Strain: A Secure Auction for Blockchains

Erik-Oliver Blass[1] and Florian Kerschbaum[2]

[1] Airbus, Munich, Germany
erik-oliver.blass@airbus.com
[2] University of Waterloo, Waterloo, Canada
florian.kerschbaum@uwaterloo.ca

Abstract. We present Strain, a new auction protocol running on top of blockchains and guaranteeing bid confidentiality against fully-malicious parties. As our goal is efficiency and low blockchain latency, we abstain from using traditional, highly interactive MPC primitives such as secret shares. We focus on a slightly weaker adversary model than MPC which allows Strain to achieve constant latency in both the number of parties and the bid length. The main idea behind Strain is a new maliciously-secure two-party comparison mechanism executed between any pair of bids in parallel. Using zero-knowledge proofs, Strain broadcasts the outcome of comparisons on the blockchain in a way that all parties can verify each outcome. Strain's latency is not only asymptotically optimal, but also efficient in practice, requiring a total of just 4 blocks of the underlying blockchain. Strain provides typical auction security requirements such as non-retractable bids against fully-malicious adversaries.

1 Introduction

Today's blockchains offer transparency and integrity features which could make them ideal for hosting auctions. Once a bid has been submitted to a smart contract managing the auction on the blockchain, the bid cannot be retracted anymore. After a deadline has passed, everybody can verify the winning bid. Due to its attractive features, blockchain auctions are already considered in the real-world. As a prominent example to fight nepotism and corruption, Ukraine will host blockchain auctions to sell previously seized goods [33].

However, today's blockchain transparency features disqualify them in scenarios where input data must remain confidential. For example, in a procurement auction, another prime application example for blockchains [1], an *auctioneer* requests offers for some good ("Need 1M grade V2X steel screws") as part of a smart contract. A set of *suppliers* submits bids for the good, and the lowest bid wins the procurement auction. Realizing a decentralized auction as a smart contract has the above transparency features, mitigates corruption, and avoids a possibly corrupt, centralized auctioneer. Yet, bids are confidential. Suppliers have mutual distrust, and leaking the value of a bid to a competitor must be avoided. In some situations, one supplier should not even learn whether or not another supplier is participating in an auction. To make matters worse, multiple suppliers might collude, be fully-malicious, behave randomly (not rationally), and abort participation in the auction to disturb its outcome. Still, the auction should run as expected.

Kosba et al. [26] already mention that one could revert to implementing the auction with Secure Multi-Party Computation (MPC) on the blockchain. While there has been a flurry of research on MPC, and generic frameworks are readily available [38], a main MPC drawback is its high interactivity. Yet, interactivity is extremely expensive on a blockchain in terms of latency. Broadcasting a message, changing the state of a smart contract (code execution), and any kind of party interactivity requires a valid transaction. As transactions are attached to blocks, any interactivity requires (at least) one block interval for completion. Block interval times are large, e.g., roughly 15 s for Ethereum [19]. Thus, high interactivity, a large number of MPC rounds, automatically rules out short-term, short living auctions.

This paper We present Strain ("Secure aucTions foR blockchAINs"), a new protocol for secure auctions on blockchains. At the heart, we improve Fischlin [21]'s comparison protocol in several key aspects tailored for adoption in blockchains. First, Strain features a distributed key generation for Goldwasser-Micali encryption based on a new mechanism to verifiably share each supplier's private key. Suppliers initially commit to their bids by encrypting them with their public key. A honest majority of suppliers can then open a commitment in case a supplier aborts the protocol.

Strain's second main feature is an efficient zero-knowledge (ZK) proof that two Goldwasser-Micali ciphertexts, encrypted under different keys, contain the same plaintext. For this proof, we require existence of a semi-honest *judge* party which must not collude with either of the comparing parties. In the context of auctions, the judge can be implemented by, e.g., the auctioneer. Using ZK proofs, the judge verifies (and publishes on the blockchain) whether both parties use previously committed values as input to the comparison. Again using a ZK proof, the comparing party then publishes the outcome of the comparison on the blockchain. Together, the two ZK proofs allow everybody to verify correctness of the comparison's result in only 3 blocks (totaling 4 blocks for the entire Strain protocol). We achieve such low latency by providing slightly weaker security guarantees than MPC would have. Specifically, the semi-honest judge would not be required in MPC. Strain also leaks the order of bids, but not their value.

Strain optionally supports anonymous auctions by using a combination of Dining Cryptographer networks and blind signatures. Suppliers can be anonymized, such that no supplier knows which other suppliers are participating in an auction. Note that we specifically avoid payment channels [37], and all communication will run through the blockchain. The advantage is no or only little data stored at parties, crucial information stored at the central ledger, and no direct network connectivity required between parties.

We benchmarked main cryptographic operations, and our analysis shows that Strain supports auctions of up to dozens of concurrent suppliers within 3 Ethereum blocks.

In summary, the *technical highlights* of this paper are:

- A new blockchain auction protocol, Strain, protecting confidentiality of bids. Strain provides provable security against fully-malicious suppliers and semi-honest auctioneers. It is efficient and completes an auction in a constant (four) number of interactions, i.e., blockchain blocks. Its round complexity is independent from the bit length of the bids (multiplicative depth of a comparison circuit) and the number of suppliers.
- After bidding, no supplier can retract or modify a bid. However, in case of dispute, commitments can be opened by an honest majority. Strain will complete, even if ma-

licious parties fail to respond and abort the auction without any supplier being able to change their bid. Computation of the winning bid is performed solely by the suppliers and entirely on the blockchain. The contribution of the auctioneer to the auction is only to verify correctness of computations in zero-knowledge.

We stress that the lack of smart contract data confidentiality is independent from privacy-preserving coin transactions, see, e.g., ZeroCash [3] for an overview. To reach consensus, blockchain miners generally require access to all contract input data. Also, permissioned blockchains such as Hyperledger (Fabric) lack confidentiality, even if contract execution can be restricted to only those parties participating in a contract.

2 Background

Let $\mathcal{S}=\{S_1,...,S_s\}$ be the set of s suppliers in the system with public-private key pairs (pk_i,sk_i). The procurement auction is run by auctioneer A having public-private key pair (pk_A,sk_A). Assume that all suppliers and A know each other's public keys, so A can run an auction accepting bids from valid suppliers only.

2.1 Preliminaries

Let λ be the security parameter. For an integer n, let QR_n be the set of quadratic residues of group $\mathbb{Z}_n$, and QNR_n is the set of quadratic non-residues of $\mathbb{Z}_n$. Function $J_n(x)$ computes the Jacobi symbol $\left(\frac{x}{n}\right)$, and we define set $\mathbb{J}_n=\{x\in\mathbb{Z}_n|J_n(x)=1\}$. Finally, $QNR_n^1=\{x\in QNR_n|J_n(x)=1\}$ (set of "pseudo-squares").

Quadratic Residues modulo Blum Integers. An integer n is a Blum integer, if $n=p\cdot q$ for two distinct primes p,q and $p=q=3 \bmod 4$. If n is a Blum integer, testing whether some $x\in\mathbb{Z}_n$ with $J_n(x)=1$ is in QR_n can be implemented by checking whether $x^{\frac{(p-1)\cdot(q-1)}{4}}=1 \bmod n$ [25]. Moreover, observe that the DDH assumption holds in group $(\mathbb{J}_n,\cdot)$. For $r\xleftarrow{\$}\mathbb{Z}_n^*$, $g=-r^2 \bmod n$ is a generator of group $(\mathbb{J}_n,\cdot)$, see Section A.1 of Couteau et al. [13]. In particular $z=-1=-(1^2) \bmod n$ is a generator of $\mathbb{J}_n$.

GM Encryption. A Goldwasser-Micali (GM) [23] key pair comprises private key sk^{GM} and public key pk^{GM}. For p and q being distinct, strong random primes of length λ, the private key is $sk^{\mathsf{GM}}=\frac{(p-1)\cdot(q-1)}{4}$. We require $p=q=3 \bmod 4$, and therefore $n=p\cdot q$ is a Blum integer. We set $z=n-1=-1 \bmod n$. The public key is $pk^{\mathsf{GM}}=(n,z)$. With n being a Blum integer, $z\in QNR_n^1$.

With randomly chosen $r_i\xleftarrow{\$}\mathbb{Z}_n^*$, GM encryption of bit string $M\in\{0,1\}^\eta$ is $C=\mathsf{Enc}^{\mathsf{GM}}_{pk^{\mathsf{GM}}}(M_1\ldots M_\eta)=(r_1^2\cdot z^{M_1} \bmod n,\ldots,r_\eta^2\cdot z^{M_\eta} \bmod n)$. All parties automatically dismiss a ciphertext C if $C\notin\mathbb{J}_n$.

Decryption of ciphertext C simply checks whether each component of $C=(c_1,...,c_\eta)$ is in QR_n. As n is a Blum integer, raising c_i to secret key sk^{GM} is sufficient, i.e., you compute $M=\mathsf{Dec}^{\mathsf{GM}}_{sk^{\mathsf{GM}}}(c_1,...,c_\eta)=(1-c_1^{sk^{\mathsf{GM}}} \bmod n,...,1-c_\eta^{sk^{\mathsf{GM}}} \bmod n)$.

Recall GM's homomorphic properties for encryptions of two bits b_1,b_2 (when obvious, we omit public-/private keys in this paper for better readability):

- $\mathsf{Dec}^{\mathsf{GM}}(\mathsf{Enc}^{\mathsf{GM}}(b_1)\cdot\mathsf{Enc}^{\mathsf{GM}}(b_2))=b_1\oplus b_2$ (plaintext XOR)
- $\mathsf{Dec}^{\mathsf{GM}}(\mathsf{Enc}^{\mathsf{GM}}(b_1)\cdot z)=1-b_1$ (flip plaintext bit b_1)
- For a GM ciphertext c, re-encryption is $\mathsf{ReEnc}^{\mathsf{GM}}(c)\leftarrow c\cdot\mathsf{Enc}^{\mathsf{GM}}(0)$.

