PEPPER: Privacy-prEserving, auditable, and fair Payment based resource discovery at the PERvasive edge

Emrah Sariboz
emrah@nmsu.com
New Mexico State University
Las Cruces, NM, USA

Reza Tourani
reza.tourani@slu.edu
Saint Louis University
St. Louis, MO, USA

Roopa Vishwanathan
roopav@nmsu.com
New Mexico State University
Las Cruces, NM, USA

Satyajayant Misra
misra@nmsu.com
New Mexico State University
Las Cruces, NM, USA

ABSTRACT

Pervasive Edge Computing (PEC), a recent addition to the edge computing paradigm, leverages the computing resources of end-user devices to execute computation tasks in close proximity to users. One of the primary challenges in the PEC environment is determining the appropriate servers for offloading computation tasks based on factors, such as computation latency, response quality, device reliability, and cost of service. Computation outsourcing in the PEC ecosystem requires additional security and privacy considerations. Finally, mechanisms need to be in place to guarantee fair payment for the executed service(s).

We present *PEPPER*, a novel, privacy-preserving, and decentralized framework that addresses aforementioned challenges by utilizing blockchain technology and trusted execution environments (TEE). *PEPPER* improves the performance of PEC by allocating resources among end-users efficiently and securely. It also provides the underpinnings for building a financial ecosystem at the pervasive edge. To evaluate the effectiveness of *PEPPER*, we developed and deployed a proof of concept implementation on the Ethereum blockchain, utilizing Intel SGX as the TEE technology. We propose a simple but highly effective remote attestation method that is particularly beneficial to PEC compared to the standard remote attestation method used today. Our extensive comparison experiment shows that *PEPPER* is 1.23× to 2.15× faster than the current standard remote attestation procedure. In addition, we formally prove the security of our system using the universal composability (UC) framework.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**; **Privacy-preserving protocols**; Trusted computing.

KEYWORDS

Auditable resource discovery, Ethereum, Privacy-preserving auction, Edge Computing, Trusted Execution Environment.

ACM Reference Format:
Emrah Sariboz, Reza Tourani, Roopa Vishwanathan, and Satyajayant Misra. 2024. PEPPER: Privacy-prEserving, auditable, and fair Payment based resource discovery at the PERvasive edge . In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3634737.3637679

1 INTRODUCTION

As technology continues to advance and develop, the number of end user devices, such as smartphones and tablets, is increasing. Cisco predicted that by the end of 2023, the number of devices connected to the internet will be three times the world population [10]. The increase in the number of devices has enabled a wide range of applications. Some emerging applications, such as metaverse [28] and industrial internet of things [32], have demanding computational needs while demanding low application latency. Cloud computing offers a solution by allowing end users to offload the execution of these applications to meet their needs.

However, uploading tasks or retrieving the computing results for time-sensitive or privacy-sensitive applications underscores the need for alternative solutions because cloud servers are usually geographically distant from consumers and could have associated privacy risks [35]. To address these challenges with cloud computing, new distributed computing paradigms such as edge computing and variants, such as fog computing and multi-access edge computing have emerged [36]. The promise of edge computing is to bring powerful computing devices closer to the end-user region to improve the response time of users' needs. Application vendors can host their services for the user in the vicinity, reducing the overall latency and the cost of data transfer to the server.

The Pervasive Edge Computing (PEC) paradigm [16, 37, 38] takes this idea one step further by creating an ecosystem where a variety of devices at the edge ranging from laptops and tablets to smartphones, which have sufficient computation and storage power can be put to use to provide services on behalf of the cloud service provider. This will help the cloud provider offer better availability of services with relatively lower latency. Another benefit of PEC is enabling computing paradigms that harness the pervasiveness of

ASIA CCS '24, July 1–5, 2024, Singapore, Singapore

ACM ISBN 979-8-4007-0482-6/24/07.
https://doi.org/10.1145/3634737.3637679

edge devices. Consequently, PEC helps offer users a wider range of server options to outsource their computation compared to traditional edge computing. Naturally, the question arises, *how to find an appropriate end-device (i.e., PEC server) to which a user's computation can be offloaded in the most cost-effective, secure, and private manner within a dynamic and distributed PEC setting?* To the best of our knowledge, existing literature falls short in addressing the question at hand, which underscores the need for this study. This paper proposes a framework to address this question.

Motivation and Challenges: Creating a usable framework that locates end devices to offload computation for the PEC ecosystem requires proper incentives. The goal is to persuade end users to join the computing pool and meet the needs of other users. We can achieve this by offering money or digital assets (tokens) in exchange for the service. The framework must also establish a secure and trustworthy environment, ensuring that end users can confidently participate without concerns regarding losses, privacy, and security implications. In addition to providing incentives and a trustworthy environment, the framework must also be fair on two key aspects. Firstly, each participating server at the edge should have an equal opportunity to execute offloaded computations–the fairness can be proportional to capabilities. Secondly, it is crucial to create a system to establish correct distribution of payment to the party that executes the computation without any parties getting shortchanged. This setup provides a transparent and unbiased opportunity for every available device at the edge to perform service and get remunerated for it. Enhancing fairness fosters trust and participation among the PEC servers, which ultimately enhances the framework's success.

To design our framework and to study how it will operate in the PEC setting, we assume the use of an auction in the PEC for choosing the specific server to perform a requested user's computing. Particularly, we assume the use of a reverse auction, where there are several sellers (servers), each offering the same service at a potentially unique price, and one buyer (end-user requesting service) using the price to choose the least expensive server. Auction (in the rest of the paper auction refers to reverse auction) as the means of identifying the appropriate edge server to use makes sense as the set of edge servers essentially represent a marketplace for a client to choose from. Moreover, an unbiased auction guarantees that the chosen edge server is chosen based on the characteristics needed for the computation and will provide the most cost-effective operation among the corresponding usable servers set. In *PEPPER*, our approach involves orchestrating an auction where all PEC servers offer their services to users at competitive prices (we use price to define cost, any other cost metric can be readily used). Each PEC server strives to minimize the cost of executing the outsourced client computation in question.

Considering its inherent data provenance, record immutability, and support of smart contracts, we find the Ethereum blockchain suitable for our design. Hence, we implemented the auction as a smart contract on the Ethereum blockchain, to achieve our goals. Our approach guarantees the integrity of the auction process, allowing anyone to publicly verify its fairness while preventing any party, including the auctioneer, from manipulating the process to their advantage.

Another important challenge is that the framework must protect the privacy of the participants by hiding their bid details. The bid amount is considered private information since it represents the cost assessment of the servers for the outsourced application. The bid amount can be used to infer the computational strategy and process that a given PEC server uses to devise its bidding price for computations. Malicious PEC servers participating in the next round of auction may use this information to gain an unfair advantage in the auction. Hence, for fairness, it is imperative that the bid amounts of PEC servers are hidden from each other using a sealed-bid approach.

In *PEPPER*, only the winning bid amount is made public, and all the losing bid amounts are kept secret, even from the auctioneer. We note that by disclosing the winning bid's details on the smart contract, *PEPPER* assures that the bidder who won the auction placed the corresponding bid, thereby eliminating any suspicions of biased decision-making. We acknowledge that revealing the winning bid from a previous round might provide the bidders with data points to predict future bids. However, it's important to note that the cost of service execution heavily depends on the specific nature of the service and the data size.

Bids are formulated per computation and can vary widely, depending on the characteristics of the service and input size (e.g., sorting an array vs. a complex machine learning application like video annotation) as well as the computation demand. Hence, the winning bid from a previous round does not necessarily provide a definitive advantage in future biddings. Also, given that all servers in the network are privy to the information on the winning bid, there is no unfair advantage for a chosen set of servers. We use the Trusted Execution Environment (TEE), particularly, Intel Software Guard Extension (SGX) [18] to meet these requirements. The TEE promises to restrict the execution of computation loaded into the secure and encrypted area of the processor – called enclaves – from privileged software, including the operating system. The entities who want to verify the correctness of the loaded software to enclave perform what is called remote attestation.

Remote attestation assures that the application loaded into the enclave is correct and the enclave is up to date. As we detail in Section 5, the bidders verify the remote device before revealing bid details. However, for applications with a high number of participants–multiple entity auctions being one of them–the time it takes to complete remote attestation increases proportionally with the number of participants. To reduce this overhead, we propose a novel technique that significantly improves remote attestation cost, enhances the system's performance, and improves privacy.

Our novel **contributions** are as follows:

(i) We propose Privacy-prEserving, auditable, and fair Payment based resource discovery at PERvasive edge, *i.e., PEPPER*, a framework that enables the selection of the proper end-device for trustworthy computation offloading. *PEPPER* is blockchain and TEE agnostic, meaning it can be deployed on any blockchain platform with smart contract support.

(ii) *PEPPER* utilizes an auction implemented on a smart contract that ensures the privacy of bidders by revealing only the winning bid while keeping the bids that are lost in the auction confidential from all parties, including the auctioneer/client.

(iii) We propose a simple yet highly effective remote attestation technique that significantly reduces the overall time required to attest a remote server and its enclave software, which is particularly crucial in scenarios involving a large number of participants, a situation commonly prominent in PEC.
(iv) We provide an end-to-end implementation of *PEPPER*, which is built on top of the Ethereum blockchain and Intel SGX, and evaluate its effectiveness. We further provide a rigorous security analysis of *PEPPER* using Universal Composability (UC) framework.

The rest of the paper is organized as follows. In Section 2, we provide the required background relevant to the problem. In Section 3, we discuss the literature review. In Section 4, we detail the system model, threat model, and our assumptions. Section 5 describes the protocols that constitute *PEPPER* and provides details of the protocols. In Section 6, we extensively analyze the security of *PEPPER* using both informal and formal methods. In Section 7, we provide details of the implementation and experimental results of *PEPPER*. Finally, we conclude the paper in Section 8.

2 BACKGROUND

In this section, we provide a concise overview of the necessary background to facilitate a better understanding of our framework.

2.1 Trusted Execution Environment

A trusted execution environment, such as Intel Software Guard Extensions (SGX), enables one to execute applications inside secure enclaves on untrusted machines [19]. The enclave secludes the execution of the application and its data from other privileged applications, including the operating system. Any remote entity can verify if an enclave is initialized correctly, is on a state-of-the-art platform, and runs the correct software by performing remote attestation. Verifying the correctness of such information is critical and essential before exposing any sensitive data. Intel SGX remote attestation comes in two flavors:
Enhanced Privacy ID (EPID): The EPID-based remote attestation employ an anonymous group signature scheme developed by Intel [18]. During remote attestation, the attesting enclave generates a data structure known as a *report*. The *report* encapsulates essential information such as the hash of the code and data loaded into the enclave (*MRENCLAVE*), a hash of the public key of the entity that signed the enclave (*MRSIGNER*), user-data field, and other pertinent details. The generated *report* is then signed using the EPID key, creating an EPID signature. The relying party[1] can ensure that the *report* is indeed signed by a valid enclave, without needing to know the exact identity of the enclave, thanks to the anonymity property using the Intel Attestation Service (IAS).
Data Center Attestation Primitive (DCAP): In DCAP-based remote attestation, the attesting enclave generates *report*, which is then signed by the Provisioning Certification Enclave (PCE) using Provisioning Certificate Key (PCK) that is unique to each platform [31]. The PCE, located on the same platform as the attesting enclave, acts as a local certificate authority. Unlike the EPID-based attestation model, there isn't a requirement connect to Intel's servers each time to verify the attestation *report*. The Intel SGX provisioning certification service (PCS) provides required APIs for parties to retrieve and cache the necessary information, such as the Provisioning Certificate Key (PCK) certificate and Certificate Revocation Lists (CRLs), to validate remote attestation locally. We employ DCAP-based remote attestation in *PEPPER*.

