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 facilitates 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.

Motivated by these concerns and spurred by the development of blockchain technology, recent work [20,23,36] propose to build *decentralized* online marketplace by replacing the middleman with a smart-contract capable blockchain. A blockchain [32,45] 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 clients 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. However, transparency of the transactions exposes the data they contain, which must remain private in marketplaces to maintain a competitive advantage. Hence, even ignoring the cost and complexity of

[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 design.

smart contracts, privacy issues represent a significant hurdle towards a blockchain-based decentralized marketplace.

Existing proposal for decentralized marketplaces [20, 23, 38, 40] 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. Privacy issues concerning the customer's identity can be mitigated with the use of anonymous wallets (e.g. Zcash [37]).

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. The sharing of some initial limited information is to provide suppliers enough of a context to help them determine if they are in a position to consider initiating a response without divulging valuable information. 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. However, rather than share the implant with the general supplier population in the previous interactions, the implant design should be *private* input in interactions involving only shortlisted suppliers that will be able to fulfill the request within the deadline. 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, for specific interactions, they also want to maintain the privacy of some inputs. For example, they may want to keep bid amounts 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 characteristics. 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* – even if it is a new company or a startup, their name should be registered somewhere. 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 [12]. 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 5). 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 [12]. In the UC framework, the security of a system is defined by means of an ideal functionality. The ideal functionality represents the ideal behavior of the system, and the influence/leakage allowed to an adversary. In this paper, we design the ideal functionality $\mathcal{F}_{\mathsf{PrivateMatch}}$ that 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 3. At high-level 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). Hence, the anonymity of the customers depends on the size of the group. However, misbehaving customers can be de-anonymized. *Customer's Input Privacy.* Requests contains public values (e.g., the type of resources required, the deadlines, etc), and private values (e.g., the product design) which may depend on the specific application. From our example earlier, the suppliers were informed that they needed to build an implant made out of medical grade titanium allow in three weeks. This does not reveal any private information about the buyer nor about what is being built. Following such real world scenarios we consider the resources required to be a public value in the rest of the paper. 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 would allow 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 are public. The fact that a supplier is interested in fulfilling a certain request is also 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. This is a desired feature in the manufacturing domain, where public and accurate information about resource utilization is desired to build reputation. We acknowledge that there might be domain where this is not desirable.

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.

Fairness. Our ideal functionality allows both customers and suppliers to show a proof of misbehavior. The validity of the proof is checked via an external algorithm, the instantiation of which depends on the application [2] A misbehaving customer is de-anonymized, while a misbehaving supplier will have its reputation marked.

Some Remarks on the Definition. Our definition of the ideal matching functionality are inspired by the service-oriented marketplaces. In such domain having the supplier's activities public is considered as a feature for building reputation rather than a drawback . Similarly, 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. We note however that one can easily modify $\mathcal{F}_{\mathsf{PrivateMatch}}$ so that, instead of the entire input x, only a predicate $f_S(x)$ of the input is leaked. Here f_S is a predicate that the supplier S can use to assess if it is able to fulfil the request on an input x. We note that f_S will have to be defined by the application domain, and by the suppliers. To realize this however, we might require some inefficient techniques such as secure multiparty computation or fully homomorphic encryption.

A Protocol for Decentralized Private Interactive Matching We provide a protocol that securely instantiates the ideal functionality $\mathcal{F}_{\mathsf{PrivateMatch}}$. Our protocol uses as building block a blockchain with scripting capability such as checking hash and validating (group) signatures. In the protocol description, we abstract the blockchain as an ideal ledger functionality, that we call $\mathcal{G}_{\mathsf{smartchain}}$. We later discuss how to instantiate it by extending an existing blockchain platform BigChainDB (see Section5). To protect the anonymity of customers while ensuring accountability, we use group signatures [4, 8]. These are signatures associated to a group. A group member can use its own private key to generate an anonymous signature on behalf of the entire group. To ensure accountability of misbehaving members, the group is curated by a group manager who has the ability to de-anonymize signatures when necessary. In Section 4.2 we present how this group manager can be decentralized and remove a single point of trust.

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 4, here we discuss only few key points related to how correctness and privacy are guaranteed.

The match is performed completely on the blockchain, by means of transactions published by customers and interested suppliers. The match starts with a customer publishing an anonymous request for proposal (`PREREQ`) transaction indicating the resources needed for its own private project. This transaction contains a unique request id RID, and is signed with a group signature, hence only the group the customer belongs to is leaked. This pre-request transaction is followed up with other transactions where (1) suppliers signal their interest in fulfilling the request; (2) the customer selects a subset of supplier and give instructions on how to retrieve its private design. This transaction will also contain a freshly sampled public key pk_{RID} that will be used to encrypt bids for the anonymous customer. (3) The suppliers publish their encrypted bids (their resources are then locked); (4) the customer announces the winner (the resources of the non-winners are then unlocked). All such transactions are linked through the request id RID (and a chain of puzzles, as described later). A final (5) fulfill transaction is then sent by the customer and supplier to signal the successful completion of the service. If anything went wrong, they could post a dispute transaction.

To ensure correctness and fairness, transactions are deemed valid only if they satisfy certain conditions, and misbehaving parties are publicly identified. For example, a supplier's transaction manifesting interest in participating in the matching is accepted only if the supplier currently has available

[2] Relegating domain-specific validation to an external algorithm is typical in definition of ideal functionality. The same approach indeed was taken by [5], when defining the validation rules of the ideal ledger functionality.

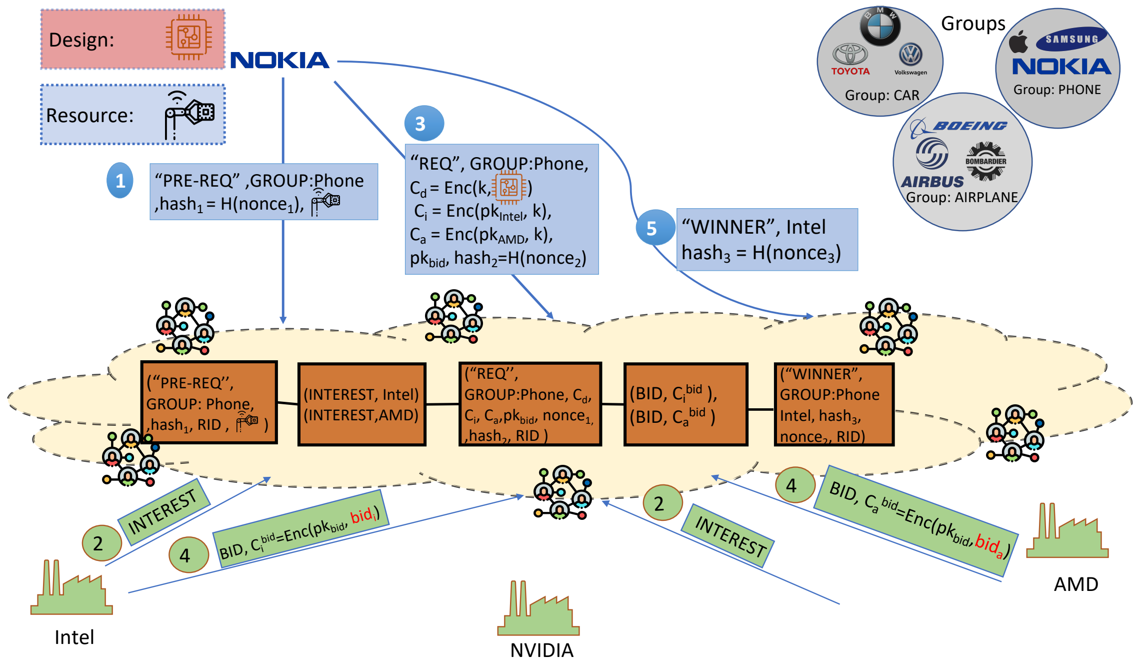
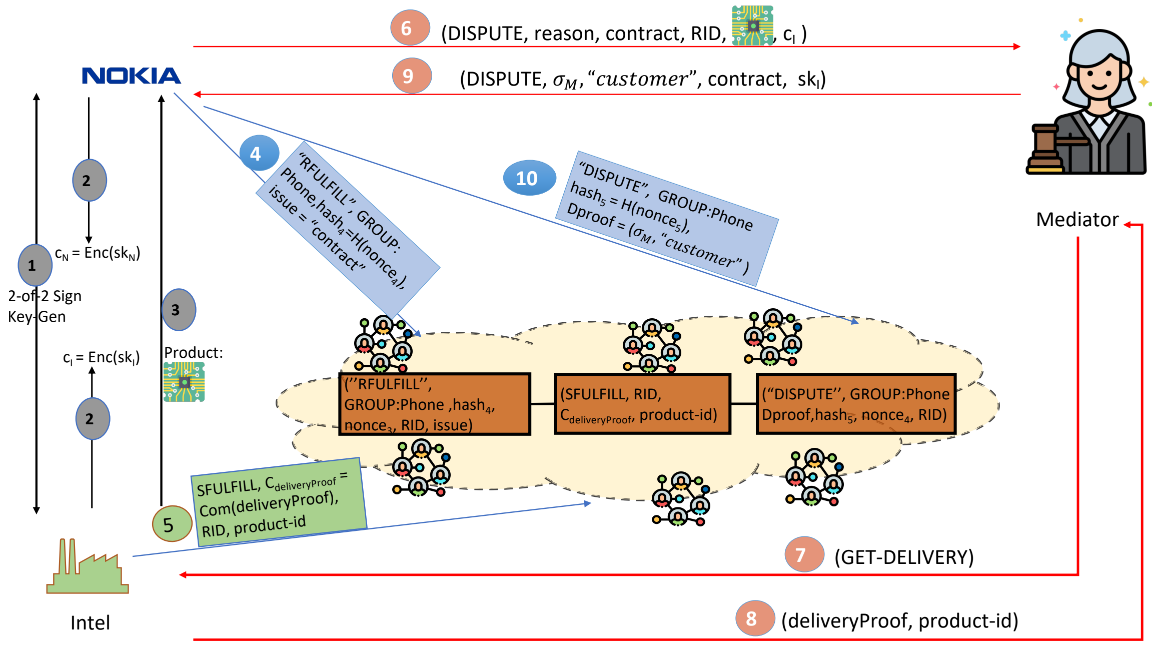

Design:
Resource:
NOKIA
Groups
Group: CAR
Group: PHONE
Group: AIRPLANE
1
"PRE-REQ" ,GROUP:Phone ,hash₁ = H(nonce₁),
3
"REQ", GROUP:Phone, C_d = Enc(k,) C_i = Enc(pk_Intel, k), C_a = Enc(pk_AMD, k), pk_bid, hash₂=H(nonce₂)
5
"WINNER", Intel hash₃ = H(nonce₃)
("PRE-REQ", GROUP: Phone, ,hash₁, RID ,)
(INTEREST, Intel) (INTEREST,AMD)
("REQ", GROUP:Phone, C_d, C_i, C_a,pk_bid, nonce₁, ,hash₂, RID)
(BID, C_i^bid), (BID, C_a^bid)
("WINNER", GROUP:Phone Intel, hash₃, nonce₂, RID)
2
INTEREST
4
BID, C_i^bid=Enc(pk_bid, bid_i)
INTEREST
BID, C_a^bid=Enc(pk_bid, bid_a)
Intel
NVIDIA
AMD

Fig. 1. Protocol Overview: 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.

resources that match the request. Furthermore, a misbehaving customer can be de-anonymized with the help of the group manager.

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, and with the use of ephemeral per-request public keys pk_{RID}. The private design of the customer is never included in a transaction. Rather, it is encrypted and uploaded to another web location (controlled by the customer). 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. The private bids of the suppliers are protected with encryptions, computed under the customer's ephemeral public key pk_{RID}.

Finally, note that, due to the use of anonymous group signatures, if the transactions for the match associated withrequest RID were chained only through RID, a malicious member of the group, could interject the flow by sending follow up transactions for the request RID – even without being the legitimate customer. To prevent this, we chain the transactions through puzzles, in such a way that a customer can compute the next transaction in the flow only if it knows the solution to the puzzle of previous transactions.



6 (DISPUTE, reason, contract, RID, , c_I)
9 (DISPUTE, σ_M, "customer", contract, sk_I)
NOKIA
Mediator
1 2-of-2 Sign Key-Gen
2 c_N = Enc(sk_N)
2 c_I = Enc(sk_I)
3 Product:
4 "RFULFILL", GROUP: Phone,hash_4=H(nonce_4), issue = "contract"
10 "DISPUTE", GROUP:Phone hash_5 = H(nonce_5), Dproof = (σ_M, "customer")
("RFULFILL", GROUP:Phone ,hash_4, nonce_3, RID, issue)
(SFULFILL, RID, C_deliveryProof, product-id)
("DISPUTE", GROUP:Phone Dproof,hash_5, nonce_4, RID)
5 SFULFILL, C_deliveryProof = Com(deliveryProof), RID, product-id
7 (GET-DELIVERY)
8 (deliveryProof, product-id)
Intel

Fig. 2. Overview of a dispute resolution. In this example, Nokia wants to raise dispute because it was delivered a faulty product. As before, 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 After Intel and Nokia were matched, the two parties run key generation for a 2-of-2 threshold signature scheme, such that Nokia receives $\mathsf{sk}_N, \mathsf{pk}$ and Intel receives $\mathsf{sk}_I, \mathsf{vk}$. This signature verification key vk is used to create a 2-of-2 address on another cryptocurrency such as Bitcoin. Nokia sends bid amount of coins to this address, and to spend money from this address one would require both Nokia and Intel's secret keys. ❷ Both parties encrypt their secret key under the public key of the mediator and send it to the other party. ❸ Intel delivers the product to Nokia. ❹ Nokia submits a `RFILL` transaction. Since there is an issue with the product, they set $\mathsf{issue} = \mathsf{contract}$, if no issue then they would set $\mathsf{issue} = \perp$. ❺ Nokia also submits a transaction to prove that it did deliver the product on time. It attaches the commitment of a delivery proof and a unique product-id. ❻ Nokia sends a `DISPUTE` to the mediator. They send the product, the contract, the reason for dispute, an identifier of the transactions on the blockchain that correspond to the current contract and the encryption c_I which is an encryption of the secret key of Intel. ❼ The mediator asks Intel for delivery proofs. ❽ Intel provides the mediator with delivery proofs and the unique digital identifier of the product ❾ The mediator checks the product with respect to the reason for dispute and that all transactions created for this contract are consistent and finally determines that the winner is Nokia. It decrypts c_I to get sk_I. The mediator signs a message ("customer", $\mathsf{contract}$) indicating that Nokia won the dispute and sends sk_I, σ_M, "*customer*" back to Nokia. ❿ Nokia now submits a DISPUTE transaction to the blockchain and sets the proof of winning the dispute as the signature it received from the mediator. Finally, Nokia is also able to retrieve the funds it put into the 2-of-2 address.

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 (to be elaborated on in Section 5). For this reason, we selected to build on a platform BigChainDB [30], which is not a specific blockchain application in itself. Rather, BigChainDB offers an architecture that is extensible and can be used to implement different kinds of blockchain applications. In addition to extending BigChainDB's transaction model, we implemented the *group signature* scheme due to [8] and incorporated it 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 created by these new, more complex marketplace transaction types, as well as the group signature scheme used for privacy. We found that latency of our marketplace transaction types took no more than 2.5× that of traditional transactions. However, in concrete terms this translated to a mere additional 2secs of processing time. The group signature scheme took up to 12× more than the traditional signature scheme. Again, in concrete terms, this translated to only 10ms overhead.

1.2 Related Work

Kosba et al. present Hawk [24], a framework for creating privacy-preserving Ethereum smart contracts. Their framework allows a set of clients to describe a functionality that they want to implement, and it outputs the code for a smart contract, and programs that should be run by the clients and 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. In other words, the facilitator acts as a middleman and learns the inputs of the party, which we want to avoid in this work.

Benhamouda et al. [6] present a framework on top of the Hyperledger Fabric that allows party to send encrypted inputs to the chain. Later, when the inputs are required for a computation, the parties must run an off-chain multiparty computation protocol over the encrypted input. Here, the blockchain is used mainly as an immutable input storage, and the actual computations is performed off-line.

The bidding and match steps in our private match functionality share similarities with sealed-bid auctions. There, bidders simultaneously privately 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. [15] on Ethereum and Xiong et al. [46]). However, such protocols 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 [42] [41]. 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 [23, 33, 36], 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 address in Beaver [39] by employing anonymous wallets and the Zcash blockchain [37]. 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. [14, 25, 31, 43, 44] just to name a few). However, all such work focuses only on the traceability and provenance of the products.

