Preserving Buyer-Privacy in Decentralized Supply Chain Marketplaces

Varun Madathil*, Alessandra Scafuro, Kemafor Anyanwu, Sen Qiao, Akash Pateria, and Binil Starly

Department of Computer Science, North Carolina State University
{vrmadath, ascafur, kogan, sqiao, apateri, bstarly}@ncsu.edu

Abstract. Technology is being used increasingly for lowering the trust barrier in domains where collaboration and cooperation are necessary, but reliability and efficiency are critical due to high stakes. An example is an industrial marketplace where many suppliers must participate in production while ensuring reliable outcomes; hence, partnerships must be pursued with care. Online marketplaces like Xometry facilitate partnership formation by vetting suppliers and mediating the marketplace. However, such an approach requires that all trust be vested in the middleman. This centralizes control, making the system vulnerable to being biased towards specific providers. The use of blockchains is now being explored to bridge the trust gap needed to support decentralizing marketplaces, allowing suppliers and customers to interact more directly by using the information on the blockchain. A typical scenario is the need to preserve privacy in certain interactions initiated by the buyer (e.g., protecting a buyer's intellectual property during outsourcing negotiations). In this work, we initiate the formal study of matching between suppliers and buyers when buyer-privacy is required for some marketplace interactions and make the following contributions. First, we devise a formal security definition for private interactive matching in the Universally Composable (UC) Model that captures the privacy and correctness properties expected in specific supply chain marketplace interactions. Second, we provide a lean protocol based on any programmable blockchain, anonymous group signatures, and public-key encryption. Finally, we implement the protocol by instantiating some of the blockchain logic by extending the BigChainDB blockchain platform.

1 Introduction

Online marketplaces like Xometry [1] provide a centralized venue for vetted suppliers and customers that significantly facilitate matching customers' needs and suppliers' offers in the manufacturing domain. On the downside, such an approach requires that all trust be vested in the middleman. This approach centralizes control, making the system vulnerable to bias towards specific providers. Furthermore, both customers and suppliers have no privacy w.r.t. the middleman.

* Varun Madathil and Alessandra Scafuro are supported by NSF Award #1764025

[1] Xometry https://www.xometry.com/ is one among many other (e.g., Fictiv, Protolab) online portals for on-demand manufacturing services that match their vetted suppliers with customers interested in 3D printing their unique designs.

Motivated by these concerns and spurred by the development of blockchain technology, recent work [14, 16, 29] propose to build *decentralized* online marketplace by replacing the middleman with a smart-contract capable blockchain. A blockchain [25, 38] is an immutable ledger that is shared among multiple *peers*. Under the assumption that the majority of the peers follow the protocol, the ledger is guaranteed to be immutable and contain only valid transactions. Validity of a transaction is assessed by the peers by running specific scripts on those transactions. At a high-level, to build a decentralized marketplace, the interaction between customers and suppliers with the middleman could be replaced with smart contracts over blockchain transactions. Correctness would be guaranteed by the transparency and consistency properties of the blockchain, which enforce trust and facilitate dispute resolution.

Existing proposal for decentralized marketplaces [14, 16, 31, 33] mostly target retail marketplaces (e.g., Amazon), where the matching between a customer C and a supplier S can be determined *non-interactively* via a payment transaction from C in favor of S for a certain item. In this work, we are interested in marketplaces where a match between a customer and a supplier is determined after multiple interactions (e.g., request for proposal, bidding, selection, etc.), and some interactions involve *private inputs* from both customers and suppliers. This is typical of outsourcing supply chain marketplaces where some interactions involve customers needing to disclose high-value data e.g. intellectual-property assets like manufacturing designs, software algorithms etc. The process usually involves an initial exploratory phase in which only limited information is shared with a large group of suppliers, followed by a narrowing down of the selection of candidate suppliers with whom subsequent interactions involving additional data that need to be kept private. As a concrete example, a customer might request proposals for the fabrication of a patient-specific craniofacial implant made out of medical-grade titanium alloy, with a 3-week deadline. This initial information may allow suppliers to determine if the request falls within their service capabilities, but yet it does not necessarily divulge high-value information. However, as negotiations proceed and potential suppliers are selected, suppliers will only be able to determine if they can meet the 3-weeks delivery time and what price to charge *after seeing* the complexity of the private implant design. Thus the implant design is shared only with the shortlisted suppliers. In addition, buyers need to keep some of their inputs in interactions private, but they may also want to keep their identities private for some interactions. This is because, in some contexts, the partnerships and collaborations that a company engages in are considered a part of its competitive advantage.

On the other hand, in supply chain marketplaces, suppliers want to share as much information about their capabilities, to be matched as candidates with as many requests as possible. However, they may want to keep their bid values for each request private. Therefore, in the context of blockchains where all transactions and transacting parties are recorded, it is essential to consider how to keep information about buyer identity and some transactional inputs of both buyer and supplier private.

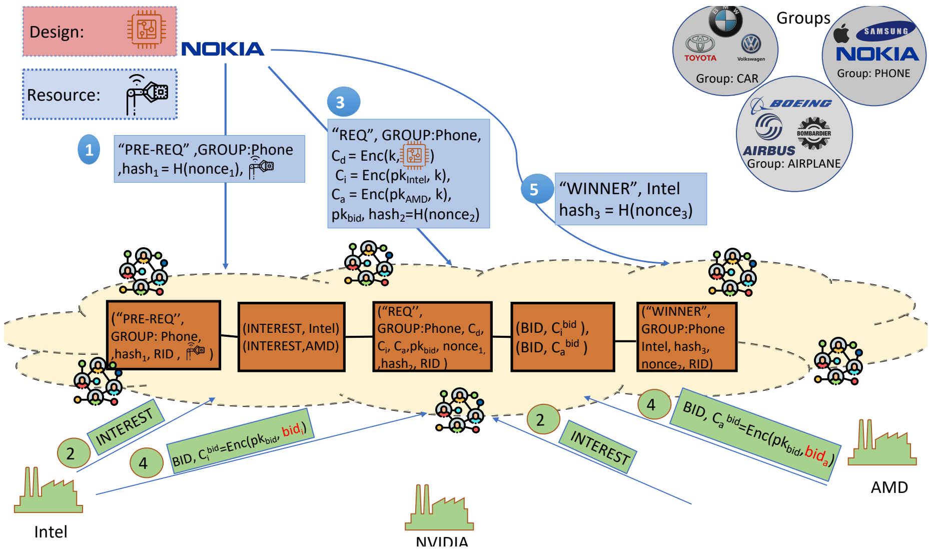
To summarize, in this work, we target interactive marketplaces that present the following privacy properties. 1. *Matching is determined from private inputs.* Private inputs from both the customers (e.g., the private product design) and the suppliers (e.g., the quotes) are required to perform the matching. Hence, the approach of simply publishing requests on a blockchain and having smart contracts matching them is not applicable here. *2. Customers should be anonymous but accountable.* The matching between the customer and the supplier should remain private. Yet, suppliers need some guarantee that they are interacting with accountable customers, i.e., belonging to a *group of verified customers*. At the same time, suppliers would also want to build a reputation by having a record of successful matches with accountable customers. This is different from the typical marketplace setting where a customer can be completely anonymous, and reputation is built only through reviews. *3. Matched resources might be exclusive.* A supplier sells the *use* of its resources rather than an item. The marketplace must guarantee that the manufacturer does not overbook its resources.

1.1 Our contribution

We initiate the study of decentralized interactive marketplaces and we build a proof-of-concept system based on blockchain technology. Specifically, our contributions are:

1. **A Formal Definition of Private Interactive Matching**. We formally capture the correctness and privacy properties of an interactive marketplace, by abstracting it as the problem of private interactive matching in the Universally Composable (UC) framework [7]. Our definitional choices are inspired by the service-oriented marketplaces such as in the manufacturing domain.
2. **A Protocol for Decentralized Private Interactive Matching.** We provide a decentralized protocol based on an ideal ledger capable of a set of validation rules we define, and on anonymous group signatures. We formally prove it is UC-secure.
3. **Implementation and Evaluations.** We provide an implementation strategy for our ledger protocol that involves extending the transaction validation framework of an open-source blockchain database BigChainDB (discussed in Section 4). We call the extended platform SmartChainDB.