2.2 Ethereum and Smart Contracts

The blockchain technology enables mutually untrusting parties to connect and transact without the need for any centralized trusted party. Ethereum is a widely utilized decentralized blockchain platform that enables the creation and execution of decentralized applications (dApps) in the form of smart contracts and uses Ether as its cryptocurrency [40]. Smart contracts are software programs written in specialized programming languages, such as Solidity or Vyper, which get executed on the blockchain whenever pre-encoded conditions are met. Interacting with a smart contract involves creating a transaction, which is then verified by the network's validators—entities that partake in establishing network consensus. These interactions are permanently recorded on an immutable ledger, ensuring transparency and integrity.

2.3 Cryptographic Preliminaries

Elliptic Curve Integrated Encryption Scheme (ECIES) [1] is a hybrid encryption scheme that combines the properties of symmetric and asymmetric cryptography. In our framework, we utilize ECIES to conceal the bid details of the auction participants and give a formal definition below.

Definition 1. **Elliptic Curve Integrated Encryption Scheme [1]**: *An Elliptic Curve Integrated Encryption Scheme (ECIES) is defined over one probabilistic (*KeyGen*) and four deterministic polynomial time algorithms: (*$\mathsf{KAF}, \mathsf{KDF}, \mathsf{Encrypt}, \mathsf{MAC}$*). Let* $\mathbb{G}$ *be a multiplicative cyclic group of prime order* p *generated by* g_1*.*

- $\mathsf{KeyGen}(params) \rightarrow (sk, pk)$: KeyGen *algorithm takes elliptic curve parameters, samples random secret key* $sk \xleftarrow{\$} \mathbb{Z}^+$ *and computes public key* $pk \leftarrow g_1^{sk}$.
- $\mathsf{KAF}(sk_x, pk_y) \rightarrow (ss)$: *Key agreement function (*KAF*), takes* sk *of executor party and the* pk *of receiver party and produces shared-secret key* ss.
- $\mathsf{KDF}(ss) \rightarrow (k_{MAC}, k_{ENC})$: *Key derivation function,* KDF*, algorithms takes* ss *and produces MAC key,* k_{MAC} *and symmetric key,* k_{ENC} *by* $k_{MAC}, k_{ENC} \xleftarrow{\$} \{0,1\}^{\lambda}$.
- $\mathsf{Encrypt}(k_{ENC}, m) \rightarrow (c)$: *Encryption algorithm,* $\mathsf{Encrypt}$*, encrypts the message* m *using* k_{ENC}.
- $\mathsf{MAC}(k_{MAC}, c) \rightarrow (tag)$: *Message authentication code function,* MAC*, generates tag on* c *using* k_{MAC}.

3 RELATED WORK

In this section, we provide a comprehensive review of the existing literature pertaining to our design. This includes an examination of research on privacy-preserving auctions and smart contracts, and incentive-based resource utilization within the context of edge computing paradigms.

[1] Intel SGX refers to parties verifying the attestation as relying parties: https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/attestation-services.html

3.1 Privacy-preserving Auction

The transparent nature of smart contracts is both a blessing and a curse. On the one hand, it can transparently mediate the interaction between mutually untrusted parties; on the other hand, the lack of privacy limits the application variety, such as auction. To this end, privacy-preserving auction frameworks for smart contracts have been proposed in the literature [4, 9, 13–15, 20–22, 33, 41].

We categorize the existing work in blockchain-based privacy preserving auction frameworks into two broad categories based on the underlying technique: zero knowledge-based solutions and TEE-based solutions. In the first category, the bidders use cryptographic commitments to conceal their bids and employ zero-knowledge proofs (ZKPs) to prove the validity of their bid values. These solutions require the bids to be revealed to a smart contract or to the auctioneer during the winner-election period [4, 9, 13, 14, 20–22, 33]. However, by revealing bid details, the bidder's strategy is compromised. In contrast, our framework does not mandate the disclosure of bids to any party, except for the winning bid.

In the second category of solutions, the bidders encrypt their bid values off-chain and utilize TEEs to safeguard the bid privacy of the participants during the decryption phase [15]. However, the number of remote-attestation that these works require is proportional to the number of bidders in the auction, making it impractical for auctions with a high number of users to adopt due to the large latency. Our framework decreases the number of remote attestation requests from one request per bidder to one request per auction, reducing the communication cost between bidders and enclave and the total time it takes to perform remote attestation.

3.2 Privacy-preserving Smart Contracts

Another area of research worth noting is the pursuit of improved confidentiality for smart contracts. This is accomplished by executing them within a TEE, which restricts access to the execution details of the smart contract. Several studies have explored this approach to address the privacy limitations of smart contract execution [8, 11, 41]. However, these solutions have their limitations. Some are specific to a particular blockchain due to the introduction of their own consensus operation, while others can be prohibitively expensive. In contrast, our solution is chain-agnostic and does not incur significant costs, as detailed in Section 7.

3.3 Incentive-based Resource Utilization

One of the primary aims of our work is to incentivize participants to contribute their resources to meet the needs of others. In this subsection, we detail the related work in methodologies used in incentive mechanisms.

Several works have applied a game-theoretic approach to incentivize the party to execute outsourced computation in the edge computing paradigms [6, 25–27, 44]. These works model the interaction of parties as a game and use mathematical theorems to deduce the parties' strategies. However, the lack of auditability and the centralized nature of these works make these approaches undesirable. On the other hand, our work is auditable due to its blockchain agnostic nature. Researchers have utilized blockchain-based auction frameworks to incentivize resources of end devices [2, 17, 23, 24, 42]. However, these works do not consider hiding the bidding details, which is critical to keep the auction strategies of the participants private. Our work does not require bidders to reveal their bids to any third party, including the auctioneer.

The closest related work to our framework is [15], where authors employ a TEE to decrypt the bids that are encrypted off the chain to achieve a privacy-preserving auction. However, our approach differs in *three* important ways: *(a) PEPPER* does not require the deployment of contracts every time a new auction is needed. The same contract is reused, which ultimately reduces the long-term costs and enables multiple auctioneers to utilize the framework simultaneously; *(b) PEPPER* reduces the per-auction number of remote attestation requests – from one request per bidder to a single request per auction; and *(c)* our framework does not require the auctioneer to be online during the declaration of the winner, whereas [15] requires the auctioneer to be online to initialize the transaction that assigns the winner's address on the smart contract. By removing the reliance on the auctioneer's online presence, our framework avoids potential delays in payment and the conclusion of the auction.

4 MODELS AND ASSUMPTIONS

In this section, we provide a detailed explanation of the parties involved in *PEPPER*, along with our threat model and underlying security assumptions.

4.1 System Model

Pervasive Edge Computing (PEC) enables devices, such as computers, tablets, Internet of Things devices, and mobile devices to dynamically join and leave pools of computing resources at the network edge [12, 34]. This approach aims to enhance the available computational resources at the edge and minimize data transmission time compared to using the Cloud, especially when meeting applications' latency requirements. Furthermore, PEC capitalizes on the pervasive presence of user-end devices, significantly enhancing the overall availability of computing services by utilizing these devices for computational tasks. With this objective in mind, our system model consists of a resource consumer ($\mathcal{RC}$), PEC servers who make up the set up bidders ($\mathbb{B}$), a service provider ($\mathcal{SP}$), and a Manager ($\mathcal{M}$).

An $\mathcal{SP}$ is the entity that owns a service(s). In this paper, we consider services that benefit from execution at the network edge, such as object detection or video annotation, either to avoid transferring large volumes of data to the Cloud or minimize the response latency. Such services may require an input that belongs to resource consumers or can be provided by the $\mathcal{SP}$. We define $\mathcal{RC}$ as a client of an $\mathcal{SP}$ who aims to outsource the execution of the $\mathcal{SP}$'s service to the more capable servers at the edge, primarily due to the limited resource of her device. PEC servers are entities that are willing to execute the outsourced application for $\mathcal{RC}$. Finally, the Manager $\mathcal{M}$ is the party that is responsible for mediating the interactions between its enclave (E) and the smart contract (*SC*).

The $\mathcal{M}$ is equipped with a processor that supports a TEE, such as Intel SGX or ARM TrustZone. In *PEPPER*, to outsource the execution of her service, $\mathcal{RC}$ initiates an auction. The auction is run by $\mathcal{M}$ as the auction manager and allows the PEC servers – which we call *bidders* in the rest of the paper – to compete against

Table 1: Notations used in *PEPPER*

Notation	Description
E	Enclave.
$\mathbb{B}$	Set of bidders.
$\mathcal{RC}$	Resource consumer.
$\mathcal{SP}$	Service provider.
$\mathcal{M}$	Manager.
SC	Smart contract.
BC	Blockchain.
id_A	Auction identifier.
agg_{nonce}	Aggregated nonce.
$Cert_x$	Certificate of entity x.
tx_x	Transaction of entity x .
$report$	Attestation report.
$addr_x$	Blockchain address of entity x.

each other by submitting their best offers to win the task. We give our table of notations in Table 1.

4.2 Threat Model

We consider the following threats from the bidders, the manager, and the resource consumer, each falling into their respective categories.

Bidders: *(a)* The bidders may attempt to learn the bid values of other bidders to adjust theirs accordingly; *(b)* the bidders may abort the auction after registering for it or may keep registering to the auction to perform Distributed Denial of Service (DDoS) attack on the framework.

Manager($\mathcal{M}$): *(a)* The manager may collude with other bidders to learn and leak the bids to others; *(b)* the manager may attempt to declare a different entity as the auction winner rather than the one reported by the enclave; *(c)* the manager may attempt to exclude one or more bids while transmitting them from the smart contract to the enclave.

Resource Consumers ($\mathcal{RC}$): *(a)* The resource consumer may create an arbitrary auction without a legitimate intention of outsourcing job; *(b)* the resource consumer may attempt to evade paying the auction winner.

4.3 Security Assumptions

The service provider $\mathcal{SP}$ is presumed honest. Since $\mathcal{SP}$ owns the service, it is in its best interests to ensure correct service execution—$\mathcal{SP}$'s reputation is at stake. We assume Manager $\mathcal{M}$ to be honest-but-curious – the manager follows the protocol but attempts to learn additional information with malicious intent. Conversely, we consider the resource consumer, denoted as $\mathcal{RC}$, as an economically-rational malicious entity, actively seeking to obtain and disclose bid information. Similarly, the bidders are also considered to be potentially malicious entities. We assume the application responsible for decrypting the bids and finding the winner inside the enclave is developed by $\mathcal{SP}$ and then provided to $\mathcal{M}$ and publicly available for the participants.

