# P2DEX: Privacy-Preserving Decentralized Cryptocurrency Exchange

Carsten Baum[1], Bernardo David[2⋆], and Tore Kasper Frederiksen[3]

[1] Aarhus University, Denmark
cbaum@cs.au.dk
[2] IT University of Copenhagen, Denmark
bernardo@bmdavid.com
[3] Alexandra Institute, Denmark
tore.frederiksen@alexandra.dk

**Abstract.** Cryptocurrency exchange services are either trusted central entities that have been routinely hacked (losing over 8 billion USD), or decentralized services that make all orders public before they are settled. The latter allows market participants to "front run" each other, an illegal operation in most jurisdictions. We extend the "Insured MPC" approach of Baum *et al.* (FC 2020) to construct an efficient universally composable privacy preserving decentralized exchange where a set of servers run private cross-chain exchange order matching in an outsourced manner, while being financially incentivised to behave honestly. Our protocol allows for exchanging assets over multiple public ledgers, given that users have access to a ledger that supports standard public smart contracts. If parties behave honestly, the on-chain complexity of our construction is as low as that of performing the transactions necessary for a centralized exchange. In case malicious behavior is detected, users are automatically refunded by malicious servers at low cost. Thus, an actively corrupted majority can only mount a denial-of-service attack that makes exchanges fail, in which case the servers are publicly identified and punished, while honest clients do not to lose their funds. For the first time in this line of research, we report experimental results on the MPC building block, showing the approach is efficient enough to be used in practice.

**Keywords:** Multiparty Computation, Secure Asset Exchange, Front-running, Blockchain

## 1 Introduction

Decentralized cryptocurrencies based on permissionless ledgers such as Bitcoin [49] allow for users to perform financial transactions without relying on central authorities. However, exchanging coins among different decentralized cryptocurrency platforms still mainly relies on centralized exchange services who must

⋆ This work was supported by the Concordium Foundation, by Protocol Labs grant S$^2$LEDGE and by the Independent Research Fund Denmark with grants number 9040-00399B (TrA$^2$C) and number 9131-00075B (PUMA).

hold tokens during the exchange process, making them vulnerable to theft. Centralized exchange hacks have resulted in over 8 Billions dollars' worth of tokens being stolen [51], out of which over 250 Million dollars' worth of tokens were stolen in 2019 alone. The main alternative is to use decentralized exchange services (*e.g.* [12] ) that do not hold tokens during the exchange process but are vulnerable to front-running attacks, since they make all orders public before they are finalized. This allows for illegal market manipulation, for example by leveraging the discrepancy between the most extreme buying and selling prices to buy tokens at the smallest offered price and immediately selling them at the highest accepted price.

Given the astounding volume of financial losses from centralized exchange hacks, constructing alternatives that are not vulnerable to token theft is clearly of great importance. However, ensuring that exchange orders remain private and avoiding front-running has also been identified as a chief concern [29,9], since this vulnerability reduces user trust and rules out regulatory compliance. In essence, a solution is needed for reconciling order privacy, market fairness and token security. In this work, we address the question:

*Can we securely & efficiently exchange cryptocurrency tokens while preserving order privacy, avoiding front-running and ensuring users never lose tokens?*

### 1.1 Our Contributions

We introduce universally composable privacy preserving decentralized exchanges immune to token theft and front-running, as well as optimizations to make our approach feasible in practice. Our main contributions are summarized as follows:

- **Privacy:** A provably secure privacy preserving decentralized exchange protocol, which is immune to both front-running and secret key theft.
- **Security:** An Universally Composable [17] analysis of our protocol, showing our approach is secure in real world settings.
- **Efficiency:** The first experimental results showing that Multiparty Computation on blockchains can be practical (*i.e.* faster than block finalization).
- **Usability:** An architecture that allows for deployment of a decentralized-exchange-as-a-service where users only need to do very lightweight computation and complete a *single* blockchain transfer in connection with a *single* round of efficient communication with the servers.

As the main building blocks of our work, we use publicly verifiable secure multiparty computation (MPC) [5] and threshold signatures with identifiable abort [39,38,20]. MPC allows for users to compute on private data without revealing this data to each other, which is a central concern in our solution. Moreover, using tools for public verifiability [4], it is possible to prove to any third party that a given computation output has been obtained without revealing inputs, which is paramount for proving validity in decentralized permissionless systems. Moreover, we use standard (public) smart contracts to implement a financial punishment system that incentivizes servers executing our protocol to behave honestly and ensures that users are reimbursed in case servers cheat.

If parties do not cheat, our protocol *requires no on-chain communication* except for the transactions needed to perform the exchanges and uses MPC only

for privately matching orders (avoiding front-running). We prove security against an actively dishonest majority and argue how clients can be refunded *even in case of total corruption*. We analyse our solution in the Universal Composability (UC) framework [17], which guarantees security even in complex situations where multiple protocols are executed concurrently with each other (*i.e.* real world scenarios as the Internet or decentralized cryptocurrency systems). To that end, we introduce a treatment of decentralized exchanges in the UC framework, allowing us to prove that our protocol is secure even in these realistic scenarios.

### 1.2 Our Techniques

***The protocol in a nutshell:*** Clients $\mathcal{C}_1, \ldots, \mathcal{C}_m$ wish to exchange tokens between ledgers $\mathcal{L}_a$ and $\mathcal{L}_b$ using servers $\mathcal{P}_1, \ldots, \mathcal{P}_n$ that facilitate the privacy-preserving exchange. Any number of ledgers can be involved, as long as all parties can use a standard smart contract (*e.g.* Ethereum) on a ledger $\mathcal{L}_{\mathsf{Ex}}$. Moreover, all ledger pairs $\mathcal{L}_a, \mathcal{L}_b$ must use cryptocurrency systems that allow for publicly proving that a double spend happened, *e.g.* Bitcoin UTXOs [49] (which can be emulated by attaching unique IDs to coins in account-based systems like Ethereum). Our protocol works as follows:

- **Smart Contract Setup:** The servers send a collateral deposit to a smart contract on ledger $\mathcal{L}_{\mathsf{Ex}}$ that guarantees that the servers do not cheat.
- **Off-chain communication:** After setup, *only off-chain protocol messages are exchanged* between the servers unless cheating happens, in which case cheating servers can be identified by publicly verifiable proofs [4] and punished.
- **Main Protocol Flow:** Performing exchanges between client $\mathcal{C}_i$ who wants to exchange tokens from ledger $\mathcal{L}_a$ held at their address $\mathcal{L}_i^{\mathsf{src}}$ with tokens from $\mathcal{C}_j$ who holds tokens in ledger $\mathcal{L}_b$ at address $\mathcal{L}_j^{\mathsf{src}}$ (or any other tuple of users/ledgers/addresses):
  1. *Burner address setup:* The servers set up threshold signature addresses $\mathtt{Addr}_i^{\mathsf{ex}}$ and $\mathtt{Addr}_j^{\mathsf{ex}}$ on each ledger $\mathcal{L}_a$ and $\mathcal{L}_b$.
  2. *Private order placement:* Clients transfer their tokens to server threshold addresses on each ledger (*i.e.* clients $\mathcal{C}_i$ and $\mathcal{C}_j$ transfer their tokens from addresses $\mathtt{Addr}_i^{\mathsf{src}}$ and $\mathtt{Addr}_j^{\mathsf{src}}$ on ledgers $\mathcal{L}_a$ and $\mathcal{L}_b$ to addresses $\mathtt{Addr}_i^{\mathsf{ex}}$ and $\mathtt{Addr}_j^{\mathsf{ex}}$, respectively) and send to the servers the addresses $\mathtt{Addr}_j^{\mathsf{trg}}$ and $\mathtt{Addr}_i^{\mathsf{trg}}$ on $\mathcal{L}_a$ and $\mathcal{L}_b$ where they will receive exchanged tokens if their orders match, respectively. They also send *secret shared* order information, describing the prices they charge for their tokens in a way that the servers do not learn the prices.
  3. *Confirmation:* If the servers have correctly received the secret orders and deposits from all the clients on each ledger, they proceed. Otherwise, they generate and send to the smart contract refund transactions transferring tokens from $\mathtt{Addr}_i^{\mathsf{ex}}$ and $\mathtt{Addr}_j^{\mathsf{ex}}$ back to client addresses $\mathtt{Addr}_i^{\mathsf{src}}$ and $\mathtt{Addr}_j^{\mathsf{src}}$, respectively. $\mathcal{C}_i$ and $\mathcal{C}_j$ retrieve these transactions and post them to $\mathcal{L}_a$ and $\mathcal{L}_b$, respectively.
  4. *Private Matching:* The servers execute a publicly verifiable MPC protocol (*e.g.* [5]) to run *any* order matching algorithm on secret-shared orders so that they can publicly prove that either a given set of orders have been matched or that a server has cheated, never learning non-matched orders.
  5. *Pay out:* Servers post final exchange operations to addresses $\mathtt{Addr}_j^{\mathsf{trg}}$ and $\mathtt{Addr}_i^{\mathsf{trg}}$ on $\mathcal{L}_a$ and $\mathcal{L}_b$ for matched order pairs.

– **Cheating Recovery:** The main cheating scenario is that a server sent an invalid message or failed to send a message. In that case, an honest server complains to the smart contract on $\mathcal{L}_{\mathsf{Ex}}$ and all servers have to send valid protocol messages to complete the protocol to the smart contract. If a server $\mathcal{P}_i$ does not send a valid message, it is identified as a cheater.
Any server $\mathcal{P}_i$ identified as a cheater loses its deposit to the smart contract, which is used to reimburse the clients and the honest servers for their work.

***Security Guarantees:*** Our main protocol achieves *security against an actively dishonest majority of servers* without requiring the clients to put up expensive collateral deposits, which is the case in previous approaches (*e.g.* [5]) where all parties must provide such deposits. Moreover, we describe a modification where even in a catastrophic failure where *all* servers become corrupted, even though the client's orders may leak, *all clients are guaranteed to be refunded* by the smart contract.

***Efficiency:*** Unless cheating happens, *all communication is off-chain* and the only information stored on-chain on any ledger are the transactions necessary to perform the exchange itself (improving on [5]). If cheating happens, the smart contract must identify and punish the cheater, but *this cost is covered by the cheater's deposit*. Moreover, *MPC is only used to match orders* in the Private Matching phase, while other operations are executed via efficient off-chain protocols. Finally, we do the first full implementation of the MPC component in a secure computation with financial incentives setting, showing that *MPC on blockchains is efficient in practice* (in particular for our matching application).

***Alternative Approaches:*** Our protocol can be modified in the following ways:

– **Incentivizing Servers:** Parties may pay exchange fees to servers so that it is profitable to execute a server.
– **Preventing Denial-of-Service attacks by Clients:** In our outlined protocol, either all clients transfer their money after registration or they all get reimbursed. Then, a client who registers but does not transfer funds could participate in a Denial-of-Service attack. We explain in Section 4.3 how to modify our protocol to avoid this.
– **Guaranteed Success with Honest Majority of Servers:** Assuming an honest majority of servers, we can obtain a much more efficient protocol by replacing the MPC protocol [5] used for the Private Matching with a much cheaper honest majority MPC protocol. Moreover, in this case we can achieve guaranteed output delivery, meaning that the *privacy preserving exchange always works* regardless of the minority of malicious servers.
– **Resilience under full corruption:** Even though we consider a dishonest majority where at least one server is honest, our technique can be modified for the setting where all servers may be corrupted. If users are allowed to register their orders (and destination addresses) on the smart contract, they can prove that all servers have misbehaved and get reimbursed with tokens from the exchange smart contract ledger, similarly to the approach of [35].

### 1.3 Related Work

*MPC with financial incentives:* A feature required by the MPC scheme in our applications is that if a cheating party obtains the output, then all the honest parties should do so as well (so all parties learn matching orders). Protocols which guarantee this are also called *fair* but known to be impossible to achieve with dishonest majorities [26]. Recently, [2,10] initiated a line of research that aims at incentivizing fairness in MPC by imposing cryptocurrency based financial penalties on misbehaving parties. Several works [47,46,11,8,5] improved the performance with respect to on-chain storage and the size of the collateral deposits from each party, while others obtained stronger notions of fairness [43,25]. None of these works implemented the MPC component of this approach.

In our work, we rely on such techniques to ensure that servers cannot profit from forcing exchange operations to fail. However, even the state-of-the-art [5] of these works only considers the single blockchain setting (not allowing for exchanges) and suffers from indefeasibly high overheads in both off-chain/on-chain complexity that would make exchange operations infeasible. We address these issues with an MPC protocol that operates on multiple blockchains, but building a decentralized exchange service where we only use MPC for matching orders; then later generating matched order transactions via an efficient threshold signature. We propose concrete improvements on the off-chain/on-chain overhead of [5] with the first concrete implementation of techniques from [4,6]. Furthermore, we achieve optimal communication (no more than in centralized exchanges) in the optimistic setting, where no party behaves maliciously. For the first time in this line of work [2,10,47,46,11,8,5] , *we fully implement the MPC component* of such a solution showing it is efficient in practice, whereas previous works only focused on on-chain efficiency (which is still optimal in our protocol).

*Privacy Preserving Smart Contracts:* Another related line of work [13,45,15] has focused on constructing privacy preserving smart contracts that can be checked for correct execution without revealing private inputs on-chain. However, these are intrinsically unfit for our application because they require a trusted party to learn all private inputs in order to generate zero-knowledge proofs showing that a given computation output was obtained. This would allow a corrupted trusted party (or insecure SGX enclave [14]) to perform front running (or even steal funds), *i.e.* the same issues of centralized exchanges.

*Distributed markets and exchanges:* The use of MPC in traditional stock market exchanges has been considered in [48,22,23] but these works focus on matching stock market orders and do not address the issue of ensuring that exchange transactions are performed correctly, which we do. Many commercial decentralized exchange services (*e.g.* [12]) exist, but they are not private and suffer from front-running as discussed before. A front-running resistant approach is suggested in [9] but it relies on insecure trusted hardware [14] and has no privacy.

*Fair Two-Party Data Exchange:* Dziembowski *et al.* [35] showed how to use financial incentives and a proof of cheating to enforce honest behaviour when two parties are exchanging pre-images of a hash function using a distributed ledger.

Despite showing security in UC, their approach would not directly be efficient for cross-chain token exchange, nor generalize to exchange order matching. Furthermore, their approach require posting auxiliary information to the ledger, however small, even in the optimistic case.

## 2 Preliminaries

Let $y \xleftarrow{\$} F(x)$ denote running the randomized algorithm $F$ with input $x$ and implicit randomness, and obtaining the output $y$. Similarly, $y \leftarrow F(x)$ is used for a deterministic algorithm. For a set $\mathcal{X}$, let $x \xleftarrow{\$} \mathcal{X}$ denote $x$ chosen uniformly at random from $\mathcal{X}$. $\tau$ denotes the computational and $\kappa$ the statistical security parameter.

### 2.1 (Global) Universal Composability and the Global Clock

In this work, the (Global) Universal Composability or (G)UC framework [17,19] is used to analyze security. Due to space constraints, we refer interested readers to the aforementioned works for more details. We generally use $\mathcal{F}$ to denote an ideal functionality and $\Pi$ for a protocol. Several functionalities in this work allow *public verifiability*. To model this, we follow the approach of Badertscher *et al.* [3] and allow the set of verifiers $\mathcal{V}$ to be dynamic by adding register and de-register instructions as well as instructions that allow $\mathcal{S}$, the adversary controlling a set of corrupted parties, to obtain the list of registered verifiers. As some parts of our work are inherently synchronous, we model the different "rounds" of it using a global clock functionality $\mathcal{F}_{\mathsf{Clock}}$ as in [3,43,42]. See Appendix A.1 for a definition of the verifier registration interfaces and Appendix A.2 for a definition of $\mathcal{F}_{\mathsf{Clock}}$. Throughout this work, we will write "update $\mathcal{F}_{\mathsf{Clock}}$" as a short-hand for "send (UPDATE, $sid$) to $\mathcal{F}_{\mathsf{Clock}}$".

### 2.2 Client-input MPC with Publicly Verifiable Output

We focus on MPC with security against a static, rushing and malicious adversary $\mathcal{A}$ corrupting up to $n-1$ of the $n$ servers where $m$ clients provide the actual inputs and where all clients might be malicious. This specific setting of MPC is called *out-sourced MPC* and can efficiently be realized in a black box manner on top of a "standard" MPC scheme where the servers are providing the output [41]. We let the MPC functionality compute the result $\mathbf{y}$ and share its output in a verifiable way such that any potential verifier can either check that the output is correct or identify a cheater, and hence allow for incentivized fairness. In Appendix A.3 we formally define functionality $\mathcal{F}_{\mathsf{Ident}}$ adapted from [5] that captures this style of MPC. We remark that differently from [5], we use a publicly verifiable version [4] of the original protocol of [5] with optimizations from [6] where no homomorphic commitments are needed and, in case no cheating is detected, no interaction with the smart contract is needed apart from the initial deposits from servers executing $\mathcal{F}_{\mathsf{Ident}}$ and the final output. Intuitively it specified an out-sourced MPC functionality where clients $\mathcal{C}_1, \ldots, \mathcal{C}_m$ supply private input that is computed on

in MPC by the servers $\mathcal{P}_1, \ldots, \mathcal{P}_n$ and where the output of the computation is verifiably shared between the servers in such a manner that the shares can verified by an external verifier $\mathcal{V}$ after the completion of the protocol to identify any potential malicious behaviour.

### 2.3 (Threshold) Signatures

In our work we rely on signatures and identifiable threshold signatures to represent transactions on ledgers. Therefore we will assume the existence of two UC functionalities: $\mathcal{F}_{\mathsf{Sig}}$ which is a standard functionality in UC [18] (with key generation, signature generation and signature verification), as well as our own formalization of general UC identifiable threshold signatures $\mathcal{F}_{\mathsf{TSig}}$ (which can be seen as a generalization of signatures such as [21,20]). In comparison to normal signatures, $\mathcal{F}_{\mathsf{TSig}}$ has a two-step process of signature generation, where the parties first generate shares $\rho$ of the overall signature $\sigma$ which later on are aggregated in a share combination phase. This share combination also exposes parties that generated some shares wrongly. Additionally, $\mathcal{F}_{\mathsf{TSig}}$ also creates the signing key in a distributed way. We treat both $\mathcal{F}_{\mathsf{Sig}}$ and $\mathcal{F}_{\mathsf{TSig}}$ as global UC functionalities, which means that both local and other global UC functionalities can verify signatures on them. This is meaningful as we assume that ledgers are global functionalities too, hence validating their transactions should be consistent among any different session. Functionalities $\mathcal{F}_{\mathsf{Sig}}, \mathcal{F}_{\mathsf{TSig}}$ are presented in Appendices A.4 and A.5, respectively.

### 2.4 Representing Cryptocurrency Transactions

In order to focus on the novel aspects of our protocol, we represent cryptocurrency transactions under a simplified version of the Bitcoin UTXO model [49]. For the sake of simplicity we only consider operations of the "Pay to Public Key" (P2PK) type, even though any other types of transaction can be supported as long as it is possible to publicly prove that a double spend happened and to generate transactions in a distributed manner. In particular, even a privacy preserving cryptocurrency that publicly reveals double spends could be integrated to our approach by constructing a specific purpose multiparty computation protocol for generating transactions in that cryptocurrency. *Representing Addresses:* An address $\mathtt{Addr} = \mathsf{vk}$ is simply a signature verification key, where $\mathsf{vk}$ and subsequent signatures $\sigma$ are generated by the signature scheme used in the cryptocurrency (represented by $\mathcal{F}_{\mathsf{Sig}}$ and $\mathcal{F}_{\mathsf{TSig}}$).

*Representing Transactions:* We represent a transaction in our simplified UTXO model by the tuple $\mathsf{tx} = (\mathtt{id}, \mathsf{In}, \mathsf{Out}, \mathsf{Sig})$, where $\mathtt{id} \in \{0,1\}^\tau$ is a unique transaction identification, $\mathsf{In} = \{(\mathtt{id}_1, \mathtt{in}_1), \ldots, (\mathtt{id}_m, \mathtt{in}_m)\}$ is a set of pairs of previous transaction id's $\mathtt{id} \in \{0,1\}^\tau$ and their values $\mathtt{in} \in \mathbb{N}$, $\mathsf{Out} = \{(\mathtt{out}_1, \mathtt{Addr}_1), \ldots, (\mathtt{out}_n, \mathtt{Addr}_n)\}$ is a set of pairs of values $\mathtt{out} \in \mathbb{N}$ and signature verification keys $\mathtt{Addr}$ and $\mathsf{Sig} = \{\sigma_1, \ldots, \sigma_m\}$ is a set of signatures $\sigma$.