2 Universal Composability

In UC security we consider the execution of the protocol in a special setting involving an environment machine $\mathcal{Z}$, in addition to the honest parties and adversary. In UC, ideal and real models are considered where a trusted party carries out the computation in the ideal model while the actual protocol runs in the real model. The trusted party is also called the ideal functionality. For example the ideal functionality $\mathcal{F}_{\mathsf{PrivateMatch}}$ is a trusted party that provides the functionality of anonymous matching. In the UC setting, there is a global environment (the distinguisher) that chooses the inputs for the honest parties, and interacts with an adversary who is the party that participates in the protocol on behalf of dishonest parties. At the end of the protocol execution, the environment receives the output of the honest parties as well as the output of the adversary which one can assume to contain the entire transcript of the protocol. When the environment activates the honest parties and the adversary, it does not know whether the parties and the adversary are running the real protocol –they are in the real world, or they are simply interacting with the trusted ideal functionality, in which case the adversary is not interacting with any honest party, but is simply "simulating" to engage in the protocol. In the ideal world the adversary is therefore called simulator, that we denote by $\mathcal{S}$.

In the UC-setting, we say that a protocol securely realizes an ideal functionality, if there exist no environment that can distinguish whether the output he received comes from a real execution of the protocol between the honest parties and a real adversary $\mathcal{A}$, or from a simulated execution of the protocol produced by the simulator, where the honest parties only forward date to and from the ideal functionality.

The transcript of the ideal world execution is denoted $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda, z)$ and the transcript of the real world execution is denoted $\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}(\lambda, z)$. A protocol is secure if the ideal world transcript and the real world transcripts are indistinguishable. That is, $\{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda, z\}_{\lambda\in\mathbb{N}, z\in\{0,1\}^*} \equiv \{\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}(\lambda, z)\}_{\lambda\in\mathbb{N}, z\in\{0,1\}^*}$.

3 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 [12](also see Appendix 2), 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. 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.

3.1 Overview of the functionality

Our functionality 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. It follows the formalisms on the UC-framework and it is described in Figures 3

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

Register : Upon receiving $(\mathtt{REG}, P_i, [\mathsf{roles}])$ from P_i, do $\mathcal{P} = \mathcal{P} \cup P_i$. Send $(\mathsf{tx} = (\mathtt{REG}, P_i, [\mathsf{roles}]))$ to $\mathcal{A}$ and do $\mathsf{buffer} = \mathsf{buffer}||\mathsf{tx}$.
Non-adaptive setup : Receive $(\mathtt{CORRUPTED}, b)$ from party P_i
Join Group : Upon receiving $(\mathtt{gJOIN}, \mathsf{GID})$ from a party P_i, update $\mathcal{G}[\mathsf{GID}] = \mathcal{G}[\mathsf{GID}] \cup \{P_i\}$. Send $(\mathtt{gJOIN}, P_i, \mathsf{GID})$ to $\mathcal{A}$ and $(\mathtt{gJOIN}, P_i, \mathsf{GID}, 1)$ to P_i.
Update Profile : Upon receiving $(\mathtt{UPD}, \mathsf{prof}, \mathsf{roles})$ 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 $(\mathsf{tx} = \mathtt{UPD}, P_i, \mathsf{GID})$ 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 $(\mathsf{tx} = \mathtt{UPD}, P_i, \mathsf{res}_i)$ to $\mathcal{A}$ and do $\mathsf{buffer} = \mathsf{buffer}||\mathsf{tx}$.
PreRequest : Upon receiving $(\mathtt{PREREQ}$, GID, 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 $(\mathsf{tx} = \mathtt{PREREQ}, (\mathsf{RID}, \mathsf{res}, \mathsf{GID}))$ to the $\mathcal{A}$ and do $\mathsf{buffer} = \mathsf{buffer}||\mathsf{tx}$.

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

Request : Upon receiving $(\mathtt{REQ}, (\mathsf{RID}, [\mathsf{design}_j]_{j \in \mathsf{bidders}}, \mathsf{GID}, \mathsf{bidders}, \mathsf{contract}))$ 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 $(\mathsf{tx} = \mathtt{REQ}, \mathsf{RID}, \mathsf{GID}, \mathsf{bidders})$ to $\mathcal{A}$ and do $\mathsf{buffer} = \mathsf{buffer}||\mathsf{tx}$.
4. For each $P_j \in \mathsf{bidders}$, send $(\mathtt{REQ}, \mathsf{RID}, \mathsf{design}_j, \mathsf{contract})$.

Bidding : Upon receiving $(\mathtt{BID}$, $(\mathsf{RID}, \mathsf{bid}_j))$ from P_j
1. Check $\mathsf{canServe}(\mathsf{RID}, P_j, \mathsf{state}$, $\mathsf{LOCKS}) = 1$ If yes,
2. Send $(\mathsf{tx} = \mathtt{BID}, (\mathsf{RID}, P_j))$ to $\mathcal{A}$ and $(\mathtt{BID}, (\mathsf{RID}, P_j, \mathsf{bid}_j))$ to P_i. Send $\mathtt{TIME}$ 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 $(\mathtt{WINNER}, \mathsf{RID}, \mathsf{GID}, P^*)$ 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 $\mathtt{TIME}$ to $\mathcal{G}_{\mathsf{refClock}}$ to receive $\mathsf{currTime}$. Set $\mathsf{TIMER}[\mathsf{RID}] = \mathsf{currTime}$.
3. Send $(\mathsf{tx} = \mathtt{WINNER}, \mathsf{GID}, \mathsf{RID}, P^*)$ to $\mathcal{A}$ and do $\mathsf{buffer} = \mathsf{buffer}||\mathsf{tx}$

Read : Upon receiving $(\mathtt{READ})$ from P_i return state to P_i
Update State : Upon receiving $(\mathtt{UPDATE}, \mathsf{tx})$ 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 $\mathtt{WINNER}$ message and no $\mathtt{RFILL}$ message and $\mathsf{currTime} - \mathsf{TIMER}[\mathsf{RID}] > \mathsf{FulfillTime}$, then delete all $(P_i, \cdot)$ from $\mathsf{LOCKS}[\mathsf{RID}]$ and $\mathsf{LOCKS}[\mathsf{RID}]$

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

Fulfillment from supplier: Upon receiving $\mathsf{tx} = (\mathtt{SFILL}, \mathsf{ContractID}, \mathsf{RID}, \mathsf{deliveryPrf})$ from P_i:
1. Send $\mathtt{TIME}$ 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 $\mathtt{SFILL}, \mathsf{ContractID}, \mathsf{RID}$ to $\mathcal{A}$ and do $\mathsf{buffer} = \mathsf{buffer}||\mathsf{tx}$

Dispute: Upon receiving $\mathsf{tx} = (\mathtt{DISPUTE}, \mathsf{ContractID}, \mathsf{GID}, pType, \mathsf{RID}, \mathsf{reason})$ from P_i:
1. Let T be set of transactions with id: $\mathsf{ContractID}$. Let P_j be the counter-party in these transactions. Let $pType \in \{$"*customer*", "*supplier*"$\}$.
2. Check that $\mathsf{Product}$ is the same one received in $\mathtt{RFILL}$. If not output 0. Check that $\mathsf{deliveryPrf}$ is the same one received in $\mathtt{SFILL}$. If not output 0.
3. Send $(\mathtt{GET\text{-}PRODUCT})$ to P_i and get back $\mathsf{Product}$ and $\mathtt{GET\text{-}DELIVERY}$ to P_j and get back $\mathsf{deliveryPrf}$
4. Run $\mathsf{PhysicalCheck}(T, pType, \mathsf{Product}, \mathsf{deliveryPrf}, \mathsf{reason}, \mathsf{currTime})$ and output winner as P_i if output 1, else output winner as P_j
5. If $pType =$ "*supplier*" and winner is P_i, then do $\mathsf{tx} = \mathsf{tx}||P_j$
6. If winner is P_i, send tx to $\mathcal{A}$ and do $\mathsf{buffer} = \mathsf{buffer}||\mathsf{tx}$

Fig. 3. An ideal functionality for private matching

- **Register**: All parties register with the functionality using this command. The party registers with a vector of roles, which can include “supplier”, “customer” or be empty.
- **Read**: Any registered party can request to read the state.
- **Update** : With this command, the adversary specifies a transaction tx to be added to the state. The functionality deletes this transaction from the buffer and then adds it to the state
- **Join Group**: The functionality is parameterized by a public list of groups - $\mathcal{G}$ where each group is identified by a GID. Parties join a group by specifying a GID. The party and the GID are then sent to the adversary.
- **Update Profile**: Each role is described by a profile. A customer ’s profile simply contains the GID it belongs to. A supplier’s profile contains the resources it possesses. A party updates its profile by adding a new group ID, or adding resources. The functionality verifies that the party had previously joined the claimed group, or that the party has the claimed resources (by running a predicate called validResource).
- **Pre-request** With this command a customer initiates a request (with id RID) for matching. It indicates the resources (res) that it requires, and the GID of the group to which this request should be associated. The functionality verifies the group membership and creates a PREREQ transaction.
- **Interest** With this command a supplier expresses interest for a request RID. The functionality checks the supplier’s capability and creates an INTRST transaction. A customer learns the suppliers that are interested fulfilling the request from state.
- **Request** With the REQ command a customer selects a set of suppliers (denoted bidders) from which it wants to receive bids. The customer also sends a list of designs such that design_j is the design for supplier P_j. The functionality sends design_j to each P_j in the bidders and but creates a REQ transaction that does not include any design.
- **Bidding** With this command a supplier P_j sends a bid for a request RID. The functionality runs a canServe predicate (Fig 18) to check if P_j can fulfill the request. If so, it creates a BID transaction with the RID and P_j. At this point, the resources of this bidder are locked.
- **Match** With this command, the customer picks a winner from the set of suppliers that bid for the request. The functionality releases the locks on the resources for the parties that are not winners.
- **Fulfilled (resp., from supplier)** With this command the customer (resp., supplier) informs the functionality that the request has been fulfilled.
- **Disputes** The supplier or the customer can raise dispute by providing a DISPUTE command along with the reason for dispute. The functionality first checks if the request is valid and then requests the product and the proof of delivery from the customer and the supplier respectively. The functionality then performs physical checks to determine the winner of the dispute and updates the state with a transaction indicating the winner of the dispute.
- **Unlock Request** The functionality is parameterized by two timeouts denoted FulfillTime and MatchTime. If a resource has been locked up for a time greater than any of these timeouts because the customer did not follow up with the next command, the functionality removes these locks by updating LOCKS.

3.2 Security Properties

The following security properties are guaranteed by $\mathcal{F}_{\mathsf{PrivateMatch}}$:

Group anonymity of customers A customer that sends a command to the functionality is anonymous within its group. This is achieved by having the functionality reveal only the group of the customer to the adversary and the state. That is in PREREQ, REQ and WINNER transactions, only the GID is added to the state. The unique identifier of the party P_i is never revealed.

Service Confidentiality Only the set of suppliers chosen by the customer can know the private details associated to the request, and no other entities. This is achieved by allowing only the bidders to receive design as shown in REQ transaction. The design is not sent to the adversary and the state.

Bid Confidentiality The bid for a certain request is known only to the customer and no other entities. This is achieved by sending the bids to the customer only as shown in BID transaction. The transaction that is sent to the adversary and added to the state only includes the identity of the bidders.

Request soundness A customer cannot add a transaction for a request unless it sent the `PREREQ` command for the request. This is achieved by the check for each command received from a party P_i for request RID, that $P_i \in \mathcal{T}[\mathsf{RID}]$.

Supplier completeness A supplier can bid using the `BID` command for a request only if its resources are available. This is achieved by locking the resources if the supplier's resources are in use for another request, and running a $\mathsf{canServe}$ predicate to verify that the resources are available.

Fairness Misbehaving parties can be caught via the `DISPUTE` command. The functionality resolves the dispute by running a physical check and verifying that the parties followed the requirements specified in the contract.

3.3 Auxiliary functionalities

We will use several building blocks such as anonymous signatures, registration authority, ledger, etc. We abstract them as ideal functionalities and we describe them below.

Clock functionality. $\mathcal{G}_{\mathsf{refClock}}$ (defined in [13]) in Fig. 4 captures a global reference clock. When queried, it provides an abstract notion of time represented by $\mathsf{currTime}$. $\mathcal{F}_{\mathsf{PrivateMatch}}$ uses this functionality as a sub-routine.

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

$\mathcal{G}_{\mathsf{refClock}}$ maintains an integer $\mathsf{currTime}$ (initialized to 0) corresponding to the reference time.
Increment Time Upon receiving an `INCREMENTTIME` request from $\mathcal{Z}$, update $\mathsf{currTime} \leftarrow \mathsf{currTime} + 1$ and send ok to $\mathcal{Z}$. Ignore `INCREMENTTIME` requests from all other entities.
Get Time Upon receiving a `TIME` command return $\mathsf{currTime}$ to the calling entity.

Fig. 4. The clock ideal functionality

Group signature functionality. $\mathcal{G}_{\mathsf{gsign}}$ is taken from [4] and is described in Fig 5. 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 only if the GM gives the approval. After joining P_j can ask the ideal functionality to generate signatures on behalf of the group. A party P_l can ask the ideal functionality to de-anonymize ("open" a certain signature, 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 $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$.

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

There are two types of players the group manager GM and parties P_i. The functionality maintains a database $\mathcal{D}$ for the list of parties in the group and a list $\mathcal{L}$ for signatures.
Non-adaptive setup : Each user P_i tells the functionality whether it is corrupted or not. Optionally, in this stage the global parameters are broadcast to all parties.
Group Setup Upon receiving ($\mathtt{gSETUP}$) from the GM, send to the adversary - ($\mathtt{gSETUP}, \mathsf{GM}$).
User Key Generation Upon receiving (GKGen) from a user P_i, send (GKGen, P_i) to the adversary.
Join Upon receiving ($\mathtt{gENROLL}, P_i$) from a party P_i, ask GM if the party can join the group. Receive $b_i \in \{0, 1\}$ from the GM and add (P_i, b_i) to $\mathcal{D}$ and send ($\mathtt{gENROLL}, b_i$) to P_i. If the group manager is corrupt, then register a user corrupt-GM.
Sign Upon receiving ($\mathtt{gSIGN}, m$) from a party P_i. Check that the entry for P_i in $\mathcal{D}$ has $b_i = 1$. If not, deny the command. If yes, send to $\mathcal{A}$ the message ($\mathtt{gSIGN}, m$). If the GM is corrupt, also send P_i to $\mathcal{A}$. Receive σ from $\mathcal{A}$. Store (P_i, m, σ) in $\mathcal{L}$ and send σ to P_i
Verify Upon receiving ($\mathtt{gVERIFY}, P_i, m, \sigma$) from a party P_i or GM. Check if (m, σ) exists in $\mathcal{L}$. If yes, return 1, else return 0.
Open Upon receiving ($\mathtt{gOPEN}, m, \sigma$) from a party P_i. Check if $\mathcal{L}$ has an entry (P_j, m, σ) for some P_j. Ask GM if it can open σ for P_i. If GM return 1 and $P_j \neq$ corrupt-GM, output P_j, else output $\perp$.
Get group members Upon receiving $\mathtt{gGET}$ from a party P_i, return the database $\mathcal{D}$ to P_i

Fig. 5. The group signature functionality

Registration functionality. $\mathcal{G}_{\mathsf{reg}}$ described in Fig 6 abstract the registration process. Command $\mathtt{REG}$ allows parties to join the system without any role, that they can later update using the $\mathtt{UPD}$ command. $\mathcal{G}_{\mathsf{reg}}$ verifies if the party is eligible for this update by evaluating predicate $\mathsf{ValidReg}$ (Fig 7), and if so it returns a certificate cert. Any party verify that a cert is valid by sending a $\mathtt{VERIFY}$ command.

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

This functionality is parameterized by a function $\mathsf{ValidReg}$(Fig 7) 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.

Fig. 6. The registration functionality

Function $\mathsf{ValidReg}(P_i, \mathsf{prof}, \mathsf{roles})$

- If $\mathsf{prof} = \mathsf{GID}$ and "$\mathsf{customer}$" $\in \mathsf{roles}$, send $\mathtt{gGET}$ to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$. 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. 7. Function to check for valid registration

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 17) the transaction and then adds the transaction to the state.

