, ..., in_m) \rightarrow k$.
   All participants run a coin-tossing protocol to agree on PRF's key, k.
2. **Encoding.** $\texttt{Encode}(k, m) \rightarrow (g, q)$.
   One of the parties takes the following steps:
   (a) for every i (where $0 \leq i \leq b'$), generates m pseudorandom values as follows.

$$\forall j, 1 \leq j \leq m-1 : z_{i,j} = \texttt{PRF}(k, i||j), \qquad z_{i,m} = -\sum_{j=1}^{m-1} z_{i,j}$$

   (b) constructs a Merkel tree on top of all pseudorandom values, $\texttt{MT.genTree}(z_{0,1}, ..., z_{b',m}) \rightarrow g$.
   (c) sends the Merkel tree's root: g, and the key's hash: $q = \texttt{H}(k)$ to adr.
3. **Verification.** $\texttt{Verify}(k, g, q, m) \rightarrow (a, s)$.
   Each party checks if, all $z_{i,j}$ values, the root g, and key's hash q have been correctly generated, by retaking step 2. If the checks pass, it sets $a = 1$, sets s to a singed "approved" message, and sends s to adr. Otherwise, it aborts by returning $a = 0$ and $s = \perp$.

Fig. 2: Zero-sum Pseudorandom Values Agreement (ZSPA)

Theorem 3. *If* ZSPA *is secure,* H *is second-preimage resistant, and the correctness of* PRF*,* H*, and Merkle tree holds, then* ZSPA-A *securely computes* $f^{\texttt{ZSPA-A}}$ *in the presence of* $m - 1$ *malicious adversaries.*

We refer readers to Appendix G for the proof of Theorem 3.

5.4 Unforgeable Polynomials

In this section, we introduce the notion of "unforgeable polynomials". Informally, an unforgeable polynomial has a secret factor. To ensure that an unforgeable polynomial has not been tampered with, a verifier can check whether the polynomial is divisible by the secret factor.

To turn an arbitrary polynomial $\boldsymbol{\pi}$ of degree d into an unforgeable polynomial $\boldsymbol{\theta}$, one can (i) pick three secret random polynomials $(\boldsymbol{\zeta}, \boldsymbol{\omega}, \boldsymbol{\gamma})$ and (ii) compute $\boldsymbol{\theta} = \boldsymbol{\zeta} \cdot \boldsymbol{\omega} \cdot \boldsymbol{\pi} + \boldsymbol{\gamma} \bmod p$, where $deg(\boldsymbol{\zeta}) = 1, deg(\boldsymbol{\omega}) = d$, and $deg(\boldsymbol{\gamma}) = 2d + 1$.

To verify whether $\boldsymbol{\theta}$ has been tampered with, a verifier (given $\boldsymbol{\theta}, \boldsymbol{\gamma}$, and $\boldsymbol{\zeta}$) can check if $\boldsymbol{\theta} - \boldsymbol{\gamma}$ is divisible by $\boldsymbol{\zeta}$. Informally, the security of an unforgeable polynomial states that an adversary (who does not know the three secret random polynomials) cannot tamper with an unforgeable polynomial without being detected, except with a negligible probability, in the security parameter. Below, we formally state it.

Theorem 4 (Unforgeable Polynomial). *Let polynomials* $\boldsymbol{\zeta}$*,* $\boldsymbol{\omega}$*, and* $\boldsymbol{\gamma}$ *be three secret uniformly random polynomials (i.e.,* $\boldsymbol{\zeta}, \boldsymbol{\omega}, \boldsymbol{\gamma} \xleftarrow{\$} \mathbb{F}_p[x]$*),* $GCD(\boldsymbol{\zeta}, \boldsymbol{\gamma}) = 1$*, polynomial* $\boldsymbol{\pi}$ *be an arbitrary polynomial,* $deg(\boldsymbol{\zeta}) = 1, deg(\boldsymbol{\omega}) = d, deg(\boldsymbol{\gamma}) = 2d + 1$*,* $deg(\boldsymbol{\pi}) = d$*, and* p *be a* λ*-bit prime number. Also, let polynomial* $\boldsymbol{\theta}$ *be defined as* $\boldsymbol{\theta} = \boldsymbol{\zeta} \cdot \boldsymbol{\omega} \cdot \boldsymbol{\pi} + \boldsymbol{\gamma} \bmod p$*. Given* $(\boldsymbol{\theta}, \boldsymbol{\pi})$*, the probability that an adversary*

- Parties. A set of clients $\{A_1, ..., A_m\}$ and an external auditor, *Aud*.
- Input. m: the total number of participants (excluding the auditor), $\boldsymbol{\zeta}$: a random polynomial of degree 1, b: the total number of vectors, and adr: a deployed smart contract's address. Let $b' = b - 1$.
- Output of each A_j. k: a secret key that generates b vectors $[z_{0,1}, ..., z_{0,m}], ..., [z_{b',1}, ..., z_{b',m}]$ of pseudorandom values, h: hash of the key, g: a Merkle tree's root, and a vector of signed messages.
- Output of *Aud*. L: a list of misbehaving parties' indices, and $\vec{\mu}$: a vector of random polynomials.

1. **ZSPA invocation.** $\texttt{ZSPA}(\perp, ..., \perp) \rightarrow \Big((k, g, q), ..., (k, g, q)\Big)$.
   All parties in $\{A_1, ..., A_m\}$ call the same instance of ZSPA, which results in $(k, g, q), ..., (k, g, q)$.
2. **Auditor computation.** $\texttt{Audit}(\vec{k}, q, \boldsymbol{\zeta}, b, g) \rightarrow (L, \vec{\mu})$.
   Aud takes the below steps. Note, each $k_j \in \vec{k}$ is given by A_j. An honest party's input, k_j, equals k, where $1 \leq j \leq m$.
   (a) runs the checks in the verification phase (i.e., Phase 3) of ZSPA for every j, i.e., $\texttt{Verify}(k_j, g, q, m) \rightarrow (a_j, s)$.
   (b) appends j to L, if any checks fails, i.e., if $a_j = 0$. In this case, it skips the next two steps for the current j.
   (c) For every i (where $0 \leq i \leq b'$), it recomputes m pseudorandom values: $\forall j, 1 \leq j \leq m-1 : z_{i,j} = \texttt{PRF}(k, i||j), \quad z_{i,m} = -\sum_{j=1}^{m-1} z_{i,j}$.
   (d) generates polynomial $\boldsymbol{\mu}^{(j)}$ as follows: $\boldsymbol{\mu}^{(j)} = \boldsymbol{\zeta} \cdot \boldsymbol{\xi}^{(j)} - \boldsymbol{\tau}^{(j)}$, where $\boldsymbol{\xi}^{(j)}$ is a random polynomial of degree $b' - 1$ and $\boldsymbol{\tau}^{(j)} = \sum_{i=0}^{b'} z_{i,j} \cdot x^i$. By the end of this step, a vector $\vec{\mu}$ containing at most m polynomials is generated.
   (e) returns list L and $\vec{\mu}$.

Fig. 3: ZSPA with an external auditor (ZSPA-A)

(which does not know $\boldsymbol{\zeta}, \boldsymbol{\omega}$, and $\boldsymbol{\gamma}$) can forge $\boldsymbol{\theta}$ to an arbitrary polynomial $\boldsymbol{\delta}$ such that $\boldsymbol{\delta} \neq \boldsymbol{\theta}$, $deg(\boldsymbol{\delta}) = const(\lambda)$, and $\boldsymbol{\zeta}$ divides $\boldsymbol{\delta} - \boldsymbol{\gamma}$ is negligible in the security parameter λ, i.e., $Pr[\boldsymbol{\zeta} \mid (\boldsymbol{\delta} - \boldsymbol{\gamma})] \leq \epsilon(\lambda)$.

Proof. Let $\boldsymbol{\tau} = \boldsymbol{\delta} - \boldsymbol{\gamma}$ and $\boldsymbol{\zeta} = a \cdot x + b$. Since $\boldsymbol{\gamma}$ is a random polynomial of degree $2d+1$ and unknown to the adversary, given $(\boldsymbol{\theta}, \boldsymbol{\pi})$, the adversary cannot learn anything about the factor $\boldsymbol{\zeta}$; as from its point of view every polynomial of degree 1 in $\mathbb{F}_p[X]$ is equally likely to be $\boldsymbol{\zeta}$. Moreover, polynomial $\boldsymbol{\tau}$ has at most $Max\big(deg(\boldsymbol{\delta}), 2d+1\big)$ irreducible non-constant factors. For $\boldsymbol{\zeta}$ to divide $\boldsymbol{\tau}$, one of the factors of $\boldsymbol{\tau}$ must be equal to $\boldsymbol{\zeta}$. We also know that $\boldsymbol{\zeta}$ has been picked uniformly at random (i.e., $a, b \xleftarrow{\$} \mathbb{F}_p$) and by definition $GCD(\boldsymbol{\zeta}, \boldsymbol{\gamma}) = 1$. Thus, the probability that $\boldsymbol{\zeta}$ divides $\boldsymbol{\tau}$ is negligible in the security parameter, λ. Specifically, $Pr[\boldsymbol{\zeta} \mid (\boldsymbol{\delta} - \boldsymbol{\gamma})] \leq \frac{Max\big(deg(\boldsymbol{\delta}), 2d+1\big)}{2^{2\lambda}} = \epsilon(\lambda)$. □

An interesting feature of an unforgeable polynomial is that the verifier can perform the check without needing to know the original polynomial $\boldsymbol{\pi}$. Another appealing feature of the unforgeable polynomial is that it supports *linear combination* and accordingly *batch verification*. Specifically, to turn n arbitrary polynomials $[\boldsymbol{\pi}_1, ..., \boldsymbol{\pi}_n]$ into unforgeable polynomials, one can construct $\boldsymbol{\theta}_i = \boldsymbol{\zeta} \cdot \boldsymbol{\omega}_i \cdot \boldsymbol{\pi}_i + \boldsymbol{\gamma}_i \bmod p$, where $\forall i, 1 \leq i \leq n$.

To check whether all polynomials $[\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_n]$ are intact, a verifier can (i) compute their sum $\boldsymbol{\chi} = \sum_{i=1}^{n} \boldsymbol{\theta}_i$ and (ii) check whether $\boldsymbol{\chi} - \sum_{i=1}^{n} \boldsymbol{\gamma}_i$ is divisible by $\boldsymbol{\zeta}$. Informally, the security of an unforgeable polynomial states that an adversary (who does not know the three secret random polynomials for each $\boldsymbol{\theta}_i$) cannot tamper with any subset of the unforgeable polynomials without being detected, except with a negligible probability. We formally state it, below.

Theorem 5 (Unforgeable Polynomials' Linear Combination). *Let polynomial $\boldsymbol{\zeta}$ be a secret polynomial picked uniformly at random; also, let $\vec{\boldsymbol{\omega}} = [\boldsymbol{\omega}_1, ..., \boldsymbol{\omega}_n]$ and $\vec{\boldsymbol{\gamma}} = [\boldsymbol{\gamma}_1, ..., \boldsymbol{\gamma}_n]$ be two vectors of secret uniformly random polynomials (i.e., $\boldsymbol{\zeta}, \boldsymbol{\omega}_i, \boldsymbol{\gamma}_i \xleftarrow{\$} \mathbb{F}_p[x]$), $GCD(\boldsymbol{\zeta}, \boldsymbol{\gamma}_i) = 1$, $\vec{\boldsymbol{\pi}} = [\boldsymbol{\pi}_1, ..., \boldsymbol{\pi}_n]$ be a vector of arbitrary polynomials, $deg(\boldsymbol{\zeta}) = 1, deg(\boldsymbol{\omega}_i) = d, deg(\boldsymbol{\gamma}_i) = 2d+1$, $deg(\boldsymbol{\pi}_i) = d$, p be a λ-bit prime number, and $1 \leq i \leq n$. Moreover, let polynomial $\boldsymbol{\theta}_i$ be defined as $\boldsymbol{\theta}_i = \boldsymbol{\zeta} \cdot \boldsymbol{\omega}_i \cdot \boldsymbol{\pi}_i + \boldsymbol{\gamma}_i \bmod p$, and $\vec{\boldsymbol{\theta}} = [\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_n]$. Given $(\vec{\boldsymbol{\theta}}, \vec{\boldsymbol{\pi}})$, the probability that an adversary (which does not know $\boldsymbol{\zeta}, \vec{\boldsymbol{\omega}}$, and $\vec{\boldsymbol{\gamma}}$) can forge t polynomials, without loss of generality, say $\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_t \in \vec{\boldsymbol{\theta}}$ to arbitrary polynomials $\boldsymbol{\delta}_1, ..., \boldsymbol{\delta}_t$ such that $\sum_{j=1}^{t} \boldsymbol{\delta}_j \neq \sum_{j=1}^{t} \boldsymbol{\theta}_j$, $deg(\boldsymbol{\delta}_j) = const(\lambda)$, and $\boldsymbol{\zeta}$ divides $(\sum_{j=1}^{t} \boldsymbol{\delta}_j + \sum_{j=t+1}^{n} \boldsymbol{\theta}_j - \sum_{j=1}^{n} \boldsymbol{\gamma}_j)$ is negligible in the security parameter λ, i.e., $Pr[\boldsymbol{\zeta} \mid (\sum_{j=1}^{t} \boldsymbol{\delta}_j + \sum_{j=t+1}^{n} \boldsymbol{\theta}_j - \sum_{j=1}^{n} \boldsymbol{\gamma}_j)] \leq \epsilon(\lambda)$.*

Proof. This proof is a generalisation of that of Theorem 4. Let $\boldsymbol{\tau}_j = \boldsymbol{\delta}_j - \boldsymbol{\gamma}_j$ and $\boldsymbol{\zeta} = a \cdot x + b$. Since every $\boldsymbol{\gamma}_j$ is a random polynomial of degree $2d+1$ and unknown to the adversary, given $(\vec{\boldsymbol{\theta}}, \vec{\boldsymbol{\pi}})$, the adversary cannot learn anything about the factor $\boldsymbol{\zeta}$. Each polynomial $\boldsymbol{\tau}_j$ has at most $Max\big(deg(\boldsymbol{\delta}_j), 2d+1\big)$ irreducible non-constant factors. We know that $\boldsymbol{\zeta}$ has been picked uniformly at random (i.e., $a, b \xleftarrow{\$} \mathbb{F}_p$), by definition $GCD(\boldsymbol{\zeta}, \boldsymbol{\gamma}_j) = 1$, and $\boldsymbol{\zeta}$ does divide every $\boldsymbol{\theta}_j$. Therefore, the probability that $\boldsymbol{\zeta}$ divides $\sum_{j=1}^{t} \boldsymbol{\delta}_j + \sum_{j=t+1}^{n} \boldsymbol{\theta}_j - \sum_{j=1}^{n} \boldsymbol{\gamma}_j$ equals the probability that $\boldsymbol{\zeta}$ equals to one of the factors of every $\boldsymbol{\tau}_j$, that is negligible in the security parameter. Concretely,

$$Pr[\boldsymbol{\zeta} \mid (\sum_{j=1}^{t} \boldsymbol{\delta}_j + \sum_{j=t+1}^{n} \boldsymbol{\theta}_j - \sum_{j=1}^{n} \boldsymbol{\gamma}_j)] \leq \frac{\prod_{j=1}^{t} Max\big(deg(\boldsymbol{\delta}_j), 2d+1\big)}{2^{2\lambda t}} = \epsilon(\lambda)$$

□

Briefly, in Justitia, we will use unforgeable polynomials (and their linear combinations) to allow a smart contract to efficiently check whether the polynomials that the clients send to it are intact, i.e., they are `VOPR`'s outputs.

6 Justitia: A Concrete Construction of $\mathcal{PSI}^{\mathcal{FC}}$

6.1 Main Challenges to Overcome

We need to address several key challenges, to design an efficient scheme that realises $\mathcal{PSI}^{\mathcal{FC}}$. Below, we outline these challenges.

Keeping Overall Complexities Low. In general, in multi-party PSIs, each client needs to send messages to the rest of the clients and/or engage in secure computation with them, e.g., in [23,31], which would result in communication and/or computation quadratic with the number of clients. To address this challenge, we (a) allow one of the clients as a dealer to interact with the rest of the clients,[2] and (b) we use a smart contract, which acts as a bulletin board to which most messages are sent and also performs lightweight computation on the clients' messages. The combination of these approaches will keep the communication and computation linear with the number of clients (and sets' cardinality).

Preserving the Privacy of Outgoing Messages. Although the use of regular public smart contracts (e.g., Ethereum) will help keep overall complexity low, it introduces another challenge; namely, if clients do not protect the privacy of the messages they send to the smart contracts, then other clients (e.g., dealer) and non-participants of PSI (i.e., the public) can learn the clients' set elements and/or the intersection. Because standard smart contracts do not automatically preserve messages' privacy. To efficiently protect the privacy of each client's messages (sent to the contracts) from the dealer, we require the clients (except the dealer) to engage in `ZSPA-A` which lets each of them generate a pseudorandom polynomial with which it can blind its message. To protect the privacy of the intersection from the public, we require all clients to run a coin-tossing protocol to agree on a blinding polynomial, with which the final result that encodes the intersection on the smart contract will be blinded.

Ensuring the Correctness of Subroutine Protocols' Outputs. In general, any MPC that must remain secure against an active adversary is equipped with a verification mechanism that ensures an adversary is detected if it deviates from the protocol and affects messages' integrity, during the protocol's execution. This is the case for the subroutine protocols that we use, i.e., `VOPR` and `ZSPA-A`. Nevertheless, this type of verification itself is not always sufficient. Because in certain cases, the output of an MPC protocol may be fed as input to another MPC and we need to ensure that the *actual/intact* output of the first MPC is fed to the second one. This is the case in our PSI's subroutines as well. To address this challenge, we use unforgeable polynomials; specifically, the output of `VOPR` is an unforgeable polynomial (that encodes the actual output); if the adversary tampers with the `VOPR`'s output and uses it later, then a verifier can detect it. We will have the same integrity guarantee for the output of `ZSPA-A` for free. Because (i) `VOPR` is called before `ZSPA-A`, and (ii) if clients use intact outputs of `ZSPA-A`, then the final result (i.e., the sum of all clients' messages) will not contain any output of `ZSPA-A`, as they would cancel out each other. Thus, by checking the correctness of the final result, one can ensure the correctness of the outputs of `VOPR` and `ZSPA-A`, in one go.

6.2 Description of Justitia (`JUS`)

An overview. At a high level, Justitia (`JUS`) works as follows. First, each client encodes its set elements into a polynomial. All clients sign a smart contract

[2] This approach has similarity with the non-secure PSIs in [21].

and deposit a predefined amount of coins into it. Next, one of the clients as a dealer, D, randomises the rest of the clients' polynomials and imposes a certain structure to their polynomials. The clients also randomise D's polynomials. The randomised polynomials reveal nothing about the clients' original polynomials representing their set elements. Then, all clients send their randomised polynomials to the smart contract. The contract combines all polynomials and checks whether the resulting polynomial still has the structure imposed by D. If the contract concludes that the resulting polynomial does not have the structure, then it invokes an auditor, Aud, to identify misbehaving clients and penalise them. Nevertheless, if the resulting polynomial has the structure, then the contract outputs an encoded polynomial and refunds the clients' deposits. In this case, all clients can use the resulting polynomial (output by the contract) to locally find the intersection.

One of the novelties of JUS is a lightweight verification mechanism which allows a smart contract to efficiently verify the correctness of the clients' messages without being able to learn/reveal the clients' set elements. To achieve this, D randomises each client's polynomials and constructs unforgeable polynomials on the randomised polynomials (in one go). If any client modifies an unforgeable polynomial that it receives and sends the modified polynomial to the smart contract, then the smart contract would detect it, by checking whether the sum of all clients' (unforgeable) polynomials is divisible by a certain polynomial of degree 1. The verification is lightweight because: (i) it does not use any public key cryptography (often computationally expensive), (ii) it needs to perform only polynomial division, and (iii) it can perform batch verification, i.e., it sums all clients randomised polynomials and then checks the result's correctness.

Now, we describe JUS in more detail. First, all clients sign and deploy a smart contract, $\mathcal{SC}_{\texttt{JUS}}$. Each of them put a certain amount of deposit into it. Then, they together run CT to agree on a key, mk, that will be used to generate a set of blinding polynomials to hide the final result from the public. Next, each client locally maps its set elements to a hash table and represents the content of each hash table's bin as a polynomial, $\boldsymbol{\pi}$. After that, for each bin, the following steps are taken. All clients, except D, engage in ZSPA-A to agree on a set of pseudorandom blinding factors such that their sum is zero.

Then, D randomises and constructs an unforgeable polynomial on each client's polynomial, $\boldsymbol{\pi}$. To do that, D and every other client engage in VOPR that returns to the client a polynomial. D and every other client invoke VOPR again to randomise D's polynomial. VOPR returns to the client another unforgeable polynomial. Note that the output of VOPR reveals nothing about any client's original polynomial $\boldsymbol{\pi}$, as this polynomial has been blinded with another random polynomial by D, during the execution of VOPR. Each client sums the two polynomials, blinds the result (using the output of ZSPA-A), and sends it to $\mathcal{SC}_{\texttt{JUS}}$.

After all of the clients send their input polynomials to $\mathcal{SC}_{\texttt{JUS}}$, D sends to $\mathcal{SC}_{\texttt{JUS}}$ a *switching polynomial* that will allow $\mathcal{SC}_{\texttt{JUS}}$ to obliviously switch the secret blinding polynomials used by D (during the execution of VOPR) to blind each client's original polynomial $\boldsymbol{\pi}$ to another blinding polynomial that all clients can

generate themselves, by using key mk. The switching polynomial is constructed in a way that does not affect the verification of unforgeable polynomials.

Next, D sends to $\mathcal{SC}_{\texttt{JUS}}$ a secret polynomial, $\boldsymbol{\zeta}$, that will let $\mathcal{SC}_{\texttt{JUS}}$ check unforgeable polynomials' correctness. $\mathcal{SC}_{\texttt{JUS}}$ sums all clients' polynomials and checks if $\boldsymbol{\zeta}$ can divide the sum. $\mathcal{SC}_{\texttt{JUS}}$ accepts the clients' inputs if the polynomial divides the sum; otherwise, it invokes Aud to identify misbehaving parties. In this case, all honest parties' deposit is refunded and the deposit of misbehaving parties is distributed among the honest ones as well. If all clients behave honestly, then each client can locally find the intersection. To do that, it uses mk to locally remove the blinding polynomial from the sum (that the contract generated), then evaluates the unblinded polynomial at each of its set elements and considers an element in the intersection when the evaluation equals zero. Figure 5, in Appendix H, outlines the interaction between parties.

Detailed Description of JUS. Below, we elaborate on how JUS exactly works (see Table 1 for description of the main notations used).

1. All clients in $CL = \{A_1, ..., A_m, D\}$ sign a smart contract: $\mathcal{SC}_{\texttt{JUS}}$ and deploy it to a blockchain. All clients get the deployed contract's address. Also, all clients engage in CT to agree on a secret master key, mk.
2. Each client in CL builds a hash table, HT, and inserts the set elements into it, i.e., $\forall i : \texttt{H}(s_i) = indx$, then $s_i \rightarrow \texttt{HT}_{indx}$. It pads every bin with random dummy elements to d elements (if needed). Then, for every bin, it constructs a polynomial whose roots are the bin's content: $\boldsymbol{\pi} = \prod_{i=1}^{d}(x - s'_i)$, where s'_i is either s_i or a random value.
3. Every client C in $CL \setminus D$, for every bin, agree on $b = 3d + 3$ vectors of pseudorandom blinding factors: $z_{i,j}$, such that the sum of each vector elements is zero, i.e., $\sum_{j=1}^{m} z_{i,j} = 0$, where $0 \leq i \leq b - 1$. To do that, they participate in step 1 of ZSPA-A. By the end of this step, for each bin, they agree on a secret key k (that will be used to generate the zero-sum values) as well as two values stored in $\mathcal{SC}_{\texttt{JUS}}$, i.e., q: the key's hash value and g: a Merkle tree's root. After time t_1, D ensures that all other clients have agreed on the vectors (i.e., all provided "approved" to the contract). If the check fails, it halts.
4. Each client in CL deposits $\ddot{y} + \ddot{ch}$ amount to $\mathcal{SC}_{\texttt{JUS}}$. After time t_2, every client ensures that in total $(\ddot{y} + \ddot{ch}) \cdot (m+1)$ amount has been deposited. Otherwise, it halts and the clients' deposit is refunded.
5. D picks a random polynomial $\boldsymbol{\zeta} \xleftarrow{\$} \mathbb{F}_p[X]$ of degree 1, for each bin. It, for each client C, allocates to each bin two random polynomials: $\boldsymbol{\omega}^{(D,C)}, \boldsymbol{\rho}^{(D,C)} \xleftarrow{\$} \mathbb{F}_p[X]$ of degree d, and two random polynomials: $\boldsymbol{\gamma}^{(D,C)}, \boldsymbol{\delta}^{(D,C)} \leftarrow \mathbb{F}_p[X]$ of degree $3d + 1$. Also, each client C, for each bin, picks two random polynomials: $\boldsymbol{\omega}^{(C,D)}, \boldsymbol{\rho}^{(C,D)} \xleftarrow{\$} \mathbb{F}_p[X]$ of degree d.
6. D randomises other clients' polynomials. To do so, for every bin, it invokes an instance of VOPR (presented in Fig. 1) with each client C; where D sends $\boldsymbol{\zeta} \cdot \boldsymbol{\omega}^{(D,C)}$ and $\boldsymbol{\gamma}^{(D,C)}$, while client C sends $\boldsymbol{\omega}^{(C,D)} \cdot \boldsymbol{\pi}^{(C)}$ to VOPR. Each client C,

for every bin, receives a blind polynomial of the following form:

$$\boldsymbol{\theta}_1^{(C)} = \boldsymbol{\zeta} \cdot \boldsymbol{\omega}^{(D,C)} \cdot \boldsymbol{\omega}^{(C,D)} \cdot \boldsymbol{\pi}^{(C)} + \boldsymbol{\gamma}^{(D,C)}$$

from VOPR. If any party aborts, the deposit would be refunded to all parties.

7. Each client C randomises D's polynomial. To do that, each client C, for each bin, invokes an instance of VOPR with D, where each client C sends $\boldsymbol{\rho}^{(C,D)}$, while D sends $\boldsymbol{\zeta} \cdot \boldsymbol{\rho}^{(D,C)} \cdot \boldsymbol{\pi}^{(D)}$ and $\boldsymbol{\delta}^{(D,C)}$ to VOPR. Every client C, for each bin, receives a blind polynomial of the following form:

$$\boldsymbol{\theta}_2^{(C)} = \boldsymbol{\zeta} \cdot \boldsymbol{\rho}^{(D,C)} \cdot \boldsymbol{\rho}^{(C,D)} \cdot \boldsymbol{\pi}^{(D)} + \boldsymbol{\delta}^{(D,C)}$$

from VOPR. If any party aborts, the deposit would be refunded to all parties.

8. Each client C, for every bin, masks the sum of polynomials $\boldsymbol{\theta}_1^{(C)}$ and $\boldsymbol{\theta}_2^{(C)}$ using the blinding factors: $z_{i,c}$, generated in step 3. Specifically, it computes the following blind polynomial (for every bin):

$$\boldsymbol{\nu}^{(C)} = \boldsymbol{\theta}_1^{(C)} + \boldsymbol{\theta}_2^{(C)} + \boldsymbol{\tau}^{(C)}$$

where $\boldsymbol{\tau}^{(C)} = \sum_{i=0}^{3d+2} z_{i,c} \cdot x^i$. Next, it sends all $\boldsymbol{\nu}^{(C)}$ to $\mathcal{SC}_{\texttt{JUS}}$. If any party aborts, the deposit would be refunded to all parties.

9. D ensures all clients sent their inputs to $\mathcal{SC}_{\texttt{JUS}}$. If the check fails, it halts and the deposit would be refunded to all parties. It allocates a fresh pseudorandom polynomial $\boldsymbol{\gamma}'$ of degree $3d$, to each bin. To do so, it uses mk to derive a key for each bin: $k_{indx} = \texttt{PRF}(mk, indx)$ and then uses the derived key to generate $3d+1$ pseudorandom coefficients $g_{j,indx} = \texttt{PRF}(k_{indx}, j)$ where $0 \leq j \leq 3d$. Also, for each bin, it allocates a fresh random polynomial: $\boldsymbol{\omega}'^{(D)}$ of degree d.
10. D, for every bin, computes a polynomial of the form:

$$\boldsymbol{\nu}^{(D)} = \boldsymbol{\zeta} \cdot \boldsymbol{\omega}'^{(D)} \cdot \boldsymbol{\pi}^{(D)} - \sum_{C=A_1}^{A_m} (\boldsymbol{\gamma}^{(D,C)} + \boldsymbol{\delta}^{(D,C)}) + \boldsymbol{\zeta} \cdot \boldsymbol{\gamma}'$$

It sends to $\mathcal{SC}_{\texttt{JUS}}$ polynomials $\boldsymbol{\nu}^{(D)}$ and $\boldsymbol{\zeta}$, for each bin.