*Transaction Validity:* A transaction $\mathsf{tx} = (\mathtt{id}, \mathsf{In}, \mathsf{Out}, \mathsf{Sig})$ is considered valid if, for all $(\mathtt{id}_i, \mathtt{in}_i) \in \mathsf{In}$ and $(\mathtt{out}_j, \mathtt{Addr}_j) \in \mathsf{Out}$, the following conditions hold:

– There exists a valid transaction $\mathsf{tx}_i = (\mathtt{id}_i, \mathsf{In}_i, \mathsf{Out}_i, \mathsf{Sig}_i)$ in the public ledger (in our case represented by $\mathcal{F}_{\mathsf{BB}}$ and $\mathcal{F}_{\mathsf{SC}}$) such that $(\mathtt{in}_i, \mathtt{Addr}_i) \in \mathsf{Out}_i$.
– There exists $\sigma_i \in \mathsf{Sig}$ such that $\sigma_i$ is a valid signature of $\mathtt{id}|\mathsf{In}|\mathsf{Out}$ under $\mathtt{Addr}_i$ according to the cryptocurrency's signature scheme ($\mathcal{F}_{\mathsf{Sig}}$ or $\mathcal{F}_{\mathsf{TSig}}$).
– It holds that $\sum_{i=1}^{m} \mathtt{in}_i = \sum_{j=1}^{n} \mathtt{out}_j$.

*Generating Transactions:* A party controlling the corresponding signing keys for valid UTXO addresses $\mathtt{Addr}_1, \ldots, \mathtt{Addr}_m$ containing values $\mathtt{in}_1, \ldots, \mathtt{in}_m$ can generate a transaction that transfers the funds in these addresses to output addresses $\mathtt{Addr}_{out,1}, \ldots, \mathtt{Addr}_{out,n}$ by proceeding as follows:

1. Choose a unique $\mathtt{id} \in \{0,1\}^{\tau}$.
2. Choose values $\mathtt{out}_1, \ldots, \mathtt{out}_n$ such that $\sum_{i=1}^{m} \mathtt{in}_i = \sum_{j=1}^{n} \mathtt{out}_j$.
3. Generate $\mathsf{In}, \mathsf{Out}$ as described above and sign $\mathtt{id}|\mathsf{In}|\mathsf{Out}$ with the instances of $\mathcal{F}_{\mathsf{Sig}}$ or $\mathcal{F}_{\mathsf{TSig}}$ corresponding to $\mathtt{Addr}_1, \ldots, \mathtt{Addr}_m$, obtaining $\mathsf{Sig} = \{\sigma_1, \ldots, \sigma_m\}$.
4. Output $\mathsf{tx} = (\mathtt{id}, \mathsf{In}, \mathsf{Out}, \mathsf{Sig})$.

### 2.5 Bulletin Boards and Smart Contracts

Our approach does not require dealing with the specifics of blockchain consensus, since we hold tokens in addresses controlled by threshold signatures in such a way that transactions can only be issued when all servers cooperate. Since at least one server is assumed to be honest, we do not have to confirm whether a transaction is registered. Even when we address the case of total server corruption, our approach only requires identifying an attempt of double spending a client's deposit regardless of which transaction of the double spends gets finalized on the ledger. Hence, in this work we do neither fully formalize distributed ledgers as was done in previous work [3], nor do we use other existing simplified formulations such as [43]. Instead, for the sake of simplicity, we use a public bulletin board functionality $\mathcal{F}_{\mathsf{BB}}$ that is presented in Appendix A to represent ledgers.

## 3 Modeling Fair Decentralized Exchanges in UC

In this section we formalize a decentralized exchange on a high level. We assume that there are $m$ clients $\mathcal{C}_1, \ldots, \mathcal{C}_m$. These clients can exchange between $\ell$ ledgers $\mathcal{L}_1, \ldots, \mathcal{L}_\ell$. Each $\mathcal{C}_j$ controls an amount of tokens $\mathsf{am}_j^{\mathsf{src}}$ in an address $\mathsf{vk}_j^{\mathsf{src}}$ which is on ledger $\mathcal{L}_j^{\mathsf{src}}$. The goal of $\mathcal{C}_j$ is to acquire $\mathsf{am}_j^{\mathsf{trg}}$ tokens on ledger $\mathcal{L}_j^{\mathsf{trg}}$ which should be transferred to address $\mathsf{vk}_j^{\mathsf{trg}}$.

In order to compute transactions there exist two deterministic poly-time algorithms `compSwap` and `makeTX`. On a high level, `compSwap` takes the exchange order of each client between two specific ledgers, $\mathcal{L}_a$ and $\mathcal{L}_b$, as input and returns the order matches. Whereas `makeTX` takes as input the order matches computed by `compSwap` for *each* possible pair of ledgers along with some metadata and returns a list of all, unsigned, transactions to be carried out to complete the

exchange over all ledgers. More concretely, a bit $\delta_j$ indicates for each client $\mathcal{C}_j$ whether said client wants to buy $\mathsf{am}_j^{\mathsf{src}}$ tokens on $\mathcal{L}_a$ using *at most* $\mathsf{am}_j^{\mathsf{trg}}$ tokens from $\mathcal{L}_b$, or if they want to sell $\mathsf{am}_j^{\mathsf{src}}$ tokens from $\mathcal{L}_a$ for *at least* $\mathsf{am}_j^{\mathsf{trg}}$ tokens on $\mathcal{L}_b$. `compSwap` then returns a list of transfer orders. More concretely, a list of tuples where each tuple contains two quantities, $\mathsf{am}_j$ and $\mathsf{am}_{j'}$ and the identifiers of two clients; client $\mathcal{C}_j$ who should have $\mathsf{am}_j$ of its tokens transferred on $\mathcal{L}_a$ to the other client $\mathcal{C}_{j'}$ and client $\mathcal{C}_{j'}$ should transfer $\mathsf{am}_{j'}$ of its tokens on $\mathcal{L}_b$ to $\mathcal{C}_j$. As our protocol will use individual burner addresses for each $\mathcal{C}_j$ controlled by the servers, we assume for the computation that the asset was transferred by $\mathcal{C}_j$ to an address $\mathsf{vk}_j^{\mathsf{ex}}$ using a transaction with id $\mathtt{id}_j$.

`compSwap` computes swaps between two ledgers. It takes as input $m' \leq m$ tuples of the form $(\mathcal{C}_j, \delta_j, \mathsf{am}_j^{\mathsf{src}}, \mathsf{am}_j^{\mathsf{trg}})$ where $\mathcal{C}_j, \mathsf{am}_j^{\mathsf{src}}, \mathsf{am}_j^{\mathsf{trg}} \in \mathbb{N}$ where $\delta_j = 0$ if $\mathcal{C}_j$ wants to swap from the first to the second and $\delta_j = 1$ if it wants to swap from the second to the first ledger. It outputs a list of tuples $(\mathcal{C}_j, \mathsf{am}_j, \mathcal{C}_{j'}, \mathsf{am}_{j'})$, where $\mathsf{am}_j, \mathsf{am}_{j'} \in \mathbb{N}$, which is viewed as a vector $\mathbf{y}^{a,b} \in \mathbb{N}^g$ for some $g$.

`makeTX` takes the $\ell \cdot (\ell - 1)$ outputs of `compSwap` for each pair of ledgers $\mathcal{L}_a, \mathcal{L}_b$ with $1 \leq a < b \leq \ell$ as well as a tuple $(\mathcal{C}_j, \mathcal{L}_j^{\mathsf{src}}, \mathcal{L}_j^{\mathsf{trg}}, \mathtt{id}_j, \mathsf{vk}_j^{\mathsf{ex}}, \mathsf{vk}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{trg}}, \mathsf{am}_j^{\mathsf{src}})$ for each $\mathcal{C}_j$ as input. It then outputs $\ell$ transaction orders $(\mathtt{id}_a, \mathsf{In}_a, \mathsf{Out}_a)$ (still missing the signatures however), one for each $\mathcal{L}_a$, such that

1. $\mathsf{In}_a = \{(\mathtt{id}_j, \mathsf{am}_j^{\mathsf{src}})\}$ for some $\mathcal{C}_j$ where $\mathcal{L}_j^{\mathsf{src}} = \mathcal{L}_a$ and $\mathcal{C}_j$ transferred the amount $\mathsf{am}_j^{\mathsf{src}}$ to $\mathsf{vk}_j^{\mathsf{ex}}$ in a transaction with id $\mathtt{id}_j$.
2. $\mathsf{Out}_a = \{(\mathtt{out}_a^i, \mathtt{Addr}_a^i)\}$ where each $\mathtt{Addr}_a^i$ is either $\mathsf{vk}_j^{\mathsf{src}}$ for $\mathcal{C}_j$ with $\mathcal{L}_j^{\mathsf{src}} = \mathcal{L}_a$ or $\mathsf{vk}_{j'}^{\mathsf{trg}}$ for $\mathcal{C}_{j'}$ with $\mathcal{L}_{j'}^{\mathsf{trg}} = \mathcal{L}_a$.
3. $\sum_j \mathtt{in}_a^j = \sum \mathtt{out}_a^i$ and $\mathtt{id}_a$ is computed as a hash of $\mathsf{In}_a, \mathsf{Out}_a$.

Further discussion on our matching algorithm `compSwap` is presented in Appendix D. Notice, however, that our approach is not limited to this algorithm, since our MPC component can accommodate any arbitrary matching algorithm.

Algorithms `compSwap` and `makeTX` only model the generation of UTXO-style transaction descriptions that can later be turned into valid transactions by creating a signature $\mathsf{Sig}_a$ for each $\mathcal{L}_a$. The exchange *security* requirements are then modeled in the functionality (*e.g.* requiring that e.g. each $\mathcal{C}_j$ either gets its asset back or also an asset on the other ledger according to some matching rule of transactions). The exchange functionality $\mathcal{F}_{\mathsf{EX}}$ as well as the protocol $\Pi_{\mathsf{Ex}}$ will later use the algorithms `compSwap`, `makeTX` to generate the transactions. Looking ahead we note that `compSwap` will be computed using outsourced MPC through $\mathcal{F}_{\mathsf{Ident}}^{a,b}$, keeping the input $\mathsf{am}_j^{\mathsf{src}}, \mathsf{am}_j^{\mathsf{trg}}$ from each client hidden from every server whereas `makeTX` will be computed openly. We furthermore note that since there is always an honest party that can influence the choice of algorithms so that the chosen algorithms are fair.

**Functionality $\mathcal{F}_{\mathsf{EX}}$** (Part 1)

$\mathcal{F}_{\mathsf{EX}}$ interacts with the set of servers $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, a set of $m$ clients $\mathcal{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$ as well as the global functionality $\mathcal{F}_{\mathsf{Clock}}$ to which it is registered. $\mathcal{F}_{\mathsf{EX}}$ is parameterized by the compensation amount $q$ and the security deposit $d = mq$. $\mathcal{S}$ specifies an adversary controlling a set of corrupted servers $I \subset [n]$ and clients $J \subseteq \mathcal{C}$. Let $\mathcal{L}_1, \ldots, \mathcal{L}_\ell$ be the available ledgers.

**Init:** Upon input $(\textsc{Input}, sid, \mathsf{coins}(d))$ by each honest $\mathcal{P}_i$:

1. Send $(\textsc{keygen}, sid)$ to $\mathcal{F}_{\mathsf{TSig}}$. If $\mathcal{F}_{\mathsf{TSig}}$ returns a key $\mathsf{vk}$ then save it internally and continue, if it instead aborts then return $\mathsf{coins}(d)$ to each honest $\mathcal{P}_i$ and stop.
2. Update $\mathcal{F}_{\mathsf{Clock}}$ and wait for a message from $\mathcal{S}$. If $\mathcal{S}$ sends $\mathsf{coins}(|I| \cdot d)$ send $(\textsc{key}, sid, \mathsf{vk})$ to each honest $\mathcal{P}_i$, otherwise reimburse all honest $\mathcal{P}_i$ with $\mathsf{coins}(d)$.

**Enroll Client:** Upon input $(\textsc{Enroll}, sid, \mathcal{L}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{src}}, \mathsf{am}_j^{\mathsf{src}}, \mathcal{L}_j^{\mathsf{trg}}, \mathsf{vk}_j^{\mathsf{trg}})$ from a client $\mathcal{C}_j \in \mathcal{C} \setminus J$ or $\mathcal{S}$ if $\mathcal{C}_j \in J$ and if **Init** finished:

1. If $\mathcal{C}_j$ was enrolled already then return.
2. Send $(\textsc{Enroll}, sid, \mathcal{L}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{src}}, \mathsf{am}_j^{\mathsf{src}}, \mathcal{L}_j^{\mathsf{trg}}, \mathsf{vk}_j^{\mathsf{trg}})$ to $\mathcal{S}$. If $\mathcal{S}$ sends $(\textsc{Ok}, sid)$ then send $(\textsc{keygen}, sid)$ to $\mathcal{F}_{\mathsf{TSig}}$ to generate $\mathsf{vk}_j^{\mathsf{ex}}$ as well as $(\textsc{sign}, sid, m)$ (obtaining $\rho$) and $(\textsc{combine}, sid, m, \rho, \mathsf{vk})$ to obtain $\sigma$.
3. If $\mathcal{F}_{\mathsf{TSig}}$ aborts then send $(\textsc{Abort}, sid)$ to $\mathcal{C}_j$. If it did not abort then send $(\textsc{Proceed?}, sid, \sigma, \mathsf{vk}_j^{\mathsf{ex}})$ to the caller. If the caller was an honest $\mathcal{C}_j$ then wait for input $(\textsc{value}, sid, \mathsf{am}_j^{\mathsf{trg}}, \mathtt{id}_j)$ where $\mathtt{id}_j$ is a transaction id of an honestly generated transaction on $\mathcal{L}_j^{\mathsf{src}}$ from $\mathsf{vk}_j^{\mathsf{src}}$ to $\mathsf{vk}_j^{\mathsf{ex}}$ with amount $\mathsf{am}_j^{\mathsf{src}}$. Send $(\textsc{InputProvided}, sid, \mathcal{C}_j)$ to $\mathcal{S}$.

**Exchange:** Upon input $(\textsc{Exchange}, sid)$ by all honest $\mathcal{P}_i$ and if **Init** finished:

1. Send $(\textsc{Clients?}, sid)$ to $\mathcal{S}$ and wait for $\mathsf{am}_j^{\mathsf{src}}, \mathtt{id}_j$ for each $\mathcal{C}_j \in J$ or alternatively for a message that $\mathcal{C}_j$ did not transfer to $\mathsf{vk}_j^{\mathsf{ex}}$.
2. Send $(\textsc{Confirm}, sid, \{\mathsf{am}_i^{\mathsf{src}}, \mathtt{id}_i\}_{i\in[m]})$ to all servers in $\mathcal{P}$. If no honest server $\mathcal{P}_i$ for $i \notin I$ answers with $(\textsc{Confirmed}, sid, \{\mathsf{am}_i^{\mathsf{src}}, \mathtt{id}_i\}_{i\in[m]})$ (meaning a client did not make a transaction) enter **Abort without Output**.
3. Send $(\textsc{sign}, sid, \mathcal{C}_j|\mathtt{id}_j, \mathsf{vk})$ (receiving $\rho_j$) and $(\textsc{combine}, sid, \mathcal{C}_j|\mathtt{id}_j, \rho_j, \mathsf{vk})$ to $\mathcal{F}_{\mathsf{TSig}}$ for each $\mathcal{C}_j \in \mathcal{C}$. If any of them aborts then enter **Abort without Output**.
4. For each $a, b \in [\ell]$, $1 \le a < b \le \ell$ let $\mathcal{C}^{a,b}$ be the clients swapping between $\mathcal{L}_a$ and $\mathcal{L}_b$. For $\mathcal{C}_j \in \mathcal{C}^{a,b}$ define the $m' \le m$ values $h_j = (\mathcal{C}_j, \delta_j, \mathsf{am}_j^{\mathsf{src}}, \mathsf{am}_j^{\mathsf{trg}})$ where $\delta_j$ is 0 if $\mathcal{L}_j^{\mathsf{src}} = \mathcal{L}_a$ and 1 otherwise. Compute $\mathbf{y}^{a,b} \leftarrow \mathtt{compSwap}(h_1, \ldots, h_{m'})$. If $\mathcal{S}$ sends $(\textsc{Abort}, sid)$ enter **Abort without Output**.

Fig. 1: Functionality $\mathcal{F}_{\mathsf{EX}}$ for Secure Decentralized Exchange.

**Functionality $\mathcal{F}_{\mathsf{EX}}$** (Part 2)

**Open:** Upon input (OPEN, *sid*) by each honest $\mathcal{P}_i$ and if **Exchange** finished:

1. Send (SIGN, $sid, open, \mathsf{vk}$) to $\mathcal{F}_{\mathsf{TSig}}$ and obtain $\rho$. Then send (COMBINE, $sid, open, \rho, \mathsf{vk}$) to $\mathcal{F}_{\mathsf{TSig}}$. If $\mathcal{F}_{\mathsf{TSig}}$ aborted then enter **Abort without Output**, otherwise send (OUTPUTS, $sid, \{\mathbf{y}^{a,b}\}$) to $\mathcal{S}$.
2. Locally compute for each $\mathcal{L}_a$ the values $t_a = (\mathtt{id}_a, \mathsf{In}_a, \mathsf{Out}_a)$ using `makeTX`. Let the length of $\mathsf{In}_a$ be $r$ and denote the burner addresses as $\mathsf{vk}_c^{\mathsf{ex}}$ for $c \in [r]$.
3. For each $\mathcal{L}_a$ and $c \in [r]$ send (SIGN, $sid, t_a, \mathsf{vk}_c^{\mathsf{ex}}$) to $\mathcal{F}_{\mathsf{TSig}}$ to obtain $\rho_a^c$. Then send (COMBINE, $sid, t_a, \rho_a^c, \mathsf{vk}_c^{\mathsf{ex}}$) to $\mathcal{F}_{\mathsf{TSig}}$. If $\mathcal{F}_{\mathsf{TSig}}$ aborted enter **Abort with Output**.
4. Send (SIGN, $sid, done, \mathsf{vk}$) to $\mathcal{F}_{\mathsf{TSig}}$ to obtain $\rho$. Then send (COMBINE, $sid, done, \rho, \mathsf{vk}$) to $\mathcal{F}_{\mathsf{TSig}}$. If $\mathcal{F}_{\mathsf{TSig}}$ aborted then enter **Abort with Output**.
5. Update $\mathcal{F}_{\mathsf{Clock}}$. Then send (SUCCESS, $sid, \mathsf{tx}_a$) to each $\mathcal{C}_j$ with $\mathcal{L}_a \in \{\mathsf{vk}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{trg}}\}$, send $\mathsf{coins}(d)$ back to each honest $\mathcal{P}_i$ and $\mathsf{coins}(|I| \cdot d)$ to $\mathcal{S}$.

**Abort with Output:**

1. Update $\mathcal{F}_{\mathsf{Clock}}$ twice and send (ABORT?, $sid$) to $\mathcal{S}$. If $\mathcal{S}$ responds with (ABORT, $sid, I_1$) where $\emptyset \neq I_1 \subseteq I$ then update $\mathcal{F}_{\mathsf{Clock}}$. Afterwards send $\mathsf{coins}(d)$ to each honest $\mathcal{P}_i$, $\mathsf{coins}((|I| - |I_1|)d)$ to $\mathcal{S}$ and $\mathsf{coins}(|I'|d/m)$ to each client $\mathcal{C}_j$.
2. If $\mathcal{S}$ did not abort then update $\mathcal{F}_{\mathsf{Clock}}$. Afterwards for each $\mathcal{L}_a$ and $c \in [r]$ send (SIGN, $sid, t_a, \mathsf{vk}_c^{\mathsf{ex}}$) to $\mathcal{F}_{\mathsf{TSig}}$ to obtain $\rho_a^c$ (with the same notation as **Open**). Then send (COMBINE, $sid, t_a, \rho_a^c, \mathsf{vk}_c^{\mathsf{ex}}$) to $\mathcal{F}_{\mathsf{TSig}}$. If $\mathcal{F}_{\mathsf{TSig}}$ aborted overall with the set of parties $I_2 \subset I$ cheating update $\mathcal{F}_{\mathsf{Clock}}$. Afterwards send $\mathsf{coins}(d)$ to each honest $\mathcal{P}_i$, $\mathsf{coins}((|I| - |I_2|)d)$ to $\mathcal{S}$, $\mathsf{coins}(|I'|d/m)$ to each client $\mathcal{C}_j$ and return.
3. Otherwise update $\mathcal{F}_{\mathsf{Clock}}$, compute $\mathsf{tx}_a$ for each $\mathcal{L}_a$, send (SUCCESS, $sid, \mathsf{tx}_a$) to each $\mathcal{C}_j$ with $\mathcal{L}_a \in \{\mathsf{vk}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{trg}}\}$, $\mathsf{coins}(d)$ to each honest $\mathcal{P}_i$ and $\mathsf{coins}(|I| \cdot d)$ to $\mathcal{S}$.