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

The functionality is parameterized by a ValidateTxn function (defined in Fig 17). The functionality maintains a global state.
Validate transactions : Upon receiving tx from a party P_i. If ValidateTxn(tx) $= 1$, do state $=$ state$||$tx. Else ignore.
Read : Upon receiving READ from a party P_i, return state.

Fig. 8. The $\mathcal{G}_{\mathsf{smartchain}}$ functionality

Non-interactive zero knowledge functionality. The NIZK functionality $\mathcal{F}_{\mathsf{nizk}}$ is an ideal functionality that allows parties to receive a proof that a particular witness w and statement x belong to an NP relation $\mathcal{R}$.

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

The non-interactive zero-knowledge functionality $\mathcal{F}_{\mathsf{nizk}}$ allows proving of statements in an NP relation $\mathcal{R}$.

```
Proving Upon receiving $(\texttt{PROVE}, x, w)$
  if $(x, w) \notin \mathcal{R}$ then
      return $\perp$
  Send $(\texttt{PROVE}, x)$ to $\mathcal{A}$ and receive $(\texttt{PROOF}, x, \pi)$
  Let $\Pi \leftarrow \Pi \cup \{(x, \pi)\}; W(x, \pi) \leftarrow w$
  return $\pi$
Verification Upon receiving $(\texttt{VERIFY}, x, \pi)$
  if $(x, \pi) \notin \Pi \wedge \pi \neq \perp$ then
      Send $(\mathsf{Vf}, x, \pi)$ to $\mathcal{A}$ and receive reply $R$
      if $R = (\texttt{WITNESS}, x, w) \wedge (x, w) \in \mathcal{R}$ then
          Let $\Pi \leftarrow \Pi \cup (x, \pi); W(x, \pi) \leftarrow w$
  return $(x, \pi) \in \Pi$
```

Fig. 9. The non-interactive zero knowledge functionality

4 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. A high-level overview of the protocol is provided below.

Protocol Overview The protocol PrivateMatch uses the following ideal functionalities : a registration functionality $\mathcal{G}_{\mathsf{reg}}$ (see Fig 6), a group signature functionality $\mathcal{G}_{\mathsf{gsign}}$ (see Fig 5) and a validation functionality $\mathcal{G}_{\mathsf{smartchain}}$ (see Fig 8), collision resistance hash functions [22] and a secure "special" private key encryption protocol [27].

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. 10. Registration and Updates

The protocol proceeds by parties creating transactions and sending them to $\mathcal{G}_{\mathsf{smartchain}}$ functionality. If a transaction is valid, the functionality adds the transaction to a global state that can be read by any party. As described earlier our protocol considers two roles for parties - the customers and the suppliers. To set notation, GID is the group identifier, reqID is an identifier for a request flow, res is used to denote resources, $\mathsf{bidders}$ is the set of bidders that are selected by the customer, k_{RID} is an encryption key for a design specification denoted design. The encrypted ciphertext is C_d. Moreover encryptions of k_{RID} and bids are denoted as C_{key} and C_{bid} repectively.

Customer: requesting and matching

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

Request If $I = (\texttt{REQ}, \mathsf{RID}, \mathsf{design}, \mathsf{bidders}, \mathsf{contract}, \mathsf{ContractID})$
1. Sample random r and compute commitment to $\mathsf{contract}$ as $C_{\mathsf{contract}} = \mathsf{Com}(\mathsf{contract}, \mathsf{ContractID}; r)$.
2. Generate design encryption key $k_{\mathsf{RID}} \leftarrow \mathsf{PrivKGen}(1^\lambda)$. Encrypt design and contract opening - $C_d \leftarrow \mathsf{Enc}(k_{\mathsf{RID}}, (\mathsf{design}, \mathsf{contract}, \mathsf{ContractID}, r))$.
3. For each $P_j \in \mathsf{bidders}$ - create $C^j_{\mathsf{key}} \leftarrow \mathsf{Enc}(\mathsf{pk}_j, k_{\mathsf{RID}})$
4. 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 $\mathsf{hash}_1 = H(\mathsf{nonce}_1 \| \mathsf{RID})$.
5. Send $(\texttt{gSIGN}, \mathsf{GID}, (\mathsf{RID}, \mathsf{pk}_{\mathsf{bid}}, \{C^j_{\mathsf{key}}\}_{j \in \mathsf{bidders}}, C_d, C_{\mathsf{contract}}, \mathsf{ContractID}, \mathsf{hash}_1, \mathsf{nonce}_0))$ to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$ and receive σ. Send $\mathsf{tx} = (\texttt{REQ}, (\mathsf{GID}, (\mathsf{RID}, \mathsf{pk}_{\mathsf{bid}}, \{C^j_{\mathsf{key}}\}_{j \in \mathsf{bidders}}, C_d, C_{\mathsf{contract}}, \mathsf{ContractID}, \mathsf{hash}_1, \mathsf{nonce}_0), \sigma))$ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive $(\texttt{ACCEPTED}, b)$

Match If $I = (\texttt{WINNER}, \mathsf{RID}, P^*)$
1. Retrieve set of encrypted bids $\{\mathsf{RID}, C^j_{\mathsf{bid}}\}_{j \in \mathsf{bidders}}$ from state
2. Decrypt each C^j_{bid} - $\mathsf{Dec}(\mathsf{sk}_{\mathsf{bid}}, C^j_{\mathsf{bid}})$ to get bid_j. 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 $(\texttt{gSIGN}, \mathsf{GID}, (\mathsf{hash}_2, \mathsf{nonce}_1, \mathsf{RID}, P^*))$ to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$ and receive σ. Send $\mathsf{tx} = (\texttt{WINNER}, (\mathsf{GID}, (\mathsf{hash}_2, \mathsf{nonce}_1, \mathsf{RID}, \mathsf{ContractID}, P^*), \sigma))$ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive $(\texttt{ACCEPTED}, b)$

Key generation
1. Run Protocol $\mathsf{KGen}_2(1^\lambda)$ with P^* to receive $(\mathsf{sk}_{\mathcal{B}}\mathsf{pk})$
2. Computes $c_{\mathcal{B}} = \mathsf{Enc}_{\mathsf{pk}_{\mathcal{M}}}(\mathsf{sk}_{\mathcal{B}})$ and sends $(\texttt{PROVE}, (c_{\mathcal{B}}, \mathsf{pk}_{\mathcal{M}}), \mathsf{sk}_{\mathcal{B}})$ to $\mathcal{F}_{\mathsf{nizk}}$ to $\mathcal{F}_{\mathsf{nizk}}$ and receives a ZKP $\pi_{\mathcal{B}}$ that proves that $c_{\mathcal{B}} = \mathsf{Enc}_{\mathsf{pk}_{\mathcal{M}}}(\mathsf{sk}_{\mathcal{B}})$.
3. Create a pre-transaction that requires both $\mathsf{sk}_{\mathcal{B}}$ and $\mathsf{sk}_{\mathcal{S}}$ to redeem it: $\mathsf{tx}_{\mathcal{B}}$ such that pays $\mathcal{S}$ the amount price and send $(\mathsf{tx}_{\mathcal{B}}, c_{\mathcal{B}}, \pi_{\mathcal{B}})$ to P^*.

Fulfilling If $I = \texttt{RFILL}$
1. Receive $(\mathsf{digProd}, \mathsf{Product}, \sigma_{P^*})$ from P^*. Check $\mathsf{ValidID}(\mathsf{Product}, \mathsf{digProd}) = 1$.
2. If $\mathsf{ValidateProduct}(\mathsf{contract}, \mathsf{Product}) = 1$, set $\mathsf{issue} = \perp$, else $\mathsf{issue} =$ "*contract*" or If $\mathsf{deadline} > \mathsf{currTime}$, set $\mathsf{issue} = (\texttt{TIME},$ "*contract*"$)$
3. Send $(\texttt{gSIGN}, \mathsf{GID}, (\mathsf{nonce}_3, \mathsf{digProd}, \sigma_{P^*}, \mathsf{issue}, \mathsf{RID}, P^*))$ to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$ and receive σ. Send $\mathsf{tx} = (\texttt{RFILL}, (\mathsf{GID}, (\mathsf{nonce}_3, \mathsf{digProd}, \sigma_{P^*}, \mathsf{ContractID}, \mathsf{issue}, \mathsf{RID}, P^*), \sigma))$ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive $(\texttt{ACCEPTED}, b)$

Dispute If $I = \texttt{DISPUTE}$
1. Send $(\texttt{DISPUTE}, \mathsf{ContractID}, \mathsf{reason}, c_{\mathcal{S}})$ to $\mathcal{M}$ and receive $(\sigma_{\mathcal{M}},$ "*customer*"$, \mathsf{sk}_{\mathcal{S}})$ if customer is the winner or receive $(\sigma_{\mathcal{M}},$ "*supplier*"$, \perp)$
2. Create $\mathsf{DProof} = (\mathsf{contract}, \mathsf{ContractID}, \sigma_{\mathcal{M}},$ "*customer*"$)$.
3. Send $(\texttt{gSIGN}, \mathsf{GID}, (\mathsf{nonce}_3, \mathsf{DProof}, \mathsf{RID}, P^*))$ to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$ and receive σ. Send $\mathsf{tx} = (\texttt{DISPUTE}, (\mathsf{GID}, (\mathsf{nonce}_3, \mathsf{DProof}, \mathsf{RID}, P^*), \sigma))$ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive $(\texttt{ACCEPTED}, b)$.

Fig. 11. Customer protocols

Registration and profile updates. Before participating in the protocol, parties must register with the system. To do so, a party first invokes the $\mathcal{G}_{\mathsf{reg}}$ functionality and receives 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. The $\mathcal{G}_{\mathsf{smartchain}}$ functionality checks if the message is valid (by running a $\mathsf{ValidateTxn}$ function) and if so, updates the state. If the message is invalid it ignores it. Hereafter we don't repeat the $\mathcal{G}_{\mathsf{smartchain}}$ operations since it is the same for all transactions.

Once registered a party updates its profile as a customer or supplier. When updating as a customer, a party must indicate which customer's group it wants to be associated with. We assume that groups are determined when the system is bootstrapped. Since each group will be associated to a group signature, we identify a group by $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$. To join a group, the party sends $\texttt{gENROLL}$ command to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$. Once joined the party interacts with $\mathcal{G}_{\mathsf{reg}}$ to get a certificate that its profile has been updated. When updating as a supplier interacts with the $\mathcal{G}_{\mathsf{reg}}$ functionality to update its profile with the resources it possesses and get a certificate.

Supplier: interest, bid, fulfill and dispute

Interest If $I = (\texttt{INTRST}, \mathsf{RID})$ **Interest** If $I = (\texttt{INTRST}, \mathsf{RID})$
1. Read $\texttt{PREREQ}$ 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 $(\texttt{INTRST}, ((\mathsf{RID}, \mathsf{ContractID}, \mathsf{pk}_i), \sigma))$ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive $(\texttt{ACCEPTED}, b)$

Bid If $I = (\texttt{BID}, \mathsf{bid})$
1. Parse $(\texttt{REQ}, \mathsf{RID}, \cdot)$ from state to get $(\mathsf{RID}, \mathsf{pk}_{\mathsf{bid}}, \{C^j_{\mathsf{key}}\}_{j \in \mathsf{bidders}}, C_d, C_{\mathsf{contract}}, \mathsf{hash}_1, \mathsf{nonce}_0)$
2. Ignore if $i \notin \mathsf{bidders}$
3. Else compute $k^*_{\mathsf{RID}} = \mathsf{Dec}(\mathsf{sk}_i, C^i_{\mathsf{key}})$ and compute $(\mathsf{design}^*, \mathsf{contract}^*, r^*) = \mathsf{Dec}(k_{\mathsf{RID}}, C_d)$
4. Check that $C_{\mathsf{contract}} = \mathsf{Com}(\mathsf{contract}^*; r)$. If not, abort.
5. Encrypt bid as $C_{\mathsf{bid}} = \mathsf{Enc}(\mathsf{pk}_{\mathsf{bid}}, \mathsf{bid})$ and commit $\mathsf{CM}_{\mathsf{bid}} = \mathsf{Com}(\mathsf{bid}, r)$, where $r \in \{0,1\}^\lambda$
6. Send $(\texttt{BID}, \mathsf{Sig}_{\mathsf{sk}_i}((\mathsf{RID}, \mathsf{ContractID}, \mathsf{CM}_{\mathsf{bid}}, C_{\mathsf{bid}})))$ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive $(\texttt{ACCEPTED}, b)$

Key generation
1. Run Protocol $\mathsf{KGen}_2(1^\lambda)$ with the customer P to receive $(\mathsf{sk}_{\mathcal{S}}, \mathsf{pk})$
2. Computes $c_{\mathcal{S}} = \mathsf{Enc}_{\mathsf{pk}_{\mathcal{M}}}(\mathsf{sk}_{\mathcal{S}})$ and sends $(\texttt{PROVE}, (c_{\mathcal{S}}, \mathsf{pk}_{\mathcal{M}}), \mathsf{sk}_{\mathcal{S}})$ to $\mathcal{F}_{\mathsf{nizk}}$ to $\mathcal{F}_{\mathsf{nizk}}$ and receives a ZKP $\pi_{\mathcal{B}}$ that proves that $c_{\mathcal{S}} = \mathsf{Enc}_{\mathsf{pk}_{\mathcal{M}}}(\mathsf{sk}_{\mathcal{S}})$.
3. Create a pre-transaction that requires both $\mathsf{sk}_{\mathcal{B}}$ and $\mathsf{sk}_{\mathcal{S}}$ to redeem it: $\mathsf{tx}_{\mathcal{B}}$ such that pays $\mathcal{S}$ the amount price and send $(\mathsf{tx}_{\mathcal{B}}, c_{\mathcal{B}}, \pi_{\mathcal{B}})$ to P.

Fulfill If $I = \texttt{SFILL}$
1. Compute unique identifier of product - $\mathsf{digProd}$
2. Compute $C_{\mathsf{deliveryPrf}} = \mathsf{Com}(\mathsf{deliveryPrf}; r)$
3. Create $\sigma = \mathsf{Sig}_{\mathsf{sk}_i}(\texttt{SFILL}, C_{\mathsf{deliveryPrf}}, \mathsf{RID}, \mathsf{digProd})$.
4. Send $(\texttt{SFILL}, (\mathsf{digProd}, \mathsf{RID}, C_{\mathsf{deliveryPrf}}, \mathsf{ContractID}, \sigma))$ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive $(\texttt{ACCEPTED}, b)$

Dispute If $I = \texttt{DISPUTE}$
1. Send $(\texttt{DISPUTE}, \mathsf{ContractID}, \mathsf{reason}, c_{\mathcal{B}})$ to $\mathcal{M}$ and receive $(\sigma_{\mathcal{M}}, \text{"}\textit{supplier}\text{"}, \mathsf{sk}_{\mathcal{B}})$ if P^* is the winner or receive $(\sigma_{\mathcal{M}}, \text{"}\textit{customer}\text{"})$
2. Compute $\sigma = \mathsf{Sig}(\mathsf{sk}_i, (\text{"}\textit{dispute}\text{"}, \mathsf{ContractID}))$. Send $\mathsf{tx} = (\texttt{DISPUTE}, (\text{"}\textit{dispute}\text{"}, \mathsf{ContractID}, \mathsf{RID}, \sigma)$ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive $(\texttt{ACCEPTED}, b)$

Fig. 12. Supplier protocols

Request for service. A customer who wishes to request resources to implement a private design, that we denote by design, put forth a request in two steps. First, the customer prepares an anonymous $\texttt{PREREQ}$ transaction which only includes the resources it would require for the service (without the design), and the id RID of the request. This transaction is signed using a group signature, and will be associated to the GID of the group the customer belongs to. Suppliers who are interested in fulfilling this request, send an $\texttt{INTRST}$ transaction.

The customer then picks a set of suppliers from the interested set of suppliers and creates the $\texttt{REQ}$ transaction, referring to request RID, such that only the chosen set of suppliers can see the private input design. To this end, the design must be encrypted. We implement this efficiently, by having the customer encrypts the design with a fresh key k_{RID} and (anonymously) post the encrypted design on a web location that it controls (this way, it can remove it when necessary). Next, in the $\texttt{REQ}$ transaction, indexed by the request ID RID, the customer will add encryptions of the key k_{RID} under the public key of each of the selected suppliers. In the $\texttt{REQ}$ transaction, the customer also adds a freshly sampled public key pk_{RID}. This will be used by the suppliers in the next stage to encrypt their private bids.

As explained in Sec. 1 we chain transactions for the same RID using puzzles. Hence, every transaction from the customer for a specific RID contains the output y of a collision-resistant hash function (CRHF), and any follow up transaction must contain the pre-image of y.

