Smarter Fair Multi-client Private Set Intersection Protocols

Aydin Abadi*

University of Edinburgh

1 Introduction

In this paper, we provide the first efficient *multi-party fair* PSI protocol (F-PSI). It allows either all clients to get the result or if the protocol aborts in an unfair manner (where only dishonest parties learn the result), then honest parties will be financially compensated. The protocol is mainly based on symmetric key primitives and a smart contract. This is the first time a smart contract is used in a PSI protocol.

Moreover, we provide another PSI protocol (Smart-PSI) that allows a buyer who initiates the PSI computation (and interested in the result) to pay other parties in a fair manner, where the amount each party receives is proportional to the number of elements the buyer learns about their inputs. Smart-PSI is mainly based on symmetric key primitives, modified F-PSI as well as a game theory based approach. The latter approach is leveraged to create tension, betrayal and distrust between the clients (who want to collude with the buyer to increase their shares) and buyer (who wants to pay less). This is the first time a game theory based approach utilised in a PSI protocol.

Also, we identify and explain several attacks that can be mounted on the state of the art, i.e. all three (two-party, multi-party, and threshold) PSI protocols very recently proposed by Gosh *et al.* in [4].

2 Notations and Monetary Variables

Let $\mathbf{P} : \{A_1, ..., A_m, D\}$ be set of all clients that engage in the protocol and $\mathbf{E} \subseteq \mathbf{P}$ be set of extractor clients, where $|\mathbf{E}| = 2$. We denote the client interested in the result, i.e. the buyer, by A_m. Furthermore, we denote the intersection size by $|S_\cap|$, and the size of smallest and largest set by $S_{\min}$ and $S_{\max}$ respectively. By $s^{(I)}$ we mean value s belongs to client I.

- y: the amount each party deposits in the fair smart PSI.
- v: the price the buyer pays to learn an element of the sets.
- l: the amount a client earns for contributing an element that appears in the intersection
- r: the reward an extractor client earns for extracting an element of the intersection and submitting it to the contract.

Therefore, we have the following relation: $v = m \cdot l + 2 \cdot r$

* aydin.abadi@ed.ac.uk

3 Preliminaries

3.1 Smart Contracts

3.2 Counter collusion smart contracts

In order to allow a client to efficiently delegate an arbitrary computation to a couple of cloud servers, Dong *et al.* in [2] propose two kinds of smart contracts; namely, Prisoner's and Traitor's contracts.

3.3 Pseudorandom Function and Permutation

3.4 Random Extraction Beacon

3.5 Commitment Scheme

3.6 Hash Tables

3.7 Merkel Tree

3.8 Polynomial Representation of Sets

3.9 Horner's Method

3.10 Oblivious Linear Function Evaluation

4 Multi-party Fair PSI (F-PSI)

In this section, we provide our multi-party fair PSI protocol that allows either all clients to get the result or if the protocol aborts in an unfair manner (where only dishonest parties learn the result), then honest parties will be financially compensated. We first provide the security model and assumptions used in F-PSI. After that, we provide our three subprotocols: `VOPR`, `ZSPA` and `Arbiter`, that will also be utilised by F-PSI. Next, we give an overview of F-PSI followed by F-PSI protocol description in detail.

4.1 Threat Model

In this section, we provide the security model of our protocol. There are two kinds of party involved in the protocol. Namely, (1) a set of clients $\{A_1, ..., A_m\}$ potentially *malicious* (i.e. active adversaries) and all may collude with each other, and (2) a non-colluding dealer: client D, potentially semi-honest (i.e. a passive adversary). In this work, we consider 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.

4.2 Verifiable Oblivious Polynomial Randomisation (VOPR)

In this section, we provide VOPR, an efficient protocol that allows a sender with input polynomials α and ψ and a receiver with input polynomial β to compute: $\theta = \psi \cdot \beta + \alpha$, such that the receiver learns only θ while the sender learns nothing, where α is always a random polynomial.

In this paper, VOPR is used for two main reasons: (a) to rerandomise each party's polynomial (representing its set) by obliviously multiplying it by (one of) the polynomial(s) its counterparty sends to VOPR, (b) to impose a MAC-like structure to the randomised polynomial that will allow later on to check the integrity of the result. In the active security model in VOPR, it is required to ensure: (1) if the receiver provides a zero polynomial (i.e. $\beta = 0$) to VOPR then it will learn nothing about the other parties input and (2) the parties follows the protocol in computing the result. To satisfy requirement (1) above, we use OLE^+ protocol proposed in [4]. To meet requirement (2), we use the following idea proposed in [1] (and used in [4] as well): the sender picks and sends a random value: z to the receiver who computes $\theta(z)$ and $\beta(z)$ and sends them to the sender. The sender sends $\psi(z)$ and $\alpha(z)$ to the receiver. Then each of them check: $\theta(z) = \psi(z) \cdot \beta(z) + \alpha(z)$. They accept if the equation holds.

We highlight that even though there are some similarities between OPA proposed in [4] and our VOPR, they have significant differences too. Specifically, as shown in Section 6, OPA suffers from serious security and technical issues, that are addressed in VOPR, e.g. by using input polynomials in coefficient forms, using input polynomials of specific structures (i.e. α and β), and using a randomised beacon (or a coin tossing protocol). We present VOPR protocol in Fig. 1.

4.3 Zero-sum Pseudorandom Values Agreement (ZSPA)

In this section, we provide ZSPA: a protocol that allows m parties (potentially malicious) to efficiently agree on (b vectors each of which having) 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 to agree on two keys: k_1, k_2. Then, one of the parties generates $m-1$ pseudorandom values z_j (where $1 \leq j \leq m-1$) using key k_1 and sets the last value as additive inverse of the sum of the values generated, i.e. $z_m = -\sum_{j=1}^{m-1} z_j$. Next, it commits to each value, where it uses k_2 to generate the randomness of each commitment. Then, it constructs a Merkel tree on top of the commitments and stores only the root of the tree and the hash value of the keys (so in total three values) in the smart contract. Then, each party (using the keys) locally checks if the values and commitments have been constructed correctly; if so, each sends an "approved" message to the contract.

Informally, there are three main security requirements that ZSPA must meet: (a) privacy, (b) non-refutability, and (c) indistinguishability. Privacy here means given the state of the contract, 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. Furthermore, indistinguishability means that every z_j

- **Input:**
  - *Public Parameters*: upper bound on input polynomials' degree: d and d'.
  - *Sender Input*: random polynomials: $\psi = \sum_{i=0}^{d} g_i \cdot x^i$ and $\alpha = \sum_{j=0}^{d+d'} a_j \cdot x^j$, where $g_i \xleftarrow{\$} \mathbb{F}_p$. Also, each a_j has the form: $a_j = \sum_{t,k=0}^{d'} a_{t,k}$, such that $\forall t, k : t + k = j$ and $a_{t,k} \xleftarrow{\$} \mathbb{F}_p$. e.g. when $d' = 4$, we have $a_3 = a_{0,3} + a_{3,0} + a_{1,2} + a_{2,1}$
  - *Receiver Input*: polynomial $\beta = \omega \cdot \phi = \sum_{i=0}^{d'} b_i \cdot x^i$, where ω is a random polynomial, ϕ is an arbitrary polynomial, $deg(\omega) \geq deg(\phi), 0 \leq deg(\phi) \leq d$ and $deg(\omega) + deg(\phi) = d'$.
- **Output:** The receiver gets $\theta = \psi \cdot \beta + \alpha$