A Formal Definition of Private Interactive Matching. To formally model the intuitive security guarantees outlined above we use the Universally Composable (UC) framework [7] to define an ideal functionality $\mathcal{F}_{\mathsf{PrivateMatch}}$. The ideal functionality $\mathcal{F}_{\mathsf{PrivateMatch}}$ describes the ideal behavior of a platform that matches customer with the correct suppliers, while guaranteeing anonymity of the customer (within a certain group of well-known customers), correctness of the match, privacy and fairness. We describe the ideal functionality in details in Section 2. At high-level the ideal functionality $\mathcal{F}_{\mathsf{PrivateMatch}}$ has the following properties. *Generality:* It captures a variety of settings since there are no fixed roles – a party can sign up as a supplier and customer; and no fixed logic – the ideal functionality is parameterized by external algorithms $\mathsf{validResource}$ and $\mathsf{canServe}$ that determines validity of the supplier commands. *Customer's (Accountable) Anonymity:* Requests are not associated to a specific customer, but to the group the customer belong to. This means that when a supplier is matched with a customer, the only information leaked to the other parties is that a supplier was matched with a member of a certain group (e.g., the group containing all the implant manufacturing companies). But a misbehaving customer can still be identified within a group and then punished. *Customer's Input Privacy:* Requests contain public values (e.g., the type of resources required, the deadlines, etc), and private values (e.g., the product design). From our example earlier, the suppliers were informed that they were to provide titanium alloy for three weeks. We consider such resources to be public as is the case in the real world. The private values will be revealed only to the suppliers who have *expressed the* *interest* in fulfilling the request and *possess* the resources to do so. Our ideal functionality allows a supplier to signal interest to all requests just to see the private inputs. Note that this models a behavior that is allowed in real world. However, note that just as in the real world, measures can be added so that if a supplier exhibits this behavior, it can be automatically discarded by the customer. *Supplier's Input Privacy:* The resources offered by a supplier and their interest in serving a request are public. However, details of their quote (e.g., price) are private for everyone, except, of course, for the customer. *Supplier's Transparency:* The resources utilization (e.g., allocation to a certain request) of the suppliers is public. *Correctness and Flexibility of the Match:* Only *capable* suppliers can bid to be matched with the customer. The winner is chosen by the customer according to its own private decision algorithm.



Design:
Resource:
NOKIA
Groups
TOYOTA
Volkswagen
Group: CAR
SAMSUNG
NOKIA
Group: PHONE
BOEING
BOMBARDIER
AIRBUS
Group: AIRPLANE
1
"PRE-REQ" ,GROUP:Phone ,hash$_1$ = H(nonce$_1$),
3
"REQ", GROUP:Phone, C$_d$ = Enc(k,) C$_i$ = Enc(pk$_{Intel}$, k), C$_a$ = Enc(pk$_{AMD}$, k), pk$_{bid}$, hash$_2$=H(nonce$_2$)
5
"WINNER", Intel hash$_3$ = H(nonce$_3$)
("PRE-REQ", GROUP: Phone, ,hash$_1$, RID ,)
(INTEREST, Intel) (INTEREST,AMD)
("REQ", GROUP:Phone, C$_d$, C$_i$, C$_a$,pk$_{bid}$, nonce$_1$, ,hash$_2$, RID)
(BID, C$_i^{bid}$), (BID, C$_a^{bid}$)
("WINNER", GROUP:Phone Intel, hash$_3$, nonce$_2$, RID)
2
INTEREST
4
BID, C$_i^{bid}$=Enc(pk$_{bid}$, bid$_i$)
2
INTEREST
4
BID, C$_a^{bid}$=Enc(pk$_{bid}$, bid$_a$)
Intel
NVIDIA
AMD

Fig. 1. Example of the protocol overview.

A Protocol for Decentralized Private Interactive Matching. We provide a protocol that securely instantiates the ideal functionality $\mathcal{F}_{\mathsf{PrivateMatch}}$. In the instantiation, we require a blockchain which we abstract as an ideal ledger functionality ($\mathcal{G}_{\mathsf{smartchain}}$). To protect the anonymity of customers while ensuring accountability, we use group signatures [2,5]. These are signatures associated to a group such that a member can generate an anonymous signature on behalf of the entire group. We also assume that there is a registration phase, where the identity of each party and the claimed resources of the suppliers are vetted. This step is application-specific, and we abstract it with an ideal functionality $\mathcal{G}_{\mathsf{reg}}$. After registration, parties can join groups. Group formation is again application-specific; in our protocol we assume groups exist and do not regulate group formation. We describe the stages of the protocol in details in Section 3. Below we give an overview of the protocol with a simple example. In Figure 1 we present an example of the flow of the protocol. We consider three groups of customers as can be seen on the top right. The customers are grouped by their industry - car manufacturers, phone manufacturers and airplane manufacturers. In this example, Nokia (from the PHONE group) wants to build a chip (the design) and needs some chip building equipment (the resource). The suppliers presented below are Intel, Nvidia and AMD. All transactions are sent to a network of validators, that determine if a transaction is valid and then add them to the state of the blockchain. ❶ Nokia creates a pre-request transaction that details the resources it needs. It only authenticates that it belongs to the group (GROUP:Phone) of phone manufacturers to achieve anonymity within the group. To link next transactions, Nokia they attaches the hash of a random nonce, and reveal the nonce with the next transaction. ❷ Intel and AMD express interest in serving Nokia by posting an INTEREST transaction. ❸ Nokia creates a REQUEST transaction where it encrypts the design with a key k, and encrypts the key k with the public keys of AMD $\mathsf{pk}_{\mathsf{AMD}}$ and Intel $\mathsf{pk}_{\mathsf{Intel}}$. It also attaches a public key $\mathsf{pk}_{\mathsf{bid}}$ for Intel and AMD to encrypt their bids. ❹ AMD and Intel retrieve the design and then determine a bid value. They encrypt their respective bids under $\mathsf{pk}_{\mathsf{bid}}$ and post their BID transactions. ❺ Nokia decrypts to retrieve the bid values and determines a winner - Intel. It posts a WINNER transaction indicating that Intel won. After this step, the interaction between Nokia and Intel will happen off-chain.

For privacy, the sensitive information of the matching is protected as follows. The identity of the customer is protected with the use of group signatures. If the customer misbehaves, they may be de-anonymized by the group manager. This functionality is not easily achieved with other privacy-enhancing techniques such as ring signatures. The private design of the customer is never included directly in a transaction. It is always encrypted. The encryption could even be uploaded to another web location (controlled by the customer) and the transaction only includes the web location. In the transaction, a customer will include encryptions of the key used to encrypt the design, under the public key of the suppliers who have shown interest in doing the job. Finally, the private bids of the suppliers are protected by encrypting them under the customer's ephemeral public key pk_{bid}.

Note that, due to the use of anonymous group signatures, a malicious member of the group can send follow up transactions for the same request. To prevent this, we chain the transactions through puzzles ($\mathsf{hash} = H(\mathsf{nonce})$), in such a way that a customer can compute the next transaction in the flow only if it knows the solution (nonce) to the puzzle of previous transactions.

Implementation: SmartChainDB. Our implementation strategy choices were between the use of *smart contracts* on platforms such as Ethereum or the development of native support for these marketplace transactions as first class blockchain transactions. We chose the latter approach which offered several benefits over the use of the smart contracts model. For this reason, we selected to build on a platform BigChainDB [23], which offers an extensible architecture to implement different kinds of blockchain applications. We also implement *group signature* [5] as a possible signature scheme in BigChainDB. We refer to the resulting extended system as SmartChainDB . We undertook a performance evaluation to assess the additional overhead our changes to BigChainDB. We found that latency of our marketplace transaction types took no more than $2.5\times$ (additional 2sec of processing time) that of traditional transactions. The group signature scheme took up to $12\times$ (additional 12ms) more than the traditional signature scheme.

Some remark on our design choices. We use a blockchain to allow a seamless interaction between suppliers and customers while maintaining transparency of this interaction. This transparency is critical when disputes occur between entities. In traditional EVM compatible blockchain environments such as Ethereum, Hyperledger Fabric, *"Smart Contracts"* are used to implement general business logic. However, because smart contracts are owned by a single entity, each customer would have to bear the burden of implementing their own contract and face the risks of errors and high economic costs (gas fees) for inefficient implementation. In addition, each supplier would need to discover and study smart contracts as they are made available and make the effort to fully understand their terms since smart contracts are binding and irreversible. We observe that there would be sufficient commonality in behavior in such marketplace smart contracts that they could be generalized and provided as system level operations (i.e. first-class blockchain transactions) which can be reused and parameterized by users as needed. An additional advantage of this approach is that moving functionality away from the smart contract layer into the blockchain transaction layer, avoids the significant additional economic costs of such applications because of the high costs of smart contracts. Given the above mentioned issues, we have pursued a different implementation model that is informed by the factors that led to the success of database systems. More specifically we extend BigChainDB an open-source blockchain database with new transactions that enable matching between suppliers and customers.