We recognize the recent attacks on Intel SGX [3, 43]; while it is important to deter such attacks and build secure enclaves, it is an orthogonal to the aim of this paper. We also assume the existence of a mechanism such as a marketplace, where the resource consumers and servers can interact directly or via a facilitating authority. While we do not consider the case where the executing party may return the incorrect execution result, the modular design of our framework enables the adoption of verifiable computation techniques, such as [29, 30], to resolve the issue. Finally, we make the assumption that the PEC servers have undergone authentication by the $\mathcal{SP}$ before executing the service on behalf of the $\mathcal{SP}$ using techniques, such as [12].

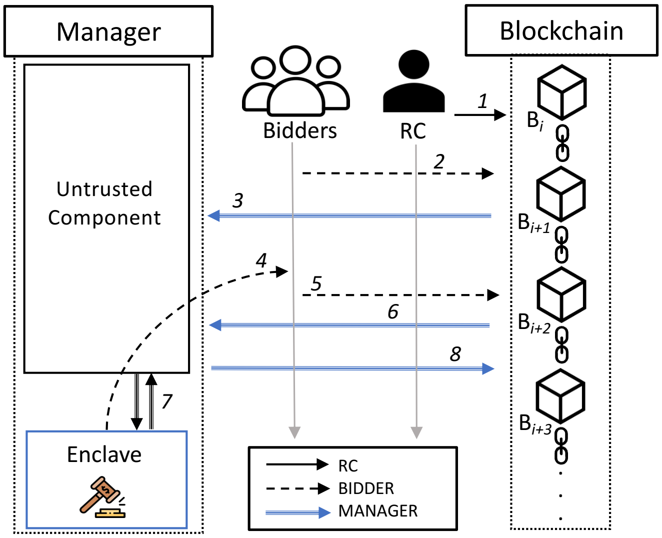

Figure 1: The interaction of parties throughout the entire auction process, from the creation of the auction to the declaration of the winner on the smart contract.

5 PEPPER CONSTRUCTION

In this section, we discuss the protocols that constitute the *PEPPER* framework. These protocols serve as the foundation for privacy-preserving PEC server selection to execute outsourced services.

5.1 Design Overview

In a nutshell (refer to Figure 1), the process in *PEPPER* begins after the service provider implements and deploys the auction smart contract that will be used for choosing the designated PEC server. We note that the service provider's presence is not needed after deploying the auction's smart contract, and the auctions can run between the resource consumers and bidders independent of the service provider.

Once the smart contract is deployed, a resource consumer interested in outsourcing their service defines the start and end time for auction registration, as well as the auction end time (Step (1)). Bidders who are interested in participating in the auction then register themselves (Step (2)). During the registration, the bidders engage in what we call "nonce aggregation", enabling them to verify enclave freshness during remote attestation. Once the bidder registration phase is complete, the Manager ($\mathcal{M}$) retrieves the aggregated nonce from the Blockchain and initializes its Enclave E (Step (3)).

As part of the enclave's remote attestation, the enclave generates an attestation report, *report*, and makes it available for the bidders

Protocol 1 System Setup

```
                 {At Service Provider}
1: $(pk, sk) \leftarrow \mathsf{KeyGen}(1^{\lambda})$.
2: $\mathcal{SP}$ implement $SC$ and deploy on Blockchain.
                      {At Bidders}
3: for each $b_i \in \mathbb{B}$ do
4:     $(pk_i, sk_i) \leftarrow \mathsf{KeyGen}(1^{\lambda})$.
5: end for
```

to verify the freshness of the enclave before sending their encrypted bids, which represents the interest to perform the outsourced task, to the smart contract (Step (4)). Upon successful attestation, the bidders encrypt their bids using the algorithm defined in Definition 1 and send them to the smart contract *SC* (Step (5)), along with a deposit for the auction. This step is essential to confirm if the enclave is legitimate before disclosing sensitive details. Once the auction period is over, the Manager collects the encrypted bids and information of bidders and sends them to the enclave along with deposits (Step (6)). The enclave decrypts the bids to find the winning bid and the winner's address, and then reliably sends the details to the Manager (Step (7)). The Manager creates a transaction including the signed message from the enclave, which indicates the winning bid, and submits it to the smart contract (Step (8)). Finally, the smart contract verifies the integrity and authenticity of the enclave's signature on the blockchain and declares the winner's address. Following this step, participants who did not win the auction can request refunds of their deposit amounts and exit the auction.

5.2 PEPPER Detailed Design

In this section, we describe the five protocols that constitute the *PEPPER* framework in detail. Protocol 1 comprises deploying a smart contract for the auction and generating key pairs for all the bidders, $\mathbb{B}$. Protocol 2 deals with auction creation, bidders registration, and enclave (*i.e.,* E) initialization. Protocol 3 details our novel remote attestation process. Protocol 4 details the secure bid submission process. Protocol 5 is used to verify the winner's details and announcement on the smart contract.

5.2.1 System Setup (Protocol 1). Initially, $\mathcal{SP}$ generates a public/private key pair, *i.e.,* (pk, sk) for the signing transaction that deploys the smart contract, *SC*. Subsequently, $\mathcal{SP}$ implements the *SC* that codifies auction logic on the Blockchain (Lines 1-2). We note that *SC* is provider-specific, allowing the service provider to adjust the auction logic as needed for the interactions, such as reverse auctions, Dutch auctions, etc., as discussed in the introduction due to the nature of the PEC. Due to nature of PEC where multiple servers offers the same service at varying prices, we have implemented a reverse auction approach. During the system setup, the bidders, which are willing to offer their computing resources for the outsourced service execution, generate their own (pk, sk) pairs using $\mathsf{KeyGen}(1^{\lambda})$, where λ is a security parameter (Line 4). The encryption process requires these keys in Protocol 4.

5.2.2 Auction Initiation (Protocol 2). In this protocol, the $\mathcal{RC}$ creates an auction by calling $SC.\mathsf{CreateAuction}()$ and sets parameters, such as the payment for the outsourced job (*payment*), the start and end time for auction registration (*regTime* and *regEndTime*), the time when the auction ends (*auctionEndTime*), and the required deposit amount to participate in the auction (*requiredDeposit*) (Line 1). The rationale for setting different times on smart contract *SC* is to divide the auction process into two periods: the registration period and the auction period. The auction period commences immediately after the *regEndTime* and remains valid until the *auctionEndTime*. During the auction creation, the smart contract computes the auction identifier id_A using the current time on the blockchain (*time.now*) and the blockchain address of the resource consumer ($addr_{\mathcal{RC}}$).

Protocol 2 Auction Initiation

```
                 {At Resource Consumer}
1: $id_A \leftarrow SC.\mathsf{CreateAuction}(payment, regTime,$
   $regEndTime, auctionEndTime, requiredDeposit)$.
                      {At Bidders}
2: for each $b_i \in \mathbb{B}$ do
3:     Pick $nonce_i \xleftarrow{\$} \mathbb{Z}^{+}$.
4:     $SC.\mathsf{RegisterBidder}(id_A, nonce_i, pk_i)$.
5: end for
                      {At Manager}
6: $agg_{nonce} \leftarrow SC.\mathsf{GetAggregatedNonce}(id_A)$.
7: Initialize E with $agg_{nonce}$.
                   {At Smart Contract}
8: function CreateAuction($payment, regTime, regEndTime,$
9: $auctionEndTime, requiredDeposit$):
10:    $id_A \leftarrow H(time.now() \| addr_{\mathcal{RC}})$.
11:    return $id_A$.
12: end function
13: function RegisterBidder($id_A$, $nonce_i$, $pk_i$):
14:    if $(regTime <= time.now() < regEndTime) \land id_A$ exists
       then
15:        Store bidder details.
16:    else:
17:        return $\perp$.
18: end function
19: function GetAggregatedNonce($id_A$):
20:    if $(regTime < time.now() < auctionEndTime) \land id_A$ exists
       $\land\ !isRequested$ then
21:        isRequested = True.
22:        return $agg_{nonce}$ of auction $id_A$.
23:    else:
24:        return $\perp$.
25: end function
```

The id_A identifies the auction and bidders use it when registering for the auction. The smart contract now locks the offered payment to prevent the resource consumer from reclaiming it before the end of the auction. Upon creation of the auction, the bidders interested in bidding for the auction randomly sample a nonce. The bidders will use the nonce values to verify the freshness of the enclave (Line 3). Each bidder then registers for the auction by calling $SC.\mathsf{RegisterBidder}()$ that accepts auction identifier (*i.e.,* id_A), the sampled nonce ($nonce_i$), and the bidder's public key (*i.e.,* pk_i) (Line 4). The id_A uniquely identifies the auction, for which the bidder is registering.

In the process, SC aggregates the nonce values provided by all participating bidders. SC achieves this by summing the individual nonce values together which forms the foundation of our novel remote attestation protocol. Aggregating nonce on smart contracts enables public verification of the total value (agg_{nonce}) and eliminates the trust assumption on any party. Unlike traditional remote attestation approaches that require each bidder to individually connect with the enclave and provide their nonce values to verify freshness, our framework employs a single nonce that is aggregated on the smart contract. The agg_{nonce} will later be used by enclave E to generate a *report* for the remote attestation (Protocol 3, Line 2). This approach not only simplifies the process but also enhances scalability. Instead of generating separate reports for each participant, our novel remote attestation model generates a single report for all participants. *As a result, the total number of remote attestations is reduced to one, regardless of the number of bidders.* Finally, the smart contract stores the bidder's public key (pk_i). The enclave requires the pk_i values to derive the necessary keys to decrypt the encrypted bids.

The smart contract first checks if the auction with ID id_A exists and if the registration period is still ongoing. If both conditions are met, the smart contract will store the bidder's details. Once the registration phase for the auction id_A is over, the manager $\mathcal{M}$ calls SC.GetAggregatedNonce() with id_A argument to retrieve the agg_{nonce} from SC to initialize the enclave (Lines 6-7). The smart contract checks if the auction registration period is over and the manager already requested the agg_{nonce} for the auction id_A. The second check is important to prevent the $\mathcal{M}$ from initializing the enclave multiple times for the same auction. The manager initializes the enclave with agg_{nonce} if all the checks hold. During initialization, the application that determines the winner undergoes compilation into a format that enables its loading into the enclave. The bidders can now utilize the loaded application.

5.2.3 Remote Attestation (Protocol 3). After $\mathcal{M}$ initializes the enclave E, the application inside the enclave initially generates public and private key pair (pk_E, sk_E) (Line 1). The public key, *i.e.,* pk_E, is required by bidders to encrypt their bids before submitting them to the smart contract. The enclave utilizes the private key, *i.e.,* sk_E, to derive the shared-secret key during the decryption of the bids.

The bidders before submitting any sensitive details (bids) need to validate the legitimacy and the freshness of the enclave. To assist bidders in achieving their goals, the enclave generates an X.509 certificate, denoted as $Cert_E$, and constructs the message M as $M = (Cert_E \| agg_{nonce})$. Additionally, the enclave generates a *report* and incorporates the digest of M (i.e., $h = H(M)$) into the user-data field of the *report* (Lines 2-3). Under the hood, the *report* is signed by the Provisioning Certification Enclave (PCE) using the Provisioning Certificate Key (PCK), a unique key for the hardware. The PCE is an Intel-provided enclave located on the same platform as the attesting enclave, which serves as a root of trust for the $Cert_E$.