1. **Computation:**
   (a) Sender and receiver together for every j, $0 \leq j \leq d'$, invoke d instances of $\texttt{OLE}^+$. In particular, $\forall j, 0 \leq j \leq d'$: sender sends g_i and $\alpha_{i,j}$ while the receiver sends b_j to $\texttt{OLE}^+$ that returns: $c_{i,j} = g_i \cdot b_j + \alpha_{i,j}$ to the receiver ($\forall i, 0 \leq i \leq d$).
   (b) The receiver sums component-wise values $c_{i,j}$ that results polynomial $\theta = \psi \cdot \beta + \alpha = \sum_{j=0}^{d+d'} c_j \cdot x^j$, where each c_j has form: $c_j = \sum_{t,k=0}^{\substack{k=d' \\ t=d}} c_{t,k}$, such that $\forall t, k : t + k = j$
2. **Verification:**
   (a) Sender: uses the beacon to extract a random value from the blockchain: $z = \texttt{Beacon}(\lambda, des)$. Also, it commits to $\psi_z = \psi(z)$ and $\alpha_z = \alpha(z)$, and sends the commitments along with z to the receiver.
   (b) Receiver: verifiers if $z = \texttt{Beacon}(\lambda, des)$. If not passed, it aborts. Otherwise (if passed), it sends $\theta_z = \theta(z)$ and $\beta_z = \beta(z)$ to the sender who sends the commitments' openings to the receiver.
   (c) Sender and Receiver: check if: $\theta_z = \psi_z \cdot \beta_z + \alpha_z$. Otherwise, they abort.
   (d) Sender: checks if $\theta_z \neq 0 \ \wedge \ \beta_z \neq 0$. Otherwise, it aborts.
   (e) Receiver: ensures $\psi_z \neq 0 \ \wedge \ \alpha_z \neq 0$, and the openings matches the commitments. Otherwise, it aborts.

Fig. 1: Verifiable Oblivious Polynomial Randomization (VOPR) Protocol

($1 \leq j \leq m$) should be indistinguishable from a truly random value. In Fig. 2, we provide ZSPA that efficiently generates b vectors where each vector elements is sum to zero.

4.4 Arbiter Protocol

In this section, we provide Arbiter protocol, that allows a third party, potentially semi-honest, to pinpoint misbehaving clients. Arbiter protocol is invoked by our fair PSI protocol when the smart contract detects that the combination of the messages sent by the clients is not well-formed.

Informally, Arbiter has two security requirements: (a) misbehaving parties are always detected except with a negligible probability and (b) it leaks no information about

- *Parties:* $\{A_l, ..., A_m\}$
- *Public Parameters and Functions:* A pseudorandom function: PRF, a deployed smart contract, and total number of participants: m.
- *Output*: All parties agree on $b+1$ vectors $[z_{0,1}, ..., z_{1,m}], ..., [z_{b,1}, ..., z_{b,m}]$, of pseudorandom values, such that the sum of each vector's elements equals zero: $\sum_{j=1}^{m} z_{i,j} = 0$

1. All participants run a coin tossing protocol to agree on two keys k_1 and k_2 of PRF.
2. One of the parties:
   (a) For every i, computes m pseudorandom values: $\forall j, 1 \leq j \leq m-1 : z_{i,j} = \texttt{PRF}(k_1, i||j)$ and sets $z_{i,m} = -\sum_{j=1}^{m-1} z_{i,j}$, where $0 \leq i \leq b$
   (b) commits to every $z_{i,j}$ as follows: $\texttt{a}_{i,j} = \texttt{Com}(z_{i,j}, q_{i,j})$, where the randomness of the commitment is computed as: $q_{i,j} = \texttt{PRF}(k_2, i||j)$ and $1 \leq j \leq m$.
   (c) constructs a Merkel tree on top of the committed values: $\texttt{MT}(\texttt{a}_{0,1}, ..., \texttt{a}_{b,m}) \rightarrow g$
   (d) sends the Merkel tree's root: g, and the keys' hashes: $\texttt{H}(k_1)$ and $\texttt{H}(k_2)$, to the contract.
3. The rest of parties (given k_1, k_2) check if, all $z_{i,j}$ values, the root g and keys' hashes have been correctly generated (by redoing step 2). If passed, each party sends a singed "approved" message to the contract. Otherwise, it aborts.

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

clients set elements (including misbehaving ones) even if the adversary observes the clients messages to the PSI's smart contract (i.e. $\mathcal{SC}_{\text{F-PSI}}$). We provide Arbiter protocol in Fig. 3.

4.5 F-PSI Overview

F-PSI lies on the following very high-level idea. First, each client encodes its set elements into a polynomial. Also, all clients sign a smart contract and deposit a predefined amount. Then, one of the clients as dealer randomises the rest of clients' polynomials. Also, it imposes a structure to their polynomials. The clients also randomise the dealers polynomial. Then, all clients send their randomized polynomials to a smart contract. The dealer also sends two polynomials to the contract. The contract combines all polynomials and checks if the result still has the structure imposed by the dealer. If so, it accepts and output the result. Also, it refunds the clients deposit. Otherwise, an arbiter is invoked to pinpoint misbehaving clients and penalise it. If the contract accepts the result, then all clients can use the combined polynomial (output but the contract) to locally find the intersection. The efficiency of the protocol mainly stems from our observation (stated in Theorem 2) which leads to an efficient verification mechanism carried out by the contract.

Now we describe the protocol in more detail. All clients first sign and deploy a smart contract: $\mathcal{SC}_{\text{F-PSI}}$. Each of them put a certain amount of deposit to the contract. Then, they all together run a coin tossing protocol to agree on a key, used to generate a set of blinding polynomials that hide the result from the public. Then, each client locally maps its set elements to a hash table and represents the content of each hash table's bin as a polynomial. After that for each bin the clients do the following. All

- Parties: clients: $\{A_1, ..., A_m\}$, the dealer and an Arbiter.
- Input: Empty malicious clients list: L and a deployed smart contract's address.
- Output: Misbehaving clients list: L

1. Every client sends to the Arbiter two keys: k_1, k_2, used to generate the zero-sum values and their commitments.
2. The Arbiter checks if the clients provided correct keys, by ensuring that the keys' hashes matches the ones stored in the contract. It appends the IDs of those provided inconsistent keys to L. If all clients provided inconsistent keys it aborts. Otherwise, it proceed to the next step where it uses correct keys: k_1, k_2.
3. The Arbiter (given correct keys) regenerate the zero-sum values $z_{i,j}$ and verify the correctness of their commitments and the Merkel tree root contracted on top of the commitments, i.e. takes the same step as step 3 in Fig 2. It aborts if any of the checks is rejected, and appends all clients' IDs to L.
4. The dealer, for each client $C \in \{A_1, ..., A_m\}$, sends to the Arbiter a blind polynomial of the form: $\zeta \cdot \eta^{(D,C)} - (\gamma^{(D,C)} + \delta^{(D,C)})$, where $\eta^{(D,C)}$ is a fresh random degree $3d + 1$ polynomial. The blind polynomial will allow the arbiter to obliviously verify the correctness of the message each client sent to the contract.
5. The Arbiter for each client C who provided correct keys:
   (a) adds together the blind polynomial above and the blind polynomial $\nu^{(C)}$ the client sent to the contract (in step 8 in the PSI protocol). Then, it removes the client's zero-sum pseudorandom values from the result. In particular, it computes:

$$\begin{aligned}\iota^{(C)} &= \zeta \cdot \eta^{(D,C)} - (\gamma^{(D,C)} + \delta^{(D,C)}) + \nu^{(C)} - \sum_{i=0}^{3d+1} z_{i,c} \cdot x^i \\ &= \zeta \cdot (\eta^{(D,C)} + \omega^{(D,C)} \cdot \omega^{(C,D)} \cdot \pi^{(C)} + \rho^{(D,C)} \cdot \rho^{(C,D)} \cdot \pi^{(D)})\end{aligned}$$

   (b) checks if ζ can divide $\iota^{(C)}$. If can not, it appends the client's ID to L.