AND-Homomorphic GM Encryption. GM encryption can be modified to support AND-homomorphism [21, 34]. Specifically, let λ' be the soundness parameter of the Sander et al. [34] technique that works as follows.

A *single* bit $b=1$ is encrypted to λ'-many random quadratic residues $\mod n$, i.e., λ' separate GM encryptions of 0. A bit $b=0$ is encrypted to a sequence of random elements x with $J_n(x)=1$, i.e., λ' encryptions of randomly chosen bits $a_1,\ldots,a_{\lambda'}$. More formally,

$\mathsf{Enc}^{\mathsf{AND}}(1)=(\mathsf{Enc}^{\mathsf{GM}}(0),\ldots,\mathsf{Enc}^{\mathsf{GM}}(0))$ and$\mathsf{Enc}^{\mathsf{AND}}(0)=(\mathsf{Enc}^{\mathsf{GM}}(a_1),\ldots,\mathsf{Enc}^{\mathsf{GM}}(a_{\lambda'}))$.

Decryption of a sequence of a λ'-element ciphertext checks whether all elements are in QR_n. That is, $\mathsf{Dec}^{\mathsf{AND}}(c_1,\ldots,c_{\lambda'})=1$, if $\forall c_i: c_i \in QR_n$, and 0 otherwise.

As an AND-encryption of 0 can result in λ' elements of QR_n, decryption is correct with probability $1-2^{-\lambda'}$.

$\mathsf{Enc}^{\mathsf{AND}}$ is homomorphic with respect to Boolean AND. For two ciphertexts $\mathsf{Enc}^{\mathsf{AND}}(b)=(c_1,\ldots,c_{\lambda'})$ and $\mathsf{Enc}^{\mathsf{AND}}(b')=(c'_1,\ldots,c'_{\lambda'})$, $\mathsf{Dec}^{\mathsf{AND}}(c_1\cdot c'_1,\ldots,c_{\lambda'}\cdot c'_{\lambda'})=b\wedge b'$. If the c_i and c'_i are all in QR_n, so is their product. If one is in QR_n and the other in QNR^1_n, their product is in QNR^1_n. Yet, if both c_i and c'_i are in QNR^1_n, their product is in QR_n. For example, if all c_i and c'_i are in QNR^1_n, $b=b'=0$, but $\mathsf{Dec}^{\mathsf{AND}}$ after their homomorphic combincation will output 1. So, $\mathsf{Dec}^{\mathsf{AND}}$ is correct with probability $1-2^{-\lambda'}$. Re-encryption for AND-encryption is simply defined as $\mathsf{ReEnc}^{\mathsf{AND}}(c_1,\ldots,c_{\lambda'})\leftarrow(\mathsf{ReEnc}^{\mathsf{GM}}(c_1),\ldots,\mathsf{ReEnc}^{\mathsf{GM}}(c_{\lambda'}))$.

Finally, we can embed an existing GM ciphertext $\gamma=\mathsf{Enc}^{\mathsf{GM}}(b)$ of bit b into an a ciphertext $\mathsf{Enc}^{\mathsf{AND}}(b)=(c_1,\ldots,c_{\lambda'})$ without decryption. First, we choose λ' random bits $a_1,\ldots,a_{\lambda'}$. Now, if $a_i=1$, then set $c_i=\mathsf{Enc}^{\mathsf{GM}}(0)$. Otherwise, set $c_i=\mathsf{Enc}^{\mathsf{GM}}(0)\cdot\gamma\cdot z \mod n$. In the first case, c_i is a quadratic residue independently of b ($c_i=\mathsf{Enc}^{\mathsf{GM}}(0)$). In the second case, we flip bit b by multiplying with z (and re-encrypt the result). So, a quadratic residue c_i becomes a non-residue and the other way around. If $b=1$, all λ' elements c_i will be quadratic residues. If $b=0$, all λ' elements c_i will be quadratic residues only with probability $2^{-\lambda'}$, such that the embedding is correct with probability $1-2^{-\lambda'}$.

2.2 Blockchain

There exist several detailed introductions to blockchain and smart contract technology such as Ethereum [18]. Here, we only briefly and informally summarize properties relevant for Strain.

A blockchain is a distributed network implementing a ledger functionality. Parties can append transactions to the ledger, if the network validates transactions in a distributed fashion. Surprisingly, such a ledger is sufficient to realize distributed execution of programs called smart contracts. Using transactions, one party uploads code and state into the blockchain, and other parties modify state by stipulating code execution. For a procurement auction, auctioneer A would upload a new smart contract and allow other parties to bid. That is, the smart contract could just implement a simple, initially empty mailbox as state, and suppliers could only append data (bids and anything else) to that mailbox by transactions. All blockchain transactions are automatically signed by their generating party, and so would be the data they carry. Such a simple mailbox smart contract provides the following properties that we will need.

First, the blockchain guarantees *reliable broadcast*. Each signed transaction appending a message to the mailbox is public. Based on the blockchain's consensus, everybody

```
1 forall $S_i$ do
2     if Pseudonymity then $S_i \rightarrow TTP$: $\mathcal{F}_{\mathsf{Pseu}}(v_i)$; else $S_i \rightarrow TTP$: $\mathcal{F}_{\mathsf{Auth}}(v_i)$;
3 for $i=1$ to $s$ do
4     forall $j \neq i$ do
5         $TTP$: Let $cmp_{i,j}=1$, if $v_i > v_j$ and $cmp_{i,j}=0$ otherwise;
6 $TTP \rightarrow \{A,S_1,...,S_s\}$: $\mathcal{F}_{\mathsf{BC}}(\{cmp_{i,j} | \forall i,j \in \{1,...,s\}\})$;
7 $TTP \rightarrow A$: $\{v_w | v_w = \min(v_1,...,v_s)\}$;
```

Algorithm 1: Ideal Functionality $\mathcal{F}_{\mathsf{Bid}}$ of the bidding algorithm

in the network observes the same message appended (if valid). Being the blockchain's core feature, reliable broadcast takes one block latency. Along the same lines, we can introduce *personal messages* between parties over the blockchain. A broadcast to supplier S_i encrypted with S_i's public key realizes a secure, reliable channel to S_i.

Moreover, a blockchain automatically allows for *deadlines*. Parties participating in the blockchain receive new blocks and therefore have (weakly) synchronized clocks. Based on the current block, an auction smart contract can specify a deadline as a function of the number of future blocks.

Note that with, e.g., Ethereum, there is essentially no limit for the number of transactions per block. Miners have an incentive to include as many transactions as possible in their block to receive transaction fees. Thus, large messages can therefore be split into multiple transactions and still sent as "one message". Consequently in this paper, we silently assume that the blockchain accepts any number of messages of arbitrary length per block. In practice with Ethereum, the GasLimit upper bounds transactions and their size, but one could imagine that a long messages m is stored in a Public Bulletin Board (PBB) system, and the blockchain only stores hash of m.

To ease exposition, we also assume the blockchain consensus to be fork-free. As today's Proof-of-Work-based blockchains accept longer forks at any time, they cannot be fork-free. However in practice, a honest majority of miners guarantees probability p of a future fork of length $k=O(\lambda)$ to become exponentially small, i.e., $p=e^{-\Omega(\lambda)}$ [22]. Parameter k is small in practice, e.g., $k=6$ in Bitcoin and $k=30$ in Ethereum. Blockchains based on Byzantine fault tolerance typically have consensus finality (and are fork-free) [39].

3 Security Definition

We define security following the standard ideal vs. real world paradigm. First, we specify an ideal functionality $\mathcal{F}_{\mathsf{Bid}}$ of our bidding protocol, see Algorithm 1.

Ideal Functionality Our protocol emulates a trusted third party TTP that, first, receives all bids from all suppliers. If supplier pseudonymity is required, all participating suppliers S_i send their bids v_i via a pseudonymous channel, or else they send it via an authenticated channel. The trusted third party then computes result $cmp_{i,j}$ of the comparsion between each bid. Finally, the trusted third party announces (broadcasts) the results of all comparisons to auctioneer A, each Supplier S_i, and all other participants of the blockchain. Similar to order preserving encryption, this reveals the total order of bids and hence the winner of the auction, but does not reveal the bids themselves.

Adversary Model We consider two adversaries $\mathcal{A}_1$ and $\mathcal{A}_2$. These adversaries have different capabilities, are non-colluding, and control different parties. The following Theorem 1 summarizes our main contribution, and we will prove it later in the paper.

Theorem 1. *If adversary $\mathcal{A}_1$ is a static, active adversary which may control up to a threshold*[3] *τ of suppliers S_i, and if Adversary $\mathcal{A}_2$ is a passive adversary which may control auctioneer A, and if $\mathcal{A}_1$ and $\mathcal{A}_2$ do not collude, then protocol* Strain *implements functionality $\mathcal{F}_{\mathsf{Bid}}$.*

The order of bids is revealed to the adversary, and the auctioneer, but not the suppliers, must be only semi-honest. While this results in slightly weaker security than offered by MPC, it allows for optimally low latency. Moreover, we conjecture that this adversary model is practical in a variety of real-world scenarios.

4 Maliciously-Secure Comparisons

The first ingredient to our main contribution of secure auctions is a generic comparison construction. It allows two parties S_i and S_j (the suppliers in our application) with inputs v_i and v_j to obliviously evaluate whether or not $v_i > v_j$ without disclosing anything else to the other party. In contrast to related work, the novelty of our construction is its efficiency in the face of fully malicious adversaries. We do not rely on general MPC primitives and have asymptotically optimal complexity (3 blocks and $O(\eta)$ computation and communication cost per comparison). This allows us to easily integrate our comparison into the auction framework of Section 5 and, e.g., tolerate parties aborting the auction without restarting comparisons.

To realize maliciously-secure comparisons, we rely on the existence of a *judge A* (the auctioneer in our application). S_i and S_j can be fully malicious, but A must be semi-honest and moreover not collude with S_i,S_j, see Section 7. As long as A does not collude with S_i,S_j, neither A nor a malicious supplier learn bids of honest suppliers. An important property of our solution is that knowledge of S_i's, S_j's, and A's public keys is sufficient to verify whether $v_i > v_j$, again without learning anything else about v_i and v_j.