Our definition of the ideal matching functionality is inspired by the service-oriented marketplaces (such as Xometry, Fictiv etc). In such domains having the supplier's activities public is considered as a feature for building reputation rather than a drawback. We allow the private input associated to the customer's request to be seen by the suppliers that are interested in bidding, and not only the supplier that is finally matched, since suppliers decide a bid value depending on the complexity of the request. We note that a supplier can try to send an interest transaction to learn the request of the customers. We note that such an interaction is always possible in interactive marketplaces and can occur even today. Furthermore, we note that depending on the application this may not be favorable to the supplier. For example, our system may easily be modified to lock resources of the supplier each time it sends an interest transaction. This will disincentivize suppliers to send interest transactions just to learn the design of customers. To protect the anonymity of the customers we use group signatures. Each group in a group signature scheme is associated with a group manager and this manager can deanonymize users. This is useful in the case of disputes and a customer needs to be de-anonymized. Furthermore, the group manager may be decentralized and we discuss this at the end of Section 3.

Related Work. Kosba et al. present Hawk [17], a framework for creating privacy-preserving Ethereum smart contracts. A set of clients describe a functionality that they want to implement, and the framework outputs the code for a smart contract, and programs that is run by a third party who is the facilitator. The data used by the smart contract is encrypted, this ensures on-chain privacy. However, the facilitator must learn the inputs of all clients in order to compute the functionality which is a scenario we avoid.

Benhamouda et al. [3] present a framework on top of the Hyperledger Fabric that allows party to send encrypted inputs to the chain. To compute a function, the parties run an off-chain multiparty computation protocol over the encrypted input. The bidding and match steps in our private match functionality share similarities with sealed-bid auctions. There, bidders simultaneously submit sealed bids to an auctioneer who then announce the winner. A few sealed-bid auctions via smart contracts have been proposed (e.g., Galal et al. [10] on Ethereum and Xiong et al. [39]). However, they cannot be extended to implement the entire flow of private matching. In terms of functionality, the closest work to our is by Thio-ac et al [35] [34]. They integrate a blockchain to an electronic procurement system (a procurement is the process of matching customers with suppliers). However, they do not consider any privacy concern, nor do they present any definitions or proofs. Recent work proposes blockchain-based solutions to decentralize e-commerce retail platforms (e.g., Amazon). In [16, 26, 29], vendors list their items as input to a smart contract and buyers input their bids. The smart contract computes the output and reveals the winner. None of these schemes consider the anonymity of the buyers. Buyers' anonymity is addressed in Beaver [32] by employing anonymous wallets and the Zcash blockchain [30]. However, this line of work is suitable only for a non-interactive match over public inputs and do not extend to the interactive setting we are interested in this paper. Finally, a rich body of work has investigated the use of blockchains to increase transparency in the supply-chain management (e.g. [9, 18, 24, 36, 37] just to name a few). However, all such work focusses only on the traceability and provenance of the products.

2 Private Interactive Matching: Formal Definition in the UC-Framework

The ideal functionality $\mathcal{F}_{\mathsf{PrivateMatch}}$ captures a private matching functionality in the UC Framework [7], where *customers* are allowed to request a service *anonymously within a group*, *suppliers* bid to fulfill these services, where the value of the bid is private and eventually a supplier is matched with the customer

The functionality maintains a global state that will contain all the transactions and can be read by all parties. It also maintains a list (buffer) of transactions that are to be added to state. The functionality keeps of track of the requests in a table $\mathcal{T}$ that is indexed by the request id (denoted RID). To set notation : $\mathcal{P}$ is the set of all parties and the adversary is denoted as $\mathcal{A}$. $\mathcal{G}$ is the list of groups initialized by the environment $\mathcal{Z}$. We denote a set of locked resources as LOCKS and TIMER as the set of times for each request. This set is used to ensure that no time-out (denoted $\mathsf{FulfillTime}$ or $\mathsf{MatchTime}$) has occurred. Upon receiving a command from a party, the functionality creates a transaction that corresponds to the command, adds the transaction to buffer and sends the same to the adversary. This reflects the fact that the adversary learns that a command has been invoked.

Overview of the functionality. Our functionality (Figure 2) captures the operations that the system should perform, the inputs that the system should protect and the information that the system is allowed to leak to an adversary. We briefly describe the security properties guaranteed by this functionality. Any party that registers with the system as a customer joins a group identified by GID. This party is **anonymous within the group**, since for every command sent by the party ($\mathtt{PRE\text{-}REQ}, \mathtt{REQ}, \mathtt{WINNER}, \mathtt{RFILL}$) its identity is not revealed, but only its GID is included. Only the set of suppliers chosen by the customer can see the design as can be observed from the $\mathtt{REQ}$ command where the design is sent only to $P_j \in \mathsf{bidderSet}$ (a set of suppliers that made bids). This guarantees **service confidentiality**. Similarly, in bid command, the adversary is only notified of the bid and the actual bid value is only revealed to P_i. This guarantees **bid confidentiality** to the bidders. **Request soundness** is the property that a customer participate for a request flow unless it had sent the `PRE-REQ` command for the request. This is achieved by checking for each command received from a party P_i for request RID, that $P_i \in \mathcal{T}[\mathsf{RID}]$. The $\mathsf{canServe}$ predicate checks if a supplier has enough available resources to serve a customer. The functionality accepts bids from bidders only if $\mathsf{canServe}$ outputs 1 on their resources. This ensures **supplier completeness**.

$\mathcal{F}_{\mathsf{PrivateMatch}}$

Register : Upon receiving $(\texttt{REG}, P_i, [\mathsf{roles}])$ from P_i, do $\mathcal{P} = \mathcal{P} \cup P_i$. Send $(\mathsf{tx} = (\texttt{REG}, P_i, [\mathsf{roles}]))$ to $\mathcal{A}$ and do $\mathsf{buffer} = \mathsf{buffer} \| \mathsf{tx}$.
Non-adaptive setup : Receive $(\texttt{CORRUPTED}, b)$ from party P_i
Join Group : Upon receiving $(\texttt{gJOIN}, \mathsf{GID})$ from a party P_i, update $\mathcal{G}[\mathsf{GID}] = \mathcal{G}[\mathsf{GID}] \cup \{P_i\}$. Send $(\texttt{gJOIN}, P_i, \mathsf{GID})$ to $\mathcal{A}$ and $(\texttt{gJOIN}, P_i, \mathsf{GID}, 1)$ to P_i.
Update Profile : Upon receiving $(\texttt{UPD}, \mathsf{prof}, \mathsf{roles})$ from P_i, verify $P_i \in \mathcal{P}$. If $\mathsf{prof} = \mathsf{GID}$, check if $P_i \in \mathcal{G}[\mathsf{GID}]$. If yes, send $(\mathsf{tx} = \texttt{UPD}, P_i, \mathsf{GID})$ to $\mathcal{A}$ and do $\mathsf{buffer} = \mathsf{buffer} \| \mathsf{tx}$. Else ignore the message. If $\mathsf{prof} = \mathsf{res}_i$, and $(P_i, \cdot) \notin \mathsf{LOCKS}[\mathsf{RID}]$ for some RID and $\mathsf{validResource}(\mathsf{res}_i, P_i) = 1$, update $\mathcal{P}$ as $\mathcal{P} \cup \{(P_i, \mathsf{res}_i)\}$ and remove other instances of $P_i \in \mathcal{P}$. Send $(\mathsf{tx} = \texttt{UPD}, P_i, \mathsf{res}_i)$ to $\mathcal{A}$ and do $\mathsf{buffer} = \mathsf{buffer} \| \mathsf{tx}$.
PreRequest : Upon receiving $(\texttt{PRE-REQ}, \mathsf{GID}, \mathsf{res}, \mathsf{RID})$ from P_i
1. Check that $P_i \in \mathcal{G}[\mathsf{GID}]$. If not, ignore.
2. Add $\mathcal{T}[\mathsf{RID}] = (P_i, \mathsf{res}, \emptyset)$.
3. Initialize $\mathsf{LOCKS}[\mathsf{RID}] = \emptyset$ and $\mathsf{TIMER}[\mathsf{RID}] = 0$.
4. Send $(\mathsf{tx} = \texttt{PRE-REQ}, (\mathsf{RID}, \mathsf{res}, \mathsf{GID}))$ to the $\mathcal{A}$ and do $\mathsf{buffer} = \mathsf{buffer} \| \mathsf{tx}$.