Fig. 3: `Arbiter` Protocol

clients, except the dealer, engage in `ZSPA` protocol to agree on a set of pseudorandom blinding factors (for each bin) such that the sum of them is zero. After that, the dealer randomises each of other clients polynomials. To do that, the dealer and every client engage in `VOPR` protocol that returns a randomised blinded polynomial to the client. The randomised blinded polynomial has also a specific structure, in the sense that after it is unblinded it is always dividable by a (secret) polynomial: ζ. The dealer and every other clients invoke `VOPR` again to randomize the dealer's polynomial. `VOPR` returns a randomised blinded polynomial to the client. This polynomial has the same structure as above. The client sums the two polynomial, masks it (using the output of `ZSPA`) and sends the result to the smart contract. The dealer also sends to the contract a polynomial that allows the contract to obliviously switch the blinding polynomials (unknown to the clients) to another blinding polynomial (known to all clients). Also, the dealer reveals the secret polynomial ζ to the contract. The contract computes the result by summing all clients polynomials (for each bin). Then, the contract checks if ζ can divide every result polynomial. The contract accepts the clients' inputs if the polynomial divides all result polynomials; otherwise, it invokes `Arbiter` protocol to pinpoint misbehaving

parties. In this case, all honest parties deposit is refunded to them 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 locally removes the blinding polynomial from the result, finds its set elements that mapped to that bin and then evaluates the unblinded polynomial (of that bin) at that elements and considers an element in the intersection if the evaluation equals zero.

4.6 Multi-party Fair PSI (F-PSI) Protocol

- **Input:** a pseudorandom function: PRF, a hash table's parameters (i.e. total number of bins: h and a bin's capacity: d), and clients' sets: $S^{(I)}$, where $I \in \mathbf{P}$.

1. All clients in $\mathbf{P}$ sign a smart contract: $\mathcal{SC}_{\textit{F-PSI}}$ and deploy it to a blockchain. All clients get the deployed contract's address. Furthermore, The clients engage in a coin tossing protocol to agree on a secret key: mk_1, of PRF.
2. Each client $I \in \mathbf{P}$ constructs hash table $\mathtt{HT}^{(I)}$, and inserts the elements into the table. $\forall i : \mathtt{H}(s_i^{(I)}) = \textit{indx}$, then $s_i^{(I)} \rightarrow \mathtt{HT}^{(I)}_{\textit{indx}}$. It pads every bin with random elements to d elements (if needed). Then, for every bin, it constructs a polynomial whose roots are the bin's content: $\pi^{(I)} = \prod_{i=1}^{c}(x - s'_i)$, where s'_i is either $s_i^{(I)}$, or a dummy value.
3. The clients $C \in \{A_1, ..., A_m\}$, excluding the dealer, for every bin, together generate and agree on $b = 3d + 2$ vectors of pseudorandom blinding factors: $z_{i,j}$, such that the sum of each vector elements is zero: $\sum_{j=1}^{m} z_{i,j} = 0$, where $0 \leq i \leq b - 1$. To do that, they participate in ZSPA protocol (presented in Fig. 2). After time t_1, the dealer ensures all other clients have agreed on the vectors (i.e. all provided "approved" to the contract); otherwise, it aborts.
4. Each client $I \in \mathbf{P}$ deposits y amount to $\mathcal{SC}_{\textit{F-PSI}}$. After time t_2, every client ensures that in total $y \cdot (m + 1)$ amount has been deposited. Otherwise, it aborts and the clients' deposit is refunded.
5. Client *D*, picks a random polynomial $\zeta \xleftarrow{\$} \mathbb{F}_p[x]$, where $deg(\zeta) = 1$. Client *D*, for each client *C*, allocates to each bin two degree d random polynomials: $\omega^{(D,C)}, \rho^{(D,C)} \xleftarrow{\$} \mathbb{F}_p[x]$, and two degree $3d + 1$ random polynomials: $\gamma^{(D,C)}$ and $\delta^{(D,C)}$. Also, each client *C*, for each bin, picks two degree d random polynomials: $\omega^{(C,D)}, \rho^{(C,D)} \xleftarrow{\$} \mathbb{F}_p[x]$
6. Client *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 client *D*, sends: $\zeta \cdot \omega^{(D,C)}$ and $\gamma^{(D,C)}$, while client *C*, sends $\omega^{(C,D)} \cdot \pi^{(C)}$ to VOPR. Each client *C*, for every bin, receives a blind polynomial:
$$\theta_1^{(C)} = \zeta \cdot \omega^{(D,C)} \cdot \omega^{(C,D)} \cdot \pi^{(C)} + \gamma^{(D,C)}$$
from VOPR. If any party aborts, the deposit would be refunded to all parties.
7. Each client *C* randomises client *D*'s polynomial. To do that, each client *C*, for each bin, invokes an instance of VOPR with client *D*; where each client *C*, sends $\rho^{(C,D)}$, while client *D* sends $\zeta \cdot \rho^{(D,C)} \cdot \pi^{(D)}$ and $\delta^{(D,C)}$ to VOPR. Every client which *C*, for each bin, gets a blind polynomial:
$$\theta_2^{(C)} = \zeta \cdot \rho^{(D,C)} \cdot \rho^{(C,D)} \cdot \pi^{(D)} + \delta^{(D,C)}$$
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 $\theta_1^{(C)}$ and $\theta_2^{(C)}$ using the blinding factors: $z_{i,c}$, generated in step 3. In particular, it computes the following blind polynomial (for every bin):
$$\nu^{(C)} = \theta_1^{(C)} + \theta_2^{(C)} + \sum_{i=0}^{3d+1} z_{i,c} \cdot x^i$$
Next, it sends all $\nu^{(C)}$ to $\mathcal{SC}_{\text{F-PSI}}$. If any party aborts, the deposit would be refunded to all parties.
9. Client D ensures all clients have sent their inputs to $\mathcal{SC}_{\text{F-PSI}}$ (otherwise, the deposit would be refunded to all parties). It allocates a fresh degree $3d$ pseudorandom polynomial: γ', to each bin. To do so, it uses k_1 to derive a key for each bin: $k_{l,indx} = \texttt{PRF}(k_1, indx)$ and then uses the derived key to generate $3d+1$ pseudorandom coefficients $g_{j,indx} = \texttt{PRF}(k_{l,indx}, j)$ where $0 \leq j \leq 3d$. Also, for each bin, it allocates a fresh degree d random polynomial: $\omega'^{(D)}$.
10. Client D, for every bin, computes a polynomial of the form:
$$\nu^{(D)} = \zeta \cdot \omega'^{(D)} \cdot \pi^{(D)} - (\sum_{C=A_1}^{A_m} \gamma^{(D,C)} + \delta^{(D,C)}) + \zeta \cdot \gamma'$$
Then, it sends to $\mathcal{SC}_{\text{F-PSI}}$ all polynomials $\nu^{(D)}$ along with the single polynomial ζ.
11. The smart contract, $\mathcal{SC}_{\text{F-PSI}}$:
    (a) for every bin, sums all polynomials provided by all clients $I \in \mathbf{P}$:
    $$\phi = \nu^{(D)} + \sum_{C=A_1}^{A_m} \nu^{(C)}$$
    $$= \zeta \cdot (\omega'^{(D)} \cdot \pi^{(D)} + (\sum_{C=A_1}^{A_m} \omega^{(D,C)} \cdot \omega^{(C,D)} \cdot \pi^{(C)}) + (\pi^{(D)} \cdot \sum_{C=A_1}^{A_m} \rho^{(D,C)} \cdot \rho^{(C,D)}) + \gamma')$$
    (b) ensures that, for every bin, ζ divides ϕ. Otherwise, it aborts and Arbiter protocol (presented in Fig. 3) is invoked to find misbehaving parties.
    (c) if the verification passed, each party gets back its deposit (i.e. y amount).
12. Each client (given ζ and k_1), finds the elements in the intersection as follows. First, it derives a bin's pseudorandom polynomial: γ' from k_1. Next, it removes the blinding polynomial from each bin's polynomial: $\phi' = \phi - \zeta \cdot \gamma'$. Then, it evaluates each bin's unblinded polynomial at every element belonging to that bin and considers the element in the intersection if the evaluation is zero: i.e. $\phi'(s_i^{(I)}) = 0$