11. $\mathcal{SC}_{\texttt{JUS}}$ takes the following steps:

(a) for every bin, sums all related polynomials provided by all clients in $\bar{P}$:

$$\boldsymbol{\phi} = \boldsymbol{\nu}^{(D)} + \sum_{C=A_1}^{A_m} \boldsymbol{\nu}^{(C)}$$

$$= \boldsymbol{\zeta} \cdot \left(\boldsymbol{\omega}'^{(D)} \cdot \boldsymbol{\pi}^{(D)} + \sum_{C=A_1}^{A_m} (\boldsymbol{\omega}^{(D,C)} \cdot \boldsymbol{\omega}^{(C,D)} \cdot \boldsymbol{\pi}^{(C)}) + \boldsymbol{\pi}^{(D)} \cdot \sum_{C=A_1}^{A_m} (\boldsymbol{\rho}^{(D,C)} \cdot \boldsymbol{\rho}^{(C,D)}) + \boldsymbol{\gamma}' \right)$$

(b) checks whether, for every bin, $\boldsymbol{\zeta}$ divides $\boldsymbol{\phi}$. If the check passes, it sets $Flag = True$. Otherwise, it sets $Flag = False$.

12. If the check passes (i.e., $Flag = True$), then the following steps are taken:
    (a) $\mathcal{SC}_{\texttt{JUS}}$ sends back each party's deposit, i.e., $\ddot{y} + \ddot{ch}$ amount.
    (b) each client (given $\boldsymbol{\zeta}$ and mk) finds the elements in the intersection as follows.
        i. derives a bin's pseudorandom polynomial, $\boldsymbol{\gamma}'$, from mk.
        ii. removes the blinding polynomial from each bin's polynomial:

$$\boldsymbol{\phi}' = \boldsymbol{\phi} - \boldsymbol{\zeta} \cdot \boldsymbol{\gamma}'$$

        iii. evaluates each bin's unblinded polynomial at every element s_i belonging to that bin and considers the element in the intersection if the evaluation is zero: i.e., $\boldsymbol{\phi}'(s_i) = 0$.
13. If the check does not pass (i.e., $Flag = False$), the following steps are taken.

(a) *Aud* asks every C to send to it the PRF's key (generated in step 3), for every bin. It inserts the keys to $\vec{k}$. It generates a list $\bar{L}$ initially empty. Then, for every bin, *Aud* takes step 2 of ZSPA-A, i.e., invokes $\texttt{Audit}(\vec{k}, q, \boldsymbol{\zeta}, 3d+3, g) \rightarrow (L, \vec{\mu})$. Every time it invokes Audit, it appends the elements of returned L to $\bar{L}$. *Aud* for each bin sends $\vec{\mu}$ to $\mathcal{SC}_{\texttt{JUS}}$. It also sends to $\mathcal{SC}_{\texttt{JUS}}$ the list $\bar{L}$ of all misbehaving clients detected so far.

(b) to help identify further misbehaving clients, D takes the following steps, for each bin of client C whose ID is not in $\bar{L}$.

i. computes polynomial $\boldsymbol{\chi}^{(D,C)}$ as follows.
$$\boldsymbol{\chi}^{(D,C)} = \boldsymbol{\zeta} \cdot \boldsymbol{\eta}^{(D,C)} - (\boldsymbol{\gamma}^{(D,C)} + \boldsymbol{\delta}^{(D,C)})$$
where $\boldsymbol{\eta}^{(D,C)}$ is a fresh random polynomial of degree $3d+1$.

ii. sends polynomial $\boldsymbol{\chi}^{(D,C)}$ to $\mathcal{SC}_{\texttt{JUS}}$.

Note, if $\bar{L}$ contains all clients' IDs, then D does not need to take the above steps 13(b)i and 13(b)ii.

(c) $\mathcal{SC}_{\texttt{JUS}}$, takes the following steps to check if the client misbehaved, for each bin of client C whose ID is not in $\bar{L}$.

i. computes polynomial $\boldsymbol{\iota}^{(C)}$ as follows:
$$\begin{aligned}\boldsymbol{\iota}^{(C)} &= \boldsymbol{\chi}^{(D,C)} + \boldsymbol{\nu}^{(C)} + \boldsymbol{\mu}^{(C)} \\ &= \boldsymbol{\zeta} \cdot (\boldsymbol{\eta}^{(D,C)} + \boldsymbol{\omega}^{(D,C)} \cdot \boldsymbol{\omega}^{(C,D)} \cdot \boldsymbol{\pi}^{(C)} + \boldsymbol{\rho}^{(D,C)} \cdot \boldsymbol{\rho}^{(C,D)} \cdot \boldsymbol{\pi}^{(D)} + \boldsymbol{\xi}^{(C)})\end{aligned}$$
where $\boldsymbol{\mu}^{(C)} \in \vec{\mu}$ generated and sent to $\mathcal{SC}_{\texttt{JUS}}$ by *Aud* in step 13a.

ii. checks if $\boldsymbol{\zeta}$ divides $\boldsymbol{\iota}^{(C)}$. If the check fails, it appends the client's ID to a list L'.

If $\bar{L}$ contains all clients' IDs, then $\mathcal{SC}_{\texttt{JUS}}$ does not take the above two steps.

(d) $\mathcal{SC}_{\texttt{JUS}}$ refunds the honest parties' deposit. Also, it retrieves the total amount of $\ddot{ch}$ from the deposit of dishonest clients (i.e., those clients whose IDs are in $\bar{L}$ or L') and sends it to *Aud*. It also splits the remaining deposit of the misbehaving parties among the honest ones. Thus, each honest client receives $\ddot{y} + \ddot{ch} + \frac{m' \cdot (\ddot{y} + \dot{ch}) - \dot{ch}}{m - m'}$ amount in total, where m' is the total number of misbehaving parties.

One may be tempted to replace *Justitia* with a scheme in which all clients send their encrypted sets to a server (potentially semi-honest and plays *Aud*'s role) which computes the result in a privacy-preserving manner. We highlight that the main difference is that in this (hypothetical) scheme the server is *always involved*; whereas, in our protocol, *Aud* remains offline as long as the clients behave honestly and it is invoked only when the contract detects misbehaviours.

Next, we present a theorem that formally states the security of JUS. We refer readers to Appendix I for the proof of this theorem.

Theorem 6. *Let polynomials $\boldsymbol{\zeta}$, $\boldsymbol{\omega}$, and $\boldsymbol{\gamma}$ be three secret uniformly random polynomials. If $\boldsymbol{\theta} = \boldsymbol{\zeta} \cdot \boldsymbol{\omega} \cdot \boldsymbol{\pi} + \boldsymbol{\gamma} \bmod p$ is an unforgeable polynomial (w.r.t. Theorem 4), ZSPA-A, VOPR, PRF, and smart contracts are secure, then JUS securely realises f^{PSI} with Q-fairness (w.r.t. Definition 7) in the presence of $m-1$ active-adversary clients (i.e., A_js) or a passive dealer client, passive auditor, or passive public.*

7 Definition of Multi-party PSI with Fair Compensation and Reward

In this section, we upgrade $\mathcal{PSI}^{\mathcal{FC}}$ to "multi-party PSI with Fair Compensation and Reward" ($\mathcal{PSI}^{\mathcal{FCR}}$), which (in addition to offering the features of $\mathcal{PSI}^{\mathcal{FC}}$) allows honest clients who contribute their set to receive a reward by a buyer who initiates the PSI computation and is interested in the result.

In $\mathcal{PSI}^{\mathcal{FCR}}$, there are (1) a set of clients $\{A_1, ..., A_m\}$ a subset of which is potentially active adversaries and may collude with each other, (2) a non-colluding dealer, D, potentially semi-honest, and (3) an auditor Aud potentially semi-honest, where all clients (except Aud) have input set. Furthermore, in $\mathcal{PSI}^{\mathcal{FCR}}$ there are two "extractor" clients, say A_1 and A_2, where $(A_1, A_2) \in \{A_1, ..., A_m\}$. These extractor clients volunteer to extract the (encoded) elements of the intersection and send them to a public bulletin board, i.e., a smart contract. In return, they will be paid. We assume these two extractors act rationally only when they want to carry out the paid task of extracting the intersection and reporting it to the smart contract, so they can maximise their profit.[3] For simplicity, we let client A_m be the buyer, i.e., the party which initiates the PSI computation and is interested in the result.

The formal definition of $\mathcal{PSI}^{\mathcal{FCR}}$ is built upon the definition of $\mathcal{PSI}^{\mathcal{FC}}$ (presented in Section 4); nevertheless, in $\mathcal{PSI}^{\mathcal{FCR}}$, we ensure that honest non-buyer clients receive a *reward* for participating in the protocol and revealing a portion of their inputs deduced from the result. We: (i) upgrade the predicate Q^{Del} to $Q_{\text{R}}^{\text{Del}}$ to ensure that when honest clients receive the result, then an honest non-buyer client receives its deposit back plus a reward and a buyer client receives its deposit back minus the paid reward, and (ii) upgrade the predicate $Q^{\text{UF-A}}$ to $Q_{\text{R}}^{\text{UF-A}}$ to ensure when an adversary aborts in an unfair manner (i.e., aborts but learns the result) then an honest party receives its deposit back plus a predefined amount of compensation plus a reward. The other two predicates (i.e., Q^{Init} and $Q^{\text{F-A}}$) remain unchanged. Given the above changes, we denote the four predicates as $\bar{Q} := (Q^{\text{Init}}, Q_{\text{R}}^{\text{Del}}, Q_{\text{R}}^{\text{UF-A}}, Q^{\text{F-A}})$. Below, we present the formal definition of predicates $Q_{\text{R}}^{\text{Del}}$ and $Q_{\text{R}}^{\text{UF-A}}$.

Definition 8 ($Q_{\mathbf{R}}^{\mathbf{Del}}$: Delivery-with-Reward predicate). *Let $\mathcal{G}$ be a stable ledger, adr_{sc} be smart contract sc's address, $adr_i \in Adr$ be the address of an honest party, $\ddot{x}$ be a fixed amount of coins, and $pram := (\mathcal{G}, adr_{sc}, \ddot{x})$. Let R be a reward function that takes as input the computation result: res, a party's address: adr_i, a reward a party should receive for each unit of revealed information: $\ddot{l}$, and input size: $inSize$. Then R is defined as follows, if adr_i belongs to a non-buyer, then it returns the total amount that adr_i should be rewarded and if adr_i belongs to a buyer client, then it returns the reward's leftover that the buyer can collect, i.e., $R(res, adr_i, \ddot{l}, inSize) \rightarrow r\ddot{e}w_i$. Then, the delivery with reward predicate $Q_R^{Del}(pram, adr_i, res, \ddot{l}, inSize)$ returns 1 if adr_i has sent $\ddot{x}$ amount to sc and received at least $\ddot{x} + r\ddot{e}w_i$ amount from it. Else, it returns 0.*

[3] Thus, similar to any A_i in $\mathcal{PSI}^{\mathcal{FC}}$, these extractors might be corrupted by an active adversary during the PSI computation.

Definition 9 ($Q_{\mathbf{R}}^{\mathbf{UF\text{-}A}}$: UnFair-Abort-with-Reward predicate). *Let* $pram := (\mathcal{G}, adr_{sc}, \ddot{x})$ *be the parameters defined above, and* $Adr' \subset Adr$ *be a set containing honest parties' addresses,* $m' = |Adr'|$*, and* $adr_i \in Adr'$*. Let also* G *be a compensation function that takes as input three parameters* $(\ddot{deps}, adr_i, m')$*, where* $\ddot{deps}$ *is the amount of coins that all* $m+1$ *parties deposit,* adr_i *is an honest party's address, and* $m' = |Adr'|$*; it returns the amount of compensation each honest party must receive, i.e.,* $G(\ddot{deps}, ard_i, m') \rightarrow \ddot{x}_i$*. Let* R *be the reward function defined above, i.e.,* $R(res, adr_i, \ddot{l}, inSize) \rightarrow r\ddot{e}w_i$*, and let* $\ddot{prâm} := (res, \ddot{l}, inSize)$*. Then, predicate* $Q_R^{UF\text{-}A}$ *is defined as* $Q_R^{UF\text{-}A}(pram, \ddot{prâm}, G, R, \ddot{deps}, m', adr_i) \rightarrow (a, b)$*, where* $a = 1$ *if* adr_i *is an honest party's address which has sent* $\ddot{x}$ *amount to sc and received* $\ddot{x} + \ddot{x}_i + r\ddot{e}w_i$ *from it, and* $b = 1$ *if* adr_i *is an auditor's address which received* $\ddot{x}_i$ *from sc. Otherwise,* $a = b = 0$*.*

Next, we present the formal definition of multi-party PSI with Fair Compensation and Reward, $\mathcal{PSI}^{\mathcal{FCR}}$.

Definition 10 ($\mathcal{PSI}^{\mathcal{FCR}}$). *Let* f^{PSI} *be the multi-party PSI functionality defined in Section 4. We say protocol* Γ *realises* f^{PSI} *with* $\bar{Q}$*-fairness-and-reward in the presence of* $m-3$ *static active-adversary clients* A_j*s and two rational clients* A_i*s or a static passive dealer* D *or passive auditor* Aud*, if for every non-uniform probabilistic polynomial time adversary* $\mathcal{A}$ *for the real model, there exists a non-uniform probabilistic polynomial-time adversary (or simulator)* Sim *for the ideal model, such that for every* $I \in \{A_1, ..., A_m, D, Aud\}$*, it holds that:*

$$\{\mathsf{Ideal}^{\mathcal{W}(f^{PSI}, \bar{Q})}_{\mathsf{Sim}(z), I}(S_1, ..., S_{m+1})\}_{S_1, ..., S_{m+1}, z} \overset{c}{\equiv} \{\mathsf{Real}^{\Gamma}_{\mathcal{A}(z), I}(S_1, ..., S_{m+1})\}_{S_1, ..., S_{m+1}, z}$$

where z *is an auxiliary input given to* $\mathcal{A}$ *and* $\mathcal{W}(f^{PSI}, \bar{Q})$ *is a functionality that wraps* f^{PSI} *with predicates* $\bar{Q} := (Q^{Init}, Q_R^{Del}, Q_R^{UF\text{-}A}, Q^{F\text{-}A})$*.*

8 Anesidora: A Concrete Construction of $\mathcal{PSI}^{\mathcal{FCR}}$

8.1 Main Challenges to Overcome

Rewarding Clients Proportionate to the Intersection Cardinality. In PSIs, the main private information about the clients which is revealed to a result recipient is the private set elements that the clients have in common. Thus, honest clients must receive a reward proportionate to the intersection cardinality, from a buyer. To receive the reward, the clients need to reach a consensus on the intersection cardinality. The naive way to do that is to let every client find the intersection and declare it to the smart contract. Under the assumption that the majority of clients are honest, then the smart contract can reward the honest result recipient (from the buyer's deposit). Nevertheless, the honest majority assumption is strong in the context of multi-party PSI. Moreover, this approach requires all clients to extract the intersection, which would increase the overall costs. Some clients may not even be interested in or available to do so. This task could also be conducted by a single entity, such as the dealer; but this approach would introduce a single point of failure and all clients have to depend on this entity. To address these challenges, we allow any two clients to become extractors. Each of them finds and sends to the contract the (encrypted) elements in the intersection. It is paid by the contract if the contract concludes that it is honest.

This allows us to avoid (i) the honest majority assumption, (ii) requiring all clients to find the intersection, and (iii) relying on a single trusted/semi-honest party to complete the task.

Dealing with Extractors' Collusion. Using two extractors itself introduces another challenge; namely, they may collude with each other (and with the buyer) to provide a consistent but incorrect result, e.g., both may declare that only s_1 is in the intersection while the actual intersection contains 100 set elements, including s_1. This behaviour will not be detected by a verifier unless the verifier always conducts the delegated task itself too, which would defeat the purpose of delegation. To efficiently address this issue, we use the counter-collusion smart contracts (outlined in Section 3.4) which creates distrust between the two extractors and incentivises them to act honestly.

8.2 Description of Anesidora (ANE)

An Overview. To construct ANE, we mainly use JUS, deterministic encryption, "double-layered" commitments, the hash-based padding technique (from Section 3.9), and the counter-collusion smart contracts. At a high level, ANE works as follows. First, all clients run step 1 of JUS to agree on a set of parameters and JUS's smart contract. They deploy another smart contract, say $\mathcal{SC}_{\texttt{ANE}}$. They also agree on a secret key, mk'. Next, the buyer places a certain deposit into $\mathcal{SC}_{\texttt{ANE}}$. This deposit will be distributed among honest clients as a reward. The extractors and D deploy one of the counter-collusion smart contracts, i.e., $\mathcal{SC}_{\text{PC}}$. These three parties deposit a certain amount on this contract. Each honest extractor will receive a portion of D's deposit for carrying out its task honestly and each dishonest extractor will lose a portion of its deposit for acting maliciously. Then, each client encrypts its set elements (under mk' using deterministic encryption) and then represents the encrypted elements as a polynomial. The reason each client encrypts its set elements is to ensure that the privacy of the plaintext elements in the intersection will be preserved from the public.

Next, the extractors commit to the encryption of their set elements and publish the commitments. All clients (including D) take the rest of the steps in JUS using their input polynomials. This results in a blinded polynomial, whose correctness is checked by JUS's smart contact.

If JUS's smart contact approves the result's correctness, then all parties receive the money that they deposited in JUS's contract. In this case, each extractor finds the set elements in the intersection. Each extractor proves to $\mathcal{SC}_{\texttt{ANE}}$ that the encryptions of the elements in the intersection are among the commitments that the extractor previously published. If $\mathcal{SC}_{\texttt{ANE}}$ accepts both extractors' proofs, then it pays each client (except the buyer) a reward, where the reward is taken from the buyer's deposit. The extractors receive their deposits back and are paid for carrying out the task honestly. Nevertheless, if $\mathcal{SC}_{\texttt{ANE}}$ does not accept one of the extractors' proofs (or one extractor betrays the other), then it invokes the auditor in the counter-collusion contracts to identify the misbehaving extractor. Then, $\mathcal{SC}_{\texttt{ANE}}$ pays each honest client (except the buyer) a reward, taken from the misbehaving extractor. $\mathcal{SC}_{\texttt{ANE}}$ also refunds the buyer's deposit.

If JUS's smart contact does not approve the result's correctness and *Aud* identified misbehaving clients, then honest clients will receive (1) their deposit

back from JUS's contract, and (2) compensation and reward, taken from misbehaving clients. Moreover, the buyer and extractors receive their deposit back from $\mathcal{SC}_{\texttt{ANE}}$. Figure 6, in Appendix J, outlines the interaction between parties.

Detailed Description of ANE. Next, we describe the protocol in more detail (Table 1 summarises the main notations used).

1. All clients in $CL = \{A_1, ..., A_m, D\}$ together run step 1 of JUS (in Section 6.2) to deploy JUS's contract $\mathcal{SC}_{\texttt{JUS}}$ and agree on a master key, mk.
2. All clients in CL deploy a new smart contract, $\mathcal{SC}_{\texttt{ANE}}$. The address of $\mathcal{SC}_{\texttt{ANE}}$ is given to all clients.
3. The buyer, client A_m, before time t_1 deposits $S_{min} \cdot \ddot{v}$ amount to $\mathcal{SC}_{\texttt{ANE}}$.
4. All clients after time $t_2 > t_1$ ensure that the buyer has deposited $S_{min} \cdot \ddot{v}$ amount on $\mathcal{SC}_{\texttt{ANE}}$. Otherwise, they abort.
5. D signs $\mathcal{SC}_{\text{PC}}$ with the extractors. $\mathcal{SC}_{\texttt{ANE}}$ transfers $S_{min} \cdot \ddot{r}$ amount (from the buyer deposit) to $\mathcal{SC}_{\text{PC}}$ for each extractor. This is the maximum amount to be paid to an honest extractor for honestly declaring the elements of the intersection. Each extractor deposits $\ddot{d}' = \ddot{d} + S_{min} \cdot \ddot{f}$ amount in $\mathcal{SC}_{\text{PC}}$ at time t_3. At time t_4 all clients ensure that the extractors deposited enough coins; otherwise, they withdraw their deposit and abort.
6. D encrypts mk under the public key of the dispute resolver (in $\mathcal{SC}_{\text{PC}}$); let ct_{mk} be the ciphertext. It also generates a commitment of mk as follows: $z' = \texttt{PRF}(mk, 0),\ com_{mk} = \texttt{Com}(mk, z')$. It stores ct_{mk} and com_{mk} in $\mathcal{SC}_{\texttt{ANE}}$.
7. All clients in CL engage in CT to agree on another key, mk'.
8. Each client in CL maps the elements of its set $S : \{s_1, ..., s_c\}$ to random values by encrypting them as: $\forall i, 1 \leq i \leq c : e_i = \texttt{PRP}(mk', s_i)$. Then, it encodes its encrypted set element as $\bar{e}_i = e_i || \texttt{H}(e_i)$. After that, it constructs a hash table HT and inserts the encoded elements into the table. $\forall i : \texttt{H}(\bar{e}_i) = j$, then $\bar{e}_i \rightarrow \texttt{HT}_j$. It pads every bin with random dummy elements to d elements (if needed). Then, for every bin, it builds a polynomial whose roots are the bin's content: $\boldsymbol{\pi}^{(I)} = \prod_{i=1}^{d}(x - e'_i)$, where e'_i is either $\bar{e}_i$, or a dummy value.
9. Every extractor in $\{A_1, A_2\}$:
   (a) for each j-th bin, commits to the bin's elements: $com_{i,j} = \texttt{Com}(e'_i, q_i)$, where q_i is a fresh randomness used for the commitment and e'_i is either $\bar{e}_i$, or a dummy value of the bin.
   (b) constructs a Merkel tree on all committed values as follows: $\texttt{MT.genTree}(com_{1,1}, ..., com_{d,h}) \rightarrow g$.
   (c) stores the Merkel tree's root g on $\mathcal{SC}_{\texttt{ANE}}$.
10. All clients in CL run steps 3–11 of JUS, where each client now deposits (in the $\mathcal{SC}_{\texttt{JUS}}$) $\ddot{y}'$ amount where $\ddot{y}' > S_{min} \cdot \ddot{v} + \ddot{ch}$. Recall, at the end of step 11 of JUS for each j-th bin (i) a random polynomial $\boldsymbol{\zeta}$ has been registered in $\mathcal{SC}_{\texttt{JUS}}$, (ii) a polynomial $\boldsymbol{\phi}$ (blinded by a random polynomial $\boldsymbol{\gamma}'$) has been extracted by $\mathcal{SC}_{\texttt{JUS}}$, and (iii) $\mathcal{SC}_{\texttt{JUS}}$ has checked this polynomial's correctness. If the latter check:

- passes (i.e., $Flag = True$): all parties run step 12 of JUS (with a minor difference, see Section L). In this case, each party receives $\ddot{y}'$ amount it deposited in $\mathcal{SC}_{\texttt{JUS}}$. They proceed to step 11 below.
- fails (i.e., $Flag = False$): all parties run step 13 of JUS. In this case, (as in JUS) Aud is paid $\ddot{ch}$ amount, and each honest party receives back its deposit, i.e., $\ddot{y}'$ amount. Also, from the misbehaving parties' deposit $\frac{m' \cdot \ddot{y}' - \ddot{ch}}{m - m'}$ amount is sent to each honest client, to reward and compensate the client $S_{min} \cdot \ddot{l}$ and $\frac{m' \cdot \ddot{y}' - \ddot{ch}}{m - m'} - S_{min} \cdot \ddot{l}$ amounts respectively, where m' is the total number of misbehaving parties. Moreover, $\mathcal{SC}_{\texttt{ANE}}$ returns to the buyer its deposit (i.e., $S_{min} \cdot \ddot{v}$ amount paid to $\mathcal{SC}_{\texttt{ANE}}$), and returns to each extractor its deposit, i.e., $\ddot{d}'$ amount paid to $\mathcal{SC}_{\text{PC}}$. Then, the protocol halts.

11. Every extractor client:
   (a) finds the elements in the intersection. To do so, it first encodes each of its set elements to get $\bar{e}_i$, as explained in step 8. Then, it determines to which bin the encrypted value belongs, i.e., $j = \texttt{H}(\bar{e}_i)$. Next, it evaluates the resulting polynomial (for that bin) at the encrypted element. It considers the element in the intersection if the evaluation is zero, i.e., $\boldsymbol{\phi}(\bar{e}_i) - \boldsymbol{\zeta}(\bar{e}_i) \cdot \boldsymbol{\gamma}'(\bar{e}_i) = 0$. If the extractor is a traitor, by this point it should have signed $\mathcal{SC}_{\text{TC}}$ with D and provided all the inputs (e.g., correct result) to $\mathcal{SC}_{\text{TC}}$.
   (b) proves that every element in the intersection is among the elements it has committed to. Specifically, for each element in the intersection, say $\bar{e}_i$, it sends to $\mathcal{SC}_{\texttt{ANE}}$:
      - commitment $com_{i,j}$ (for $\bar{e}_i$) and its opening $\hat{x}' := (\bar{e}_i, q_i)$.
      - proof h_i asserting $com_{i,j}$ is a leaf node of a Merkel tree with root g.
   (c) sends the opening of commitment com_{mk}, i.e., pair $\hat{x} := (mk, z')$, to $\mathcal{SC}_{\texttt{ANE}}$. This is done only once for all elements in the intersection.
12. Contract $\mathcal{SC}_{\texttt{ANE}}$:
   (a) verifies the opening of the commitment for mk, i.e., $\texttt{Ver}(com_{mk}, \hat{x}) = 1$. If accepted, then it generates the bin's index to which $\bar{e}_i$ belongs, i.e., $j = \texttt{H}(\bar{e}_i)$. It uses mk to derive the pseudorandom polynomial $\boldsymbol{\gamma}'$ for j-th bin.
   (b) checks whether (i) the opening of commitment is valid, (ii) the Merkle tree proof is valid, and (iii) the encrypted element is the resulting polynomial's root. Specifically, it ensures that the following relation holds:
$$\left(\texttt{Ver}(com_{i,j}, \hat{x}') = 1\right) \wedge \left(\texttt{MT.verify}(h_i, g) = 1\right) \wedge \left(\boldsymbol{\phi}(\bar{e}_i) - \boldsymbol{\zeta}(\bar{e}_i) \cdot \boldsymbol{\gamma}'(\bar{e}_i) = 0\right)$$
13. The parties are paid as follows.
   - if all proofs of extractors are valid, both extractors provided identical elements of the intersections (for each bin), and there is no traitor, then $\mathcal{SC}_{\texttt{ANE}}$:
      (a) takes $|S_\cap| \cdot m \cdot \ddot{l}$ amount from the buyer's deposit (in $\mathcal{SC}_{\texttt{ANE}}$) and distributes it among all clients, except the buyer.
      (b) calls $\mathcal{SC}_{\text{PC}}$ which returns the extractors' deposit (i.e., $\ddot{d}'$ amount each) and pays each extractor $|S_\cap| \cdot \ddot{r}$ amount, for doing their job correctly.
      (c) checks if $|S_\cap| < S_{min}$. If the check passes, then it returns $(S_{min} - |S_\cap|) \cdot \ddot{v}$ amount to the buyer.