Interest : Upon receiving $(\texttt{INTRST}, \mathsf{RID})$ from some supplier P_j:
1. Check if $(\mathsf{res} \in \mathcal{T}[\mathsf{RID}]) \subset \mathsf{res}_j$
2. If yes, send $(\mathsf{tx} = \texttt{INTRST}, \mathsf{RID}, P_j)$ to $\mathcal{A}$ and do $\mathsf{buffer} = \mathsf{buffer} \| \mathsf{tx}$.

Request : Upon receiving $(\texttt{REQ}, (\mathsf{RID}, [\mathsf{design}_j]_{j \in \mathsf{bidders}}, \mathsf{GID}, \mathsf{bidders}))$ from P_i
1. Check $P_i \in \mathcal{T}[\mathsf{RID}]$ and $P_i \in \mathcal{G}[\mathsf{GID}]$
2. Update $\mathcal{T}[\mathsf{RID}] = (P, \mathsf{res}, \mathsf{bidders})$
3. Send $(\mathsf{tx} = \texttt{REQ}, \mathsf{RID}, \mathsf{GID}, \mathsf{bidders})$ to $\mathcal{A}$ and do $\mathsf{buffer} = \mathsf{buffer} \| \mathsf{tx}$.
4. For each $P_j \in \mathsf{bidders}$, send $(\texttt{REQ}, \mathsf{RID}, \mathsf{design}_j)$.

Bidding : Upon receiving $(\texttt{BID}, (\mathsf{RID}, \mathsf{bid}_j))$ from P_j
1. Check $\mathsf{canServe}(\mathsf{RID}, P_j, \mathsf{state}, \mathsf{LOCKS}) = 1$ If yes,
2. Send $(\mathsf{tx} = \texttt{BID}, (\mathsf{RID}, P_j))$ to $\mathcal{A}$ and $(\texttt{BID}, (\mathsf{RID}, P_j, \mathsf{bid}_j))$ to P_i. Send $\texttt{TIME}$ to $\mathcal{G}_{\mathsf{refClock}}$ to receive $\mathsf{currTime}$. Set $\mathsf{TIMER}[\mathsf{RID}] = \mathsf{currTime}$.
3. Add $(P_j, \mathsf{RID}, \mathsf{res})$ to LOCKS

Match : Upon receiving $(\texttt{WINNER}, \mathsf{RID}, \mathsf{GID}, P^*)$ from P_i
1. Check that $P_i \in \mathcal{T}[\mathsf{RID}]$ and that it belongs to $\mathcal{G}[\mathsf{GID}]$
2. For each $(P_j, \mathsf{RID}, \cdot) \in \mathsf{LOCKS}[\mathsf{RID}]$, delete $(P_j, \mathsf{RID}, \cdot)$ from $\mathsf{LOCKS}[\mathsf{RID}]$. Send $\texttt{TIME}$ to $\mathcal{G}_{\mathsf{refClock}}$ to receive $\mathsf{currTime}$. Set $\mathsf{TIMER}[\mathsf{RID}] = \mathsf{currTime}$.
3. Send $(\mathsf{tx} = \texttt{WINNER}, \mathsf{GID}, \mathsf{RID}, P^*)$ to $\mathcal{A}$ and do $\mathsf{buffer} = \mathsf{buffer} \| \mathsf{tx}$

Fulfillment from customer: Upon receiving $\mathsf{tx} = (\texttt{RFILL}, \mathsf{RID}, \mathsf{GID})$ from P_i:
1. Check that $P_i \in \mathcal{T}[\mathsf{RID}]$ and that it belongs to $\mathcal{G}[\mathsf{GID}]$
2. Send $\mathsf{tx} = (\texttt{RFILL}, \mathsf{RID}, \mathsf{GID})$ to $\mathcal{A}$ and do $\mathsf{buffer} = \mathsf{buffer} \| \mathsf{tx}$

Fulfillment from supplier: Upon receiving $\mathsf{tx} = (\texttt{SFILL}, \mathsf{RID})$ from P_i:
1. Send $\texttt{TIME}$ to $\mathcal{G}_{\mathsf{refClock}}$ and receive $\mathsf{currTime}$. Set $\mathsf{TIMER}[\mathsf{RID}] = \mathsf{currTime}$
2. Delete $(P_i, \mathsf{RID}, \cdot)$ from $\mathsf{LOCKS}[\mathsf{RID}]$.
3. Send $\texttt{SFILL}, \mathsf{RID}$ to $\mathcal{A}$ and do $\mathsf{buffer} = \mathsf{buffer} \| \mathsf{tx}$

Read : Upon receiving $(\texttt{READ})$ from P_i return state to P_i
Update State : Upon receiving $(\texttt{UPDATE}, \mathsf{tx})$ from $\mathcal{A}$: Delete tx from buffer. Update $\mathsf{state} = \mathsf{state} \| \mathsf{tx}$.
Unlock resources on time-out :
1. If $\mathsf{currTime} - \mathsf{TIMER}[\mathsf{RID}] > \mathsf{MatchTime}$, then delete $(P_i, \mathsf{RID}, \cdot)$ from $\mathsf{LOCKS}[\mathsf{RID}]$
2. For RID if there exist $\texttt{WINNER}$ message and no $\texttt{RFILL}$ message and $\mathsf{currTime} - \mathsf{TIMER}[\mathsf{RID}] > \mathsf{FulfillTime}$, then delete all $(P_i, \cdot)$ from $\mathsf{LOCKS}[\mathsf{RID}]$ and $\mathsf{LOCKS}[\mathsf{RID}]$

Fig. 2. An ideal functionality for private matching

Auxiliary functionalities We will use several building blocks such as anonymous signatures, registration authority, ledger, etc. in our protocols. We describe them briefly:

Clock functionality. $\mathcal{G}_{\mathsf{refClock}}$ (defined in [8]) captures a global reference clock. When queried with (`TIME`) command it returns $\mathsf{currTime}$ to the calling entity. This functionality provides an abstract notion of time and only the environment $\mathcal{Z}$ can update it. Parties do not use this function, only $\mathcal{F}_{\mathsf{PrivateMatch}}$ uses this functionality as a sub-routine. This functionality is a simple counter that is incremented by the environment. For our protocols we only require such an incrementing counter. Alternatively one can also assume that time is realized with respect to block height. For example, a supplier's resources may be locked for k blocks where k is a parameter of the system.

Group signature functionality. $\mathcal{G}_{\mathsf{gsign}}$ (defined in [2]) provides an interface of `gSETUP`, `gJOIN`, GKGen, `gENROLL`, `gSIGN`, `gVERIFY`, `gOPEN`, `gGET`. There are two types of players associated to the functionality. The group manager GM and the set of parties. The functionality allows a party P_j to join the group (using `gENROLL`) only if the GM gives the approval. After joining P_j can ask the ideal functionality to generate signatures (using `gSIGN`) on behalf of the group. A party P_l can ask the ideal functionality to de-anonymize ("open" a certain signature (using `gOPEN`), and the $\mathcal{G}_{\mathsf{gsign}}$ will do so if allowed by GM. An instance of the functionality for group with identifier GID is denoted as $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$.

Registration functionality. $\mathcal{G}_{\mathsf{reg}}$ described in Fig 3 abstracts the registration process. Command `REG` allows parties to join the system without any role, that they can later update using the `UPD` command. $\mathcal{G}_{\mathsf{reg}}$ verifies if the party is eligible for this update by evaluating predicate $\mathsf{ValidReg}$, and if so it returns a certificate cert. Any party verify that a cert is valid by sending a `VERIFY` command.