**Abort without Output:**

1. Update $\mathcal{F}_{\mathsf{Clock}}$ twice.
2. Send (ABORTC?, $sid$) to $\mathcal{S}$ which responds with (ABORTC, $sid, J_1, \{\mathtt{id}_j\}_{j \in J \setminus J_1}$) where $J_1 \subseteq J$. Then send (ABORT?, $sid$) to $\mathcal{S}$. If $\mathcal{S}$ responds with (ABORT, $sid, I_1$) where $\emptyset \neq I_1 \subseteq I$ update $\mathcal{F}_{\mathsf{Clock}}$, send $\mathsf{coins}(d)$ to each honest $\mathcal{P}_i$, send $\mathsf{coins}((|I| - |I_1|)d)$ to $\mathcal{S}$ and $\mathsf{coins}(|I'|d/(m - |J_1|))$ to each $\mathcal{C}_j \in \mathcal{C} \setminus J_1$.
3. If $\mathcal{S}$ did not abort then update $\mathcal{F}_{\mathsf{Clock}}$. Afterwards for each $\mathcal{C}_j$ compute $t_j = (\mathtt{id}_j, \mathsf{In}_j, \mathsf{Out}_j)$ by setting $\mathsf{In}_j = (\mathtt{id}_j, \mathsf{am}_j^{\mathsf{src}})$ and $\mathsf{Out}_j = (\mathsf{am}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{src}})$ and determining $\mathtt{id}_j$ as the hash of $\mathsf{In}_j, \mathsf{Out}_j$.
4. For each $\mathcal{C}_j$ send (SIGN, $sid, t_j, \mathsf{vk}_j^{\mathsf{ex}}$) to $\mathcal{F}_{\mathsf{TSig}}$ and obtain $\rho_j$. Then send (COMBINE, $sid, t_j, \rho_j, \mathsf{vk}_j^{\mathsf{ex}}$) to $\mathcal{F}_{\mathsf{TSig}}$. If $\mathcal{F}_{\mathsf{TSig}}$ aborted overall with the set of parties $I_2 \subset I$ cheating then update $\mathcal{F}_{\mathsf{Clock}}$. Afterwards send $\mathsf{coins}(d)$ to each honest $\mathcal{P}_i$, $\mathsf{coins}((|I| - |I_2|)d)$ to $\mathcal{S}$ and $\mathsf{coins}(|I_2|d/m)$ to each client $\mathcal{C}_j$.
5. If $\mathcal{F}_{\mathsf{TSig}}$ did not abort update $\mathcal{F}_{\mathsf{Clock}}$. Compute $\mathsf{tx}_j$ for each $\mathcal{C}_j$ and send (REIMBURSE, $sid, \mathsf{tx}_j$) to $\mathcal{C}_j$, $\mathsf{coins}(d)$ to each honest $\mathcal{P}_i$ and $\mathsf{coins}(|I| \cdot d)$ to $\mathcal{S}$.

Fig. 2: Functionality $\mathcal{F}_{\mathsf{EX}}$ for Secure Decentralized Exchange.

### 3.1 The Fair Exchange Functionality

Functionality[4] $\mathcal{F}_{\mathsf{EX}}$ as depicted in Fig. 1 & 2 generates the transactions for clients via burner addresses. As our protocol later uses the global clock, $\mathcal{F}_{\mathsf{EX}}$ also accesses $\mathcal{F}_{\mathsf{Clock}}$. The transactions that $\mathcal{F}_{\mathsf{EX}}$ will generate must be verifiable on other ledgers and its signatures therefore come from a global threshold signing functionality $\mathcal{F}_{\mathsf{TSig}}$. $\mathcal{F}_{\mathsf{EX}}$ accesses $\mathcal{F}_{\mathsf{TSig}}$ in order to generate these, giving the adversary $\mathcal{S}$ extra influence in the process. Notice that $\mathcal{F}_{\mathsf{EX}}$ implements a fair secure swap given that `compSwap` and `makeTX` implement swapping algorithms. $\mathcal{S}$ only learns the swap data $\mathbf{y}^{a,b}$ after all transactions were determined but does not learn the input orders like a centralized exchange does. $\mathcal{S}$ can only cause an abort before the output is released by causing signature generation to fail. In this case, all clients get reimbursed with either their original assets or with a deposit (here $\mathsf{coins}$) from $\mathcal{S}$. If the functionality progressed far enough that $\mathcal{S}$ learned the output, then it can only abort by losing its $\mathsf{coins}$. Otherwise, the swap output transactions will always be given to the clients.

## 4 Realizing the Exchange Functionality

We now describe a protocol $\Pi_{\mathsf{Ex}}$ that GUC-realizes $\mathcal{F}_{\mathsf{EX}}$, making sketch from Section 1.2 more precise. As the smart contract which is used, abstractly described in $\mathcal{F}_{\mathsf{SC}}$ (see Appendix B), is rather complex, we outline the interplay between $\Pi_{\mathsf{Ex}}$ and $\mathcal{F}_{\mathsf{SC}}$ beforehand. Due to space limitations, the smart contract description and its formalization $\mathcal{F}_{\mathsf{SC}}$ as well as the full protocol $\Pi_{\mathsf{Ex}}$ can be found in Appendix B.

### 4.1 Overview of the Protocol

$\Pi_{\mathsf{Ex}}$ runs between $n$ servers $\mathcal{P}$ and $m$ clients $\mathcal{C}$. The clients can exchange between $\ell$ ledgers $\mathcal{L}$. Each $\mathcal{C}_j$ controls an amount of tokens $\mathsf{am}_j^{\mathsf{src}}$ in an address $\mathsf{vk}_j^{\mathsf{src}}$ on ledger $\mathcal{L}_j^{\mathsf{src}}$. The goal of $\mathcal{C}_j$ is to acquire $\mathsf{am}_j^{\mathsf{trg}}$ tokens on $\mathcal{L}_j^{\mathsf{trg}}$ in address $\mathsf{vk}_j^{\mathsf{trg}}$.

***Smart Contract Setup.*** Initially, no servers or clients are registered anywhere. The servers $\mathcal{P}$ run a pre-processing step where they use $\mathcal{F}_{\mathsf{TSig}}$ to sample a common key $\mathsf{vk}$ for threshold-signing. Each $\mathcal{P}_i$ also sends its individual verification key $\widehat{\mathsf{vk}}_i$ to all other servers. Finally the servers set up $\ell \cdot (\ell - 1)$ instances $\mathcal{F}_{\mathsf{Ident}}^{a,b}$ to accept inputs by clients who want to transfer between $\mathcal{L}_a$ and $\mathcal{L}_b$.

To initiate the protocol each server sends $\mathsf{vk}, \widehat{\mathsf{vk}}_1, \ldots, \widehat{\mathsf{vk}}_n, \mathsf{coins}(d)$ to $\mathcal{F}_{\mathsf{SC}}$. $\mathcal{F}_{\mathsf{SC}}$ will wait for a certain period and then check if all servers put in enough deposit and signed messages consistently. If so, then the coins are locked and $\mathcal{F}_{\mathsf{SC}}$ transitions to a state `ready`, otherwise the servers are reimbursed and $\mathcal{F}_{\mathsf{SC}}$ goes back to its initial state `init`.

[4] Throughout this work, we treat $\mathcal{F}_{\mathsf{EX}}$ as an ordinary UC functionality and not a global functionality (which would intuitively make more sense). This is due to subtle issues that would arise in the proof if $\mathcal{F}_{\mathsf{EX}}$ was global, namely the simulator would not be able to equivocate the necessary outputs.

***1. Burner Address Setup.*** If a client $\mathcal{C}_j$ wants to exchange an asset from $\mathcal{L}_j^{\mathsf{src}}$ to $\mathcal{L}_j^{\mathsf{trg}}$ it checks if $\mathcal{F}_{\mathsf{SC}}$ is in state `ready`. If so then $\mathcal{C}_j$ sends a registration message to all $\mathcal{P}$, who generate a burner address $\mathsf{vk}_j^{\mathsf{ex}}$ on $\mathcal{L}_j^{\mathsf{src}}$ using $\mathcal{F}_{\mathsf{TSig}}$. The servers sign the client's data and $\mathsf{vk}_j^{\mathsf{ex}}$ using $\mathsf{sk}$ and send the signature to $\mathcal{C}_j$.

***2. Private Order Placement.*** If the signature is correct, then $\mathcal{C}_j$ transfers $\mathsf{am}_j^{\mathsf{src}}$ from $\mathsf{vk}_j^{\mathsf{src}}$ to $\mathsf{vk}_j^{\mathsf{ex}}$ and inputs its transfer information into the correct $\mathcal{F}_{\mathsf{Ident}}^{a,b}$.

***3. Confirmation.*** Servers wait until all transfers to burner addresses were made and inputs were provided to $\mathcal{F}_{\mathsf{Ident}}^{a,b}$ by all clients. They then sign information about the transactions to the burner addresses using $\widehat{\mathsf{sk}}$. If either of this fails, then at least one $\mathcal{P}_i$ signs an "abort" message using $\widehat{\mathsf{sk}}_i$ and sends it to $\mathcal{F}_{\mathsf{SC}}$.

***4. Private Matching.*** Afterwards the servers run `compSwap` on all $\mathcal{F}_{\mathsf{Ident}}^{a,b}$ to match transactions. If **Share** of each $\mathcal{F}_{\mathsf{Ident}}$ is completed, all servers sign a message "ok" using $\mathsf{sk}$ that every server obtains. If this fails then each $\mathcal{P}_i$ signs an "abort" message and sends it to $\mathcal{F}_{\mathsf{SC}}$. If signing succeeded but a server sent "abort", then all $\mathcal{P}_i$ respond by sending the signed "ok" to $\mathcal{F}_{\mathsf{SC}}$. If signing "ok" was successful then all $\mathcal{P}_i$ use **Optimistic Reveal** of $\mathcal{F}_{\mathsf{Ident}}^{a,b}$, i.e. the swaps.

***5. Pay Out.*** The servers will compute the resulting transactions $\mathsf{tx}_a = (\mathtt{id}_a, \mathsf{In}_a, \mathsf{Out}_a, \mathsf{Sig}_a)$ using `makeTX` and by making signatures using $\mathcal{F}_{\mathsf{TSig}}$ under all burner addresses of each $\mathcal{L}_a$. These $\mathsf{tx}_a$ are then sent to clients $\mathcal{C}_j$ that are touched by the transfer. In case of an error a server sends "abort" to $\mathcal{F}_{\mathsf{SC}}$. Once all transactions were signed the servers sign a message *done* using $\mathsf{sk}$ and send this message to $\mathcal{F}_{\mathsf{SC}}$. Upon receiving *done* signed by $\mathsf{sk}$ $\mathcal{F}_{\mathsf{SC}}$ reimburses all $\mathcal{P}_i$.

***Cheating Recovery.*** If any server sends an "abort" to $\mathcal{F}_{\mathsf{SC}}$, then $\mathcal{F}_{\mathsf{SC}}$ waits if any other server publishes an "ok" or "done" signed by $\mathsf{sk}$. If "done" is published after an "abort" then $\mathcal{F}_{\mathsf{SC}}$ reimburses all $\mathcal{P}_i$.

If "ok" is published then each $\mathcal{P}_i$ runs **Reveal** and **Allow Verify** for each $\mathcal{F}_{\mathsf{Ident}}^{a,b}$ and then posts all $\mathsf{sk}$-signed client registrations, transaction ids as well as $\mathcal{F}_{\mathsf{Ident}}$-shares $\mathbf{s}_i^{a,b}$. This allows each server to compute $\mathbf{y}^{a,b} = \sum_i \mathbf{s}_i^{a,b}$ and hence $\mathtt{id}_a, \mathsf{In}_a, \mathsf{Out}_a$ for each $\mathcal{L}_a$ using `makeTX`. For each such transaction the server then computes its shares of $\mathsf{Sig}_a$ which it also posts. All this information allows $\mathcal{F}_{\mathsf{SC}}$ to check if all servers revealed correct shares and signature shares or not. In case of cheating $\mathcal{F}_{\mathsf{SC}}$ sends the cheaters' deposit to all registered clients and reimburses all honest servers. If posted data was correct, $\mathcal{F}_{\mathsf{SC}}$ instead reimburses all servers. Each client $\mathcal{C}_j$ identifies from $\mathcal{F}_{\mathsf{SC}}$ the parts necessary to compute its swap transactions locally and then posts these to finalize the swap.

If no "ok" or "done" is published then each server posts all client registrations signed by $\mathsf{sk}$ to $\mathcal{F}_{\mathsf{SC}}$ as well as the transactions that each client $\mathcal{C}_j$ made to $\mathsf{vk}_j^{\mathsf{ex}}$ from $\mathsf{vk}_j^{\mathsf{src}}$. After a certain delay passed the servers create reimbursing transactions $\mathsf{tx}_j$ for each $\mathcal{C}_j$. For this, each server $\mathcal{P}_i$ generates its share of the signature using $\mathcal{F}_{\mathsf{TSig}}$ and sends this share to $\mathcal{F}_{\mathsf{SC}}$, signed under $\widehat{\mathsf{sk}}_i$.

$\mathcal{F}_{\mathsf{SC}}$ parses all signed client registrations and transactions, locally generates $\mathtt{id}_j, \mathsf{In}_j, \mathsf{Out}_j$ for each $\mathcal{C}_j$ and checks that each $\mathcal{P}_i$ generated its signature share correctly. If any share is missing or if $\mathcal{P}_i$ did not provide a valid share, then the deposit of all cheaters is shared among all clients. Finally, each honest server is reimbursed. If all reimbursement transactions can be made, then each $\mathcal{C}_j$ reads its transactions from $\mathcal{F}_{\mathsf{SC}}$ and posts them on $\mathcal{F}_{\mathsf{BB}}$ while $\mathcal{F}_{\mathsf{SC}}$ reimburses all servers.

### 4.2 The Protocol

We discuss the practical deployment considerations of the protocol in Sec. 5 and describe the full protocol in Figures 11-13 in Appendix C, where we also prove the following statement:

**Theorem 1.** *The protocol $\Pi_{\mathsf{Ex}}$ GUC-implements the functionality $\mathcal{F}_{\mathsf{EX}}$ in the $\mathcal{F}_{\mathsf{SC}}, \mathcal{F}_{\mathsf{Ident}}, \mathcal{F}_{\mathsf{Clock}}, \mathcal{F}_{\mathsf{Sig}}, \mathcal{F}_{\mathsf{TSig}}, \mathcal{F}_{\mathsf{BB}}$-hybrid model against any PPT-adversary corrupting at most $n-1$ of the $n$ servers statically.*

### 4.3 Preventing Denial of Service Attacks by Clients

In $\Pi_{\mathsf{Ex}}$ as sketched above, any client that registers but does not transfer funds causes an abort of the exchange. This can be avoided with an extension of $\Pi_{\mathsf{Ex}}$ that we outline below. We did not include this extension in the formalization of $\Pi_{\mathsf{Ex}}$ in order to keep the protocol description as small as possible.

To avoid the attack, all $\mathcal{P}_i$ in Step 3 of the protocol (**Confirmation**) sign information on which clients will be included and which will be excluded in the exchange, using $\mathcal{F}_{\mathsf{TSig}}$. For each excluded $\mathcal{C}_j$, the servers send the signed message of exclusion to $\mathcal{C}_j$. If $\mathcal{C}_j$'s transaction $\mathsf{tx}_j$ to $\mathsf{vk}_j^{\mathsf{ex}}$ on $\mathcal{L}_j^{\mathsf{src}}$ was confirmed after this signature was generated, then $\mathcal{C}_j$ sends this signature together with $\mathsf{tx}_j$ and its registration information to $\mathcal{F}_{\mathsf{SC}}$ as proof that it should be reimbursed[5]. Upon having obtained this, $\mathcal{F}_{\mathsf{SC}}$ requires all $\mathcal{P}_i$ to create a signature on a reimbursement transaction to $\mathcal{C}_j$. If a server $\mathcal{P}_i$ does not send it's signature share, then $\mathcal{C}_j$ will be reimbursed from the collateral of $\mathcal{P}_i$ on the exchange ledger.

Observe that $\mathsf{tx}_j$ might not be a valid transaction on $\mathcal{L}_j^{\mathsf{src}}$, but the reimbursement transaction will only be valid if $\mathsf{tx}_j$ was valid (as it spends $\mathsf{tx}_j$), hence a malicious $\mathcal{C}_j$ cannot obtain funds of an honest party. Also, if a dishonest $\mathcal{P}_i$ refuses to sign an honest $\mathcal{C}_j$'s reimbursement transaction, then $\mathcal{C}_j$ will be reimbursed on the exchange ledger with funds of $\mathcal{P}_i$.

## 5 Implementation

To demonstrate feasibility of our approach, we describe a simple algorithm for order matching, i.e. `compSwap`, along with an efficient MPC-based privacy preserving implementation of this, based on the ring-variant of the SPDZ protocol [33,32] , called SPD$\mathbb{Z}_{2^k}$ [31].

We implemented an order-matching function, $C_{\texttt{compSwap}}$ realizing the algorithm `compSwap`, which we describe in full detail in Fig. 14 in Appendix D . We

[5] All this information was signed by $\mathcal{F}_{\mathsf{TSig}}$ and must therefore be valid.

note that $C_{\texttt{compSwap}}$ is a simplified version of standard limit-order price-focused matching algorithms, in the sense that it assumes all orders are for a constant amount of tokens from ledger $\mathcal{L}_a$ and only matches according to price. That is, the algorithm sorts the buy and sell orders according to their maximum and minimum limits respectively. Then the largest acceptable buy price is then matched with the smallest acceptable sell price, assuming the buy price is larger than the sell price. The clearing price will then be the average of the two prices. The price describes how many tokens of $\mathcal{L}_b$ must be swapped to get a constant amount of the tokens on $\mathcal{L}_a$. Orders which are not matched are simply discarded, without disclosing their price limits. We chose to implement this simple matching algorithm as it provides a minimally useful example of what is required by `compSwap` by performing oblivious sorting and selecting, which are they key aspects we can expect from any reasonable choice of algorithm.

Furthermore, based on benchmarks of the most efficient implementation for full-threshold ECDSA [30,34], we provide estimates for an implementation of our full protocol showing that it can be realized efficiently, with the bottleneck being the time it takes a transaction to be confirmed on the underlying ledgers.

**Complexity of Matching.** It is easy to see that comparison is the key primitive used in our order matching. In most arithmetic MPC schemes this primitive can be realized using $O(\log(k))$ multiplication gates and rounds of communication where $k$ is the max bit-length of the numbers being computed on [24,31] . Besides being used for deciding whether a buy and sell order should be matched, comparison is also the key component in most oblivious sorting algorithms. In particular, Batcher's Odd-Even Merge-sort [44, Sec. 5.4.3], which our implementation uses, has comparison depth $O(\log^2(m)))$ and uses a total of $O(m\log^2(m))$ comparisons. For this reason the overall round complexity of our concrete order-matching algorithm, $C_{\texttt{compSwap}}$, ends up at $O(\log(k)\cdot\log^2(m))$ with multiplication gate complexity of $O(k\cdot m\log^2(m))$. The communication complexity associated with a multiplication gate is $O(k+\kappa)$ bits for $\text{SPD}\mathbb{Z}_{2^k}$ [31]. Hence we get at most $O((k+\kappa)\cdot m)$ bit in bandwidth usage per round. The total amount used for each of the different choices of $m$ and $k$ by our implementation, including overhead by the framework, is expressed in Table 1.

Table 1: Complexity of the matching protocol implementation in $\text{SPD}\mathbb{Z}_{2^k}$.

| Orders ($m$) | $k=32,\ \kappa=26$ | | $k=64,\ \kappa=57$ | |
|---|---|---|---|---|
| | #mult | #rounds | #mult | #rounds |
| 4 | 2269 | 151 | 4075 | 158 |
| 8 | 7417 | 262 | 13351 | 275 |
| 16 | 22833 | 410 | 41151 | 431 |
| 32 | 66465 | 595 | 119871 | 626 |
| 64 | 184449 | 817 | 332799 | 862 |
| 128 | 492097 | 1077 | 888127 | 1169 |

**Implementation of Matching.** We implemented and ran $C_{\texttt{compSwap}}$ using the Fresco framework [1], which is an open-source MPC framework in Java for secure computation in the dishonest majority setting. We chose Fresco as it offers a

simple API allowing quick construction of MPC applications. Furthermore, since it is Java-based, it also allows for easy cross-platform deployment and integration with other software. Additionally, Fresco is a commonly used framework for prototyping MPC applications [31,28] and supports a bring-your-own-backend approach, allowing easy switching of the underlying MPC scheme without having to modify the program to be executed. Most importantly however, Fresco is actively maintained, has high test coverage and offers an extensive API of operations, making it a great choice for prototyping MPC protocols, with a focus on potential real-world deployment, despite it not being the fastest option available.