Remark 1. After the Arbiter detects misbehaving parties, in step 11b, it sends their ID's to $\mathcal{SC}_{\text{F-PSI}}$ which refunds the honest parties' deposit and splits the misbehaving parties' deposit among the honest ones. Thus, each honest party would receive: $y + \frac{m' \cdot y}{m - m'}$ amount in total, where m' is the total number of misbehaving parties.

5 Smart PSI Protocol

In this section we present an efficient smart PSI that allows honest parties who contribute their set to get paid by a buyer who initiates the PSI computation and interested in the result.

5.1 Threat model

In this section, we provide the security model of our smart-PSI protocol. There are two kind of parties involved in the protocol. Namely, (1) a set of clients $\{A_1, ..., A_m\}$ potentially *rational* (i.e. an adversary that picks the best strategy to maximise its profit) and all may collude with each other, and (2) a non- colluding dealer: client D, potentially semi-honest (i.e. a passive adversary). We assume client A_m is the buyer, i.e. the party initiates the PSI computation and is interested in the result. Similar to F-PSI, we consider static adversary, we assume there is an authenticated private (off- chain) channel between the clients and we consider a standard public blockchain.

5.2 Overview

The main building blocks of the protocol is (modified) F-PSI protocol, a smart contract, and the counter collusion smart contracts proposed in [2].

Two arbitrary clients become volunteers to be result extractors, responsible for finding the elements in the intersection. In return, they (if act honestly) will be rewarded for doing so. The extractors sign a smart contract: $\mathcal{SC}_{EXT}$, with the rest of the clients. Then, the extractors individually commit to their set elements and store a batch of the commitments in $\mathcal{SC}_{EXT}$. Then all clients run F-PSI protocol that outputs blinded result polynomials on F-PSI's smart contract: $\mathcal{SC}_{F\text{-}PSI}$. Next, each extractor proves to $\mathcal{SC}_{EXT}$ that it knows a relation in each blinded result polynomial (that requires the knowledge of an element in the intersection). If the extractors' proofs are accepted and both provided consistent results, then the contract distributes the buyer deposit among all clients and rewards the extractors.

However, we have to apply two essential modifications to F-PSI protocol. Generally speaking, the modifications are applied to (a) preserve the privacy of the elements in the intersection when each extractor proves the knowledge of them to the contract and (b) to identify misbehaving extractor without needing to have access to the extractor's set elements. The role of collusion smart contracts is to create a distrust between the extractors and buyer who may collude (out of the bound) to increase their profit.

5.3 Smart PSI Protocol

1. All clients in **P** together run step 1 of the PSI protocol (in Section 4) to deploy and get the address of deployed fair PSI contract: $\mathcal{SC}_{F\text{-}PSI}$.
2. All clients in **P** with the extractor clients sign and deploy extractor contract: $\mathcal{SC}_{EXT}$. The address of the deployed extractor contract is given to all clients.
3. The buyer, client A_m, before time t_1 deposits: $S_{min} \cdot v$ amount to $\mathcal{SC}_{EXT}$.
4. All clients after time $t_2 > t_1$ ensure the buyer has deposited $S_{min} \cdot v$ amount on $\mathcal{SC}_{EXT}$. Otherwise, they abort.
5. The dealer client signs the Prisoner's contract with the extractor clients. $\mathcal{SC}_{EXT}$ transfers: $S_{min} \cdot (v - m \cdot l)$ amount (from the buyer deposit) to the Prisoner's contract. The transferred amount is considered as the dealer's deposit to pay the extractors.
6. All clients in **P** engage in a coin tossing protocol to agree on two keys: mk_1 and mk_2.