$\mathcal{G}_{\mathsf{reg}}$

This functionality is parameterized by a function $\mathsf{ValidReg}$ and maintains a list $\mathcal{L}_{\mathtt{REG}}$
- Upon receiving $(\mathtt{REG}, \mathsf{roles})$ from a party P_i send (P_i, roles) to $\mathcal{A}$ and get back cert_i. Store $(P_i, \mathsf{roles}, \mathsf{cert}_i)$ in $\mathcal{L}_{\mathtt{REG}}$.
- Upon receiving $(\mathtt{UPD}, \mathsf{prof}, \mathsf{roles})$ from a party P_i, check if $\mathsf{ValidReg}(P_i, \mathsf{prof}, \mathsf{roles}) = 1$. If yes, send $(P_i, \mathsf{prof}, \mathsf{roles})$ to $\mathcal{A}$ and get back cert_i. Update entry $(P_i, \mathsf{roles}, \cdot)$ in $\mathcal{L}_{\mathtt{REG}}$, with $(P_i, \mathsf{roles}, \mathsf{cert}_i)$.
- Upon receiving $(\mathtt{VERIFY}, \mathsf{cert}^*, P^*, \mathsf{roles})$ from a party P_i or a functionality $\mathcal{F}$, check if $(P^*, \mathsf{roles}, \mathsf{cert}^*)$ exists in $\mathcal{L}_{\mathtt{REG}}$. If yes, return 1 else 0.

$\mathsf{ValidReg}(P_i, \mathsf{prof}, \mathsf{roles})$:
- If $\mathsf{prof} = \mathsf{GID}$ and "$\mathsf{customer}$" $\in \mathsf{roles}$, send gGET to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$ to receive $\mathcal{D}$. If $P_i \in \mathcal{D}$, output 1.
- If $\mathsf{prof} = \mathsf{res}$ and "$\mathsf{supplier}$" $\in \mathsf{roles}$, check $\mathsf{validResource}(P_i, \mathsf{res}) = 1$. If yes, return 1.

Fig. 3. The registration functionality

Smart Ledger functionality. The smart-ledger functionality $\mathcal{G}_{\mathsf{smartchain}}$ abstract the operations of a shared ledger where transactions are validated and then added to the ledger. The ledger is denoted by the global state state that all parties can read. Upon receiving a transaction from a party, the $\mathcal{G}_{\mathsf{smartchain}}$ functionality first validates (see Fig 8) the transaction and then adds the transaction to the state.

$\mathcal{G}_{\mathsf{smartchain}}$

The functionality is parameterized by a $\mathsf{ValidateTxn}$ function (defined in Fig 8). The functionality maintains a global state.
Validate transactions : Upon receiving tx from a party P_i. If $\mathsf{ValidateTxn}(\mathsf{tx}) = 1$, do $\mathsf{state} = \mathsf{state} \| \mathsf{tx}$. Else ignore.
Read : Upon receiving `READ` from a party P_i, return state.

Fig. 4. The $\mathcal{G}_{\mathsf{smartchain}}$ functionality

3 The **PrivateMatch** Protocol

In this section we provide a detailed description of our PrivateMatch, and prove that securely realizes the ideal functionality $\mathcal{F}_{\mathsf{PrivateMatch}}$. We describe our protocol using the UC formalism below:
Protocol Overview. The protocol PrivateMatch uses the ideal functionalities $\mathcal{G}_{\mathsf{reg}}$, $\mathcal{G}_{\mathsf{gsign}}$ [2] and $\mathcal{G}_{\mathsf{smartchain}}$ described above. Parties create and send transactions to the $\mathcal{G}_{\mathsf{smartchain}}$ functionality. If valid, the transaction is added to a global state that can be read by any party.

Registration and Profile Updates

Upon receiving command I from the environment $\mathcal{Z}$ the customer does the following:
Register If $I = \texttt{REG}$
1. Send $(\texttt{REG}, [\mathsf{roles}])$ to $\mathcal{G}_{\mathsf{reg}}$ and receive cert. Send $(\mathsf{tx} = \texttt{REG}, (P_i, \mathsf{cert}))$ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive $(\texttt{ACCEPTED}, b)$.
2. Create keys : Generate encryption keys $(\mathsf{Enc.pk}_i, \mathsf{Enc.sk}_i) \leftarrow \mathsf{Enc.KGen}(1^\lambda)$ and signature keys $(\mathsf{Sig.vk}_i, \mathsf{Sig.sk}_i) \leftarrow \mathsf{Sig.KGen}(1^\lambda)$. Publish $(\mathsf{Sig.pk}_i, \mathsf{Enc.pk}_i)$

Customer : Join group If $I = (\texttt{gJOIN}, \mathsf{GID})$, send $(\texttt{gENROLL})$ to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$ and receive back bit b.
Update profile If $I = (\texttt{UPD}, \mathsf{prof}, \mathsf{roles})$
1. As supplier : Send $(\texttt{UPD}, \mathsf{res}_i, [\mathsf{roles}])$ to $\mathcal{G}_{\mathsf{reg}}$ and receive cert. Send $(\mathsf{tx} = (\texttt{UPD}, (P_i, \mathsf{cert})))$ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive $(\texttt{ACCEPTED}, b)$
2. As customer : Send $(\texttt{UPD}, \mathsf{GID}, [\mathsf{roles}])$ to $\mathcal{G}_{\mathsf{reg}}$. Receive cert and send $(\mathsf{tx} = (\texttt{UPD}, (P_i, \mathsf{cert})))$ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive $(\texttt{ACCEPTED}, b)$

Fig. 5. Registration and Updates

Registration and profile updates. Before participating in the protocol, parties must register with the system by invoking the $\mathcal{G}_{\mathsf{reg}}$ functionality and receiving a certificate cert. The party then prepares a transaction with the certificate and its identity $\mathsf{tx} = (\texttt{REG}, (P_i, \mathsf{cert}))$ and sends it to the $\mathcal{G}_{\mathsf{smartchain}}$ functionality who updates state. Once registered, a party updates its profile as a customer or supplier (or both). To join a group GID, the party sends gENROLL command to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$. The party's profile is then updated using the UPD interface of $\mathcal{G}_{\mathsf{reg}}$. Similarly the party uses the UPD interface to update its profile as a supplier.
Request for service. To request resources for an implementation of design the customer first prepares an anonymous PRE-REQ transaction (signed under its group GID) which only includes the resources (denoted res) it would require. Suppliers who are interested in fulfilling this request, send an INTRST transaction, which includes an encryption key pk_j The customer then picks a set of suppliers from the interested set of suppliers and creates the REQ transaction, where the design is encrypted (denoted C_d) with a fresh key k_{RID}. The key k_{RID} is then encrypted (denoted C^j_{key}) under the public keys (pk_j) of the interested suppliers. Lastly, an encryption $\mathsf{pk}_{\mathsf{bid}}$ is also included that will be used by the suppliers to encrypt their bid values. As described earlier, we chain transactions for the same RID using puzzles. Hence, every transaction from the customer for a specific RID contains the output hash of a collision-resistant hash function (CRHF), and any follow up transaction must contain the pre-image of hash. This is done to ensure that the same group member in the group is continuing the protocol. Specifically the transaction is $(\texttt{REQ}, (\mathsf{GID}, (\mathsf{RID}, \mathsf{pk}_{\mathsf{bid}}, \{C^j_{\mathsf{key}}\}_{j \in \mathsf{bidders}}, C_d, \mathsf{hash}_1, \mathsf{nonce}_0), \sigma))$. Note that the design is encrypted and can be decrypted only by the chosen suppliers.
Bidding and matching. To bid on a request, a supplier first decrypts the encrypted keys to retrieve the symmetric key k_{RID} with which they decrypt the ciphertext and get the design. The supplier then encrypts its bid using the

public key pk_{bid}, and send BID transaction containing the encrypted bid, where $\mathtt{BID} = \mathsf{Sig}_{\mathsf{sk}_i}((\mathsf{RID}, C_{\mathsf{bid}}))$. Since the bid is encrypted under the pk_{bid}, only the customer can learn the bid value of the supplier. The $\mathcal{G}_{\mathsf{smartchain}}$ functionality *locks* the resources of the bidders at this point. The customer then decrypts the encryptions to get the bids, perform its local decision to select a winner, and finally creates a transaction (WINNER) that includes P^* which is the identity of the winner. Once confirmed, the resources of the suppliers that were not selected as winner are unlocked.