We chose to base our implementation on arithmetic MPC instead of binary MPC to closer reflect the domain where real-world matching algorithms work. This also allows for more advanced computation of matches and prices through arithmetic computation on the orders. This also avoids the large overhead there would otherwise be present when doing arithmetic calculations on large integers using an MPC protocol working over binary values. Furthermore, we also chose SPD$\mathbb{Z}_{2^k}$ [27] as our MPC protocol to closer reflect the real-world setting. In particular SPD$\mathbb{Z}_{2^k}$ can work over the same domain as standard processors, i.e. 32 or 64 bit integers, and hence allows for simple integration with already existing algorithms, and systems that have been optimized for working over these domains. Because of this, the online computation of SPD$\mathbb{Z}_{2^k}$ is significantly faster than for SPDZ when computing comparison [31].

We note that we could have optimized our implementation by using newer and more efficient approaches to MPC, such as approaches mixing both the binary and arithmetic setting [37]. However, the goal of our implementation has been to show feasibility and an upper bound on the time it would require to execute our protocol using a fully tested, documented and maintained framework. Thus we leave such possible optimizations as future work.

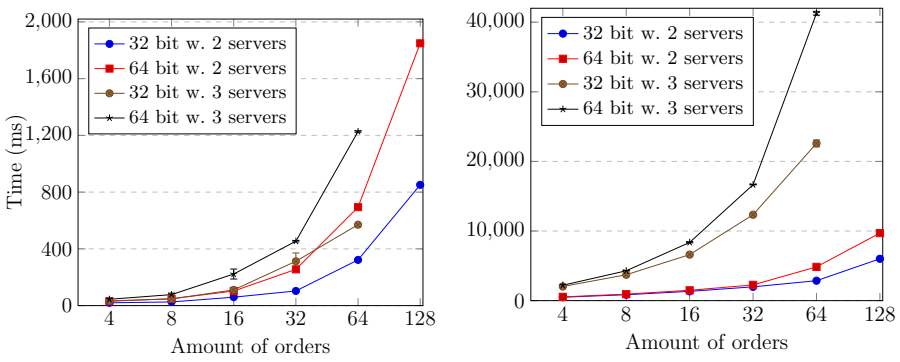


Fig. 3: Online phase of LAN (left) and WAN (right) execution of oblivious matching using SPD$\mathbb{Z}_{2^k}$ on AWS m5.xlarge.

**Benchmarking Matching.** We benchmarked the online execution of the matching protocol in several different settings using Amazon's Web Services EC2. We show these benchmarks in Fig. 3 based on the average of 30 iterations (executed after 30 "warm-up" iterations) and note that memory usage never gets above 536 MB regardless of test. For all tests we used m5.xlarge instances running Ubuntu 18.04 LTS. This means that each instance has 4 virtual CPUs and 16 GiB of RAM along with a LAN connections of up to a 10 Gbps with less than 0.1 ms latency, when running instances in the same data-center. For the WAN setting we also used m5.xlarge instances, but data centers in different countries. For the setting with 2 servers, one was located in Ireland and another in the U.K., with a latency of around 10 ms. For the setting with 3 servers, we kept the servers in Ireland and the U.K., but added a third server in Germany, with a latency of up to 25 ms between the Germany and Ireland servers.

When working over 32-bit integers ($k = 32$), the statistical security parameter is $\kappa = 26$, whereas when working over the 64-bit integers ($k = 64$) we have $\kappa = 57$. This is due to the underlying implementation of SPD$\mathbb{Z}_{2^k}$ in Fresco, and was originally made for efficiency choices [31].

From Fig. 3 we notice a steep increase in execution time when expanding the amount of orders and when moving from 2 to 3 servers. This is especially true in the WAN setting. This is to be expected due to the underlying SPD$\mathbb{Z}_{2^k}$ protocol having round complexity in the multiplicative depth of the circuit being computed and where each round involves every party sending two ring elements to *all* other parties. Thus the overall execution time of the protocol becomes highly depended on the underlying network, in particular its latency, when increasing the amount of servers. This is also the case when increasing the amount of orders as the round complexity is bounded by $O(\log(k) \cdot \log^2(m))$ and multiplication gate complexity is bounded by $O(k \cdot m \log^2(m))$.

We imagine our protocol being run by a small set of servers, either run by a single or a few public organizations, whose servers will be in physically distinct locations, running distinct systems and administered by different people. For this reason we only benchmark for 2-3 servers, but note that if high scalability is desired one can simply change the underlying MPC scheme to a constant round scheme such as BMR [7] as done in "Insured MPC" [5].

We note that Figure 3 only shows the online execution time. However, each multiplication gate used for such an execution requires to be preprocessed in advance in SPD$\mathbb{Z}_{2^k}$. This preprocessing is independent of the input of the function to be computed and can thus be done before the clients even submit their exchange orders. The most efficient SPD$\mathbb{Z}_{2^k}$ triple generation protocol with benchmarks is due to Damgård *et al.* [31], offering a throughput of 26,455 triples/second for $k = 32$ and 9,496 triples/second for $k = 64$ in the 2 server setting on LAN. Thus this can be done completed in about 1.5 minutes for $m = 128$ and $k = 64$. However, a more band-width friendly solution has since been introduced by Orsini *et al.* [50] that will most likely be more efficient for both the LAN and WAN, but has currently not been implemented with benchmarks.

Table 2: Complexity of the (threshold) signatures required by $\Pi_{\mathsf{Ex}}$ for its different phases. Time estimates are based on [39, Table 1].

| | $\mathcal{F}_{\mathsf{TSig}}$ KEYGEN | $\mathcal{F}_{\mathsf{TSig}}$ SIGN | $\mathcal{F}_{\mathsf{Sig}}$ SIGN pr server | $\mathcal{F}_{\mathsf{Sig}}/\mathcal{F}_{\mathsf{TSig}}$ VERIFY pr server | Wall-clock time (sec.) for $m = 128$ 2 servers LAN | 2 servers WAN | 3 servers LAN | 3 servers WAN |
|---|---|---|---|---|---|---|---|---|
| Initialize | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| Enroll | $m$ | $m$ | $m$ | 0 | 125 | 125 | 246 | 246 |
| Exchange | 0 | $m+1$ | 0 | 0 | 63 | 63 | 124 | 124 |
| Open | 0 | $< m+1$ | 0 | 0 | 63 | 63 | 124 | 124 |
| Abort w. out. | 0 | $< m$ | 0 | 0 | 62 | 62 | 123 | 123 |
| Abort wo. out. | 0 | $m$ | 0 | 0 | 62 | 62 | 123 | 123 |

**Signatures.** We outline the computational need and timing estimates of $\mathcal{F}_{\mathsf{TSig}}$ and $\mathcal{F}_{\mathsf{Sig}}$, which is required by our protocol $\Pi_{\mathsf{Ex}}$ in Table 2. Note that timings in this table are estimates based on benchmark of a the recent work by Gennaro and Goldfeder [39] for threshold ECDSA (currently the most popular scheme in use by cryptocurrencies) with identifiable abort (which is needed for our protocol). The estimates are computed by adding the raw, single-core computation benchmarks of Gennaro and Goldfeder and combine it with the latency between the servers we used in Sec. 5 required for *all* rounds of the signing protocol (including preprocessing time). We note that Gennaro and Goldfeder do not include benchmarks for the key generation phase so we have benchmarked this as signing. This is quite conservative since the key generation phase of their protocol is faster then the signing phase.

**Total Execution Time.** We only implemented and benchmarked the MPC implementation of the $C_{\texttt{compSwap}}$ algorithm from Fig. 14 since it is clearly going to be the computation and communication related bottleneck when realizing $\mathcal{F}_{\mathsf{Ident}}$ in the way discussed in Sec. 2.2. The only other cryptographic computation of this realization is limited to committing and opening of $O(n(m+s))$ commitments and sampling an equivalent amount of random elements in the MPC computation. Such commitments can be constructed very simply and efficiently in the ROM [16]. In particular in the order of microseconds, when using a standard hash function like SHA-256 to realize the random oracle. Furthermore, we note that besides the realization of $\mathcal{F}_{\mathsf{Ident}}$ and $\mathcal{F}_{\mathsf{TSig}}$ there are no other heavy communication or computation involved in realizing $\Pi_{\mathsf{Ex}}$, since the other parts is basically straight-line executable business logic.

Taking the above discussion into account, it is easy to see that in a full online execution, the execution $C_{\texttt{compSwap}}$ with no malicious behaviour, will be slower than the time required for computing the threshold signatures[6].

However, we now argue that even with the unoptimized implementation of $C_{\texttt{compSwap}}$ and $\mathcal{F}_{\mathsf{TSig}}$ these are not going to be the bottle-neck in regards to wall-clock execution time in practice. The reason being the time it takes to *finalize*

[6] Although we should note that the benchmarks of threshold signatures by Gennaro and Goldfeder [39] are not optimized and run on a single-core consumer laptop whereas our benchmark of $C_{\texttt{compSwap}}$ runs on a powerful AWS instance. We expect that the time required for the threshold signatures can be reduced significantly.

transactions on the ledgers, which is necessary for *any* cross-ledger exchange. A block has been finalized once it has been written to the ledger *and* a certain amount of other blocks have been written afterwards to the same chain. This ensures that one can be reasonably certain that the block with the transaction will not be overwritten. The frequency at which blocks are constructed, and how many should be constructed before a transaction can be considered finalized, depends highly on the specific choice of ledger.

For Bitcoin, a new block is expected to be constructed every 10 minutes, and a rule of thumb is that 6 further blocks must be written before a transaction can be considered finalized. On Ethereum a block is constructed every 10-20 seconds and can *optimistically* be considered finalized after 3 minutes. In case of Cardano, blocks are considered finalized after 10 minutes. This means that each round of ledger I/O will generally involve a latency of several minutes and hence be the bottleneck in practice.

**Comparison to current solutions.** As discussed in Sec. 1, besides our solution, there are generally only two other approaches to cross-chain exchange: 1) A centralized exchange where a trusted party receives tokens on the different ledgers and transfers the exchanged tokens accordingly based on some centrally decided exchange rate. An example of this is Coinbase or Kraken. 2) A solution using atomic-swap as employed by current decentralized exchanges [12,40,52], where hash time locked contracts are used to ensure that the required transfers actually get carried out. Only considering efficiency, even in the case of the centralized exchange, two transactions must be carried out and finalized on the underlying ledgers; one transfer from ledger $\mathcal{L}_a$ to the exchange and one from the exchange to ledger $\mathcal{L}_b$. This means that we can expect a centralized exchange to take an order of tens of minutes to hours before it is has been fully finalized. If a system based on atomic swaps is used, an extra transaction is required. Thus requiring 3 sequential transactions to finalize before the exchange can be considered complete.

When there is no malicious behaviour occurring, our system uses the *optimal* 2 sequential transactions[7] similarly to centralized exchanges. Specifically our system involves the transaction by clients transferring their tokens to the burner addresses and then the transaction of paying out the exchanged tokens. Furthermore, we note that while waiting for the first transaction to finalize the servers are sitting idle. This means that they could *optimistically* leverage this finalization time and start the actual matching computation. If the tokens are not transferred by all clients, the servers will generate refund transactions instead of opening the transactions computed by the private matching. Since the total

[7] Technically 4 transactions are needed since the servers must put down a deposit to the smart-contract, and receive this back at the end. However, the deposit can be reused for an arbitrary amount of executions of exchanges, and we consider this as purely overhead related to system setup. In case of malicious behaviour our protocol uses at most 7 transactions to either complete the exchange or refund the clients and return the honest servers' deposits.

computation time of $\Pi_{\mathsf{Ex}}$ is less than the finalization time of most blockchains, it means that its execution time will not be noticeable in the time it takes to carry out and finalize a complete exchange. Hence using our protocol will not add any time overhead to the current solutions (centralized or swapping). Furthermore, since nothing (besides the one-time initialization of the server) is written to the ledger, or executed by smart contracts, unless malicious behaviour occurs, it means that there will be no added mining-related cost by our approach. Thus achieving the added distributed security comes with no penalty in price or execution time as long as no malicious behaviour occurs. The only actual cost is what is needed to keep the servers running and the one-time cost of initialization.

**Deployment Considerations.** Despite increasing the latency of an exchange, we note that in a real-world deployment the servers would likely want to wait for the clients' transfers to the burner addresses get finalized before starting the matching computation to prevent wasting computation and transfer/smart contract fees. The reason being that it is unlikely that a client manages to enroll in the exchange with the servers but fails to publish a transaction to its source ledger. Even though it is formally considered a malicious deviation this could happen without malice in many real world situations, either where the client simply loses Internet connectivity or where miners chose to block a specific client's transfer. Thus the servers would only do an exchange on the request from clients that have actually been finalized at a certain point in time (and refund any transactions that happen to complete afterwards). This way it is possible to prevent a single unstable client from delaying the entire protocol. More importantly it will also prevent malicious clients from forcing the servers (and honest clients) to spend money on smart contract and transaction fees of exchanges that cannot be completed as discussed in Sec. 4.3. In regards to creating a highly usable deployment we note that the clients actually don't need to be involved during the payout phase of the protocol. Instead of having each client $\mathcal{C}_j$ post the signed transaction transferring their exchanged tokens to $\mathsf{vk}_j^{\mathsf{trg}}$ on ledger $\mathcal{L}_j^{\mathsf{trg}}$ themselves, we can simply have one of the honest servers post this transaction on their behalf. This means that they do neither need to wait for all other clients' transfers of source tokens to be finalized, nor for all the servers to finish computing the order-matching. This means that, unless one or more servers are malicious, the only time the clients need to interact with the protocol is during enrollment where they must register at the servers (and receive a signed confirmation). For clients this only involves giving outsourced input to $\mathcal{F}_{\mathsf{Ident}}^{a,b}$, which can be done very efficiently and independently of other clients or the size of the circuit to compute [41]. Thus this is something that can easily be done in the user's browser using a JavaScript web-app that integrates with a user's browser-wallet (e.g. MetaMask for Ethereum), where the server's signed response can be saved in the local web-page cache. Even if some servers act maliciously and the abort branch of the protocol ends up being executed, assuming there is still a single honest server, that honest server could simply act on behalf of the client to ensure they get refunded. This is because all the

information the client would need to post to $\mathcal{F}_{\mathsf{SC}}$ is actually constructed and known by all the servers.

**Future Work.** Certain centralized exchanges are able to carry out huge amounts of exchanges per second, without barely any latency in reception of the exchanged tokens. This is through a state-channel [36] approach where they hold the client's tokens persistently and simply carry out the transactions "in-the-head", keeping an internal ledger of which client owns what amount of tokens on which ledger. However, such approaches necessarily require the same 2 sequential finalized transactions when they are to be committed on their respective ledgers. We consider generalizing our approach to this setting, by implementing a state-channel in MPC, an interesting direction for future work.

Even without realizing state-channels, simply employing a reactive MPC functionality to carry out unmatched orders into a future iteration of `compSwap` would be interesting. Along with adding more advanced features to $C_{\texttt{compSwap}}$, such as expiration time and fully variable quantities to exchange, this would allow for a complete and continues decentralized exchange platform that clients can use "as-a-service" through a simple web-platform and a standard crypto-wallet.

## References


1. Alexandra Institute. FRESCO - a FRamework for Efficient Secure COmputation. `https://github.com/aicis/fresco`.
2. M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2014.
3. C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas. Bitcoin as a transaction ledger: A composable treatment. In *CRYPTO 2017, Part I*, LNCS. Springer, Heidelberg, Aug. 2017.
4. C. Baum, B. David, and R. Dowsley. A framework for universally composable publicly verifiable cryptographic protocols. Cryptology ePrint Archive, Report 2020/207, 2020. `https://eprint.iacr.org/2020/207`.
5. C. Baum, B. David, and R. Dowsley. Insured MPC: Efficient secure computation with financial penalties. In *FC 2020*, LNCS. Springer, Heidelberg, Feb. 2020.
6. C. Baum, B. David, R. Dowsley, J. B. Nielsen, and S. Oechsner. Craft: Composable randomness and almost fairness from time. Cryptology ePrint Archive, Report 2020/784, 2020. `https://eprint.iacr.org/2020/784`.
7. D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*. ACM Press, May 1990.
8. F. Benhamouda, S. Halevi, and T. Halevi. Supporting private data on hyperledger fabric with secure multiparty computation. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 357–363, April 2018.
9. I. Bentov, Y. Ji, F. Zhang, L. Breidenbach, P. Daian, and A. Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. In *ACM CCS 2019*. ACM Press, Nov. 2019.

10. I. Bentov and R. Kumaresan. How to use bitcoin to design fair protocols. In *CRYPTO 2014, Part II*, LNCS. Springer, Heidelberg, Aug. 2014.
11. I. Bentov, R. Kumaresan, and A. Miller. Instantaneous decentralized poker. In *ASIACRYPT 2017, Part II*, LNCS. Springer, Heidelberg, Dec. 2017.
12. Binance. Binance DEX Documentation. `https://docs.binance.org`, 2020.
13. S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu. ZEXE: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2020.
14. J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium 2018*, pages 991–1008. USENIX Association, 2018.
15. B. Bünz, S. Agrawal, M. Zamani, and D. Boneh. Zether: Towards privacy in a smart contract world. In *FC 2020*, LNCS. Springer, Heidelberg, Feb. 2020.
16. J. Camenisch, M. Drijvers, T. Gagliardoni, A. Lehmann, and G. Neven. The wonderful world of global random oracles. In *EUROCRYPT 2018, Part I*, LNCS. Springer, Heidelberg, Apr. / May 2018.
17. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*. IEEE Computer Society Press, Oct. 2001.
18. R. Canetti. Universally composable signature, certification, and authentication. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004)*, page 219. IEEE Computer Society, 2004.
19. R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. In *TCC 2007*, LNCS. Springer, Heidelberg, Feb. 2007.
20. R. Canetti, R. Gennaro, S. Goldfeder, N. Makriyannis, and U. Peled. Uc non-interactive, proactive, threshold ecdsa with identifiable aborts. Cryptology ePrint Archive, Report 2021/060, 2021. `https://eprint.iacr.org/2021/060`.
21. R. Canetti, N. Makriyannis, and U. Peled. Uc non-interactive, proactive, threshold ecdsa. Cryptology ePrint Archive, Report 2020/492, 2020. `https://eprint.iacr.org/2020/492`.
22. J. Cartlidge, N. P. Smart, and Y. T. Alaoui. Multi-party computation mechanism for anonymous equity block trading: A secure implementation of turquoise plato uncross. Cryptology ePrint Archive, Report 2020/662, 2020. `https://eprint.iacr.org/2020/662`.
23. J. Cartlidge, N. P. Smart, and Y. Talibi Alaoui. MPC joins the dark side. In *ASIACCS 19*. ACM Press, July 2019.
24. O. Catrina and S. de Hoogh. Improved primitives for secure multiparty integer computation. In *SCN 10*, LNCS. Springer, Heidelberg, Sept. 2010.
25. A. R. Choudhuri, M. Green, A. Jain, G. Kaptchuk, and I. Miers. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In *ACM CCS 2017*. ACM Press, Oct. / Nov. 2017.
26. R. Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *18th ACM STOC*. ACM Press, May 1986.
27. R. Cramer, I. Damgård, N. Döttling, I. Giacomelli, and C. Xing. Linear-time non-malleable codes in the bit-wise independent tampering model. In *ICITS 17*, LNCS. Springer, Heidelberg, Nov. / Dec. 2017.
28. R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod $2^k$ for dishonest majority. In *CRYPTO 2018, Part II*, LNCS. Springer, Heidelberg, Aug. 2018.