The E initializes a TLS server using $Cert_E$ and makes the pk_E and signed *report* available to bidders. We note that the root of trust of $Cert_E$ is the enclave itself [19]. Then, participating bidders retrieve the *report*, $Cert_E$, and pk_E from enclave E (Line 4). The bidders, using the retrieved values, verify the *report* by calling VerifyRemoteReport, which checks if the code loaded into the enclave is correct. If it is, the signature on the *report* belongs to the specific enclave and is up to date, verified by checking the Provisioning Certification Key (PCK) certificate and Certificate Revocation Lists (CRLs) (Line 6).

Protocol 3 Remote Attestation

```
                    {At Enclave}
 1: E creates $(pk_E, sk_E) \leftarrow$ ECIES.KeyGen$(1^{\lambda})$.
 2: E generates $Cert_E$, sets $M = (Cert_E \| agg_{nonce})$ and computes
    $h = H(M)$.
 3: E generates $report \leftarrow$ GenerateReport$(h)$.
                    {At Bidders}
 4: Retrieve $report$, $pk_E$ and $Cert_E$ from E.
 5: for each $b_i \in \mathbb{B}$ do
 6:   if $true \leftarrow$ VerifyRemoteReport$(report)$ then
 7:     $agg_{nonce} \leftarrow SC$.GetAggregatedNonce$(id_A)$.
 8:     $h = H(Cert_E \| agg_{nonce})$.
 9:     if $report.userData == h$ then
10:       Call Protocol 4.
11:     else
12:       return $\perp$.
13:     end if
14:   else
15:     return $\perp$.
16:   end if
17: end for
```

Intel provides necessary APIs to retrieve information from Intel Provisioning Certification Service[2]. The PCK and CRLs are cacheable to further reduce the report validation time for future interactions. Upon successful verification, each bidder checks the enclave's freshness by obtaining the agg_{nonce} from SC and computing $h = H(Cert_E \| agg_{nonce})$. To verify the freshness, bidders compare the value of the user-data field in the report with h (Line 9). Verifying freshness is crucial in eliminating successful replay attacks, in which $\mathcal{M}$ potentially uses a previously initialized enclave rather than a new instance [7]. If the verification of enclave freshness and authenticity holds, the bidders interact with Protocol 4 to encrypt and submit their bids.

5.2.4 Bid Submission (Protocol 4). Bidders use this protocol to encrypt their bids and submit them to SC. Initially, each registered bidder generates a shared secret denoted as ss using the enclave's public key (pk_{E}) and its own private key (sk_i). The ss key facilitates the establishment of a shared secret between the bidder and the enclave without the need for a direct key exchange. Each bidder further derives k_{MAC} and k_{ENC} keys from ss. Bidders use k_{ENC} to encrypt their bid and use k_{MAC} to generate a tag on the encrypted bid. The enclave will use tag to ensure the integrity and authenticity of the encrypted bid value.

To submit their bid to the smart contract, each bidder concatenates the *bid* and *tag* and invokes the SC.SendBid function, passing the auction identifier (id_A), encrypted and tagged bid (c_{bid}), and deposit amount ($deposit$) as arguments (Lines 1–9). The deposit amount, which is set to the same amount for every participant, will be locked in SC until the auction winner is determined. The

[2] https://api.portal.trustedservices.intel.com/provisioning-certification

Protocol 4 Bid Submission

```
{At Bidders}
for each $b_i \in \mathbb{B}$ do
    $ss_i \leftarrow$ ECIES.KAF$(sk_i, pk_{\mathrm{E}})$.
    $(k_{MAC}, k_{ENC})$ = ECIES.KDF$(ss_i)$.
    $pbid_i \leftarrow \mathbb{Z}^+$.
    $bid_i \leftarrow$ ECIES.Encrypt$(k_{ENC}, pbid_i)$.
    $tag_i \leftarrow$ ECIES.MAC$(k_{MAC}, bid_i)$.
    $cbid_i = (bid_i \| tag_i)$.
    Call $SC$.SendBid$(id_A, cbid_i, deposit)$.
end for
{At Smart Contract}
function SendBid($id_A$, $cbid_i$, $deposit$):
    if $(regEndTime < time.now() < auctionEndTime)$
$\wedge$ $id_A$ exists $\wedge$ $deposit == requiredDeposit$ then
        Store bidder details on smart contract.
    else:
        return $\perp$.
end function
function GetBidDetails($id_A$):
    if $(auctionEndTime < time.now())$ then
        return bids and public keys of bidders.
    else:
        return $\perp$.
end function
```

deposit aims to prevent malicious bidders from registering multiple times but not participating in the auction. The smart contract first verifies that the auction with the ID id_A exists, the auction is not over, and the *deposit* amount matches the required amount. If all conditions are met, the bid details are stored by the smart contract. Once the auction period is completed, the Manager $\mathcal{M}$ retrieves the encrypted bids and the bidders' public keys and forwards them to E through the TLS endpoint in the form of a dictionary (*Bids*).

5.2.5 Winner Announcement (Protocol 5). After receiving the bids and public keys of the bidders, the enclave iterates over each bid and creates the shared secret *ss* using its private key sk_E and the participants' public keys pk_i. Using *ss*, the enclave derives two symmetric keys, denoted as k_{MAC} and k_{ENC}, by calling ECIES.KDF (Line 6). To verify the integrity of bids, the E first computes $tag' \leftarrow$ ECIES.MAC(k_{MAC}, bid_i) and checks it against the *tag*, which is included in the ciphertext. If the integrity verification is successful, then E decrypts the bid using k_{ENC}, and finds the minimum bid amount and the winner's address (Lines 8–12).

After determining the winning bid and winner's address, the enclave concatenates the winner's bid amount (*minBid*), the winner's address ($addr_{winner}$), and the *Bids* dictionary. Subsequently, it signs the concatenated data to ensure integrity and provenance (Lines 16-17). The enclave forwards σ_{E} along with *minBid* and $addr_{winner}$ to $\mathcal{M}$, who in turn, verifies the enclave signature, constructs transaction $tx_{\mathcal{M}}$, and calls SetWinner function on *SC* (Line 18). To prevent a malicious $\mathcal{M}$ from setting a different winner's address than the one determined by E, *SC* needs to ensure that the message is signed by E rather than $\mathcal{M}$. In order to achieve this, we need to extract the signer of the message from the given message and compare the address with the Manager's address on the chain.

Protocol 5 Winner Announcement

```
{At Enclave}
1:  $minBid \leftarrow uint.Max()$.
2:  $index \leftarrow 0$.
3:  $addr_{winner} \leftarrow 0$.
4:  for each $bid_i \in Bids.items()$ do
5:      $ss \leftarrow$ ECIES.KAF$(sk_{\mathrm{E}}, bid_i.pk_i)$.
6:      $(k_{MAC}, k_{ENC}) \leftarrow$ ECIES.KDF$(ss)$.
7:      $tag' \leftarrow$ ECIES.MAC$(k_{MAC}, bid_i)$.
8:      if $tag' == bid_i.tag_i$ then
9:          $pbid_i \leftarrow$ ECIES.Encrypt$^{-1}(k_{ENC}, bid_i)$.
10:         if $pbid_i < minBid$ then
11:             $minBid = pbid_i$.
12:             $addr_{winner} = bid_i.pk_i$.
13:         end if
14:     end if
15: end for
16: $h \leftarrow H(minBid \| addr_{winner} \| Bids)$.
17: $\sigma_{\mathrm{E}} \leftarrow$ Sign$(h)$.
{At Manager}
18: $\mathcal{M}$ creates $tx_{\mathcal{M}} = SC$.SetWinner$(id_A, \sigma_{\mathrm{E}}, minBid, addr_{winner})$.
{At Smart Contract}
19: function SetWinner($id_A$, $\sigma_{\mathrm{E}}$, $minBid$, $addr_{winner}$):
20:     if $addr_{\mathrm{E}} \overset{?}{=}$ ecrecover$(\sigma_{\mathrm{E}}, H(minBid \| addr_{winner} \| Bids))$ $\wedge$
    $id_A$ then
21:         $auctionWinner = addr_{winner}$.
22:         $winnerBid = minBid$.
23:     else:
24:         return $\perp$.
25: end function
26: function RefundDeposit($id_A$):
27:     if $caller \neq auctionWinner$ $\wedge$ auction is over $\wedge$ $caller \in \mathbf{B}$
    then
28:         refund $deposit$.
29:     else:
30:         return $\perp$.
31: end function
```

In Solidity, the ecrecover opcode is used to extract the address of the signer from a given signature and a message [39].

The *SC* computes the digest of concatenation of *minBid*, $addr_{winner}$ and *Bids* and use ecrecover low-level instruction (*i.e.,* opcode) to extract the *addr* of the signer from the given signature σ (*i.e.,* σ_{E}) on message *M*. It is important to note that, during the extraction of the winner's address, the *SC* includes *Bids* in the hash function to verify whether $\mathcal{M}$ excluded any bids during transmission to E or modified the original bidder's list. In the case of malicious activity from *M*, the extracted address would not match the address of the enclave. If the check holds, the *auctionWinner* and *winnerBid* details are now stored on *SC* and are available to everyone. Every entity, except the winner bidder, can now withdraw their *deposit* by calling *SC*.RefundDeposit() (Line 26).

We note that the enclave's address, $addr_{\mathrm{E}}$, is stored within *SC*, and a specific function is implemented to allow only the service provider to update this address, especially when there is a need for a new manager. This functionality is achieved through the use of modifiers, which enable function-level access control in the smart contracts. For the withdrawal, the smart contract checks if the auction is over, if the caller is not the winner, and if the caller is one of the participants in the auction. If all conditions are met, the deposit will be refunded to the bidder.

6 SECURITY ANALYSIS

In this section, we present a comprehensive security analysis of our framework, describing potential attack vectors per our adversary model, and detailing how *PEPPER* effectively mitigates them. Furthermore, we establish the security of our framework formally using the Universal Composability (UC) framework [5], providing a rigorous validation of its robustness against various security threats.

6.1 Informal Security Analysis

Malicious Bidder: Following the threat model, the malicious bidders may attempt to learn and leak the bid details of others. However, in *PEPPER*, the bidders encrypt their bids, which are subsequently decrypted within an enclave. Unless bidders collude and share their bids among themselves, the auction only discloses the winning bid at the end of the auction. We emphasize that bidders' primary goal is to secure victory in the auction by offering their service at a competitive price. Sharing the bid details with other participants undermines their competitive advantage.

Additionally, malicious bidders may attempt to disrupt the auction process by either aborting the auction after registering or continuously registering without actually participating in the auction. However, the security of the system remains intact as the registration period is the initial step before the auction period. Consequently, the bidder will incur losses as the transaction to register requires gas fees.

Malicious Manager: The malicious manager may attempt to selectively exclude a subset of bidders by removing their bids when transferring them from the smart contract to the enclave, *i.e.,* during the winner election process. We eliminate this attack by including all the bids when creating a signature on the digest of the bids inside the enclave (Protocol 5, Line 16) and also when verifying the signature of the enclave (Protocol 5, Line 20). We can perform such verification since the encrypted bids are permanently recorded on the blockchain during the bid submission process. In the case that a bid is excluded during transfer, the enclave will not have all the bids. The signature verification of the enclave will fail, hence the malicious attempt of the manager will be detected.