7. Each client $I \in \mathbf{P}$ maps the elements of its set $S^{(I)} : \{s_1^{(I)}, ..., s_c^{(I)}\}$ to random values by encrypting them as follows. $\forall i, 1 \leq i \leq c : e_i^{(I)} = \mathtt{PRP}(mk_1, s_i^{(I)})$. Then, it encodes its encrypted set element as $\hat{e}_i^{(I)} = e_i^{(I)} || \mathtt{H}(e_i^{(I)})$. After that, it constructs a hash table $\mathtt{HT}^{(I)}$ and inserts the encrypted elements into the table. $\forall i : \mathtt{H}(\hat{e}_i^{(I)}) = j$, then $\hat{e}_i^{(I)} \rightarrow \mathtt{HT}_j^{(I)}$. It pads every bin with random elements to d elements (if needed). Then, for every bin, it constructs a polynomial whose roots are the bin's content: $\pi^{(I)} = \prod_{i=1}^{d}(x - e'_i)$, where e'_i is either $\hat{e}_i^{(I)}$, or a dummy value.
8. Every extractor client $I \in \mathbf{E}$:
   (a) for each bin, derives a pseudorandom polynomial: γ'_j, using key mk_2.
   (b) for each bin, evaluates γ'_j at the encrypted set elements of that bin: $\gamma'_{j,i} = \gamma'_j(\hat{e}_i^{(I)})$.
   (c) for each bin, commits to the evaluations: $\mathtt{a}_{j,i}^{(I)} = \mathtt{Com}(\gamma'_{j,i}, q_i^{(I)})$, where $q_i^{(I)}$ is a fresh randomness used for the commitment. If the bin contains paddings, then it commits to the paddings as well.
   (d) constructs a Merkel tree on top of all committed values: $\mathtt{MT}(\mathtt{a}_{1,1}^{(I)}, ..., \mathtt{a}_{h,d}^{(I)}) \rightarrow g^{(I)}$
   (e) stores the Merkel tree's root: $g^{(I)}$, on $\mathcal{SC}_{EXT}$.
9. All clients in $\mathbf{P}$ (given mk_2 and the address of $\mathcal{SC}_{F\text{-}PSI}$) run steps 3-11 of the fair PSI protocol. At the end of this step, result polynomials are output by $\mathcal{SC}_{F\text{-}PSI}$.
10. Every extractor client $I \in \mathbf{E}$:
    (a) Finds the elements in the intersection. To do so, first it encrypts each of its set element(i.e. $e_i^{(I)} = \mathtt{PRP}(mk_1, s_i^{(I)})$) and then encodes it: (i.e. $\hat{e}_i^{(I)} = e_i^{(I)} || \mathtt{H}(e_i^{(I)})$). Next, it determines to which bin the encrypted value belongs (i.e. $j = \mathtt{H}(\hat{e}_i^{(I)})$). Next, it evaluates the result polynomial for that bin at the encrypted element. It considers the element in the intersection if the evaluation is zero (i.e. $\phi(\hat{e}_i^{(I)}) - \zeta(\hat{e}_i^{(I)}) \cdot \gamma'_j(\hat{e}_i^{(I)}) = 0$).
    (b) Proves that every element in the intersection is among the elements it has committed to. In particular, for each element in the intersection, e.g. $\hat{e}_i^{(I)}$, it sends to $\mathcal{SC}_{EXT}$:
        - the evaluation commitment for that element: $\mathtt{a}_{j,i}^{(I)} = \mathtt{Com}(\gamma'_{j,i}, q_i^{(I)})$, where $q_i^{(I)}$ was generated in step 8.
        - $\hat{e}_i^{(I)}$ and it commitment's opening: $\mathtt{m}_i^{(I)} = (\gamma'_{j,i}, q_i^{(I)})$.
        - a proof: $\mathtt{h}_i^{(I)}$ that proves $\mathtt{a}_{j,i}^{(I)}$ is a leaf node of a Merkel tree with root $g^{(I)}$.
        - the index of the bin to which $\hat{e}_i^{(I)}$ belongs, i.e. $j = \mathtt{H}(\hat{e}_i^{(I)})$.
11. Contract $\mathcal{SC}_{EXT}$:
    (a) verifies the commitment openings, $\mathtt{h}_i^{(I)}$ and if the encrypted element is the polynomial's root, in that bin.
    $$\mathtt{Ver}_{\mathtt{com}}(\mathtt{a}_{j,i}^{(I)}, \mathtt{m}_i^{(I)}) = 1 \quad \wedge \quad \mathtt{Ver}_{\mathtt{MT}}(\mathtt{h}_i^{(I)}, g^{(I)}) = 1 \quad \wedge \quad \phi(\hat{e}_i^{(I)}) - \zeta(\hat{e}_i^{(I)}) \cdot \gamma'_{j,i} = 0$$
    (b) for every accepting proof, it takes $m \cdot e$ coins from the buyer's deposit and distributes it among all clients (except the buyer).
12. Contract $\mathcal{SC}_{EXT}$ after time t_3 checks if $|S_\cap| < S_{min}$. In this case, it returns $(S_{min} - |S_\cap|) \cdot v$ amount to the buyer.

Remark 1. The protocol requires extractor clients to commit to their encrypted set elements before the protocol starts to prevent the clients to find some random roots introduced in the result and sell them to the buyer. Furthermore, instead of the extractor's

commitment scheme we proposed, one could use polynomial commitment scheme [5]. However, our scheme (tailored for our Smart PSI) is more efficient, as it is based on efficient symmetric key primitives; whereas, [5] requires expensive public key and bilinear pairing operations.

Remark 2. The Traitor's contract must be signed between the dealer and the extractor client who betrays the other colluding extractor or buyer before step 10b. Also, the dealer pays the refundable deposits in the Traitor's contract.

Remark 3. For ease of exposition, ZSPA protocol (in Fig 2) was presented for single bin. However, it can easily be extended to multiple bins (using only two keys: k_1, k_2). To do so, for each bin l (in step 2) each blinding factor is computed as $z_{i,j,l} = \texttt{PRF}(k_1, i||j||l)$ and the randomness of the commitment is generated as: $q_{i,j,l} = \texttt{PRF}(k_2, i||j||l)$. Also, a Merkel tree is built on top of all bins' committed values that yields single root.

Remark 4. The Merkle three is used to reduce the contract-side storage cost each time an instance of the PSI is run.

Remark 5. The reason smart PSI has a verification mechanism for the extractors (instead of solely relying on the arbiter in the counter collusion contracts) is to minimise the role the arbiter, and let $\mathcal{SC}_{EXT}$ resolve most of dispute.

Remark 6. In Smart PSI (unlike F-PSI) the intersection cardinality is revealed to the public. The clients can use padding to hide the exact number of elements. Specifically, that all clients agree on a set of elements and all insert them to their set in setup phase.

6 Related Work and Attacks

The closest protocol to ours is [4] that proposes three efficient PSI protocols: two-party, multi-party, and threshold, all in the malicious model. It mainly utilises polynomial representation of sets and a variant of oblivious linear function evaluation [3] called oblivious polynomial addition (OPA). For the sake of simplicity, we explain the two-party PSI protocol in [4] where clients A and B are involved. At a high level, each party generates a polynomial representing their set. Then, they invoking oblivious polynomial addition (OPA) protocol, presented in Fig. 5, to randomize each other polynomials. After that, they exchange messages that allow them to remove a part of random polynomials, and find the result. The two-party PSI protocol is provided in detail in Fig. 4.

Attack 1: *Learning the result without letting other client(s) do so*: In the following, for the sake of simplicity, we explain an attack mounted on the two-party PSI. We observed that the check in step 2c is not sufficient, as a malicious party can mount an attack, and deviate from the protocol without being detected. The attack allows a malicious party to modify the output of OPA such that it can always learn the actual PSI result but make its counter-party (honest party) believe that there is no element in the intersection. In the following we explain the attack in detail.