4.1 Secure Comparisons Against Semi-Honest Adversaries

We begin by presenting Fischlin [21]'s technique for comparisons, secure against semi-honest adversaries. Subsequently, we extend comparisons to be secure against fully malicious adversaries.

Given bit representations $v_i = v_{i,1} \dots v_{i,\eta}$ and $v_j = v_{j,1} \dots v_{j,\eta}$, we can compute $v_i > v_j$ by evaluating Boolean circuit $F = \bigvee_{\ell=1}^{\eta} (v_{i,\ell} \wedge \neg v_{j,\ell} \wedge \bigwedge_{u=\ell+1}^{\eta} (v_{i,u} = v_{j,u}))$. We have $F = 1$ *iff* $v_i > v_j$. Observe that the main $\bigvee_{t=1}^{\eta}$ is actually an XOR: if $v_i > v_j$, exactly one term will be 1, and all other terms are 0. If $v_i \leq v_j$, all terms will be 0. Moreover, $(v_{i,u} = v_{j,u})$ equals $\neg(v_{i,u} \oplus v_{j,u})$. That can be exploited to homomorphically evaluate F using GM encryption.

1. S_i sends its GM public key $pk_i^{\mathsf{GM}} = (z_i, n_i)$ and encrypted value $C_i = \mathsf{Enc}^{\mathsf{GM}}_{pk_i^{\mathsf{GM}}}(v_i)$, a sequence of GM ciphertexts, to S_j.

[3] Threshold τ will later be used to open commitments using Shamir's secret sharing of the key, cf. Section 5.1.

2. S_j encrypts its own value v_j with S_i's public key, $C_{i,j} = \mathsf{Enc}^{\mathsf{GM}}_{pk_i^{\mathsf{GM}}}(v_j)$. S_j then homomorphically computes all $\neg(v_{i,u} \oplus v_{j,u})$ and $\neg v_{j,\ell}$ from F.
3. S_j embeds C_i and its own sequence of ciphertexts $C_{i,j}$ into AND-homomorphic GM ciphertexts as described in Section 2.1. Using AND-homomorphism, S_j computes a sequence $\ell = \{1,...,\eta\}$ of ciphertexts $c_\ell = (v_{i,\ell} \wedge \neg v_{j,\ell} \wedge \bigwedge_{u=\ell+1}^{\eta} (v_{i,u} = v_{j_u}))$. Finally, S_j randomly shuffles the order of ciphertexts c_ℓ and sends resulting permutation $res_{i,j} = \pi(c_1,...,c_\eta)$ back to S_i.
4. S_i can decrypt the c_ℓ in $res_{i,j}$ and learns whether $v_i \leq v_j$, if all c_ℓ decrypt to 0, or $v_i > v_j$, if exactly one ciphertext decrypts to 1 and all other to 0.

The purpose of S_j shuffling ciphertexts is to hide the position of the potential 1 decryption, thereby not leaking the position of the lowest bit differing between v_i and v_j.

Steps 2 and 3 implement a functionality which we call $\mathsf{Eval}(C_i, v_j)$ from now on.

4.2 Secure Comparisons Between Two Malicious Adversaries

Fischlin's protocol is only secure against semi-honest adversaries. However, one or *even both* parties may have behaved maliciously during comparison. Both suppliers S_i and S_j may submit different bids to distinct comparisons and supplier S_j could just encrypt any result of their choice using S_i's public key. That is, Fischlin's protocol does not ensure that $res_{i,j}$ has been computed according to the protocol specification and the fixed inputs of the suppliers.

We tackle this problem by, first, requiring both S_i and S_j to commit to their own input, simply by publishing GM encryptions C_i, C_j of v_i, v_j with their public key including a proof of knowledge of the plaintext. During comparison, S_j will prove to a judge A in zero-knowledge that S_j used the same value v_j in $C_{i,j}$ as in commitment C_j, and that S_j has performed homomorphic computation of $res_{i,j}$ according to Fischlin's algorithm. Therewith, S_i is sure that $res_{i,j}$ contains the result of comparing inputs behind ciphertexts C_i and C_j.

In the following description, we allow parties to either *publish* data or to send data from one to another. In reality, one could use the blockchain's broadcast feature to efficiently and reliably publish data to all parties or to just send a private (automatically signed) message, see Section 2.2.

Details First, party S_i commits to v_i by publishing $\{pk_i^{\mathsf{GM}}, C_i = \mathsf{Enc}^{\mathsf{GM}}_{pk_i^{\mathsf{GM}}}(v_i)\}$, and party S_j commits to v_j by publishing $\{pk_j^{\mathsf{GM}}, C_j = \mathsf{Enc}^{\mathsf{GM}}_{pk_j^{\mathsf{GM}}}(v_j)\}$. Then, S_i and S_j compare their v_i, v_j following Fischlin [21]'s homomorphic circuit evaluation above. After S_j has computed $res_{i,j}$, S_j additionally computes a ZK proof $P^{\mathsf{eval}}_{i,j}$ as follows.

1. S_j adds $C_{i,j}$ and random coins for both the shuffle of $res_{i,j}$ and the AND-homomorphic embeddings to initially empty proof $P^{\mathsf{eval}}_{i,j}$.
   Let $v_{j,\ell}$ be the ℓ^{th} bit of v_j. Let $(C_j)_\ell$ be the ℓ^{th} ciphertext of GM commitment C_j, i.e., the encryption of $v_{j,\ell}$ (the ℓ^{th} bit of v_j). Let $(C_{i,j})_\ell$ be the ℓ^{th} ciphertext of $C_{i,j}$.
2. Let λ'' be the soundness parameter of our ZK proof. S_j flips $\eta \cdot \lambda''$ coins $\delta_{\ell,m}$, $1 \leq \ell \leq \eta, 1 \leq m \leq \lambda''$.
3. S_j computes $\eta \cdot \lambda''$ encryptions $\gamma_{\ell,m} \leftarrow \mathsf{Enc}^{\mathsf{GM}}_{pk_j^{\mathsf{GM}}}(\delta_{\ell,m})$ and $\gamma'_{\ell,m} \leftarrow \mathsf{Enc}^{\mathsf{GM}}_{pk_i^{\mathsf{GM}}}(\delta_{\ell,m})$ and appends them to proof $P^{\mathsf{eval}}_{i,j}$.

4. S_j also computes $\eta \cdot \lambda''$ products $\Gamma_{\ell,m} = (C_j)_\ell \cdot \gamma_{\ell,m} \bmod n_j$ and $\Gamma'_{\ell,m} = (C_{i,j})_\ell \cdot \gamma'_{\ell,m} \bmod n_i$ and appends them to proof $P^{\mathsf{eval}}_{i,j}$. A product $\Gamma_{\ell,m}$ is an encryption of $\delta_{\ell,m} \oplus v_{j,\ell}$ under key pk^{GM}_j, and $\Gamma'_{\ell,m}$ is an encryption of $\delta_{\ell,m} \oplus v_{j,\ell}$ under key pk^{GM}_i.
5. S_j sends $P^{\mathsf{eval}}_{i,j}$ to judge A.
6. Our ZK proof can either be interactive or non-interactive. We first consider the interactive version of our proof. Here, A sends back the challenge h, a sequence of $\eta \cdot \lambda''$ bits $b_{\ell,m}$, to S_j.
7. If $b_{\ell,m} = 0$, S_j sends plaintext and random coins of $\gamma_{\ell,m}$ and $\gamma'_{\ell,m}$ to A. If $b_{\ell,m} = 1$, S_j sends plaintext and random coins of $\Gamma_{\ell,m}$ and $\Gamma'_{\ell,m}$ to A.

The non-interactive version of our proof is a standard application of Fiat-Shamir's heuristic [20] to Σ-protocols and imposes slight changes to steps 5 to 7. So, let $h = H((\gamma_{1,1}, \gamma'_{1,1}, \Gamma_{1,1}\Gamma'_{1,1}), \ldots, (\gamma_{\eta,\lambda''}, \gamma'_{\eta,\lambda''}, \Gamma_{\eta,\lambda''}, \Gamma'_{\eta,\lambda''}), C_i, C_j, C_{i,j})$ for random oracle $H : \{0,1\}^* \rightarrow \{0,1\}^{\eta \cdot \lambda''}$. Instead of sending $P^{\mathsf{eval}}_{i,j}$ to A, receiving the challenge, and replying to the challenge, S_j parses h as a series of $\eta \cdot \lambda''$ bits $b_{\ell,m}$. S_j does not send plaintexts and random coins of either $(\gamma_{\ell,m}, \gamma'_{\ell,m})$ or $(\Gamma_{\ell,m}, \Gamma'_{\ell,m})$ as above to A, but simply appends them to $P^{\mathsf{eval}}_{i,j}$ and then sends $P^{\mathsf{eval}}_{i,j}$ to A. In practice, we implement H by a cryptographic hash function.

So in conclusion, S_j sends proof $P^{\mathsf{eval}}_{i,j}$ to judge A who has to verify it. Note that $P^{\mathsf{eval}}_{i,j}$ contains ciphertext $C_{i,j}$ of S_j's input v_j under S_i's public key. The proof is zero-knowledge for judge A and very efficient, but must not be shared with party S_i. A's verification steps are as follows:

8. Judge A verifies that homomorphic computations for $res_{i,j}$ have been computed correctly, according to $C_{i,j}$,C_j, and random coins of $res_{i,j}$'s shuffle, simply by re-performing the computation.
9. For $\ell = \{1, \ldots, \eta\}$ and $m = \{1, \ldots, \}$, A verifies that homomorphic relations between $(C_i)_\ell, \gamma_{\ell,m}, \Gamma_{\ell,m}$ as well as for $(C_{i,j})_\ell, \gamma'_{\ell,m}, \Gamma'_{\ell,m}$ hold.
10. For each triple of plaintext, random coins, and ciphertexts of *either* $\gamma_{\ell,m}$ and $\gamma'_{\ell,m}$ *or* $\Gamma_{\ell,m}$ and $\Gamma'_{\ell,m}$, A checks that ciphertext results from the plaintext and random coins and that the plaintexts are the same.
11. If all checks pass, the judge A outputs $\top$, else $\perp$.