Customer: requesting and matching

Pre-request If $I = (\mathtt{PRE\text{-}REQ}, \mathsf{RID}, \mathsf{res})$
1. Sample $\mathsf{nonce}_0 \leftarrow \{0,1\}^\lambda$ and compute $\mathsf{hash}_0 = H(\mathsf{nonce}_0 \| \mathsf{RID})$
2. Send $(\mathtt{gSIGN}, \mathsf{GID}, (\mathsf{hash}_0, \mathsf{res}, \mathsf{RID}))$ to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$ and receive back σ. Send $\mathsf{tx} = (\mathtt{PRE\text{-}REQ}, \mathsf{GID}, ((\mathsf{hash}_0, \mathsf{res}, \mathsf{RID}), \sigma))$ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive $(\mathtt{ACCEPTED}, b)$

Request If $I = (\mathtt{REQ}, \mathsf{RID}, \mathsf{design}, \mathsf{bidders})$
1. Generate design encryption key $k_{\mathsf{RID}} \leftarrow \mathsf{PrivKGen}(1^\lambda)$. Encrypt design: $C_d \leftarrow \mathsf{Enc}(k_{\mathsf{RID}}, \mathsf{design})$.
2. For each $P_j \in \mathsf{bidders}$ - create $C^j_{\mathsf{key}} \leftarrow \mathsf{Enc}(\mathsf{pk}_j, k_{\mathsf{RID}})$
3. Generate bid encryption keys $(\mathsf{pk}_{\mathsf{bid}}, \mathsf{sk}_{\mathsf{bid}}) \leftarrow \mathsf{KGen}(1^\lambda)$. Sample $\mathsf{nonce}_1 \leftarrow \{0,1\}^\lambda$ and compute $\mathsf{hash}_1 = H(\mathsf{nonce}_1 \| \mathsf{RID})$.
4. Send $(\mathtt{gSIGN}, \mathsf{GID}, (\mathsf{RID}, \mathsf{pk}_{\mathsf{bid}}, \{C^j_{\mathsf{key}}\}_{j \in \mathsf{bidders}}, C_d, \mathsf{hash}_1, \mathsf{nonce}_0))$ to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$ and receive σ. Send $\mathsf{tx} = (\mathtt{REQ}, (\mathsf{GID}, (\mathsf{RID}, \mathsf{pk}_{\mathsf{bid}}, \{C^j_{\mathsf{key}}\}_{j \in \mathsf{bidders}}, C_d, \mathsf{hash}_1, \mathsf{nonce}_0), \sigma))$ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive $(\mathtt{ACCEPTED}, b)$

Match If $I = (\mathtt{WINNER}, \mathsf{RID}, P^*)$
1. Retrieve set of encrypted bids $\{\mathsf{RID}, C^j_{\mathsf{bid}}\}_{j \in \mathsf{bidders}}$ from state
2. Compute $\mathsf{bid}_j = \mathsf{Dec}(\mathsf{sk}_{\mathsf{bid}}, C^j_{\mathsf{bid}})$. Ignore, if decryption fails.
3. Sample $\mathsf{nonce}_2 \leftarrow \{0,1\}^\lambda$ and compute $\mathsf{hash}_2 = H(\mathsf{nonce}_2 \| \mathsf{RID})$
4. Send $(\mathtt{gSIGN}, \mathsf{GID}, (\mathsf{hash}_2, \mathsf{nonce}_1, \mathsf{RID}, P^*))$ to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$ and receive σ. Send $\mathsf{tx} = (\mathtt{WINNER}, (\mathsf{GID}, (\mathsf{hash}_2, \mathsf{nonce}_1, \mathsf{RID}, P^*), \sigma))$ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive $(\mathtt{ACCEPTED}, b)$

Fulfilling If $I = \mathtt{RFILL}$
1. Send $(\mathtt{gSIGN}, \mathsf{GID}, \mathsf{fulfill})$ to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$ and receive σ. Send $\mathsf{tx} = (\mathtt{RFILL}, (\mathsf{GID}, \mathsf{fulfill}, \sigma))$ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive $(\mathtt{ACCEPTED}, b)$

Fig. 6. Customer protocols

Supplier: interest, bid, fulfill and dispute

Interest If $I = (\mathtt{INTRST}, \mathsf{RID})$
1. Read PRE-REQ message from state with RID.
2. Create interest message $(\mathsf{RID}, \mathsf{pk}_i)$ and create a signature $\sigma = \mathsf{Sig}_{\mathsf{sk}_i}(\mathsf{RID}, \mathsf{pk}_i)$.
3. Send $(\mathtt{INTRST}, ((\mathsf{RID}, \mathsf{pk}_i), \sigma))$ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive $(\mathtt{ACCEPTED}, b)$

Bid If $I = (\mathtt{BID}, \mathsf{bid})$
1. From state get $(\mathsf{RID}, \mathsf{pk}_{\mathsf{bid}}, \{C^j_{\mathsf{key}}\}_{j \in \mathsf{bidders}}, C_d, \mathsf{hash}_1, \mathsf{nonce}_0)$
2. Ignore if $i \notin \mathsf{bidders}$. Else compute $k^*_{\mathsf{RID}} = \mathsf{Dec}(\mathsf{sk}_i, C^i_{\mathsf{key}})$ and compute $\mathsf{design}^* = \mathsf{Dec}(k_{\mathsf{RID}}, C_d)$.
3. Encrypt bid as $C_{\mathsf{bid}} = \mathsf{Enc}(\mathsf{pk}_{\mathsf{bid}}, \mathsf{bid})$. Send $(\mathtt{BID}, \mathsf{Sig}_{\mathsf{sk}_i}((\mathsf{RID}, C_{\mathsf{bid}})))$ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive $(\mathtt{ACCEPTED}, b)$

Fulfill If $I = \mathtt{SFILL}$
1. Create $\sigma = \mathsf{Sig}_{\mathsf{sk}_i}(\mathtt{SFILL}, C_{\mathsf{deliveryPrf}}, \mathsf{RID})$.
2. Send $(\mathtt{SFILL}, (\mathsf{RID}, \sigma))$ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive $(\mathtt{ACCEPTED}, b)$

Fig. 7. Supplier protocols

Function ValidateTxn

Unlock Resources As is done in $\mathcal{F}_{\mathsf{PrivateMatch}}$.
Validate registration and updates If $\mathsf{tx} = (\mathtt{REG}, v, \mathsf{cert}, \mathsf{roles})$ or $(\mathtt{UPD}, v, \mathsf{cert}, \mathsf{roles})$, send $(\mathtt{VERIFY}, \mathsf{cert}, P, \mathsf{roles})$ to $\mathcal{G}_{\mathsf{reg}}$. Output the bit returned.
Validate pre-request If $\mathsf{tx} = (\mathtt{PRE\text{-}REQ}, m, \sigma)$ from a party P_i, send $(\mathtt{gVERIFY}, m, \sigma)$ to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$. Output the bit returned.
Validate interest : Upon receiving $(\mathtt{INTRST}, m, \sigma)$ from a party P_i, check $\mathsf{Sig.Vrf}(\mathsf{vk}_i, m, \sigma) = 1$. If $\mathsf{res} \subset \mathsf{res}_i$, retrun 1.
Validate request : If $\mathsf{tx} = (\mathtt{REQ}, m, \sigma)$ Send $(\mathtt{gVERIFY}, (m, \sigma))$ to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$. If 1 returned, check $\mathsf{VrfSame}(m, \mathsf{state}) = 1$, if yes, return 1.
Validate bid: If $\mathsf{tx} = (\mathtt{BID}, m, \sigma)$ Check $\mathsf{Sig.Vrf}(\mathsf{vk}_i, m, \sigma) = 1$. Get RID from m and check $\mathsf{canServe}(P_i, \mathsf{state}, \mathsf{RID}, \mathsf{LOCKS})$, add $(P_i, \mathsf{RID}, \mathsf{res})$ to $\mathsf{LOCKS}[\mathsf{RID}]$, update $\mathsf{TIMER}[\mathsf{RID}] = \mathsf{currTime}$ and return 1.
Validate Match: If $\mathsf{tx} = (\mathtt{WINNER}, m, \sigma)$ send $(\mathtt{gVERIFY}, m, \sigma)$ to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$. If 1 returned, check $\mathsf{VrfSame}(m, \mathsf{state}) = 1$, if yes, return 1. Let P^* be the winner according to m and res be the resource for RID. For all $P_j \neq P^*$, remove $(P_j, \cdot)$ from the $\mathsf{LOCKS}[\mathsf{RID}]$ and update $\mathsf{TIMER}[\mathsf{RID}] = \mathsf{currTime}$
Validate requester fulfill : If $\mathsf{tx} = (\mathtt{RFILL}, m, \sigma)$, check $\mathsf{VrfSame}(m, \mathsf{state}) = 1$. If yes , send $(\mathtt{gVERIFY}, m)$ to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$. If 1 returned, output 1.
Validate supplier fulfill : If $\mathsf{tx} = (\mathtt{SFILL}, m, \sigma)$: Get RID from m. Check $\mathsf{Sig.Vrf}_{\mathsf{vk}_i}(m)$. If yes, then return 1, and remove $(P_j, \cdot)$ from LOCKS
Function $\mathsf{VrfSame}(m, \mathsf{state})$: Retrieve nonce_i from m. If $\exists \mathsf{hash}_i \in \mathsf{state}$ s.t. $\mathsf{hash}_i = H(\mathsf{nonce}_i)$, output 1, else output 0.
Function canServe checks if the supplier has sufficient unlocked resources to serve the customer.