1. **PSI Computation**
   (a) Client $I \in \{A, B\}$: represent its set elements as a degree m polynomial: $p_I = \prod_{j=1}^{c}(x - s_j^A) \cdot \omega_I(x)$, where $\omega_I(x)$ is a random polynomial. Each client I picks three random polynomials: r_I, u_I and r'_I. Then each client picks three random polynomials: r_I, u_I and r'_I.
   (b) The clients invoke OPA where client A inserts r_A, u_A and client B inserts p_B to OPA, which outputs $s_B = p_B \cdot r_A + u_A$ to client B.
   (c) The clients again invoke OPA, this time client A inserts p_A while client B inserts r_B, u_B to OPA that outputs $s_A = p_A \cdot r_B + u_B$ to client A.
   (d) Client A: sends $s'_A = s_A - u_A + p_A \cdot r'_A$ to client B, where r'_A is a random polynomial.
   (e) Client B: computes: $p_\cap = s'_A + s_B + p_B \cdot r'_B - u_B = p_A \cdot r'_A + p_A \cdot r_B + p_B \cdot r_A + p_B \cdot r'_B$. It sends the result polynomial: $p_\cap$ to client A.
   (f) To find the intersection, client I evaluates polynomial $p_\cap$ at every element of its set, s_j^I, and consider the element in the intersection if $p_\cap(s_j^I) = 0$.
2. **Output Verification**
   (a) The clients agree on two random values: z and q, e.g. via a coin tossing protocol.
   (b) Client B sends $\alpha_B = p_B(z)$, $\beta_B = r_B(z)$, and $\delta_B = r'_B(z)$ to client A.
   (c) Client A: checks if: $p_\cap(z) \stackrel{?}{=} p_A(z) \cdot (\beta_B + r'_A(z)) + \alpha_B \cdot (r_A(z) + \delta_B)$
   (d) Client A sends $\alpha_A = p_A(q)$, $\beta_A = r_A(q)$, and $\delta_A = r'_A(q)$ to client A.
   (e) Client B: checks if: $p_\cap(q) \stackrel{?}{=} p_B(q) \cdot (\beta_A + r'_B(q)) + \alpha_A \cdot (r_B(q) + \delta_A)$

Fig. 4: Two-party PSI Protocol in [4]

Let client B be a malicious party. Client B in step 1e, instead of inserting the product $p_B \cdot r'_B$ it inserts r'_B. Therefore, now in the equation we have:

$$\begin{aligned} p_\cap &= s'_A + s_B + r'_B - u_B \\ &= p_A \cdot r'_A + p_A \cdot r_B + p_B \cdot r_A + r'_B \end{aligned} \tag{1}$$

Client B sends $p_\cap$ generated in equation 1 to client A. Client B, in step 2b, computes $\alpha_B = p_B(z)$, $\beta_B = r_B(z)$ honestly as before, but it sets $\delta'_B = r'_B(z) \cdot (\alpha_B)^{-1}$ (instead of setting $\delta'_B = r'_B(z)$). It sends α_B, β_B and δ'_B to client A who checks:

$$\begin{aligned} p_\cap(z) &\stackrel{?}{=} p_A(z) \cdot (\beta_B + r'_A(z)) + \alpha_B \cdot (r_A(z) + \delta'_B) \\ &\stackrel{?}{=} p_A(z) \cdot (\beta_B + r'_A(z)) + \alpha_B \cdot r_A(z) + r'_B(z) \end{aligned} \tag{2}$$

Note that client B can always pass the above check, even though it has provided an incorrect input. Moreover, the above modification prevents client A from learning the actual intersection. Because, the sum of the polynomial representing the intersection of the clients set: $p_A \cdot r'_A + p_A \cdot r_B + p_B \cdot r_A$ and random polynomial r'_B would be a uniformly random polynomial that does have common roots with the clients' polynomials with a high probability, therefore the intersection would be empty. This security issue is inherited by both the multi-client and multi-client threshold PSI protocols as well, as they utilise the same verification mechanism.

Attack 2: *Learning other client's set element with a non-negligible probability:* A main component of the three PSI protocols is OPA required to be secure against malicious adversaries, in the sense that it ensures that the parties insert well-formed inputs and do not learn anything by inserting 0 as their input. In particular, OPA uses: (a) OLE protocol with no checks, called $\mathcal{F}_{OLE}$, and (b) OLE that ensures when an adversary inserts 0 it learns nothing, it is denoted by $\texttt{OLE}^+$.

- **Public parameters**: a vector of distinct non-zero elements: $\vec{\boldsymbol{x}} = [x_1, ...x_{2d+1}]$

1. **Computing** $s(x) = p(x) \cdot r(x) + u(x)$, where the sender has $u(x), r(x)$ and the receiver has $p(x)$ as inputs, $deg(u) \leq 2d, deg(r) = d, deg(p) \leq d$
   (a) Sender: $\forall j, 1 \leq j \leq 2d+1$, computes $u_j = u(x_j)$ and $r_j = r(x_j)$. Then, it inserts (u_j, r_j) into $\texttt{OLE}^+$
   (b) Receiver: $\forall j, 1 \leq j \leq 2d+1$, computes $p_j = p(x_j)$. Then, it inserts p_j into $\texttt{OLE}^+$ and receives $s_j = p_j \cdot r_j + u_j$. It interpolates a polynomial $s(x)$ using pairs (x_j, s_j). Next, it checks if $deg(s) \leq 2d$. Otherwise, it aborts.
2. **Consistency check**:
   (a) Sender: picks a random x_s, and sends it to the receiver.
   (b) Receiver: picks random values f, v and inserts them into an instance of $\mathcal{F}_{OLE}$, denoted by $\mathcal{F}^1_{OLE}$. It inserts $(p(x_s), -s(x_s) + f)$ into another instance of $\mathcal{F}_{OLE}$, say $\mathcal{F}^2_{OLE}$.
   (c) Sender: picks a random value t, and inserts it to $\mathcal{F}^1_{OLE}$ that sends $c = f \cdot t + v$ to the sender. It also inputs $r(x_s)$ into $\mathcal{F}^2_{OLE}$ that sends $r(x_s) \cdot p(x_s) - s(x_s) + f$ to the sender who sums it with $u(x_s)$. This yields $f' = r(x_s) \cdot p(x_s) - s(x_s) + f + u(x_s)$. The sender sends f' to the receiver.
   (d) Receiver: It aborts if $f' \neq f$; otherwise, it sends v to the sender.
   (e) Sender: It aborts if $f' \cdot t + v \neq c$
3. Receiver: picks x_r and runs similar consistency check with the sender.

Fig. 5: Oblivious Polynomial Addition (OPA) in [4]

In the following, we explain an attack that can be carried out in OPA protocol. We observed that a malicious sender can use OPA as a subroutine to guess one of the receiver's set elements without being caught with a non-negligible probability (i.e. probability independent of the security parameter). In particular, in step 2a, the malicious sender instead of picking a uniformly random value: x_s, it guesses an element of the receiver's set, e.g. x'_s, and sends that value to the receiver. The sender takes the rest of the steps as before. But, in step 2c, it replaces $w = r(x_s)$ with an arbitrary value, say $w' = 1$, and inserts w' to $\mathcal{F}^2_{OPA}$. It computes f' that now has the following form: $f' = w' \cdot p(x'_s) - s(x'_s) + f + u(x'_s) = w' \cdot p(x'_s) - p(x'_s) \cdot r(x'_s) + f$. Note that, if it correctly guesses the set element, then $p(x'_s) = 0$, which means $f' = f$. Therefore, it can pass the check in step 2d.

One might be tempted to replace $\mathcal{F}^2_{OLE}$ by $\texttt{OLE}^+$ in step 2b to prevent the honest receiver from sending $p(x'_s) = 0$. However, this would not work, as $\texttt{OLE}^+$ only ensures that when someone sends x and another one sends l, g, the party who sends x cannot learn any extra information when $x = 0$. But in OPA, the honest receiver sends l, g and there is no mechanism in place to prevent it from sending zero value(s). The probability

that a malicious sender correctly guesses the receiver's set element is $Pr = \frac{1}{|\mathcal{U}|}$, where $\mathcal{U}$ is the universe of set elements. This probability is based on the assumption that all set elements have the same distribution (which is not always the case). For the cases where the elements have not the same distribution, the probability of correctly guessing can be even higher, i.e. $Pr \geq \frac{1}{|\mathcal{U}|}$.