29. P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2020.
30. A. P. K. Dalskov, C. Orlandi, M. Keller, K. Shrishak, and H. Shulman. Securing DNSSEC keys via threshold ECDSA from generic MPC. In *ESORICS 2020, Part II*, LNCS. Springer, Heidelberg, Sept. 2020.
31. I. Damgård, D. Escudero, T. K. Frederiksen, M. Keller, P. Scholl, and N. Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2019.
32. I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS 2013*, LNCS. Springer, Heidelberg, Sept. 2013.
33. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO 2012*, LNCS. Springer, Heidelberg, Aug. 2012.
34. J. Doerner, Y. Kondi, E. Lee, and a. shelat. Threshold ECDSA from ECDSA assumptions: The multiparty case. In *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2019.
35. S. Dziembowski, L. Eckey, and S. Faust. FairSwap: How to fairly exchange digital goods. In *ACM CCS 2018*. ACM Press, Oct. 2018.
36. S. Dziembowski, S. Faust, and K. Hostáková. General state channel networks. In *ACM CCS 2018*. ACM Press, Oct. 2018.
37. D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *CRYPTO 2020, Part II*, LNCS. Springer, Heidelberg, Aug. 2020.
38. A. Gągol, J. Kula, D. Straszak, and M. Świętek. Threshold ecdsa for decentralized asset custody. Cryptology ePrint Archive, Report 2020/498, 2020. `https://eprint.iacr.org/2020/498`.
39. R. Gennaro and S. Goldfeder. One round threshold ecdsa with identifiable abort. Cryptology ePrint Archive, Report 2020/540, 2020. `https://eprint.iacr.org/2020/540`.
40. IDEX. IDEX. `https://idex.market/`, 2020.
41. T. P. Jakobsen, J. B. Nielsen, and C. Orlandi. A framework for outsourcing of secure computation. In G. Ahn, A. Oprea, and R. Safavi-Naini, editors, *ACM CCSW 2014*, pages 81–92. ACM, 2014.
42. J. Katz, U. Maurer, B. Tackmann, and V. Zikas. Universally composable synchronous computation. In *TCC 2013*, LNCS. Springer, Heidelberg, Mar. 2013.
43. A. Kiayias, H.-S. Zhou, and V. Zikas. Fair and robust multi-party computation using a global transaction ledger. In *EUROCRYPT 2016, Part II*, LNCS. Springer, Heidelberg, May 2016.
44. D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, second edition, 1998.
45. A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2016.
46. R. Kumaresan and I. Bentov. Amortizing secure computation with penalties. In *ACM CCS 2016*. ACM Press, Oct. 2016.

47. R. Kumaresan, T. Moran, and I. Bentov. How to use bitcoin to play decentralized poker. In *ACM CCS 2015*. ACM Press, Oct. 2015.
48. F. Massacci, C. N. Ngo, J. Nie, D. Venturi, and J. Williams. FuturesMEX: Secure, distributed futures market exchange. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2018.
49. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
50. E. Orsini, N. P. Smart, and F. Vercauteren. Overdrive2k: Efficient secure MPC over $\mathbb {Z}_{2^k}$ from somewhat homomorphic encryption. In S. Jarecki, editor, *Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings*, volume 12006 of *Lecture Notes in Computer Science*, pages 254–283. Springer, 2020.
51. Selfkey. A Comprehensive List of Cryptocurrency Exchange Hacks. `https://selfkey.org/list-of-cryptocurrency-exchange-hacks/`, 2020.
52. Uniswap. Uniswap Documentation. `https://uniswap.org/docs/v2/`, 2020.

## Appendix A Remaining Notation and Basic Definitions

For any $k \in \mathbb{N}$ we write $[k]$ for the set $\{1, \ldots, k\}$. A function $f(x)$ is negligible in $x$ (or $\mathrm{negl}(x)$ to denote an arbitrary such function) if $f(x)$ is positive and for every positive polynomial $p(x) \in \mathrm{poly}(x)$ there exists a $x' \in \mathbb{N}$ such that $\forall x \geq x' : \ f(x) < 1/p(x)$. Two ensembles $X = \{X_{\kappa,z}\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$ and $Y = \{Y_{\kappa,z}\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$ of binary random variables are said to be *statistically indistinguishable*, denoted by $X \approx_s Y$, if for all $z$ it holds that $\mid \Pr[\mathcal{D}(X_{\kappa,z}) = 1] - \Pr[\mathcal{D}(Y_{\kappa,z}) = 1] \mid$ is negligible in $\kappa$ for every probabilistic algorithm (distinguisher) $\mathcal{D}$. In case this only holds for computationally bounded (non-uniform probabilistic polynomial-time ($PPT$)) distinguishers we say that $X$ and $Y$ are *computationally indistinguishable* and denote it by $\approx_c$.

### A.1 Public Verifiability in Global UC

We follow the approach of Badertscher *et al.* [3] and allow the set of verifiers $\mathcal{V}$ to be dynamic by adding register and de-register instructions as well as instructions that allow $\mathcal{S}$ to obtain the list of registered verifiers. All functionalities with public verifiability include the following interfaces (which are omitted henceforth for simplicity):

**Register:** Upon receiving (Register, *sid*) from some verifier $\mathcal{V}_i$, set $\mathcal{V} = \mathcal{V} \cup \mathcal{V}_i$ and return (Registered, *sid*, $\mathcal{V}_i$) to $\mathcal{V}_i$.

**Deregister:** Upon receiving (Deregister, *sid*) from some verifier $\mathcal{V}_i$, set $\mathcal{V} = \mathcal{V} \setminus \mathcal{V}_i$ and return (Deregistered, *sid*, $\mathcal{V}_i$) to $\mathcal{V}_i$.

**Is Registered:** Upon receiving (Is-Registered, *sid*) from $\mathcal{V}_i$, return (Is-Registered, *sid*, $b$) to $\mathcal{V}_i$, where $b = 1$ if $\mathcal{V}_i \in \mathcal{V}$ and $b = 0$ otherwise.

**Get Registered:** Upon receiving (Get-Registered, *sid*) from the ideal adversary $\mathcal{S}$, the functionality returns (Get-Registered, *sid*, $\mathcal{V}$) to $\mathcal{S}$.

The above instructions can also be used by other functionalities to register as a verifier of a publicly verifiable functionality.

### A.2 Global Clocks and Bulletin Boards

$\mathcal{F}_{\mathsf{Clock}}$ (see Fig. 4) is assumed to be a global functionality, which means that other ideal functionalities will be granted access to it. And while in the real protocol execution all parties send messages to and receive them from $\mathcal{F}_{\mathsf{Clock}}$, in the simulated case only the ideal functionality, other global functionalities as well as the dishonest parties will do so. Hybrid functionalities in the simulation might also be given access, but this is not necessary in our setting. For simplicity, we do not introduce a session management in $\mathcal{F}_{\mathsf{Clock}}$ as it is not necessary to state our result. $\mathcal{F}_{\mathsf{BB}}$ (see Fig. 5) is a Bulletin-Board global functionality that can store authenticated messages as well as make them available for all users. We use $\mathcal{F}_{\mathsf{BB}}$ as a dummy functionality for the ledgers among which assets will be transferred.

**Functionality $\mathcal{F}_{\mathsf{Clock}}$**

$\mathcal{F}_{\mathsf{Clock}}$ is parameterized by a variable $\nu$, sets $\mathcal{P}, \mathcal{F}$ of parties and functionalities respectively. It keeps a Boolean variable $d_{\mathcal{J}}$ for each $\mathcal{J} \in \mathcal{P} \cup \mathcal{F}$, a counter $\nu$ as well as an additional variable $\mathtt{u}$. All $d_{\mathcal{J}}$, $\nu$ and $\mathtt{u}$ are initialized as 0.

**Clock Update:** Upon receiving a message (UPDATE) from $\mathcal{J} \in \mathcal{P} \cup \mathcal{F}$:

1. Set $d_{\mathcal{J}} = 1$.
2. If $d_F = 1$ for all $F \in \mathcal{F}$ and $d_p = 1$ for all honest $p \in \mathcal{P}$, then set $\mathtt{u} \leftarrow 1$ if it is 0.

**Clock Read:** Upon receiving a message (READ) from any entity:

1. If $\mathtt{u} = 1$ then first send (TICK, $sid$) to $\mathcal{S}$. Next set $\nu \leftarrow \nu + 1$, reset $d_{\mathcal{J}}$ to 0 for all $\mathcal{J} \in \mathcal{P} \cup \mathcal{F}$ and reset $\mathtt{u}$ to 0.
2. Answer the entity with (READ, $\nu$).

Fig. 4: Functionality $\mathcal{F}_{\mathsf{Clock}}$ for a Global Clock.

**Functionality $\mathcal{F}_{\mathsf{BB}}$**

$\mathcal{F}_{\mathsf{BB}}$ contains a list $\mathcal{M} \in \{0,1\}^*$ of messages $m \in \{0,1\}^*$ which is initially empty.

**Post:** Upon receiving (POST, $m, \mathsf{vk}, \sigma$) from some entity contact $\mathcal{F}_{\mathsf{Sig}}$ or $\mathcal{F}_{\mathsf{TSig}}$ belonging to $\mathsf{vk}$. If $\sigma$ verifies for $m$ and $\mathsf{vk}$ then send (POST, $m, \mathsf{vk}, \sigma$) to $\mathcal{S}$ and append $(m, \mathsf{vk})$ to $\mathcal{M}$.

**Read:** Upon receiving (READ) from some entity, return $\mathcal{M}$.

Fig. 5: The bulletin board functionality $\mathcal{F}_{\mathsf{BB}}$ that abstractly describes the source and target public ledgers of transactions.

**Functionality $\mathcal{F}_{\mathsf{Ident}}$**

For each session, $\mathcal{F}_{\mathsf{Ident}}$ interacts with servers $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, clients $\mathcal{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$ and also provides an interface to register external verifiers $\mathcal{V}$. It is parameterized by a circuit $C$ with inputs $x^{(1)}, \ldots, x^{(m)}$ and output $\mathbf{y} \in \mathbb{F}^g$. $\mathcal{S}$ provides a set $I_{\mathcal{P}} \subset [n]$ of corrupt parties and $I_{\mathcal{C}} \subseteq [m]$ of corrupt clients. $\mathcal{F}_{\mathsf{Ident}}$ only interacts with $\mathcal{P}, \mathcal{C}, \mathcal{V}$ and $\mathcal{S}$ of the respective session *sid*.

Throughout **Init**, **Input**, **Evaluate** and **Share**, $\mathcal{S}$ can at any point send (ABORT, *sid*) to $\mathcal{F}_{\mathsf{Ident}}$, upon which it sends (ABORT, $sid, \perp$) to all parties and terminates. Throughout **Reveal**, $\mathcal{S}$ at any point is allowed to send (ABORT, $sid, J$) to $\mathcal{F}_{\mathsf{Ident}}$. If $J \subseteq I_{\mathcal{P}}$ then $\mathcal{F}_{\mathsf{Ident}}$ will send (ABORT, $sid, J$) to all honest parties and terminate.

**Init:** Upon first input (INIT, *sid*) by all parties in $\mathcal{P}$ set $\mathtt{rev}, \mathtt{ver}, \mathtt{ref} \leftarrow \emptyset$.

**Input:** Upon first input (INPUT, $sid, j, x^{(j)}$) by $\mathcal{C}_j$ and input (INPUT, $sid, j, \cdot$) by all servers the functionality stores the value $(j, x^{(j)})$ internally.

**Evaluate:** Upon first input (COMPUTE, *sid*) by all parties in $\mathcal{P}$ and if the inputs $(j, x^{(j)})_{j \in [m]}$ for all clients have been stored internally, compute $\mathbf{y} \leftarrow C(x^{(1)}, \ldots, x^{(m)})$ and store $\mathbf{y}$ locally.

**Share:** Upon first input (SHARE, *sid*) by $\mathcal{P}_i \in \mathcal{P}$ and if **Evaluate** was finished:

1. For each $i \in I_{\mathcal{P}}$ let $\mathcal{S}$ provide $\mathbf{s}^{(i)} \in \mathbb{F}^g$.
2. For each $\mathcal{P}_i \in \overline{I_{\mathcal{P}}}$ sample $\mathbf{s}^{(i)} \xleftarrow{\$} \mathbb{F}^g$ subject to the constraint that $\mathbf{y} = \sum_{i \in [n]} \mathbf{s}^{(i)}$.

**Optimistic Reveal:** Upon input (OPTIMIST-OPEN, $sid, i$) by each honest $\mathcal{P}_i$ and if **Share** is completed, then send (OUTPUT, $sid, \mathbf{y}$) to $\mathcal{S}$. If $\mathcal{S}$ sends (CONTINUE, *sid*) then send (OUTPUT, $sid, \mathbf{y}$) to each honest $\mathcal{P}_i$, otherwise send (OUTPUT, $sid, \perp$).

**Reveal:** On input (REVEAL, $sid, i$) by $\mathcal{P}_i$, if $i \notin \mathtt{rev}$ send (REVEAL, $sid, i, \mathbf{s}^{(i)}$) to $\mathcal{S}$.

- If $\mathcal{S}$ sends (REVEAL-OK, $sid, i$) then set $\mathtt{rev} \leftarrow \mathtt{rev} \cup \{i\}$, send (REVEAL, $sid, i, \mathbf{s}^{(i)}$) to all parties in $\mathcal{P}$.
- If $\mathcal{S}$ sends (REVEAL-NOT-OK, $sid, i, J$) with $J \subseteq I_{\mathcal{P}}, J \neq \emptyset$ then send (REVEAL-FAIL, $sid, i$) to all parties in $\mathcal{P}$ and set $\mathtt{ref} \leftarrow \mathtt{ref} \cup J$.

**Test Reveal:** Upon input (TEST-REVEAL, *sid*) from a party in $\mathcal{P} \cup \mathcal{V}$ return (REVEAL-FAIL, $sid, \mathtt{ref}$) if $\mathtt{ref} \neq \emptyset$. Otherwise return (REVEAL-FAIL, $sid, [n] \setminus \mathtt{rev}$).

**Allow Verify:** Upon input (START-VERIFY, $sid, i$) from party $\mathcal{P}_i \in \mathcal{P}$ set $\mathtt{ver} \leftarrow \mathtt{ver} \cup \{i\}$. If $\mathtt{ver} = [n]$ then deactivate all interfaces except **Test Reveal** and **Verify**.

**Verify:** Upon input (VERIFY, $sid, \mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(n)}$) by $\mathcal{V}_i \in \mathcal{V}$ with $\mathbf{z}^{(j)} \in \mathbb{F}^g$:

- If $\mathtt{ver} \neq [n]$ then return (VERIFY-FAIL, $sid, [n] \setminus \mathtt{ver}$).
- Else, if $\mathtt{ver} = [n]$ and $\mathtt{rev} \neq [n]$ then send to $\mathcal{V}_i$ what **Test Reveal** sends.
- Else set $\mathtt{ws} \leftarrow \{j \in [n] \mid \mathbf{z}^{(j)} \neq \mathbf{s}^{(j)}\}$ and return (OPEN-FAIL, $sid, \mathtt{ws}$).

Fig. 6: Functionality $\mathcal{F}_{\mathsf{Ident}}$ for MPC with Publicly Verifiable Output.

### A.3 Client-Input Publicly Verifiable Multiparty Computation Functionality $\mathcal{F}_{\mathsf{Ident}}$

In Fig. 6 we formally define functionality $\mathcal{F}_{\mathsf{Ident}}$ for client-input publicly verifiable multiparty computation adapted from [5]. In comparison to [5] we use non-interactive UC commitments to realize the functionality, which simplifies it. As $\mathcal{F}_{\mathsf{SC}}$ will need to interact with $\mathcal{F}_{\mathsf{Ident}}$ to verify outputs, we consider $\mathcal{F}_{\mathsf{Ident}}$ as a global functionality. This does, however, not change anything concerning it's implementation or security proof, as $\mathcal{F}_{\mathsf{Ident}}$ does not keep a common state across multiple sessions and ignores requests from other sessions.

**Functionality $\mathcal{F}_{\mathsf{Sig}}$**

Given ideal adversary $\mathcal{S}$, verifiers $\mathcal{V}$ and a signer $\mathcal{P}_s$, $\mathcal{F}_{\mathsf{Sig}}$ performs:

**Key Generation:** Upon receiving a message $(\text{KEYGEN}, sid)$ from $\mathcal{P}_s$, verify that $sid = (\mathcal{P}_s, sid')$ for some $sid'$. If not, ignore the request. Else, hand $(\text{KEYGEN}, sid)$ to the adversary $\mathcal{S}$. Upon receiving $(\text{VERIFICATION KEY}, sid, \mathsf{SIG.vk})$ from $\mathcal{S}$, output $(\text{VERIFICATION KEY}, sid, \mathsf{SIG.vk})$ to $\mathcal{P}_s$, and record the pair $(\mathcal{P}_s, \mathsf{SIG.vk})$.

**Signature Generation:** Upon receiving a message $(\text{SIGN}, sid, m)$ from $\mathcal{P}_s$, verify that $sid = (\mathcal{P}_s, sid')$ for some $sid'$ . If not, then ignore the request. Else, send $(\text{SIGN}, sid, m)$ to $\mathcal{S}$. Upon receiving $(\text{SIGNATURE}, sid, m, \sigma)$ from $\mathcal{S}$, verify that no entry $(m, \sigma, \mathsf{SIG.vk}, 0)$ is recorded. If it is, then output an error message to $\mathcal{P}_s$ and halt. Else, output $(\text{SIGNATURE}, sid, m, \sigma)$ to $\mathcal{P}_s$, and record the entry $(m, \sigma, \mathsf{SIG.vk}, 1)$.

**Signature Verification:** Upon receiving a message $(\text{VERIFY}, sid, m, \sigma, \mathsf{SIG.vk}')$ from some party $\mathcal{V}_i \in \mathcal{V}$, hand $(\text{VERIFY}, sid, m, \sigma, \mathsf{SIG.vk}')$ to $\mathcal{S}$. Upon receiving $(\text{VERIFIED}, sid, m, \phi)$ from $\mathcal{S}$ do:

1. If $\mathsf{SIG.vk}' = \mathsf{SIG.vk}$ and the entry $(m, \sigma, \mathsf{SIG.vk}, 1)$ is recorded, then set $f = 1$. (This condition guarantees completeness: If the verification key $\mathsf{SIG.vk}'$ is the registered one and $\sigma$ is a legitimately generated signature for $m$, then the verification succeeds.)
2. Else, if $\mathsf{SIG.vk}' = \mathsf{SIG.vk}$, the signer $\mathcal{P}_s$ is not corrupted, and no entry $(m, \sigma', \mathsf{SIG.vk}, 1)$ for any $\sigma'$ is recorded, then set $f = 0$ and record the entry $(m, \sigma, \mathsf{SIG.vk}, 0)$. (This condition guarantees unforgeability: If $\mathsf{SIG.vk}'$ is the registered one, the signer is not corrupted, and never signed $m$, then the verification fails.)
3. Else, if there is an entry $(m, \sigma, \mathsf{SIG.vk}', f')$ recorded, then let $f = f'$. (This condition guarantees consistency: All verification requests with identical parameters will result in the same answer.)
4. Else, let $f = \phi$ and record the entry $(m, \sigma, \mathsf{SIG.vk}', \phi)$.

Output $(\text{VERIFIED}, sid, m, f)$ to $\mathcal{V}_i$.

Fig. 7: Functionality $\mathcal{F}_{\mathsf{Sig}}$ for Digital Signatures [18].

### A.4 Standard Digital Signatures

The standard UC ideal functionality for digital signatures $\mathcal{F}_{\mathsf{Sig}}$ from [18] is presented in Fig. 7.

**Functionality $\mathcal{F}_{\mathsf{TSig}}$**

$\mathcal{F}_{\mathsf{TSig}}$ is parameterized with an ideal adversary $\mathcal{S}$, a set of signers $\mathcal{P}$ and functionalities $\mathcal{F}$, a verifiers $\mathcal{V}$ (which automatically contains $\mathcal{P}$ and $\mathcal{F}$) and a set of corrupted signers $I \subset \mathcal{P}$. $\mathcal{F}_{\mathsf{TSig}}$ has two internal lists $\mathtt{Sh}$ and $\mathtt{Sig}$.

**Key Generation:** Upon receiving a message $(\text{KEYGEN}, sid)$ from each $\mathcal{P}_i \in \mathcal{P}$ or a functionality $\mathcal{F}_j \in \mathcal{F}$ hand $(\text{KEYGEN}, sid)$ to the adversary $\mathcal{S}$. Upon receiving $(\text{VERIFICATION KEY}, sid, \mathsf{SIG.vk})$ from $\mathcal{S}$, if $(\cdot, \mathsf{SIG.vk})$ was not recorded yet then output $(\text{VERIFICATION KEY}, sid, \mathsf{SIG.vk})$ to each $\mathcal{P}_i \in \mathcal{P}$ (or to $\mathcal{F}_j$), and record the pair $(\mathcal{P}, \mathsf{SIG.vk})$. If $\mathsf{SIG.vk}$ was recorded before then output $(\text{ABORT}, sid)$ to $\mathcal{S}$ and stop.