Another possible attack from a malicious manager is to change the winner's address to another bidder. Our design inherently prevents such an attack as it requires the enclave to digitally sign the digest of the winner's address, inside the secure environment, using the enclave's private key. By doing so, *PEPPER* ensures that any modification of the winner's address by the manager can be detected. When the enclave's signature is verified on the blockchain, any discrepancy, such as a change in the winner's address, would result in the failure of signature verification. The verification process effectively reveals and thwarts any such malicious attempts by a manager to modify the auction outcome.

Functionality $\mathcal{F}_{\text{bc}}$

Smart contract deployment: Upon receiving the tuple $(\text{sid}, \text{deploy}, SC, code, addr_{\text{E}})$ from the service provider $\mathcal{SP}$, $\mathcal{F}_{\text{bc}}$ will first deploy the smart contract SC and then store the corresponding tuple $(SC.address, code, addr_{\text{E}})$ in a table called scTable. $\mathcal{F}_{\text{bc}}$ will then return the tuple to the service provider $\mathcal{SP}$ and $\mathcal{S}$.

Figure 2: Ideal functionality for blockchain

Malicious Resource Consumer: In our framework, we address the issue of resource consumers attempting to acquire and disclose bid details of unsuccessful bidders during the bid reveal period. Although such actions may not be driven by malicious intent, the leaked information could be exploited by other participants to adjust their bids and secure victory in subsequent compute requests. However, unlike other frameworks, our framework does not reveal the bids of the losing participants to any party, including the resource consumer. The only information we disclose is the winner's bid and address. By doing so, our framework achieves a fully privacy-preserving bidding period.

Another potential threat by a malicious resource consumer involves the creation of arbitrary auctions without any genuine intention to outsource computation, aiming to waste the bidders and the manager's resources. When creating an auction, the $\mathcal{RC}$ is required to specify the payment amount (*payment*) willing to offer to the winning bidder, which remains locked throughout the auction's duration. The deposit amount is visible to the bidders and serves as an incentive for them to participate in the auction. The potential for the loss of the $\mathcal{RC}$'s deposit on incorrect/arbitrary auction discourages an economically rational $\mathcal{RC}$ (per Section 4.3) from creating such a fake auction.

6.2 Formal Security Analysis

We analyze the security of our framework in the Universal Composability (UC) framework [5]. The notion of UC security and indistinguishability is captured by the following two definitions.

Definition 2. *(UC-emulation [5]) Let π and ϕ be probabilistic polynomial-time (PPT) protocols. We say that π UC-emulates ϕ if for any PPT adversary $\mathcal{A}$ there exists a PPT adversary $\mathcal{S}$ such that for any balanced PPT environment $\mathcal{Z}$ we have $\text{EXEC}_{\phi,\mathcal{S},\mathcal{Z}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$.*

Definition 3. *(UC-realization [5]) Let $\mathcal{F}$ be an ideal functionality and let π be a protocol. We say that π UC-realizes $\mathcal{F}$ if π UC-emulates the ideal protocol for $\mathcal{F}$.*

We define an ideal functionality, $\mathcal{F}_{\text{PEPPER}}$, which is composed of two independent ideal functionalities: $\mathcal{F}_{\text{bc}}$, and $\mathcal{F}_{\text{auction}}$ as depicted in Figures 2-3. Additionally, we utilize the helper functionality $\mathcal{F}_{\text{sig}}$ [5] and give the formal definition in Figure 8.1. We assume the existence of four tables that store the internal state of $\mathcal{F}_{\text{PEPPER}}$: scTable, aTable, bTable, nTable. These tables are accessible by all functionalities at any time. Specifically, scTable stores the contract address code, and enclave details, aTable stores the auction details, bTable stores the bidder details, and nTable keeps track of the aggregated nonce for the auction. We need to show that a simulator $\mathcal{S}$ can simulate the actions of all honest parties by interacting with

Functionality $\mathcal{F}_{\text{auction}}$

Auction Creation: Upon receiving (sid, createAuction, *payment*, *regTime*, *regEndTime*, *auctionEndTime*, *requiredDeposit*) from $\mathcal{SP}$, $\mathcal{F}_{\text{auction}}$ first computes auction identifier id_A, records (id_A, *payment*, *regTime*, *regEndTime*, *auctionEndTime*, *requiredDeposit*) tuple in aTable and returns it to both $\mathcal{SP}$ and $\mathcal{S}$.

Bidder Registration: Upon receiving a request (sid, registerBidder, id_A, *unonce*, pk_u) from user u, $\mathcal{F}_{\text{auction}}$ first checks if a tuple (id_A, $\cdot$, *regTime*, *regEndTime*, $\cdot$, $\cdot$) exists in aTable and retrieves if so. $\mathcal{F}_{\text{auction}}$ then checks if the current time is within the registration period; if so it adds the tuple (id_A, pk_u, "not-used", "not-winner") to the bTable and returns to u and $\mathcal{S}$. If not, $\mathcal{F}_{\text{auction}}$ returns $\perp$ to both u and $\mathcal{S}$. Then, $\mathcal{F}_{\text{auction}}$ checks if the tuple (id_A, agg_{nonce}) exists in the nTable. If it does, $\mathcal{F}_{\text{auction}}$ retrieves (id_A, agg_{nonce}) and adds the *unonce* to the agg_{nonce} value and updates the tuple. Otherwise it sets $agg_{nonce} = unonce$ and store (id_A, agg_{nonce}) in nTable and returns to u and $\mathcal{S}$.

Bid Submission: Upon receiving a request (sid, bidSubmit, id_A, *ubid*, *udeposit*) from user u, $\mathcal{F}_{\text{auction}}$ first checks if tuple (id_A, $\cdot$, $\cdot$, $\cdot$, $\cdot$, *requiredDeposit*) exists in aTable; if so the *udeposit* amount is equal to the *requiredDeposit*; if so, $\mathcal{F}_{\text{auction}}$ updates the row (id_A, $\cdot$, "not-used", $\cdot$) in bTable to (id_A, $\cdot$, *ubid*, $\cdot$) and returns to u and $\mathcal{S}$. If not, $\mathcal{F}_{\text{auction}}$ returns $\perp$ to both u and $\mathcal{S}$.

Setting Winner: Upon receiving a request (sid, setWinner, id_A, σ_{E}, *minBid*, $addr_{winner}$) from manager $\mathcal{M}$, $\mathcal{F}_{\text{auction}}$ retrieves ($\cdot$, $\cdot$, $addr_{\text{E}}$) from scTable, constructs tuple (sid, Sign, *uid*, *minBid*, $addr_{winner}$, $addr_{\text{E}}$) and forwards to $\mathcal{F}_{\text{sig}}$ where *uid* is user id. Upon receiving (sid, Signature, *uid*, *minBid*, $addr_{winner}$, $addr_{\text{E}}$, σ'_{E}), $\mathcal{F}_{\text{auction}}$ checks if σ_{E} equals to σ'_{E}. If so, $\mathcal{F}_{\text{auction}}$ updates the row in bTable from (id_A, pk_u, $\cdot$, "not-winner") to (id_A, pk_u, $\cdot$, "winner") and sends to all users including $\mathcal{S}$.

Auction Aggregated Nonce Request: Upon receiving a request (sid, getAggNonce, id_A) from user u, $\mathcal{F}_{\text{auction}}$ checks if tuple (id_A, $\cdot$, $\cdot$, $\cdot$) exists in bTable; if so it retrieves ($\cdot$, $\cdot$, *regEndTime*, *auctionEndTime*, $\cdot$) from aTable and checks if the current time is between *regEndTime* and *auctionEndTime*. If so, $\mathcal{F}_{\text{auction}}$ retrieves (id_A, agg_{nonce}) from nTable and returns (sid, agg_{nonce}) to the user u and $\mathcal{S}$.

Auction Bid Detail Request: Upon receiving (sid, getBidDetails, id_A, *uid*) from user u, $\mathcal{F}_{\text{auction}}$ checks if tuple (id_A, $\cdot$, $\cdot$, $\cdot$) exists in bTable; if so retrieves ($\cdot$, $\cdot$, *regEndTime*, $\cdot$, $\cdot$) from aTable and checks if auction registration period has ended. If so, $\mathcal{F}_{\text{auction}}$ then returns bTable to both user u and $\mathcal{S}$.

Initialize Enclave: Upon receiving a request (sid, initEnclave, agg_{nonce}, *uid*) from user u, $\mathcal{F}_{\text{auction}}$ constructs tuple (agg_{nonce}, "not-used", "not-used", "not-used") and adds to eTable. $\mathcal{F}_{\text{auction}}$ returns (sid, initialized, agg_{nonce}) to user u and $\mathcal{S}$.

Generate Report: Upon receiving a request (sid, reportGen, h) from user u, $\mathcal{F}_{\text{auction}}$ generate certificate *ecert*, attestation report *ereport*, updates tuple ($\cdot$, *ecert*, *ereport*, h) in eTable and returns back to user u and $\mathcal{S}$.

Verify Report: Upon receiving a request (sid, reportVer, id_A, *report*) from user u, $\mathcal{F}_{\text{auction}}$ verifies the report. If the report is valid, $\mathcal{F}_{\text{auction}}$ retrieves (id_A, agg_{nonce}) from nTable and the tuple ($\cdot$, *ecert*, $\cdot$, $\cdot$) from eTable. $\mathcal{F}_{\text{auction}}$ then computes $H(ecert \| agg_{nonce})$ and compares it to the *userData* field in the report. If the two values match, $\mathcal{F}_{\text{auction}}$ sends (sid, verified) response to both u and $\mathcal{S}$.

Figure 3: Ideal functionality for auction

the ideal functionalities. Due to space constraints, we provide the proof of the following theorem in Appendix 8.2.

THEOREM 1. *Let $\mathcal{F}_{\text{PEPPER}}$ be an ideal functionality for PEPPER. Let $\mathcal{A}$ be a probabilistic polynomial-time (PPT) adversary for PEPPER, and let $\mathcal{S}$ be an ideal-world PPT simulator for $\mathcal{F}_{\text{PEPPER}}$. PEPPER UC-realizes $\mathcal{F}_{\text{PEPPER}}$ for any PPT distinguishing environment $\mathcal{Z}$.*

In order to prove the theorem mentioned above, we need to prove that no environment (trusted or untrusted) outside the protocol execution can distinguish between the execution of real-world protocols in *PEPPER* and the execution of ideal-world functionalities in $\mathcal{F}_{\text{PEPPER}}$. We use $\mathcal{A}$ to denote real-world adversaries and $\mathcal{S}$ to denote ideal-world adversaries. Our goal is to show that $\mathcal{S}$ can simulate the actions of the real-world protocol by interacting with $\mathcal{F}_{\text{PEPPER}}$ and can produce the same outputs and messages as in the real-world protocols.

7 EXPERIMENTAL RESULTS AND ANALYSIS

This section provides a comprehensive overview of the implementation scope, experimental setup, and evaluation of *PEPPER*.

7.1 Implementation Scope

Our proof of concept implementation of the framework is composed of four main components: the bidder engine, the manager engine, the enclave engine, and the client engine. All of these components were implemented using Go (v.1.17.5). In our enclave engine, we used EGo[3](v.1.2.0) framework to implement enclave-related operations and used the SGX driver (v.2.11.0). To secure communication between the components, we employed TLS (v.1.2). Additionally, we used the curve secp256k1 and the ECIES[4] library (v.2.0.4) in our cryptographic operations.