If A outputs $\top$, S_i decrypts $res_{i,j}$ and learns the outcome of the comparison, i.e., whether $v_i > v_j$.

Steps 1 to 7 implement a functionality that we call $\mathsf{ProofEval}(C_i, C_j, C_{i,j}, res_{i,j}, v_j)$ from now on. $\mathsf{ProofEval}$ is executed by S_j and uses commitments C_i and C_j and S_j's input v_j and outputs $\{C_{i,j}, res_{i,j}\}$ of $\mathsf{Eval}(C_i, v_j)$. Similarly, steps 8 to 11 realize functionality $\mathsf{VerifyEval}(P^{\mathsf{eval}}_{i,j}, res_{i,j}, C_i, C_j)$. Executed by judge A, it outputs either $\top$ or $\perp$.

Lemma 1. *The above scheme of computing and verifying proof $P^{\mathsf{eval}}_{i,j}$ with* $\mathsf{ProofEval}$ *and* $\mathsf{VerifyEval}$ *is a ZK proof of knowledge of v_j, such that $C_j = \mathsf{Enc}^{\mathsf{GM}}_{PK_j}(v_j)$, $\{C_{i,j}, res_{i,j}\} = \mathsf{Eval}(C_i, v_j)$, and if it is performed in λ'' rounds, the probability that S_j has cheated, but A outputs $\top$, is $2^{-\lambda''}$.*

Proof. As completeness follows directly from our description, we focus on soundness (extractability) and zero-knowledge.

(1) Knowledge Soundness. Judge A can extract v_j from S_j with rewinding access. Let $tr1(C_{i,j}, res_{i,j}, \gamma_{\ell,m}, \gamma'_{\ell,m}, \Gamma_{\ell,m}, \Gamma'_{\ell,m}, b_{\ell,m}, ...)$ be the trace of the first execution of $P^{\mathsf{eval}}_{i,j}$. Then judge A rewinds S_j to Step 5 and continues the protocol. Let $tr2(C_{i,j}, res_{i,j}, \gamma_{\ell,m}, \gamma'_{\ell,m}, \Gamma_{\ell,m}, \Gamma'_{\ell,m}, b_{\ell,m}, ...)$ be the trace of the second execution of $P^{\mathsf{eval}}_{i,j}$. If $tr1(b_{\ell,m}) = 0$ and $tr2(b_{\ell,m}) = 1$, then A learns $tr1(\delta_{\ell,m})$ and $tr2(\delta_{\ell,m} \oplus v_{j,\ell})$. Therewith, A computes $v_{j,\ell}$. As $v_{j,\ell}$ can be extracted, our Σ-protocol achieves special soundness. With challenge length λ'' for each bit of v_j, it is moreover a proof of knowledge with knowledge error $2^{-\lambda''}$ [14].

(2) Zero-Knowledge. Intuitively, the auctioneer learns nothing from the opening of either $\gamma_{\ell,m}$ and $\gamma'_{\ell,m}$ or $\Gamma_{\ell,m}$ and $\Gamma'_{\ell,m}$, since the plaintext value is always chosen uniformly random due to the uniform distribution of $\delta_{\ell,m}$. More formally, in the interactive case, we can construct a simulator $\mathsf{Sim}^{A(\{C_i,C_j\})}_{P^{\mathsf{eval}}_{i,j}}(res_{i,j})$ with rewinding access to judge $A(\{C_i,C_j\})$ following a standard simulation paradigm [27]. This ensures that we can construct a simulation of the ZK proof in the malicious model of secure computation even if bid v_j does not correspond to ciphertext $C_{i,j}$ and commitments C_i, C_j, since the simulator generates an accepting, indistinguishable output even if v_j is unknown. In the non-interactive case with Fiat-Shamir's heuristic, our ZK proof is secure in the random oracle model. □

Note: Our proof here shows something stronger than required by the general auction protocol. We show our ZK proof to be secure even against malicious verifiers. However, auctioneer A, serving as the judge in the main protocol, is supposed to be semi-honest.

5 Blockchain Auction Protocol

After having presented our core technique for secure comparisons, we now turn to our main auction protocol Strain. Imagine that, at some point, A announces a new auction and uploads a smart contract to the blockchain. The smart contract is very simple and allows parties to comfortably exchange messages as mentioned before. The contract is signed by sk_A, so everybody understands that this is a valid procurement auction.

Overview. With the smart contract posted, the actual auction starts. In Strain, each supplier must first publicly commit to their bid. For this, we use a new verifiable commitment scheme which allows a majority of honest suppliers to open other suppliers' commitments. Therewith, we can at any time open commitments of malicious suppliers blocking or aborting the auction's progress.

After suppliers have committed to their bids (or after a deadline has passed), the protocol to determine the winning bid starts. Strain uses the new comparison technique from Section 4.2 to compare bids of any two parties. Auctioneer A serves as the judge. However, using our new comparison in the auctions turns out to be a challenge. Recall that, when S_i and S_j compare their bids, only S_i knows the outcome of the comparison, but nobody else. We therefore augment our comparison such that S_i can publish the outcome of the comparison, together with a (zero knowledge) proof of correctness.

To improve readability, we present Strain without optional pseudonymity and postpone pseudonymity to Section 5.4. For now, assume that a subset $\mathcal{S}' \subset \mathcal{S}, |\mathcal{S}'| = s' \leq s$ participates in the auction. Either a pseudonymous subset or all suppliers participate.

5.1 Verifiable Key Distribution for Commitments

To be able to commit to their bids, suppliers in Strain initially distribute their keying material. In the following, we devise a new key distribution technique for our specific setting. It permits supplier S_i to publish a GM public key and verifiably secret share the corresponding secret key. The crucial property of our key distribution is that a majority of honest suppliers can decrypt ciphertexts encrypted with S_i's public key. To then later commit to a value v_i, S_i encrypts v_i with their public key. For ease of exposition, we describe our key distribution with *s-out-of-s* threshold secret sharing. However, we stress that many different schemes exist for *s'-out-of-s* sharing modulo an RSA integer. For example, one could adopt and employ the schemes by Desmedt and Frankel [16] or Katz and Yung [25]. See also Shoup [35] for an overview.

Key Distribution Each supplier S_i generates a GM key pair $(pk_i^{\mathsf{GM}} = (n_i = p_i \cdot q_i, z_i = n_i - 1), sk_i^{\mathsf{GM}} = \frac{(p_i-1)\cdot(q_i-1)}{4})$. To allow other suppliers S_j to open commitments from supplier S_i, S_i first computes a non-interactive ZK proof P_i^{Blum} that n_i is a Blum integer, see Blum [5] for details. Moreover, S_i computes secret shares of $\frac{(p_i-1)\cdot(q_i-1)}{4}$ for all suppliers as follows: S_i computes $s'-1$ random shares $r_{i,1},\ldots,r_{i,s'-1} \xleftarrow{\$} \{0, (p_i-1)\cdot(q_i-1)\}$ such that $\sum_{j=1}^{s'-1} r_{i,j} = \frac{(p_i-1)\cdot(q_i-1)}{4} \mod (p_i-1)\cdot(q_i-1)$. This can easily be converted into a threshold scheme using Shamir's secret shares where τ is the threshold for reconstructing a secret. Supplier S_i computes signature $\mathsf{sig}_{sk_i}(r_{i,j})$ and encrypts share $r_{i,j}$ and signature $\mathsf{sig}_{sk_i}(r_{i,j})$ for supplier S_j using S_j's public key pk_j. Finally, S_i broadcasts resulting $s'-1$ ciphertexts of share and signature pairs as well as pk_i^{GM} and P_i^{Blum} on the blockchain.

All suppliers can send their broadcasts in parallel, requiring only one block latency.

Key Verification All s' participating suppliers start a sub-protocol to verify all s' public keys pk_i^{GM}. For each pk_i^{GM}:

1. All suppliers check proof P_i^{Blum}. If supplier S_j fails to verify the proof, S_j publishes $(i, \perp)$ on the blockchain.
2. Each supplier S_j selects a random $\rho_{i,j} \xleftarrow{\$} \mathbb{Z}_{n_i}^*$ and employs a traditional commitment scheme commit to commit to $\rho_{i,j}$. That is, each supplier S_j publishes $\mathsf{commit}(\rho_{i,j})$ on the blockchain.
3. After a deadline has passed, all suppliers open their commitments, by publishing $\rho_{i,j}$ and the random nonce used for the commitment.
   All suppliers compute $x_i = \sum_{j \neq i} \rho_{i,j} \mod n_i$ and $y_i = x_i^2$.
4. Each supplier S_j raises y_i to their share $r_{i,j}$ of $\frac{(p_i-1)\cdot(q_i-1)}{4}$ and publishes $\gamma_{i,j} = y_i^{r_{i,j}}$ on the blockchain. S_j also raises z_i to their $r_{i,j}$, i.e., $\zeta_{i,j} = z_i^{r_{i,j}}$. S_j then prepares a non-interactive ZK proof $P_{i,j}^{\mathsf{DLOG}}$ of statement $\log_{y_i} \gamma_{i,j} = \log_{z_i} \zeta_{i,j}$, see Appendix A for details. Supplier S_j publishes $\{\gamma_{i,j}, \zeta_{i,j}, P_{i,j}^{\mathsf{DLOG}}\}$ on the blockchain.
5. Finally, all $s'-1$ suppliers verify soundness of pk_i^{GM}. Each supplier S_j computes $b_i = \prod_{j \neq i} \gamma_{i,j} = y_i^{\sum_{j=1}^{s'-1} r_{i,j}} = y_i^{\frac{(p_i-1)\cdot(q_i-1)}{4}} \mod n_i$ and $b'_i = \prod_{j \neq i} \zeta_i = z_i^{\sum_{j=1}^{s'-1} r_{i,j}} =$

$z_i^{\frac{(p_i-1)\cdot(q_i-1)}{4}}$ mod n_i. If S_j detects that $b_i \neq 1$ or $b'_i \neq -1 \bmod n_i$, S_j publishes $(i,\perp)$ on the blockchain. Supplier S_j also checks $s'-1$ proofs $P_{i,k}^{\mathsf{DLOG}}$. If one of the κ rounds outputs $\perp$ during verification, S_j publishes $(k,\perp)$ on the blockchain.