- if both extractors failed to deliver any result, then $\mathcal{SC}_{\texttt{ANE}}$:
  (a) refunds the buyer, by sending $S_{min} \cdot \ddot{v}$ amount (deposited in $\mathcal{SC}_{\texttt{ANE}}$) back to the buyer.
  (b) retrieves each extractor's deposit (i.e., $\ddot{d}$ amount) from $\mathcal{SC}_{\text{PC}}$ and distributes it among the rest of the clients (except the buyer and extractors).
- Otherwise (e.g., if some proofs are invalid, if an extractor's result is inconsistent with the other extractor's result, or there is a traitor), $\mathcal{SC}_{\texttt{ANE}}$ invokes (steps 8.c and 9 of) $\mathcal{SC}_{\text{PC}}$ and its auditor to identify the misbehaving extractor, with the help of ct_{mk} after decrypting it. Then, $\mathcal{SC}_{\texttt{ANE}}$ asks $\mathcal{SC}_{\text{PC}}$ to pay the auditor the total amount of $\ddot{ch}$ taken from the deposit of the extractor(s) who provided incorrect result to $\mathcal{SC}_{\texttt{ANE}}$. Moreover,
  (a) if <u>both extractors cheated</u>:
    i. if there <u>is no traitor</u>, then $\mathcal{SC}_{\texttt{ANE}}$ refunds the buyer, by sending $S_{min} \cdot \ddot{v}$ amount (deposited in $\mathcal{SC}_{\texttt{ANE}}$) back to the buyer. It also distributes $2 \cdot \ddot{d}' - \ddot{ch}$ amount (taken from the extractors' deposit in $\mathcal{SC}_{\text{PC}}$) among the rest of clients (except the buyer and extractors).
    ii. if there <u>is a traitor</u>, then:
      A. if the traitor delivered a <u>correct result</u> in $\mathcal{SC}_{\text{TC}}$, $\mathcal{SC}_{\texttt{ANE}}$ retrieves $\ddot{d}' - \ddot{d}$ amount from the other dishonest extractor's deposit (in $\mathcal{SC}_{\text{PC}}$) and distributes it among the rest of the clients (except the buyer and dishonest extractor). Also, it asks $\mathcal{SC}_{\text{PC}}$ to send $|S_{\cap}| \cdot \ddot{r} + \ddot{d}' + \ddot{d} - \ddot{ch}$ amount to the traitor (via $\mathcal{SC}_{\text{TC}}$). $\mathcal{SC}_{\text{TC}}$ refunds the traitor's deposit, i.e., $\ddot{ch}$ amount. It refunds the buyer, by sending $S_{min} \cdot \ddot{v} - |S_{\cap}| \cdot \ddot{r}$ amount (deposited in $\mathcal{SC}_{\texttt{ANE}}$) back to it.
      B. if the traitor delivered an <u>incorrect result</u> in $\mathcal{SC}_{\text{TC}}$, $\mathcal{SC}_{\texttt{ANE}}$ pays the buyer and rest of clients in the same way it does in step 13(a)i. $\mathcal{SC}_{\text{TC}}$ refunds the traitor, i.e., $\ddot{ch}$ amount.
  (b) if <u>one of the extractors cheated</u>:
    i. if there <u>is no traitor</u>, $\mathcal{SC}_{\texttt{ANE}}$ calls $\mathcal{SC}_{\text{PC}}$ that (a) returns the honest extractor's deposit (i.e., $\ddot{d}'$ amount), (b) pays this extractor $|S_{\cap}| \cdot \ddot{r}$ amount, for doing its job, and (c) pays this extractor $\ddot{d} - \ddot{ch}$ amount taken from the dishonest extractor's deposit. $\mathcal{SC}_{\texttt{ANE}}$ pays the buyer and the rest of the clients in the same way it does in step 13(a)iiA.
    ii. if there <u>is a traitor</u>
      A. if the traitor delivered a <u>correct result</u> in $\mathcal{SC}_{\text{TC}}$ (but it cheated in $\mathcal{SC}_{\texttt{ANE}}$), then $\mathcal{SC}_{\texttt{ANE}}$ calls $\mathcal{SC}_{\text{PC}}$ that (a) returns the other honest extractor's deposit (i.e., $\ddot{d}'$ amount), (b) pays the honest extractor $|S_{\cap}| \cdot \ddot{r}$ amount taken from the buyer's deposit, for doing its job honestly, (c) pays the honest extractor $\ddot{d} - \ddot{ch}$ amount taken from the traitor's deposit, (d) pays to the traitor $|S_{\cap}| \cdot \ddot{r}$ amount taken from the buyer's deposit (via the $\mathcal{SC}_{\text{TC}}$), and (e) refunds the traitor $\ddot{d}' - \ddot{d}$ amount taken from its own deposit. $\mathcal{SC}_{\text{TC}}$ refunds the traitor's deposit (i.e., $\ddot{ch}$ amount). $\mathcal{SC}_{\texttt{ANE}}$ takes $|S_{\cap}| \cdot m \cdot \ddot{l}$ amount from the buyer's deposit (in $\mathcal{SC}_{\texttt{ANE}}$) and distributes it among all clients, except the buyer. If $|S_{\cap}| < S_{min}$, then $\mathcal{SC}_{\texttt{ANE}}$ returns $(S_{min} - |S_{\cap}|) \cdot \ddot{v}$ amount (deposited in $\mathcal{SC}_{\texttt{ANE}}$) back to the buyer.

B. if the traitor delivered an incorrect result in $\mathcal{SC}_{\text{TC}}$ (and it cheated in $\mathcal{SC}_{\texttt{ANE}}$), then $\mathcal{SC}_{\texttt{ANE}}$ pays the honest extractor in the same way it does in step 13(b)iiA. $\mathcal{SC}_{\text{TC}}$ refunds the traitor's deposit, i.e., $\ddot{ch}$ amount. Also, $\mathcal{SC}_{\texttt{ANE}}$ pays the buyer and the rest of the clients in the same way it does in step 13(a)iiA.

Theorem 7. *If* PRP, PRF, *the commitment scheme, smart contracts, the Merkle tree scheme,* JUS *and the counter-collusion contracts are secure and the public key encryption is semantically secure, then* ANE *realises* f^{PSI} *with* $\bar{Q}$*-fairness-and-reward (w.r.t. Definition 10) in the presence of* $m-3$ *static active-adversary clients* A_j*s and two rational clients* A_i*s or a static passive dealer* D *or passive auditor Aud, or passive public which sees the intersection cardinality.*

We refer readers to Appendices K and L for the proof of Theorem 7 and several remarks on the ANE respectively.

9 Evaluation

In this section, we analyse the asymptotic costs of ANE. We also compare its costs and features with the fastest two and multiple parties PSIs in [2,31,35,37]) and with the fair PSIs in [13,15]. Tables 2 and 3 summarise the result of the cost analysis and the comparison respectively.

Table 2: Asymptotic costs of different parties in ANE. In the table, h is the total number of bins, d is a bin's capacity (i.e., $d = 100$), m is the total number of clients (excluding D), $|S|$ is a set cardinality, and $\bar{\xi}$ is OLE's security parameter.

Party	Computation Cost	Communication Cost
Client $A_3, ..., A_m$	$O\Big(h \cdot d(m+d) + \lvert S\rvert(\frac{d^2+d}{2})\Big)$	$O\Big(h \cdot d^2 \cdot \bar{\xi}\Big)$
Dealer D	$O\Big(h \cdot m(d^2+d) + \lvert S\rvert(\frac{d^2+d}{2})\Big)$	$O\Big(h \cdot d^2 \cdot \bar{\xi} \cdot m\Big)$
Auditor Aud	$O\Big(h \cdot m \cdot d\Big)$	$O\Big(h \cdot d\Big)$
Extractor A_1, A_2	$O\Big(h \cdot d(m+d) + \lvert S\rvert(\frac{d^2+d}{2})\Big)$	$O\Big(\lvert S_\cap\rvert \cdot \log_2 \lvert S\rvert\Big)$
Smart contract $\mathcal{SC}_{\texttt{ANE}}$ & $\mathcal{SC}_{\texttt{JUS}}$	$O\Big(\lvert S_\cap\rvert(d + \log_2 \lvert S\rvert) + h \cdot m \cdot d\Big)$	—
Overal Complexity	$O\Big(h \cdot d^2 \cdot m\Big)$	$O\Big(h \cdot d^2 \cdot \bar{\xi} \cdot m\Big)$

Table 3: Comparison of the asymptotic complexities and features of state-of-the-art PSIs. In the table, t is a parameter that determines the maximum number of colluding parties, κ is a security parameter, and c is a set cardinality.

Schemes	Asymptotic Cost		Features				
	Computation	Communication	Fairness	Rewarding	Sym-key based	Multi-party	Active Adversary
[2]	$O(h \cdot d^2 \cdot m)$	$O(h \cdot d \cdot m)$	×	×	✓	✓	×
[13]	$O(c)$	$O(c)$	✓	×	×	×	✓
[15]	$O(c^2)$	$O(c)$	✓	×	×	×	✓
[31]	$O(c \cdot m^2 + c \cdot m)$	$O(c \cdot m^2)$	×	×	✓	✓	×
[35]	$O(c \cdot \kappa(m + t^2 - t(m+1)))$	$O(c \cdot m \cdot \kappa)$	×	×	✓	✓	✓
[37]	$O(c)$	$O(c \cdot \kappa)$	×	×	✓	×	✓
Ours: ANE	$O(h \cdot d^2 \cdot m)$	$O(h \cdot d^2 \cdot \bar{\xi} \cdot m)$	✓	✓	✓	✓	✓

9.1 Computation Cost

In step 1, each client's cost is $O(m)$ and mainly involves an invocation of CT. In steps 2–5, the clients' cost is negligible as it involves deploying smart contracts and reading from the deployed contracts. Step 6 involves only D whose cost in this step is constant, as it involves invoking a public key encryption, PRF, and commitment only once. In step 7, the clients' cost is $O(m)$, as they need to invoke an instance of CT. In step 8, each client invokes PRP and H linear with its set's cardinality. In the same step, it also constructs h polynomials, where the construction of each polynomial involves d modular multiplications and additions. Thus, its complexity in this step is $O(h \cdot d)$. It has been shown in [2] that $O(h \cdot d) = O(c)$ and $d = 100$ for all set sizes. In step 9, each extractor invokes the commitment scheme linear with the number of its set cardinality $|S|$ and constructs a Merkle tree on top of the commitments. Therefore, its complexity is $O(|S|)$.

In step 10, each client $A_1, ..., A_m$: (i) invokes an instance of ZSPA-A which involves $O(h \cdot m)$ invocation of CT, $3h \cdot m(d+1)$ invocation of PRF, $3h \cdot m(d+1)$ addition, and $O(h \cdot m \cdot d)$ invocation of H (in step 3 of subroutine JUS), (ii) invokes $2h$ instances of VOPR, where each VOPR invocation involves $2d(1+d)$ invocations of OLE^+, multiplications, and additions (in steps 6 and 7 of JUS), and (iii) performs $h(3d+2)$ modular addition (in step 8 of JUS). The dealer D: (a) invokes $2h \cdot m$ instances of VOPR (in steps 6 and 7 of JUS), (b) invokes PRF $h(3d+1)$ times (in step 9 of JUS), and (c) performs $h(d^2+1)$ multiplications and $3h \cdot m \cdot d$ additions (in step 10 of JUS). In the same step, the subroutine smart contract $\mathcal{SC}_{\text{JUS}}$ performs $h \cdot m(3d+1)$ additions and h polynomial divisions, where each division includes dividing a polynomial of degree $3d+1$ by a polynomial of degree 1 (in step 11 of JUS). Moreover, if $Flag = True$, then each client invokes PRF $h(3d+1)$ times, and performs $h(3d+1)$ additions, and performs polynomial evaluations linear with its set cardinality, where each evaluation involves $O(d)$ additions and $O(\frac{d^2+d}{2})$ multiplications (in step 12 of JUS). If $Flag = False$, then (a) Aud invokes PRF $3h \cdot m(d+1)$ times and invokes H $O(h \cdot m \cdot d)$ times, and (b) D performs $O(h \cdot m \cdot d)$ multiplications and additions (in step 13 of JUS).

In step 11, each extractor invokes H linear with its set cardinality $|S|$; it also performs polynomial evaluations linear with $|S|$. In step 12a, $\mathcal{SC}_{\text{ANE}}$ invokes the commitment's verification algorithm Ver once, H at most $|S_\cap|$ times, and PRF $|S_\cap|(3d+1)$ times. In step 12b, $\mathcal{SC}_{\text{ANE}}$ invokes Ver at most $|S_\cap|$ times, and calls H $O(|S_\cap| \cdot \log_2 |S|)$ times. In the same step, it performs polynomial evaluation $|S_\cap|$ times. Thus, its overall complexity is $O(|S_\cap|(d + \log_2 |S|))$.

9.2 Communication Cost

In steps 1 and 7, the communication cost of the clients is dominated by the cost of CT which is $O(m)$. In steps 2–6, the clients' cost is negligible, as it involves sending a few transactions to the smart contracts, e.g., $\mathcal{SC}_{\text{JUS}}$, $\mathcal{SC}_{\text{ANE}}$, and $\mathcal{SC}_{\text{PC}}$. Step 9 involves only extractors whose cost is $O(h)$ as each of them only sends to $\mathcal{SC}_{\text{ANE}}$ a single value for each bin. In step 10, the clients' cost is dominated by VOPR's cost; specifically, each pair of client and D invokes VOPR $O(d^2)$ times for

each bin; therefore, the cost of each client (excluding D) is $O(h \cdot d^2 \cdot \bar{\xi})$ while the cost of D is $O(h \cdot d^2 \cdot \bar{\xi} \cdot m)$, where $\bar{\xi}$ is the subroutine OLE's security parameter. Step 11 involves only the extractors, where each extractor's cost is dominated by the size of the Merkle tree's proof it sends to $\mathcal{SC}_{\texttt{ANE}}$, i.e., $O(|S_{\cap}| \cdot \log_2 |S|)$, where $|S|$ is the extractor's set cardinality. In step 13, Aud sends h polynomials of degree $3d + 1$ to $\mathcal{SC}_{\texttt{JUS}}$; thus, its complexity is $O(h \cdot d)$. The rest of the steps impose negligible communication costs.

9.3 Comparison

Below we show that ANE offers various features that the state-of-the-art PSIs do not offer simultaneously while keeping its overall overheads similar to the efficient PSIs.

Computation Complexity. The computation complexity of ANE is similar to that of PSI in [2], but is better than the multiparty PSI's complexity in [31] as the latter's complexity is quadratic with the number of parties. Also, ANE's complexity is better than the complexity of the PSI in [35] that is quadratic with parameter t, i.e., the total number of parties that may collude. Similar to the two-party PSIs in [13,37], ANE's complexity is linear with c. The two-party PSI in [15] imposes a higher computation overhead than ANE does, as its complexity is quadratic with sets' cardinality. Hence, the complexity of ANE is: (i) linear with the set cardinality, similar to the above schemes except the one in [15] and (ii) linear with the total number of parties, similar to the above multi-party schemes, except the one in [31]. Hence, the computation complexity of ANE is linear with the set cardinality and the number of parties, similar to the above schemes except for the ones in [31,15] whose complexities are quadratic with the set cardinality or the number of parties.

Communication Complexity. ANE's communication complexity is slightly higher than the complexity of the PSI in [2], by a factor of $d \cdot \bar{\xi}$. However, it is better than the PSI's complexity in [31] as the latter has a complexity quadratic with the number of parties. ANE's complexity is slightly higher than the one in [35], by a factor of d. Similar to the two-party PSIs in [13,37,15], ANE's complexity is linear with c. Therefore, the communication complexity of ANE is linear with the set cardinality and number of parties, similar to the above schemes except the one in [31] whose complexity is quadratic with the number of parties.

Features. ANE is the only scheme that offers all the five features, i.e., supports fairness, rewards participants, is based on symmetric key primitives, supports multi-party, and is secure against active adversaries. After ANE is the scheme in [35] which offers three of the above features. The rest of the schemes support only two of the above features.

10 Conclusion and Future Direction

PSI is a crucial protocol with numerous real-world applications. In this work, we proposed, Justitia, the first multi-party fair PSI that ensures that either all parties get the result or if the protocol aborts in an unfair manner, then honest parties will receive financial compensation. We then upgraded it to Anesidora,

the first PSI ensuring that honest parties who contribute their private sets receive a reward proportionate to the number of elements they reveal. Since an MPC that rewards participants for contributing their private inputs would help increase its real-world adoption, an interesting open question is: *How can we generalise the idea of rewarding participants to MPC?*

References

1. Abadi, A., Terzis, S., Metere, R., Dong, C.: Efficient delegated private set intersection on outsourced private datasets. IEEE TDSC (2018)
2. Abadi, A., Dong, C., Murdoch, S.J., Terzis, S.: Multi-party updatable delegated private set intersection. In: FC (2022)
3. Abadi, A., Murdoch, S.J., Zacharias, T.: Polynomial representation is tricky: Maliciously secure private set intersection revisited. IACR Cryptol. ePrint Arch. (2021)
4. Beimel, A., Omri, E., Orlov, I.: Protocols for multiparty coin toss with dishonest majority. In: CRYPTO (2010)
5. Ben-Efraim, A., Nissenbaum, O., Omri, E., Paskin-Cherniavsky, A.: Psimple: Practical multiparty maliciously-secure private set intersection, eprint arch. (2021)
6. Berenbrink, P., Czumaj, A., Steger, A., Vöcking, B.: Balanced allocations: the heavily loaded case. In: STOC (2000)
7. Blum, M.: Coin flipping by telephone - A protocol for solving impossible problems. In: COMPCON'82 (1982)
8. Boneh, D., Gentry, C., Halevi, S., Wang, F., Wu, D.J.: Private database queries using somewhat homomorphic encryption. In: ACNS (2013)
9. Brickell, J., Porter, D.E., Shmatikov, V., Witchel, E.: Privacy-preserving remote diagnostics. In: CCS (2007)
10. Chandran, N., Dasgupta, N., Gupta, D., Obbattu, S.L.B., Sekar, S., Shah, A.: Efficient linear multiparty PSI and extensions to quorum PSI. In: CCS (2021)
11. Cheng, Y., Liu, Y., Chen, T., Yang, Q.: Federated learning for privacy-preserving AI. Commun. ACM (2020)
12. Debnath, S.K., Dutta, R.: A fair and efficient mutual private set intersection protocol from a two-way oblivious pseudorandom function. In: ICISC (2014)
13. Debnath, S.K., Dutta, R.: New realizations of efficient and secure private set intersection protocols preserving fairness. In: ICISC (2016)
14. Debnath, S.K., Dutta, R.: Towards fair mutual private set intersection with linear complexity. Secur. Commun. Networks (2016)
15. Dong, C., Chen, L., Camenisch, J., Russello, G.: Fair private set intersection with a semi-trusted arbiter. In: DBSec (2013)
16. Dong, C., Wang, Y., Aldweesh, A., McCorry, P., van Moorsel, A.: Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. In: ACM CCS (2017)
17. Dorn, W.S.: Generalizations of horner's rule for polynomial evaluation. IBM Journal of Research and Development (1962)
18. Duong, T., Phan, D.H., Trieu, N.: Catalic: Delegated PSI cardinality with applications to contact tracing. In: ASIACRYPT (2020)
19. Freedman, M.J., Nissim, K., Pinkas, B.: Efficient private matching and set intersection. In: EUROCRYPT (2004)
20. Ghosh, S., Nielsen, J.B., Nilges, T.: Maliciously secure oblivious linear function evaluation with constant overhead. In: ASIACRYPT (2007)

21. Ghosh, S., Nilges, T.: An algebraic approach to maliciously secure private set intersection. In: EUROCRYPT (2019)
22. Goldreich, O.: The Foundations of Cryptography - Volume 2, Basic Applications. Cambridge University Press (2004)
23. Inbar, R., Omri, E., Pinkas, B.: Efficient scalable multiparty private set-intersection via garbled bloom filters. In: SCN (2018)
24. Ion, M., Kreuter, B., Nergiz, A.E., Patel, S., Saxena, S., Seth, K., Raykova, M., Shanahan, D., Yung, M.: On deploying secure computing: Private intersection-sum-with-cardinality. In: EuroS&P (2020)
25. Kamara, S., Mohassel, P., Raykova, M., Sadeghian, S.: Scaling private set intersection to billion-element sets. In: FC (2014)
26. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. CRC Press (2007)
27. Kerschbaum, F.: Outsourced private set intersection using homomorphic encryption. In: ASIACCS (2012)
28. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: CRYPTO (2017)
29. Kiayias, A., Zhou, H., Zikas, V.: Fair and robust multi-party computation using a global transaction ledger. In: Fischlin, M., Coron, J. (eds.) EUROCRYPT (2016)
30. Kissner, L., Song, D.X.: Privacy-preserving set operations. In: CRYPTO (2005)
31. Kolesnikov, V., Matania, N., Pinkas, B., Rosulek, M., Trieu, N.: Practical multi-party private set intersection from symmetric-key techniques. In: CCS (2017)
32. Lu, L., Ding, N.: Multi-party private set intersection in vertical federated learning. In: TrustCom (2020)
33. Moran, T., Naor, M., Segev, G.: An optimally fair coin toss. In: TCC (2009)
34. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Tech. rep. (2019)
35. Nevo, O., Trieu, N., Yanai, A.: Simple, fast malicious multiparty private set intersection. In: CCS (2021)
36. Papamanthou, C., Tamassia, R., Triandopoulos, N.: Authenticated hash tables. In: ACM CCS (2008)
37. Raghuraman, S., Rindal, P.: Blazing fast PSI from improved OKVS and subfield VOLE. In: CCS (2022)
38. Tamrakar, S., Liu, J., Paverd, A., Ekberg, J., Pinkas, B., Asokan, N.: Scalable private membership test using trusted hardware. In: AsiaCCS (2017)
39. Thomas, K., Pullman, J., Yeo, K., Raghunathan, A., Kelley, P.G., Invernizzi, L., Benko, B., Pietraszek, T., Patel, S., Boneh, D., Bursztein, E.: Protecting accounts from credential stuffing with password breach alerting. In: USENIX Security (2019)
40. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper (2014)
41. Zhang, E., Liu, F., Lai, Q., Jin, G., Li, Y.: Efficient multi-party private set intersection against malicious adversaries. In: CCSW (2019)
42. Zhao, Y., Chow, S.S.M.: Can you find the one for me? privacy-preserving matchmaking via threshold PSI. ePrint Arch. (2018)

A Full Security Model

In this paper, we use the simulation-based paradigm of secure computation [22] to define and prove the proposed protocols. Since both types of (static) active and passive adversaries are involved in our protocols, we will provide formal definitions for both types.

Two-party Computation. A two-party protocol Γ problem is captured by specifying a random process that maps pairs of inputs to pairs of outputs, one for each party. Such process is referred to as a functionality denoted by $f : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^*$, where $f := (f_1, f_2)$. For every input pair (x, y), the output pair is a random variable $(f_1(x, y), f_2(x, y))$, such that the party with input x wishes to obtain $f_1(x, y)$ while the party with input y wishes to receive $f_2(x, y)$. When f is deterministic, then $f_1 = f_2$. In the setting where f is asymmetric and only one party (say the first one) receives the result, f is defined as $f := (f_1(x, y), \perp)$.

Security in the Presence of Passive Adversaries. In the passive adversarial model, the party corrupted by such an adversary correctly follows the protocol specification. Nonetheless, the adversary obtains the internal state of the corrupted party, including the transcript of all the messages received, and tries to use this to learn information that should remain private. Loosely speaking, a protocol is secure if whatever can be computed by a party in the protocol can be computed using its input and output only. In the simulation-based model, it is required that a party's view in a protocol's execution can be simulated given only its input and output. This implies that the parties learn nothing from the protocol's execution. More formally, party i's view (during the execution of Γ) on input pair (x, y) is denoted by $\mathsf{View}_i^{\Gamma}(x, y)$ and equals $(w, r^i, m_1^i, ..., m_t^i)$, where $w \in \{x, y\}$ is the input of i^{th} party, r_i is the outcome of this party's internal random coin tosses, and m_j^i represents the j^{th} message this party receives. The output of the i^{th} party during the execution of Γ on (x, y) is denoted by $\mathsf{Output}_1^{\Gamma}(x, y)$ and can be generated from its own view of the execution. The joint output of both parties is denoted by $\mathsf{Output}^{\Gamma}(x, y) := (\mathsf{Output}_1^{\Gamma}(x, y), \mathsf{Output}_2^{\Gamma}(x, y))$.

Definition 11. *Let f be the deterministic functionality defined above. Protocol Γ security computes f in the presence of a static passive adversary if there exist polynomial-time algorithms $(\mathsf{Sim}_1, \mathsf{Sim}_2)$ such that:*

$$\{\mathsf{Sim}_1(x, f_1(x, y))\}_{x,y} \stackrel{c}{\equiv} \{\mathsf{View}_1^{\Gamma}(x, y)\}_{x,y}$$

$$\{\mathsf{Sim}_2(x, f_2(x, y))\}_{x,y} \stackrel{c}{\equiv} \{\mathsf{View}_2^{\Gamma}(x, y)\}_{x,y}$$

Security in the Presence of Active Adversaries. In this adversarial model, the corrupted party may arbitrarily deviate from the protocol specification, to learn the private inputs of the other parties or to influence the outcome of the computation. In this case, the adversary may not use the input provided. Therefore, beyond the possibility that a corrupted party may learn more than it should, correctness is also required. This means that a corrupted party must not be able to cause the output to be incorrectly distributed. Moreover, we require independence of inputs meaning that a corrupted party cannot make its input depend on the other party's input. To capture the threats, the security of a protocol is analyzed by comparing what an adversary can do in the real protocol to what it can do in an ideal scenario that is secure by definition. This

is formalized by considering an ideal computation involving an incorruptible Trusted Third Party (TTP) to whom the parties send their inputs and receive the output of the ideal functionality. Below, we describe the executions in the ideal and real models.

First, we describe the execution in the ideal model. Let P_1 and P_2 be the parties participating in the protocol, $i \in \{0,1\}$ be the index of the corrupted party, and $\mathcal{A}$ be a non-uniform probabilistic polynomial-time adversary. Also, let z be an auxiliary input given to $\mathcal{A}$ while x and y be the input of party P_1 and P_2 respectively. The honest party, P_j, sends its received input to TTP. The corrupted party P_i may either abort (by replacing the input with a special abort message Λ_i), send its received input or send some other input of the same length to TTP. This decision is made by the adversary and may depend on the input value of P_i and z. If TTP receives Λ_i, then it sends Λ_i to the honest party and the ideal execution terminates. Upon obtaining an input pair (x, y), TTP computes $f_1(x, y)$ and $f_2(x, y)$. It first sends $f_i(x, y)$ to P_i which replies with "continue" or Λ_i. In the former case, TTP sends $f_j(x, y)$ to P_j and in the latter it sends Λ_i to P_j. The honest party always outputs the message that it obtained from TTP. A malicious party may output an arbitrary function of its initial inputs and the message it has obtained from TTP. The ideal execution of f on inputs (x, y) and z is denoted by $\mathsf{Ideal}^{f}_{\mathcal{A}(z),i}(x, y)$ and is defined as the output pair of the honest party and $\mathcal{A}$ from the above ideal execution. In the real model, the real two-party protocol Γ is executed without the involvement of TTP. In this setting, $\mathcal{A}$ sends all messages on behalf of the corrupted party and may follow an arbitrary strategy. The honest party follows the instructions of Γ. The real execution of Γ is denoted by $\mathsf{Real}^{\Gamma}_{\mathcal{A}(z),i}(x, y)$, it is defined as the joint output of the parties engaging in the real execution of Γ (on the inputs), in the presence of $\mathcal{A}$.

Next, we define security. At a high level, the definition states that a secure protocol in the real model emulates the ideal model. This is formulated by stating that adversaries in the ideal model can simulate executions of the protocol in the real model.

Definition 12. *Let f be the two-party functionality defined above and Γ be a two-party protocol that computes f. Protocol Γ securely computes f with abort in the presence of static active adversaries if for every non-uniform probabilistic polynomial time adversary $\mathcal{A}$ for the real model, there exists a non-uniform probabilistic polynomial-time adversary (or simulator)* Sim *for the ideal model, such that for every $i \in \{0,1\}$, it holds that:*

$$\{\mathsf{Ideal}^{f}_{\mathsf{Sim}(z),i}(x, y)\}_{x,y,z} \stackrel{c}{\equiv} \{\mathsf{Real}^{\Gamma}_{\mathcal{A}(z),i}(x, y)\}_{x,y,z}$$

B Hash Tables

We set the table's parameters appropriately to ensure the number of elements in each bin does not exceed a predefined capacity. Given the maximum number of elements c and the bin's maximum size d, we can determine the number of bins by analysing hash tables under the balls into bins model [6].

Theorem 8. ***(Upper Tail in Chernoff Bounds)*** *Let X_i be a random variable defined as $X_i = \sum_{i=1}^{c} Y_i$, where $Pr[Y_i = 1] = p_i$, $Pr[Y_i = 0] = 1 - p_i$, and all Y_i are independent. Let the expectation be $\mu = \mathrm{E}[X_i] = \sum_{i=1}^{h} p_i$, then $Pr[X_i > d = (1+\sigma)\cdot\mu] < \left(\frac{e^{\sigma}}{(1+\sigma)^{(1+\sigma)}}\right)^{\mu}, \forall\sigma > 0$*

In this model, the expectation is $\mu = \frac{c}{h}$, where c is the number of balls and h is the number of bins. The above inequality provides the probability that bin i gets more than $(1+\sigma)\cdot\mu$ balls. Since there are h bins, the probability that at least one of them is overloaded is bounded by the union bound:

$$Pr[\exists i, X_i > d] \leq \sum_{i=1}^{h} Pr[X_i > d] = h \cdot \left(\frac{e^{\sigma}}{(1+\sigma)^{(1+\sigma)}}\right)^{\frac{c}{h}} \quad (1)$$

Thus, for a hash table of length $h = O(c)$, there is always an *almost constant* expected number of elements, d, mapped to the same bin with a high probability [36], e.g., $1 - 2^{-40}$.