**Share Generation:** Upon receiving a message $(\text{SIGN}, sid, m, \mathsf{SIG.vk})$ from all honest parties or a functionality $\mathcal{F}_j \in \mathcal{F}$ send $(\text{SIGN}, sid, m)$ to $\mathcal{S}$. Upon receiving $(\text{SIGNATURE}, sid, m, \rho, \sigma, J, f)$ from $\mathcal{S}$, verify that
- no entry $(m, \rho, J', \mathsf{SIG.vk}')$ with $J' \neq J$ is recorded in $\mathtt{Sh}$, and
- no entry $(m, \sigma, \mathsf{SIG.vk}, 0)$ is recorded in $\mathtt{Sig}$ if $J = \emptyset$.

If either is, then output an error message to $\mathcal{S}$ and halt. Else, let $f' = 1$ if $J = \emptyset$ and $f' = f$ otherwise, record the entry $(m, \rho, J, \mathsf{SIG.vk})$ in $\mathtt{Sh}$, $(m, \sigma, \mathsf{SIG.vk}, f')$ in $\mathtt{Sig}$ and return $(\text{SHARES}, sid, m, \rho)$.

**Share Combination:** Upon receiving a message $(\text{COMBINE}, sid, m, \rho, \mathsf{SIG.vk})$ from any party in $\mathcal{P}$ or functionality $\mathcal{F}_j \in \mathcal{F}$, find $(m, \rho, J, \mathsf{SIG.vk})$ in $\mathtt{Sh}$ and $(m, \sigma, \mathsf{SIG.vk}, b)$ in $\mathtt{Sig}$. If $J \neq \emptyset$ then return $(\text{FAILURE}, sid, J)$. If $J = \emptyset$ return $(\text{COMBINED}, sid, m, \sigma, \mathsf{SIG.vk})$. If no entry could be found in $\mathtt{Sh}$ and $\mathtt{Sig}$ then return $(\text{NOT-GENERATED}, sid)$.

**Signature Verification:** Upon receiving a message $(\text{VERIFY}, sid, m, \sigma, \mathsf{SIG.vk}')$ from some entity in $\mathcal{V}$, hand $(\text{VERIFY}, sid, m, \sigma, \mathsf{SIG.vk}')$ to $\mathcal{S}$. Upon receiving $(\text{VERIFIED}, sid, m, \phi)$ from $\mathcal{S}$ do:

1. If $\mathsf{SIG.vk}' = \mathsf{SIG.vk}$ and $(m, \sigma, \mathsf{SIG.vk}, 1) \in \mathtt{Sig}$, then set $f = 1$.
2. Else, if $\mathsf{SIG.vk}' = \mathsf{SIG.vk}$ and $(m, \sigma', \mathsf{SIG.vk}, 1) \notin \mathtt{Sig}$ for any $\sigma'$, then set $f = 0$ and record the entry $(m, \sigma, \mathsf{SIG.vk}, 0)$ in $\mathtt{Sig}$.
3. Else, if there is an entry $(m, \sigma, \mathsf{SIG.vk}', f') \in \mathtt{Sig}$ recorded, then let $f = f'$.
4. Else, let $f = \phi$ and record the entry $(m, \sigma, \mathsf{SIG.vk}', \phi)$ in $\mathtt{Sig}$.

Return $(\text{VERIFIED}, sid, m, f)$.

Fig. 8: Functionality $\mathcal{F}_{\mathsf{TSig}}$ for Threshold Signatures.

### A.5 Threshold Digital Signatures

Functionality $\mathcal{F}_{\mathsf{TSig}}$ is defined in Fig. 8. The functionality is tailored to threshold $n$, meaning that the adversary may control up to $n - 1$ parties but will still

not be able to forge signatures. This coincides with the security model for the MPC scheme $\mathcal{F}_{\mathsf{Ident}}$ that we use. $\mathcal{F}_{\mathsf{TSig}}$ exposes a behavior such as $\mathcal{F}_{\mathsf{Sig}}$, but it additionally allows $\mathcal{S}$ to choose the string of shares that later get combined into a signature (although $\mathcal{S}$ has to choose both the shares $\rho$ and the signature $\sigma$ together). Parties can learn $\sigma$ from $\rho$ via **Share Combination**, but cheating during **Share Generation** leads to dishonest parties being exposed when trying to combine the values from $\rho$. Observe that the actual choice of $\rho$ binds $\mathcal{S}$ to a certain consistent set of dishonest parties that are exposed via **Share Combination**. In $\mathcal{F}_{\mathsf{TSig}}$ $\mathcal{S}$ will always be able to make **Share Generation** output dishonest shares that lead to a signature that is valid (if the set of cheating parties $J$ is empty), but it never can make an invalid signature in an honest instance of **Share Generation**.

## Appendix B Smart Contract Functionality $\mathcal{F}_{\mathsf{SC}}$

We now describe the smart contract on a high level, meaning its different states and state transitions. This is to ease understanding, the full description will be presented later. The Smart Contract will have 7 different states `init`, `ready`, `abort`, `ok1`, `ok2`, `reimburse1`, `reimburse2` where `init` is the initial state. State transitions are performed whenever the global clock $\mathcal{F}_{\mathsf{Clock}}$ changes and depending on the messages that are present on the ledger that $\mathcal{F}_{\mathsf{SC}}$ acts upon.

`init` If a tick happens, then check if all servers signed the same $\mathsf{vk}, \widehat{\mathsf{vk}}_1, \ldots, \widehat{\mathsf{vk}}_n$ using their individual $\widehat{\mathsf{sk}}_i$ and that $\mathcal{P}_i$ sent $\mathsf{coins}(d)$. If so then change state to `ready`, otherwise reimburse all servers and stay in `init`.

`ready` If a tick happens and a message "done" is present, signed by $\mathsf{sk}$, then reimburse all servers and set the state to `init`. If a tick happens and a message "abort" is present, signed by a $\widehat{\mathsf{sk}}_i$ that initialized the contract, then change the state to `abort`.

`abort` If a tick happens and a message "done" is present, signed by $\mathsf{sk}$, then reimburse all servers and set the state to `init`. Else, if a tick happens and a message "ok", signed by $\mathsf{sk}$, is present, then change state to `ok1`. Else, if no such message is present at the tick, then change to `reimburse1`.

`ok1` Call **Test Reveal** on each $\mathcal{F}_{\mathsf{Ident}}^{a,b}$. If no parties $J$ are returned as cheaters by **Test Reveal** then check if for each $\mathcal{P}_i$ and each $\mathcal{F}_{\mathsf{Ident}}^{a,b}$ a message $\mathbf{s}_i^{a,b}$ signed by $\widehat{\mathsf{sk}}_i$ is present. If indeed, then verify the output for each $\mathcal{F}_{\mathsf{Ident}}^{a,b}$ using **Verify**. If any of the aforementioned steps fails, then let $I_1$ be the set of cheating servers.
If $I_1 = \emptyset$ then change the state to `ok2`. If $I_1 \neq \emptyset$ then identify all the $m$ clients by finding all messages of the form $(\mathcal{C}_j|\mathcal{L}_j^{\mathsf{src}}|\mathsf{am}_j^{\mathsf{src}}|\mathsf{vk}_j^{\mathsf{src}}|\mathsf{vk}_j^{\mathsf{ex}}|\mathcal{L}_j^{\mathsf{trg}}|\mathsf{vk}_j^{\mathsf{trg}})$ that are signed by $\mathsf{sk}$. Furthermore, identify all the transaction ids $\mathtt{id}_j$ to burner addresses $\mathsf{vk}_j^{\mathsf{ex}}$ signed by $\mathsf{sk}$. For each party in $I_1$ share the deposit among all $m$ clients. Then return the deposit of the parties in $[n] \setminus I_1$ and change the state to `init`.

`ok2` Compute for each $\mathcal{L}_a$ in clear text the values $\mathtt{id}_a, \mathsf{In}_a, \mathsf{Out}_a$ from the outputs of each $\mathcal{F}_{\mathsf{Ident}}^{a,b}$ as well as the client registration data and transaction ids using `makeTX`. For each $\mathcal{L}_a$ check if each $\mathcal{P}_i$ sent shares of $\mathsf{Sig}_a$ signed with $\widehat{\mathsf{sk}}_i$. If so, then check that each share of $\mathsf{Sig}_a$ is valid using $\mathcal{F}_{\mathsf{TSig}}$ by running **Share Combination**. If any of the previous steps fails, then let $I_2$ be the set of cheaters.

If $I_2 = \emptyset$ then reimburse all $\mathcal{P}_i$ with their deposit and change the state to `init`. Otherwise identify all the $m$ clients by finding all client registration data and transaction ids to burner addresses signed by $\mathsf{sk}$. For each party in $I_2$ share the deposit among all $m$ clients. Then return the deposit of the parties in $[n] \setminus I_2$ and change the state to `init`.

`reimburse1` If a tick happens, then continue to `reimburse2`. Intuitively, during this step all clients that get reimbursed are already fixed so the servers will create the signatures on reimbursement transactions.

`reimburse2` If a tick happens, consider all messages provided by each $\mathcal{P}_i$ to $\mathcal{F}_{\mathsf{SC}}$ that are of the form $(\mathcal{C}_j|\mathcal{L}_j^{\mathsf{src}}|\mathsf{am}_j^{\mathsf{src}}|\mathsf{vk}_j^{\mathsf{src}}|\mathsf{vk}_j^{\mathsf{ex}}|\mathcal{L}_j^{\mathsf{trg}}|\mathsf{vk}_j^{\mathsf{trg}})$ that are signed by $\mathsf{sk}$ as well as all messages $(\overline{\mathtt{id}}_j|\overline{\mathsf{In}}_j|\overline{\mathsf{Out}}_j, \mathsf{vk}_j^{\mathsf{src}}) \in \mathcal{M}$ (transaction ids) where $\overline{\mathsf{In}}_j = (\mathsf{vk}_j^{\mathsf{src}}, \mathsf{am}_j^{\mathsf{src}})$ and $\overline{\mathsf{Out}}_j = (\mathsf{am}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{ex}})$. If there are multiple messages for $\mathcal{C}_j$ then ignore $\mathcal{C}_j$

Locally compute for each $\mathcal{C}_j$ the transaction $\mathsf{tx}_j$ to reimburse $\mathcal{C}_j$. Therefore set $\mathsf{In}_j = (\overline{\mathtt{id}}_j, \mathsf{am}_j^{\mathsf{src}})$, $\mathsf{Out}_j = (\mathsf{am}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{src}})$ and set $\mathtt{id}_j$ as the hash of both. If each $\mathcal{P}_i^j$ provided $\rho_j^i$ then check using **Share Combination** on $\mathcal{F}_{\mathsf{TSig}}$ that it outputs a valid signature $\mathsf{Sig}_j$ on $\mathtt{id}_j, \mathsf{In}_j, \mathsf{Out}_j$. If all such $\mathsf{Sig}_j$ are valid signatures then reimburse all $\mathcal{P}_i$ and set the state to `init`. If some signature shares are not valid or some shares $\rho_j^i$ are not present on $\mathcal{F}_{\mathsf{SC}}$ then let $J$ be the set of cheaters. Reimburse all servers $[n] \setminus J$ and distribute the deposit of the parties of $J$ evenly to all $\mathcal{C}_j$. Then set the state to `init`.

**Formalizing the Smart Contract** We use a combined smart contract and public ledger functionality $\mathcal{F}_{\mathsf{SC}}$. It is an extension to $\mathcal{F}_{\mathsf{BB}}$, tailored to be combined with an MPC protocol and similar to the functionality used in [5]. For technical reasons, $\mathcal{F}_{\mathsf{SC}}$ has a hard-coded reference to the publicly verifiable MPC functionality $\mathcal{F}_{\mathsf{Ident}}$ in order to be able to verify outputs. $\mathcal{F}_{\mathsf{SC}}$ is described in Fig. 9 and Fig. 10 and considered as an ordinary UC functionality in our work. Again, this is due to technical limitations of UC, which would not make it possible for the simulator we construct in our security proof to equivocate the necessary outputs.

**Functionality $\mathcal{F}_{\mathsf{SC}}$** (Part 1)

$\mathcal{F}_{\mathsf{SC}}$ interacts with the global functionalities $\mathcal{F}_{\mathsf{Ident}}, \mathcal{F}_{\mathsf{Clock}}, \mathcal{F}_{\mathsf{Sig}}, \mathcal{F}_{\mathsf{TSig}}$. It is parameterized by the compensation $q$, the maximal number of exchange clients $m$ and the security deposit $d = m \cdot q$. $\mathcal{F}_{\mathsf{SC}}$ has an initially empty list $\mathcal{M}$ of messages posted to the authenticated bulletin board and a public state $\mathsf{state}$ which is initially $\mathtt{init}$. It furthermore has the sets $\mathcal{P}$ of servers which is initially empty. Upon each activation $\mathcal{F}_{\mathsf{SC}}$ first sends a message (READ, $sid$) to $\mathcal{F}_{\mathsf{Clock}}$. If $\nu$ has changed since the last call to $\mathcal{F}_{\mathsf{Clock}}$ then it does the following:

$\mathsf{state} = \mathtt{init}$**:** Check if $((\mathsf{vk}, \{\widehat{\mathsf{vk}}_i\}_{i\in[n]}, \{\mathcal{L}_j\}_{j\in[\ell]}, \{\mathcal{F}^{a,b}_{\mathsf{Ident}}\}^{a<b}_{a,b\in[\ell]}), \widehat{\mathsf{vk}}_i) \in \mathcal{M}$ for all $i \in [n]$ via $\mathcal{F}_{\mathsf{Sig}}$. Furthermore check that each $\mathcal{P}_i$ with key $\widehat{\mathsf{vk}}_i$ sent $\mathsf{coins}(d)$. If so then change $\mathsf{state}$ to $\mathtt{ready}$ and set $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$. If not, for $\mathcal{P}'$ as the parties that provided $\mathsf{coins}(d)$ send $\mathsf{coins}(d)$ to each party in $\mathcal{P}'$ and let $\mathsf{state} = \mathtt{init}$.

$\mathsf{state} = \mathtt{ready}$**:** If $(done, \mathsf{vk}) \in \mathcal{M}$ then run $\mathsf{CC}(\emptyset, \emptyset)$ and set $\mathsf{state}$ to $\mathtt{init}$. Else if $(abort, \widehat{\mathsf{vk}}_i) \in \mathcal{M}$ for some $\mathcal{P}_i \in \mathcal{P}$ then change $\mathsf{state}$ to $\mathtt{abort}$. Else do not change $\mathsf{state}$.

$\mathsf{state} = \mathtt{abort}$**:** If $(done, \mathsf{vk}) \in \mathcal{M}$ then run $\mathsf{CC}(\emptyset, \emptyset)$ and set $\mathsf{state}$ to $\mathtt{init}$. Else, if $(ok, \mathsf{vk}) \in \mathcal{M}$ then change $\mathsf{state}$ to $\mathtt{ok1}$. Else change $\mathsf{state}$ to $\mathtt{reimburse1}$.

$\mathsf{state} = \mathtt{ok1}$**:** Parse all messages $(\mathcal{C}_j|\mathcal{L}^{\mathsf{src}}_j|\mathsf{am}^{\mathsf{src}}_j|\mathsf{vk}^{\mathsf{src}}_j|\mathsf{vk}^{\mathsf{ex}}_j|\mathcal{L}^{\mathsf{trg}}_j|\mathsf{vk}^{\mathsf{trg}}_j, \mathsf{vk}) \in \mathcal{M}$ ( client registrations) and all $(\mathcal{C}_j|\overline{\mathtt{id}}_j, \mathsf{vk}) \in \mathcal{M}$ (transaction ids). Then call **Test Reveal** on each $\mathcal{F}^{a,b}_{\mathsf{Ident}}$. Afterwards check if $(\mathbf{s}^{a,b}_i, \widehat{\mathsf{vk}}_i) \in \mathcal{M}$ for each $\mathcal{P}_i$ and each $\mathcal{F}^{a,b}_{\mathsf{Ident}}$ and send (VERIFY, $sid, \mathbf{s}^{a,b}_1, \ldots, \mathbf{s}^{a,b}_n$) to each $\mathcal{F}^{a,b}_{\mathsf{Ident}}$. If all passes, then let $\mathbf{y}^{a,b} = \sum_{i\in[n]} \mathbf{s}^{a,b}_i$ be the output of $\mathcal{F}^{a,b}_{\mathsf{Ident}}$. If any of the aforementioned steps fails, then let $I_1$ be the set of cheaters.
If $I_1 = \emptyset$ then set $\mathsf{state}$ to $\mathtt{ok2}$, else let $\mathcal{C}$ be the set of clients, run $\mathsf{CC}(I_1, \mathcal{C})$ and change $\mathsf{state}$ to $\mathtt{init}$.

$\mathsf{state} = \mathtt{ok2}$**:** For each ledger $\mathcal{L}_a$ compute $\mathtt{id}_a, \mathsf{In}_a, \mathsf{Out}_a$ from $\mathbf{y}^{a,b}$ as well as the client registrations and transaction ids using $\mathtt{makeTX}$. Let $\mathsf{vk}^{\mathsf{ex}}_{a,1}, \ldots, \mathsf{vk}^{\mathsf{ex}}_{a,r}$ be the source transactions in $\mathtt{in}_a$. For each $\mathcal{L}_a$ check if $((\rho^{a,1}_i, \ldots, \rho^{a,r}_i), \widehat{\mathsf{vk}}_i) \in \mathcal{M}$ and send (COMBINE, $sid, \mathtt{id}_a|\mathsf{In}_a|\mathsf{Out}_a, \rho^{a,c}_i, \mathsf{vk}^{\mathsf{ex}}_{a,c}$) to $\mathcal{F}_{\mathsf{TSig}}$ for each $c \in [r]$ and $i \in [n]$. If any of the steps fails, then let $I_2$ be the set of cheaters.
Let $\mathcal{C}$ be the set of clients. If $I_2 = \emptyset$ then run $\mathsf{CC}(\emptyset, \emptyset)$, otherwise run $\mathsf{CC}(I_2, \mathcal{C})$. Finally change $\mathsf{state}$ to $\mathtt{init}$.

$\mathsf{state} = \mathtt{reimburse1}$**:** Change $\mathsf{state}$ to $\mathtt{reimburse2}$.

$\mathsf{state} = \mathtt{reimburse2}$**:** Parse all messages $(\mathcal{C}_j|\mathcal{L}^{\mathsf{src}}_j|\mathsf{am}^{\mathsf{src}}_j|\mathsf{vk}^{\mathsf{src}}_j|\mathsf{vk}^{\mathsf{ex}}_j|\mathcal{L}^{\mathsf{trg}}_j|\mathsf{vk}^{\mathsf{trg}}_j, \mathsf{vk}) \in \mathcal{M}$ (i.e. client registrations) as well as $(\overline{\mathtt{id}}_j|\overline{\mathsf{In}}_j|\overline{\mathsf{Out}}_j, \mathsf{vk}^{\mathsf{src}}_j) \in \mathcal{M}$ (transaction ids) where $\overline{\mathsf{In}}_j = (\mathsf{vk}^{\mathsf{src}}_j, \mathsf{am}^{\mathsf{src}}_j)$ and $\overline{\mathsf{Out}}_j = (\mathsf{am}^{\mathsf{src}}_j, \mathsf{vk}^{\mathsf{ex}}_j)$. If there are multiple messages for $\mathcal{C}_j$ then ignore $\mathcal{C}_j$.
Compute for $\mathcal{C}_j$ the values $\mathtt{id}_j, \mathsf{In}_j, \mathsf{Out}_j$ for reimbursement by setting $\mathsf{In}_j = (\overline{\mathtt{id}}_j, \mathsf{am}^{\mathsf{src}}_j), \mathsf{Out}_j = (\mathsf{am}^{\mathsf{src}}_j, \mathsf{vk}^{\mathsf{src}}_j)$ and $\mathtt{id}_j$ as the hash of $\mathsf{In}_j, \mathsf{Out}_j$.
Check if $(\rho^i_j, \widehat{\mathsf{vk}}_i) \in \mathcal{M}$ for each $\mathcal{C}_j$ and each $\mathcal{P}_i$, then send (COMBINE, $sid, \mathtt{id}_j|\mathsf{In}_j|\mathsf{Out}_j, \rho^i_j, \mathsf{vk}^{\mathsf{ex}}_j$) to $\mathcal{F}_{\mathsf{TSig}}$ for each $i \in [n]$. If all values were present on $\mathcal{M}$ and all queries to $\mathcal{F}_{\mathsf{TSig}}$ were positive then run $\mathsf{CC}(\emptyset, \emptyset)$ and set $\mathsf{state}$ to $\mathtt{init}$. Otherwise let $J$ be the set of cheaters and $\mathcal{C}$ be the set of all clients. Run $\mathsf{CC}(J, \mathcal{C})$ and set $\mathsf{state}$ to $\mathtt{init}$.