Lemma 2. *Let n_i be a Blum integer and α the sum of shares distributed by S_i. If no honest supplier publishes $(i,\perp)$, then $Pr[\alpha \neq \frac{(p_i-1)\cdot(q_i-1)}{4}] \in O(2^{-\lambda})$.*

Proof. Let y_i have no roots in $\mathbb{Z}_{n_i}$ dividing $\frac{(p_i-1)(q_i-1)}{4}$. For uniformly chosen y_i, this happens with overwhelming probability $\in O(1-2^{-\lambda})$. As $y_i \in QR_{n_i}$, it has order $\frac{(p_i-1)(q_i-1)}{4}$. So, $b_i = 1$ implies (I) $\alpha \bmod \frac{(p_i-1)(q_i-1)}{4} = 0$; further, since $z_i = -1 \bmod n_i$, we have $z_i^{\frac{(p_i-1)(q_i-1)}{4}} \in \{-1,1\}$, and so (II) $z_i^{\frac{(p_i-1)(q_i-1)}{2}} = 1$. Hence $b'_i = -1$ implies $\alpha \bmod \frac{(p_i-1)(q_i-1)}{2} \neq 0$. From (I) and (II), we conclude $(\alpha \bmod \frac{(p_i-1)(q_i-1)}{4}) \bmod 2 = 1$. However, all those values will serve as private keys in GM encryption. □

In conclusion, supplier S_i can verify whether their shares for supplier S_j's secret key sk_j^{GM} matches public key pk_j^{GM}. Therewith, an honest majority of suppliers will later be able to open commitments of malicious suppliers trying to block the smart contract or cheat.

Excluding malicious suppliers Strain's key verification easily allows detection and exclusion of malicious suppliers. First, as all suppliers can verify proofs P_i^{Blum} and $P_{i,j}^{\mathsf{DLOG}}$ of a supplier S_i, honest suppliers can exclude S_i or S_j from further participating in the protocol in case of a bad proof.

Moreover, following our assumption of up to τ malicious suppliers, Strain allows to systematically detect and exclude malicious suppliers. Supplier S_j will reconstruct $b_i = 1$ and $b'_i = -1$ from the set of secret shares $(\gamma_{i,j},\zeta_{i,j})$. If no subset reconstructs the correct plaintexts, S_j deduces that distributor S_i is malicious and excludes S_i. Otherwise, S_j checks that each supplier S_k's share reconstructs the correct plaintext. If any does not, S_j asks S_k publicly on the blockchain to reveal their exponent $r_{i,k}$ and signature $\mathsf{sig}_{sk_i}(r_{i,k})$. If at least $\tau+1$ suppliers ask S_k to reveal, S_k will reveal, and honest suppliers can detect whether S_k should be excluded (signature does not verify or exponent does not match secret shares) or S_i (signature verifies and exponent matches secret shares).

5.2 Determining the Winning Bid

Strain's main protocol Π_{Strain} to determine the winning bid is depicted in Algorithm 2. Within Algorithm 2, we use three ZK proofs as sub-protocols.

- $\mathsf{ProofEnc}(C_i, v_i)$ proves in zero-knowledge the knowledge of v_i, such that $C_i = \mathsf{Enc}_{PK_i}^{\mathsf{GM}}(v_i)$. For an exemplary implementation we refer to Katz [24].
- $\mathsf{ProofEval}(C_j,C_i,C_{i,j},res_{i,j},v_j)$ has been introduced in Section 4.2.
- $\mathsf{ProofShuffle}(shuffle_{i,j},res_{i,j})$ proves in zero-knowledge the knowledge of a permutation Shuffle with $shuffle_{i,j} = \mathsf{Shuffle}(res_{i,j})$. There exist a large number of implementations of shuffle proofs. For one that is straightforward to adapt to GM encryption, see Ogata et al. [31]. Using this technique, one can even create shuffles with a restricted structure [32]. That is, the shuffle is only chosen from a pre-defined subset of all possible shuffles. In our case this is necessary, since we do not randomly shuffle all GM ciphertexts, but only AND-homomorphic blocks of GM ciphertexts.

```
1  for $i=1$ to $s'$ do
2      $S_i$: publish $\{C_i \leftarrow \mathsf{Enc}^{\mathsf{GM}}_{PK_i}(v_i), P^{\mathsf{enc}}_i \leftarrow \mathsf{ProofEnc}(C_i, v_i)\}$ on blockchain;
3  for $i=1$ to $s'$ do
4      forall $j \neq i$ do
5          $S_j: \{C_{i,j}, res_{i,j}\} \leftarrow \mathsf{Eval}(C_i, v_j)$;
6          $S_j: P^{\mathsf{eval}}_{i,j} \leftarrow \mathsf{ProofEval}(C_j, C_i, C_{i,j}, res_{i,j}, v_j)$;
7          $S_j$: publish $\{\mathsf{Enc}_{pk_A}(P^{\mathsf{eval}}_{i,j}), res_{i,j}\}$ on blockchain;
8          $A$: publish $\mathsf{VerifyEval}(P^{\mathsf{eval}}_{i,j}, res_{i,j}, C_i, C_j)$ on blockchain;
9          $S_i: bitset_{i,j} = \mathsf{Dec}^{\mathsf{AND}}_{pk^{\mathsf{GM}}_j}(res_{i,j})$;
10         $S_i: shuffle_{i,j} \leftarrow \mathsf{Shuffle}(res_{i,j})$;
11         $S_i: P^{\mathsf{shuffle}}_{i,j} \leftarrow \mathsf{ProofShuffle}(shuffle_{i,j}, res_{i,j})$;
12         $S_i$: let $\gamma_{\ell,m} \leftarrow \mathsf{Enc}^{\mathsf{GM}}_{PK_i}(\beta_{\ell,m}) \in shuffle_{i,j}$ be the shuffled ciphertexts
13         with their random coins $r_{\ell,m}$. Publish $\{P^{\mathsf{shuffle}}_{i,j}, shuffle_{i,j}, \beta_{\ell,m}, r_{\ell,m}\}$;
```

Algorithm 2: Blockchain auction protocol Π_{Strain}

ZK proofs ProofEnc and ProofShuffle are verified by all suppliers active in the auction, and, hence, verification is not explicitly shown. ZK proof ProofEval, however, is verified only by the semi-honest judge and auctioneer A.

Let $\eta \ll \lambda$ be a public system parameter determining the bit length of each bid. That is, any bid $v_i = v_{i,1}...v_{i,\eta}$ can take values from $\{0,...,2^\eta - 1\}$.

Π_{Strain} starts with each supplier S_i committing to their bid v_i by publishing GM-encryption $C_i = (\mathsf{Enc}^{\mathsf{GM}}_{pk^{\mathsf{GM}}_i}(v_{i,1}), \ldots, \mathsf{Enc}^{\mathsf{GM}}_{pk^{\mathsf{GM}}_i}(v_{i,\eta}))$ on the blockchain. Recall that all messages on the blockchain are automatically signed by their generating party.

After a deadline has passed, suppliers determine index w of winning bid v_w by running our maliciously-secure comparison mechanism of Section 4.2. Any pair (S_i, S_j) of suppliers computes the comparison and publishes the result on the blockchain.

Specifically, after judge/auctioneer A has published whether S_j's computation of $C_{i,j}$ corresponds to S_j's commitment C_j, supplier S_i can decrypt $res_{i,j}$ and learn whether $v_i > v_j$. To publish whether $v_i > v_j$, S_i shuffles $res_{i,j}$ to $shuffle_{i,j}$, publishes a ZK proof of shuffle, and publicly decrypts $shuffle_{i,j}$. Therewith, everybody can verify $v_i > v_j$. If A has output $\top$, if the proof of shuffle is correct, and if $shuffle_{i,j}$ contains exactly a single 1, then $v_i > v_j$. If A has output $\top$, the shuffle proof is correct, and if $shuffle_{i,j}$ contains only 0s, then $v_i > v_j$.

A supplier S_i is the winner of the auction, if all their shuffles prove that their bid is the lowest among all suppliers. S_i can prove this by opening the plaintext and random coins of $shuffle_{i,j}$. If $v_i \leq v_j$, at least one plaintext in each consecutive sequence of λ' plaintexts is 0. If $v_i > v_j$, a consecutive sequence of λ' plaintexts is 1. Strain concludes with auction winner S_w revealing bid v_w and a plaintext equality ZK proof that commitment C_w is for v_w to auctioneer A.

5.3 Latency Evaluation

The performance of any interactive protocol or application running on top of a blockchain is dominated by block interval times. With today's block interval times in the order of several seconds, protocols requiring a lot of party interaction significantly increase the protocol's total latency, i.e., its total run time. A secure auction protocol with high latency is useless in many scenarios with automated, short-living auctions.

Table 1. Execution time for Strain's main cryptographic operations

$\mathsf{Enc}^{\mathsf{GM}}$	$\mathsf{Dec}^{\mathsf{GM}}$	$\mathsf{Enc}^{\mathsf{AND}}$	$\mathsf{Dec}^{\mathsf{AND}}$	$\mathsf{ProofEnc}$	$\mathsf{VerifyEnc}$	Eval	$\mathsf{ProofEval}$	$\mathsf{VerifyEval}$
0.08 ms	46 ms	60 ms	980 ms	10 ms	9 ms	390 ms	107 ms	15 ms

$\mathsf{ProofDLOG}$	$\mathsf{VerifyDLOG}$	$\mathsf{ProofShuffle}$	$\mathsf{VerifyShuffle}$
154 ms	339 ms	633 ms	198 ms

As a crucial performance metric, we therefore investigate Strain's latency. As key distribution is a setup-like initial process, necessary only once, and independent of actual auctions, we focus on Π_{Strain}'s latency.