C Merkle Tree

A Merkle tree is a data structure that supports a compact commitment of a set of values/blocks. As a result, it includes two parties, prover $\mathcal{P}$ and verifier $\mathcal{V}$. The Merkle tree scheme includes three algorithms (`MT.genTree`, `MT.prove`, `MT.verify`), defined as follows:

- The algorithm that constructs a Merkle tree, `MT.genTree`, is run by $\mathcal{V}$. It takes blocks, $u := u_1, ..., u_n$, as input. Then, it groups the blocks in pairs. Next, a collision-resistant hash function, `H`(.), is used to hash each pair. After that, the hash values are grouped in pairs and each pair is further hashed, and this process is repeated until only a single hash value, called "root", remains. This yields a tree with the leaves corresponding to the input blocks and the root corresponding to the last remaining hash value. $\mathcal{V}$ sends the root to $\mathcal{P}$.
- The proving algorithm, `MT.prove`, is run by $\mathcal{P}$. It takes a block index, i, and a tree as inputs. It outputs a vector proof, of $\log_2(n)$ elements. The proof asserts the membership of i-th block in the tree, and consists of all the sibling nodes on a path from the i-th block to the root of the Merkle tree (including i-th block). The proof is given to $\mathcal{V}$.
- The verification algorithm, `MT.verify`, is run by $\mathcal{V}$. It takes as an input i-th block, a proof, and the tree's root. It checks if the i-th block corresponds to the root. If the verification passes, it outputs 1; otherwise, it outputs 0.

The Merkle tree-based scheme has two properties: *correctness* and *security*. Informally, the correctness requires that if both parties run the algorithms correctly, then a proof is always accepted by $\mathcal{V}$. The security requires that a computationally bounded malicious $\mathcal{P}$ cannot convince $\mathcal{V}$ into accepting an incorrect

proof, e.g., proof for a non-member block. The security relies on the assumption that it is computationally infeasible to find the hash function's collision. Usually, for the sake of simplicity, it is assumed that the number of blocks, n, is a power of 2. The height of the tree, constructed on m blocks, is $\log_2(n)$.

D Enhanced OLE's Ideal Functionality and Protocol

The PSIs proposed in [21] use an enhanced version of the $\mathtt{OLE}$. The enhanced $\mathtt{OLE}$ ensures that the receiver cannot learn anything about the sender's inputs, in the case where it sets its input to 0, i.e., $c = 0$. The enhanced $\mathtt{OLE}$'s protocol (denoted by $\mathtt{OLE}^+$) is presented in Figure 4.

1. Receiver (input $c \in \mathbb{F}$): Pick a random value, $r \xleftarrow{\$} \mathbb{F}$, and send $(\mathtt{inputS}, (c^{-1}, r))$ to the first $\mathcal{F}_{\mathtt{OLE}}$.
2. Sender (input $a, b \in \mathbb{F}$): Pick a random value, $u \xleftarrow{\$} \mathbb{F}$, and send $(\mathtt{inputR}, u)$ to the first $\mathcal{F}_{\mathtt{OLE}}$, to learn $t = c^{-1} \cdot u + r$. Send $(\mathtt{inputS}, (t + a, b - u))$ to the second $\mathcal{F}_{\mathtt{OLE}}$.
3. Receiver: Send $(\mathtt{inputR}, c)$ to the second $\mathcal{F}_{\mathtt{OLE}}$ and obtain $k = (t + a) \cdot c + (b - u) = a \cdot c + b + r \cdot c$. Output $s = k - r \cdot c$.

Fig. 4: Enhanced Oblivious Linear function Evaluation ($\mathtt{OLE}^+$) [21].

E Proof of $\mathtt{VOPR}$

This section presents the proof of $\mathtt{VOPR}$, i.e., Theorem 1.

Proof. Before proving Theorem 1, we present Lemma 1 and Theorem 9 that will be used in the proof of Theorem 1. Informally, Lemma 1 states that the evaluation of a random polynomial at a fixed value results in a uniformly random value.

Lemma 1. *Let x_i be an element of a finite field $\mathbb{F}_p$, picked uniformly at random and $\boldsymbol{\mu}(x)$ be a random polynomial of constant degree d and defined over $\mathbb{F}_p[X]$. Then, the evaluation of $\boldsymbol{\mu}(x)$ at x_i is distributed uniformly at random over the non-zero elements of the field, i.e., $Pr[\boldsymbol{\mu}(x_i) = y] = \frac{1}{p-1}$, where y is arbitrary elements of $\mathbb{F}_p^*$.*

Proof. Let $\boldsymbol{\mu}(x) = a_0 + \sum_{j=1}^{d} a_j x^j$, where the coefficients are distributed uniformly at random over the field. We know that if x_i is a root of $\boldsymbol{\mu}(x)$, then because the polynomial can have at most d roots, we have $Pr[\boldsymbol{\mu}(x_i) = 0] = \frac{d}{p}$. Next, we focus on the case where x_i is not a root of the polynomial (thus $y \neq 0$). For any choice of $x_i, a_1, ..., a_d$, there exists exactly one value of a_0 that makes $\boldsymbol{\mu}(x_i) = y$,

i.e., $\boldsymbol{\mu}(x_i) = y$ iff $a_0 = y - \sum_{j=1}^{d} a_j x_i^j$. As a_0 is picked uniformly at random, the probability that it equals a certain value that makes $\boldsymbol{\mu}(x_i) = y$ is $\frac{1}{p-1}$. Thus, $Pr[\boldsymbol{\mu}(x_i) = y] = \frac{1}{p-1}$, $\forall y \in \mathbb{F}_p^*$. □

Informally, Theorem 9 states that the product of two arbitrary polynomials (in coefficient form) is a polynomial whose roots are the union of the two original polynomials. Below, we formally state it. The theorem has been taken from [3].

Theorem 9. *Let* $\mathbf{p}$ *and* $\mathbf{q}$ *be two arbitrary non-constant polynomials of degree* d *and* d' *respectively, such that* $\mathbf{p}, \mathbf{q} \in \mathbb{F}_p[X]$ *and they are in coefficient form. Then, the product of the two polynomials is a polynomial whose roots include precisely the two polynomials' roots.*

We refer readers to Appendix N for the proof of Theorem 9. Next, we prove the main theorem, i.e., Theorem 1, by considering the case where each party is corrupt, in turn.

Case 1: Corrupt sender. In the real execution, the sender's view is defined as follows:

$$\mathsf{View}_S^{\mathtt{VOPR}}\Big((\boldsymbol{\psi}, \boldsymbol{\alpha}), \boldsymbol{\beta}\Big) = \{\boldsymbol{\psi}, \boldsymbol{\alpha}, r_S, \boldsymbol{\beta}(z), \boldsymbol{\theta}(z), \mathsf{View}_S^{\mathtt{OLE}^+}, \perp\}$$

where r_S is the outcome of internal random coins of the sender and $\mathsf{View}_S^{\mathtt{OLE}^+}$ refers to the sender's real-model view during the execution of $\mathtt{OLE}^+$. The simulator $\mathsf{Sim}_S^{\mathtt{VOPR}}$, which receives $\boldsymbol{\psi}$ and $\boldsymbol{\alpha}$, works as follows.

1. generates an empty view. It appends to the view polynomials ($\boldsymbol{\psi}$, $\boldsymbol{\alpha}$) and coins r'_S chosen uniformly at random.
2. computes polynomial $\boldsymbol{\beta} = \boldsymbol{\beta}_1 \cdot \boldsymbol{\beta}_2$, where $\boldsymbol{\beta}_1$ is a random polynomial of degree 1 and $\boldsymbol{\beta}_2$ is an arbitrary polynomial of degree $e' - 1$. Next, it constructs polynomial $\boldsymbol{\theta}$ as follows: $\boldsymbol{\theta} = \boldsymbol{\psi} \cdot \boldsymbol{\beta} + \boldsymbol{\alpha}$.
3. picks value $z \xleftarrow{\$} \mathbb{F}_p$. Then, it evaluates polynomials $\boldsymbol{\beta}$ and $\boldsymbol{\theta}$ at point z. This results in values $\boldsymbol{\beta}_z$ and $\boldsymbol{\theta}_z$ respectively. It appends these two values to the view.
4. extracts the sender-side simulation of $\mathtt{OLE}^+$ from $\mathtt{OLE}^+$'s simulator. Let $\mathsf{Sim}_S^{\mathtt{OLE}^+}$ be this simulation. Note, the latter simulation is guaranteed to exist, as $\mathtt{OLE}^+$ has been proven secure (in [21]). It appends $\mathsf{Sim}_S^{\mathtt{OLE}^+}$ and $\perp$ to its view.

Now, we are ready to show that the two views are computationally indistinguishable. The sender's inputs are identical in both models, so they have identical distributions. Since the real-model semi-honest adversary samples its randomness according to the protocol's description, the random coins in both models have identical distributions. Next, we explain why values $\boldsymbol{\beta}(z)$ in the real model and $\boldsymbol{\beta}_z$ in the ideal model are (computationally) indistinguishable. In the real model, $\boldsymbol{\beta}(z)$ is the evaluation of polynomial $\boldsymbol{\beta} = \boldsymbol{\beta}_1 \cdot \boldsymbol{\beta}_2$ at random point z, where $\boldsymbol{\beta}_1$ is a random polynomial. We know that $\boldsymbol{\beta}(z) = \boldsymbol{\beta}_1(z) \cdot \boldsymbol{\beta}_2(z)$, for any (non-zero) z. Moreover, by Lemma 1, we know that $\boldsymbol{\beta}_1(z)$ is a uniformly random value.

Therefore, $\boldsymbol{\beta}(z) = \boldsymbol{\beta}_1(z) \cdot \boldsymbol{\beta}_2(z)$ is a uniformly random value as well. In the ideal world, polynomial $\boldsymbol{\beta}$ has the same structure as $\boldsymbol{\beta}$ has (i.e., $\boldsymbol{\beta} = \boldsymbol{\beta}_1 \cdot \boldsymbol{\beta}_2$, where $\boldsymbol{\beta}_1$ is a random polynomial). That means $\boldsymbol{\beta}_z$ is a uniformly random value too. Thus, $\boldsymbol{\beta}(z)$ and $\boldsymbol{\beta}_z$ are computationally indistinguishable. Next, we turn our attention to values $\boldsymbol{\theta}(z)$ in the real model and $\boldsymbol{\theta}_z$ in the ideal model. We know that $\boldsymbol{\theta}(z)$ is a function of $\boldsymbol{\beta}_1(z)$, as polynomial $\boldsymbol{\theta}$ has been defined as $\boldsymbol{\theta} = \boldsymbol{\psi} \cdot (\boldsymbol{\beta}_1 \cdot \boldsymbol{\beta}_2) + \boldsymbol{\alpha}$. Similarly, $\boldsymbol{\theta}_z$ is a function of $\boldsymbol{\beta}_z$. As we have already discussed, $\boldsymbol{\beta}(z)$ and $\boldsymbol{\beta}_z$ are computationally indistinguishable, so are their functions $\boldsymbol{\theta}(z)$ and $\boldsymbol{\theta}_z$. Moreover, as $\mathtt{OLE}^+$ has been proven secure, $\mathsf{View}_S^{\mathtt{OLE}^+}$ and $\mathsf{Sim}_S^{\mathtt{OLE}^+}$ are computationally indistinguishable. It is also clear that $\perp$ is identical in both models. We conclude that the two views are computationally indistinguishable.

Case 2: Corrupt receiver. Let $\mathsf{Sim}_R^{\mathtt{VOPR}}$ be the simulator, in this case, which uses a subroutine adversary, $\mathcal{A}_R$. $\mathsf{Sim}_R^{\mathtt{VOPR}}$ works as follows.

1. simulates $\mathtt{OLE}^+$ and receives $\mathcal{A}_R$'s input coefficients b_j for all j, $0 \leq j \leq e'$, as we are in $f_{\mathtt{OLE}^+}$-hybrid model.
2. reconstructs polynomial $\boldsymbol{\beta}$, given the above coefficients.
3. simulates the honest sender's inputs as follows. It picks two random polynomials: $\boldsymbol{\psi} = \sum_{i=0}^{e} g_i \cdot x^i$ and $\boldsymbol{\alpha} = \sum_{j=0}^{e+e'} a_j \cdot x^j$, such that $g_i \xleftarrow{\$} \mathbb{F}_p$ and every a_j has the form: $a_j = \sum_{\substack{t,k=0 \\ t=e}}^{k=e'} a_{t,k}$, where $t + k = j$ and $a_{t,k} \xleftarrow{\$} \mathbb{F}_p$.
4. sends to $\mathtt{OLE}^+$'s functionality values g_i and $a_{i,j}$ and receives $c_{i,j}$ from this functionality (for all i, j).
5. sends all $c_{i,j}$ to TTP and receives polynomial $\boldsymbol{\theta}$.
6. picks a random value z from $\mathbb{F}_p$. Then, it computes $\psi_z = \boldsymbol{\psi}(z)$ and $\alpha_z = \boldsymbol{\alpha}(z)$.
7. sends z and all $c_{i,j}$ to $\mathcal{A}_R$ which sends back θ_z and β_z to the simulator.
8. sends ψ_z and α_z to $\mathcal{A}_R$.
9. checks if the following relation hold:
$$\beta_{\bar{z}} = \boldsymbol{\beta}(z) \quad \wedge \quad \theta_z = \boldsymbol{\theta}(z) \quad \wedge \quad \boldsymbol{\theta}(z) = \psi_z \cdot \beta_z + \alpha_z \qquad (2)$$
If Relation 2 does not hold, it aborts (i.e., sends abort signal Λ to the sender) and still proceeds to the next step.
10. outputs whatever $\mathcal{A}_R$ outputs.

We first focus on the adversary's output. Both values of z in the real and ideal models have been picked uniformly at random from $\mathbb{F}_p$; therefore, they have identical distributions. In the real model, values ψ_z and α_z are the result of the evaluations of two random polynomials at (random) point z. In the ideal model, values ψ_z and α_z are also the result of the evaluations of two random polynomials (i.e., $\boldsymbol{\psi}$ and $\boldsymbol{\alpha}$) at point z. By Lemma 1, we know that the evaluation of a random polynomial at an arbitrary value yields a uniformly random value in $\mathbb{F}_p$. Therefore, the distribution of pair (ψ_z, α_z) in the real model is identical to that of pair (ψ_z, α_z) in the ideal model. Moreover, the final result (i.e., values $c_{i,j}$) in the real model has the same distribution as the final result (i.e., values

$c_{i,j}$) in the ideal model, as they are the outputs of the ideal calls to $f_{\texttt{OLE}^+}$, as we are in the $f_{\texttt{OLE}^+}$-hybrid model.

Next, we turn our attention to the sender's output. We will show that the output distributions of the honest sender in the ideal and real models are statistically close. Our focus will be on the probability that it aborts in each model, as it does not receive any other output. In the ideal model, $\mathsf{Sim}_R^{\texttt{VOPR}}$ is given the honestly generated result polynomial $\boldsymbol{\theta}$ (computed by TTP) and the adversary's input polynomial $\boldsymbol{\beta}$. $\mathsf{Sim}_R^{\texttt{VOPR}}$ aborts with a probability of 1 if Relation 2 does not hold. However, in the real model, the honest sender (in addition to its inputs) is given only β_z and θ_z and is not given polynomials $\boldsymbol{\beta}$ and $\boldsymbol{\theta}$; it wants to check if the following equation holds, $\theta_z = \psi_z \cdot \beta_z + \alpha_z$. Note, polynomial $\boldsymbol{\theta} = \boldsymbol{\psi} \cdot \boldsymbol{\beta} + \boldsymbol{\alpha}$ (resulted from $c_{i,j}$) is well-structured, as it satisfies the following three conditions, regardless of the adversary's input $\boldsymbol{\beta}$ to $\texttt{OLE}^+$, (i) $deg(\boldsymbol{\theta}) = Max\Big(deg(\boldsymbol{\beta}) + deg(\boldsymbol{\psi}), deg(\boldsymbol{\alpha})\Big)$, as $\mathbb{F}_p[X]$ is an integral domain and $(\boldsymbol{\psi}, \boldsymbol{\alpha})$ are random polynomials, (ii) the roots of the product polynomial $\boldsymbol{\nu} = \boldsymbol{\psi} \cdot \boldsymbol{\beta}$ contains exactly both polynomials' roots, by Theorem 9, and (iii) the roots of $\boldsymbol{\nu} + \boldsymbol{\alpha}$ is the intersection of the roots of $\boldsymbol{\nu}$ and $\boldsymbol{\alpha}$, as shown in [30]. Furthermore, polynomial $\boldsymbol{\theta}$ reveals no information (about $\boldsymbol{\psi}$ and $\boldsymbol{\alpha}$ except their degrees) to the adversary and the pair (ψ_z, α_z) is given to the adversary after it sends the pair (θ_z, β_z) to the sender. There are exactly four cases where pair (θ_z, β_z) can be constructed by the real-model adversary. Below, we state each case and the probability that the adversary is detected in that case during the verification, i.e., $\theta_z \stackrel{?}{=} \psi_z \cdot \beta_z + \alpha_z$.

1. $\theta_z = \boldsymbol{\theta}(z) \wedge \beta_z = \boldsymbol{\beta}(z)$. This is a trivial non-interesting case, as the adversary has behaved honestly, so it can always pass the verification.
2. $\theta_z \neq \boldsymbol{\theta}(z) \wedge \beta_z = \boldsymbol{\beta}(z)$. In this case, the adversary is detected with a probability of 1.
3. $\theta_z = \boldsymbol{\theta}(z) \wedge \beta_z \neq \boldsymbol{\beta}(z)$. In this case, the adversary is also detected with a probability of 1.
4. $\theta_z \neq \boldsymbol{\theta}(z) \wedge \beta_z \neq \boldsymbol{\beta}(z)$. In this case, the adversary is detected with an overwhelming probability, i.e., $1 - \frac{1}{2^{2\lambda}}$.

As we illustrated above, in the real model, the lowest probability that the honest sender would abort in case of adversarial behaviour is $1 - \frac{1}{2^{2\lambda}}$. Thus, the honest sender's output distributions in the ideal and real models are statistically close, i.e., 1 vs $1 - \frac{1}{2^{2\lambda}}$.

We conclude that the distribution of the joint outputs of the honest sender and adversary in the real and ideal models are computationally indistinguishable. □

F Proof of ZSPA

This section presents the proof of ZSPA, i.e., Theorem 2.

Proof. For the sake of simplicity, we will assume the sender, which generates the result, sends the result directly to the rest of the parties, i.e., receivers, instead of sending it to a smart contract. We first consider the case in which the sender is corrupt.

Case 1: Corrupt sender. Let $\mathsf{Sim}_S^{\mathtt{ZSPA}}$ be the simulator using a subroutine adversary, $\mathcal{A}_S$. $\mathsf{Sim}_S^{\mathtt{ZSPA}}$ works as follows.

1. simulates $\mathtt{CT}$ and receives the output value k from $f_{\mathtt{CT}}$, as we are in $f_{\mathtt{CT}}$-hybrid model.
2. sends k to TTP and receives back from it m pairs, where each pair is of the form (g, q).
3. sends k to $\mathcal{A}_S$ and receives back from it m pairs where each pair is of the form (g', q').
4. checks whether the following equations hold (for each pair): $g = g' \ \wedge \ q = q'$. If the two equations do not hold, then it aborts (i.e., sends abort signal Λ to the receiver) and proceeds to the next step.
5. outputs whatever $\mathcal{A}_S$ outputs.

We first focus on the adversary's output. In the real model, the only messages that the adversary receives are those messages it receives as the result of the ideal call to $f_{\mathtt{CT}}$. These messages have identical distribution to the distribution of the messages in the ideal model, as the $\mathtt{CT}$ is secure. Now, we move on to the receiver's output. We will show that the output distributions of the honest receiver in the ideal and real models are computationally indistinguishable. In the real model, each element of pair (g, p) is the output of a deterministic function on the output of $f_{\mathtt{CT}}$. We know the output of $f_{\mathtt{CT}}$ in the real and ideal models have an identical distribution, and so do the evaluations of deterministic functions (i.e., Merkle tree, $\mathtt{H}$, and $\mathtt{PRF}$) on them, as long as these three functions' correctness holds. Therefore, each pair (g, q) in the real model has an identical distribution to pair (g, q) in the ideal model. For the same reasons, the honest receiver in the real model aborts with the same probability as $\mathsf{Sim}_S^{\mathtt{ZSPA}}$ does in the ideal model. We conclude that the distributions of the joint outputs of the adversary and honest receiver in the real and ideal models are (computationally) indistinguishable.

Case 2: Corrupt receiver. Let $\mathsf{Sim}_R^{\mathtt{ZSPA}}$ be the simulator that uses subroutine adversary $\mathcal{A}_R$. $\mathsf{Sim}_R^{\mathtt{ZSPA}}$ works as follows.

1. simulates $\mathtt{CT}$ and receives the output value k from $f_{\mathtt{CT}}$.
2. sends k to TTP and receives back m pairs of the form (g, q) from TTP.
3. sends (k, g, q) to $\mathcal{A}_R$ and outputs whatever $\mathcal{A}_R$ outputs.

In the real model, the adversary receives two sets of messages, the first set includes the transcripts (including k) it receives when it makes an ideal call to $f_{\mathtt{CT}}$ and the second set includes pair (g, q). As we already discussed above (because we are in the $f_{\mathtt{CT}}$-hybrid model) the distributions of the messages it receives from $f_{\mathtt{CT}}$ in the real and ideal models are identical. Moreover, the distribution

of f_{CT}'s output (i.e., $\bar{k}$ and k) in both models is identical; therefore, the honest sender's output distribution in both models is identical. As we already discussed, the evaluations of deterministic functions (i.e., Merkle tree, H, and PRF) on f_{CT}'s outputs have an identical distribution. Therefore, each pair (g, q) in the real model has an identical distribution to the pair (g, q) in the ideal model. Hence, the distribution of the joint outputs of the adversary and honest receiver in the real and ideal models is indistinguishable. □

In addition to the security guarantee (i.e., computation's correctness against malicious sender or receiver) stated by Theorem 2, ZSPA offers (a) privacy against the public, and (b) non-refutability. Informally, privacy here means that given the state of the contract (i.e., g and q), an external party cannot learn any information about any of the pseudorandom values, z_j; while non-refutability means that if a party sends "approved" then in future cannot deny the knowledge of the values whose representation is stored in the contract.

Theorem 10. *If* H *is preimage resistance,* PRF *is secure, the signature scheme used in the smart contract is secure (i.e., existentially unforgeable under chosen message attacks), and the blockchain is secure (i.e., offers liveness property and the hash power of the adversary is lower than those of honest miners) then* ZSPA *offers (i) privacy against the public and (ii) non-refutability.*

Proof. First, we focus on privacy. Since key k, for PRF, has been picked uniformly at random and H is preimage resistance, the probability that given g the adversary can find k is negligible in the security parameter, i.e., $\epsilon(\lambda)$. Furthermore, because PRF is secure (i.e., its outputs are indistinguishable from random values) and H is preimage resistance, given the Merkle tree's root g, the probability that the adversary can find a leaf node, which is the output of PRF, is $\epsilon(\lambda)$ too. □

G Proof of ZSPA-A

This section provides the proof of ZSPA-A, i.e., Theorem 3.

Proof. First, we consider the case where a sender, who (may collude with $m-2$ senders and) generates pairs (g, q), is corrupt.

Case 1: Corrupt sender. Let $\mathsf{Sim}_S^{\text{ZSPA-A}}$ be the simulator using a subroutine adversary, $\mathcal{A}_S$. Below, we explain how $\mathsf{Sim}_S^{\text{ZSPA-A}}$ works.

1. simulates CT and receives the output value k from f_{CT}.
2. sends k to TTP and receives back from it m pairs, where each pair is of the form (g, q).
3. sends k to $\mathcal{A}_S$ and receives back from it m pairs where each pair is of the form (g', q').
4. constructs an empty vector L. $\mathsf{Sim}_S^{\text{ZSPA-A}}$ checks whether the following equations hold for each j-th pair: $g = g' \ \wedge \ q = q'$. If these two equations do not hold, it sends an abort message Λ to other receiver clients, appends the index of the pair (i.e., j) to L, and proceeds to the next step for the valid pairs. In the case where there are no valid pairs, it moves on to step 9.

5. picks a random polynomial $\boldsymbol{\zeta}$ of degree 1. Moreover, for every $j \notin L$, $\mathsf{Sim}_S^{\texttt{ZSPA-A}}$ picks a random polynomial $\boldsymbol{\xi}^{(j)}$ of degree $b'-1$, where $1 \leq j \leq m$.
6. computes m pseudorandom values for every i, j', where $0 \leq i \leq b'$ and $j' \notin L$ as follows. $\forall j', 1 \leq j' \leq m-1 : z_{i,j} = \texttt{PRF}(k, i||j')$ and $z_{i,m} = -\sum_{j'=1}^{m-1} z_{i,j}$.
7. generates polynomial $\boldsymbol{\mu}^{(j)}$, for every $j \notin L$, as follows: $\boldsymbol{\mu}^{(j)} = \boldsymbol{\zeta} \cdot \boldsymbol{\xi}^{(j)} - \boldsymbol{\tau}^{(j)}$, where $\boldsymbol{\tau}^{(j)} = \sum_{i=0}^{b'} z_{i,j} \cdot x^i$.
8. sends the above $\boldsymbol{\zeta}$, $\boldsymbol{\xi}^{(j)}$, and $\boldsymbol{\mu}^{(j)}$ to all parties (i.e., $\mathcal{A}_S$ and the receivers), for every $j \notin L$.
9. outputs whatever $\mathcal{A}_S$ outputs.

Now, we focus on the adversary's output. In the real model, the messages that the adversary receives include those messages it receives as the result of the ideal call to $f_{\texttt{CT}}$ and $(\boldsymbol{\zeta}, \boldsymbol{\xi}^{(j)}, \boldsymbol{\mu}^{(j)})$, where $j \notin L$ and $1 \leq j \leq m$. Those messages yielded from the ideal calls have identical distribution to the distribution of the messages in the ideal model, as $\texttt{CT}$ is secure. The distribution of each $\boldsymbol{\mu}^{(j)}$ depends on the distribution of its components; namely, $\boldsymbol{\zeta}, \boldsymbol{\xi}^{(j)}$, and $\boldsymbol{\tau}^j$. As we are in the $f_{\texttt{CT}}$-hybrid model, the distributions of $\boldsymbol{\tau}^{(j)}$ in the real model and $\boldsymbol{\tau}^{(j)}$ in the ideal model are identical, as they were derived from the output of $f_{\texttt{CT}}$. Furthermore, in the real model, each polynomial $\boldsymbol{\zeta}$ and $\boldsymbol{\xi}^{(j)}$ has been picked uniformly at random and they are independent of the clients' and the adversary's inputs. The same arguments hold for $(\boldsymbol{\zeta}, \boldsymbol{\xi}^{(j)}, \boldsymbol{\mu}^{(j)})$ in the ideal model. Therefore, $(\boldsymbol{\zeta}, \boldsymbol{\xi}^{(j)}, \boldsymbol{\mu}^{(j)})$ in the real model and $(\boldsymbol{\zeta}, \boldsymbol{\xi}^{(j)}, \boldsymbol{\mu}^{(j)})$ in the ideal model have identical distributions.

Next, we turn our attention to the receiver's output. We will show that the output distributions of an honest receiver and the auditor in the ideal and real models are computationally indistinguishable. In the real model, each element of the pair (g, p) is the output of a deterministic function on the output of $f_{\texttt{CT}}$. We know the outputs of $f_{\texttt{CT}}$ in the real and ideal models have an identical distribution, and so do the evaluations of deterministic functions (namely Merkle tree, $\texttt{H}$, and $\texttt{PRF}$) on them. Therefore, each pair (g, q) in the real model has an identical distribution to the pair (g, q) in the ideal model. For the same reasons, the honest receiver in the real model aborts with the same probability as $\mathsf{Sim}_S^{\texttt{ZSPA-A}}$ does in the ideal model. The same argument holds for the arbiter's output, as it performs the same checks that an honest receiver does. Thus, the distribution of the joint outputs of the adversary, honest receiver, and honest in the real and ideal models is computationally indistinguishable.

Case 2: Corrupt receiver. Let $\mathsf{Sim}_R^{\texttt{ZSPA-A}}$ be the simulator that uses subroutine adversary $\mathcal{A}_R$. Below, we explain how $\mathsf{Sim}_R^{\texttt{ZSPA-A}}$ works.

1. simulates $\texttt{ZSPA}$ and receives the m output pairs of the form (k, g, q) from $f^{\texttt{ZSPA}}$.
2. sends (k, g, q) to $\mathcal{A}_R$ and receives m keys, k'_j, where $1 \leq j \leq m$.
3. generates an empty vector L. Next, for every j, $\mathsf{Sim}_R^{\texttt{ZSPA-A}}$ computes q'_j as $\texttt{H}(k'_j) = q_j$. It generates g_j as follows.

(a) for every i (where $0 \leq i \leq b'$), generates m pseudorandom values as below.

$$\forall j, 1 \leq j' \leq m-1 : z_{i,j} = \mathtt{PRF}(k'_j, i||j'), \qquad z_{i,m} = -\sum_{j=1}^{m-1} z_{i,j}$$

(b) constructs a Merkel tree on top of all pseudorandom values as follows, $\mathtt{MT.genTree}(z_{0,1}, ..., z_{b',m}) \rightarrow g'_j$.