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_{RID}, and create BID transaction contained the encrypted bid. 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 with the identity of the winner. Once confirmed, the resources of the suppliers that were not selected as winner are unlocked.

Mediator protocol

The customer is denoted as $\mathcal{B}$ and the supplier as $\mathcal{S}$.
Key Generation: Run $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Enc.KGen}(1^\lambda)$ to get encryption keys and announce pk
Dispute resolution: Upon receiving $(\texttt{DISPUTE}, \mathsf{ContractID}, \mathsf{reason}, pType)$ from some party P:

1. Send READ to $\mathcal{G}_{\mathsf{smartchain}}$ and receive state
2. Let $\mathcal{T}$ be the set of transactions with id $\mathsf{ContractID}$
3. Send $(\texttt{GET-DELIVERY}, \mathsf{ContractID})$ to $\mathcal{S}$ and receive $d = (\mathsf{deliveryPrf}, r_S)$. Send $(\texttt{GET-CONTRACT}, \mathsf{ContractID})$ to $\mathcal{B}$ and receive $c = (\mathsf{contract}, r_B)$
4. Run $b \leftarrow \mathsf{transactionCheck}(\mathcal{T}, pType, c, d)$. If $b = 1$, send $(\texttt{GET-PRODUCT}, \mathsf{ContractID})$ to $\mathcal{B}$ and $\mathcal{S}$ and receive $\mathsf{Product}_{\mathcal{B}}$ and $\mathsf{deliveryPrf}$ respectively.
5. Send TIME to $\mathcal{G}_{\mathsf{refClock}}$ and receive back $\mathsf{currTime}$.
6. Run $b_1 = \mathsf{PhysicalCheck}(\mathcal{T}, \mathcal{B}, \mathsf{Product}_{\mathcal{B}}, \mathsf{deliveryPrf}, \mathsf{reason}, \mathsf{currTime})$.
7. Let $P' = \{\mathcal{B}, \mathcal{S}\} \setminus P$ If $b_1 = 1$, compute $\mathsf{sk}_{P'} = \mathsf{Dec}(\mathsf{sk}_{\mathcal{M}}, c_{P'})$, else $\mathsf{sk}_{P'} = \perp$
8. Compute $\sigma_{\mathcal{M}} = \mathsf{Sign}_{\mathcal{M}}(W)$, where $W \in \{$"*customer*", "*supplier*"$\}$
9. Send $(\sigma_{\mathcal{M}}, W, \mathsf{sk}_{P'})$ to P

Fig. 13. Mediator protocol for dispute resolution

Dispute and fulfillment of requests Our idea to resolve disputes is inspired by the encrypt-and-swap escrow protocol presented in [18]. The protocol has the following entities: a buyer $\mathcal{B}$, a seller $\mathcal{S}$ and a (trusted) mediator $\mathcal{M}$. The main idea of this protocol is the following:

1. The buyer and seller agree on a transaction $\mathsf{tx}_{\mathcal{B}}$ that pays the seller some amount of its funds when the buyer receives the product from the seller. They also agree on a refund transaction $\mathsf{tx}_{\mathcal{S}}$ in the case that the buyer is not satisfied with this product. Crucially, the secret keys of both the buyer and the supplier are required to redeem these transactions. This can be implemented using the 2-of-2 threshold signing protocol of Gennaro et. al. [16]. We do not specify how these transactions will be implemented, but one could use the Bitcoin blockchain ($\mathcal{G}_{\mathsf{ledger}}$ [5]) to make these payments. The buyer first submits a transaction tx that sends the amount bid to the 2-of-2 address. Note that if the transaction $\mathsf{tx}_{\mathcal{B}}$ does not pay the amount bid to the supplier , then it simply does not proceed with the production.
2. Each party encrypts its secret key under the public key of the mediator and sends the ciphertext to the other party.
3. In the case of a dispute, the party submits the reason of dispute along with the ciphertext it received from the counter-party. The mediator determines the winner of the dispute by running a $\mathsf{transactionCheck}$ (that checks that all the transactions submitted on the blockchain with id $\mathsf{ContractID}$ are computed correctly) and $\mathsf{PhysicalCheck}$ (a physical check to determine if the product was correctly created with respect to the contract). The mediator then decrypts the ciphertext of the counter-party and announces the secret key to the winner. Now the winner has both the secret keys and can redeem its transaction to claim the funds.

Function $\mathsf{transactionCheck}(\mathcal{T}, pType, c, d)$

```
1. Parse $\mathcal{T}$ as $(\mathtt{REQ}, \mathtt{RFILL}, \mathtt{SFILL})$
2. Parse $(\mathsf{contract}, r_B) \leftarrow c$
3. Parse $(\mathsf{deliveryPrf}, r_S) \leftarrow d$
4. If no RFILL, and $pType =$ "customer", output 0.
5. If no SFILL and $pType =$ "supplier", output 0.
6. **Contract check, if** $pType =$ "customer"
   (a) If $\mathsf{issue} =$ "contract" in RFILL check that $\mathsf{Com}(\mathsf{contract}, rB) = C_{\mathsf{contract}}$ as posted with REQ.
   (b) Else if $\mathsf{issue} = \mathtt{TIME}$ check that $\mathsf{currTime} > \mathsf{contract}[time] + \mathsf{matchtime}$ and no SFILL is in $\mathcal{T}$
   (c) If either of the two are false, output 0.
7. **Timeout check, if** $pType =$ "supplier"
   (a) Check $\mathsf{Com}(\mathsf{deliveryPrf}, r_S) = C_{\mathsf{deliveryPrf}}$ as posted with SFILL. If not return 0.
   (b) If no RFILL and if $\mathsf{currTime} < \mathsf{contract}[time] + \mathsf{matchtime}$, return 0.
8. Return 1
```

Fig. 14. Checking consistency of transactions

$\mathsf{PhysicalCheck}(T, pType, \mathsf{Product}, \mathsf{deliveryPrf}, \mathsf{reason})$

These are non-cryptographic checks that the mediator needs to do

1. **Physical check of product** Check that $\mathsf{productCheck}(\mathsf{Product}, \mathsf{reason}, \mathsf{contract}) = 1$
2. **Payment check** If $\mathsf{reason} = (\text{"}payment\text{"}, \mathsf{sk}', c')$, the mediator checks if $\mathsf{Dec}_{\mathsf{sk}_M}(c') = \mathsf{sk}'$ and if that is the case output 1, else output 0
3. **Wrong payment** If $\mathsf{reason} = (\text{"}wrong\text{-}bid\text{"}, \mathsf{CM}_{\mathsf{bid}}, \mathsf{bid}, r, \mathsf{tx})$, the mediator checks if the $\mathsf{CM}_{\mathsf{bid}} = \mathsf{Com}(\mathsf{bid}, r)$ and if the transaction in $\mathcal{G}_{\mathsf{ledger}}$ corresponds to bid. If yes output 1, else output 0.
4. **Delivery check** Check that the $\mathsf{deliveryPrf}$ is valid, and is within the date specified in contract.

Fig. 15. Winner checking function

Before we present our proofs we present the definition of the function $\mathsf{ValidateTxn}$. This is the function run by $\mathcal{G}_{\mathsf{smartchain}}$ to ensure that each transaction it receives is valid.

Predicate $\mathsf{VrfSame}$

This function takes as input a message which contains solution to a puzzle nonce_i and the RID and the state of the chain state_b. Let hash_i be the puzzle that was posted to the state.
If $\mathsf{hash}_i = H(\mathsf{nonce}_i)$, output 1, else output 0.

Fig. 16. Check if two tx are from the same requester

Function $\mathsf{ValidateTxn}$

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 $\mathsf{Vrf}(\mathsf{cert}, P_i, \mathsf{roles})$ to $\mathcal{G}_{\mathsf{reg}}$. Output the bit returned.
Validate pre-request If $\mathsf{tx} = (\mathtt{PREREQ}, m, \sigma)$ 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, 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
Validate requester dispute : If $\mathsf{tx} = (\mathtt{DISPUTE}, (\mathsf{GID}, (\mathsf{nonce}_3, \mathsf{DProof}, \mathsf{RID}, P^*), \sigma))$
1. send $(\mathtt{gVERIFY}, (\mathsf{nonce}_3, \mathsf{DProof}, \mathsf{RID}, P^*), \sigma)$ to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$.
2. If 1 returned, check $\mathsf{VrfSame}(m, \mathsf{state}) = 1$. If not, return 0.
3. Parse DProof to get $\mathsf{contract}$ and $\mathsf{ContractID}$
4. Compute $b = \mathsf{Sig.Vrf}_{\mathsf{vk}_{\mathcal{M}}}(\mathsf{DProof})$
5. Return b.

Validate supplier dispute: If $\mathsf{tx} = (\mathtt{DISPUTE}, (\text{“}\textit{dispute}\text{”}, \mathsf{ContractID}, \mathsf{RID}, \sigma)$
1. Parse DProof to get $\mathsf{contract}$ and $\mathsf{ContractID}$
2. Compute $b = \mathsf{Sig.Vrf}_{\mathsf{vk}_{\mathcal{M}}}(\mathsf{DProof})$
3. If $b = 1$, send $(\mathtt{gOPEN}, m, \sigma)$ where (m, σ) correspond to any previous message with id RID abd receive P
4. Update $\mathsf{tx} = \mathsf{tx} \| P$
5. Return b.

Fig. 17. The validation function

Predicate $\mathsf{canServe}$

The predicate takes as inputs a supplier P_j, state, a request identifier RID and the list LOCKS.
1. Get $\mathcal{P}$ from state. And get res from the $\mathtt{PREREQ}$ message for RID. Check that $P_j \in \mathcal{P}$. If not, return 0.
2. Let $\mathsf{AllResource}_j = \mathsf{res}_j$ s.t. $(\mathtt{UPD}, (P_j, \mathsf{res}_j, [\mathsf{roles}], \mathsf{cert})) \in \mathsf{state}$ is the latest $\mathtt{UPD}$ transaction for P_j in state
3. Let $\mathsf{Locked}_j = \bigcup \mathsf{res}_j$ s.t. $(P_j, \cdot, \mathsf{res}_j)$ in LOCKS.
4. If res not in $\mathsf{AllResource}_j \setminus \mathsf{Locked}_j$, return 0, where $(P_j, \mathsf{AllResource}) \in \mathcal{P}$. Else return 1.

Fig. 18. Predicate to check if an interested supplier can service a request.

4.1 Security Proof

We present an overview of the proofs below. The formal proofs are presented in Appendix A

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 [22], secure “special” symmetric key encryption [27], EUF-CMA signature*

[22], secure commitment schemes [22] and CPA-secure encryption [22] in the presence of a PPT adversary that corrupts a subset of the customers .

Overview of the simulator 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. In this case, we assume that the adversary corrupts a set of requesters only.

The REG, UPD and gJOIN commands are simulated by internally simulating the $\mathcal{G}_{\mathsf{gsign}}, \mathcal{G}_{\mathsf{reg}}$ and $\mathcal{G}_{\mathsf{smartchain}}$ functionalities.

For a PREREQ command, the simulator only receives the GID and not the identity of the party calling the PREREQ command. The simulator samples a random nonce_0 and computes $H(\mathsf{nonce}_0 \| \mathsf{RID}) = \mathsf{hash}_0$. And then the simulator simulates the $\mathcal{G}_{\mathsf{gsign}}$ functionality and records the message-signature pair without the identity of the party. Since the simulator is able to simulate a transaction that is indistinguishable from the real-world with only the GID we claim that the party P_i is **anonymous within the group** GID. For the REQ command, the simulator needs to simulate the REQ transaction as in the real-world without knowing the design. To this end, the simulator encrypts to 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**. In the case that $\mathcal{A}$ sends the REQ transaction, the simulator aborts if it is able to create a transaction that corresponds to the RID of an honest user. Since we use CRHF, the probability of this happening is negligible. Thus we guarantee **requester soundness**. For the BID command the simulator needs to simulate a bid transaction that is indistinguishable from the real-world transaction without knowing the bid value. To this end the simulator encrypts to the 0 string. By the CPA security of the encryption scheme the simulated encryption and the real-world encryptions are indistinguishable. This guarantees the property of **bid confidentiality** For a DISPUTE transaction, the simulator receives from the $\mathcal{F}_{\mathsf{PrivateMatch}}$ functionality the reason and the ContractID. This information is enough to simulate the DISPUTE command in the real world. Note that since we assume that the mediator is a trusted party, the simulator simulates this mediator towards the adversary. 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. By the CPA security of the encryption scheme, the adversary cannot learn the secret key of the supplier. Upon receiving a dispute message on behalf of the mediator, the simulator creates a DISPUTE command and forwards it to the $\mathcal{F}_{\mathsf{PrivateMatch}}$ functionality. Upon receiving a GET-PRODUCT command from the functionality the simulator on behalf of the mediator sends the adversary a GET-PRODUCT command and receives back the Product. The simulator sends the Product to the $\mathcal{F}_{\mathsf{PrivateMatch}}$ functionality and creates a signature on the winner determined by the functionality. This guarantees the properties of **fairness** in our protocols.

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 [22], EUF-CMA signature [22], secure commitment schemes [22] and CPA-secure encryption [22] in the presence of a PPT adversary that corrupts a subset of the suppliers.*

Overview of the simulator 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 checks that the signature can be verified. If it corresponds to that of an honest party, then the simulator aborts with UnforgeabilityError. 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 1. In our protocols we do not specify how the parties are punished if found guilty of misbehaving. Here we discuss one potential punishment mechanism. Upon registration, each party creates a collateral transaction that sends a fixed amount to an address that is owned by the set of validators. That is, to spend this transaction, a threshold number of validators will need to collaborate. In case of misbehavior, each of the validators sign a transaction that spends the collateral amount associated with the misbehaving party to a new address. The new address is such that no party in the system knows the secret key to this address and can therefore spend the funds associated to this address. Such an address is called a burn addresses [21]. Using collateral as a punishment mechanism is well known in the literature and is used in Ethereum to punish validators [10]. Identifying a misbehaving supplier is straightforward, since the validators can simply observe the transactions on the blockchain. On the other hand, a misbehaving customer will first be de-anonymized via the `gOPEN` command to $\mathcal{G}_{\mathsf{gsign}}$ functionality. Once the party is de-anonymized its collateral is then burned by the mechanism described above.

4.2 Implementing auxiliary functionalities

We present intuition on how to realize the functionalities - $\mathcal{G}_{\mathsf{reg}}$, $\mathcal{G}_{\mathsf{gsign}}$ and $\mathcal{G}_{\mathsf{smartchain}}$.

Registration functionality $\mathcal{G}_{\mathsf{reg}}$. The $\mathcal{G}_{\mathsf{reg}}$ functionality will be implemented by a registration authority which need to verify that a registering party has valid credentials. If verified the authority sends a certificate that allows other parties to verify that the party is registered. This verification can be done with existing systems like CanDID [29]. CanDID leverages decentralized oracles like DECO [48] or Town Crier [47] that allows parties to port credentials from legacy systems. For e.g. a registering party can use the profile page of their Social Security Administration (SSA) account to generate a credential attesting to their Social Security Number (SSN). CanDID is run by a committee that together store a secret key sk_C which is used to issue credentials to users. The corresponding pk_C can be used by the registration authority to verify the credentials. When a party registers as a supplier the registration authority evaluates the $\mathsf{validResource}$ predicate which determines if the supplier has the resources it claims to own. This can be implemented via auditing authorities that sign a message stating the exact resources owned by the suppliers. The supplier then sends these signatures and list of resources it owns to the registration authority. The registration authority simply verifies if there exist some threshold number of signature that certify this list of resources.

Group signing functionality $\mathcal{G}_{\mathsf{gsign}}$. We use the $\mathcal{G}_{\mathsf{gsign}}$ functionality as defined in the work by Ateniese et al. [4] and the protocol realizing this functionality is presented in Section 5 of [4]. `gJOIN`: The group manager gives the user a re-randomizable signature on the user's public key. `gSIGN` is done by attaching the re-randomized signature, a re-randomized public key, and a key-private signature on the message. For the `gOPEN` protocol, the group manager uses a tracing value (that they would have received in the `gJOIN` protocol per user), to identify the party who signed a particular message. Note that this single GM goes against the spirit of a decentralized setting. In settings where a single GM is undesirable one can replace the GM with multiple managers that enable threshold group signatures. [3] A naive idea would be to share the tracing value among all the group managers. To de-anonymize, the group managers will have to collaborate to reconstruct the tracing value. There exist other implementations of threshold group signatures. For instance, [7] present a fair traceable group signature scheme that introduces new entities called fairness authorities that are responsible for the opening and revealing procedures. The main intuition is that during the `gJOIN` protocol 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 to determine the identity of the signer. Similarly [9, 17] present protocols for distributed tracing using tag-based encryption to open the signatures of parties.