Fig. 8. The validation function

3.1 Security Proof

Theorem 1. (Security in Presence of Malicious Customers) *The protocol* PrivateMatch *UC realizes the* $\mathcal{F}_{\mathsf{PrivateMatch}}$ *ideal functionality in the* $\mathcal{G}_{\mathsf{gsign}}, \mathcal{G}_{\mathsf{reg}}, \mathcal{G}_{\mathsf{smartchain}}$*-hybrid world assuming collision-resistant hash functions [15], secure "special" symmetric key encryption [20], EUF-CMA signature [15], secure commitment schemes [15] and CPA-secure encryption [15] in the presence of a PPT adversary that corrupts a subset of the customers .*

Proof. In order to prove UC security we show that there exists a simulator interacting with $\mathcal{F}_{\mathsf{PrivateMatch}}$ that generates a transcript that is indistinguishable from the transcript generated by the real-world adversary running protocol PrivateMatch. We give a high-level description of the simulator $\mathcal{S}_r$ and give an intuition why security is guaranteed. We present detailed proofs in the full version of the paper [21]. For a PRE-REQ command, the simulator only receives the GID and not the identity of the party calling the PRE-REQ command. The simulator simulates the $\mathcal{G}_{\mathsf{gsign}}$ functionality and records the message-signature pair without the identity of the party. This guarantees anonymity within the group GID. For the REQ command, the simulator encrypts 0 instead of design. By CPA security of the encryption scheme the simulation is indistinguishable from the real-world and thus we achieve **service confidentiality**. The simulator aborts if it is able to create a REQ transaction that corresponds to the RID of an honest user. This occurs with negligible probability since we use CRHF and thus we guarantee **requester soundness**. For the BID command the simulator encrypts 0 instead of the bid value to get **bid confidentiality**. In the case of a malicious customer, the simulator simulates a key-exchange and sends an encryption of 0 to the customer instead of encryption of its secret key.

Theorem 2. *(***Security in Presence of Malicious Suppliers***) The protocol* PrivateMatch *UC realizes the* $\mathcal{F}_{\mathsf{PrivateMatch}}$ *ideal functionality in the* $\mathcal{G}_{\mathsf{gsign}}, \mathcal{G}_{\mathsf{reg}}, \mathcal{G}_{\mathsf{smartchain}}$*-hybrid world assuming collision-resistant hash functions [15], EUF-CMA signa-*

ture [15], secure commitment schemes [15] and CPA-secure encryption [15] in the presence of a PPT adversary that corrupts a subset of the suppliers.

Proof. Like the malicious requesters case we need to show that there exists a simulator ($\mathcal{S}_s$) interacting with $\mathcal{F}_{\mathsf{PrivateMatch}}$ that generates a transcript that is indistinguishable from the transcript generated by the real-world adversary running protocol PrivateMatch. For an `INTRST` transaction from a corrupt P_i, the simulator aborts with UnforgeabilityError if the signature corresponds to that of an honest party. By the unforgeability property of the signature schemes, this abort occurs with negligible probability. Moreover, when the command is sent to the $\mathcal{F}_{\mathsf{PrivateMatch}}$ functionality, it checks the supplier is capable of fulfilling the request. This guarantees the **supplier completeness property**

Remark on fairness. We do not tackle the problem of fairness or disputes in this work and consider it out of scope. There are numerous dispute resolution solutions in the literature [12] [16] and we claim that an appropriate technique could be used in our setting as well.
Implementing auxiliary functionalities. We present some intuition on how to realize the auxiliary functionalities - $\mathcal{G}_{\mathsf{reg}}$: Verification of identity can be done with systems like CanDID [22] that allows parties to port credentials from legacy systems (e.g. social security numbers) whereas verifying resources of suppliers may be done by some external auditing agencies. $\mathcal{G}_{\mathsf{gsign}}$: We use the $\mathcal{G}_{\mathsf{gsign}}$ functionality as defined in the work by Ateniese et al. [2] and the protocol realizing this functionality is presented in Section 5 of [2]. This protocol has a single group manager(GM) which goes against the spirit of a decentralized setting. One can replace the GM with multiple managers that enable threshold group signatures. [2] [4] present a fair traceable group signature scheme that allow specific fairness authorities to open signatures where the group manager encrypts the identity of the party under the pk of the fairness authorities, and to open, the fairness authorities run a threshold decryption protocol. Similarly [6, 11] present protocols for distributed tracing using tag-based encryption to open the signatures of parties.

4 Implementation and evaluation

Our implementation framework for PrivateMatch focuses on (i.) transactional behavior that captures general marketplace business logic e.g., requesting for quotes, bidding, etc; (ii.) transaction anonymity using group signatures.

We introduce new blockchain transaction types into an open-source blockchain platform that is amenable to the desired extensions. BigchainDB [23] is a blockchain database that possesses blockchain characteristics. Its architecture involves a fixed set of nodes - *validators*, and is Byzantine Fault Tolerant (BFT) (up to a third nodes may fail). Its key architectural components include Tendermint [19] (for consensus), the BigchainDB Server (for syntactic and semantic validation of transactions), and a local MongoDB [1] database (for blockchain storage) on every validator node.
Extending BigChainDB's Transaction Model. BigChainDB allows transfer of assets, and its transaction model is a "declarative", attribute/key-value model.

[2] In threshold group signatures, a signature can be de-anonymized only if a threshold of managers all agree to perform de-anonymization

We refer to our extension of BigChainDB as SmartChainDB. SmartChainDB extends the validator algorithms in BigChainDB according to the ValidateTxn. We implement "locking" of resources as a transfer of resources to an "escrow" account (a designated non-user account used for holding resources). To release the resources, the validators issue a transaction back to the owner. Note that SmartChainDB is deployed as a network of its own with the validators running new validation algorithms. They run the same consensus algorithms (Tendermint) and are incentivized to do as in BigChainDB
Extending BigChainDB's Privacy Model. We enable parties to use group signatures [5] instead of regular Ed25519 signatures provided by BigChainDB's library. The core building block of this scheme is a re-randomizable signature scheme (Pointcheval-Sanders scheme from [27]) and implemented in [13]. Our implementation is in Rust and uses python-based Cherrypy server as a wrapper to call the Rust cryptographic functions as a service.

The objective of our evaluation of SmartChainDB was twofold: (i.) verify that the newly introduced transaction types can support simulated marketplace workloads under reasonable performance bounds and (ii.) that the overhead of the group signature implementation did not deem the protocol impractical

Experimental Setup We set up a private test network on 16 machines with an Intel Westmere E56 Quad-core 3.46 GHz CPU, 8 GB memory, running 64-bit Ubuntu with kernel v4.15.0. We set up 12 validator nodes, with each node running its SmartchainDB server, Tendermint v0.31.5, and MongoDB v3.6 instances. For workload simulations, we set up the driver on 4 VM instances, running the customer and supplier code to trigger different transaction types. The drivers produce 80-100 transactions per second and send them to the validator nodes. The `CREATE` and `TRANSFER` (available in the vanilla BigChainDB implementation) transactions are evaluated under the same workload.

Latency Overhead We measure the commit latency (time between a validator node receiving a transaction and its commit into the blockchain. We compare the `PRE-REQ` and `REQ` with `CREATE` transactions because those transactions are semantically closest. Similarly, the vanilla `TRANSFER` transaction is

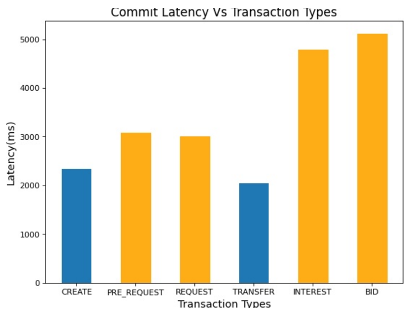


Fig. 9. Performance Comparison of Transactions

	Mean	Median	Std. Dev
`gSIGN` ([5] + [28])	9.77	9.65	0.92
BigchainDB Sign	0.75	0.52	0.84
`gVERIFY` ([5] + [28])	11.13	11.03	0.30
BigchainDB Verify	1.00	0.76	0.64

Fig. 10. Comparing group signatures and BigchainDB signatures (ms)

similar to `INTRST` and `BID` transaction. Figure 9 shows the average commit latency for every transaction types under the workload discussed above. The blue bars are for the native transactions and the orange ones for the new transactions. Overall, the results show the expected trend with the newer, more complex transactions having higher latency than their traditional "counterparts" due to the required additional validation overhead. In practical terms, these latency differences can be considered as a relatively minor trade-off for supporting more involved market-place events. Note that, these experiments were carried out in the non-private and non-anonymous settings, where we do not consider group signatures, encryptions, etc. Finally for the group signatures, we measure the time taken to sign and verify messages using our implementation of group signatures with the signature scheme used in BigchainDB (eddsa-sha512). We ran 200 sign and verify algorithms and observe that the group signature signing and verification take 10× that of regular signatures.