4. checks if the following equations hold for each j-th pair: $(k = k'_j) \ \wedge \ (g = g'_j) \ \wedge \ (q = q'_j)$.
5. If these equations do not hold for j-th value, it appends j to L and proceeds to the next step for the valid value. In the case where there is no valid value, it moves on to step 9.
6. picks a random polynomial $\boldsymbol{\zeta}$ of degree 1. Also, for every $j \notin L$, it picks a random polynomial $\boldsymbol{\xi}^{(j)}$ of degree $b'-1$, where $1 \leq j \leq m$.
7. generates polynomial $\boldsymbol{\mu}^{(j)}$, for every $j \notin L$, as follows: $\boldsymbol{\mu}^{(j)} = \boldsymbol{\zeta} \cdot \boldsymbol{\xi}^{(j)} - \boldsymbol{\tau}^{(j)}$, where $\boldsymbol{\tau}^{(j)} = \sum_{i=0}^{b'} z_{i,j} \cdot x^i$, and values $z_{i,j}$ were generated in step 3a.
8. sends the above $\boldsymbol{\zeta}$, $\boldsymbol{\xi}^{(j)}$, and $\boldsymbol{\mu}^{(j)}$ to $\mathcal{A}_R$, for every $j \notin L$ and $1 \leq j \leq m$.
9. outputs whatever $\mathcal{A}_R$ outputs.

In the real model, the adversary receives two sets of messages, the first set includes the transcripts (including k, g, q) it receives when it makes an ideal call to $f^{\mathtt{ZSPA}}$ and the second set includes pairs $(\boldsymbol{\zeta}, \boldsymbol{\xi}^{(j)}, \boldsymbol{\mu}^{(j)})$, for every $j \notin L$ and $1 \leq j \leq m$. Since we are in the $f^{\mathtt{ZSPA}}$-hybrid model and (based on our assumption) there is at least one honest party participated in ZSPA (i.e., there are at most $m-1$ malicious participants of ZSPA), the distribution of the messages it receives from $f^{\mathtt{ZSPA}}$ in the real and ideal models is identical. Furthermore, as we discussed in Case 1, $(\boldsymbol{\zeta}, \boldsymbol{\xi}^{(j)}, \boldsymbol{\mu}^{(j)})$ in the real model and $(\boldsymbol{\zeta}, \boldsymbol{\xi}^{(j)}, \boldsymbol{\mu}^{(j)})$ in the ideal model have identical distribution. The honest sender's output distribution in both models is identical, as the distribution of $f_{\mathtt{CT}}$'s output (i.e., k) in both models is identical.

Now we show that the probability that the auditor aborts in the ideal and real models are statistically close. In the ideal model, $\mathsf{Sim}_R^{\mathtt{ZSPA\text{-}A}}$ is given the ideal functionality's output that includes key k. Therefore, it can check whether the key that $\mathcal{A}_R$ sends to it equals k, i.e., $k \stackrel{?}{=} k'_j$. Thus, it aborts with the probability 1. However, in the real model, an honest auditor is not given the output of CT (say key k) and it can only check whether the key is consistent with the hash value q and the Merkle tree's root g stored on the blockchain. This means the adversary can distinguish the two models if in the real model it sends a key $\ddot{k}$, such that $\ddot{k} \neq k$ and still passes the checks. Specifically, it sends the invalid key $\ddot{k}$ that can generate valid pair (g, q), as follows: $\mathtt{H}(\ddot{k}) = q$ and $\mathtt{MT.genTree}(z'_{0,1}, ..., z'_{b',m}) \rightarrow g$, where each $z'_{i,j}$ is derived from $\ddot{k}$ using the same technique described in step 3 above. Nevertheless, this means that the adversary breaks the second preimage resistance property of H; however, H is the second-preimage resistance and the probability that the adversary succeeds in finding the second preimage is negligible in the security parameter, i.e., $\epsilon(\lambda)$ where λ is the hash function's security parameter. Therefore, in the real model, the auditor aborts if an invalid key is

provided with a probability $1-\epsilon(\lambda)$ which is statically close to the probability that $\mathsf{Sim}_R^{\text{ZSPA-A}}$ aborts in the same situation in the ideal model, i.e., $1-\epsilon(\lambda)$ vs 1. Hence, the distribution of the joint outputs of the adversary, honest sender, and honest auditor in the real and ideal models is indistinguishable. □

H Workflow of Justitia

Figure 5 outlines the interaction between parties in Justitia.

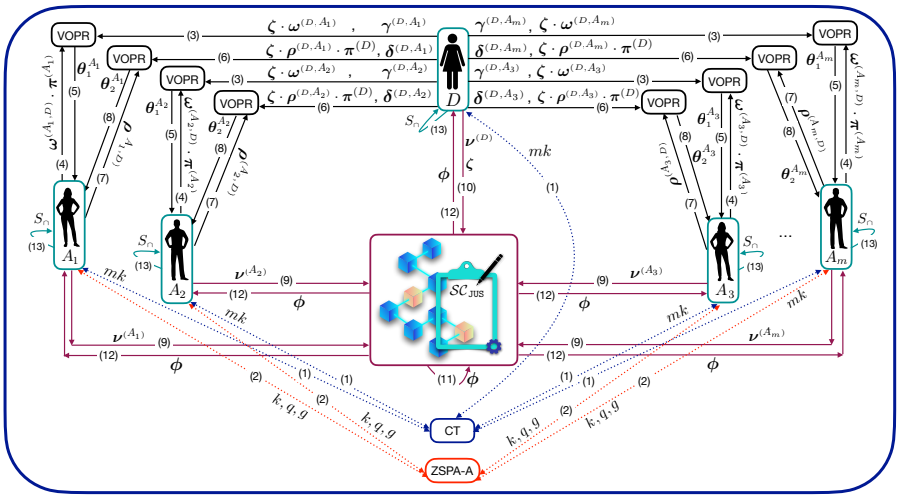

Fig. 5: Outline of the interactions between parties in Justitia

I Proof of JUS

In this section, we prove Theorem 6, i.e., the security of JUS.

Proof. We prove Theorem 6 by considering the case where each party is corrupt, at a time.

Case 1: Corrupt $m-1$ clients in $\{A_1,...,A_m\}$. Let P' be a set of at most $m-1$ corrupt clients, where $P' \subset \{A_1,...,A_m\}$. Let set $\hat{P}$ be defined as follows: $\hat{P} = \{A_1,...,A_m\} \setminus P'$. Also, let $\mathsf{Sim}_A^{\text{JUS}}$ be the simulator, which uses a subroutine adversary, $\mathcal{A}$. Below, we explain how $\mathsf{Sim}_A^{\text{JUS}}$ (which receives the input sets of honest dealer D and honest client(s) in $\hat{P}$) works.

1. constructs and deploys a smart contract. It sends the contract's address to $\mathcal{A}$.

2. simulates CT and receives the output value, mk, from its functionality, f_{CT}.
3. simulates ZSPA-A for each bin and receives the output value, (k, g, q), from its functionality, $f^{\text{ZSPA-A}}$.
4. deposits in the contract the total amount of $(\ddot{y} + \ddot{ch}) \cdot (m - |P'| + 1)$ coins on behalf of D and honest client(s) in $\hat{P}$. It sends to $\mathcal{A}$ the amount deposited in the contract.
5. checks if $\mathcal{A}$ has deposited $(\ddot{y} + \ddot{ch}) \cdot |P'|$ amount. If the check fails, it instructs the ledger to refund the coins that every party deposited and sends message $abort_1$ to TTP (and accordingly to all parties); it outputs whatever $\mathcal{A}$ outputs and then halts.
6. picks a random polynomial $\boldsymbol{\zeta}$ of degree 1, for each bin. Also, $\mathsf{Sim}_A^{\text{JUS}}$, for each client $C \in \{A_1, ..., A_m\}$ allocates to each bin two random polynomials: $(\boldsymbol{\omega}^{(D,C)}, \boldsymbol{\rho}^{(D,C)})$ of degree d and two random polynomials: $(\boldsymbol{\gamma}^{(D,C)}, \boldsymbol{\delta}^{(D,C)})$ of degree $3d + 1$. Moreover, $\mathsf{Sim}_A^{\text{JUS}}$ for every honest client $C' \in \hat{P}$, for each bin, picks two random polynomials: $(\boldsymbol{\omega}^{(C',D)}, \boldsymbol{\rho}^{(C',D)})$ of degree d.
7. simulates VOPR using inputs $\boldsymbol{\zeta} \cdot \boldsymbol{\omega}^{(D,C)}$ and $\boldsymbol{\gamma}^{(D,C)}$ for each bin. Accordingly, it receives the inputs of clients $C'' \in P'$, i.e., $\boldsymbol{\omega}^{(C'',D)} \cdot \boldsymbol{\pi}^{(C'')}$, from its functionality f^{VOPR}, for each bin.
8. extracts the roots of polynomial $\boldsymbol{\omega}^{(C'',D)} \cdot \boldsymbol{\pi}^{(C'')}$ for each bin and appends those roots that are in the sets universe to a new set $S^{(C'')}$.
9. simulates VOPR again using inputs $\boldsymbol{\zeta} \cdot \boldsymbol{\rho}^{(D,C)} \cdot \boldsymbol{\pi}^{(D)}$ and $\boldsymbol{\delta}^{(D,C)}$, for each bin.
10. sends to TTP the input sets of all parties; namely, (i) client D's input set: $S^{(D)}$, (ii) honest clients' input sets: $S^{(C')}$ for all C' in $\hat{P}$, and (iii) $\mathcal{A}$'s input sets: $S^{(C'')}$, for all C'' in P'. For each bin, it receives the intersection set, $S_\cap$, from TTP.
11. represents the intersection set for each bin as a polynomial, $\boldsymbol{\pi}$, as follows: $\boldsymbol{\pi} = \prod_{i=1}^{|S_\cap|} (x - s_i)$, where $s_i \in S_\cap$.
12. constructs polynomials $\boldsymbol{\theta}_1^{(C')} = \boldsymbol{\zeta} \cdot \boldsymbol{\omega}^{(D,C')} \cdot \boldsymbol{\omega}^{(C',D)} \cdot \boldsymbol{\pi} + \boldsymbol{\gamma}^{(D,C')}$, $\boldsymbol{\theta}_2^{(C')} = \boldsymbol{\zeta} \cdot \boldsymbol{\rho}^{(D,C')} \cdot \boldsymbol{\rho}^{(C',D)} \cdot \boldsymbol{\pi} + \boldsymbol{\delta}^{(D,C')}$, and $\boldsymbol{\nu}^{(C')} = \boldsymbol{\theta}_1^{(C')} + \boldsymbol{\theta}_2^{(C')} + \boldsymbol{\tau}^{(C')}$, for each bin and each honest client $C' \in \hat{P}$, where $\boldsymbol{\tau}^{(C)} = \sum_{i=0}^{3d+2} z_{i,c} \cdot x^i$ and each $z_{i,c}$ is derived from k.
13. sends to $\mathcal{A}$ polynomial $\boldsymbol{\nu}^{(C')}$ for each bin and each client C'.
14. receives $\boldsymbol{\nu}^{(C'')}$ from $\mathcal{A}$, for each bin and each client $C'' \in P'$. It ensures that the output for every C'' has been provided. Otherwise, it halts.
15. if there is any abort within steps 7–14, then it sends $abort_2$ to TTP and instructs the ledger to refund the coins that every party deposited. It outputs whatever $\mathcal{A}$ outputs and then halts.
16. constructs polynomial $\boldsymbol{\nu}^{(D)} = \boldsymbol{\zeta} \cdot \boldsymbol{\omega'}^{(D)} \cdot \boldsymbol{\pi} - \sum_{C=A_1}^{A_m} (\boldsymbol{\gamma}^{(D,C)} + \boldsymbol{\delta}^{(D,C)}) + \boldsymbol{\zeta} \cdot \boldsymbol{\gamma}'$ for each bin, where $\boldsymbol{\omega'}^{(D)}$ is a fresh random polynomial of degree d and $\boldsymbol{\gamma}'$ is a pseudorandom polynomial derived from mk.
17. sends to $\mathcal{A}$ polynomials $\boldsymbol{\nu}^{(D)}$ and $\boldsymbol{\zeta}$ for each bin.
18. given each $\boldsymbol{\nu}^{(C'')}$, computes polynomial $\boldsymbol{\phi}'$ as follows $\boldsymbol{\phi}' = \sum_{\forall C'' \in P'} \boldsymbol{\nu}^{(C'')} - \sum_{\forall C'' \in P'} (\boldsymbol{\gamma}^{(D,C'')} + \boldsymbol{\delta}^{(D,C'')})$, for every bin. Then, $\mathsf{Sim}_A^{\text{JUS}}$ checks whether $\boldsymbol{\zeta}$ divides

$\boldsymbol{\phi}'$, for every bin. If the check passes, it sets $Flag = True$. Otherwise, it sets $Flag = False$.

19. if $Flag = True$:
    (a) instructs the ledger to send back each party's deposit, i.e., $\ddot{y}+\ddot{ch}$ amount. It sends a message *deliver* to TTP.
    (b) outputs whatever $\mathcal{A}$ outputs and then halts.
20. if $Flag = False$:
    (a) receives $|P'|$ keys of the PRF from $\mathcal{A}$, i.e., $\vec{k}' = [k'_1, ..., k'_{|P'|}]$, for every bin.
    (b) checks whether the following equation holds: $k'_j = k$, for every $k'_j \in \vec{k}'$. Note that k is the output of $f^{\text{ZSPA-A}}$ generated in step 3. It constructs an empty list L' and appends to it the indices (e.g., j) of the keys that do not pass the above check.
    (c) simulates ZSPA-A and receives from $f^{\text{ZSPA-A}}$ the output that contains a vector of random polynomials, $\vec{\mu}'$, for each valid key.
    (d) sends to $\mathcal{A}$, the list L' and vector $\vec{\mu}'$, for every bin.
    (e) for each bin of client C whose index (or ID) is not in L' computes polynomial $\boldsymbol{\chi}^{(D,C)}$ as follows: $\boldsymbol{\chi}^{(D,C)} = \boldsymbol{\zeta} \cdot \boldsymbol{\eta}^{(D,C)} - (\boldsymbol{\gamma}^{(D,C)} + \boldsymbol{\delta}^{(D,C)})$, where $\boldsymbol{\eta}^{(D,C)}$ is a fresh random polynomial of degree $3d + 1$. Note, C includes both honest and corrupt clients, except those clients whose index is in L'. $\mathsf{Sim}_A^{\text{JUS}}$ sends every polynomial $\boldsymbol{\chi}^{(D,C)}$ to $\mathcal{A}$.
    (f) given each $\boldsymbol{\nu}^{(C'')}$ (by $\mathcal{A}$ in step 14), computes polynomial $\boldsymbol{\phi}'^{(C'')}$ as follows: $\boldsymbol{\phi}'^{(C'')} = \boldsymbol{\nu}^{(C'')} - \boldsymbol{\gamma}^{(D,C'')} - \boldsymbol{\delta}^{(D,C'')}$, for every bin. Then, $\mathsf{Sim}_A^{\text{JUS}}$ checks whether $\boldsymbol{\zeta}$ divides $\boldsymbol{\phi}'^{(C'')}$, for every bin. It appends the index of those clients that did not pass the above check to a new list, L''. Note that $L' \cap L'' = \perp$.
    (g) if L' or L'' is not empty, then it instructs the ledger: (i) to refund the coins of those parties whose index is not in L' and L'', (ii) to retrieve $\ddot{ch}$ amount from the adversary (i.e., one of the parties whose index is in one of the lists) and send the $\ddot{ch}$ amount to the auditor, and (iii) to compensate each honest party (whose index is not in the two lists) $\frac{m' \cdot (\ddot{y}+\ddot{ch}) - \ddot{ch}}{m-m'}$ amount, where $m' = |L'| + |L''|$. Then, it sends message $abort_3$ to TTP.
    (h) outputs whatever $\mathcal{A}$ outputs and halts.

Next, we show that the real and ideal models are computationally indistinguishable. We first focus on the adversary's output. In the real and ideal models, the adversary sees the transcripts of ideal calls to f_{CT} as well as this functionality outputs, i.e., mk. Due to the security of CT (as we are in the f_{CT}-hybrid world), the transcripts of f_{CT} in both models have identical distribution, so have the random output of f_{CT}, i.e., mk. The same holds for (the transcripts and) outputs (i.e., (k, g, q)) of $f^{\text{ZSPA-A}}$ that the adversary observes in the two models. Also, the deposit amount is identical in both models. Thus, in the case where $abort_1$ is disseminated at this point; the adversary's output distribution in both models is identical.

The adversary also observes (the transcripts and) outputs of ideal calls to $f^{\texttt{VOPR}}$ in both models, i.e., output $(\boldsymbol{\theta}_1^{(C'')} = \boldsymbol{\zeta} \cdot \boldsymbol{\omega}^{(D,C'')} \cdot \boldsymbol{\omega}^{(C'',D)} \cdot \boldsymbol{\pi}^{(C'')} + \boldsymbol{\gamma}^{(D,C'')}, \boldsymbol{\theta}_2^{(C'')} = \boldsymbol{\zeta} \cdot \boldsymbol{\rho}^{(D,C'')} \cdot \boldsymbol{\rho}^{(C'',D)} \cdot \boldsymbol{\pi}^{(D)} + \boldsymbol{\delta}^{(D,C'')})$ for each corrupted client C''. However, due to the security of VOPR, the $\mathcal{A}$'s view, regarding VOPR, in both models have identical distribution. In the real model, the adversary observes the polynomial $\boldsymbol{\nu}^{(C)}$ that each honest client C stores in the smart contract. Nevertheless, this is a blinded polynomial comprising of two uniformly random blinding polynomials (i.e., $\boldsymbol{\gamma}^{(D,C)}$ and $\boldsymbol{\delta}^{(D,C)}$) unknown to the adversary. In the ideal model, $\mathcal{A}$ is given polynomial $\boldsymbol{\nu}^{(C')}$ for each honest client C'. This polynomial has also been blinded via two uniformly random polynomials (i.e., $\boldsymbol{\gamma}^{(D,C')}$ and $\boldsymbol{\delta}^{(D,C')}$) unknown to $\mathcal{A}$. Thus, $\boldsymbol{\nu}^{(C)}$ in the real model and $\boldsymbol{\nu}^{(C)}$ in the ideal model have identical distributions. As a result, in the case where $abort_2$ is disseminated at this point; the adversary's output distribution in both models is identical.

Furthermore, in the real world, the adversary observes polynomials $\boldsymbol{\zeta}$ and $\boldsymbol{\nu}^{(D)}$ that D stores in the smart contract. Nevertheless, $\boldsymbol{\zeta}$ is a uniformly random polynomial, also polynomial $\boldsymbol{\nu}^{(D)}$ has been blinded; its blinding factors are the additive inverse of the sum of the random polynomials $\boldsymbol{\gamma}^{(D,C)}$ and $\boldsymbol{\delta}^{(D,C)}$ unknown to the adversary, for every client $C \in \{A_1, ..., A_m\}$ and D. In the ideal model, $\mathcal{A}$ is given $\boldsymbol{\zeta}$ and $\boldsymbol{\nu}^{(D)}$, where the former is a random polynomial and the latter is a blinded polynomial that has been blinded with the additive inverse of the sum of random polynomials $\boldsymbol{\gamma}^{(D,C)}$ and $\boldsymbol{\delta}^{(D,C)}$ unknown to it, for all client C. Therefore, $(\boldsymbol{\zeta}, \boldsymbol{\nu}^{(D)})$ in the real model and $(\boldsymbol{\zeta}, \boldsymbol{\nu}^{(D)})$ in the ideal model component-wise have identical distribution.

Also, the sum of less than $m+1$ blinded polynomials $\boldsymbol{\nu}^{(A_1)}, ..., \boldsymbol{\nu}^{(A_m)}, \boldsymbol{\nu}^{(D)}$ in the real model has identical distribution to the sum of less than $m+1$ blinded polynomials $\boldsymbol{\nu}^{(A_1)}, ..., \boldsymbol{\nu}^{(A_m)}, \boldsymbol{\nu}^{(D)}$ in the ideal model, as such a combination would still be blinded by a set of random blinding polynomials unknown to the adversary. Now we discuss why the two polynomials $\frac{\boldsymbol{\phi}}{\boldsymbol{\zeta}} - \boldsymbol{\gamma}'$ in the real model and $\frac{\boldsymbol{\phi}}{\boldsymbol{\zeta}} - \boldsymbol{\gamma}'$ in the ideal model are indistinguishable. Note that we divide and then subtract polynomials $\boldsymbol{\phi}$ because the adversary already knows (and must know) polynomials $(\boldsymbol{\zeta}, \boldsymbol{\gamma}')$. In the real model, polynomial $\frac{\boldsymbol{\phi}}{\boldsymbol{\zeta}} - \boldsymbol{\gamma}'$ has the following form:

$$\frac{\boldsymbol{\phi}}{\boldsymbol{\zeta}} - \boldsymbol{\gamma}' = \boldsymbol{\omega}'^{(D)} \cdot \boldsymbol{\pi}^{(D)} + \sum_{C=A_1}^{A_m} (\boldsymbol{\omega}^{(D,C)} \cdot \boldsymbol{\omega}^{(C,D)} \cdot \boldsymbol{\pi}^{(C)}) + \boldsymbol{\pi}^{(D)} \cdot \sum_{C=A_1}^{A_m} (\boldsymbol{\rho}^{(D,C)} \cdot \boldsymbol{\rho}^{(C,D)}) = \boldsymbol{\mu} \cdot gcd(\boldsymbol{\pi}^{(D)}, \boldsymbol{\pi}^{(A_1)}, ..., \boldsymbol{\pi}^{(A_m)}) \tag{3}$$

In Equation 3, every element of $[\boldsymbol{\omega}'^{(D)}, ..., \boldsymbol{\omega}^{(D,C)}, \boldsymbol{\rho}^{(D,C)}]$ is a uniformly random polynomial for every client $C \in \{A_1, ..., A_m\}$ (including corrupt ones) and client D; because it has been picked by (in this case honest) client D. Thus, as shown in Section 3.9, $\frac{\boldsymbol{\phi}}{\boldsymbol{\zeta}} - \boldsymbol{\gamma}'$ has the form $\boldsymbol{\mu} \cdot gcd(\boldsymbol{\pi}^{(D)}, \boldsymbol{\pi}^{(A_1)}, ..., \boldsymbol{\pi}^{(A_m)})$, where $\boldsymbol{\mu}$ is a uniformly random polynomial and $gcd(\boldsymbol{\pi}^{(D)}, \boldsymbol{\pi}^{(A_1)}, ..., \boldsymbol{\pi}^{(A_m)})$ represents the intersection of the input sets.

In the ideal model, $\mathcal{A}$ can construct polynomial $\boldsymbol{\phi}$ using its (well-formed) inputs $\boldsymbol{\nu}^{(C'')}$ and polynomials $\boldsymbol{\nu}^{(C')}$ that the simulator has sent to it, for all $C' \in \hat{P}$ and all $C'' \in P'$. Thus, in the ideal model, polynomial $\frac{\boldsymbol{\phi}}{\boldsymbol{\zeta}} - \boldsymbol{\gamma}'$ has the

following form:

$$\begin{aligned}\frac{\phi}{\zeta} - \gamma' &= \pi \cdot \Big(\sum_{\forall C' \in \hat{P}} (\omega^{(D,C')} \cdot \omega^{(C',D)}) + \sum_{\forall C' \in \hat{P}} (\rho^{(D,C')} \cdot \rho^{(C',D)}) \Big) + \\ &+ \Big(\sum_{\forall C'' \in P'} (\omega^{(D,C'')} \cdot \omega^{(C'',D)} \cdot \pi^{(C'')}) + \pi^{(D)} \cdot \sum_{\forall C'' \in P'} (\rho^{(D,C'')} \cdot \rho^{(C'',D)}) \Big) \\ &= \mu \cdot gcd(\pi, \pi^{(D)}, \pi^{(C'')}) \end{aligned} \tag{4}$$

In Equation 4, every element of the vector $[\omega^{(D,C')}, \omega^{(D,C'')}, \rho^{(D,C')}, \rho^{(D,C'')}]$ is a uniformly random polynomial for all $C' \in \hat{P}$ and all $C'' \in P'$, as they have been picked by $\mathsf{Sim}_A^{\text{JUS}}$. Therefore, $\frac{\phi}{\zeta} - \gamma'$ equals $\mu \cdot gcd(\pi, \pi^{(D)}, \pi^{(C'')})$, such that μ is a uniformly random polynomial and $gcd(\pi, \pi^{(D)}, \pi^{(C'')})$ represents the intersection of the input sets. We know that $gcd(\pi^{(D)}, \pi^{(A_1)}, ..., \pi^{(A_m)}) = gcd(\pi, \pi^{(D)}, \pi^{(C'')})$, as π includes the intersection of all clients' sets. Also, μ has identical distribution in the two models, because they are uniformly random polynomials. Thus, $\frac{\phi}{\zeta} - \gamma'$ in the real model and $\frac{\phi}{\zeta} - \gamma'$ in the ideal model are indistinguishable.

Now we focus on the case where $Flag = False$. In the real model, the adversary observes the output of `Audit`(.) which is a list of indices L and a vector of random polynomials $\vec{\mu}$ picked by an honest auditor. In the ideal model, $\mathcal{A}$ is given a list L' of indices and a vector of random polynomials $\vec{\mu}'$ picked by the simulator. Thus, the pair $(L, \vec{\mu})$ in the real model has identical distribution to the pair $(L', \vec{\mu}')$ in the ideal model. Moreover, in the real model, the adversary observes each polynomial $\chi^{(D,C)} = \zeta \cdot \eta^{(D,C)} - (\gamma^{(D,C)} + \delta^{(D,C)})$ that D stores in the contract, for each bin and each client C whose index is not in L. This is a blinded polynomial with blinding factor $\eta^{(D,C)}$ which itself is a uniformly random polynomial picked by D. In the ideal model, $\mathcal{A}$ is given a polynomial of the form $\chi^{(D,C)} = \zeta \cdot \eta^{(D,C)} - (\gamma^{(D,C)} + \delta^{(D,C)})$, for each bin and each client C whose index is not in L'. This is also a blinded polynomial whose blinding factor is $\eta^{(D,C)}$ which itself is a random polynomial picked by the simulator. Therefore, $\chi^{(D,C)}$ in the real model has identical distribution to $\chi^{(D,C)}$ in the ideal model. In the real model, the adversary observes polynomial $\iota^{(C)} = \zeta \cdot (\eta^{(D,C)} + \omega^{(D,C)} \cdot \omega^{(C,D)} \cdot \pi^{(C)} + \rho^{(D,C)} \cdot \rho^{(C,D)} \cdot \pi^{(D)} + \xi^{(C)})$ which is a blinded polynomial whose blinding factor is the sum of the above random polynomials, i.e., $\eta^{(D,C)} + \xi^{(C)}$. In the ideal model, $\mathcal{A}$ already has polynomials $\chi^{(D,C)}, \nu^{(C)}$, and $\mu^{(C)}$, where $\mu^{(C)} \in \vec{\mu}'$; this lets $\mathcal{A}$ compute $\iota^{(C)} = \chi^{(D,C)} + \nu^{(C)} + \mu^{(C)} = \zeta \cdot (\eta^{(D,C)} + \omega^{(D,C')} \cdot \omega^{(C',D)} \cdot \pi + \rho^{(D,C')} \cdot \rho^{(C',D)} \cdot \pi + \xi^{(C)})$, where $\xi^{(C)}$ is a random blinding polynomial used in $\mu^{(C)}$. Nevertheless, $\iota^{(C)}$ itself is a blinded polynomial whose blinding factor is the sum of random polynomials, i.e., $\eta^{(D,C)} + \xi^{(C)}$. Hence, the distribution of polynomial $\iota^{(C)}$ in the real model and $\iota^{(C)}$ in the ideal model are identical. Moreover, the integer $\ddot{y} + c\ddot{h} + \frac{m' \cdot (\ddot{y} + c\ddot{h}) - c\ddot{h}}{m - m'}$ has identical distribution in both models.

Next, we show that the honest party aborts with the same probability in the real and ideal models. Due to the security of `CT`, an honest party (during the execution of `CT`) aborts with the same probability in both models; in this case, the adversary learns nothing about the parties' input set and the sets' intersection

as the parties have not sent out any encoded input set yet. The same holds for the probability that honest parties abort during the execution of ZSPA-A. In this case, an aborting adversary also learns nothing about the parties' input set and the sets' intersection. Since all parties' deposit is public, an honest party can look up and detect if not all parties have deposited a sufficient amount with the same probability in both models. If parties halt because of insufficient deposit, no one learns about the parties' input set and the sets' intersection because the inputs (representation) have not been sent out at this point.