Clock functionality $\mathcal{G}_{\mathsf{refClock}}$. The $\mathcal{G}_{\mathsf{refClock}}$ functionality provides an exact clock. This functionality presents a global reference clock which is monotonic and increasing. This clock functionality is presented in Section 3.1 of [13].

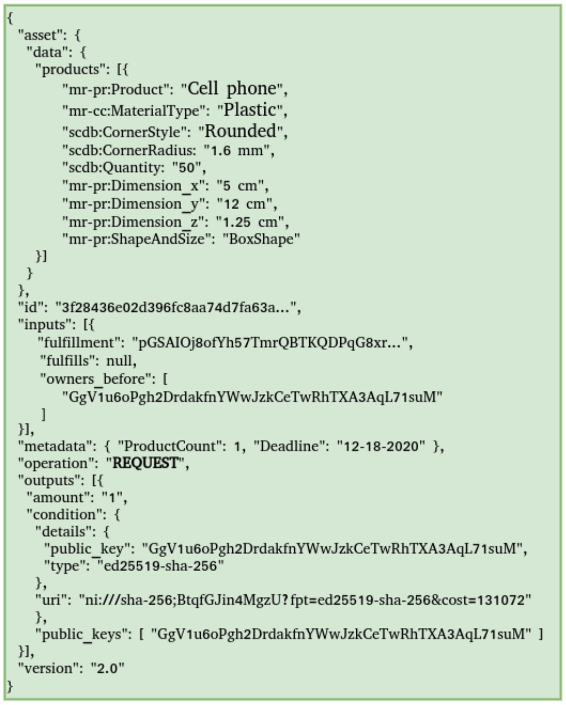
[3] In threshold group signatures, a signature can be de-anonymized only if a threshold of managers all agree to perform de-anonymization

SmartChainDB functionality $\mathcal{G}_{\mathsf{smartchain}}$. The ideal functionality $\mathcal{G}_{\mathsf{smartchain}}$ consists of two parts: the validation part and the consensus part. The validators first collect transactions, validate them and then run the consensus protocol to agree on a set of validated messages. The consenus algorithm may be implemented by a consensus algorithm such as Tendermint [26], and the validation rules for each transaction are determined by the cases described in Figure 17.

5 Implementation

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

In traditional blockchain environments such as Ethereum, Hyperledger Fabric, *"Smart Contracts"* are used to implement general business logic. Intuitively, a smart contract parallels a real-world contract process between parties that are assumed to at least be in some communication about the details of the contract. However, because smart contracts are owned by a single entity, each customer/requestor 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/provider 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 any 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.



```
{
  "asset": {
   "data": {
    "products": [{
        "mr-pr:Product": "Cell phone",
        "mr-cc:MaterialType": "Plastic",
        "scdb:CornerStyle": "Rounded",
        "scdb:CornerRadius: "1.6 mm",
        "scdb:Quantity: "50",
        "mr-pr:Dimension_x": "5 cm",
        "mr-pr:Dimension_y": "12 cm",
        "mr-pr:Dimension_z": "1.25 cm",
        "mr-pr:ShapeAndSize": "BoxShape"
    }]
   }
  },
  "id": "3f28436e02d396fc8aa74d7fa63a...",
  "inputs": [{
     "fulfillment": "pGSAIOj8ofYh57TmrQBTKQDPqG8xr...",
     "fulfills": null,
     "owners_before": [
        "GgV1u6oPgh2DrdakfnYWwJzkCeTwRhTXA3AqL71suM"
     ]
  }],
  "metadata": { "ProductCount": 1, "Deadline": "12-18-2020" },
  "operation": "REQUEST",
  "outputs": [{
   "amount": "1",
   "condition": {
    "details": {
     "public_key": "GgV1u6oPgh2DrdakfnYWwJzkCeTwRhTXA3AqL71suM",
     "type": "ed25519-sha-256"
    },
    "uri": "ni:///sha-256;BtqfGJin4MgzU?fpt=ed25519-sha-256&cost=131072"
    },
    "public_keys": [ "GgV1u6oPgh2DrdakfnYWwJzkCeTwRhTXA3AqL71suM" ]
  }],
  "version": "2.0"
}
```

Fig. 19. A Sample Request Transaction for the *supply of 50 plastic cellphone covers.*

Consequently, our implementation approach focused on introducing new blockchain transaction types into an open-source blockchain platform that was amenable to the desired extensions that we sought. BigchainDB [30] 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 [26] (for consensus), the BigchainDB Server (for syntactic and semantic validation of transactions), and a local MongoDB [3] database (for blockchain storage) on every validator node. BigChainDb also has a rich permissioning model allowing for the implementation of both public and private blockchains.

Extending BigChainDB's Transaction Model. A core functionality that is supported by BigChainDB out-of-the-box is the ability to transfer assets (including validation checks e.g. double spend). BigChainDB's transaction model is a "declarative", attribute/key-value model, where transaction structure is defined by a schema and therefore is naturally extensible. We refer to our extension of BigChainDB as SmartChainDB.

For each new transaction introduced by SmartChainDB, a schema is introduced (semantic technologies like ontologies are also used to support terminological expansions and variations). Figure 19 shows an example of one SmartChainDB transaction `REQUEST`.

SmartChainDB extends the validator algorithms in BigChainDB according to the ValidateTxn. For the `BID` transaction where validation involves "locking" of resources, it is implemented as a transfer of resources to an "escrow" account (a designated non-user account used for holding resources). As part of the validation of the match transaction - `WINNER`, when resources are to be released, the validator initiates transactions back to the non-winners. The key components of the SmartChainDB architecture and the lifecycle of a transaction is shown in Figure 20. The components in purple indicate either newly introduced software components or the BigChainDB software components that were modified.)

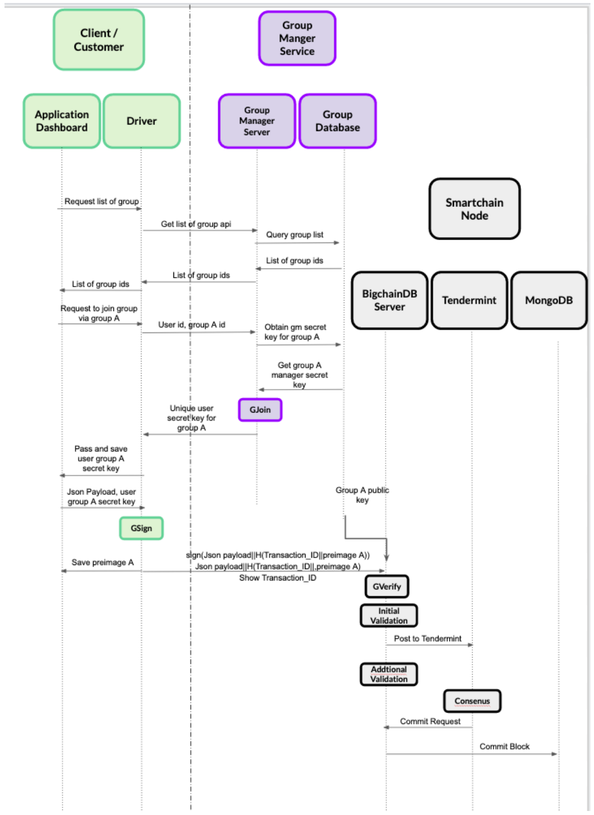


Fig. 20. Overview of SmartchainDB

Extending BigChainDB's Privacy Model. Each transaction in BigChainDB includes *Conditions* specified on a transaction's output that must be satisfied. These *Conditions* are specified using the cryptoconditions specification [1]. In general these are signatures that must be verified. The vanilla BigChainDB implementation uses the Ed25519 public-key signature.

	Mean	Median	Std. Dev
`gSIGN` ([8] + [35])	7.3	6.6	4.4
BigchainDB Sign	1.2	0.9	1.1
`gVERIFY` ([8] + [35])	8.9	7.1	5.4
BigchainDB Verify	0.9	0.7	0.7

Table 1. Comparing group signatures and BigchainDB signatures

New conditions can be composed using boolean operators to create more complex conditional logical circuits. BigChainDB includes a reference implementation [2] of cryptoconditions used to check the fulfilment of conditions.

Group Signature Implementation and Integration. We implemented the group signature scheme due to Bichsel and Camenisch [8] and integrated it into the set of possible signature schemes available in SmartChainDB as a cryptocondition. The core building block of this scheme is a re-randomizable signature scheme. They use CL signature [11] in their protocol. We replace this with re-randomizable signatures of Pointcheval & Sanders (PS) [35] as described in [34]. To implement the group signature we use (i.) the Hyperledger Ursa library [19] that provides an implementation for the PS signatures. We rely on the same library for digital signature signing and proof of knowledge functions that are the necessary backbones of `gSIGN` and `gVERIFY` in [8]. Our implementation is in Rust and used python-based Cherrypy server as a wrapper to call the Rust cryptographic functions as a service.

In Tendermint, there are multiple phases to processing a transaction, which is interpreted as *mode*:

- *broadcast_tx_async:* No-processing. It will return right away without waiting to hear if the transaction is even valid. If no mode is specified, broadcast_tx_async mode is used.
- *broadcast_tx_sync:* Processing-through-the-mempool. It will return with the result of running the transaction through CheckTx.
- *broadcast_tx_commit:* Processing-through-a-block. It will wait until the transaction is committed in a block or until some timeout is reached, however, it will return immediately if the transaction does not pass CheckTx.

6 Evaluation

The objective of our evaluation of SmartChainDB was twofold: (i.) verify that the newly introduced transaction types can support simulated marketplace workloads under the reasonable performance bounds and (ii.) that the overhead of the group signature implementation was not a limitation when compared with the existing signature scheme

Experimental Setup Although BigChainDB has a publicly-available test network, this could not be used for our experiments because the vanilla platform would be unable to process our transactions. Therefore, a private network was set up 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 – `REG`, `PREREQ`, `INTRST`, `REQ`, `BID`, and `WINNER`.

We developed a workload generator, packaged inside the driver, to produce random transaction sequences and test the blockchain setup. Each driver produces 20-25 transactions per second, i.e. 80-100 transactions per second in total(due to 4 driver instances), and sends them to the available validator nodes for the commit process. Driver instances act both as customer and supplier and create 2000 transactions for each transaction type during the experiment run, i.e. 8000 total transactions per transaction type. For the `CREATE` and `TRANSFER` transactions are evaluated under the same workload. The resource attributes within the transaction payload are randomly populated using the ManuService [28] ontology. The experimental results are described in the next subsections in detail.

Latency Overhead We compare the `CREATE` and `TRANSFER` transaction that are available in the vanilla BigchainDB implementation with the transactions used in our protocol.

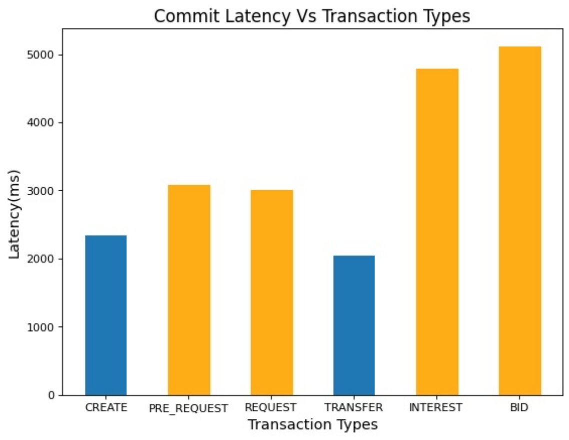


Fig. 21. Performance Comparison of Transactions

	Mean	Median	Std. Dev
`gSIGN` ([8] + [35])	9.77	9.65	0.92
BigchainDB Sign	0.75	0.52	0.84
`gVERIFY` ([8] + [35])	11.13	11.03	0.30
BigchainDB Verify	1.00	0.76	0.64

Table 2. Comparing group signatures and BigchainDB signatures (ms)

We measure the commit latency (time between a validator node receiving a transaction and its commit into the blockchain. We compare the `PREREQ` and `REQ` with `CREATE` transactions because those transactions are semantically closest but the former more complex since they require additional validations – valid `PREREQ` input transaction in the case of `REQ`, ensuring that the `REQ` initiator is same as the `PREREQ`, etc. Similarly, the vanilla `TRANSFER` transaction has some similarity to `INTRST` and `BID` transaction. However, `INTRST` adds a check to verify that the supplier has declared asset with capabilities required to service the request and *sbid* goes beyond this to actually hold a suppliers resources (an embedded `TRANSFER` transaction) in an escrow to limit supplier overbidding.

Figure 21 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. We observe a similar pattern when we compare `PREREQ` and `REQ` with `CREATE` results, i.e. the request transactions require a few more seconds to carry out protocol-specific validations. Likewise, `INTRST` and `bid` take longer to commit than the vanilla `TRANSFER`. 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. **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 $10\times$ that of regular signatures. We attribute the high costs of group signing and verifying to the computation of Tate pairings done in our implementations.

References

1. Crypto-conditions - draft-thomas-crypto-conditions-03. https://tools.ietf.org/html/draft-thomas-crypto-conditions-03
2. Cryptoconditions reference implementations. https://github.com/rfcs/crypto-conditions#implementations
3. Mongodb, the most popular database for modern apps. https://www.mongodb.com/

4. Ateniese, G., Camenisch, J., Hohenberger, S., De Medeiros, B.: Practical group signatures without random oracles. (2005)
5. Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: Bitcoin as a transaction ledger: A composable treatment. In: Annual international cryptology conference. pp. 324–356. Springer (2017)
6. 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)
7. 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)
8. 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)
9. 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)
10. Buterin, V.: Minimal slashing conditions (2017), https://medium.com/@VitalikButerin/minimal-slashing-conditions-20f0b500fc6c
11. Camenisch, J., Lysyanskaya, A.: A signature scheme with efficient protocols. In: International Conference on Security in Communication Networks. pp. 268–289. Springer (2002)
12. 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)
13. 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)
14. 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)
15. 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)
16. Gennaro, R., Goldfeder, S., Narayanan, A.: Threshold-optimal dsa/ecdsa signatures and an application to bitcoin wallet security. In: International Conference on Applied Cryptography and Network Security. pp. 156–174. Springer (2016)
17. 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)
18. 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)
19. Hyperledger: hyperledger/ursa. https://github.com/hyperledger/ursa/tree/master/libzmix/src/signatures/ps
20. Kabi, O.R., Franqueira, V.N.: Blockchain-based distributed marketplace. In: International Conference on Business Information Systems. pp. 197–210. Springer (2018)
21. Karantias, K., Kiayias, A., Zindros, D.: Proof-of-burn. In: International Conference on Financial Cryptography and Data Security. pp. 523–540. Springer (2020)
22. Katz, J., Lindell, Y.: Introduction to modern cryptography. CRC press (2020)
23. 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)
24. 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)
25. 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)
26. Kwon, J.: Tendermint: Consensus without mining. Draft v. 0.6, fall **1**(11) (2014)
27. Lindell, Y., Pinkas, B.: A proof of security of yao's protocol for two-party computation (2006)
28. Lu, Y., Wang, H., Xu, X.: Manuservice ontology: a product data model for service-oriented business interactions in a cloud manufacturing environment:. Journal of Intelligent Manufacturing **33**, 317–334 (01 2019). https://doi.org/10.1007/s10845-016-1250-x
29. 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)

30. 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)
31. Montecchi, M., Plangger, K., Etter, M.: It's real, trust me! establishing supply chain provenance using blockchain. Business Horizons **62**(3), 283–293 (2019)
32. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2012), 28 (2008), https://bitcointalk.org/index.php?topic=321228.0
33. 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)
34. Pointcheval, D., Sanders, O.: Short randomizable signatures https://eprint.iacr.org/2015/525.pdf
35. Pointcheval, D., Sanders, O.: Short randomizable signatures. In: Cryptographers' Track at the RSA Conference. pp. 111–126. Springer (2016)
36. 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)
37. 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)
38. 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)
39. Soska, K., Kwon, A., Christin, N., Devadas, S.: Beaver: A decentralized anonymous marketplace with secure reputation. IACR Cryptol. ePrint Arch. **2016**, 464 (2016)
40. Subramanian, H.: Decentralized blockchain-based electronic marketplaces. Communications of the ACM **61**(1), 78–84 (2017)
41. 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)
42. 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)
43. 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)
44. 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)
45. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger
46. Xiong, J., Wang, Q.: Anonymous auction protocol based on time-released encryption atop consortium blockchain. arXiv preprint arXiv:1903.03285 (2019)
47. Zhang, F., Cecchetti, E., Croman, K., Juels, A., Shi, E.: Town crier: An authenticated data feed for smart contracts. In: Proceedings of the 2016 aCM sIGSAC conference on computer and communications security. pp. 270–282 (2016)
48. Zhang, F., Maram, D., Malvai, H., Goldfeder, S., Juels, A.: Deco: Liberating web data using decentralized oracles for tls. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. pp. 1919–1938 (2020)

A Formal Proofs

A.1 The malicious requester case

Theorem 1 *(restated)* **:** *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 [22], secure "special" symmetric key encryption [27], EUF-CMA signature [22], secure commitment schemes [22] and CPA-secure encryption [22] in the presence of a PPT adversary that corrupts a subset of the customers* .