To conduct our experiments, we utilized Intel SGX as a TEE platform. We deployed a virtual machine on Microsoft Azure with the following specifications: Ubuntu 18.04.6 operating system, Intel Xeon CPU with 3.70 GHz clock speed and 2 processor, 8 GB RAM, and 100 GB solid-state drive (SSD) storage to execute SGX instructions. We refer to this virtual machine as the manager and dedicated this virtual machine to confidential operations using its enclave. In addition, we deployed two more virtual machines with identical specifications to act as Bidders in the same geographical region. These machines have the following specifications: Ubuntu

[3] https://github.com/edgelesssys/ego

[4] https://pkg.go.dev/github.com/ecies/go/v2@v2.0.4

Table 2: Gas consumption and corresponding dollar amounts used in the auction.

Function	Gas Consumption	USD Equivalent
constructor()	2091834	$8.36
RegisterBidder()	434962	$1.73
RefundDeposit()	130000	$0.52
CreateAuction()	117457	$0.46
SetWinner()	41310	$0.16

18.04.6 LTS operating system, Intel Xeon Platinum processor with 2.60 GHz clock speed and 2 cores, 4 GM RAM, and 30 GB SSD. We selected Ethereum as the underlying blockchain platform. We implemented and deployed our auction smart contract using Solidity (v.0.8.0) on the Sepolia testnet[5].

For comprehensive performance analysis of *PEPPER*'s remote attestation, we further implemented Trustee's remote attestation mechanism as described in [15] within the EGo framework (same as *PEPPER*), utilizing Intel's Data Center Attestation Primitives (DCAP) model. We will present our comparison results in Figure 5.

7.2 Results and Analysis

In Table 2, we present the total gas consumption of functions used in the auction smart contract. The gas consumption analysis details the total cost of creating an auction and participating in it. For instance, deploying smart contract, *i.e.,* constructor function, on the Sepolia testnet required a total of 2091834 units of gas, which is equivalent to $8.36 at an average rate of 2.5 GWei (*i.e.,* one-billionth of one Ether) per unit of gas and a cost of $1600 per Ether. It is important to note that the deployment cost is a one-time expense.

The contract can be utilized by both the bidders and the auctioneers multiple times and even simultaneously, thereby amortizing the cost of running auctions. In contrast, the transaction that sets the winner (i.e., the SetWinner() function) only costs $0.52 due to the low gas consumption of the ecrecover opcode. We emphasize that GetAggregatedNonce() and GetBidDetails() functions do not cost any Ether as they do not modify any state; hence, excluded from Table 2. Moreover, the deployment gas amount remains constant irrespective of the number of registered bidders or auctioneers. Finally, *PEPPER* drastically reduces gas consumption by delegating the auction winner logic to an enclave, which runs off the chain.

In contrast, Trustee [15] requires the redeployment of the auction contract for every individual auction event, incurring a cost of $5.4 at the current Ether rate. These characteristics make *PEPPER* a more cost-effective and scalable solution in comparison. For example, in 1000 auctions, the total cost for *PEPPER* amounts to $8.36, while Trustee incurs a total cost of $5.4 × 1000.

We benchmark enclave-related operations, including signing the executable (depicted in turquoise; bottom), generating the report (depicted in green; middle), and building the enclave (depicted in orange; top), as depicted in Figure 4. We averaged the numbers over 100 iterations to provide a robust analysis. The X-axis in Figure 4 shows the number of bidders, and the Y-axis shows the time it takes to complete the operations in milliseconds. Enclave building is a process of converting the application logic implemented in a high-level language into an executable that enables loading it into an enclave. The executable is then signed before being loaded into the enclave.

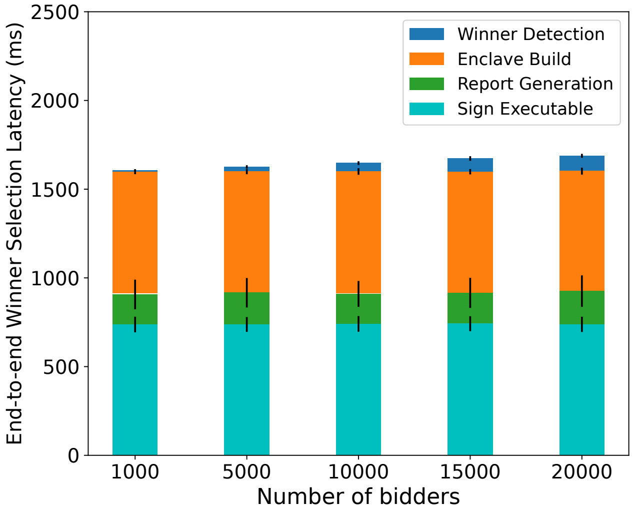

Figure 4: Increasing the number of bidders has negligible impact on the latency which shows scalability.

The purpose behind signing the executable is to check if the loaded executable is indeed legitimate and has not been altered by a malicious operating system during loading. Our experiments showed that the operations of signing the executable, generating reports, and building the enclave take approximately 1600 milliseconds, and these times are independent of the number of bidders.

We benchmarked the winner selection performance in terms of latency. In these experiments, we generated a series of encrypted bids across different numbers of bidders and measured the time it takes for the enclave to decrypt the corresponding number of encrypted bids, find the minimum among them, and construct a signature that will be used by the smart contract to verify the winner's address (*i.e.,* operations in Lines 1–17 of Protocol 5). As shown in Figure 4 (the blue bar at the top), the time it takes to perform the aforementioned operations marginally increases with the number of participants. For instance, it only takes 100 milliseconds for the enclave to identify the winner's bid with 20 thousand bidders.

We also compare the performance of our proposed remote attestation with Trustee [15].

In Figure 5, the X-axis represents the number of bidders, and the Y-axis depicts the time in seconds required to complete the corresponding number of remote attestation requests. Our focus in this experiment is on assessing how both *PEPPER* and Trustee respond to concurrent remote attestation requests originating from the bidders. To this end, we initiated concurrent remote attestation requests, from bidders, using *PEPPER* and Trustee. For each request, we permitted up to 5 retries in case the remote attestation request failed due to issues, such as request timeouts. While the number of remote attestation requests in a typical PEC environment might not be as numerous, our objective is to demonstrate that *PEPPER*'s performance excels even in a generic remote attestation approach when compared to the most relevant method. For more realistic

[5] https://sepolia.etherscan.io/

Table 3: Communication Complexity of *PEPPER* in Comparison with Trustee.

	Auction Creation.	Bidder Registration.	Remote Attestation.	Contract Deployment.
PEPPER	$O(1)$	$O(\|\mathbb{B}\|)$	$O(1)$	$O(1)$
Trustee	$O(1)$	$O(\|\mathbb{B}\|)$	$O(\|\mathbb{B}\|)$	$O(1)$

scenarios, for instance, with 50 bidders (not depicted in the Figure), *PEPPER* still outperformed Trustee, on average 14.5ms faster completion time of remote attestation requests (*PEPPER*: 284.2ms, Trustee: 298.7ms)—5%-10% speed up on an average.

In the experiment with a high number of bidders, both approaches exhibited similar performance, with *PEPPER* marginally outperforming Trustee for lower numbers of bidders, specifically at 2000 and 3000 – roughly a 1.23× speedup on average. However, as the number of bidders increased, Trustee's remote attestation requests encountered connection timeouts, leading to delays for 4000 bidders and a sudden jump in remote attestation latency. On the other hand, *PEPPER* maintained a more consistent and robust performance, avoiding any significant delays in request completion, and completed all the remote attestation requests 2.15× faster. Although both methods experienced delays with settings of 5000 and 6000 bidders, *PEPPER*'s performance remained more stable, consistently taking less time across all categories. In a setting with 6000 bidders, the server that hosts the enclave began to slow down in responding to both types of requests, as evidenced by the small difference in the performance of the two approaches. Consequently, we limited the total number of bidders in this experiment to 6000, as the server hosting the enclave failed to respond and crashed with more than 6000 concurrent requests. The failure was due to an "out-of-memory" error, leading to a cut in the connection.

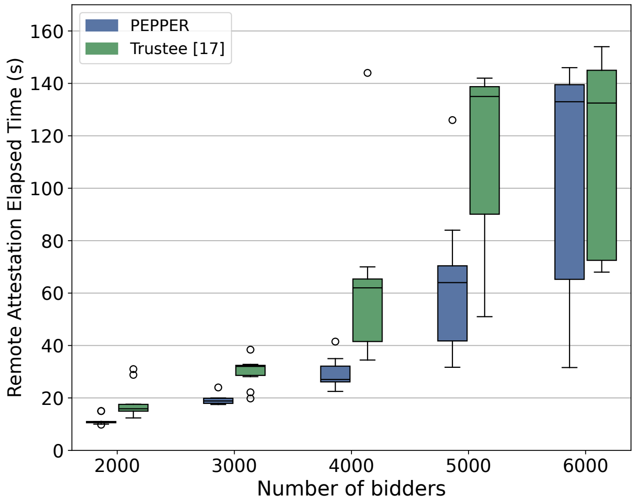

Figure 5: Comparison of remote attestation times for various numbers of bidders, showing *PEPPER*'s faster performance compared to Trustee [15] across all categories.

7.3 Complexity Analysis

We also assess the communication complexity of *PEPPER* in Table 3. When creating an auction, the $\mathcal{RC}$ submits a single transaction to the smart contract for registration, which results in a constant time communication complexity. With our improved remote attestation protocol, generating an attestation request results in a communication complexity of $O(1)$, since the nonce values of bidders are aggregated and only one single request is sent to the enclave. This is a significant improvement compared to the original remote attestation protocol, in which the communication complexity increases linearly with the number of bidders, *i.e.,* $O(|\mathbb{B}|)$ for $\mathbb{B}$ bidders. The communication complexity for auction registration by $\mathbb{B}$ bidders is $O(|\mathbb{B}|)$ as each bidder needs to send a separate request to the smart contract. Finally, deploying a smart contract in the *PEPPER* system has a constant communication complexity for $\mathcal{SP}$, as it involves a single transaction sent to the network.

8 CONCLUSION AND FUTURE WORK

In this paper, we proposed a decentralized framework that enables users to select the PEC servers at the pervasive edge to outsource their computation. The proposed solution is designed to address the challenges of providing proper incentives and a trustworthy environment to persuade end users to participate in the PEC ecosystem, as well as to ensure that the allocation of computation tasks is fair, transparent, and privacy-preserving. We implemented and evaluated our framework on top of Ethereum using Intel SGX, and demonstrated its effectiveness through experimentation. Furthermore, we introduced an innovative method for remote attestation, which is particularly advantageous in the context of PEC. Notably, our experiments revealed that *PEPPER*'s remote attestation outpaces standard remote attestation, offering a more efficient alternative.

As future work, our research will probe into challenges related to potentially malicious bidders within the PEC system. Specifically, we plan to explore the use of verifiable computation techniques to deter bidders from submitting incorrect results. Additionally, we will investigate means to build and utilize reputation within the PEC ecosystem, enabling clients to review and rate bidders based on the quality of the executed service.