Due to the security of VOPR, honest parties abort with the same probability in both models. In the case of an abort during VOPR execution, the adversary would learn nothing (i) about its counter party' input set due to the security of VOPR, and (ii) about the rest of the honest parties' input sets and the intersection as the other parties' input sets are still blinded by random blinding factors known only to D. In the real model, D can check if all parties provided their encoded inputs, by reading from the smart contract. The simulator also can do the same check to ensure $\mathcal{A}$ has provided the encoded inputs of all corrupt parties. Therefore, in both models, an honest party with the same probability would detect the violation of such a requirement, i.e., providing all encoded inputs. Even in this case, if an adversary aborts (by not proving its encoded inputs), then it learns nothing about the honest parties' input sets and the intersection for the reason explained above.

In the real model, the smart contract sums every client C's polynomial $\boldsymbol{\nu}^{(C)}$ with each other and with D's polynomial $\boldsymbol{\nu}^{(D)}$, which removes the blinding factors that D initially inserted (during the execution of VOPR), and then checks whether the result is divisible by $\boldsymbol{\zeta}$. Due to (a) Theorem 5 (i.e., unforgeable polynomials' linear combination), (b) the fact that the smart contract is given the random polynomial $\boldsymbol{\zeta}$ in plaintext, (c) no party (except honest client D) knew anything about $\boldsymbol{\zeta}$ before they send their input to the contract, and (d) the security of the contract (i.e., the adversary cannot influence the correctness of the above verification performed by the contract), the contract can detect if a set of outputs of VOPR has been tampered with, with a probability at least $1-\epsilon(\lambda)$. In the ideal model, $\mathsf{Sim}_A^{\text{JUS}}$ also can remove the blinding factors and it knows the random polynomial $\boldsymbol{\zeta}$, unlike the adversary who does not know $\boldsymbol{\zeta}$ when it sends the outputs of VOPR to $\mathsf{Sim}_A^{\text{JUS}}$. So, $\mathsf{Sim}_A^{\text{JUS}}$ can detect when $\mathcal{A}$ modifies a set of the outputs of VOPR that were sent to $\mathsf{Sim}_A^{\text{JUS}}$ with a probability at least $1-\epsilon(\lambda)$, due to Theorem 5. Hence, the smart contract in the real model and the simulator in the ideal model would abort with a similar probability.

Moreover, due to the security of ZSPA-A, the probability that an invalid key $k_i \in \overrightarrow{k}$ is added to the list L in the real world is similar to the probability that $\mathsf{Sim}_A^{\text{JUS}}$ detects an invalid key $k'_i \in \overrightarrow{k'}$ in the ideal world. In the real model, when $Flag = False$, the smart contract can identify each ill-structured output of VOPR (i.e., $\boldsymbol{\nu}^{(C)}$) with a probability of at least $1-\epsilon(\lambda)$ by checking whether $\boldsymbol{\zeta}$ divides $\boldsymbol{\iota}^{(C)}$, due to (a) Theorem 4 (i.e., unforgeable polynomial), (b) the fact that the smart contract is given $\boldsymbol{\zeta}$ in plaintext, (c) no party (except honest client D) knew anything about $\boldsymbol{\zeta}$ before they send their input to the contract,

and (d) the security of the contract. In the ideal model, when $Flag = False$, given each $\boldsymbol{\nu}^{(C'')}$, $\mathsf{Sim}_A^{\text{JUS}}$ can remove its blinding factors from $\boldsymbol{\nu}^{(C'')}$ which results in $\boldsymbol{\phi}'^{(C'')}$ and then check if $\boldsymbol{\zeta}$ divides $\boldsymbol{\phi}'^{(C'')}$. The simulator can also detect an ill-structured $\boldsymbol{\nu}^{(C'')}$ with a probability of at least $1-\epsilon(\lambda)$, due to Theorem 4, the fact that the simulator is given $\boldsymbol{\zeta}$ in plaintext, and the adversary is not given any knowledge about $\boldsymbol{\zeta}$ before it sends to the simulator the outputs of VOPR. Hence, the smart contract in the real model and $\mathsf{Sim}_A^{\text{JUS}}$ in the ideal model would detect an ill-structured input of an adversary with the same probability.

Now, we analyse the output of the predicates $(Q^{\text{Init}}, Q^{\text{Del}}, Q^{\text{UF-A}}, Q^{\text{F-A}})$ in the real and ideal models. In the real model, all clients proceed to prepare their input set only if the predefined amount of coins has been deposited by the parties; otherwise, they will be refunded and the protocol halts. In the ideal model also the simulator proceeds to prepare its inputs only if a sufficient amount of deposit has been put in the contract; otherwise, it would send message $abort_1$ to TTP. Thus, in both models, the parties proceed to prepare their inputs only if $Q^{\text{Init}}(.) \rightarrow 1$. In the real model, if there is an abort after the parties ensure there is enough deposit and before client D provides its encoded input to the contract, then all parties would be able to retrieve their deposit in full; in this case, the aborting adversary would not be able to learn anything about honest parties input sets, because the parties' input sets are still blinded by random blinding polynomials known only to client D. In the ideal model, if there is any abort during steps 7–14, then the simulator sends $abort_2$ to TTP and instructs the ledger to refund the coins that every party deposited. Also, in the case of an abort (within the above two points of time), the auditor is not involved. Thus, in both models, in the case of an abort within the above points of time, we would have $Q^{\text{F-A}}(.) \rightarrow 1$. In the real model, if $Flag = True$, then all parties would be able to learn the intersection and the smart contract refunds all parties, i.e., sends each party $\ddot{y} + \ddot{ch}$ amount which is the amount each party initially deposited.

In the ideal model, when $Flag = True$, then $\mathsf{Sim}_A^{\text{JUS}}$ can extract the intersection (by summing the output of VOPR provided by all parties and removing the blinding polynomials) and sends back each party's deposit, i.e., $\ddot{y} + \ddot{ch}$ amount. Hence, in both models in the case of $Flag = True$, when all of the parties receive the result, we would have $Q^{\text{Del}}(.) \rightarrow 1$. In the real model, when $Flag = False$, only the adversary which might corrupt m' clients would be able to learn the result; in this case, the contract sends (i) $\ddot{ch}$ amount to the auditor, and (ii) $\frac{m' \cdot (\ddot{y}+\ddot{ch}) - \ddot{ch}}{m-m'}$ amount as a compensation, to each honest party, in addition to each party's deposit $\ddot{y}+\ddot{ch}$. In the ideal model, when $Flag = False$, $\mathsf{Sim}_A^{\text{JUS}}$ sends $abort_3$ to TTP and instructs the ledger to distribute the same amount among the auditor (e.g., with address adr_j) and every honest party (e.g., with address adr_i) as the contract does in the real model. Thus, in both models when $Flag = False$, we would have $Q^{\text{UF-A}}(.,.,.,.,adr_i) \rightarrow (a = 1, .)$ and $Q^{\text{UF-A}}(.,.,.,.,adr_j) \rightarrow (., b = 1)$.

We conclude that the distributions of the joint outputs of the honest client $C \in \hat{P}$, client D, Aud, and the adversary in the real and ideal models are computationally indistinguishable.

Case 2: Corrupt dealer D. In the real execution, the dealer's view is defined as follows:

$$\mathsf{View}_D^{\texttt{JUS}}\Big(S^{(D)}, (S^{(1)}, ..., S^{(m)})\Big) =$$

$$\{S^{(D)}, adr_{sc}, m \cdot (\ddot{y} + \ddot{ch}), r_D, \mathsf{View}_D^{\texttt{CT}}, k, g, q, \mathsf{View}_D^{\texttt{VOPR}}, \boldsymbol{\nu}^{(A_1)}, ..., \boldsymbol{\nu}^{(A_m)}, S_{\cap}\}$$

where $\mathsf{View}_D^{\texttt{CT}}$ and $\mathsf{View}_D^{\texttt{VOPR}}$ refer to the dealer's real-model view during the execution of CT and VOPR respectively. Also, r_D is the outcome of internal random coins of client D and adr_{sc} is the address of contract $\mathcal{SC}_{\texttt{JUS}}$. The simulator $\mathsf{Sim}_D^{\texttt{JUS}}$, which receives all parties' input sets, works as follows.

1. receives from the subroutine adversary polynomials $\boldsymbol{\zeta}, (\boldsymbol{\gamma}^{(A_1)}, \boldsymbol{\delta}^{(A_1)}), ..., (\boldsymbol{\gamma}^{(A_m)}, \boldsymbol{\delta}^{(A_m)})$, $(\boldsymbol{\omega}'^{(A_1)}, \boldsymbol{\rho}'^{(A_1)}), ..., (\boldsymbol{\omega}'^{(A_m)}, \boldsymbol{\rho}'^{(A_m)})$, where $deg(\boldsymbol{\gamma}^{(C)}) = deg(\boldsymbol{\delta}^{(C)}) = 3d+1, deg(\boldsymbol{\omega}'^{(C)}) = deg(\boldsymbol{\rho}'^{(C)}) = d$, and $deg(\boldsymbol{\zeta}) = 1$, where $C \in \{A_1, ..., A_m\}$.
2. generates an empty view. It appends to the view, the input set $S^{(D)}$. It constructs and deploys a smart contract. Let adr_{sc} be the contract's address. It appends adr_{sc} to the view.
3. appends to the view integer $m \cdot (\ddot{y} + \ddot{ch})$ and coins r'_D chosen uniformly at random.
4. extracts the simulation of CT from CT's simulator for client D. Let $\mathsf{Sim}_D^{\texttt{CT}}$ be the simulation. It appends $\mathsf{Sim}_D^{\texttt{CT}}$ to the view.
5. picks a random key, k', and derives pseudorandom values $z'_{i,j}$ from the key (the same way is done in Figure 2). It constructs a Merkle tree on top of all values $z'_{i,j}$. Let g' be the root of the resulting tree. It appends k', g', and $q' = \mathtt{H}(k')$ to the view.
6. invokes VOPR's functionality twice and extracts the simulation of VOPR from VOPR's simulator for client D. Let $\mathsf{Sim}_D^{\texttt{VOPR}}$ be the simulation. It appends $\mathsf{Sim}_D^{\texttt{VOPR}}$ to the view.
7. given the parties' input sets, computes a polynomial $\boldsymbol{\pi}$ that represents the intersection of the sets.
8. picks m random polynomials $\boldsymbol{\tau}^{(A_1)}, ..., \boldsymbol{\tau}^{(A_m)}$ of degree $3d+1$ such that their sum is 0.
9. picks m pairs of random polynomials $(\boldsymbol{\omega}^{(A_1)}, \boldsymbol{\rho}^{(A_1)}), ..., (\boldsymbol{\omega}^{(A_m)}, \boldsymbol{\rho}^{(A_m)})$, where each polynomial is of degree d. Then, $\mathsf{Sim}_D^{\texttt{JUS}}$ for each client $C \in \{A_1, ..., A_m\}$ computes polynomial $\boldsymbol{\nu}^{(C)} = \boldsymbol{\zeta} \cdot \boldsymbol{\pi} \cdot (\boldsymbol{\omega}^{(C)} \cdot \boldsymbol{\omega}'^{(C)} + \boldsymbol{\rho}^{(C)} \cdot \boldsymbol{\rho}'^{(C)}) + \boldsymbol{\delta}^{(C)} + \boldsymbol{\gamma}^{(C)} + \boldsymbol{\tau}^{(C)}$.
10. appends $\boldsymbol{\nu}^{(A_1)}, ..., \boldsymbol{\nu}^{(A_m)}$ and the intersection of the sets $S'_{\cap}$ to the view.

Next, we will show that the two views are computationally indistinguishable. D's input $S^{(D)}$ is identical in both models; therefore, they have identical distributions. Also, the contract's address has the same distribution in both views, and so has the integer $m \cdot (\ddot{y} + \ddot{ch})$. Since the real-model semi-honest adversary samples its randomness according to the protocol's description, the random coins in both models (i.e., r_D and r'_D) have identical distributions. Moreover, due to the security of CT, $\mathsf{View}_D^{\texttt{CT}}$ and $\mathsf{Sim}_D^{\texttt{CT}}$ have identical distributions. Keys k and k' have identical distributions, as both have been picked uniformly at random from the same domain. In the real model, each element of the pair (g, p) is the output of a deterministic function on a random value k. We know that k in the real

model has identical distribution to k' in the ideal model, so do the evaluations of deterministic functions (i.e., Merkle tree, H, and PRF) on them. Therefore, each pair (g, q) in the real model component-wise has an identical distribution to each pair (g', q') in the ideal model. Furthermore, due to the security of the VOPR, $\mathsf{View}_D^{\text{VOPR}}$ and $\mathsf{Sim}_D^{\text{VOPR}}$ have identical distributions.

In the real model, each $\boldsymbol{\nu}^{(C)}$ has been blinded by a pseudorandom polynomial (i.e., derived from PRF's output) unknown to client D. In the ideal model, however, each $\boldsymbol{\nu}^{(C)}$ has been blinded by a random polynomial unknown to client D. Due to the security of PRF, its outputs are computationally indistinguishable from truly random values. Therefore, $\boldsymbol{\nu}^{(C)}$ in the real model and $\boldsymbol{\nu}^{(C)}$ in the ideal model are computationally indistinguishable. Now we focus on the sum of all $\boldsymbol{\nu}^{(C)}$ in the real model and the sum of all $\boldsymbol{\nu}^{(C)}$ in the ideal model, as adding them together would remove the above blinding polynomials that are unknown to client D. Specifically, in the real model, after client D sums all $\boldsymbol{\nu}^{(C)}$ and removes the blinding factors and $\boldsymbol{\zeta}$ that it initially imposed, it would get a polynomial of the form $\hat{\boldsymbol{\phi}} = \frac{\sum_{C=A_1}^{A_m} \boldsymbol{\nu}^{(C)} - \sum_{C=A_1}^{A_m} (\boldsymbol{\gamma}^{(D,C)} + \boldsymbol{\delta}^{(D,C)})}{\boldsymbol{\zeta}} = \sum_{C=A_1}^{A_m} (\boldsymbol{\omega}^{(D,C)} \cdot \boldsymbol{\omega}^{(C,D)} \cdot \boldsymbol{\pi}^{(C)}) + \boldsymbol{\pi}^{(D)} \cdot \sum_{C=A_1}^{A_m} (\boldsymbol{\rho}^{(D,C)} \cdot \boldsymbol{\rho}^{(C,D)})$, where $\boldsymbol{\omega}^{(C,D)}$ and $\boldsymbol{\rho}^{(C,D)}$ are random polynomials unknown to client D. In the ideal model, after summing all $\boldsymbol{\nu}^{(C)}$ and removing the random polynomials that it already knows, it would get a polynomial of the following form: $\hat{\boldsymbol{\phi}}' = \frac{\sum_{C=A_1}^{A_m} \boldsymbol{\nu}^{(C)} - \sum_{C=A_1}^{A_m} (\boldsymbol{\gamma}^{(C)} + \boldsymbol{\delta}^{(C)})}{\boldsymbol{\zeta}} = \boldsymbol{\pi} \cdot \sum_{C=A_1}^{A_m} (\boldsymbol{\omega}'^{(C)} \cdot \boldsymbol{\omega}^{(C)}) + (\boldsymbol{\rho}'^{(C)} \cdot \boldsymbol{\rho}^{(C)})$.

As shown in Section 3.9, polynomial $\hat{\boldsymbol{\phi}}$ has the form $\boldsymbol{\mu} \cdot gcd(\boldsymbol{\pi}^{(D)}, \boldsymbol{\pi}^{(A_1)}, ..., \boldsymbol{\pi}^{(A_m)})$, where $\boldsymbol{\mu}$ is a uniformly random polynomial and $gcd(\boldsymbol{\pi}^{(D)}, \boldsymbol{\pi}^{(A_1)}, ..., \boldsymbol{\pi}^{(A_m)})$ represents the intersection of the input sets. Moreover, it is evident that $\hat{\boldsymbol{\phi}}'$ has the form $\boldsymbol{\mu} \cdot \boldsymbol{\pi}$, where $\boldsymbol{\mu}$ is a random polynomial and $\boldsymbol{\pi}$ represents the intersection. We know that both $gcd(\boldsymbol{\pi}^{(D)}, \boldsymbol{\pi}^{(A_1)}, ..., \boldsymbol{\pi}^{(A_m)})$ and $\boldsymbol{\pi}$ represent the same intersection, also $\boldsymbol{\mu}$ in the real model and $\boldsymbol{\mu}$ in the ideal model have identical distribution as they are uniformly random polynomials. Thus, two polynomials $\hat{\boldsymbol{\phi}}$ and $\hat{\boldsymbol{\phi}}'$ are indistinguishable. Also, the output $S_\cap$ is identical in both views. We conclude that the two views are computationally indistinguishable.

Case 3: Corrupt auditor. In this case, by using the proof that we have already provided for Case 1 (i.e., $m-1$ client A_js are corrupt), we can easily construct a simulator that generates a view computationally distinguishable from the real-model semi-honest auditor. The reason is that, in the worst-case scenario where $m-1$ malicious client A_js reveal their input sets and randomness to the auditor, the auditor's view would be similar to the view of these corrupt clients, which we have shown to be indistinguishable. The only extra messages the auditor generates, that a corrupt client A_j would not see in plaintext, are random blinding polynomials $(\boldsymbol{\xi}^{(A_1)}, ..., \boldsymbol{\xi}^{(A_m)})$ generated during the execution of Audit(.) of ZSPA-A; however, these polynomials are picked uniformly at random and independent of the parties' input sets. Thus, if the smart contract detects

misbehaviour and invokes the auditor, even if $m-1$ corrupt client A_j reveals their input sets, then the auditor cannot learn anything about honest parties' input sets.

Case 4: Corrupt public. In the real model, the view of the public (i.e., non-participants of the protocol) is defined as below:

$$\mathsf{View}^{\mathtt{JUS}}_{Pub}\Big(\perp, S^{(D)}, (S^{(A_1)}, ..., S^{(A_m)})\Big) =$$
$$\{\perp, adr_{sc}, (m+1)\cdot(\ddot{y}+\ddot{ch}), k, g, q, \boldsymbol{\nu}^{(A_1)}, ..., \boldsymbol{\nu}^{(A_m)}, \boldsymbol{\nu}^{(D)}\}$$

Now, we describe how the simulator $\mathsf{Sim}^{\mathtt{JUS}}_{Pub}$ works.

1. generates an empty view and appends to it an empty symbol, $\perp$. It constructs and deploys a smart contract. It appends the contract's address, adr_{sc} and integer $(m+1)\cdot(\ddot{y}+\ddot{ch})$ to the view.
2. picks a random key, k', and derives pseudorandom values $z'_{i,j}$ from the key, in the same way, done in Figure 2. It constructs a Merkle tree on top of the $z'_{i,j}$ values. Let g' be the root of the resulting tree. It appends k', g', and $q' = \mathtt{H}(k')$ to the view.
3. for each client $C \in \{A_1, ..., A_m\}$ and client D generates a random polynomial of degree $3d+1$ (for each bin), i.e., $\boldsymbol{\nu}^{(A_1)}, ..., \boldsymbol{\nu}^{(A_m)}, \boldsymbol{\nu}^{(D)}$.

Next, we will show that the two views are computationally indistinguishable. In both views, $\perp$ is identical. Also, the contract's addresses (i.e., adr_{sc}) has the same distribution in both views, and so has the integer $(m+1)\cdot(\ddot{y}+\ddot{ch})$. Keys k and k' have identical distributions as well, because both of them have been picked uniformly at random from the same domain. In the real model, each element of pair (g,p) is the output of a deterministic function on the random key k. We know that k in the real model has identical distribution to k' in the ideal model, and so do the evaluations of deterministic functions on them. Hence, each pair (g,q) in the real model component-wise has an identical distribution to each pair (g',q') in the ideal model. In the real model, each polynomial $\boldsymbol{\nu}^{(C)}$ is a blinded polynomial comprising of two uniformly random blinding polynomials (i.e., $\boldsymbol{\gamma}^{(D,C)}$ and $\boldsymbol{\delta}^{(D,C)}$) unknown to the adversary. In the ideal model, each polynomial $\boldsymbol{\nu}^{(C)}$ is a random polynomial; thus, polynomials $\boldsymbol{\nu}^{(A_1)}, ..., \boldsymbol{\nu}^{(A_m)}$ in the real model have identical distribution to polynomials $\boldsymbol{\nu}^{(A_1)}, ..., \boldsymbol{\nu}^{(A_m)}$ in the ideal model. Similarly, polynomial $\boldsymbol{\nu}^{(D)}$ has been blinded in the real model; its blinding factors are the additive inverse of the sum of the random polynomials $\boldsymbol{\gamma}^{(D,C)}$ and $\boldsymbol{\delta}^{(D,C)}$ unknown to the adversary. In the ideal model, polynomial $\boldsymbol{\nu}^{(D)}$ is a uniformly random polynomial; thus, $\boldsymbol{\nu}^{(D)}$ in the real model and $\boldsymbol{\nu}^{(D)}$ in the ideal model have identical distributions. Moreover, in the real model even though the sum $\boldsymbol{\phi}$ of polynomials $\boldsymbol{\nu}^{(A_1)}, ..., \boldsymbol{\nu}^{(A_m)}, \boldsymbol{\nu}^{(D)}$ would remove some of the blinding random polynomials, it is still a blinded polynomial with a pseudorandom blinding factor $\boldsymbol{\gamma}'$ (derived from the output of $\mathtt{PRF}$), unknown to the adversary. In the ideal model, the sum of polynomials $\boldsymbol{\nu}^{(A_1)}, ..., \boldsymbol{\nu}^{(A_m)}, \boldsymbol{\nu}^{(D)}$ is also a random polynomial. Thus, the sum of the above polynomials in the real model is computationally indistinguishable from the sum of those polynomials in the ideal model. We conclude that the two views are computationally indistinguishable. □

J Workflow of Anesidora

Figure 6 outlines the interaction between parties in Anesidora.

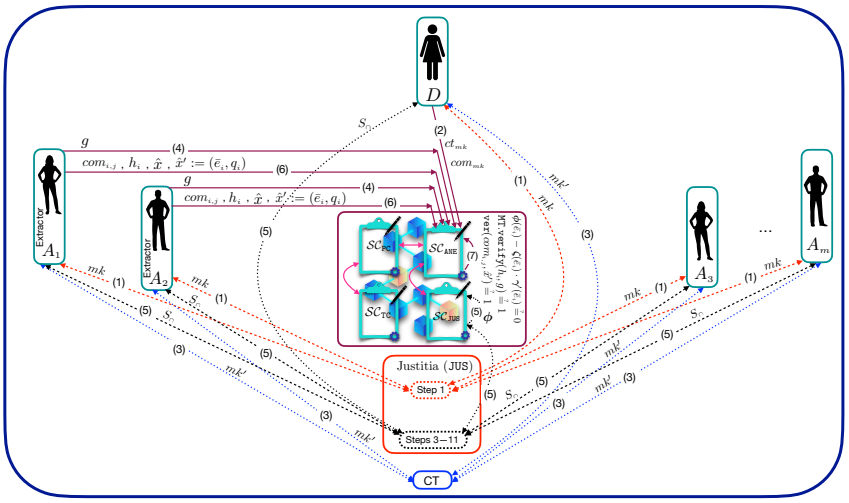

Fig. 6: Outline of the interactions between parties in Anesidora

K Proof of ANE

In this section, we prove Theorem 7, i.e., the security of ANE.

Proof. We prove the theorem by considering the case where each party is corrupt, at a time.

Case 1: Corrupt extractors $\{A_1, A_2\}$ and $m-3$ clients in $\{A_3, ..., A_m\}$. Let set G include extractors $\{A_1, A_2\}$ and a set of at most $m-3$ corrupt clients in $\{A_3, ..., A_m\}$. Let set $\hat{G}$ be a set of honest clients in $\{A_3, ..., A_m\}$. Also, let $\mathsf{Sim}_{\mathcal{A}}^{\mathtt{ANE}}$ be the simulator. We let the simulator interact with (i) active adversary $\mathcal{A}'$ that may corrupt $m-3$ clients in $\{A_3, ..., A_m\}$, and (ii) two rational adversaries $\mathcal{A}'' := (\mathcal{A}_1, \mathcal{A}_2)$ that corrupt extractors (A_1, A_2) component-wise. In the simulation, before the point where the extractors are invoked to provide proofs and results, the simulator directly deals with active adversary $\mathcal{A}'$. However, when the extractors are involved (to generate proofs and extract the result) we require the simulator to interact with each rational adversary $\mathcal{A}_1$ and $\mathcal{A}_2$. We allow these two subroutine adversaries $(\mathcal{A}', \mathcal{A}'')$ to internally interact with each other. Now, we explain how $\mathsf{Sim}_{\mathcal{A}}^{\mathtt{ANE}}$, which receives the input sets of honest dealer D and honest client(s) in $\hat{G}$, works.

1. constructs and deploys two smart contracts (for JUS and ANE). It sends the contracts' addresses to $\mathcal{A}'$. It also simulates CT and receives the output value, mk, from its functionality, f_{CT}.
2. deposits $S_{min} \cdot \ddot{v}$ amount to $\mathcal{SC}_{\text{ANE}}$ if buyer A_m is honest, i.e., $A_m \in \hat{G}$. Otherwise, $\mathsf{Sim}_A^{\text{ANE}}$ checks if $\mathcal{A}'$ has deposited $S_{min} \cdot \ddot{v}$ amount in $\mathcal{SC}_{\text{ANE}}$. If the check fails, it instructs the ledger to refund the coins that every party deposited and sends message $abort_1$ to TTP (and accordingly to all parties); it outputs whatever $\mathcal{A}'$ outputs and then halts.
3. constructs and deploys a (Prisoner's) contract and transfers $\ddot{w} = S_{min} \cdot \ddot{r}$ amount for each extractor. $\mathsf{Sim}_A^{\text{ANE}}$ ensures that each extractor deposited $\ddot{d}' = \ddot{d} + S_{min} \cdot \ddot{f}$ coins in this contract; otherwise, it instructs the ledger to refund the coins that every party deposited and sends message $abort_1$ to TTP; it outputs whatever $\mathcal{A}'$ outputs and then halts.
4. encrypts mk under the public key of the dispute resolver; let ct_{mk} be the resulting ciphertext. It also generates a commitment of mk as follows: $com_{mk} = \text{Com}(mk, \text{PRF}(mk, 0))$. It stores ct_{mk} and com_{mk} in $\mathcal{SC}_{\text{ANE}}$.
5. simulates CT again and receives the output value mk' from f_{CT}.
6. receives from $\mathcal{A}'$ a Merkle tree's root g' for each extractor.
7. simulates the steps of 3–11 in JUS. For completeness, we include the steps that the simulator takes in this proof. Specifically, $\mathsf{Sim}_A^{\text{ANE}}$:
   (a) simulates ZSPA-A for each bin and receives the output value (k, g, q) from $f^{\text{ZSPA-A}}$.
   (b) deposits in the contract the amount of $\ddot{y}' = S_{min} \cdot \ddot{v} + \ddot{ch}$ for client D and each honest client in $\hat{G}$. It sends to $\mathcal{A}'$ the amount deposited in the contract.
   (c) checks if $\mathcal{A}'$ has deposited $\ddot{y}' \cdot |G|$ amount (in addition to $\ddot{d}'$ amount deposited in step 3 above). If the check fails, it instructs the ledger to refund the coins that every party deposited and sends message $abort_1$ to TTP (and accordingly to all parties); it outputs whatever $\mathcal{A}'$ outputs and then halts.
   (d) picks a random polynomial $\boldsymbol{\zeta}$ of degree 1, for each bin. $\mathsf{Sim}_A^{\text{ANE}}$, for each client $C \in \{A_1, ..., A_m\}$ allocates to each bin two degree d random polynomials: $(\boldsymbol{\omega}^{(D,C)}, \boldsymbol{\rho}^{(D,C)})$, and two degree $3d+1$ random polynomials: $(\boldsymbol{\gamma}^{(D,C)}, \boldsymbol{\delta}^{(D,C)})$. Also, $\mathsf{Sim}_A^{\text{ANE}}$ for each honest client $C' \in \hat{G}$, for each bin, picks two degree d random polynomials: $(\boldsymbol{\omega}^{(C',D)}, \boldsymbol{\rho}^{(C',D)})$.
   (e) simulates VOPR using inputs $\boldsymbol{\zeta} \cdot \boldsymbol{\omega}^{(D,C)}$ and $\boldsymbol{\gamma}^{(D,C)}$ for each bin. It receives the inputs of clients $C'' \in G$, i.e., $\boldsymbol{\omega}^{(C'',D)} \cdot \boldsymbol{\pi}^{(C'')}$, from its functionality f^{VOPR}, for each bin.
   (f) extracts the roots of polynomial $\boldsymbol{\omega}^{(C'',D)} \cdot \boldsymbol{\pi}^{(C'')}$ for each bin and appends those roots that are in the sets universe to a new set $S^{(C'')}$.
   (g) simulates again VOPR using inputs $\boldsymbol{\zeta} \cdot \boldsymbol{\rho}^{(D,C)} \cdot \boldsymbol{\pi}^{(D)}$ and $\boldsymbol{\delta}^{(D,C)}$, for each bin.
   (h) sends to TTP the input sets of all parties; specifically, (i) client D's input set: $S^{(D)}$, (ii) honest clients' input sets: $S^{(C')}$ for all C' in $\hat{G}$, and (iii) $\mathcal{A}'$'s input sets: $S^{(C'')}$, for all C'' in G. For each bin, it receives the intersection set, $S_{\cap}$, from TTP.