Asymptotic Analysis In Algorithm 2, Π_{Strain} starts in Line 2 by all suppliers sending a commitment to their bid together with P^{enc}. There is no interactivity between by suppliers, so all suppliers can send in parallel, requiring one block latency. After that first block has been mined, all suppliers send their P^{eval} for each other supplier to A, lines 5 to 7. Each supplier can send all P^{eval} for all other suppliers at once $(s' \cdot (s'-1)$ hash values of the PBB). Again, there is no interactivity between suppliers, so all suppliers send in parallel in one block. Then, auctioneer A sends all $\mathsf{VerifyEval}$ for all comparisons at once (1 hash), Line 8, in another block. In a final block, all suppliers disclose in parallel (s' hashes) their shuffles, random coins, and corresponding P^{shuffle} (Line 13).

In conclusion, one run of Π_{Strain} requires a total of 4 blocks latency: 1 block for suppliers to commit, and then 3 blocks for core comparisons and computation of the winning bid. This number is constant in both bit length η of each bid and the number of suppliers s. In contrast, practical MPC protocols require at least $\Omega(\eta)$ rounds. Although Fischlin's protocol only evaluates a circuit of constant multiplicative depth, it is capable of evaluating a comparison due to the shuffle of the ciphertexts before decryption.

Prototypical Implementation To indicate its real-world practicality, we have prototypically implemented and benchmarked Π_{Strain}'s core cryptographic operations in Python. The source code is available for download [36].

In our measurements, we have set bid length η to 32 bit, allowing for either large bids or very fine-grained bids. For good security, we set the bit length of primes for Blum integers n to $|p| = |q| = 768$ bit. To achieve a small probability for soundness errors of 2^{-40}, we choose $\lambda' = \lambda'' = \kappa = 40$. We have implemented the non-interactive versions of our ZK proofs and used SHA256 as hash function. All experiments were performed on a mostly idle Linux laptop with Intel i7-6560U CPU, clocked at 2.20 GHz. Our prototypical implementation uses only one core of the CPU's four virtual cores available, but we emphasize that our cryptographic operations can run independently in parallel, e.g., for each supplier. They scale linearly in the number of (virtual) cores.

Table 1 summarizes timings for cryptographic operations. All values are the average of ten runs. Relative standard deviation for each average was low with less than 9%.

Eval. Inside the main for-loop in Π_{Strain}, operation Eval and computation of ZK proof $\mathsf{ProofEval}$ for A take roughly 0.5 s. Taking Ethereum's 15 s blockchain interval, a supplier could compute proofs for up to 30 other suppliers using a single core. Again, with the availability of x many cores, this number multiplies by x.

Auctioneer A executes $\mathsf{VerifyEval}$ for which we have implemented verification of homomorphic relations between Cs, γs, and Γs and (expensive) verification of encryptions for given random coins. Yet, verification is just (re-)computing GM encryptions with fixed coins which are included in P^{Eval}. As you can see, VerifyEval is very fast (15 ms), allowing roughly thousand comparisons in one Ethereum block interval.

ProofShuffle. As a supplier needs to compute ProofShuffle, we have modified Ogata et al. [31]'s standard shuffle to our setting. Very briefly, the idea of proving $shuffle$ to be a re-encrypted shuffle of res in zero-knowledge is to generate κ re-encrypted intermediate shuffles $shuffle'_i$ of res. For each intermediate shuffle $shuffle'_i$, the verifier ask *either* to show the permutation between res and $shuffle'_i$ and all random coins used during re-encryption *or* to show the permutation between $shuffle'_i$ and $shuffle$ and random coins used during re-encryption. Recall that re-encryption in our setting is simply multiplication with a random quadratic residue. Computing ProofShuffle is an expensive operation, taking 600 ms. Thus, in our non-optimized implementation, a supplier could prepare ≈ 25 proofs of shuffle per CPU core in one block interval. We stress that our modification to Ogata et al. [31]'s shuffle is straightforward and leave the design of more performance optimized shuffles for future work.

Note that Enc_{pk_A} is not GM encryption, but a regular hybrid encryption for auctioneer A, e.g., AES-ECC. As hybrid encryption is extremely fast compared to computation of our ZK proofs, we ignore it in our latency analysis.

ProofEnc. For the initial commitment of each supplier, we have adopted Katz [24]'s standard technique for proving plaintext knowledge to GM encryption. Again, we only summarize the main idea of our (straightforward) adoption. To prove knowledge of a single plaintext bit m, encrypted to GM ciphertext $C = r^2 \cdot z^m$, prover and verifier engage in a κ-round Σ-protocol. In each round i, the prover randomly chooses r_i and sends $A_i = r_i^4$ to the verifier. The verifier replies by sending random bit q_i, and the prover concludes the proof by sending $R_i = r^{q_i} \cdot r_i$. The verifier accepts the round, if $R_i^4 = A_i \cdot C^{2 \cdot q_i}$. For our evaluation, we have implemented a non-interactive version of this Σ-protocol. Both, computation of the ZK proof (VerifyEnc) as well as its verification (VerifyEnc) are extremely fast, taking only 10 ms for all rounds and all encrypted bits together. Note that computation of this proof is independent of the number of suppliers and has to be performed only once per auction.

ProofDLOG. Albeit part of only the initial key distribution phase, we also include computation times for computation and verification of proof P^{DLOG}. In Table 1, ProofDLOG denotes the algorithm computing proof P^{DLOG}, and VerifyDLOG is the algorithm verifying P^{DLOG}, see Appendix A for details. These computations are efficient: within one block interval, a supplier can generate ≈ 100 shares for other suppliers and verify ≈ 45.

Having in mind that our Python implementation is prototypical and not optimized for speed, we conclude that Π_{Strain}'s cryptographic operations are very efficient, allowing Strain's deployment in many short-term auction scenarios with dozens of suppliers.

5.4 Optional: Preparation of Pseudonyms

To pseudonymously place a bid in Strain, suppliers must decouple their blockchain transactions from their regular key pair (pk_i, sk_i). Ideally for each auction, supplier S_i generates a fresh random key pair (rpk_i, rsk_i) for bidding. In practice, e.g., with Ethereum, this turns out to be a challenge. To interact with a smart contract, S_i must send a transaction. Yet, to mitigate DoS attacks in Ethereum, transactions cost money of the blockchain's virtual currency. If a fresh key pair wants to send a transaction, someone must send funds to it. S_i cannot send funds to their fresh key, as this would create a visible link between S_i and (rpk_i,rsk_i).

Our idea is that A sends funds to keys that have previously been registered. To do so, S_i will register their fresh key pair (rpk_i,rsk_i) using a blind RSA signature.As a result, S_i has received a valid signature sig'_i of its random key rpk_i. Besides s, the adversary learns nothing about the rpk_is.

All suppliers send their blinded rpk_i in parallel, and A then replies with blind signatures in parallel, too. Communication latency is constant in the number of suppliers s. Note that all suppliers must request a blind signature for a random rpk_i, regardless of whether a supplier is interested in an auction or not. If a supplier does not request a blind signature, the adversary knows that they will not participate in the auction.

After a supplier has recovered their key pair (rpk_i,rsk_i), they broadcast it to the blockchain. All suppliers run a Dining Cryptographer network in parallel, see Appendix C. A supplier S_i interested in participating in the auction will broadcast (rpk_i, sig'_i), and a supplier not interested will broadcast 0s.

As a result of the DC network, everybody knows fresh, random public keys of a list of suppliers participating in the auction. Due to A's signature, everybody knows that these suppliers are valid suppliers, but nobody can link a key rpk_i to supplier S_i. Starting from now, only suppliers interested in the auction will continue by submitting a bid and determining the winning bid. Running a DC network is communication efficient. That is, all suppliers submit their s powers of rpk_i in parallel in $O(1)$ blocks.

Finally, A transfers money to each public key rpk_i, just enough such that suppliers can use their (rpk_i,rsk_i) keys to interact with the smart contract. Supplier S_i will use their new key pair (rpk_i,rsk_i) to pseudonymously participate in the rest of the protocol.

Security Analysis. For space reasons, we move the security analysis to Appendix B.

6 Related Work

MPC Current maliciously-secure protocols of practical performance for *more than two* parties are based on secret shares [2]. They require at least as many rounds of interaction as the multiplicative depth of the circuit evaluated [28]. For comparisons this is the bit length η of the bids. Even for tiny auctions this will exceed Strain's total of four blocks. Constant-round MPC protocols, e.g. [28, 29], exceed four blocks already in their pre-computation phase before any comparison has taken place. Benhamouda et al. [4] present an MPC auction protocol running on Hyberledger Fabric. The underlying primitive is Yao's MPC requiring $\Omega(\eta)$ rounds of interactivity, and it does not provide security against malicious bidders (Strain does).

Dedicated auction protocols There exists a large number of specialized secure auctions protocols; for a survey see Brandt [9]. Among them, the one that compares closely to Strain is Brandt's very own auction protocol [8]. There, suppliers compute the winner of the auction, as with Strain, and the protocol requires a constant number of party interactions – as does Strain. However, Brandt encodes bids in unary notation making the protocol impractical for all but the simplest auctions. Instead, Strain encodes bids in binary notation, thus enabling efficient auctions for realistic bid values. Brandt cannot guarantee output delivery which Strain does and which we consider crucially important in practice. Brandt claims full privacy in the malicious model, but formal verification has shown that this does not necessarily hold, cf. Dreier et al. [17].