Afterwards it executes the operation and finalizes by sending (UPDATE, $sid$) to $\mathcal{F}_{\mathsf{Clock}}$.

Fig. 9: The stateful smart contract functionality $\mathcal{F}_{\mathsf{SC}}$.

**Functionality $\mathcal{F}_{\mathsf{SC}}$** (Part 2)

**Post:** Upon receiving $(\textsc{Post}, sid, m, \mathsf{vk}, \sigma)$ from some entity contact the instance of $\mathcal{F}_{\mathsf{Sig}}$ or $\mathcal{F}_{\mathsf{TSig}}$ belonging to $\mathsf{vk}$. If $\sigma$ verifies for $m$ and $\mathsf{vk}$ then send $(\textsc{Post}, sid, m, \mathsf{vk}, \sigma)$ to $\mathcal{S}$ and append $(m, \mathsf{vk})$ to the list $\mathcal{M}$.
**Read:** Upon receiving $(\textsc{Read}, sid)$ from some entity, return $\mathcal{M}$.
**Send Deposits:** Upon receiving $(\textsc{Lock-In}, sid, \mathsf{coins}(d))$ from some entity $\mathcal{P}_i$ containing the $d$ coins of the security deposit send $(\textsc{Lock-In}, sid, \mathsf{coins}(d))$ to $\mathcal{S}$. Then if $\mathsf{state} = \mathtt{init}$ accept the money, otherwise return it to $\mathcal{P}_i$.
**Macro** $\mathsf{CC}(\mathtt{punish}, \mathcal{C})$**:** Let $\mathtt{punish} \subset \mathcal{P}$, $|\mathcal{C}| \leq m$ and $\mathtt{reimburse} = \mathcal{P} \setminus \mathtt{punish}$. Send $\mathsf{coins}(d)$ to each party $\mathtt{reimburse}$ and $\mathsf{coins}(|\mathtt{punish}| \cdot d/|\mathcal{C}|)$ to each party in $\mathcal{C}$.

Fig. 10: The stateful smart contract functionality $\mathcal{F}_{\mathsf{SC}}$.

**Protocol $\Pi_{\mathsf{Ex}}$** (Part 1)

We have $n$ servers $\mathcal{P}$ and $m$ clients $\mathcal{C}$. The clients exchange between $\ell$ ledgers $\mathcal{L}$ known to $\mathcal{P}$. Each $\mathcal{C}_j$ starts with an amount of asset $\mathsf{am}_j^{\mathsf{src}}$ signed by $\mathsf{vk}_j^{\mathsf{src}}$ on ledger $\mathcal{L}_j^{\mathsf{src}}$. The goal of $\mathcal{C}_j$ is to acquire $\mathsf{am}_j^{\mathsf{trg}}$ on $\mathcal{L}_j^{\mathsf{trg}}$ by a transfer to $\mathsf{vk}_j^{\mathsf{trg}}$.
The protocol runs in the presence of $\mathcal{F}_{\mathsf{Clock}}, \mathcal{F}_{\mathsf{SC}}$ and multiple instances of $\mathcal{F}_{\mathsf{Sig}}, \mathcal{F}_{\mathsf{TSig}}$.

**Initialize:**

1. All $\mathcal{P}$ set up $\ell \cdot (\ell - 1)$ instances $\mathcal{F}_{\mathsf{Ident}}^{a,b}$ where $1 \leq a < b \leq \ell$ to accept inputs by clients who want to transfer between $\mathcal{L}_a$ and $\mathcal{L}_b$ and run **Init** on each $\mathcal{F}_{\mathsf{Ident}}^{a,b}$.
2. The servers $\mathcal{P}$ use $\mathcal{F}_{\mathsf{TSig}}$ to sample a common key $\mathsf{vk}$. Furthermore each $\mathcal{P}_i$ uses $\mathcal{F}_{\mathsf{Sig}}$ to generate an individual public verification key $\widehat{\mathsf{vk}}_i$, which it shares with all $\mathcal{P}$.
3. Let $t = (\mathsf{vk}, \{\widehat{\mathsf{vk}}_i\}_{i\in[n]}, \{\mathcal{L}_j\}_{j\in[\ell]}, \{\mathcal{F}_{\mathsf{Ident}}^{a,b}\}_{a,b\in[\ell]}^{a<b})$. If $\mathcal{F}_{\mathsf{SC}}$ is in state $\mathtt{init}$ then each $\mathcal{P}_i$ first computes a signature $\sigma_{\widehat{\mathsf{vk}}_i}(t)$ on $t$ using $\mathcal{F}_{\mathsf{Sig}}$ for the key $\widehat{\mathsf{vk}}_i$ and then sends $(\textsc{Post}, sid, t, \widehat{\mathsf{vk}}_i, \sigma_{\widehat{\mathsf{vk}}_i}(t))$ to $\mathcal{F}_{\mathsf{SC}}$. Afterwards, each $\mathcal{P}_i$ sends $(\textsc{Lock-In}, sid, \mathsf{coins}(d))$ to $\mathcal{F}_{\mathsf{SC}}$ updates $\mathcal{F}_{\mathsf{Clock}}$.

**Enroll Client:** Upon message $(\textsc{Enroll}, sid, \mathsf{am}_j^{\mathsf{src}}, \mathcal{L}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{src}}, \mathcal{L}_j^{\mathsf{trg}}, \mathsf{vk}_j^{\mathsf{trg}})$ by $\mathcal{C}_j$:

1. If $\mathsf{state} = \mathtt{ready}$ then all $\mathcal{P}_i$ use $\mathcal{F}_{\mathsf{TSig}}$ to generate a fresh key $\mathsf{vk}_j^{\mathsf{ex}}$ on $\mathcal{L}_j^{\mathsf{src}}$. Let $t_j = (\mathcal{C}_j, \mathcal{L}_j^{\mathsf{src}}, \mathsf{am}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{ex}}, \mathcal{L}_j^{\mathsf{trg}}, \mathsf{vk}_j^{\mathsf{trg}})$. Then the servers use $\mathcal{F}_{\mathsf{TSig}}$ to generate a signature $\sigma_{\mathsf{vk}}(t_j)$ for the key $\mathsf{vk}$. All $\mathcal{P}_i$ send $(\textsc{Ok}, sid, t_j, \sigma_{\mathsf{vk}}(t_j))$ to $\mathcal{C}_j$.
2. Upon receiving $(\textsc{Ok}, sid, t_j, \sigma_{\mathsf{vk}}(t_j))$ $\mathcal{C}_j$ checks if $\sigma_{\mathsf{vk}}(t_j)$ is valid using $\mathsf{vk}$ from $\mathcal{F}_{\mathsf{SC}}$. Then it creates a transaction $\overline{\mathsf{tx}}_j = (\overline{\mathsf{id}}_j, \overline{\mathsf{In}}_j, \overline{\mathsf{Out}}_j, \overline{\mathsf{Sig}}_j)$ where $\overline{\mathsf{id}}_j$ is fresh for $\mathcal{L}_j^{\mathsf{src}}$, $\overline{\mathsf{In}}_j = (\mathtt{id}_j, \mathsf{am}_j^{\mathsf{src}})$, $\overline{\mathsf{Out}}_j = (\mathsf{am}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{ex}})$ and $\overline{\mathsf{Sig}}_j = \sigma_{\mathsf{vk}_j^{\mathsf{src}}}(\mathtt{id}|\mathsf{In}|\mathsf{Out})$ where $\mathtt{id}$ is the id of an unspent transaction for $\mathsf{vk}_j^{\mathsf{src}}$ and the signature is produced by $\mathcal{F}_{\mathsf{Sig}}$ for the key $\mathsf{vk}_j^{\mathsf{src}}$. Finally, $\mathcal{C}_j$ sends $\overline{\mathsf{tx}}_j$ to $\mathcal{F}_{\mathsf{BB}}$ corresponding to $\mathcal{L}_j^{\mathsf{src}}$.
3. Let $\mathcal{L}_a = \mathcal{L}_j^{\mathsf{src}}$ and $\mathcal{L}_b = \mathcal{L}_j^{\mathsf{trg}}$. Each $\mathcal{C}_j$ sends $(\textsc{Input}, sid, j, (\mathcal{C}_j, 0, \mathsf{am}_j^{\mathsf{src}}, \mathsf{am}_j^{\mathsf{trg}}))$ to $\mathcal{F}_{\mathsf{Ident}}^{a,b}$ if $a < b$, otherwise it sends $(\textsc{Input}, sid, j, (\mathcal{C}_j, 1, \mathsf{am}_j^{\mathsf{src}}, \mathsf{am}_j^{\mathsf{trg}}))$ to $\mathcal{F}_{\mathsf{Ident}}^{b,a}$.

Fig. 11: The protocol $\Pi_{\mathsf{Ex}}$.

**Protocol $\Pi_{\mathsf{Ex}}$** (Part 2)

**Exchange:** Upon (EXCHANGE, *sid*) by all servers:

1. Each $\mathcal{P}_i$ checks that each $\mathcal{C}_j$ provided input to its respective $\mathcal{F}_{\mathsf{Ident}}^{a,b}$ and that $\overline{\mathsf{tx}}_j$ has been approved (with the right amount of asset) on the respective $\mathcal{F}_{\mathsf{BB}}$-instance. If this is the case then all $\mathcal{P}$ use $\mathcal{F}_{\mathsf{TSig}}$ to generate a signature $\sigma_{\mathsf{vk}}(u_j)$ with $u_j = (\mathcal{C}_j, \overline{\mathtt{id}}_j)$ for the transaction id $\overline{\mathtt{id}}_j$ of $\overline{\mathsf{tx}}_j$. If the transaction is not present or the signing with $\mathcal{F}_{\mathsf{TSig}}$ fails then $\mathcal{P}_i$ uses $\mathcal{F}_{\mathsf{Sig}}$ to generate the signature $\sigma_{\widehat{\mathsf{vk}}_i}(u_j)$ and sends (POST, *sid*, *abort*, $\widehat{\mathsf{vk}}_i, \sigma_{\widehat{\mathsf{vk}}_i}(u_j)$) to $\mathcal{F}_{\mathsf{SC}}$. It then updates $\mathcal{F}_{\mathsf{Clock}}$, sends (ABORT, *sid*) to all $\mathcal{P}$ and enters **Abort**.
2. $\mathcal{P}_i$ runs **Evaluate** and **Share** on each $\mathcal{F}_{\mathsf{Ident}}^{a,b}$ where the circuit evaluated runs `compSwap` for the registered number of parties. If $\mathcal{P}_i$ obtained an abort from any $\mathcal{F}_{\mathsf{Ident}}^{a,b}$ then it sends (POST, *sid*, *abort*, $\widehat{\mathsf{vk}}_i, \sigma_{\widehat{\mathsf{vk}}_i}(u_j)$) to $\mathcal{F}_{\mathsf{SC}}$, updates $\mathcal{F}_{\mathsf{Clock}}$, sends (ABORT, *sid*) to all $\mathcal{P}$ and enters **Abort**.
3. If all of this succeeds, then all $\mathcal{P}_i$ compute $\sigma_{\mathsf{vk}}(ok)$ using $\mathcal{F}_{\mathsf{TSig}}$ and send it to all servers. If $\mathcal{P}_i$ did not obtain the signature then it sends (POST, *sid*, *abort*, $\widehat{\mathsf{vk}}_i, \sigma_{\widehat{\mathsf{vk}}_i}(u_j)$) to $\mathcal{F}_{\mathsf{SC}}$, updates $\mathcal{F}_{\mathsf{Clock}}$, sends (ABORT, *sid*) to all $\mathcal{P}$ and enters **Abort**.
4. If no abort message was obtained then each $\mathcal{P}_i$ updates $\mathcal{F}_{\mathsf{Clock}}$. Then it checks $\mathsf{state}$ of $\mathcal{F}_{\mathsf{SC}}$. If $\mathsf{state} = \mathtt{abort}$ then it runs **Abort**, else it runs **Open**.

**Open:**

1. Each $\mathcal{P}_i$ runs **Optimistic Reveal** for each $\mathcal{F}_{\mathsf{Ident}}^{a,b}$. If a server $\mathcal{P}_i$ obtained an abort from any $\mathcal{F}_{\mathsf{Ident}}^{a,b}$ then it sends (POST, *sid*, *abort*, $\widehat{\mathsf{vk}}_i, \sigma_{\widehat{\mathsf{vk}}_i}(u_j)$) to $\mathcal{F}_{\mathsf{SC}}$, updates $\mathcal{F}_{\mathsf{Clock}}$, sends (ABORT, *sid*) to all $\mathcal{P}$ and enters **Abort**.
2. If no abort was received then each $\mathcal{P}_i$ obtained $\mathbf{y}^{a,b} \in \mathbb{F}^g$ for each $\mathcal{F}_{\mathsf{Ident}}^{a,b}$. For each $\mathcal{L}_a$ $\mathcal{P}_i$ then computes $\mathsf{tx}_a = (\mathtt{id}_a, \mathsf{In}_a, \mathsf{Out}_a, \mathsf{Sig}_a)$ as follows:
   (a) Compute $(\mathtt{id}_a, \mathsf{In}_a, \mathsf{Out}_a)$ using `makeTX` from all $\mathbf{y}^{a,b}$ as well as all client registrations and $\overline{\mathtt{id}}_j$.
   (b) Compute $\mathsf{Sig}_a = \sigma_{\mathsf{vk}_c^{\mathsf{ex}}}(\mathtt{id}_a | \mathsf{In}_a | \mathsf{Out}_a)$ using $\mathcal{F}_{\mathsf{TSig}}$ for each $\mathsf{vk}_c^{\mathsf{ex}}$ where $c \in [r]$ and $r$ is the number of burner addresses in $\mathsf{In}_a$.

   Finally, all servers will compute $\sigma_{\mathsf{vk}}(done)$ using $\mathcal{F}_{\mathsf{TSig}}$. If a server $\mathcal{P}_i$ obtained an abort from any $\mathcal{F}_{\mathsf{TSig}}$ then it sends (POST, *sid*, *abort*, $\widehat{\mathsf{vk}}_i, \sigma_{\widehat{\mathsf{vk}}_i}(u_j)$) to $\mathcal{F}_{\mathsf{SC}}$, updates $\mathcal{F}_{\mathsf{Clock}}$, sends (ABORT, *sid*) to all $\mathcal{P}$ and enters **Abort**.
3. If no $\mathcal{F}_{\mathsf{TSig}}$ aborted then send $\mathsf{tx}_a$ to each $\mathcal{C}_j$ with $\mathcal{L}_a \in \{\mathsf{vk}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{trg}}\}$.
4. All $\mathcal{P}_i$ send (POST, *sid*, *done*, $\sigma_{\mathsf{vk}}(done), \mathsf{vk}$) to $\mathcal{F}_{\mathsf{SC}}$.

**Abort:** If $\mathcal{P}_i$ sends or receives (ABORT, *sid*) at any point:

1. If $\mathcal{P}_i$ received (ABORT, *sid*) from another server then update $\mathcal{F}_{\mathsf{Clock}}$.
2. If $\mathcal{F}_{\mathsf{SC}}$ is in state `abort` and $\mathcal{P}_i$ has the message $\sigma_{\mathsf{vk}}(done)$ then it sends (POST, *sid*, *done*, $\sigma_{\mathsf{vk}}(done), \mathsf{vk}$) to $\mathcal{F}_{\mathsf{SC}}$. Else if $\mathcal{P}_i$ has the message $\sigma_{\mathsf{vk}}(ok)$ then it sends (POST, *sid*, *ok*, $\sigma_{\mathsf{vk}}(ok), \mathsf{vk}$) to $\mathcal{F}_{\mathsf{SC}}$. Afterwards it updates $\mathcal{F}_{\mathsf{Clock}}$.
3. If it obtains $\mathsf{coins}(d)$ from $\mathcal{F}_{\mathsf{SC}}$ then it outputs $\mathsf{coins}(d)$ and terminates.
4. If $\mathcal{F}_{\mathsf{SC}}$ is in state `reimburse1` then run **Abort without Output**. If $\mathcal{F}_{\mathsf{SC}}$ is in state `ok1` then run **Abort with Output**.

Fig. 12: The protocol $\Pi_{\mathsf{Ex}}$.

**Protocol $\Pi_{\mathsf{Ex}}$** (Part 3)

**Abort with Output:**

1. For each registered $\mathcal{C}_j$ the servers send $(\text{POST}, sid, t_j, \sigma_{\mathsf{vk}}(t_j), \mathsf{vk})$ and $(\text{POST}, sid, u_j, \sigma_{\mathsf{vk}}(u_j), \mathsf{vk})$ to $\mathcal{F}_{\mathsf{SC}}$ where $t_j = (\mathcal{C}_j, \mathcal{L}_j^{\mathsf{src}}, \mathsf{am}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{ex}}, \mathcal{L}_j^{\mathsf{trg}}, \mathsf{vk}_j^{\mathsf{trg}})$ and $u_j = (\mathcal{C}_j, \overline{\mathtt{id}}_j)$.
2. Each $\mathcal{P}_i$ runs **Reveal** and **Allow Verify** for each $\mathcal{F}_{\mathsf{Ident}}^{a,b}$, followed by sending $(\text{POST}, sid, \mathbf{s}_i^{a,b}, \sigma_{\widehat{\mathsf{vk}}_i}(\mathbf{s}_i^{a,b}), \widehat{\mathsf{vk}}_i)$ to $\mathcal{F}_{\mathsf{SC}}$ and updates $\mathcal{F}_{\mathsf{Clock}}$. If $\mathcal{P}_i$ or $\mathcal{C}_j$ obtains $\mathsf{coins}(d)$ from $\mathcal{F}_{\mathsf{SC}}$ then it outputs $\mathsf{coins}(d)$ and terminates.
3. Otherwise $\mathcal{P}_i$ recovers all $\mathbf{y}^{a,b}$ from $\mathcal{F}_{\mathsf{SC}}$ and does the following for each $\mathcal{L}_a$:
   (a) Determine $\mathtt{id}_a, \mathsf{In}_a, \mathsf{Out}_a$ using $\mathtt{makeTX}$ from all $\mathbf{y}^{a,b}$ as well as all $t_j, u_j$.
   (b) Compute $\rho_i^{a,c}$ by sending $(\text{SIGN}, sid, \mathtt{id}_a | \mathsf{In}_a | \mathsf{Out}_a)$ to $\mathcal{F}_{\mathsf{TSig}}$ for each $\mathsf{vk}_c^{\mathsf{ex}}$ where $c \in [r]$ and $r$ is the length of $\mathsf{In}_a$. Set $v_j = (\rho_i^{a,1}, \ldots, \rho_i^{a,r})$.
4. $\mathcal{P}_i$ sends $(\text{POST}, sid, v_j, \sigma_{\widehat{\mathsf{vk}}_i}(v_j), \widehat{\mathsf{vk}}_i)$ for $a \in [\ell]$ to $\mathcal{F}_{\mathsf{SC}}$ and updates $\mathcal{F}_{\mathsf{Clock}}$.
5. If $\mathcal{P}_i$ obtains $\mathsf{coins}(d)$ from $\mathcal{F}_{\mathsf{SC}}$ then it outputs $\mathsf{coins}(d)$ and terminates.
6. Each $\mathcal{C}_j$ checks if $\mathcal{F}_{\mathsf{SC}}$ contains data for a transaction $\mathsf{tx}_a$ if $\mathcal{L}_a \in \{\mathsf{vk}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{trg}}\}$. If so then $\mathcal{C}_j$ reconstructs $\mathsf{tx}_a$. Otherwise it outputs the $\mathsf{coins}$ obtained from $\mathcal{F}_{\mathsf{SC}}$.