$\mathsf{UpdateState}(\mathsf{tx})$

Global state : state
1. (Simulate $\mathcal{G}_{\mathsf{smartchain}}$) Update $\mathsf{state} = \mathsf{state}||\mathsf{tx}$.
2. Let $\mathsf{tx}' = \mathsf{Idealify}(\mathsf{tx})$, where $\mathsf{Idealify}$ returns the transaction that $\mathcal{F}_{\mathsf{PrivateMatch}}$ sends to $\mathcal{S}$.
3. Send $(\mathtt{UPDATE}, \mathsf{tx}')$ to $\mathcal{F}_{\mathsf{PrivateMatch}}$.

Simulator - $\mathcal{S}_r$

Sets maintained :

$\mathcal{L}_{\mathtt{REG}}$: To keep track of the certificates that are registered.

$\mathcal{D}$: Maintain groups IDs of parties.

$\mathcal{L}$: Keep track of signatures received

Registration.
Upon receiving $(\mathtt{REG}, P_i, [\mathsf{roles}])$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$:
1. Activate P_i and if "$\mathsf{supplier}$" $\in \mathsf{roles}$: generate encryption and signature keys $(\mathsf{Sig}.pk_i, \mathsf{Sig}.sk_i), (\mathsf{Enc.pk}_i, \mathsf{Enc.sk}_i)$.
2. (Simulate $\mathcal{G}_{\mathsf{reg}}$) : Send (P_i, roles) to $\mathcal{A}$ and receive cert_i.
3. Store (P_i, cert_i) in $\mathcal{L}_{\mathtt{REG}}$.
4. Call $\mathsf{UpdateState}(\mathsf{tx})$.

Upon receiving $(\mathtt{REG}, [\mathsf{roles}])$ from a party P_i (corrupted) :
1. Send $(\mathtt{REG}, P_i, [\mathsf{roles}])$ to $\mathcal{F}_{\mathsf{PrivateMatch}}$. If $\mathtt{REG}, P_i, [\mathsf{roles}]$ not received from $\mathcal{F}_{\mathsf{PrivateMatch}}$, ignore request. Else continue.
2. (Simulate $\mathcal{G}_{\mathsf{reg}}$) Send (P_i, roles) to $\mathcal{A}$ and receive back cert_i
3. Store cert_i in $\mathcal{L}_{\mathtt{REG}}$, send $(\mathsf{cert}_i, \mathsf{roles})$ to P_i.
4. (Simulating $\mathcal{G}_{\mathsf{smartchain}}$) Upon receiving $\mathsf{tx} = (P_i, \mathsf{cert}_i)$ from P_i, check if cert_i in $\mathcal{L}_{\mathtt{REG}}$. If yes, call $\mathsf{UpdateState}$.
5. Send $(\mathtt{CORRUPTED}, 1)$ to $\mathcal{F}_{\mathsf{PrivateMatch}}$

Join Group.
Upon receiving $(\mathtt{gJOIN}, P_i, \mathsf{GID})$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$ (Simulate $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$) :
1. Send (GKGen, P_i) to $\mathcal{A}$
2. Add $(P_i, 1, \mathsf{GID})$ to $\mathcal{D}$

Upon receiving GKGen from P_i, send GKGen, P_i to $\mathcal{A}$

Upon receiving $(\mathtt{gENROLL}, \mathsf{GID})$ from some party P_i
1. Send $(\mathtt{gJOIN}, \mathsf{GID})$ on behalf of P_i to $\mathcal{F}_{\mathsf{PrivateMatch}}$.
2. If $\mathtt{gJOIN}, P_i, \mathsf{GID}$ received, Send (GKGen, P_i) to $\mathcal{A}$, add $(P_i, 1, \mathsf{GID})$ to $\mathcal{D}$ and send $\mathtt{gENROLL}, \mathsf{GID}$ to P_i.

Signing by corrupt P_i. Upon receiving $(\mathsf{Sign}, \mathsf{GID}, m)$ on behalf of $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$:
1. Check $(P_i, 1, \mathsf{GID})$ exists in $\mathcal{D}$
2. If yes, send (GID, m) to $\mathcal{A}$ and receive back σ
3. Store (P_i, m, σ) in list $\mathcal{L}$ and send σ to P_i.

Update Profile.
Upon receiving $(\mathtt{UPD}, P_i, \mathsf{prof})$ from the $\mathcal{F}_{\mathsf{PrivateMatch}}$:
1. (Simulate $\mathcal{G}_{\mathsf{reg}}$) : Send $(P_i, \mathsf{prof}, \mathsf{roles})$ to $\mathcal{A}$ and receive cert_i.
2. Store $(P_i, \mathsf{roles}, \mathsf{cert}_i)$ in $\mathcal{L}_{\mathtt{REG}}$.
3. Call $\mathsf{UpdateState}(\mathsf{tx})$, where $\mathsf{tx} = (\mathtt{UPD}, (P_i, \mathsf{cert}))$.

Upon receiving $(\mathtt{UPD}, \mathsf{prof}, \mathsf{roles})$ from P_i (corrupted) on behalf of $\mathcal{G}_{\mathsf{reg}}$
1. If $\mathsf{prof} = \mathsf{GID}$, check that $P_i \in \mathcal{D}$. If not, ignore. Else continue.
2. Send $(\mathtt{UPD}, \mathsf{prof}, \mathsf{roles})$ to $\mathcal{F}_{\mathsf{PrivateMatch}}$. If $(\mathtt{UPD}, P_i, \mathsf{prof})$ not received from $\mathcal{F}_{\mathsf{PrivateMatch}}$, ignore request. Else continue.
3. (Simulate $\mathcal{G}_{\mathsf{reg}}$) Send $(P_i, \mathsf{prof}, \mathsf{roles})$ to $\mathcal{A}$ and receive back cert_i

4. Store cert_i in $\mathcal{L}_{\mathtt{REG}}$, send $(\mathsf{cert}_i, \mathsf{roles})$ to P_i.
5. (Simulating $\mathcal{G}_{\mathsf{smartchain}}$) Upon receiving $\mathsf{tx} = (P_i, \mathsf{prof}, \mathsf{cert}_i)$ from P_i, check if cert_i in $\mathcal{L}_{\mathtt{REG}}$. If yes, call $\mathsf{UpdateState}(\mathsf{tx})$.

Pre-request.
Upon receiving $\mathtt{PREREQ}, (\mathsf{RID}, \mathsf{res}, \mathsf{GID})$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$:
1. Sample $\mathsf{nonce}_0 \leftarrow \{0,1\}^\lambda$ and compute the hash function $H(\mathsf{nonce}_0 \| \mathsf{RID}) = \mathsf{hash}_0$
2. (Simulate $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$) Send $(\mathsf{GID}, m = (\mathsf{hash}_0, \mathsf{res}, \mathsf{RID}))$ to $\mathcal{A}$ and receive back σ. Store $(\cdot, m, \sigma)$ in $\mathcal{L}$.
3. Call $\mathsf{UpdateState}(\mathsf{tx})$, where $\mathsf{tx} = (\mathtt{PREREQ}, m, \sigma)$

Upon receiving $(\mathtt{gSIGN}, \mathsf{GID}, m)$ from P_i on behalf of $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$, where $m = (\mathsf{hash}_0, \mathsf{res}, \mathsf{RID})$:
1. Send $(\mathsf{GID}, m = (\mathsf{hash}_0, \mathsf{res}, \mathsf{RID}))$ to $\mathcal{A}$ and receive back σ. Store (P_i, m, σ) in $\mathcal{L}$.

Upon receiving $\mathsf{tx} = (\mathtt{PREREQ}, m, \sigma)$ from P_i, where $m = (\mathsf{hash}_0, \mathsf{res}, \mathsf{RID}, \mathsf{GID})$
1. If $(P_i, m, \sigma) \in \mathcal{L}$, send $(\mathtt{PREREQ}, \mathsf{GID}, \mathsf{res}, \mathsf{RID})$ to $\mathcal{F}_{\mathsf{PrivateMatch}}$.
2. Upon receiving $(\mathtt{PREREQ}, (\mathsf{RID}, \mathsf{res}, \mathsf{GID}))$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$, call $\mathsf{UpdateState}(\mathsf{tx})$.

Interest.
Upon receiving $(\mathtt{INTRST}, \mathsf{RID}, P_j)$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$:
1. Compute $\sigma = \mathsf{Sig}(\mathsf{sk}_j, (\mathsf{RID}, \mathsf{pk}_j))$
2. Call $\mathsf{UpdateState}(\mathsf{tx})$, where $\mathsf{tx} = (\mathtt{INTRST}, \sigma, \mathsf{RID}, \mathsf{pk}_j)$

Request.
Upon receiving $\mathsf{tx} = (\mathtt{REQ}, \mathsf{RID}, \mathsf{GID}, \mathsf{bidders})$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$
1. Generate design encryption key $k_{\mathsf{RID}} \leftarrow \mathsf{PrivKGen}(1^\lambda)$. Encrypt design - $C_d \leftarrow \mathsf{Enc}(k_{\mathsf{RID}}, 0)$.
2. Sample random $r \leftarrow \{0,1\}^\lambda$ and compute $C_{\mathsf{contract}} = \mathsf{Com}(0; r)$
3. For each $P_j \in \mathsf{bidders}$ - create $C^j_{\mathsf{key}} \leftarrow \mathsf{Enc}(\mathsf{pk}_j, 0)$
4. Generate bid encryption keys $(\mathsf{pk}_{\mathsf{bid}}, \mathsf{sk}_{\mathsf{bid}}) \leftarrow \mathsf{KGen}(1^\lambda)$.
5. Sample $\mathsf{nonce}_1 \leftarrow \{0,1\}^\lambda$ and compute the hash function $H(\mathsf{nonce}_1 \| \mathsf{RID}) = \mathsf{hash}_1$
6. (Simulate $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$) Send $(\mathtt{gSIGN}, \mathsf{GID}, (\mathsf{RID}, \mathsf{pk}_{\mathsf{bid}}, \{C^j_{\mathsf{key}}\}_{j \in \mathsf{bidders}}, C_d, C_{\mathsf{contract}}, \mathsf{hash}_1, \mathsf{nonce}_0))$ to $\mathcal{A}$ and receive σ. Let $\mathsf{signedRequest} = (\mathsf{GID}, (\mathsf{RID}, \mathsf{pk}_{\mathsf{bid}}, \{C^j_{\mathsf{key}}\}_{j \in \mathsf{bidders}}, C_d, C_{\mathsf{contract}}, \mathsf{hash}_1, \mathsf{nonce}_0), \sigma)$. Call $\mathsf{UpdateState}(\mathsf{tx})$, where $\mathsf{tx} = (\mathtt{REQ}, \mathsf{signedRequest})$.

Upon receiving $(\mathtt{gSIGN}, \mathsf{GID}, (\mathsf{RID}, \mathsf{pk}_{\mathsf{bid}}, \{C^j_{\mathsf{key}}\}_{j \in \mathsf{bidders}}, C_d, C_{\mathsf{contract}}, \mathsf{hash}_1, \mathsf{nonce}_0))$ from P_i
1. If $\mathsf{hash}_0 = H(\mathsf{nonce}_0 \| \mathsf{RID})$ and $\mathsf{RID}, \mathsf{hash}_0$ corresponds to an honest party - then ABORT with $\mathsf{CRHFError}$.
2. (Simulating $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$) Check that $(P_i, 1, \mathsf{GID}) \in \mathcal{D}$. If yes, send GID, m to $\mathcal{A}$ and receive σ, Send σ to P_i and store $(P_i, m, \sigma) \in \mathcal{L}$.

Upon receiving $\mathsf{tx} = (\mathtt{REQ}, m, \sigma)$ from P_i , where $m = (\mathsf{RID}, \mathsf{pk}_{\mathsf{bid}}, \{C^j_{\mathsf{key}}\}_{j \in \mathsf{bidders}}, C_d, C_{\mathsf{contract}}, \mathsf{hash}_1, \mathsf{nonce}_0)$
1. Check that $(\cdot, m, \sigma) \in \mathcal{L}$
2. For each $j \in \mathsf{bidders}$:
   (a) Get $k^j_{\mathsf{RID}} = \mathsf{Dec}(\mathsf{sk}_j, C^j_{\mathsf{key}})$
   (b) Compute $\mathsf{design}_j, \mathsf{contract}_j, r_j = \mathsf{Dec}(k^j_{\mathsf{RID}}, C_d)$, which includes either $\perp$ or design, if not $\mathsf{PrivEncError}$.
   (c) Check $C_{\mathsf{contract}} = \mathsf{Com}(\mathsf{contract}_j, r_j)$. If not, stop the execution as an honest supplier would do.
   (d) Send $\mathtt{REQ}, (\mathsf{RID}, [\mathsf{design}_j]_{j \in \mathsf{bidders}}, \mathsf{GID}, \mathsf{bidders})$ to $\mathcal{F}_{\mathsf{PrivateMatch}}$.

Bidding.
Upon receiving $(\mathtt{BID}, (\mathsf{RID}, P_j))$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$:
CASE 1 : Honest requester
1. Compute $C_{\mathsf{bid}} = \mathsf{Enc}(\mathsf{pk}_{\mathsf{bid}}, 0)$
2. Compute $\mathsf{signedBid} = \mathsf{Sig}(\mathsf{sk}_i, C_{\mathsf{bid}})$
3. Call $\mathsf{UpdateState}(\mathsf{tx})$, where $\mathsf{tx} = (\mathtt{BID}, \mathsf{signedBid})$

CASE 2 : Malicious Requester

1. Receive $(\texttt{BID}, (\mathsf{RID}, P_j, \mathsf{bid}_j))$ from the $\mathcal{F}_{\mathsf{PrivateMatch}}$
2. Compute $C_{\mathsf{bid}} = \mathsf{Enc}(\mathsf{pk}_{\mathsf{bid}}, \mathsf{bid}_j)$
3. Compute $\mathsf{signedBid} = \mathsf{Sig}(\mathsf{sk}_i, C_{\mathsf{bid}})$ and call $\mathsf{UpdateState}(\mathsf{tx})$, where $\mathsf{tx} = (\texttt{BID}, \mathsf{signedBid})$.

Match.
Upon receiving $(\mathsf{tx} = \texttt{WINNER}, \mathsf{GID}, \mathsf{RID})$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$:
1. Sample $\mathsf{nonce}_2 \leftarrow \{0,1\}^\lambda$ and compute the hash function $H(\mathsf{nonce}_2 \| \mathsf{RID}) = \mathsf{hash}_2$
2. Send $m = (\mathsf{hash}_2, \mathsf{nonce}_1, \mathsf{ContractID}, \mathsf{RID}, P^*), \mathsf{GID}$ to $\mathcal{A}$ and receive σ
3. Call $\mathsf{UpdateState}(m, \sigma)$

Upon receiving $(\texttt{gSIGN}, m)$, where $m = (\mathsf{hash}_2, \mathsf{nonce}_1, \mathsf{ContractID}, \mathsf{RID}, P^*)$.
1. If nonce_1 corresponds to RID of an honest party - ABORT with $\mathsf{CRHFError}$
2. Send m, GID to $\mathcal{A}$ and receive back σ. Store (m, σ) in $\mathcal{L}$ and send to P_i

Upon receiving $\mathsf{tx} = (\texttt{WINNER}, m, \sigma)$ from P_i, where $m = (\mathsf{hash}_2, \mathsf{ContractID}, \mathsf{nonce}_1, \mathsf{RID}, P^*)$
1. Check that m, σ in $\mathcal{L}$
2. If yes, send $(\texttt{WINNER}, \mathsf{RID}, P^*)$ to $\mathcal{F}_{\mathsf{PrivateMatch}}$
3. Upon receiving $(\texttt{WINNER}, \mathsf{RID}, \mathsf{GID}, P^*)$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$, call $\mathsf{UpdateState}(\mathsf{tx})$

Simulating key generation.
Receiving encryption of key from malicious party:
1. On behalf of $\mathcal{F}_{\mathsf{nizk}}$ receive $(\texttt{PROVE}, (c_{\mathcal{B}}, \mathsf{pk}), \mathsf{sk}_{\mathcal{B}})$. Store $\mathsf{sk}_{\mathcal{B}}$ and send $(\texttt{PROVE}, (c_{\mathcal{B}}, \mathsf{pk}))$ to $\mathcal{A}$ and receive back $(\texttt{PROOF}, (c_{\mathcal{B}}, \mathsf{pk}), \pi)$. Send to malicious party the message $(\texttt{PROOF}, (c_{\mathcal{B}}, \mathsf{pk}), \pi)$
2. Receive $\mathsf{tx}_{\mathcal{B}}, c_{\mathcal{B}}, \pi$ on behalf of the supplier P^*
3. If $\mathsf{tx}_{\mathcal{B}}$ is not valid, abort the request as an honest seller would do.