Fischlin [21] also presents a variant of his main protocol which is secure against a malicious adversary. However, that variant requires an oblivious third party A providing a public/private key pair. All homomorphic computations in Fischlin's protocol are then performed under A's public key. Simulating A on the blockchain requires distributing the private key over multiple parties. As a result, one would need a secure, distributed computation of a Goldwasser-Micali key pair. Even for the case of RSA, this is complex and requires many rounds of interactions [6], rendering it impractical on a blockchain. Instead in Strain, each party creates its own key pair and only proves correct key sharing. Furthermore, even in case A's key has been set up, Fischlin's protocol still requires six rounds for each core comparison, whereas Strain requires only three (plus one for commitments) – a noticeable difference on the blockchain. We also stress that Fischlin's protocol targets a setup with 2 parties and cannot trivially be extended to multiple parties: 2 colluding malicious parties can convince oblivious party A of any outcome of the comparison they desire. In a multi-party setting, this allows an adversary to undermine the result of an auction, even after bids have been placed. Instead in this paper, we prove that Strain is secure against a collusion of up to τ suppliers.

Cachin [10] presents a protocol for secure auctions based on the Φ-hiding assumption. A variant secure against *one* malicious party (§3.3 in [10]) requires at least 7 blocks per comparison. Instead, Strain compares in only three blocks and supports both parties to be malicious during comparisons. Moreover similar to Fischlin [21]'s protocol, it is not trivial to extend [10] to support more than one fully malicious party. The auction protocol by Naor et al. [30] requires another trusted party (the auction issuer), is based on garbled circuits, therefore communication and computation inefficient, and secure only in the semi-honest model. Damgård et al. [15]'s auction considers the very different scenario of comparing a secret value m with a public integer m. The fully malicious version of their auction (§5.3 in [15]) only copes with up to one fully malicious party. Another version (§5.1 in [15]) addresses comparing secret inputs m and x, but only with semi-honest security.

7 Conclusion

Strain is a new protocol for secure auctions on blockchains. Strain allows, for the first time, to execute a sealed bid auction on a blockchain, secure against malicious bidders, with optional bidder anonymity, and guaranteed output delivery. Strain is efficient, and its main auction part runs in a constant number of blocks. Such low latency is crucial for practical adoption and a basis for a new implementation of sealed-bid auctions over blockchains where auction results can be observed by all participants.

Bibliography

[1] Accenture. How blockchain can bring greater value to procure-to-pay processes, 2017. `https://www.accenture.com`.

[2] D.W. Archer, D. Bogdanov, B. Pinkas, and P. Pullonen. Maturity and Performance of Programmable Secure Computation. *IEEE Security and Privacy*, 14(5):48–56, 2016.

[3] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *Symposium on Security and Privacy, Berkeley, CA, USA, 2014*, pages 459–474, 2014.
[4] F. Benhamouda, S. Halevi, and T. Halevi. Supporting Private Data on Hyperledger Fabric with Secure Multiparty Computation. In *International Conference on Cloud Engineering*, pages 357–363, 2018.
[5] M. Blum. Coin Flipping by Telephone. In *Advances in Cryptology: A Report on CRYPTO 81, Santa Barbara, California, USA, August 24-26*, pages 11–15, 1981.
[6] D. Boneh and M.K. Franklin. Efficient Generation of Shared RSA Keys (Extended Abstract). In *Proceedings of International Cryptology Conference*, CRYPTO, 1997.
[7] J. Bos and B. den Boer. Detection of Disrupters in the DC Protocol. In *EUROCRYPT*, pages 320–327, 1990.
[8] F. Brandt. Fully Private Auctions in a Constant Number of Rounds. In *Proceedings of the 7th International Conference on Financial Cryptography, FC 2003*, pages 223–238, 2003.
[9] F. Brandt. Auctions. In Burton Rosenberg, editor, *Handbook of Financial Cryptography and Security.*, pages 49–58. Chapman and Hall/CRC, 2010.
[10] C. Cachin. Efficient Private Bidding and Auctions with an Oblivious Third Party. In *Conference on Computer and Communications Security, Singapore*, pages 120–127, 1999.
[11] D. Chaum. The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.
[12] D. Chaum and T.P. Pedersen. Wallet Databases with Observers. In *Advances in Cryptology - CRYPTO '92, Santa Barbara, USA*, pages 89–105, 1992.
[13] G. Couteau, T. Peters, and D. Pointcheval. Encryption Switching Protocols. Cryptology ePrint Archive, Report 2015/990, 2015. `http://eprint.iacr.org/2015/990`.
[14] I. Damgård. On Σ-protocols, 2010. `http://www.cs.au.dk/~ivan/Sigma.pdf`.
[15] I. Damgård, M. Geisler, and M. Krøigaard. Efficient and Secure Comparison for On-Line Auctions. In *Information Security and Privacy Conference*, pages 416–430, 2007.
[16] Y. Desmedt and Y. Frankel. Shared generation of authenticators and signatures (extended abstract). In *CRYPTO, Santa Barbara, USA*, pages 457–469, 1991.
[17] J. Dreier, J.-G. Dumas, and P. Lafourcade. Brandt's fully private auction protocol revisited. *Journal of Computer Security*, 23(5):587–610, 2015.
[18] Ethereum. White Paper, 2017. `https://github.com/ethereum/wiki/wiki/`.
[19] Etherscan. The Ethereum Block Explorer, 2017. `https://etherscan.io/`.
[20] A. Fiat and A. Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO, Santa Barbara, USA*, pages 186–194, 1986.
[21] M. Fischlin. A Cost-Effective Pay-Per-Multiplication Comparison Method for Millionaires. In *CT Track – RSA Conference, San Francisco, USA*, pages 457–472, 2001.
[22] J.A. Garay, A. Kiayias, and N. Leonardos. The Bitcoin Backbone Protocol: Analysis and Applications. In *EUROCRYPT*, pages 281–310, 2015.
[23] S. Goldwasser and S. Micali. Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information. In *STOCS*, pages 365–377, 1982.
[24] J. Katz. Efficient and Non-malleable Proofs of Plaintext Knowledge and Applications. In *EUROCRYPT*, pages 211–228, 2003.
[25] J. Katz and M. Yung. Threshold Cryptosystems Based on Factoring. Cryptology ePrint Archive, Report 2001/093, 2001. `http://eprint.iacr.org/2001/093`.
[26] A.E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy, San Jose, USA*, pages 839–858, 2016.
[27] Y. Lindell. How To Simulate It – A Tutorial on the Simulation Proof Technique. Cryptology ePrint Archive, Report 2016/046, 2016. `http://eprint.iacr.org/2016/046`.
[28] Y. Lindell, B. Pinkas, N.P. Smart, and A. Yanai. Efficient Constant Round Multi-party Computation Combining BMR and SPDZ. In *CRYPTO*, 2015.
[29] Y. Lindell, N.P. Smart, and E. Soria-Vazquez. More Efficient Constant-Round Multi-party Computation from BMR and SHE. In *International Conference on Theory of Cryptography*, 2016.
[30] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *ACM Conference on Electronic Commerce*, pages 129–139, 1999.
[31] W. Ogata, K. Kurosawa, K. Sako, and K. Takatani. Fault tolerant anonymous channel. In *International Conference on Information and Communication Security*, pages 440–444, 1997.

[32] M.K. Reiter and X. Wang. Fragile mixing. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004*, pages 227–235, 2004.
[33] Reuters. Ukrainian ministry carries out first blockchain transactions, 2017. `https://www.reuters.com`.
[34] T. Sander, A.L. Young, and M. Yung. Non-Interactive CryptoComputing For NC^1. In *FOCS*, pages 554–567, 1999.
[35] V. Shoup. Practical threshold signatures. In *EUROCRYPT, Bruges, Belgium*, pages 207–220, 2000.
[36] Strain. Source Code, 2017. `https://github.com/strainprotocol/`.
[37] S. Tual. What are State Channels?, 2017. `https://www.stephantual.com`.
[38] University of Bristol. Multiparty computation with SPDZ online phase and MASCOT offline phase, 2017. `https://github.com/bristolcrypto/SPDZ-2`.
[39] M. Vukolic. The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication. In *Open Problems in Network Security*, pages 112–125, 2015.
[40] M. Waidner. Unconditional Sender and Recipient Untraceability in Spite of Active Attacks. In *EUROCRYPT, Houthalen, Belgium*, pages 302–319, 1989.
[41] M. Waidner and B. Pfitzmann. The Dining Cryptographers in the Disco: Unconditional Sender and Recipient Untraceability with Computationally Secure Serviceability. In *EUROCRYPT*, 1989.

A Proofs of DLOG Equivalence

As the DDH assumption holds in group $(\mathbb{J}_n, \cdot)$ for Blum integers n [13], we adopt standard ZK proofs of DLOG equivalence to our setting.

Let $y,z \in \mathbb{J}_n$ and z be a generator of group $(\mathbb{J}_n,\cdot)$. A prover knows an integer σ such that $y^\sigma = \gamma \bmod n$ and $z^\sigma = \zeta \bmod n$. For public values $\{y,z,\gamma,\zeta\}$, the prover wants to compute the statement $\log_y \gamma = \log_z \zeta$ to a verifier in zero-knowledge, i.e., without revealing any additional information about σ. This boils down to Chaum and Pedersen's ZK proof that $(y,z,Y=y^\sigma,Z=z^\sigma)$ is a DDH tuple [12]. The protocol runs in κ rounds.

In each round, *(1)* The prover computes $r \overset{\$}{\leftarrow} \mathbb{J}_n$ and sends $(t_1 = y^r, t_2 = z^r)$ to the verifier. *(2)* The verifier sends challenge $c \overset{\$}{\leftarrow} \mathbb{J}_n$ to the prover. *(3)* The prover sends $s = r + c\cdot\sigma$ to the verifier. *(4)* The verifier checks $y^s \overset{?}{=} t_1 \cdot Y^c \wedge z^s \overset{?}{=} t_2 \cdot Z^c$. If the check fails, the verifier outputs $\perp$.