The problem can be tackled by at least two approaches: (a) assuming that the universe of set elements is very big, so the adversary can guess a set element with only a negligible probability. But, this is a rare and strong assumption, the PSI would have very limited applications and we might be able to design more efficient PSI protocol based on this assumption. (b) running a coin-tossing protocol by sender and receiver to compute x_s, with a small added cost.

Attack 3: *Deleting Other Client's Set Elements*. In the two-party PSI protocol in step 1b, clients A and B invoke an instance of OPA such that client A (as sender) inserts two random polynomials: r_A, u_A, and client B (as receiver) inserts polynomial p_B that represents its set. In the end of this step, OPA returns $s_B = p_B \cdot r_A + u_A$ to client B. The proposed OPA (as shown in steps 1a and 1b) accepts y-coordinates of the polynomials as input. This means, the parties first evaluate their polynomials at every element of $\vec{\boldsymbol{x}}$ to get a set of y-coordinates and then send the corresponding y-coordinates to OPA. However, we observed that this leads to a serious security issue; namely, a malicious sender can guess some of the receivers set elements and delete them. See Proposition 1 below, for a formal definition and proof. The probability of successfully mounting this attack is independent of the security parameter and only depends on the size of elements' universe. This means that the probability can be high when the size is not very large. Before we elaborate on the attack, we provide the following theorem and proposition used in the attack's explanation.

Theorem 1. *(Uniqueness of Interpolating Polynomial [6]) Let $\vec{\boldsymbol{x}} = [x_1, ...x_h]$ be a vector of non-zero distinct elements. For h arbitrary values: $y_1, ...y_h$ there is a unique polynomial: τ, of degree at most $h-1$ such that: $\forall j, 1 \leq j \leq h : \tau(x_j) = y_j$, where $x_j, y_j \in \mathbb{F}_p$.*

Proposition 1. *Let $\vec{\boldsymbol{x}}$ be a vector defined above and $\mu \in \mathbb{F}_p[x]$ be a degree $d < h$ polynomial with d' distinct roots, i.e. $\mu = \prod_{i=1}^{d}(x - e_i)$, and let $\mu_j = \mu(x_j)$, where $1 \leq j \leq d$. Polynomial μ' interpolated from pairs $(x_1, \mu_1 \cdot (x_1 - e_c)^{-1}), ..., (x_h, \mu_h \cdot (x_h - e_c)^{-1})$ will not have e_c as root, i.e. $\mu'(e_c) \neq 0$, where $1 \leq c \leq d$.*

Proof. For the sake of simplicity and without loss of generality, let $c = 1$. We can rewrite polynomial μ as $\mu = (x - e_1) \cdot \prod_{i=2}^{d}(x - e_i)$, then every μ_j $(1 \leq j \leq h)$ can be written as: $\mu_j = (x_j - e_1) \cdot \prod_{i=2}^{d}(x_j - e_i)$. Accordingly, for every j, the product $\alpha_j = \mu_j \cdot (x_j - e_1)^{-1}$ has the following form: $\alpha_j = \mu_j \cdot (x_j - e_1)^{-1} = \prod_{i=2}^{d}(x_j - e_i)$. Let μ' be another polynomial with $d-1$ distinct roots identical to the roots of μ excluding

e_1, i.e. $\mu'(e_1) \neq 0$. Each μ'_j can also be written as $\mu'_j = \prod_{i=2}^{d}(x_j - e_i)$. Due to Theorem 1, given h pairs (x_j, μ'_j) there is at most one polynomial of degree at most $h-1$, such that its evaluation on x_j yields μ'_j, and that polynomial is μ' (defined above). Since, $\mu'_j = \alpha_j$, the polynomial interpolated from h pairs (x_j, α_j) is μ' as well. Since μ' does not have e_1 as root, the polynomial interpolated from d pairs $(x_1, \mu_1 \cdot (x_1 - e_1)^{-1}), ..., (x_h, \mu_h \cdot (x_h - e_1)^{-1})$ would not have e_1 as root either. □

Now we explain the attack in detail. Recall that in step 1b, $(\forall j, 1 \leq j \leq h)$ client B sends $p_B(x_j) = \gamma_B(x_j) \cdot \prod_{i=1}^{d}(x_j - s_i^{(B)})$ while client A sends $r_A(x_j)$ and $u_A(x_j)$ to OPA, where values $s_i^{(B)}$ are client B's set elements and γ_B is a random polynomial. Assume that malicious client A knows at least one of client B set elements, e.g. $s_1^{(B)}$. In this case, client A replaces $r_A(x_j)$ with $r_j = r'_A(x_j) \cdot (x_j - s_1^{(B)})^{-1}$, where r'_A is a random polynomial. Next, client A (in step 1b) sends r_j and $u_A(x_j)$ to OPA which returns:

$$s'_B(x_j) = p_B(x_j) \cdot r_j + u_A(x_j) = \gamma_B(x_j) \cdot r'_A(x_j) \cdot \prod_{i=2}^{d}(x_j - s_i^{(B)}) + u_A(x_j) \quad (3)$$

to client B who uses pairs $(x_j, s'_B(x_j))$ to interpolate a polynomial: s'_B. As evident above and according to Proposition 1, malicious client A has managed to remove $s_1^{(B)}$ from the output of OPA sent to client B. Client A can also easily pass the verifications with the following trick. In step 2b in OPA (Fig. 5), any time it (as sender) receives a random challenge: x_s, it acts as it was honest with an exception. Namely, it sends $r'_A(x_s) \cdot (x_s - s_1^{(B)})^{-1}$, instead of $r_A(x_s)$, to $\mathcal{F}_{OLE}^2$ that returns $f' = r'_A(x_s) \cdot (x_s - s_1^{(B)})^{-1} \cdot p_B(x_s) - s'_B(x_s) + f$ to client A. Recall that $s'_B(x_s)$ and $p_B(x_s)$ were inserted by client B, where $s_1^{(B)}$ is not a root of s'_B (as client A has already removed it in Equation 3), whereas s'_B is a root of $p_B(x_s)$. However, during the verification, client A's input: r_j, can remove s'_B from the root list of $p_B(x_s)$ too, according to Proposition 1. Therefore, client A can pass the verification above. Client A can utilise the trick above to pass the verification in the PSI protocol (i.e. step 2e in Fig. 4) too. In particular, given a fresh random challenge: x_B, it sends $r'_A(x_B) \cdot (x_B - s_1^{(B)})^{-1}$, instead of $r_A(x_B)$, to client B.

Technical issue : Another issue in OPA protocol is that the receiver in step 1b checks the degree of polynomial $s(x)$ after it interpolates the polynomial from $m+1$ pairs (x_j, s_j) and it aborts if $deg(s) > 2d$. The protocol's proof also highly depends on this check. Nonetheless, according to Theorem 1, there exists a polynomial that *always* have a degree at most $2d$. Thus, the check in step 1b does not work and is always passed.