**Abort without Output:**

1. For each registered client $\mathcal{C}_j$ the servers send $(\text{POST}, sid, t_j, \sigma_{\mathsf{vk}}(t_j), \mathsf{vk})$ to $\mathcal{F}_{\mathsf{SC}}$ where $t_j = (\mathcal{C}_j, \mathcal{L}_j^{\mathsf{src}}, \mathsf{am}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{ex}}, \mathcal{L}_j^{\mathsf{trg}}, \mathsf{vk}_j^{\mathsf{trg}})$. Furthermore, each client $\mathcal{C}_j$ sends $(\text{POST}, sid, \overline{\mathtt{id}}_j | \overline{\mathsf{In}}_j | \overline{\mathsf{Out}}_j, \sigma_{\mathsf{vk}_j^{\mathsf{src}}}(\overline{\mathtt{id}}_j | \overline{\mathsf{In}}_j | \overline{\mathsf{Out}}_j), \mathsf{vk}_j^{\mathsf{src}})$ to $\mathcal{F}_{\mathsf{SC}}$ where $\overline{\mathtt{id}}_j | \overline{\mathsf{In}}_j | \overline{\mathsf{Out}}_j$ are from the transaction $\overline{\mathsf{tx}}_j$ that $\mathcal{C}_j$ made to $\mathsf{vk}_j^{\mathsf{ex}}$ and where $\sigma_{\mathsf{vk}_j^{\mathsf{src}}}(\overline{\mathtt{id}}_j | \overline{\mathsf{In}}_j | \overline{\mathsf{Out}}_j)$ is from the signatures $\overline{\mathsf{Sig}}_j$ that are part of $\overline{\mathsf{tx}}_j$. Then each $\mathcal{P}_i$ updates $\mathcal{F}_{\mathsf{Clock}}$.
2. Next, each $\mathcal{P}_i$ for each $\mathcal{C}_j$ computes the transaction $\mathsf{tx}_j$ as follows:
   (a) Set $\mathsf{In}_j = (\mathtt{id}_j, \mathsf{am}_j^{\mathsf{src}})$ and $\mathsf{Out}_j = (\mathsf{am}_j^{\mathsf{src}}, \mathsf{vk}_j^{\mathsf{src}})$.
   (b) Determine $\mathtt{id}_j$ as the hash of $\mathsf{In}_j, \mathsf{Out}_j$. Set $t_j = (\mathtt{id}_j, \mathsf{In}_j, \mathsf{Out}_j)$.
   (c) Compute $\rho_j^i$ by sending $(\text{SIGN}, sid, t_j, \mathsf{vk}_j^{\mathsf{ex}})$ to $\mathcal{F}_{\mathsf{TSig}}$.
3. Each $\mathcal{P}_i$ sends $(\text{POST}, sid, \rho_j^i, \sigma_{\widehat{\mathsf{vk}}_i}(\rho_j^i), \widehat{\mathsf{vk}}_i)$ to $\mathcal{F}_{\mathsf{SC}}$ and updates $\mathcal{F}_{\mathsf{Clock}}$.
4. If $\mathcal{P}_i$ obtains $\mathsf{coins}(d)$ from $\mathcal{F}_{\mathsf{SC}}$ then it outputs $\mathsf{coins}(d)$ and terminates.
5. Each $\mathcal{C}_j$ checks if $\mathcal{F}_{\mathsf{SC}}$ contains data for a transaction $\mathsf{tx}_j$. If so then $\mathcal{C}_j$ reconstructs $\mathsf{tx}_j$. Otherwise it outputs the coins $\mathsf{coins}$ it obtained from $\mathcal{F}_{\mathsf{SC}}$.

Fig. 13: The protocol $\Pi_{\mathsf{Ex}}$.

## Appendix C The Exchange Protocol, continued

We present the full protocol $\Pi_{\mathsf{Ex}}$ in Figures 11-13 and the algorithm `compSwap` in Figure 14, showing a sketch for the proof of Theorem 1.

### C.1 Proof of Theorem 1

*Proof.* In order to prove the claim we construct a PPT simulator $\mathcal{S}$ which will, in the ideal setting, interact with $\mathcal{A}$, $\mathcal{F}_{\mathsf{EX}}$ and all the hybrid and global functionalities in such a way that $\mathcal{F}_{\mathsf{EX}} \circ \mathcal{S} \approx \Pi_{\mathsf{Ex}} \circ \mathcal{A}$ for any PPT environment $\mathcal{Z}$, i.e. that the interaction created by $\mathcal{S}$ is indistinguishable from a protocol transcript in a composed setting. Additionally, the global functionalities will be present in both cases and $\mathcal{Z}$ will be able to perform queries to these.

$\mathcal{S}$, on a high level, runs as follows:

- Upon learning the sets $I, J$ of corrupted servers and clients $\mathcal{S}$ simulates honest servers $\mathcal{P}_i$ and honest clients $\mathcal{C}_j$ in a simulated instance of $\Pi_{\mathsf{Ex}}$.
- $\mathcal{S}$ follows the protocol $\Pi_{\mathsf{Ex}}$ but with dummy inputs for $\mathcal{P}_i$ and $\mathcal{C}_j$. $\mathcal{S}$ will observe $\mathcal{A}$'s behaviour during the protocol execution and from it extract inputs that it sends to $\mathcal{F}_{\mathsf{EX}}$ on behalf of the dishonest servers and clients.
- During **Initialize** $\mathcal{S}$ will forward all messages sent by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{TSig}}$ for generating the key $\mathsf{vk}$. It generates keys $\widehat{\mathsf{vk}}_i$ for all the simulated servers and sends these to $\mathcal{A}$. It then for each party sends a signature on $t$ as in the protocol, where different values are signed by different honest parties if $\mathcal{A}$ sent different keys to different honest servers. $\mathcal{S}$ will additionally provide messages $\mathsf{coins}(d)$ by each simulated honest party to $\mathcal{F}_{\mathsf{SC}}$. If $\mathcal{F}_{\mathsf{EX}}$ activates the clock and time progresses then it sends $\mathsf{coins}(|I| \cdot d)$ to $\mathcal{F}_{\mathsf{EX}}$, otherwise it aborts.
- During **Enroll Client** $\mathcal{S}$ will simulate sign-up of honest clients based on the output of $\mathcal{F}_{\mathsf{EX}}$ by sending the respective message to $\mathcal{A}$. It forwards all interactions of $\mathcal{A}$ to $\mathcal{F}_{\mathsf{TSig}}$ concerning the burner address. Ultimately it creates "fake" inputs for each honest $\mathcal{C}_j$ to the respective $\mathcal{F}_{\mathsf{Ident}}$ instance. For all the dishonest clients $\mathcal{S}$ observes $\mathcal{F}_{\mathsf{SC}}$ as well as the instances of $\mathcal{F}_{\mathsf{BB}}$. Upon receiving an enrollment message to all simulated honest servers it sends the respective message to $\mathcal{F}_{\mathsf{EX}}$. Their inputs will be extracted from the respective instances of $\mathcal{F}_{\mathsf{Ident}}$ which can be observed by $\mathcal{S}$. Alternatively send a message to $\mathcal{F}_{\mathsf{EX}}$ if a dishonest client neither provided any input nor made a transfer.
- During **Exchange** $\mathcal{S}$ follows what the honest servers would do in the protocol and forwards messages to $\mathcal{F}_{\mathsf{TSig}}$ accordingly. It aborts if the computation on any $\mathcal{F}_{\mathsf{Ident}}$ fails or if the sharing fails. Finally, it sends the respective message to $\mathcal{F}_{\mathsf{EX}}$.
- During **Open** either start the abort if $\mathcal{F}_{\mathsf{EX}}$ does so or obtain the outputs $\mathbf{y}^{a,b}$ from $\mathcal{F}_{\mathsf{EX}}$. In case that $\mathcal{S}$ obtains the output then make the output from $\mathcal{F}_{\mathsf{Ident}}$ to $\mathcal{A}$ appear to be the correct corresponding output and run their output phases[8]. Then forward all interactions towards $\mathcal{F}_{\mathsf{TSig}}$ of $\mathcal{A}$. If a

[8] This is possible, even if $\mathcal{F}_{\mathsf{Ident}}$ is global, as $\mathcal{S}$ can alter all messages between $\mathcal{A}$ and global functionalities. This will not be noticeable for $\mathcal{Z}$ as $\mathcal{F}_{\mathsf{Ident}}$ only outputs information for a specific *sid* to TMs acting in that session.

"done" message gets signed then let each honest server send it to $\mathcal{F}_{\mathsf{SC}}$. Upon obtaining $\mathsf{coins}(|I| \cdot d)$ back from $\mathcal{F}_{\mathsf{EX}}$ let $\mathcal{F}_{\mathsf{SC}}$ forward these to $\mathcal{A}$.
- If $\mathcal{F}_{\mathsf{EX}}$ runs **Abort without Outputs** $\mathcal{S}$ simulates the behavior of the honest servers by sending the signed messages that they learned during **Enroll Client** to $\mathcal{F}_{\mathsf{SC}}$. Furthermore it fetches the transaction $\mathsf{tx}$ from $\mathcal{F}_{\mathsf{BB}}$ that the honest client made to send money to the burner address and puts it on $\mathcal{F}_{\mathsf{SC}}$ as an honest client does in the protocol. If $\mathcal{A}$ does not send certain messages to $\mathcal{F}_{\mathsf{SC}}$ then identify the respective sets $J_1, I_1$ and send them to $\mathcal{F}_{\mathsf{EX}}$. If $\mathcal{F}_{\mathsf{EX}}$ sends any $\mathsf{coins}(d)$ back to $\mathcal{S}$ then let $\mathcal{F}_{\mathsf{SC}}$ distribute these accordingly.
- Simulate **Abort with Output** like **Output without Abort** is simulated.

It is easy to see that the messages which an adversary reads during the simulated protocol are consistent with the values that are returned both to the honest parties and servers and with the outputs of the ideal functionality. The state of $\mathcal{F}_{\mathsf{Clock}}$ during each protocol step is identical with that during the simulated protocol instance. Moreover, whenever the simulated protocol aborts then this also leads to an abort of $\mathcal{F}_{\mathsf{EX}}$ and when honest servers let $\mathcal{F}_{\mathsf{EX}}$ abort then this is reflected in $\mathcal{F}_{\mathsf{SC}}$. Honest clients never abort in the simulation as they cannot abort in $\Pi_{\mathsf{Ex}}$ either. Finally, the coins that $\mathcal{A}$ puts into $\mathcal{F}_{\mathsf{SC}}$ and obtains from $\mathcal{F}_{\mathsf{SC}}$ are identical with those that $\mathcal{S}$ inputs into $\mathcal{F}_{\mathsf{EX}}$ or obtains from it. □

## Appendix D The Private Matching Algorithm

For our simple proof-of-concept of `compSwap`, we only assume an exchange between two distinct tokens and only allow exchanges of a preset amount in a pivot tokens. However, since we build on general purpose MPC for matching, other algorithms and rules could be implemented when implementing `compSwap`.

Concretely our prototype implementation could for example perform exchanges between Bitcoin and Ethereum and each exchange would be of, say of 1 BTC. Hence a buy order would only contain the *upper limit* of Ether one would maximally be willing to pay for 1 BTC, which could for example be 30 ETH. Similarly a sell order would contain the *lower limit* of Ether one would minimally be willing to accept in exchange for 1 BTC. To get higher granularity the units can of course be changed so that one rather express how many 0.00001 ETH one will pay for 1 BTC, for example $2,985,184 \cdot 10^{-4}$ ETH. To buy more than 1 BTC, one could simply issue multiple orders.

Orders are matched such that the *largest* buy limit is matched with the *smallest* sell limit, under the constraint that buy limit is larger than, or equal to, the sell limit. This match is removed from the set of orders, and the new remaining *largest* buy limit is matched with the new remaining *smallest* sell order. This continues until all orders are fulfilled, or the remaining largest buy order is smaller than the smallest sell order, when the remaining orders are discarded.

The clearing price of an order is then defined to be the average of the buy limit of this specific order's buyer and the sell limit of this specific order's seller. This means that all matched orders will likely have different clearing prices. To

implement this securely we require that the buy/sell limits of every client's order is hidden and only the clearing price is leaked, and *only after* all orders in the set have matched or discarded.

Since the buy/sell limits are hidden, orders must be distinguished, thus each order is assigned a distinct, yet public ID, that is then linked to their client's identity outside of the matching protocol, $\mathsf{vk}_j^{\mathsf{src}}$. However, this must then be oblivious to the servers *during* execution as this would otherwise leak information about the relative difference between the different client's limits during execution, thus giving the servers a potential advantage in front-running the client's in other exchanges. But it could simply just be the client's public key $\mathsf{vk}_j^{\mathsf{src}}$, however, for efficiency reason it makes sense to map this to a small index such that it can be represented by a single element in the MPC computation.

With this description in mind we see that the matching protocol is simply to sort, based on the buy limits and the sell limits, in parallel, followed by an iteration through the sorted buy/sell limits, matching as long as the $i$'th buy limit is greater than the $i$'th sell limit. However, even doing this in a secure way becomes non-trivial since we cannot allow branching in MPC without requiring *all* branches being executed. Hence it will not be efficient to use a standard approach to sorting, such as quick-sort. Even merge-sort will be highly inefficient since a trivial approach to oblivious merging would require $O(m^2)$ time for $m$ elements. For this reason an oblivious sorting algorithm (sorting network) must be used. For example Batcher's Odd-Even Merge sort [44, Sec. 5.4.3], which we used in our implementation. This algorithm requires $O(m \log^2(m))$ comparisons and has depth $O(\log^2(m))$ for $m$ elements.

We describe the secure computation logic of our matching algorithm in Fig. 14, using the common notation of $[\![\cdot]\!]$ to indicate values hidden in MPC. We include a subscript, $[\![\cdot]\!]_i$, to indicate a specific index of the internal representation hiding the value when this would otherwise not be known to the server. In particular, this is the case after obliviously sorting the the input elements.

We note that in this protocol we assume that clients can send secret input to the servers in a consistent way. We will not dwell on how exactly this is done, but simply mention that this can be done efficient, black-box in the underlying MPC scheme (assuming that it allows for reactive computation) using an approach called *out-sourced MPC* [41].

In our algorithm we assume access to an algorithm, `Sort`, for oblivious sorting of a list in *increasing* order and an algorithm `Rev` for reversing the order of a list. Thus $\texttt{Rev}(\texttt{Sort}(\cdot))$ returns a list in *decreasing* order. We also assume that we have an oblivious method for comparison, that is $[\![z]\!] := [\![x]\!] \geq^? [\![y]\!]$ where $z = 1$ if $x \geq y$ and $z = 0$ otherwise. Finally we abuse notation and assume the permutations $\text{sort}^{\mathsf{buy}}(j)$, $\text{sort}^{\mathsf{sell}}(j)$ map an index $j$ in an unsorted list to its sorted position in said list, for the buy, respectively sell orders.

We note that in a real-world deployment, more logic would probably be added to the matching. In particular, allowing for exchanges of variable amounts and include time-stamps such that the oldest orders get priority in case of multiple possible matches.

**Algorithm** $C_{\texttt{compSwap}}$

$C_{\texttt{compSwap}}$ realizes a limited `compSwap` functionality for order matching of $\mathsf{const}$ tokens from $\mathcal{L}_a$ to a number of tokens on $\mathcal{L}_b$. The algorithm is run between the servers $\mathcal{P}_1, \ldots, \mathcal{P}_n$ on $m = m_{\mathsf{buy}} + m_{\mathsf{sell}}$ order inputs, with order $m_j$ from client $\mathcal{C}_j$, and outputs at most $\mathsf{min}(m_{\mathsf{buy}}, m_{\mathsf{sell}})$ order matches.

**Input:** Upon input $(\mathcal{C}_j, \delta_j, \mathsf{am}_j^{\mathsf{src}}, \mathsf{const})$ from client $\mathcal{C}_j$ for $j \in [m]$ with $\delta_j = 0$ if $\mathcal{C}_j$ wishes to *buy* $\mathsf{const}$ tokens on $\mathcal{L}_a$ using *at most* $\mathsf{am}_j^{\mathsf{src}}$ tokens on $\mathcal{L}_b$ and $\delta_j = 1$ if $\mathcal{C}_j$ wishes to sell $\mathsf{const}$ tokens from $\mathcal{L}_a$ for *at least* $\mathsf{am}_j^{\mathsf{src}}$ on $\mathcal{L}_b$. Concretely client $\mathcal{C}_j$ inputs $(\textsc{Input}, sid, j, \mathsf{am}_j^{\mathsf{src}})$ on $\mathcal{F}_{\mathsf{Ident}}^{a,b}$ and sends $\delta_j$ in plain to the servers. Furthermore, every server $\mathcal{P}_i$ for $i \in [n]$ inputs $(\textsc{Input}, sid, j, \cdot)$ on $\mathcal{F}_{\mathsf{Ident}}^{a,b}$. If $\delta_j = 0$ the servers map the inputs to $(\llbracket \mathsf{am}_j^{\mathsf{src}} \rrbracket_i, \llbracket j \rrbracket_i, \mathsf{buy})$ for some unused $i \in [m_{\mathsf{buy}}]$, otherwise they map it to $(\llbracket \mathsf{am}_j^{\mathsf{src}} \rrbracket_i, \llbracket j \rrbracket_i, \mathsf{sell})$ for some unused $i \in [m_{\mathsf{sell}}]$. The servers also input a special value indicating null by calling $(\textsc{Input}, sid, \texttt{null})$ on $\mathcal{F}_{\mathsf{Ident}}^{a,b}$ to get $\llbracket \texttt{null} \rrbracket$.

**Evaluate:** The servers input $(\textsc{Compute}, sid)$ on $\mathcal{F}_{\mathsf{Ident}}^{a,b}$ for computing the following:

1. Compute

$$\begin{pmatrix} \llbracket \mathsf{am}^{\mathsf{src}}_{\mathrm{sort}^{\mathsf{buy}}(j)} \rrbracket_i, \\ \llbracket \mathrm{sort}^{\mathsf{buy}}(j) \rrbracket_i, \mathsf{buy} \end{pmatrix}_{i \in [m_{\mathsf{buy}}]} \leftarrow \texttt{Rev}\left(\texttt{Sort}\left(\left(\llbracket \mathsf{am}_j^{\mathsf{src}} \rrbracket_i, \llbracket j \rrbracket_i, \mathsf{buy}\right)_{i \in [m_{\mathsf{buy}}]}\right)\right),$$

$$\begin{pmatrix} \llbracket \mathsf{am}^{\mathsf{src}}_{\mathrm{sort}^{\mathsf{sell}}(j)} \rrbracket_i, \\ \llbracket \mathrm{sort}^{\mathsf{sell}}(j) \rrbracket_i, \mathsf{sell} \end{pmatrix}_{i \in [m_{\mathsf{sell}}]} \leftarrow \texttt{Sort}\left(\left(\llbracket \mathsf{am}_j^{\mathsf{src}} \rrbracket_i, \llbracket j \rrbracket_i, \mathsf{sell}\right)_{i \in [m_{\mathsf{sell}}]}\right)$$

2. For $i \in [\mathsf{min}(m_{\mathsf{buy}}, m_{\mathsf{sell}})]$ let

$$\llbracket c_i \rrbracket := \left\llbracket \mathsf{am}^{\mathsf{src}}_{\mathrm{sort}^{\mathsf{buy}}(j)} \right\rrbracket_i \geq^? \left\llbracket \mathsf{am}^{\mathsf{src}}_{\mathrm{sort}^{\mathsf{sell}}(j)} \right\rrbracket_i$$

and compute

$$\llbracket x_i \rrbracket = \left\llbracket \mathrm{sort}^{\mathsf{buy}}(j) \right\rrbracket_i \cdot \llbracket c_i \rrbracket + (1 - \llbracket c_i \rrbracket) \cdot \llbracket \texttt{null} \rrbracket$$

$$\llbracket y_i \rrbracket = \left\llbracket \mathrm{sort}^{\mathsf{sell}}(j) \right\rrbracket_i \cdot \llbracket c_i \rrbracket + (1 - \llbracket c_i \rrbracket) \cdot \llbracket \texttt{null} \rrbracket$$

$$\llbracket p_i \rrbracket = \frac{\left(\left\llbracket \mathsf{am}^{\mathsf{src}}_{\mathrm{sort}^{\mathsf{buy}}(j)} \right\rrbracket_i + \left\llbracket \mathsf{am}^{\mathsf{trg}}_{\mathrm{sort}^{\mathsf{sell}}(j)} \right\rrbracket_i\right) \cdot \llbracket c_i \rrbracket}{2} + (1 - \llbracket c_i \rrbracket) \cdot \llbracket \texttt{null} \rrbracket$$

3. Open the vector $\mathbf{y}^{a,b\prime} = (\llbracket x_i \rrbracket, \llbracket y_i \rrbracket, \llbracket p_i \rrbracket)_{i \in [\mathsf{min}(m_{\mathsf{buy}}, m_{\mathsf{sell}})]}$.
4. Return the vector $\mathbf{y}^{a,b}$ which is $\mathbf{y}^{a,b\prime}$ after removing all entries equal to $(\texttt{null}, \texttt{null}, \texttt{null})$ and mapping each entry $(x_i, y_i, p_i)$ to $(x_i, p_i, y_i, 1)$.

Fig. 14: The Private Matching Algorithm