ACKNOWLEDGEMENTS

This research was partially funded by the US National Science Foundation under grants #2148358 and #1914635, and the US Department of Energy grant #DE-SC0023392. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US federal agencies.

REFERENCES

[1] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. 1999. DHAES: An Encryption Scheme Based on the Diffie-Hellman Problem. *IACR Cryptol. ePrint Arch.* 1999 (1999), 7.

[2] Gaurav Baranwal, Dinesh Kumar, and Deo Prakash Vidyarthi. 2022. BARA: A blockchain-aided auction-based resource allocation in edge computing enabled industrial internet of things. *Future Generation Computer Systems* 135 (2022), 333–347.

[3] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3917–3934. https://www.usenix.org/conference/usenixsecurity22/presentation/borrello

[4] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. 2020. Zether: Towards privacy in a smart contract world. In *International Conference on Financial Cryptography and Data Security*. Springer International Publishing, Cham, 423–443.

[5] Ran Canetti. 2004. Universally composable signature, certification, and authentication. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* IEEE, Pacific Grove, CA, 219–233.

[6] Valeria Cardellini, Vittoria De Nitto Personé, Valerio Di Valerio, Francisco Facchinei, Vincenzo Grassi, Francesco Lo Presti, and Veronica Piccialli. 2016. A game-theoretic approach to computation offloading in mobile cloud computing. *Mathematical Programming* 157, 2 (2016), 421–449.

[7] Guoxing Chen, Mengyuan Li, Fengwei Zhang, and Yinqian Zhang. 2019. Defeating speculative-execution attacks on SGX with HyperRace. In *2019 IEEE Conference on Dependable and Secure Computing (DSC)*. IEEE, Hangzhou, China, 1–8.

[8] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2019. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, Stockholm, Sweden, 185–200.

[9] K. Chin, K. Emura, K. Omote, and S. Sato. 2022. A Sealed-bid Auction with Fund Binding: Preventing Maximum Bidding Price Leakage. In *2022 IEEE International Conference on Blockchain (Blockchain)*. IEEE Computer Society, Los Alamitos, CA, USA, 398–405.

[10] CISCO. 2023. Cisco Annual Internet Report (2018–2023) White Paper. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[11] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. 2019. FastKitten: Practical Smart Contracts on Bitcoin.. In *USENIX security symposium*. USENIX Association, Santa Clara, CA, 801–818.

[12] Sean Dougherty, Reza Tourani, Gaurav Panwar, Roopa Vishwanathan, Satyajayant Misra, and Srikathyayani Srikanteswara. 2021. APECS: A distributed access control framework for pervasive edge computing services. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA, 1405–1420.

[13] Hisham S Galal and Amr M Youssef. 2018. Succinctly verifiable sealed-bid auction smart contract. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, Cham, 3–19.

[14] Hisham S Galal and Amr M Youssef. 2018. Verifiable sealed-bid auction on the ethereum blockchain. In *International Conference on Financial Cryptography and Data Security*. Springer-Verlag, Berlin, Heidelberg, 265–278.

[15] Hisham S Galal and Amr M Youssef. 2019. Trustee: full privacy preserving vickrey auction on top of ethereum. In *International conference on financial cryptography and data security*. Springer, St. Kitts, 190–207.

[16] Yaodong Huang, Jiarui Zhang, Jun Duan, Bin Xiao, Fan Ye, and Yuanyuan Yang. 2022. Resource Allocation and Consensus of Blockchains in Pervasive Edge Computing Environments. *IEEE Transactions on Mobile Computing* 21, 9 (2022), 3298–3311. https://doi.org/10.1109/TMC.2021.3053230

[17] Vibha Jain and Bijendra Kumar. 2022. Auction based cost-efficient resource allocation by utilizing blockchain in fog computing. *Transactions on Emerging Telecommunications Technologies* 33, 7 (2022), e4469.

[18] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. 2016. Intel software guard extensions: EPID provisioning and attestation services. *White Paper* 1, 1-10 (2016), 119.

[19] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. 2018. Integrating remote attestation with transport layer security.

[20] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, San Jose, CA, 839–858.

[21] Michal Król, Alberto Sonnino, Argyrios Tasiopoulos, Ioannis Psaras, and Etienne Rivière. 2020. PASTRAMI: privacy-preserving, auditable, Scalable & Trustworthy Auctions for multiple items. Proceedings of the 21st International Middleware Conference, Delft, 296–310.

[22] Honglei Li and Weilian Xue. 2021. A blockchain-based sealed-bid e-auction scheme with smart contract and zero-knowledge proof. *Security and Communication Networks* 2021 (2021), 1–10.

[23] Li Li, Yue Li, and Ruotong Li. 2021. Double auction-based two-level resource allocation mechanism for computation offloading in mobile blockchain application. *Mobile Information Systems* 2021 (2021), 1–15.

[24] Xuelian Liu, Jigang Wu, Long Chen, and Chengpeng Xia. 2019. Efficient auction mechanism for edge computing resource allocation in mobile blockchain. In *2019 IEEE 21st international conference on high performance computing and communications; IEEE 17th international conference on smart city; IEEE 5th international conference on data science and systems (HPCC/SmartCity/DSS)*. IEEE, Zhangjiajie, China, 871–876.

[25] Yujiong Liu, Shangguang Wang, Jie Huang, and Fangchun Yang. 2018. A computation offloading algorithm based on game theory for vehicular edge networks. In *2018 IEEE International Conference on Communications (ICC)*. IEEE, Kansas City, MO, 1–6.

[26] Minghui Liwang, Jiexiang Wang, Zhibin Gao, Xiaojiang Du, and Mohsen Guizani. 2019. Game theory based opportunistic computation offloading in cloud-enabled IoV. *Ieee Access* 7 (2019), 32551–32561.

[27] Mohamed-Ayoub Messous, Sidi-Mohammed Senouci, Hichem Sedjelmaci, and Soumaya Cherkaoui. 2019. A game theory based efficient computation offloading in an UAV network. *IEEE Transactions on Vehicular Technology* 68, 5 (2019), 4964–4974.

[28] Stylianos Mystakidis. 2022. Metaverse. *Encyclopedia* 2, 1 (2022), 486–497.

[29] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2016. Pinocchio: Nearly practical verifiable computation. *Commun. ACM* 59, 2 (2016), 103–112.

[30] Emrah Sariboz, Kartick Kolachala, Gaurav Panwar, Roopa Vishwanathan, and Satyajayant Misra. 2021. Off-chain execution and verification of computationally intensive smart contracts. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, Sydney, Australia, 1–3.

[31] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. 2018. Supporting third party attestation for Intel® SGX with Intel® data center attestation primitives. , 12 pages.

[32] Emiliano Sisinni, Abusayeed Saifullah, Song Han, Ulf Jennehag, and Mikael Gidlund. 2018. Industrial internet of things: Challenges, opportunities, and directions. *IEEE transactions on industrial informatics* 14, 11 (2018), 4724–4734.

[33] Alberto Sonnino, Michał Król, Argyrios G Tasiopoulos, and Ioannis Psaras. 2019. Asterisk: Auction-based shared economy resolution system for blockchain. *arXiv preprint arXiv:1901.07824* (2019).

[34] Yuhu Sun, Qiang He, Lianyong Qi, Wajid Rafique, and Wanchun Dou. 2020. Dpoda: Differential privacy-based online double auction for pervasive edge computing resource allocation. In *Proceedings of the 2nd ACM International Symposium on Blockchain and Secure Critical Infrastructure*. Association for Computing Machinery, New York, NY, USA, 130–141.

[35] Hamed Tabrizchi and Marjan Kuchaki Rafsanjani. 2020. A survey on security challenges in cloud computing: issues, threats, and solutions. *The journal of supercomputing* 76, 12 (2020), 9493–9532.

[36] Tarik Taleb, Konstantinos Samdanis, Badr Mada, Hannu Flinck, Sunny Dutta, and Dario Sabella. 2017. On multi-access edge computing: A survey of the emerging 5G network edge cloud architecture and orchestration. *IEEE Communications Surveys & Tutorials* 19, 3 (2017), 1657–1681.

[37] Reza Tourani, Srikathyayani Srikanteswara, Satyajayant Misra, Richard Chow, Lily Yang, Xiruo Liu, and Yi Zhang. 2020. Democratizing the Edge: A Pervasive Edge Computing Framework. *arXiv preprint arXiv:2007.00641* 1, 1 (2020), 1–7.

[38] Xiaojie Wang, Zhaolong Ning, and Song Guo. 2020. Multi-agent imitation learning for pervasive edge computing: A decentralized computation offloading algorithm. *IEEE Transactions on Parallel and Distributed Systems* 32, 2 (2020), 411–425.

[39] Will Warren and Amir Bandeali. 2017. 0x: An open protocol for decentralized exchange on the Ethereum blockchain. , 04–18 pages.

[40] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[41] Rui Yuan, Yu-Bin Xia, Hai-Bo Chen, Bin-Yu Zang, and Jan Xie. 2018. Shadoweth: Private smart contract on public blockchain. *Journal of Computer Science and Technology* 33, 3 (2018), 542–556.

[42] Jixian Zhang, Wenlu Lou, Hao Sun, Qian Su, and Weidong Li. 2022. Truthful auction mechanisms for resource allocation in the Internet of Vehicles with public blockchain networks. *Future Generation Computer Systems* 132 (2022), 11–24.

[43] Yahui Zhang, Min Zhao, Tingquan Li, and Huan Han. 2020. Survey of Attacks and Defenses against SGX. In *2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC)*. IEEE, Chongqing, China, 1492–1496. https://doi.org/10.1109/ITOEC49072.2020.9141835

[44] Shuchen Zhou and Waqas Jadoon. 2020. The partial computation offloading strategy based on game theory for multi-user in mobile edge computing environment. *Computer Networks* 178 (2020), 107334.

8.1 UC Functionalities

Functionality $\mathcal{F}_{\text{sig}}$

Key Generation: Upon receiving a value (KeyGen, sid) from some party S, verify that sid $= (S, \text{sid}')$ for some sid$'$. If not, then ignore the request. Else, hand (KeyGen, sid) to the adversary. Upon receiving (VerificationKey, sid, v) from the adversary, output (VerificationKey, sid, v) to S, and record the pair (S, v).

Signature Generation: Upon receiving a value (Sign, sid, m) from S, verify that sid $= (S, \text{sid}')$ for some sid$'$. If not, then ignore the request. Else, send (Sign, sid, m) to the adversary. Upon receiving (Signature, sid, m, σ) from the adversary, verify that no entry $(m, \sigma, v, 0)$ is recorded. If it is, then output an error message to S and halt. Else, output (Signature, sid, m, σ) to S, and record the entry $(m, \sigma, v, 1)$.