7 Some lemmas and proofs

Informally, the following lemma states that evaluation of a random polynomial at a fixed value results in a uniformly random value. It will be used in VOPR's proof to show that during the verification (in VOPR) a malicious party cannot learn anything about its counter party's input.

Lemma 1. *Let x_i, y_i be arbitrary elements of a finite field $\mathbb{F}_p$, where p is a security parameter and sufficiently large prime number. The probability that the evaluation of a random polynomial $\mu(x)$ at x_0 equals y_0 is negligible. More formally, for $x_i, y_i \in \mathbb{F}_p$ and $\mu(x) \xleftarrow{\$} \mathbb{F}_p[x]$, $1 \leq deg(\mu) \leq d$, we have: $Pr[\mu(x_i) = y_i] = \epsilon(p)$*

Proof. Let $\mu(x) = a_0 + \sum_{j=1}^{d} a_j x^j$, where the coefficients are distributed uniformly at random over the field. For any choice of x_i, y_i and $a_1, ..., a_d$, there exists exactly one value of a_0 that makes $\mu(x_i) = y_i$, i.e. $\mu(x_i) = y_i$ iff $a_0 = y_i - \sum_{j=1}^{d} a_j x_i^j$. As a_0 is picked uniformly at random, the probability that it equals a certain value that makes $\mu(x_i) = y_i$ is $\frac{1}{p}$, which is negligible in the security parameter. Thus, $Pr[\mu(x_i) = y_i] = \frac{1}{p}$ □

The following lemma states that the product of two polynomials preserves their roots.

Corollary 1. *Let $\tau(x)$ and $\gamma(x)$ be two arbitrary non-constant polynomials of degree at most d such that $\tau(x), \gamma(x) \in \mathbb{F}_p[x]$. Then, the product of the two polynomials does not cancel out the polynomials roots, so the roots of $\alpha(x) = \tau(x) \cdot \gamma(x)$ include all roots of both $\tau(x)$ and $\gamma(x)$.*

Proof. Let $\gamma(x) = \prod_{i=1}^{b}(x - c_1) \prod_{j=1}^{d-b}(x - e_j)$, where $1 \leq b \leq d$. The only way to remove b roots, e.g. c_i, from $\gamma(x)$ via polynomial multiplication is either to set $\tau(x)$ to $\frac{\sigma(x)}{\prod_{i=1}^{b}(x-c_1)}$ for any non-zero $\sigma(x)$, not divisible by the denominator. But, $\frac{\sigma(x)}{\prod_{i=1}^{b}(x-c_1)}$ is undefined in $\mathbb{F}_p[x]$. Or, to $\tau(x)$ to the multiplicative inverse of $\prod_{i=1}^{b}(x - c_1)$. Nonetheless, there is no non scalar polynomials in $\mathbb{F}_p[x]$ that has a multiplicative inverse. Thus, the product of the two polynomials preserves all roots of both. □

The following lemma will be used in the proof to show that in the F-PSI if clients modify the output of `VOPR` then the contract will detect it with a high probability, using the proposed verification mechanism.

Lemma 2. *Let $\alpha(x)$ and $\xi(x)$ be two non-zero random polynomials, where $\alpha(x), \xi(x) \xleftarrow{\$} \mathbb{F}_p[x], deg(\alpha) = d \geq 1, deg(\xi) = 2d$, and $gcd(\alpha, \xi) = 1$. The two polynomials are unknown to an adversary. Let f be an arbitrary non-zero polynomial of degree d picked and known by the adversary. Given polynomial $s = \alpha \cdot f + \xi$, if the adversary changes s to s', then α will divide $s' - \xi$ only with a negligible probability in the security parameter p. Formally:*

$$Pr[\alpha \mid (s' - \xi)] \leq \epsilon(p)$$

Proof. We show that if an arbitrary non-zero degree d polynomial $\eta(x)$ is applied to polynomial $s(x)$ via any of polynomial arithmetics, by an adversary without the knowledge of the random polynomials $\alpha(x)$ and $\xi(x)$, then α will divide $s'(x) - \xi(x)$ only with a negligible probability.

- *Addition*: in this case, s' is of the form: $s' = s + \eta = \alpha \cdot f + \xi + \eta$, therefore $s' - \xi = \alpha \cdot f + \eta$. In order for α to divide $s' - \xi$, it must divide both $\alpha \cdot f$ and η. As evident, it can divide $\alpha \cdot f$. For α to divide η the following must hold: $gcd(\alpha, \eta) = \alpha$. However, α is picked uniformly at random unknown to the adversary. So we have $Pr[\alpha \mid (s' - \xi)] = Pr[gcd(\alpha, \eta) = \alpha] \leq \frac{1}{p^{d+1}}$ which is negligible.
- *Subtraction*: it is very similar to the addition above.
- *Multiplication*: in this case s' is of the form: $s' = s \cdot \eta = \alpha \cdot f \cdot \eta + \xi \cdot \eta$, therefore $s' - \xi = \alpha \cdot f \cdot \eta + \xi \cdot (\eta - 1)$. For α to divide $s' - \xi$, it must divide both $\alpha \cdot f \cdot \eta$ and $\xi \cdot (\eta - 1)$. It can divide the first term so we focus on the second term: $\xi \cdot (\eta - 1)$. Since $gcd(\alpha, \xi) = 1$, α must divide $\eta - 1$, which means the following must hold: $gcd(\alpha, \eta - 1) = \alpha$. But, as we stated above, α is picked uniformly at random. Therefore, we will have $Pr[\alpha \mid (s' - \xi)] = Pr[gcd(\alpha, \eta - 1) = \alpha] \leq \frac{1}{p^{d+1}}$ which is negligible as well.
- *Division*: in this case, s' is of the form: $s' = \frac{s}{\eta} = \frac{\alpha \cdot f + \xi}{\eta}$, so $s' - \xi = \frac{\alpha \cdot f + \xi}{\eta} - \xi$. We have two cases:
  1. $\eta | s$: The probability that this event happens is at most $\frac{1}{p}$, negligible, as η must divide ξ.
  2. $\eta \not| s$: In this case, we have $s' = \frac{s}{\eta} = \eta \cdot q + r$, for some quotient q and remainder r. In this case, $s' - \xi = \eta \cdot q + r - \xi$. For α to divide $s' - \xi$, it must divide $\eta \cdot q, r$, and ξ, but it cannot divide ξ, because $gcd(\alpha, \xi) = 1$ □

References

1. Ben-Sasson, E., Fehr, S., Ostrovsky, R.: Near-linear unconditionally-secure multiparty computation with a dishonest minority. In: Annual Cryptology Conference. pp. 663–680. Springer (2012)
2. Dong, C., Wang, Y., Aldweesh, A., McCorry, P., van Moorsel, A.: Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 211–227. ACM (2017)
3. Ghosh, S., Nielsen, J.B., Nilges, T.: Maliciously secure oblivious linear function evaluation with constant overhead. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 629–659. Springer (2017)
4. Ghosh, S., Nilges, T.: An algebraic approach to maliciously secure private set intersection. In: Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III. pp. 154–185 (2019)
5. Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: Abe, M. (ed.) Advances in Cryptology - ASIACRYPT 2010. pp. 177–194. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
6. Quarteroni, A., Sacco, R., Saleri, F.: Numerical mathematics, vol. 37. Springer Science & Business Media (2010)