We target non-interactive ZK proofs, so challenge c can be replaced in round $i \leq \kappa$ by a random oracle call $c = H(y,z,Y,Z,t_1,t_2,i)$ [20]. Let P^{DLOG} be an initially empty proof. For each round, the prover would add t_1,t_2, and s to P^{DLOG}, and then send P^{DLOG} to the verifier. Note that, if $z = -1 \bmod n$, as in our main protocol, then $z = -(1^2)$ is indeed a generator of $\mathbb{J}_n$. This ZK proof is secure in the random oracle model.

B Security Analysis

We now prove Theorem 1. Our proof is a simulation-based proof in the hybrid model [27]. In the hybrid model, simulator $\mathcal{S}$ generates messages of honest parties interacting with malicious parties and the trusted third party TTP. Since the simulator does not use inputs of honest parties (except for forwarding to the TTP which does not leak any information), it is ensured that the protocol does not reveal any information except the result, i.e., the output of the TTP. Messages generated by the simulator must be indistinguishable from messages in the real execution of the protocol.

Proof. Let $\mathcal{S}$ be the set of all suppliers and $\overline{\mathcal{S}}$ be the suppliers controlled by adversary $\mathcal{A}_1$. We prove $IDEAL_{\mathcal{F}_{Bid},\mathcal{S},\overline{\mathcal{S}}}(v_1,\ldots,v_s) \equiv REAL_{\Pi_{\mathsf{Strain}},\mathcal{A},\overline{\mathcal{S}}}(v_1,\ldots,v_s)$.

We either establish pseudonymous (broadcast) channels over the blockchain using the protocol of Section 5.4 or use regular authenticated channels.

I) In the first step of the protocol, honest suppliers $\mathcal{S} \setminus \overline{\mathcal{S}}$ commit to random bids r_i and publish corresponding ZK proofs P_i^{enc} on the blockchain. The simulator reads $P_{\bar{i}}^{\mathsf{enc}}$ of the malicious parties $\overline{\mathcal{S}}$ from the blockchain. Using the extractor for the zero-knowledge argument, the simulator extracts $v_{\bar{i}}$. The simulator sends all v_i (including those of the honest parties) to the TTP. The simulator receives from the TTP results $cmp_{i,j}$ of all comparisons and winning bid v_w for auctioneer A.

II) For each honest party $S_i \in \mathcal{S} \setminus \overline{\mathcal{S}}$, the simulator prepares a message of random AND-homomorphic encryptions $res_{j,i}$ following Fischlin's circuit output and the result of the comparison $cmp_{j,i}$. The simulator also invokes the simulator $\mathsf{Sim}_{P_{j,i}^{\mathsf{eval}}}^{A(\{C_i,C_j\})}(res_{j,i})$ which is guaranteed to exist. Then, the simulator sends the messages to the blockchain. For each malicious party $S_{\bar{i}} \in \overline{\mathcal{S}}$ that is still active, the simulator reads $P_{j,\bar{i}}^{\mathsf{eval}}$ and $res_{j,\bar{i}}$ from the blockchain. If judge A determines that $\mathsf{VerifyEval}(P_{j,\bar{i}}^{\mathsf{eval}},res_{j,\bar{i}},C_j,C_{\bar{i}})$ does not check, it publishes $\perp$ on the blockchain, and supplier $S_{\bar{i}}$ is dropped from the auction. We describe later how we deal with suppliers aborting the protocol.

III) For each honest party $S_i \in \mathcal{S} \setminus \overline{\mathcal{S}}$, the simulator prepares a message of random AND-homomorphic encryptions $shuffle_{i,j}$ following Fischlin's circuit output and the result of the comparison $cmp_{i,j}$. The simulator also invokes simulator $\mathsf{Sim}_{P^{\mathsf{shuffle}}}(shuffle_{i,j})$ for the shuffle ZK proof. It also opens the corresponding ciphertexts $\gamma_{\ell,m} \in shuffle_{i,j}$. Then the simulator sends the messages to the blockchain. For each malicious party $S_{\bar{i}} \in \overline{\mathcal{S}}$, the simulator reads $P_{\bar{i},j}^{\mathsf{shuffle}}$, $shuffle_{\bar{i},j}$, $\beta_{\ell,m}$, and $r_{\ell,m}$ from the blockchain. In case $\mathsf{VerifyShuffle}(P_{\bar{i},j}^{\mathsf{shuffle}},shuffle_{\bar{i},j},res_{\bar{i},j})$ does not check, the supplier $S_{\bar{i}}$ is dropped from the auction. If encrypting plaintexts $\beta_{\ell,m}$ and random coins $r_{\ell,m}$ do not result in $shuffle_{\bar{i},j}$, supplier $S_{\bar{i}}$ is dropped from the auction.

IV) If the winner S_w of the auction is honest, i.e., $S_w \in \mathcal{S} \setminus \overline{\mathcal{S}}$, then the simulator invokes the simulator for the ZK proof and sends it and v_w (received from the TTP) to auctioneer A. In case the ZK proof does not check, S_w is removed from the auction. If the winner S_w of the auction is malicious, i.e., $S_w \in \overline{\mathcal{S}}$, then the simulator receives the winning bid value v_w and the ZK proof that it corresponds to commitment C_w. If the ZK proof does not check, S_w is removed from the auction.

It remains to show that there exists is a simulator for the view of $\mathcal{A}_2$ (the semi-honest auctioneer/judge A): in the first step of the protocol, $\mathcal{A}_2$ receives IND-CPA secure ciphertexts and zero-knowledge proofs P^{enc}. In the second step $\mathcal{A}_2$ receives further IND-CPA secure ciphertexts and zero-knowledge proofs P^{eval}. We have shown in Section 4.2 that P^{eval} is zero-knowledge for the auctioneer. In the third step $\mathcal{A}_2$ receives IND-CPA secure ciphertexts, ZK proofs P^{shuffle} and the opened plaintext and randomness of some ciphertexts. The plaintexts are either all 1 or all 0 depending on $cmp_{i,j}$, and the randomness can be chosen consistently for each ciphertext. Finally, $\mathcal{A}_2$ receives v_w and the ZK proof of plaintext equality to C_w. Hence the view of $\mathcal{A}_2$ is simulatable from the TTP's output, i.e., the set of results of comparisons $\{cmp_{i,j}\}$ and winning bid v_w. □

Dealing with Early Aborts. Strain is particularly suitable for the blockchain, as it can handle any early abort after bids have been committed. Assume supplier $S_{\bar{i}}$ has aborted the protocol or has been caught cheating. Then, all others suppliers S_i can recover its bid $v_{\bar{i}}$ using the shares of its private key $sk_{\bar{i}}^{\mathsf{GM}}$ from commitment $C_{\bar{i}} = \mathsf{Enc}_{PK_{\bar{i}}}^{\mathsf{GM}}(v_{\bar{i}})$. We emphasize that our bid opening is secure against malicious suppliers due to ZK-proof P^{DLOG}. Suppliers publish $v_{\bar{i}}$ on the blockchain, and, after the bidding protocol, winning supplier S_w reveals bid v_w to semi-honest auctioneer A (proving plaintext equality to commitment C_w in zero-knowledge). The auctioneer compares v_w to all opened bids $v_{\bar{i}}$ and, in case, chooses a different winner w'. Hence, after commitments have been sent to the blockchain, no supplier can abort the auction. Even worse, aborting the auction reveals one's bid to all other suppliers.

C Dining Cryptographer Networks

A standard technique we use as an ingredient in Strain is a Dining Cryptographer (DC) network [11]. If out of a set of s parties (suppliers) $\{S_1, \ldots, S_s\}$ exactly *one* party S_i wants to broadcast their message m_i to all other parties, a DC network guarantees delivery of m_i to all other parties without revealing i, i.e., who has sent m_i.

Assume that all parties have exchanged pairwise secret keys $k_{i,j}$ with each other. In one round of a DC network, parties communicate in a daisy chain where party S_i sends a sum sum_i to party S_{i+1}. Upon receipt, S_{i+1} superposes sum_i with their own data and sends sum_{i+1} to S_{i+2}. Again, S_{i+2} superposes sum_{i+1} with their own data and sends sum_{i+2} to S_3 and so on. *Superposing* is simple: each party S_i XORs all pairwise keys $k_{i,j}$ of all other parties S_j to whatever previous party S_{i-1} has broadcast. Only one party S_* that wants to publish message m_* additionally XORs m_* to the previous sum. The last XOR of all data sent cancels out keys $k_{i,j}$ and m_* remains. So, a one round DC network allows one party dissemination of one message, protected by the DC network. Message m_* is public, but the sender's identity is protected. Thus, one supplier anonymously disseminates their public key, and everybody knows that this is a new valid key from one of the suppliers. Daisy chain communication can trivially be replaced by per party broadcasts, e.g., publishing to the blockchain. The advantage of the blockchain is efficiency: all parties broadcast their sums at the same time.

Multiple messages. To disseminate multiple parties' messages, several different strategies exist to resolve *collisions* in DC networks [11]. In Strain, we employ the approach by Bos and den Boer [7]. Assume that each party S_i has exchanged $s-1$ different pairwise keys $k_{i,j,u}, 1 \leq u \leq s-1$ with each other party S_j. Now, party S_i broadcasts all s powers $< m_i^1, \ldots, m_i^n >$ of their message m_i protected by the DC network. Instead of XORing messages broadcast with keys for protection, we now operate over $GF(2^q), q \geq |m|$, and use the following trick to cancel out keys. To protect the u^{th} power m_i^u of m_i, S_i adds all keys $k_{i,j,u}$ for $j > i$ to $K_{i,u}$ and subtracts keys $k_{i,j,u}$ for $j < i$ from $K_{i,u}$. S_i broadcasts $m_i^u + K_{i,u}$. All parties compute power sums $p_u(m_1, \ldots, m_s) = \sum_{i=1}^{s} m_i^u, 1 \leq u \leq s$. Each party uses Newton identities to compute m_i from power sums. All parties publish their output at the same time in parallel which is very efficient on a blockchain.

For space reasons, we do not discuss standard approaches realizing fully-malicious security for DC networks. These approaches use "traps" to identify and blame other parties, see, e.g., [7, 40, 41] for an overview.