5 Conclusion and Future Work

In this paper we present a protocol for decentralized private interactive matching and prove that it is UC secure. We also extend an existing blockchain database system (BigChainDB) to implement private matching by introducing new transactions

An interesting future direction would be to enhance the privacy of the request that is sent by the customer to the suppliers. This may be achieved using techniques such as fully homomorphic encryption or garbled circuits. Another future direction would be to specify protocols and transactions for dispute resolution for the SmartChainDB system.

References

1. Mongodb, the most popular database for modern apps. https://www.mongodb.com/
2. Ateniese, G., Camenisch, J., Hohenberger, S., De Medeiros, B.: Practical group signatures without random oracles. (2005)
3. Benhamouda, F., Halevi, S., Halevi, T.: Supporting private data on hyperledger fabric with secure multiparty computation. IBM Journal of Research and Development **63**(2/3), 3–1 (2019)
4. Benjumea, V., Choi, S.G., Lopez, J., Yung, M.: Fair traceable multi-group signatures. In: International Conference on Financial Cryptography and Data Security. pp. 231–246. Springer (2008)
5. Bichsel, P., Camenisch, J., Neven, G., Smart, N.P., Warinschi, B.: Get shorty via group signatures without encryption. In: International Conference on Security and Cryptography for Networks. pp. 381–398. Springer (2010)
6. Blömer, J., Juhnke, J., Löken, N.: Short group signatures with distributed traceability. In: International Conference on Mathematical Aspects of Computer and Information Sciences. pp. 166–180. Springer (2015)
7. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: Proceedings 42nd IEEE Symposium on Foundations of Computer Science. pp. 136–145. IEEE (2001)
8. Canetti, R., Hogan, K., Malhotra, A., Varia, M.: A universally composable treatment of network time. In: 2017 IEEE 30th Computer Security Foundations Symposium (CSF). pp. 360–375. IEEE (2017)
9. Chang, S.E., Chen, Y.C., Lu, M.F.: Supply chain re-engineering using blockchain technology: A case of smart contract based tracking process. Technological Forecasting and Social Change **144**, 1–11 (2019)
10. Galal, H.S., Youssef, A.M.: Verifiable sealed-bid auction on the ethereum blockchain. In: International Conference on Financial Cryptography and Data Security. pp. 265–278. Springer (2018)
11. Ghadafi, E.: Efficient distributed tag-based encryption and its application to group signatures with efficient distributed traceability. In: International Conference on Cryptology and Information Security in Latin America. pp. 327–347. Springer (2014)
12. Goldfeder, S., Bonneau, J., Gennaro, R., Narayanan, A.: Escrow protocols for cryptocurrencies: How to buy physical goods using bitcoin. In: International Conference on Financial Cryptography and Data Security. pp. 321–339. Springer (2017)
13. Hyperledger: hyperledger/ursa. https://github.com/hyperledger/ursa/tree/master/libzmix/src/signatures/ps
14. Kabi, O.R., Franqueira, V.N.: Blockchain-based distributed marketplace. In: International Conference on Business Information Systems. pp. 197–210. Springer (2018)
15. Katz, J., Lindell, Y.: Introduction to modern cryptography. CRC press (2020)
16. Klems, M., Eberhardt, J., Tai, S., Härtlein, S., Buchholz, S., Tidjani, A.: Trustless intermediation in blockchain-based decentralized service marketplaces. In: International Conference on Service-Oriented Computing. pp. 731–739. Springer (2017)

17. Kosba, A., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In: 2016 IEEE symposium on security and privacy (SP). pp. 839–858. IEEE (2016)
18. Kumar, G., Saha, R., Buchanan, W.J., Geetha, G., Thomas, R., Rai, M.K., Kim, T.H., Alazab, M.: Decentralized accessibility of e-commerce products through blockchain technology. Sustainable Cities and Society **62**, 102361 (2020)
19. Kwon, J.: Tendermint: Consensus without mining. Draft v. 0.6, fall **1**(11) (2014)
20. Lindell, Y., Pinkas, B.: A proof of security of yao's protocol for two-party computation (2006)
21. Madathil, V., Scafuro, A., Anyanwu, K., Qiao, S., Pateria, A., Starly, B.: Preserving buyer-privacy in decentralized supply chain marketplaces. Cryptology ePrint Archive, Report 2022/105 (2022), https://ia.cr/2022/105
22. Maram, D., Malvai, H., Zhang, F., Jean-Louis, N., Frolov, A., Kell, T., Lobban, T., Moy, C., Juels, A., Miller, A.: Candid: Can-do decentralized identity with legacy compatibility, sybil-resistance, and accountability. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1348–1366. IEEE (2021)
23. McConaghy, T., Marques, R., Müller, A., De Jonghe, D., McConaghy, T., McMullen, G., Henderson, R., Bellemare, S., Granzotto, A.: Bigchaindb: a scalable blockchain database. white paper, BigChainDB (2016)
24. Montecchi, M., Plangger, K., Etter, M.: It's real, trust me! establishing supply chain provenance using blockchain. Business Horizons **62**(3), 283–293 (2019)
25. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2012), 28 (2008), https://bitcointalk.org/index.php?topic=321228.0
26. Özyilmaz, K.R., Doğan, M., Yurdakul, A.: Idmob: Iot data marketplace on blockchain. In: 2018 crypto valley conference on blockchain technology (CVCBT). pp. 11–19. IEEE (2018)
27. Pointcheval, D., Sanders, O.: Short randomizable signatures https://eprint.iacr.org/2015/525.pdf
28. Pointcheval, D., Sanders, O.: Short randomizable signatures. In: Cryptographers' Track at the RSA Conference. pp. 111–126. Springer (2016)
29. Ranganthan, V.P., Dantu, R., Paul, A., Mears, P., Morozov, K.: A decentralized marketplace application on the ethereum blockchain. In: 2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC). pp. 90–97. IEEE (2018)
30. Sasson, E.B., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: 2014 IEEE Symposium on Security and Privacy. pp. 459–474. IEEE (2014)
31. Soska, K., Christin, N.: Measuring the longitudinal evolution of the online anonymous marketplace ecosystem. In: 24th {USENIX} security symposium ({USENIX} security 15). pp. 33–48 (2015)
32. Soska, K., Kwon, A., Christin, N., Devadas, S.: Beaver: A decentralized anonymous marketplace with secure reputation. IACR Cryptol. ePrint Arch. **2016**, 464 (2016)
33. Subramanian, H.: Decentralized blockchain-based electronic marketplaces. Communications of the ACM **61**(1), 78–84 (2017)
34. Thio-ac, A., Domingo, E.J., Reyes, R.M., Arago, N., Jorda Jr, R., Velasco, J.: Development of a secure and private electronic procurement system based on blockchain implementation. arXiv preprint arXiv:1911.05391 (2019)
35. Thio-ac, A., Serut, A.K., Torrejos, R.L., Rivo, K.D., Velasco, J.: Blockchain-based system evaluation: The effectiveness of blockchain on e-procurements. arXiv preprint arXiv:1911.05399 (2019)
36. Uesugi, T., Shijo, Y., Murata, M.: Short paper: Design and evaluation of privacy-preserved supply chain system based on public blockchain. arXiv preprint arXiv:2004.07606 (2020)
37. Westerkamp, M., Victor, F., Küpper, A.: Blockchain-based supply chain traceability: Token recipes model manufacturing processes. In: 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). pp. 1595–1602. IEEE (2018)
38. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger
39. Xiong, J., Wang, Q.: Anonymous auction protocol based on time-released encryption atop consortium blockchain. arXiv preprint arXiv:1903.03285 (2019)