Sending encryption of key to malicious party:
1. Compute $c_{\mathcal{S}} = \mathsf{Enc}(\mathsf{pk}_{\mathcal{M}}, 0)$
2. Send $(\texttt{PROVE}, c_{\mathcal{S}}, \mathsf{pk}_{\mathcal{M}})$ to $\mathcal{A}$ and receive back $\texttt{PROOF}, c_{\mathcal{S}}, \mathsf{pk}_{\mathcal{M}}, \pi$
3. Create $\mathsf{tx}_{\mathcal{S}}$ and send $\mathsf{tx}_{\mathcal{S}}, c_{\mathcal{S}}, \pi$ to $\mathcal{A}$

Customer Fulfill.
Upon receiving $(\mathsf{tx} = \texttt{RFILL}, \mathsf{GID}, \mathsf{RID}, P^*)$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$:
1. Send $m = (\mathsf{nonce}_2, \mathsf{RID}, P^*), \mathsf{GID}$ to $\mathcal{A}$ and receive σ
2. Call $\mathsf{UpdateState}(\mathsf{tx})$, where $\mathsf{tx} = (m, \sigma)$

Upon receiving $(\texttt{gSIGN}, \mathsf{GID}, (\mathsf{nonce}_2, \mathsf{RID}, P^*))$ from P_i
1. If nonce_2 corresponds to a RID of an honest party, ABORT with $\mathsf{CRHFError}$
2. Send $m = (\mathsf{nonce}_2, \mathsf{RID}, P^*), \mathsf{GID}$ to $\mathcal{A}$ and receive σ. Store (m, σ) in $\mathcal{L}$ and send σ to P_i

Upon receiving $\mathsf{tx} = (\texttt{RFILL}, m, \sigma)$ where $m = (\mathsf{nonce}_2, \mathsf{RID}, P^*)$
1. Check if (m, σ) in $\mathcal{L}$
2. If yes, send $(\texttt{RFILL}, \mathsf{RID}, P^*)$ to $\mathcal{F}_{\mathsf{PrivateMatch}}$
3. Upon receiving $(\texttt{RFILL}, \mathsf{GID}, \mathsf{RID})$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$, call $\mathsf{UpdateState}(\mathsf{tx})$

Supplier Fulfill.
Upon receiving $(\texttt{SFILL}, \mathsf{RID})$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$:
1. Compute $\sigma = \mathsf{Sig}(\mathsf{sk}_i, (\mathsf{RID}))$, where i is the winner of the bid.
2. Call $\mathsf{UpdateState}(\mathsf{tx})$, where $\mathsf{tx} = (\texttt{SFILL}, \mathsf{RID}, \sigma)$

Requester Dispute.
Upon receiving $(\texttt{DISPUTE}, \mathsf{ContractID}, \mathsf{GID}, \mathsf{RID}, \mathsf{reason})$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$:
1. Send $m = (\mathsf{GID}, (\mathsf{DProof}, \mathsf{nonce}_3, \mathsf{ContractID}, \mathsf{RID}, P^*))$ to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$ and receive σ
2. Call $\mathsf{UpdateState}(\mathsf{tx})$, where $\mathsf{tx} = (\texttt{DISPUTE}, m, \sigma)$

Upon receiving $(\texttt{DISPUTE}, P_i, \mathsf{RID}, 0)$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$:
1. Create a $\mathsf{tx} = (\texttt{DISPUTE}, \mathsf{RID}, P_i, 0)$ and call $\mathsf{UpdateState}(\mathsf{tx})$

Upon receiving $(\texttt{gSIGN}, (\mathsf{GID}, (\mathsf{DProof}, \mathsf{nonce}_2, \mathsf{RID}, P^*)))$ from a corrupt party P_i on behalf of $\mathcal{G}_{\mathsf{gsign}}$:
1. If nonce_2 corresponds to a RID of an honest party, ABORT with $\mathsf{CRHFError}$

2. Send $m = (\mathsf{GID}, (\mathsf{DProof}, \mathsf{nonce}_2, \mathsf{RID}, P^*)), \mathsf{GID}$ to $\mathcal{A}$ and receive σ. Store (m, σ) in $\mathcal{L}$ and send σ to P_i

Upon receiving $\mathsf{tx} = (\mathtt{DISPUTE}, m = (\mathsf{DProof}, \mathsf{nonce}_2, \mathsf{RID}, P^*), \sigma)$ on behalf of $\mathcal{G}_{\mathsf{smartchain}}$:

1. Check if $(m, \sigma) \in \mathcal{L}$
2. If yes, send $(\mathtt{DISPUTE}, \mathsf{RID}, \mathsf{GID}, \mathsf{DProof})$ to $\mathcal{F}_{\mathsf{PrivateMatch}}$
3. If $(\mathtt{DISPUTE}, \mathsf{DProof}, \mathsf{RID}, 1)$ received, call $\mathsf{UpdateState}(\mathsf{tx})$, where $\mathsf{tx} = (\mathtt{DISPUTE}, m, \sigma)$
4. If $(\mathtt{DISPUTE}, P_j, \mathsf{RID}, 0)$ received, send $(\mathtt{gOPEN}, m, \sigma)$ to $\mathcal{G}_{\mathsf{gsign}}$ and receive P^*. If $P_j \neq P^*$, abort with $\mathsf{GroupSignatureFailure}$. Else set $\mathsf{tx} = (\mathtt{DISPUTE}, \mathsf{RID}, P_j, 0)$ and call $\mathsf{UpdateState}(\mathsf{tx})$.

Supplier Dispute.

Upon receiving $(\mathtt{DISPUTE}, P_i, \mathsf{DProof}, \mathsf{RID}, (1, P^*))$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$:

1. Compute $\sigma = \mathsf{Sig}(\mathsf{sk}_i, \mathsf{DProof})$
2. Set $\mathsf{tx} = (\mathtt{DISPUTE}, \mathsf{RID}, P^*, \mathsf{DProof}, 1)$

Supplier Messages.

Upon receiving $(\cdot, \cdot\sigma)$ from the adversary on behalf of a supplier P_i, such that $\mathsf{Vrf}(\mathsf{pk}_i, \sigma) = 1$, then abort with $\mathsf{UnforgeabilityError}$

A.2 Proof by Hybrids

Our strategy to prove that the real world and the ideal world are indistinguishable is by starting from the real world and through a series of hybrid worlds we reach the ideal world. More specifically, we prove that Hybrid_i and $\mathsf{Hybrid}i-1$ are indistinguishable for all i from the real world to the ideal world then we prove that the real and ideal worlds are indistinguishable.

Hybrids.

1. Hybrid_0 is the real world execution.
2. Hybrid_1 is the same as Hybrid_0 except that the simulation can now ABORT with $\mathsf{CRHFError}$ message. We prove in Lemma 1 that by the *collision resistance* property of collision-resistant hash functions Hybrid_1 and Hybrid_0 are indistinguishable.
3. $\mathsf{Hybrid2}$ is the same as Hybrid_1 except that the simulation may now abort with $\mathsf{GroupSignatureFailure}$ message. Since we use an ideal groups signature functionality $\mathcal{G}_{\mathsf{gsign}}$, we note the $\mathtt{gOPEN}$ command will always return the correct party P^*. This P^* is the same party in the real world and the simulated world and therefore the two hybrid worlds are indistinguishable, since the simulator will never abort with $\mathsf{GroupSignatureFailure}$.
4. $\mathsf{Hybrid3}$ is the same as Hybrid_2 except that the simulation may abort with the $\mathsf{UnforgeabilityError}$ message. We prove in Lemma 2 that simulator aborts with negligible probability and therefore Hybrid_3 is indistinguishable from Hybrid_2
5. Hybrid_4 is the same as Hybrid_3 except that the simulation can now ABORT with $\mathsf{PrivEncError}$ message. We prove in Lemma 3 that by the *elusive range* property of the "Special" private key encryption scheme that Hybrid_4 and Hybrid_3 are indistinguishable.
6. Hybrid_5 is the same as Hybrid_4 except that the encryptions to the decryption keya replaced with encryptions to 0. We prove in Lemma 4 that Hybrid_5 is indistinguishable from Hybrid_4 by CPA security of the encryption scheme.
7. Hybrid_6 is the same as Hybrid_5 except that the encryptions to the design for the suppliers are now replaced with encryptions to 0. And this is equivalent to the ideal world. We prove in Lemma 5 that Hybrid_6 is indistinguishable from Hybrid_5 by CPA security of the encryption scheme.

Lemma 1. *Assuming the collision resistance of hash functions,* Hybrid_1 *and* Hybrid_0 *are indistinguishable.*

Proof. The difference between Hybrid_1 and Hybrid_0 is that the simulator may abort with a $\mathsf{CRHFError}$. Now the simulator aborts with an $\mathsf{CRHFError}$ if an adversary ($\mathcal{A}$) is able to guess the nonce for a H that was computed by the simulator for an honest party.

Now let D be a distinguisher that distinguishes between Hybrid_1 and Hybrid_0 with non-negligible probability :

$Pr[D(\mathsf{Hybrid}_1) = 1] - Pr[D(\mathsf{Hybrid}_0) = 1] > \mathsf{negl}(n)$

As noted above,

$Pr[D(\mathsf{Hybrid}_1) = 1] - Pr[D(\mathsf{Hybrid}_0) = 1] = Pr[\mathcal{S}_r(\mathsf{Hybrid}_1) = \mathsf{CRHFError}]$

and,

$Pr[\mathcal{S}_r(\mathsf{Hybrid}_1) = \mathsf{CRHFError}] = Pr[\mathcal{A}(\mathsf{hash}) = \mathsf{nonce} \wedge H(\mathsf{nonce} \| \mathsf{RID})$

Thus $Pr[D(\mathsf{Hybrid}_1) = 1] - Pr[D(\mathsf{Hybrid}_0) = 1] = Pr[\mathcal{A}(\mathsf{hash}) = \mathsf{nonce} \wedge H(\mathsf{nonce} \| \mathsf{RID}) = \mathsf{hash}]$

Now a hash function is collision resistant if the advantage of the adversary in the HashColl experiment is negligible.

We now show a reduction B that uses $\mathcal{A}$ and wins the HashColl experiment.

1. Activate the adversary $\mathcal{A}$and execute the PrivateMatch protocol as in Hybrid_1.
2. Receive H from the challenger.
3. Upon receiving a command for the requester from the $\mathcal{F}_{\mathsf{PrivateMatch}}$ functionality $(\texttt{PREREQ}, \texttt{REQ}, \texttt{WINNER})$, simulate the real world transaction with hash_i. Note that $\mathsf{hash}_i = H(x \| \mathsf{RID})$, where x is randomly sampled.
4. Upon receiving $\texttt{gSIGN}, \mathsf{GID}, \cdot, \mathsf{hash}_{i+1}, \mathsf{nonce}_i))$ from $\mathcal{A}$, check that $puz_i = H(\mathsf{nonce}_i \| \mathsf{RID})$ and $\mathsf{nonce}_i \neq x$.
5. Output $(x \| \mathsf{RID})$ and $\mathsf{nonce}_i \| \mathsf{RID}$ to the challenger.

This implies that the reduction wins the HashColl game with non-negligble probability. But this is a contradiction since we assume collision resistant hash functions.

Therefore $Pr[\mathcal{A}(\mathsf{hash}) = \mathsf{nonce} \wedge H(\mathsf{nonce} \| \mathsf{RID}) = \mathsf{hash}] = \mathsf{negl}(n)$ and hence $Pr[D(\mathsf{Hybrid}_1) = 1] - Pr[D(\mathsf{Hybrid}_0) = 1] = \mathsf{negl}(n)$

Lemma 2. *Assuming unforgeable signatures,* Hybrid_3 *and* Hybrid_2 *are indistinguishable.*

Proof. The difference between Hybrid_3 and Hybrid_2 is that the simulator may abort with a UnforgeabilityError. Now the simulator aborts with an UnforgeabilityError if an adversary ($\mathcal{A}$) is able to compute a signature for an honest supplier.

Let D be a distinguisher that distinguishes between Hybrid_3 and Hybrid_2 with non-negligble probability.

As noted above $Pr[D(\mathsf{Hybrid}_3) = 1] - Pr[D(\mathsf{Hybrid}_2) = 1] = Pr[\mathcal{S}_r(\mathsf{Hybrid}_3) = \mathsf{UnforgeabilityError}]$

Moreover, $Pr[\mathcal{S}_r(\mathsf{Hybrid}_3) = \mathsf{UnforgeabilityError}] = Pr[\mathcal{A}$ outputs valid (m,σ) pair]

We construct a reduction B that uses $\mathcal{A}$ to break the unforgeability game for signature schemes, assuming that the reduction has access to a signature oracle.

1. Activate the adversary $\mathcal{A}$
2. Upon receiving any $\texttt{INTRST}, \texttt{BID}, \texttt{SFILL}$ or $\texttt{DISPUTE}$ command for the supplier from $\mathcal{F}_{\mathsf{PrivateMatch}}$, update state with the corresponding tx that includes a signature for supplier P_i, where the signatures are created using the signature oracle.
3. Upon receiving any $\texttt{INTRST}, \texttt{BID}, \texttt{SFILL}$ or $\texttt{DISPUTE}$ transaction for the supplier from $\mathcal{A}$, extract m and σ from the transaction and output (m, σ)

Since we assume secure unforgeable signatures, we note that $Pr[B$ outputs (m, σ) s.t. $\mathsf{Vrf}(\mathsf{pk}, \sigma, m) = 1] < \mathsf{negl}(n)$.

Thus we arrive at a contradiction.

Thus $Pr[\mathcal{A}$ outputs valid (m,σ) pair] $< \mathsf{negl}(n)$, implying that $Pr[\mathcal{S}_r(\mathsf{Hybrid}_3) = \mathsf{UnforgeabilityError}] < \mathsf{negl}(n)$ and hence $Pr[D(\mathsf{Hybrid}_3) = 1] - Pr[D(\mathsf{Hybrid}_2) = 1] < \mathsf{negl}(n)$. Thus completing our proof.

Lemma 3. *Assuming the elusive range property of "special" private key encryption,* Hybrid_4 *and* Hybrid_3 *are indistinguishable.*

Proof. Note that the difference between Hybrid_4 and Hybrid_3 is that in Hybrid_4 the simulator aborts with a message PrivEncError.

Assume a distinguisher D can distinguish between Hybrid_4 and Hybrid_4, i,e. $Pr[D(\mathsf{Hybrid}_4) = 1] - Pr[D(\mathsf{Hybrid}_3) = 1] > \mathsf{negl}(n)$

As noted above,

$Pr[D(\mathsf{Hybrid}_4) = 1] - Pr[D(\mathsf{Hybrid}_3) = 1] = Pr[\mathcal{S}_r(\mathsf{Hybrid}_4) = \mathsf{PrivEncError}]$

Now $Pr[\mathcal{S}_r(\mathsf{Hybrid}_4) = \mathsf{PrivEncError}] = Pr[\mathcal{A}$ outputs C_d s.t. $C_d = \mathsf{Enc}(k_1, x_1) = \mathsf{Enc}(k_2, x_2) \wedge k_1 \neq k_2 \wedge x_1 \neq x_2 \neq \perp]$

Now we describe a reduction B that breaks the elusive-range property of the encryption scheme using the adversary $\mathcal{A}$

1. Activate adversary $\mathcal{A}$
2. Simulate Hybrid4. Upon receiving `REQ` transaction from the $\mathcal{A}$, compute $k^j_{\mathsf{RID}} = \mathsf{Dec}(\mathsf{sk}_j, C^j_{\mathsf{key}})$ for each $j \in \mathsf{bidders}$
3. If there exists, k^a_{RID} and k^b_{RID}, such that $k^a_{\mathsf{RID}} \neq k^b_{\mathsf{RID}}$, compute $\mathsf{design}_a = \mathsf{Dec}(k^a_{\mathsf{RID}}, C_d)$ and $\mathsf{design}_b = \mathsf{Dec}(k^a_{\mathsf{RID}}, C_d)$. If $\mathsf{design}_a \neq \mathsf{design}_b \neq \perp$, simply return C_d.