(i) represents the intersection set for each bin as a polynomial, $\boldsymbol{\pi}$, as follows. First, it encrypts each element s_i of $S_\cap$ as $e_i = \mathtt{PRP}(mk', s_i)$. Second, it encodes each encrypted element as $\bar{e}_i = e_i || \mathtt{H}(e_i)$. Third, it constructs $\boldsymbol{\pi}$ as $\boldsymbol{\pi} = \prod_{i=1}^{|S_\cap|}(x - s_i) \cdot \prod_{j=1}^{d-|S_\cap|}(x - u_j)$, where u_j is a dummy value.

(j) constructs polynomials $\boldsymbol{\theta}_1^{(C')} = \boldsymbol{\zeta} \cdot \boldsymbol{\omega}^{(D,C')} \cdot \boldsymbol{\omega}^{(C',D)} \cdot \boldsymbol{\pi} + \boldsymbol{\gamma}^{(D,C')}, \boldsymbol{\theta}_2^{(C')} = \boldsymbol{\zeta} \cdot \boldsymbol{\rho}^{(D,C')} \cdot \boldsymbol{\rho}^{(C',D)} \cdot \boldsymbol{\pi} + \boldsymbol{\delta}^{(D,C')}$, and $\boldsymbol{\nu}^{(C')} = \boldsymbol{\theta}_1^{(C')} + \boldsymbol{\theta}_2^{(C')} + \boldsymbol{\tau}^{(C')}$, for each bin and each honest client $C' \in \hat{G}$, such that $\boldsymbol{\tau}^{(C)} = \sum_{i=0}^{3d+2} z_{i,c} \cdot x^i$ and each value $z_{i,c}$ is derived from key k generated in step 7a. It sends to $\mathcal{A}'$ polynomial $\boldsymbol{\nu}^{(C')}$ for each bin and each honest client $C' \in \hat{G}$.

(k) receives $\boldsymbol{\nu}^{(C'')}$ from $\mathcal{A}'$, for each bin and each corrupt client $C'' \in G$. It checks whether the output for every C'' has been provided. Otherwise, it halts.

(l) if there is any abort within steps 7e–7k, then it sends $abort_2$ to TTP and instructs the ledger to refund the coins that every party deposited. It outputs whatever $\mathcal{A}'$ outputs and then halts.

(m) constructs polynomial $\boldsymbol{\nu}^{(D)} = \boldsymbol{\zeta} \cdot \boldsymbol{\omega'}^{(D)} \cdot \boldsymbol{\pi} - \sum_{C=A_1}^{A_m} (\boldsymbol{\gamma}^{(D,C)} + \boldsymbol{\delta}^{(D,C)}) + \boldsymbol{\zeta} \cdot \boldsymbol{\gamma}'$ for each bin on behalf of client D, where $\boldsymbol{\omega'}^{(D)}$ is a fresh random polynomial of degree d and $\boldsymbol{\gamma}'$ is a pseudorandom polynomial derived from mk.

(n) sends to $\mathcal{A}'$ polynomials $\boldsymbol{\nu}^{(D)}$ and $\boldsymbol{\zeta}$ for each bin.

(o) computes polynomial $\boldsymbol{\phi}'$ as $\boldsymbol{\phi}' = \sum_{\forall C'' \in G} \boldsymbol{\nu}^{(C'')} - \sum_{\forall C'' \in G} (\boldsymbol{\gamma}^{(D,C'')} + \boldsymbol{\delta}^{(D,C'')})$, for every bin. Next, it checks if $\boldsymbol{\zeta}$ divides $\boldsymbol{\phi}'$, for every bin. If the check passes, it sets $Flag = True$. Otherwise, it sets $Flag = False$.

(p) if $Flag = True$, then instructs the ledger to send back each party's deposit, i.e., $\ddot{y}'$ amount. It sends a message *deliver* to TTP. It proceeds to step 8 below.

(q) if $Flag = False$:

i. receives $|G|$ keys of the PRF from $\mathcal{A}'$, i.e., $\vec{k}' = [k'_1, ..., k'_{|G|}]$, for every bin.

ii. checks if $k'_j = k$, for every $k'_j \in \vec{k}'$. Recall, k was generated in step 3. It constructs an empty list L' and appends to it the indices (e.g., j) of the keys that do not pass the above check.

iii. receives from $f^{\mathtt{ZSPA\text{-}A}}$ the output containing a vector of random polynomials, $\vec{\mu}'$, for each valid key.

iv. sends to $\mathcal{A}'$, L' and $\vec{\mu}'$, for every bin.

v. for each bin of client C whose index is not in L' computes polynomial $\boldsymbol{\chi}^{(D,C)}$ as $\boldsymbol{\chi}^{(D,C)} = \boldsymbol{\zeta} \cdot \boldsymbol{\eta}^{(D,C)} - (\boldsymbol{\gamma}^{(D,C)} + \boldsymbol{\delta}^{(D,C)})$, where $\boldsymbol{\eta}^{(D,C)}$ is a fresh random polynomial of degree $3d + 1$. Note that C includes both honest and corrupt clients, except those clients whose index is in L'. $\mathsf{Sim}_A^{\mathtt{ANE}}$ sends every polynomial $\boldsymbol{\chi}^{(D,C)}$ to $\mathcal{A}'$.

vi. given each $\boldsymbol{\nu}^{(C'')}$ (by $\mathcal{A}'$ in step 7k), computes polynomial $\boldsymbol{\phi'}^{(C'')}$ as follows: $\boldsymbol{\phi'}^{(C'')} = \boldsymbol{\nu}^{(C'')} - \boldsymbol{\gamma}^{(D,C'')} - \boldsymbol{\delta}^{(D,C'')}$, for every bin. $\mathsf{Sim}_A^{\mathtt{ANE}}$ checks

if $\boldsymbol{\zeta}$ divides $\boldsymbol{\phi}'^{(C'')}$, for every bin. It appends the index of those clients that did not pass the above check to a new list, L''.

vii. if L' or L'' is not empty, then instructs the ledger: (a) to refund $\ddot{y}'$ amount to each client whose index is not in L' and L'', (b) to retrieve $\ddot{ch}$ amount from the adversary (i.e., one of the parties whose index is in one of the lists) and send the $\ddot{ch}$ amount to Aud, and (c) to reward and compensate each honest party (whose index is not in the two lists) $\frac{m' \cdot \ddot{y}' - \ddot{ch}}{m - m'}$ amount, where $m' = |L'| + |L''|$. Then, it sends message $abort_3$ to TTP.

viii. outputs whatever $\mathcal{A}'$ outputs and halts.

8. for each $I \in \{1, 2\}$, receives from $\mathcal{A}_I$ (1) a set $E^{(I)}$ of encoded encrypted elements, e.g., $\bar{e}_i$, in the intersection, (2) each $\bar{e}_i$'s commitment $com_{i,j}$, (3) each $com_{i,j}$'s opening $\hat{x}'$, (4) a proof h_i that each $com_{i,j}$ is a leaf node of a Merkle tree with root g' (given to simulator in step 6 above), and (5) the opening $\hat{x}$ of commitment com_{mk}.
9. encrypts each element s_i of $S_\cap$ as $e_i = \mathtt{PRP}(mk', s_i)$. Then, it encodes each encrypted element as $\bar{e}_i = e_i || \mathtt{H}(e_i)$. Let set S' include all encoded encrypted elements in the intersection.
10. sings a $\mathcal{SC}_{\text{TC}}$ with $\mathcal{A}_I$, if $\mathcal{A}_I$ decides to be a traitor extractor. In this case, $\mathcal{A}_I$, provides the intersection to $\mathcal{SC}_{\text{TC}}$. $\mathsf{Sim}_A^{\text{ANE}}$ checks this intersection's validity. Shortly (in step 16b), we will explain how $\mathsf{Sim}_A^{\text{ANE}}$ acts based on the outcome of this check.
11. checks if each set $E^{(I)}$ equals set S'.
12. checks if $com_{i,j}$ matches the opening $\hat{x}'$ and the opening corresponds to a unique element in S'.
13. verifies each commitment's proof, h_i. Specifically, given the proof and root g, it ensures the commitment $com_{i,j}$ is a leaf node of a Merkle tree with a root node g'. It also checks whether the opening $\hat{x}$ matches com_{mk}.
14. if all the checks in steps 11–13 pass, then instructs the ledger (i) to take $|S_\cap| \cdot m \cdot \ddot{l}$ amount from the buyer's deposit and distributes it among all clients, except the buyer, (ii) to return the extractors deposit (i.e., $\ddot{d}'$ amount each) and pay each extractor $|S_\cap| \cdot \ddot{r}$ amount, and (iii) to return $(S_{min} - |S_\cap|) \cdot \ddot{v}$ amount to the buyer.
15. if neither extractor sends the extractor's set intersection $(E^{(A_1)}, E^{(A_2)})$ in step 8, then instructs the ledger (i) to refund the buyer, by sending $S_{min} \cdot \ddot{v}$ amount back to the buyer and (ii) to retrieve each extractor's deposit (i.e., $\ddot{d}'$ amount) from $\mathcal{SC}_{\text{PC}}$ and distribute it among the rest of the clients (except the buyer and extractors).
16. if the checks in step 14 fail or in step 15 both $\mathcal{A}_1$ and $\mathcal{A}_2$ send the extractors' set intersection but they are inconsistent with each other, then it tags the extractor whose proof or set intersection was invalid as a misbehaving extractor. $\mathsf{Sim}_A^{\text{ANE}}$ instructs the ledger to pay the auditor (of $\mathcal{SC}_{\text{PC}}$) the total amount of $\ddot{ch}$ coins taken from the misbehaving extractor(s) deposit. Furthermore, $\mathsf{Sim}_A^{\text{ANE}}$ takes the following steps.
    (a) if both extractors cheated and there is no traitor, then instructs the ledger (i) to refund the buyer $S_{min} \cdot \ddot{v}$ amount, and (ii) to take $2\ddot{d}' - \ddot{ch}$

amount from the misbehaving extractors' deposit and distribute it to the rest of the clients except the buyer and extractors.

(b) if both extractors cheated, there is a traitor, and the traitor delivered a correct result (in step 10), then instructs the ledger (i) to take $\ddot{d}' - \ddot{d}$ amount from the other misbehaving extractor's deposit and distribute it among the rest of the clients (except the buyer and dishonest extractor), (ii) to distribute $|S_{\cap}| \cdot \ddot{r} + \ddot{d}' + \ddot{d} - \ddot{ch}$ amount to the traitor, (iii) to refund the traitor $\ddot{ch}$ amount, and (iv) to refund the buyer $S_{min} \cdot \ddot{v} - |S_{\cap}| \cdot \ddot{r}$ amount.

(c) if both extractors cheated, there is a traitor, and the traitor delivered an incorrect result (in step 10), then instructs the ledger to distribute coins the same way it does in step 16a.

(d) if one of the extractors cheated and there is no traitor, then instructs the ledger (i) to return the honest extractor's deposit (i.e., $\ddot{d}'$ amount), (ii) to pay the honest extractor $|S_{\cap}| \cdot \ddot{r}$ amount, (iii) to pay this extractor $\ddot{d} - \ddot{ch}$ amount taken from the dishonest extractor's deposit, and (iv) to pay the buyer and the rest of the clients the same way it does in step 16b.

(e) if one of the extractors cheated, there is a traitor, and the traitor delivered a correct result (in step 10), then instructs the ledger (i) to return the other honest extractor's deposit (i.e., $\ddot{d}'$ amount), (ii) to pay the honest extractor $|S_{\cap}| \cdot \ddot{r}$ amount, taken from the buyer's deposit, (iii) to pay the honest extractor $\ddot{d} - \ddot{ch}$ amount, taken from the traitor's deposit, (iv) to pay to the traitor $|S_{\cap}| \cdot \ddot{r}$ amount, taken from the buyer's deposit, (v) to refund the traitor $\ddot{d}' - \ddot{d}$ amount, (vi) to refund the traitor $\ddot{ch}$ amount, (vii) to take $|S_{\cap}| \cdot m \cdot \ddot{l}$ amount from the buyer's deposit and distribute it among all clients, except the buyer, and (viii) to return $(S_{min} - |S_{\cap}|) \cdot \ddot{v}$ amount back to the buyer.

(f) if one of the extractors cheated, there is a traitor, and the traitor delivered an incorrect result (in step 10), then instructs the ledger (i) to pay the honest extractor the same way it does in step 13(b)iiA, (ii) to refund the traitor $\ddot{ch}$ amount, and (iii) to pay the buyer and the rest of the clients in the same way it does in step 16b.

(g) outputs whatever $\mathcal{A}$ outputs and then halts.

Next, we show that the real and ideal models are computationally indistinguishable. We first focus on the adversary's output. The addresses of the smart contracts have identical distribution in both models. In the real and ideal models, the adversary sees the transcripts of ideal calls to f_{CT} as well as the functionality outputs (mk, mk'). Due to the security of CT (as we are in the f_{CT}-hybrid world), the transcripts of f_{CT} in both models have identical distribution, so have the random outputs of f_{CT}, i.e., (mk, mk'). Also, the deposit amounts $S_{min} \cdot \ddot{v}$ and $\ddot{w}$ have identical distributions in both models. Due to the semantical security of the public key encryption, the ciphertext ct_{mk} in the real model is computationally indistinguishable from the ciphertext ct_{mk} in the ideal model. Due to the hiding property of the commitment scheme, commitment com_{mk} in the real model is

computationally indistinguishable from commitment com_{mk} in the ideal model. Moreover, due to the security of JUS, all transcripts and outputs produced in the ideal model, in step 7 above, have identical distribution to the corresponding transcripts and outputs produced in JUS in the real model. The address of $\mathcal{SC}_{\text{TC}}$ has the same distribution in both models. The amounts each party receives in the real and ideal models are the same, except when both extractors produce an identical and incorrect result (i.e., intersection) in the real model, as we will shortly discuss, this would not occur under the assumption that the extractors are rational and due to the security of the counter collusion smart contracts.

Now, we show that an honest party aborts with the same probability in the real and ideal models. As before, for the sake of completeness, we include the JUS in the following discussion as well. Due to the security of CT, an honest party, during CT invocation, aborts with the same probability in both models; in this case, the adversary learns nothing about the parties' input set and the sets' intersection as the parties have not sent out any encoded input set yet. In both models, an honest party can read the smart contract and check if sufficient amounts of coins have been deposited. Thus, it would halt with the same probability in both models. If the parties halt because of insufficient amounts of deposit, no one could learn about (i) the parties' input set and (ii) the sets' intersection because the inputs (representation) have not been dispatched at this point. Due to the security of ZSPA-A, an honest party during ZSPA-A execution aborts with the same probability in both models. In this case, an aborting adversary also learns nothing about the parties' input set and the sets' intersection.

Due to the security of VOPR, honest parties abort with the same probability in both models. In the case where a party aborts during the execution of VOPR, the adversary would learn nothing (i) about its counter party's input set, and (ii) about the rest of the honest parties' input sets and the intersection as the other parties' input sets remain blinded by random blinding factors known only to client D. In the real model, client D can check if all parties provided their encoded inputs via reading the state of the smart contract. The simulator can perform the same check to ensure $\mathcal{A}'$ has provided the encoded inputs of all corrupt parties. So, in both models, an honest party with the same probability detects if not all encoded inputs have been provided. In this case, if an adversary aborts and does not provide its encoded inputs (i.e., polynomials $\boldsymbol{\nu}^{(C'')}$), then it learns nothing about the honest parties' input sets and the intersection, for the same reason explained above.

In the real model, the contract sums every client C's polynomial $\boldsymbol{\nu}^{(C)}$ with each other and with client D's polynomial $\boldsymbol{\nu}^{(D)}$, that ultimately removes the blinding factors that D initially inserted (during the VOPR execution), and then checks if the result is divisible by $\boldsymbol{\zeta}$. Due to (a) Theorem 5, (b) the fact that the smart contract is given the random polynomial $\boldsymbol{\zeta}$ in plaintext, (c) no party (except honest D) knew polynomial $\boldsymbol{\zeta}$ before they send their input to the contract, and (d) the security of the contract (i.e., the adversary cannot influence the correctness of the smart contract's verifications), the contract can detect if a set of outputs of VOPR were tampered with, with a probability at least $1 - \epsilon(\lambda)$.

In the ideal model, $\mathsf{Sim}_A^{\text{ANE}}$ (in step 7o) can remove the blinding factors and it knows the random polynomial $\boldsymbol{\zeta}$. So, $\mathsf{Sim}_A^{\text{ANE}}$ can detect when $\mathcal{A}'$ tampers with a set of the outputs of VOPR (sent to $\mathsf{Sim}_A^{\text{ANE}}$) with a probability at least $1-\epsilon(\lambda)$, due to Theorem 5. Therefore, the smart contract in the real model and the simulator in the ideal model would abort with a similar probability.

Due to the security of ZSPA-A, the probability that in the real model an invalid $k_i \in \overrightarrow{k}$ is appended to L is similar to the probability that $\mathsf{Sim}_A^{\text{ANE}}$ detects an invalid $k_i' \in \overrightarrow{k'}$ in the ideal model. In the real model, when $Flag = False$, the smart contract can identify each ill-structured output of VOPR (i.e., $\boldsymbol{\nu}^{(C)}$) with a probability of at least $1 - \epsilon(\lambda)$ by checking whether $\boldsymbol{\zeta}$ divides $\boldsymbol{\iota}^{(C)}$, due to (a) Theorem 4 (i.e., unforgeable polynomial), (b) the fact that the smart contract is given $\boldsymbol{\zeta}$ in plaintext, (c) no party (except honest client D) knew anything about $\boldsymbol{\zeta}$ before they send their input to the contract, and (d) the security of the contract. In the ideal model, when $Flag = False$, given each $\boldsymbol{\nu}^{(C'')}$, $\mathsf{Sim}_A^{\text{ANE}}$ can remove its blinding factors from $\boldsymbol{\nu}^{(C'')}$ which results in $\boldsymbol{\phi}'^{(C'')}$ and then can check if $\boldsymbol{\zeta}$ divides $\boldsymbol{\phi}'^{(C'')}$, in step 7(q)vi. $\mathsf{Sim}_A^{\text{ANE}}$ can detect an ill-structured $\boldsymbol{\nu}^{(C'')}$ with a probability of at least $1 - \epsilon(\lambda)$, due to Theorem 4, the fact that the simulator is given $\boldsymbol{\zeta}$ in plaintext, and the adversary is not given any knowledge about $\boldsymbol{\zeta}$ before it sends to the simulator the outputs of VOPR. Therefore, the smart contract in the real model and $\mathsf{Sim}_A^{\text{ANE}}$ in the ideal model can detect an ill-structured input of an adversary with the same probability. The smart contract in the real model and $\mathsf{Sim}_A^{\text{ANE}}$ in the ideal model can detect and abort with the same probability if the adversary provides an invalid opening to each commitment $com_{i,j}$ and com_{mk}, due to the binding property of the commitment scheme. Also, the smart contract in the real model and $\mathsf{Sim}_A^{\text{ANE}}$ in the ideal model, can abort with the same probability if a Merkle tree proof is invalid, due to the security of the Merkle tree, i.e., due to the collision resistance of Merkle tree's hash function.

Note that in the ideal model, $\mathsf{Sim}_A^{\text{ANE}}$ can detect and abort with a probability of 1, if $\mathcal{A}_I$ does not send to the simulator all encoded encrypted elements of the intersection, i.e., when $E^{(I)} \neq S'$. Because the simulator already knows all elements in the intersection (and the encryption key). Thus, it can detect with a probability of 1 if both the intersection sets that the extractors provide are identical but incorrect. In the real world, if the extractors collude with each other and provide identical but incorrect intersections, then an honest client (or the smart contract) cannot detect it. Thus, the adversary can distinguish the two models, based on the probability of aborting. However, under the assumption that the smart contracts (of Dong *et al.* [16]) are secure (i.e., are counter-collusion), and the extractors are rational, such an event (i.e., providing identical but incorrect result without one extractor betraying the other) would not occur in either model, as the real model and $(\mathcal{A}_1, \mathcal{A}_2)$ rational adversaries follow the strategy that leads to a higher payoff. Specifically, as shown in [16], providing incorrect but identical results is not the preferred strategy of the extractors; instead, the betrayal of one extractor by the other is the most profitable strategy in the case of (enforceable) collusion between the two extractors. This also implies that the amounts that the extractors would receive in both models are identical.

Now, we analyse the output of the predicates $\bar{Q} := (Q^{\text{Init}}, Q_{\text{R}}^{\text{Del}}, Q_{\text{R}}^{\text{UF-A}}, Q^{\text{F-A}})$ in the real and ideal models. In the real model, all clients proceed to prepare their input set only if the predefined amount of coins have been deposited by the parties; otherwise (if in steps 2,4,5 of ANE and step 4 of JUS there is not enough deposit), they will be refunded and the protocol halts. In the ideal model, the simulator proceeds to prepare its inputs only if enough deposit has been placed in the contract. Otherwise, it would send message $abort_1$ to TTP, during steps 2–7c. Thus, in both models, the parties proceed to prepare their inputs only if $Q^{\text{Init}}(.) \rightarrow 1$. In the real model, if there is an abort after the parties ensure there is enough deposit and before client D provides its encoded input to the contract, then all parties can retrieve their deposit in full. In this case, the aborting adversary cannot learn anything about honest parties' input sets, as the parties' input sets have been blinded by random blinding polynomials known only to client D. In the ideal model, if there is any abort during steps 7e–7k, then the simulator sends $abort_2$ to TTP and instructs the ledger to refund every party's deposit. In the case of an abort, within the above two steps, the auditor is not involved, and paid. Therefore, in both models, in the case of an abort within the above steps, we would have $Q^{\text{F-A}}(.) \rightarrow 1$.

In the real model, if $Flag = True$, then all parties can locally extract the intersection, regardless of the extractors' behaviour. In this case, each honest party receives $\ddot{y}'$ amount that it initially deposited in $\mathcal{SC}_{\texttt{JUS}}$. Moreover, each honest party receives *at least* $|S_{\cap}| \cdot \ddot{l}$ amount as a reward, for contributing to the result. In this case, the honest buyer always collects the leftover of its deposit. Specifically, if both extractors act honestly, and the intersection cardinality is smaller than $|S_{min}|$, then the buyer collects its deposit leftover, after paying all honest parties. If any extractor misbehaves, then the honest buyer fully recovers its deposit (and the misbehaving extractor pays the rest). Even in the case that an extractor misbehaves and then becomes a traitor to correct its past misbehaviour, the buyer collects its deposit leftover if the intersection cardinality is smaller than $|S_{min}|$. In the ideal model, when $Flag = True$, then $\mathsf{Sim}_A^{\texttt{ANE}}$ can extract the intersection by summing the output of VOPR provided by all parties and removing the blinding polynomials. In this case, it sends back each party's deposit placed in $\mathcal{SC}_{\texttt{JUS}}$, i.e., $\ddot{y}'$ amount. Also, in this case, each honest party receives at least $|S_{\cap}| \cdot \ddot{l}$ amount as a reward and the honest buyer always collects the leftover of its deposit. Thus, in both models in the case of $Flag = True$, we would have $Q_{\text{R}}^{\text{Del}}(.) \rightarrow 1$.

In the real model, when $Flag = False$, only the adversary can learn the result. In this case, the contract sends (i) $\ddot{ch}$ amount to Aud, and (ii) $\frac{m' \cdot \ddot{y}' - \ddot{ch}}{m - m'}$ amount, as compensation and reward, to each honest party, in addition to each party's initial deposit. In the ideal model, when $Flag = False$, $\mathsf{Sim}_A^{\texttt{ANE}}$ sends $abort_3$ to TTP and instructs the ledger to distribute the same amount the contract distributes among the auditor (e.g., with address adr_j) and every honest party (e.g., with address adr_i) in the real model. Thus, in both models when $Flag = False$, we would have $Q_{\text{R}}^{\text{UF-A}}(., ., ., ., adr_i) \rightarrow (a = 1, .)$ and $Q_{\text{R}}^{\text{UF-A}}(., ., ., ., adr_j) \rightarrow (., b = 1)$.

We conclude that the distribution of the joint outputs of the honest client $C \in \hat{G}$, client D, Aud, and the adversary in the real and ideal models are computationally indistinguishable.

Case 2: Corrupt dealer D. In the real execution, the dealer's view is defined as follows:

$$\mathsf{View}_D^{\texttt{ANE}}\Big(S^{(D)}, (S^{(1)}, ..., S^{(m)})\Big) =$$

$$\{S^{(D)}, adr_{sc}, S_{\min}\cdot\ddot{v}, 2\cdot\ddot{d}', r_D, \mathsf{View}_D^{\texttt{JUS}}, \mathsf{View}_D^{\texttt{CT}}, (com_{1,j}, \bar{e}_1, q_1, h_1)..., (com_{sz,j'}, \bar{e}_{sz}, q_{sz}, h_{sz}), g, \hat{x} := (mk, z'), S_\cap\}$$

where $\mathsf{View}_D^{\texttt{CT}}$ and $\mathsf{View}_D^{\texttt{VOPR}}$ refer to D's real-model view during the execution of `CT` and `VOPR` respectively. Also, r_D is the outcome of internal random coins of D, adr_{sc} is the address of contract, $\mathcal{SC}_{\texttt{ANE}}$, $(j, ..., j') \in \{1, ..., h\}$, $z' = \texttt{PRF}(\bar{mk}, 0)$, $sz = |S_\cap|$, and h_i is a Merkle tree proof asserting that $com_{i,j}$ is a leaf node of a Merkle tree with root node g. The simulator $\mathsf{Sim}_D^{\texttt{ANE}}$, which receives all parties' input sets, works as follows.

1. generates an empty view. It appends to the view, the input set $S^{(D)}$. It constructs and deploys a smart contract. It appends the contract's address, adr_{sc}, to the view.
2. appends to the view integer $S_{min} \cdot \ddot{v}$, and $2 \cdot \ddot{d}'$. Also, it appends uniformly random coins r'_D to the view.
3. extracts the simulation of `JUS` from `JUS`'s simulator for client D. Let $\mathsf{Sim}_D^{\texttt{JUS}}$ be the simulation, that also includes a random key mk. It appends $\mathsf{Sim}_D^{\texttt{JUS}}$ to the view.
4. extracts the simulation of `CT` from `CT`'s simulator, yielding the simulation $\mathsf{Sim}_D^{\texttt{CT}}$ that includes its output mk'. It appends $\mathsf{Sim}_D^{\texttt{CT}}$ to the view.
5. encrypts each element $s_{i,j}$ in the intersection set $S_\cap$ as $e_{i,j} = \texttt{PRP}(mk', s_{i,j})$ and then encodes the result as $\bar{e}_{i,j} = e_{i,j}||\texttt{H}(e_{i,j})$. It commits to each encoded value as $com_{i,j} = \texttt{Com}(\bar{e}_{i,j}, q_{i,j})$, where j is the index of the bin to which $\bar{e}_{i,j}$ belongs and $q_{i,j}$ is a random value.
6. It constructs $(com'_{1,1}, ..., com'_{d,h})$ where each $com'_{i,j}$ is a value picked uniformly at random from the commitment scheme output range. For every j-th bin, it sets each $com'_{i',j}$ to unique $com_{i,j}$ if value $com_{i,j}$ for j-th bin has been generated in step 5. Otherwise, the original values of $com'_{i',j}$ remains unchanged.
7. constructs a Merkle tree on top of the values $(com'_{1,1}, ..., com'_{d,h})$ generated in step 6. Let g be the resulting tree's root.
8. for each element $s_{i,j}$ in the intersection, it constructs $(com'_{i',j}, \bar{e}_{i,j}, q_{i,j}, h_{i,j})$, where $com'_{i',j}$ is the commitment of $\bar{e}_{i,j}$, $com'_{i',j} \in com$, $(\bar{e}_{i,j}\ q_{i,j})$ is the commitment's opening generated in step 5, and $h_{i,j}$ is a Merkle tree proof asserting that $com'_{i',j}$ is a leaf of a Merkle tree with root g. It appends all $(com'_{i',j}, \bar{e}_{i,j}, q_{i,j}, h_{i,j})$ along with g to the view.
9. generates a commitment to mk as $com_{mk} = \texttt{Com}(mk, z')$ where $z' = \texttt{PRF}(mk, 0)$. It appends (mk, z') along with $S_\cap$ to the view.

Now, we will discuss why the two views are computationally indistinguishable. D's input $S^{(D)}$ is identical in both models, so they have identical distributions. The contract's address has the same distribution in both models. The same holds for the integers $S_{min} \cdot \ddot{v}$ and $2 \cdot \ddot{d}'$. Also, because the real-model semi-honest adversary samples its randomness according to the protocol's description, the random coins in both models (i.e., r_D and r'_D) have identical distribution. Due to the security of $\mathtt{JUS}$, $\mathsf{View}_D^{\mathtt{JUS}}$ and $\mathsf{Sim}_D^{\mathtt{JUS}}$ have identical distribution, so do $\mathsf{View}_D^{\mathtt{CT}}$ and $\mathsf{Sim}_D^{\mathtt{CT}}$ due to the security of $\mathtt{CT}$. The intersection elements in both models have identical distributions and the encryption scheme is schematically secure. Therefore, each intersection element's representation (i.e., $\bar{e}_i$ in the real model and $\bar{e}_{i,j}$ in the deal model) are computationally indistinguishable. Each q_i in the real model and $q_{i,j}$ in the ideal model have identical distributions as they have been picked uniformly at random. Each commitment $com_{i,j}$ in the real model is computationally indistinguishable from commitment $com'_{i',j}$ in the ideal model.

In the real model, each Merkle tree proof h_i contains two leaf nodes (along with intermediary nodes that are the hash values of a subset of leaf nodes) that are themselves the commitment values. Also, for each h_i, only one of the leaf node's openings (that contains an element in the intersection) is seen by D. The same holds in the ideal model, with the difference that for each Merkle tree proof $h_{i,j}$ the leaf node whose opening is not provided is a random value, instead of an actual commitment. However, due to the hiding property of the commitment scheme, in the real and ideal models, these two leaf nodes (whose openings are not provided) and accordingly the two proofs are computationally indistinguishable. In both models, values mk and z' are random values, so they have identical distributions. Furthermore, the intersection $S_\cap$ is identical in both models. Thus, we conclude that the two views are computationally indistinguishable.

Case 3: Corrupt auditor. In this case, the real-model view of the auditor is defined as

$$\mathsf{View}_{Aud}^{\mathtt{ANE}}\Big(\perp, S^{(D)}, (S^{(1)}, ..., S^{(m)})\Big) =$$

$$\{\mathsf{View}_{Aud}^{\mathtt{JUS}}, adr_{sc}, S_{min} \cdot \ddot{v}, 2 \cdot \ddot{d}', (com_{1,j}, \bar{e}_1, q_1, h_1) ..., (com_{sz,j'}, \bar{e}_{sz}, q_{sz}, h_{sz}), g, \hat{x} := (mk, z')\}$$

Due to the security of $\mathtt{JUS}$, Aud's view $\mathsf{View}_{Aud}^{\mathtt{JUS}}$ during the execution of $\mathtt{JUS}$ can be easily simulated. As we have shown in Case 2, for the remaining transcript, its real-model view can be simulated too. However, there is a difference between Case 3 and Case 2; namely, in the former case, Aud does not have the $\mathtt{PRP}$'s key mk' used to encrypt each set element. However, due to the security of $\mathtt{PRP}$, it cannot distinguish each encrypted encoded element from a uniformly random element and cannot distinguish $\mathtt{PRP}(mk', .)$ from a uniform permutation. Therefore, each value $\bar{e}_j$ in the real model has identical distribution to each value $\bar{e}_{i,j}$ (as defined in Case 2) in the ideal model, as both are the outputs of $\mathtt{PRP}$.

Case 4: Corrupt public. In the real model, the view of the public (i.e., non-participants of the protocol) is defined as below:

$$\mathsf{View}_{Pub}^{\mathtt{ANE}}\Big(\perp, S^{(D)}, (S^{(A_1)}, ..., S^{(A_m)})\Big) =$$

$$\{\mathsf{View}_{Pub}^{\mathtt{JUS}}, adr_{sc}, S_{min} \cdot \ddot{v}, 2 \cdot \ddot{d}', (com_{1,j}, \bar{e}_1, q_1, h_1) ..., (com_{sz,j'}, \bar{e}_{sz}, q_{sz}, h_{sz}), g, \hat{x} := (mk, z')\}$$

Due to the security of JUS, the public's view $\mathsf{View}^{\mathtt{JUS}}_{Pub}$ during JUS's execution can be simulated in the same way which is done in Case 4, in Section I. The rest of the public's view overlaps with Aud's view in Case 3, excluding $\mathsf{View}^{\mathtt{JUS}}_{Aud}$. Therefore, we can use the argument provided in Case 3 to show that the rest of the public's view can be simulated. We conclude that the public's real and ideal views are computationally indistinguishable. □

L Further Discussion on Anesidora

There is a simpler but costlier approach to finding the intersection without involving the extractors; that is the smart contract finds the (encoded) elements of the intersection and distributes the parties' deposit according to the number of elements it finds. This approach is simpler, as we do not need the involvement of (i) the extractors and (ii) the three counter collusion contracts. Nevertheless, it is costlier, because the contract itself needs to factorise the unblinded resulting polynomial and find the roots, which would cost it $O(d^2)$ for each bin, where d is the size of each bin. Our proposed approach however moves such a computation off-chain, leading to a lower monetary computation cost.

The reason each client uses the hash-based padding to encode each encrypted element e_i as $\bar{e}_i = e_i||\mathtt{H}(e_i)$ is to allow the auditor in the counter collusion contracts to find the error-free intersection, without having to access to one of the original (encrypted) sets.

Compared to JUS, there is a minor difference in finding the result in ANE. Specifically, because in ANE each set element s_i is encoded as (i) $e_i = \mathtt{PRP}(mk', s_i)$ and then (ii) $\bar{e}_i = e_i||\mathtt{H}(e_i)$ by a client, then when the client wants to find the intersection it needs to first regenerate $\bar{e}_i$ as above and then treat it as a set element to check if $\phi'(\bar{e}_i) = 0$, in step 12(b)iii of JUS.

In ANE, each extractor uses double-layered commitments (i.e., it first commits to the encryption of each element and then constructs a Merkle tree on top of all commitments) for efficiency and privacy purposes. Constructing a Merkle tree on top of the commitments allows the extractor to store only a single value in $\mathcal{SC}_{\mathtt{ANE}}$ would impose a much lower storage cost compared to the case where it would store all commitments in $\mathcal{SC}_{\mathtt{ANE}}$. Also, committing to the elements' encryption allows it to hide from other clients the encryption of those elements that are not in the intersection. Recall that encrypting each element is not sufficient to protect one client's elements from the rest of the clients, as they all know the decryption key.

To increase their reward, malicious clients may be tempted to insert "garbage" elements into their sets with the hope that those garbage elements appear in the result and accordingly they receive a higher reward. However, they would not succeed as long as there exists a semi-honest client (e.g., dealer D) which uses actual set elements. In this case, by the set intersection definition, those garbage elements will not appear in the intersection.

In ANE, for the sake of simplicity, we let each party receive a fixed reward, i.e., $\ddot{l}$, for every element it contributes to the intersection. However, it is possible to

make the process more flexible/generic. For instance, we could define a Reward Function RF that takes $\ddot{l}$, an (encoded) set element e_i in the intersection, its distribution/value val_{e_i}, and output a reward rew_{e_i} that each party should receive for contributing that element to the intersection, i.e., $RF(\ddot{l}, e_i, val_{e_i}) \rightarrow rew_{e_i}$.

M Counter Collusion Contracts

In this section, we present Prisoner's Contract ($\mathcal{SC}_{\text{PC}}$), Colluder's Contract ($\mathcal{SC}_{\text{CC}}$), Traitor's Contract ($\mathcal{SC}_{\text{TC}}$) originally proposed by Dong *et al.* [16]. As we previously stated, we have slightly adjusted the contracts. We will highlight the adjustments in blue. For the sake of completeness, below we restate the parameters used in these contracts and their relation.

- $\ddot{b}$: the bribe paid by the ringleader of the collusion to the other server in the collusion agreement, in $\mathcal{SC}_{\text{CC}}$.
- $\ddot{c}$: a server's cost for computing the task.
- $\ddot{ch}$: the fee paid to invoke an auditor for recomputing a task and resolving disputes.
- $\ddot{d}$: the deposit a server needs to pay to be eligible for getting the job.
- $\ddot{t}$: the deposit the colluding parties need to pay in the collusion agreement, in $\mathcal{SC}_{\text{CC}}$.
- $\ddot{w}$: the amount that a server receives for completing the task.
- $\ddot{w} \geq \ddot{c}$: the server would not accept underpaid jobs.
- $\ddot{ch} > 2\ddot{w}$: If it does not hold, then there would be no need to use the servers and the auditor would do the computation.
- (pk, sk): an asymmetric-key encryption's public-private key pair belonging to the auditor.

The following relations need to hold when setting the contracts in order for the desirable equilibria to hold: (i) $\ddot{d} > \ddot{c}+\ddot{ch}$, (ii) $\ddot{b} < \ddot{c}$, and (iii) $\ddot{t} < \ddot{w}-\ddot{c}+2\ddot{d}-\ddot{ch}-\ddot{b}$.

M.1 Prisoner's Contract ($\mathcal{SC}_{\text{PC}}$)

$\mathcal{SC}_{\text{PC}}$ has been designed for outsourcing a certain computation. It is signed by a client who delegates the computation and two other parties (or servers) who perform the computation. This contract tries to incentivize correct computation by using the following idea. It requires each server to pay a deposit before the computation is delegated. If a server behaves honestly, then it can withdraw its deposit. However, if a server cheats (and is detected), its deposit is transferred to the client. When one of the servers is honest and the other one cheats, the honest server receives a reward. This reward is taken from the deposit of the cheating server. Hence, the goal of $\mathcal{SC}_{\text{PC}}$ is to create a Prisoner's dilemma between the two servers in the following sense. Although the servers may collude with each other (to cut costs and provide identical but incorrect computation results) which leads to a higher payoff than both behaving honestly, there is an even higher

payoff if one of the servers manages to persuade the other server to collude and provide an incorrect result while itself provides a correct result. In this setting, each server knows that collusion is not stable as its counterparty will always try to deviate from the collusion to increase its payoff. If a server tries to convince its counterparty (without a credible and enforceable promise), then the latter party will consider it as a trap; consequently, collusion will not occur. Below, we restate the contract.

1. The contract is signed by three parties; namely, client D and two other parties E_1 and E_2. A third-party auditor will resolve any dispute between D and the servers. The address of another contract, called $\mathcal{SC}_{\texttt{ANE}}$, is hardcoded in this contract.
2. The servers agree to run computation f on input x, both of which have been provided by D.
3. The parties agree on three deadlines (T_1, T_2, T_3), where $T_{i+1} > T_i$.
4. D agrees to pay $\ddot{w}$ to each server for the correct and on-time computation. Therefore, D deposits $2{\cdot}\ddot{w}$ amount in the contract. This deposit is transferred from $\mathcal{SC}_{\texttt{ANE}}$ to this contract.
5. Each server deposits $\ddot{d}' = \ddot{d} + \ddot{X}$ amount in the contract.
6. The servers must pay the deposit before T_1. If a server fails to meet the deadline, then the contract would refund the parties' deposit (if any) and terminates.
7. The servers must deliver the computation's result before T_2.
8. The following steps are taken when (i) both servers provided the computation's result or (ii) deadline T_2 elapsed.
   (a) if both servers failed to deliver the computation's result, then the contract transfers their deposits to $\mathcal{SC}_{\texttt{ANE}}$.
   (b) if both servers delivered the result, and the results are equal, then (after verifying the results) this contract must pay the agreed amount $\ddot{w}$ and refund the deposit $\ddot{d}'$ to each server.
   (c) otherwise, D raises a dispute with the auditor.
9. When a dispute is raised, the auditor (which is independent of Aud in JUS) re-generates the computation's result, by using algorithm `resComp`(.) described shortly in Appendix M.1. Let y_t, y_1, and y_2 be the result computed by the auditor, E_1, and E_2 respectively. The auditor uses the following role to identify the cheating party.
   - if E_i failed to deliver the result (i.e., y_i is null), then it has cheated.
   - if a result y_i has been delivered before the deadline and $y_i \neq y_t$, then E_i has cheated.

   The auditor sends its verdict to $\mathcal{SC}_{\text{PC}}$.
10. Given the auditor's decision, the dispute is settled according to the following rules.
    - if none of the servers cheated, then this contract transfers to each server (i) $\ddot{w}$ amount for performing the computation and (ii) its deposit, i.e., $\ddot{d}'$ amount. The client also pays the auditor $\ddot{ch}$ amount.

- if both servers cheated, then this contract (i) pays the auditor the total amount of $\ddot{ch}$, taken from the servers' deposit, and (ii) transfers to $\mathcal{SC}_{\texttt{ANE}}$ the rest of the deposit, i.e., $2 \cdot \ddot{d'} - \ddot{ch}$ amount.
- if one of the servers cheated, then this contract (i) pays the auditor the total amount of $\ddot{ch}$, taken from the misbehaving server's deposit, (ii) transfers the honest server's deposit (i.e., $\ddot{d'}$ amount) back to this server, (iii) transfers to the honest server $\ddot{w} + \ddot{d} - \ddot{ch}$ amount (which covers its computation cost and the reward), and (iv) transfers to $\mathcal{SC}_{\texttt{ANE}}$ the rest of the misbehaving server's deposit, i.e., $\ddot{X}$ amount. The cheating server receives nothing.

11. After deadline T_3, if D has neither paid nor raised a dispute, then this contract pays $\ddot{w}$ to each server which delivered a result before deadline T_2 and refunds each server its deposit, i.e., $\ddot{d'}$ amount. Any deposit left after that will be transferred to $\mathcal{SC}_{\texttt{ANE}}$.

Now, we explain why we have made the above changes to the original $\mathcal{SC}_{\text{PC}}$ of Dong *et al.* [16]. In the original $\mathcal{SC}_{\text{PC}}$ (a) the client does not deposit any amount in this contract; instead, it directly sends its coins to a server (and auditor) according to the auditor's decision, (b) the computation correctness is determined only within this contract (with the involvement of the auditor if required), and (c) the auditor simply re-generates the computation's result given the computation's plaintext inputs. Nevertheless, in `ANE`, (1) *all clients* need to deposit a certain amount in $\mathcal{SC}_{\texttt{ANE}}$ and only the contracts must transfer the parties' deposit, (2) $\mathcal{SC}_{\texttt{ANE}}$ also needs to verify a part of the computation's correctness without the involvement of the auditor and accordingly distribute the parties deposit based on the verification's outcome, and (3) the auditor must be able to re-generate the computation's result without being able to learn the computation's plaintext input, i.e., elements of the set. Therefore, we have included the address of $\mathcal{SC}_{\texttt{ANE}}$ in $\mathcal{SC}_{\text{PC}}$ to let the parties' deposit move between the two contracts (if necessary) and allowed $\mathcal{SC}_{\text{PC}}$ to distribute the parties' deposit; thus, the requirements in points (1) and (2) are met. To meet the requirement in point (3) above, we have included a new algorithm, called `resComp`(.), in $\mathcal{SC}_{\text{PC}}$. Shortly, we will provide more detail about this algorithm. Moreover, to make this contract compatible with `ANE`, we increased the amount of each server's deposit by $\ddot{X}$. Nevertheless, this adjustment does not change the logic behind $\mathcal{SC}_{\text{PC}}$'s design and its analysis.

Auditor's Result-Computation Algorithm. In this work, we use $\mathcal{SC}_{\text{PC}}$ to delegate the computation of intersection cardinality to two extractor clients, a.k.a. servers in the original $\mathcal{SC}_{\text{PC}}$. In this setting, the contract's auditor is invoked when an inconsistency is detected in step 13 of `ANE`. For the auditor to recompute the intersection cardinality, we have designed `resComp`(.) algorithm. The auditor uses this algorithm for every bin's index $indx$, where $1 \leq indx \leq h$ and h is the hash table's length. We present this algorithm in Figure 7. The auditor collects the inputs of this algorithm as follows: (a) reads random polynomial $\boldsymbol{\zeta}$, and blinded polynomial $\boldsymbol{\phi}$ from contract $\mathcal{SC}_{\texttt{JUS}}$, (b) reads the ciphertext if secret key

mk from $\mathcal{SC}_{\texttt{ANE}}$, and (c) fetches public parameters $(des_{\texttt{H}}, h)$ from the hash table's public description.

Note that in the original $\mathcal{SC}_{\text{PC}}$ of Dong *et al.* [16], the auditor is assumed to be fully trusted. However, in this work, we have relaxed such an assumption. We have designed `ANE` and `resComp`(.) in such a way that even a semi-honest auditor cannot learn anything about the actual elements of the sets, as they have been encrypted under a key unknown to the auditor.

$\texttt{resComp}(\boldsymbol{\zeta}, \boldsymbol{\phi}, sk, ct_{mk}, indx, des_{\texttt{H}}) \rightarrow R$

- *Input.* $\boldsymbol{\zeta}$: a random polynomial of degree 1, $\boldsymbol{\phi}$: a blinded polynomial of the form $\boldsymbol{\zeta}\cdot(\boldsymbol{\epsilon}+\boldsymbol{\gamma}')$ where $\boldsymbol{\epsilon}$ and $\boldsymbol{\gamma}'$ are arbitrary and pseudorandom polynomials respectively, $deg(\boldsymbol{\phi})-1 = deg(\boldsymbol{\gamma}')$, sk: the auditor's secret key, ct_{mk}: ciphertext of mk which is a key of `PRF`, $indx$: an input of `PRF`, and $des_{\texttt{H}}$: a description of hash function `H`.
- *Output.* R: a set containing valid roots of unblinded $\boldsymbol{\phi}$.

1. decrypts the ciphertext ct_{mk} under key sk. Let mk be the result.
2. unblinds polynomial $\boldsymbol{\phi}$, as follows:
   (a) re-generates pseudorandom polynomial $\boldsymbol{\gamma}'$ using key mk. Specifically, it uses mk to derive a key: $k = \texttt{PRF}(mk, indx)$. Then, it uses the derived key to generate $3d+1$ pseudorandom coefficients, i.e., $\forall j, 0 \leq j \leq deg(\boldsymbol{\phi}) - 1 : g_j = \texttt{PRF}(k, j)$. Next, it uses these coefficients to construct polynomial $\boldsymbol{\gamma}'$, i.e., $\boldsymbol{\gamma}' = \sum_{j=0}^{deg(\boldsymbol{\phi})-1} g_j \cdot x^j$.
   (b) removes the blinding factor from $\boldsymbol{\phi}$. Specifically, it computes polynomial $\boldsymbol{\phi}'$ of the following form $\boldsymbol{\phi}' = \boldsymbol{\phi} - \boldsymbol{\zeta} \cdot \boldsymbol{\gamma}'$.
3. extracts roots of polynomial $\boldsymbol{\phi}'$.
4. finds valid roots, by (i) parsing each root $\bar{e}$ as (e_1, e_2) with the assistance of $des_{\texttt{H}}$ and (ii) checking if $e_2 = \texttt{H}(e_1)$. It considers a root valid, if this equation holds.
5. returns set R containing all valid roots.

Fig. 7: Auditor's result computation, `resComp`(.), algorithm

M.2 Colluder's Contract ($\mathcal{SC}_{\text{CC}}$)

Recall that $\mathcal{SC}_{\text{PC}}$ aimed at creating a dilemma between the two servers. However, this dilemma can be addressed if they can make an enforceable promise. This enforceable promise can be another smart contract, called Colluder's Contract ($\mathcal{SC}_{\text{CC}}$). This contract imposes additional rules that ultimately would affect the parties' payoffs and would make collusion the most profitable strategy for the colluding parties. In $\mathcal{SC}_{\text{CC}}$, the party who initiates the collusion would pay its counterparty a certain amount (or bribe) if both follow the collusion and provide an incorrect result of the computation to $\mathcal{SC}_{\text{PC}}$. Note, $\mathcal{SC}_{\text{CC}}$ requires both servers to send a fixed amount of deposit when signing the contract. The party who deviates from collusion will be punished by losing the deposit. Below, we restate $\mathcal{SC}_{\text{CC}}$.

1. The contract is signed between the server who initiates the collusion, called ringleader (LDR) and the other server called follower (FLR).
2. The two agree on providing to $\mathcal{SC}_{\text{PC}}$ a different result res' than a correct computation of f on x would yield, i.e., $res' \neq f(x)$. Parameter res' is recorded in this contract.
3. LDR and FLR deposit $\ddot{t} + \ddot{b}$ and $\ddot{t}$ amounts in this contract respectively.
4. The above deposit must be paid before the result delivery deadline in $\mathcal{SC}_{\text{PC}}$, i.e., before deadline T_2. If this condition is not met, the parties' deposit in this contract is refunded and this contract is terminated.
5. When $\mathcal{SC}_{\text{PC}}$ is finalised (i.e., all the results have been provided), the following steps are taken.
   (a) both follow the collusion: if both LDR and FLR provided res' to $\mathcal{SC}_{\text{PC}}$, then $\ddot{t}$ and $\ddot{t} + \ddot{b}$ amounts are delivered to LDR and FLR respectively. Therefore, FLR receives its deposit plus the bribe $\ddot{b}$.
   (b) only FLR deviates from the collusion: if LDR and FLR provide res' and $res'' \neq res'$ to $\mathcal{SC}_{\text{PC}}$ respectively, then $2 \cdot \ddot{t} + \ddot{b}$ amount is transferred to LDR while nothing is sent to FLR.
   (c) only LDR deviates from the collusion: if LDR and FLR provide $res'' \neq res'$ and res' to $\mathcal{SC}_{\text{PC}}$ respectively, then $2 \cdot \ddot{t} + \ddot{b}$ amount is sent to FLR while nothing is transferred to LDR.
   (d) both deviate from the collusion: if LDR and FLR deviate and provide any result other than res' to $\mathcal{SC}_{\text{PC}}$, then $2 \cdot \ddot{t} + \ddot{b}$ amount is sent to LDR and $\ddot{t}$ amount is sent to FLR.

We highlight that the amount of bribe a rational LDR is willing to pay is less than its computation cost (i.e., $\ddot{b} < \ddot{c}$); otherwise, such collusion would not bring a higher payoff. We refer readers to [16] for further discussion.

M.3 Traitor's Contract ($\mathcal{SC}_{\text{TC}}$)

$\mathcal{SC}_{\text{TC}}$ incentivises a colluding server (who has had signed $\mathcal{SC}_{\text{CC}}$) to betray its counterparty and report the collusion without being penalised by $\mathcal{SC}_{\text{PC}}$. The Traitor's contract promises that the reporting server will not be punished by $\mathcal{SC}_{\text{PC}}$ which makes it safe for the reporting server to follow the collusion strategy (of $\mathcal{SC}_{\text{CC}}$), and get away from the punishment imposed by $\mathcal{SC}_{\text{PC}}$. Below, we restate $\mathcal{SC}_{\text{TC}}$.

1. This contract is signed among D and the traitor server (TRA) who reports the collusion. This contract is signed only if the parties have already signed $\mathcal{SC}_{\text{PC}}$.
2. D signs this contract only with the first server who reports the collusion.
3. The traitor TRA must also provide to this contract the result of the computation, i.e., $f(x)$. The result provided in this contract could be different than the one provided in $\mathcal{SC}_{\text{PC}}$, e.g., when TRA has to follow $\mathcal{SC}_{\text{CC}}$ and provide an incorrect result to $\mathcal{SC}_{\text{PC}}$.

4. D needs to pay $\ddot{w} + \ddot{d'} + \ddot{d} - \ddot{ch}$ amount to this contract. This amount equals the maximum amount TRA could lose in $\mathcal{SC}_{\text{PC}}$ plus the reward. This deposit will be transferred via $\mathcal{SC}_{\texttt{ANE}}$ to this contract. TRA must deposit in this contract $\ddot{ch}$ amount to cover the cost of resolving a potential dispute.
5. This contract should be signed before the deadline T_2 for the delivery of the computation result in $\mathcal{SC}_{\text{PC}}$. If it is not signed on time, then this contract would be terminated and any deposit paid will be refunded.
6. It is required that the TRA provide the computation result to this contract before the above deadline T_2.
7. If this contract is fully signed, then during the execution of $\mathcal{SC}_{\text{PC}}$, D always raises a dispute, i.e., takes step 8c in $\mathcal{SC}_{\text{PC}}$.
8. After $\mathcal{SC}_{\text{PC}}$ is finalised (with the involvement of the auditor), the following steps are taken to pay the parties involved.
   (a) if none of the servers cheated in $\mathcal{SC}_{\text{PC}}$ (according to the auditor), then amount $\ddot{w} + \ddot{d'} + \ddot{d} - \ddot{ch}$ is refunded to $\mathcal{SC}_{\texttt{ANE}}$ and TRA's deposit (i.e., $\ddot{ch}$ amount) is transferred to D. Nothing is paid to TRA.
   (b) if in $\mathcal{SC}_{\text{PC}}$, the other server did not cheat and TRA cheated; however, TRA provided a correct result in this contract, then $\ddot{d'} + \ddot{d} - \ddot{ch}$ amount is transferred to $\mathcal{SC}_{\texttt{ANE}}$. Also, TRA gets its deposit back (i.e., $\ddot{ch}$ amount) plus $\ddot{w}$ amounts for providing a correct result to this contract.
   (c) if in $\mathcal{SC}_{\text{PC}}$, both servers cheated; however, TRA delivered a correct computation result to this contract, then TRA gets its deposit back (i.e., $\ddot{ch}$ amount), it also receives $\ddot{w} + \ddot{d'} + \ddot{d} - \ddot{ch}$ amount.
   (d) otherwise, $\ddot{w} + \ddot{d'} + \ddot{d} - \ddot{ch}$ and $\ddot{ch}$ amounts are transferred to $\mathcal{SC}_{\texttt{ANE}}$ and TRA respectively.
9. If TRA provided a result to this contract, and deadline T_3 (in $\mathcal{SC}_{\text{PC}}$) has passed, then all deposits, if any left, will be transferred to TRA.

TRA must take the following three steps to report collusion: (i) it waits until $\mathcal{SC}_{\text{CC}}$ is signed by the other server, (ii) it reports the collusion to D before signing $\mathcal{SC}_{\text{CC}}$, and (iii) it signs $\mathcal{SC}_{\text{CC}}$ only after it signed $\mathcal{SC}_{\text{TC}}$ with D. Note, the original analysis of $\mathcal{SC}_{\text{TC}}$ does not require $\mathcal{SC}_{\text{TC}}$ to remain secret. Therefore, in our smart PSI, parties TRA and D can sign this contract and then store its address in $\mathcal{SC}_{\texttt{ANE}}$. Alternatively, to keep this contract confidential, D can deploy a template $\mathcal{SC}_{\text{TC}}$ to the blockchain and store the commitment of the contract's address (instead of the plaintext address) in $\mathcal{SC}_{\texttt{ANE}}$. When a traitor (TRA) wants to report collusion, it signs the deployed $\mathcal{SC}_{\text{TC}}$ with D which provides the commitment opening to TRA. In this case, at the time when the deposit is distributed, either D or TRA provides the opening of the commitment to $\mathcal{SC}_{\texttt{ANE}}$ which checks whether it matches the commitment. If the check passes, then it distributes the deposit as before.

N Proof of Theorem 9

Below, we restate the proof of Theorem 9, taken from [3].

Proof. Let $P = \{p_1, ..., p_t\}$ and $Q = \{q_1, ..., q_{t'}\}$ be the roots of polynomials $\mathbf{p}$ and $\mathbf{q}$ respectively. By the Polynomial Remainder Theorem, polynomials $\mathbf{p}$ and $\mathbf{q}$ can be written as $\mathbf{p}(x) = \mathbf{g}(x) \cdot \prod_{i=1}^{t}(x - p_i)$ and $\mathbf{q}(x) = \mathbf{g}'(x) \cdot \prod_{i=1}^{t'}(x - q_i)$ respectively, where $\mathbf{g}(x)$ has degree $d - t$ and $\mathbf{g}'(x)$ has degree $d' - t'$. Let the product of the two polynomials be $\mathbf{r}(x) = \mathbf{p}(x) \cdot \mathbf{q}(x)$. For every $p_i \in P$, it holds that $\mathbf{r}(p_i) = 0$. Because (a) there exists no non-constant polynomial in $\mathbb{F}_p[X]$ that has a multiplicative inverse (so it could cancel out factor $(x - p_i)$ of $\mathbf{p}(x)$) and (b) p_i is a root of $\mathbf{p}(x)$. The same argument can be used to show for every $q_i \in Q$, it holds that $\mathbf{r}(q_i) = 0$. Thus, $\mathbf{r}(x)$ preserves roots of both $\mathbf{p}$ and $\mathbf{q}$. Moreover, $\mathbf{r}$ does not have any other roots (than P and Q). In particular, if $\mathbf{r}(\alpha) = 0$, then $\mathbf{p}(\alpha) \cdot \mathbf{q}(\alpha) = 0$. Since there is no non-trivial divisors of zero in $\mathbb{F}_p[X]$ (as it is an integral domain), it must hold that either $\mathbf{p}(\alpha) = 0$ or $\mathbf{q}(\alpha) = 0$. Hence, $\alpha \in P$ or $\alpha \in Q$. □