Note that the reduction outputs C_d that is a valid encryption and in $\mathsf{Range}(k^a_{\mathsf{RID}})$ and $\mathsf{Range}(k^b_{\mathsf{RID}})$.

Now by the elusive range property $Pr_{k \leftarrow G(1^n)}[\mathcal{A}(1^n) \in \mathsf{Range}_n(k)] < \frac{1}{p(n)}$.

This implies we arrive at a contradiction, since we assume the "special" private key encryption with the elusive range property and therefore $Pr[\mathcal{S}_r(\mathsf{Hybrid}_4) = \mathsf{PrivEncError}] < \frac{1}{p(n)}$, which implies $Pr[D(\mathsf{Hybrid}_4) = 1] - Pr[D(\mathsf{Hybrid}_3) = 1] < \frac{1}{p(n)}$

Lemma 4. *Assuming the CPA security of our public key encryption scheme* Hybrid_5 *and* Hybrid_4 *are computationally indistinguishable.*

Proof. Note that the difference between Hybrid_4 and Hybrid_5 is that in Hybrid_5 the encryptions to the keys (C_{key}) are replaced by encryptions to 0.

Assume a distinguisher D can distinguish between Hybrid_4 and Hybrid_5, i,e. $Pr[D(\mathsf{Hybrid}_5) = 1] - Pr[D(\mathsf{Hybrid}_4) = 1] > \mathsf{negl}(n)$

Using this distinguisher D we construct a reduction B that can break the CPA security of encryption scheme.

1. Activate the distinguisher D
2. The reduction simulated PrivateMatch and upon receiving a `REQ` command from $\mathcal{F}_{\mathsf{PrivateMatch}}$, simulate the command.
3. Send $m_0 = k_{\mathsf{RID}}, m_1 = 0$ to the challenger and receive C_{key}.
4. Update state with the `REQ` transaction where the encryption to the key is C_{key}
5. Output whatever D outputs.

Note that in the case C_{key} was the encryption of m_0 the distinguisher sees the hybrid world - Hybrid_4 and on the other when encryption of m_1 is returned the distinguisher sees the hybrid world Hybrid_5.

Now since $Pr[D(\mathsf{Hybrid}_5) = 1] - Pr[D(\mathsf{Hybrid}_4) = 1] > \mathsf{negl}(n)$, we have the adversary winning the CPA with probability $> 1/2 + \mathsf{negl}(n)$ which is a contradiction since we assume CPA secure encryption. This implies $Pr[D(\mathsf{Hybrid}_5) = 1] - Pr[D(\mathsf{Hybrid}_4) = 1] < \mathsf{negl}(n)$.

Lemma 5. *Assuming the CPA security of our public key encryption scheme* Hybrid_6 *and* Hybrid_5 *are computationally indistinguishable.*

Proof. Note that the differnce between Hybrid_6 and Hybrid_5 is that in Hybrid_6 the encryptions to the design (C_d) are replaced by encryptions to 0.

Assume a distinguisher D can distinguish between Hybrid_6 and Hybrid_5, i,e. $Pr[D(\mathsf{Hybrid}_6) = 1] - Pr[D(\mathsf{Hybrid}_5) = 1] > \mathsf{negl}(n)$

Using this distinguisher D we construct a reduction B that can break the CPA security of encryption scheme.

1. Activate the distinguisher D
2. The reduction simulated PrivateMatch and upon receiving a `REQ` command from $\mathcal{F}_{\mathsf{PrivateMatch}}$, simulate the command.

3. Send $m_0 = \mathsf{design}, m_1 = 0$ to the challenger and receive C_{key}.
4. Update state with the $\mathtt{REQ}$ transaction where the encryption to the key is C_d
5. Output whatever D outputs.

Note that in the case C_d was the encryption of m_0 the distinguisher sees the hybrid world - Hybrid_5 and on the other when encryption of m_1 is returned the distinguisher sees the hybrid world Hybrid_6.

Now since $Pr[D(\mathsf{Hybrid}_6) = 1] - Pr[D(\mathsf{Hybrid}_5) = 1] > \mathsf{negl}(n)$, we have the adversary winning the CPA with probability $> 1/2 + \mathsf{negl}(n)$ which is a contradiction since we assume CPA secure encryption. This implies $Pr[D(\mathsf{Hybrid}_6) = 1] - Pr[D(\mathsf{Hybrid}_5) = 1] < \mathsf{negl}(n)$.

A.3 The malicious supplier case

Theorem 2 *(restated)* **:** *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 [22], EUF-CMA signature [22], secure commitment schemes [22] and CPA-secure encryption [22] in the presence of a PPT adversary that corrupts a subset of the suppliers.*

Proof. Simulator - $\mathcal{S}_s$

Registration

Same as Sim $\mathcal{S}_r$

Join Group Upon receiving $(\mathtt{gJOIN}, P_i, \mathsf{GID})$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$ (Simulate $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$) :
1. Send (GKGen, P_i) to $\mathcal{A}$
2. Add $(P_i, 1, \mathsf{GID})$ to $\mathcal{D}$

Update Profile

Same as Sim $\mathcal{S}_r$

Pre-request.

Upon receiving $\mathtt{PREREQ}, (\mathsf{RID}, \mathsf{RequestResource}, \mathsf{GID})$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$:
1. Sample $\mathsf{nonce}_0 \leftarrow \{0,1\}^\lambda$ and compute the hash function $H(\mathsf{nonce}_0 \| \mathsf{RID}) = \mathsf{hash}_0$
2. (Simulate $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$) Send $(\mathsf{GID}, m = (\mathsf{hash}_0, \mathsf{RequestResource}, \mathsf{RID}))$ to $\mathcal{A}$ and receive back σ. Store $(\cdot, m, \sigma)$ in $\mathcal{L}$.
3. Call $\mathsf{UpdateState}(\mathsf{tx})$, where $\mathsf{tx} = (\mathtt{PREREQ}, m, \sigma)$

Interest.

Upon receiving $(\mathtt{INTRST}, \mathsf{RID}, P_j)$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$:
1. Compute $\sigma = \mathsf{Sig}(\mathsf{sk}_j, (\mathsf{RID}, \mathsf{pk}_j))$
2. Call $\mathsf{UpdateState}(\mathsf{tx})$, where $\mathsf{tx} = (\mathtt{INTRST}, \sigma, \mathsf{RID}, \mathsf{pk}_j)$

Upon receiving $\mathtt{INTRST}, ((\mathsf{RID}, \mathsf{pk}_i), \sigma)$ from a party P_i (on behalf of $\mathcal{G}_{\mathsf{smartchain}}$)
1. Send $(\mathtt{INTRST}, \mathsf{RID})$ on behalf of P_i to $\mathcal{F}_{\mathsf{PrivateMatch}}$
2. If $\mathsf{tx} = (\mathtt{INTRST}, \mathsf{RID}, P_i)$ received back, then call $\mathsf{UpdateState}(\mathsf{tx})$

Request.

Upon receiving $\mathsf{tx} = (\mathtt{REQ}, \mathsf{RID}, \mathsf{GID}, \mathsf{bidders})$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$, for all malicious $P_i \in \mathsf{bidders}$ also receive $(\mathtt{REQ}, \mathsf{RID}, \mathsf{design})$:
1. For honest suppliers, generate design encryption key $k_{\mathsf{RID}} \leftarrow \mathsf{PrivKGen}(1^\lambda)$. Encrypt design - $C_d \leftarrow \mathsf{Enc}(k_{\mathsf{RID}}, 0)$. For malicious suppliers P_i, $C_d \leftarrow \mathsf{Enc}(k_{\mathsf{RID}}, \mathsf{design})$.
2. For each honest $P_j \in \mathsf{bidders}$ - create $C^j_{\mathsf{key}} \leftarrow \mathsf{Enc}(\mathsf{pk}_j, 0)$ and for each malicious $P_i \in \mathsf{bidders}$ - create $C^i_{\mathsf{key}} \leftarrow \mathsf{Enc}(\mathsf{pk}_i, k_{\mathsf{RID}})$
3. Generate bid encryption keys $(\mathsf{pk}_{\mathsf{bid}}, \mathsf{sk}_{\mathsf{bid}}) \leftarrow \mathsf{KGen}(1^\lambda)$.
4. Sample $\mathsf{nonce}_1 \leftarrow \{0,1\}^\lambda$ and compute the hash function $H(\mathsf{nonce}_1 \| \mathsf{RID}) = \mathsf{hash}_1$
5. (Simulate $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$) 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 $\mathcal{A}$ and receive σ. Let $\mathsf{signedRequest} = (\mathsf{GID}, (\mathsf{RID}, \mathsf{pk}_{\mathsf{bid}}, \{C^j_{\mathsf{key}}\}_{j \in \mathsf{bidders}}, C_d, \mathsf{hash}_1, \mathsf{nonce}_0), \sigma)$. Call $\mathsf{UpdateState}(\mathsf{tx})$, where $\mathsf{tx} = (\mathtt{REQ}, \mathsf{signedRequest})$.

Bidding.
Upon receiving $(\texttt{BID}, (\mathsf{RID}, P_j))$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$:
1. Compute $C_{\mathsf{bid}} = \mathsf{Enc}(\mathsf{pk}_{\mathsf{bid}}, 0)$
2. Compute $\mathsf{signedBid} = \mathsf{Sig}(\mathsf{sk}_i, C_{\mathsf{bid}})$
3. Call $\mathsf{UpdateState}(\mathsf{tx})$, where $\mathsf{tx} = (\texttt{BID}, \mathsf{signedBid})$

Upon receiving $\mathsf{tx} = (\texttt{BID}, \mathsf{signedBid})$ from P_i on $\mathcal{G}_{\mathsf{smartchain}}$:
1. Verify that $\mathsf{signedBid} = \mathsf{Sig}_{\mathsf{sk}_i}(\mathsf{RID}, C_{\mathsf{bid}})$ is verified using pk_i
2. Decrypt C_{bid} using $\mathsf{sk}_{\mathsf{bid}}$ and get bid.
3. Send $(\texttt{BID}, (\mathsf{RID}, \mathsf{bid}))$ on behalf of P_i to $\mathcal{F}_{\mathsf{PrivateMatch}}$. Upon receiving $(\texttt{BID}, (\mathsf{RID}, P_i))$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$, call $\mathsf{UpdateState}(\mathsf{tx})$.

Match.
Upon receiving $(\mathsf{tx} = \texttt{WINNER}, \mathsf{GID}, \mathsf{RID}, P^*)$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$:
1. Sample $\mathsf{nonce}_2 \leftarrow \{0,1\}^\lambda$ and compute the hash function $H(\mathsf{nonce}_2 \| \mathsf{RID}) = \mathsf{hash}_2$
2. Send $m = (\mathsf{hash}_2, \mathsf{nonce}_1, \mathsf{RID}, P_j^*), \mathsf{GID}$ to $\mathcal{A}$ and receive σ
3. Call $\mathsf{UpdateState}(m, \sigma)$

Requester Fulfill.
Upon receiving $(\mathsf{tx} = \texttt{RFILL}, \mathsf{GID}, \mathsf{RID}, P_j^*)$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$:
1. Send $m = (\mathsf{nonce}_2, \mathsf{RID}, P_j^*), \mathsf{GID}$ to $\mathcal{A}$ and receive σ
2. Call $\mathsf{UpdateState}(\mathsf{tx})$, where $\mathsf{tx} = (m, \sigma)$

Supplier Fulfill.
Upon receiving $(\texttt{SFILL}, \mathsf{RID})$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$:
1. Compute $\sigma = \mathsf{Sig}(\mathsf{sk}_i, (\mathsf{RID}))$, where i is the winner of the bid.
2. Call $\mathsf{UpdateState}(\mathsf{tx})$, where $\mathsf{tx} = (\texttt{SFILL}, \mathsf{RID}, \sigma)$

Upon receiving $(\texttt{SFILL}, \mathsf{signedFulfillS})$ from a party P_i on behalf of $\mathcal{G}_{\mathsf{smartchain}}$:
1. Verify $\mathsf{signedFulfillS} = (\mathsf{RID}, \sigma)$ can be verified using pk_i
2. Send $(\texttt{SFILL}, \text{“}\mathsf{fulfill}\text{”}, \mathsf{RID})$ to $\mathcal{F}_{\mathsf{PrivateMatch}}$ and upon receiving $(\texttt{SFILL}, \mathsf{RID})$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$, call $\mathsf{UpdateState}(\mathsf{tx})$.

Requester Dispute. Upon receiving $(\texttt{DISPUTE}, \mathsf{DProof}, \mathsf{RID}, 1)$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$:
1. Send $m = (\mathsf{GID}, (\mathsf{DProof}, \mathsf{nonce}_2, \mathsf{RID}, P^*))$ to $\mathcal{G}_{\mathsf{gsign}}[\mathsf{GID}]$ and receive σ
2. Call $\mathsf{UpdateState}(\mathsf{tx})$, where $\mathsf{tx} = (\texttt{DISPUTE}, m, \sigma)$

Supplier Dispute. Upon receiving $(\texttt{DISPUTE}, P_i, \mathsf{DProof}, \mathsf{RID}, (1, P^*))$ from $\mathcal{F}_{\mathsf{PrivateMatch}}$:
1. Compute $\sigma = \mathsf{Sig}(\mathsf{sk}_i, \mathsf{DProof})$
2. Set $\mathsf{tx} = (\texttt{DISPUTE}, \mathsf{RID}, P^*, \mathsf{DProof}, 1)$

Upon receiving $\mathsf{tx} = (\texttt{DISPUTE}, (\mathsf{DProof}, \mathsf{RID}, \sigma))$ from a party P_i on behalf of $\mathcal{G}_{\mathsf{smartchain}}$
1. Send $(\texttt{DISPUTE}, \mathsf{RID}, \mathsf{DProof})$ on behalf of P_i to $\mathcal{F}_{\mathsf{PrivateMatch}}$
2. If $(\texttt{DISPUTE}, P_i, \mathsf{DProof}, \mathsf{RID}, (1, P^*))$ received, set $\mathsf{tx} = (\texttt{DISPUTE}, \mathsf{RID}, P^*, \mathsf{DProof}, \sigma, 1)$ and call $\mathsf{UpdateState}(\mathsf{tx})$
3. Else ignore the message.

A.4 Proof by Hybrids

Our strategy to prove that the real world and the ideal world are indistinguishable is by starting from the real world and through a series of hybrid worlds we reach the ideal world. More specifically, we prove that Hybrid_i and $\mathsf{Hybrid}i-1$ are indistinguishable for all i from the real world to the ideal world then we prove that the real and ideal worlds are indistinguishable. **Hybrids**.

1. Hybrid_0 is the real world execution.
2. Hybrid_1 is the same as Hybrid_0 except that the simulation can now ABORT with $\mathsf{CRHFError}$ message. Similar to the proof in Lemma 1 we can prove that by the *collision resistance* property of Collision Resistant Hash Functions functions Hybrid_1 and Hybrid_0 are indistinguishable.

3. Hybrid2 is the same as Hybrid_1 except that the simulation may abort with the UnforgeabilityError message. Similar to the proof in Lemma 2 we prove that simulator aborts with negligible probability and therefore Hybrid_2 is indistinguishable from Hybrid_1
4. Hybrid_3 is the same as Hybrid_2 except that the encryptions to the decryption keya replaced with encryptions to 0. Similar to the proof in Lemma 4 we can prove that Hybrid_3 is indistinguishable from Hybrid_2 by CPA security of the encryption scheme.
5. Hybrid_4 is the same as Hybrid_3 except that the encryptions to the design for the suppliers are now replaced with encryptions to 0. And this is equivalent to the ideal world. Similar to the proof in Lemma 5 we can prove that Hybrid_4 is indistinguishable from Hybrid_3 by CPA security of the encryption scheme.