Signature Verification: Upon receiving a value (Verify, sid, m, σ, v_0) from some party P, hand (Verify, sid, m, σ, v_0) to the adversary. Upon receiving (Verified, sid, m, φ) from the adversary do:

(1) If $v_0 = v$ and the entry $(m, \sigma, v, 1)$ is recorded, then set $f = 1$. (This condition guarantees completeness: If the verification key v_0 is the registered one and σ is a legitimately generated signature for m, then the verification succeeds.)
(2) Else, if $v_0 = v$, the signer is not corrupted, and no entry $(m, \sigma_0, v, 1)$ for any σ_0 is recorded, then set $f = 0$ and record the entry $(m, \sigma, v, 0)$. (This condition guarantees unforgeability: If v_0 is the registered one, the signer is not corrupted, and never signed m, then the verification fails.)
(3) Else, if there is an entry (m, σ, v_0, f_0) recorded, then let $f = f_0$. (This condition guarantees consistency: All verification requests with identical parameters will result in the same answer.)
(4) Else, let $f = \varphi$ and record the entry $(m, \sigma, v_0, \varphi)$.

Output (Verified, id, m, f) to P.

Figure 6: Ideal functionality for signature generation and verification [5]

8.2 Proof of Theorem 1

Proof 1. *We separate the details of protocol execution into two worlds and demonstrate that $\mathcal{Z}$'s view remains the same in both worlds.*

Part 1: We consider the system setup and auction initiation protocols described in Protocols 1 and 2.

1) Case 0: Bidders and Manager are honest.

a) **Real-world**: In the real-world (Protocols 1 and 2), $\mathcal{SP}$ generates key pair (pk, sk), implements auction smart contract $\mathcal{SC}$, deploys it on BC. The PEC servers in the vicinity who wants to participate in auction (i.e., bidders) generate their key pairs, (pk_i, sk_i), $i \in [1 \dots n]$ where $n = |\mathbb{B}|$. Next, $\mathcal{SP}$ creates auction on $\mathcal{SC}$ with auction id id_A. $\mathcal{Z}$ sees the SC, pk's of every entity and the id_A. Next, each $b_i \in \mathbb{B}$ samples $nonce_i$ from $\mathbb{Z}^+$ and registers auction on SC. The provided nonces are aggregated on SC. Once the registration period is over, the manager $\mathcal{M}$ retrieves the aggregated nonce agg_{nonce} and initializes enclave E with it. Note that since both $\mathcal{M}$ and $\mathbb{B}$ are honest, the secret keys of the parties are not visible to $\mathcal{Z}$. It is important to remember per our adversary model, the $\mathcal{SP}$ is trusted. Hence, the view of $\mathcal{Z}$ will be (SC, pk, $pk_1 \dots pk_n$, id_A, $nonce_1 \dots nonce_n$, agg_{nonce}, λ) where λ is the security parameter.

b) **Ideal-world**: In the ideal-world, $\mathcal{S}$ picks security parameter λ, and sends (KeyGen, sid) where sid is id of $\mathcal{SP}$ and sends it to $\mathcal{F}_{\text{sig}}$ and receives (VerificationKey, sid, pk). Then, $\mathcal{S}$ calls $\mathcal{F}_{\text{bc}}$ with (deploy, SC, $code$, $addr_{\text{E}}$) to simulate the deployment of contract on BC. $\mathcal{S}$ makes n unique calls to $\mathcal{F}_{\text{sig}}$ with (KeyGen, sid_i) where $i \in [1 \dots n]$. $\mathcal{F}_{\text{sig}}$ returns (VerificationKey, sid$_i$, pk_i) to $\mathcal{Z}$. Next, $\mathcal{S}$ simulating $\mathcal{SP}$ creates auction by sending (createAuction, *payment*, *regTime*, *regEndTime*, *auctionEndTime*, *requiredDeposit*) to $\mathcal{F}_{\text{auction}}$ and receives id_A. Next, to simulate the bidder registration, $\mathcal{S}$ generates n nonce's, and makes n calls to $\mathcal{F}_{\text{auction}}$ with (registerBidder, id_A, $unonce$, pk_u). To simulate retrieval of agg_{nonce} for the auction with id id_A, $\mathcal{S}$ sends (getAggNonce, id_A) to $\mathcal{F}_{\text{auction}}$. Upon receiving agg_{nonce}, $\mathcal{S}$ sends (initEnclave, agg_{nonce}, uid) to $\mathcal{F}_{\text{auction}}$ to initialize the enclave. The view of $\mathcal{Z}$ remains the same as in the real-world, i.e., (SC, pk, $pk_1 \dots pk_n$, id_A, $nonce_1 \dots nonce_n$, agg_{nonce}, λ).

2) Case 1: Malicious bidders and honest manager.

a) **Real-world**: Per our adversary model, some bidders may act maliciously. As in previous case, the $\mathcal{SP}$ generates (pk, sk) pair and deploys SC on BC. Bidder generate their keypairs, (pk_i, sk_i), $i \in [1 \dots n]$ where $n = |\mathbb{B}|$. Then, $\mathcal{SP}$ creates auction as in previous step and receives id_A. $\mathcal{Z}$ has access to the public and private keys of malicious bidders, the public keys of honest bidders, $\mathcal{SP}$'s public key and id_A. Next, each bidder provides their random nonce to smart contract by registering. The provided nonce's are aggregated on SC. The malicious bidders may attempt to notify $\mathcal{M}$ to initialize the E before the end of registration period; however, SC would check and revert the request that attempts to retrieve aggregated nonce. Manager, since honest, retrieves agg_{nonce} and initializes enclave. Let the set of malicious bidders be $\mathbb{B}'$, such that $\mathbb{B}' \subset \mathbb{B}$. The view of $\mathcal{Z}$ will be (SC, $\{pk_i, sk_i\}_{i \in \mathbb{B}'}$, $\{pk_j\}_{j \in \mathbb{B}}$, id_A, $nonce_1 \dots nonce_n$, agg_{nonce}).

b) **Ideal-world**: As in Case 0, $\mathcal{S}$ simulates the role of $\mathcal{SP}$ and generates (pk, sk) from $\mathcal{F}_{\text{sig}}$. $\mathcal{S}$ sends (deploy, SC, $code$, $addr_{\text{E}}$) to $\mathcal{F}_{\text{bc}}$ and receives id_A for the auction. For the honest bidders, $\mathbb{B} - \mathbb{B}'$, $\mathcal{S}$ creates $pk \leftarrow\!\!\$ \{0,1\}^k$. Corrupt bidders in $\mathbb{B}' \subset \mathbb{B}$ are handled by $\mathcal{A}$. Following the same approach as in Case 0's ideal-world, $\mathcal{S}$ simulates the actions of $\mathbb{B}$ by creating $nonce$ for every $b_i \in \mathbb{B}$ and calls $\mathcal{F}_{\text{auction}}$ to register by sending (registerBidder, id_A, $unonce$, pk_u) for each bidder. Next, $\mathcal{S}$ retrieves agg_{nonce} for the auction with id id_A. Upon receiving agg_{nonce}, $\mathcal{S}$ simulates initialization of enclave by sending (initEnclave, agg_{nonce}, uid) to $\mathcal{F}_{\text{auction}}$. The view of $\mathcal{Z}$ will be (SC, $\{pk_i, sk_i\}_{i \in \mathbb{B}'}$, $\{pk_j\}_{j \in \mathbb{B} \setminus \mathbb{B}'}$, id_A, $nonce_1 \dots nonce_n$, agg_{nonce}) which is the same as in the real-world.

3) Case 2: Honest bidders and malicious manager.

a) **Real-world**: As in previous cases, $\mathcal{SP}$ generates key pairs and deploys SC on BC. Next, $\mathcal{SC}$ generates auction with id_A on BC. Bidders interested in auction, generates key-pairs, samples *nonce* and register for the auction with id_A. Once registration period is over, $\mathcal{M}$ retrieves the aggregated nonce agg_{nonce} from *SC* and initializes the E. During the initialization, the malicious $\mathcal{M}$ can initiate the E with different *nonce* than the legitimate agg_{nonce}. However, this attempt will be caught by the honest biddders during remote attestation—the enclave generated report will include *nonce* that is different than the one on *SC* that is publicly available to all users. Hence, the view of $\mathcal{Z}$ is (SC, pk, $pk_1 \dots pk_n$, id_A, $nonce_1 \dots nonce_n$, agg_{nonce}, λ) where λ is the security parameter.

b) **Ideal-world**: As in ideal-world of Case 1's, the simulator $\mathcal{S}$ is responsible for simulating the key-pair generation, the deployment of the smart contract $\mathcal{SC}$, and the creation of the auction on the blockchain for the service provider $\mathcal{SP}$. For each bidder, $\mathcal{S}$ simulates the key generation and generates a nonce, which is used to call the function $\mathcal{F}_{\text{auction}}$ with the arguments (registerBidder, id_A, $unonce$, pk_u). Then, $\mathcal{Z}$ retrieves the agg_{nonce} for the auction with id id_A by sending (getAggNonce, id_A) to $\mathcal{F}_{\text{auction}}$. During initialization of E, if $\mathcal{Z}$ use wrong agg_{nonce} than the legitimate one, the $\mathcal{S}$ will reveal it during the verification of report on behalf of honest bidders. Thus, the view of $\mathcal{Z}$ will be (SC, pk, $pk_1 \dots pk_n$, id_A, $nonce_1 \dots nonce_n$, agg_{nonce}, λ) which is same as real-world case.

4) Case 3: Malicious bidders and malicious manager.

a) **Real-world**: As in Case 1's real-world, $\mathcal{SP}$ generates key-pairs, deploys *SC* and creates auction. Similarly, interested bidders generate key-pairs, samples nonce and register for the auction. Upon end of registration period, $\mathcal{M}$ retrieves auction. Since malicious, the $\mathcal{M}$ attempts to initialize E with different agg_{nonce}. However, the M's attempt will be revealed during remote attestation by the honest bidders. Hence, the view of $\mathcal{Z}$ is (SC, $\{pk_i, sk_i\}_{i\in\mathbb{B}'}$, $\{pk_j\}_{j\in\mathbb{B}}$, id_A, $nonce_1 \dots nonce_n$, agg_{nonce}) where $\mathbb{B}'$ is set of malicious bidders such that $\mathbb{B}' \subset \mathbb{B}$.

b) **Ideal-world**: In the ideal-world scenario of Case 1, the simulator $\mathcal{S}$ simulates the role of the service provider by generating key-pairs, deploying the smart contract *SC* on the blockchain, and creating the auction. Honest bidders generate their key-pairs, sample nonces, and register for the auction. The environment handles the corrupted bidders. Once the registration period ends, the environment retrieves the aggregate nonce agg_{nonce} from the smart contract *SC*. However, as a malicious party, the environment may attempt to initialize the enclave with a different value than the legitimate agg_{nonce}. This attempt will fail during remote attestation because $\mathcal{S}$ will abort the protocol execution on behalf of the honest parties. Thus, the environment's view is (SC, $\{pk_i, sk_i\}_{i\in\mathbb{B}'}$, $\{pk_j\}_{j\in\mathbb{B}}$, id_A, $nonce_1 \dots nonce_n$, agg_{nonce}).

Part 2: We now consider the remote attestation and bid submission protocol that runs between the bidders and enclave as described in Protocol 3 and 4. We note that the enclave is honest party per our adversary model; hence, we do not consider any malicious case from the enclave. Morever, the ECIES related operations for the honest parties are omitted due to them being honest by definition.

1) Case 0: Honest bidders.

a) **Real-world**: In the real world of Protocol 3, E generates x509 certificate $Cert_{\text{E}}$ and concatenates it with agg_{nonce}. Next, enclave computes the digest of the result and use it to generate a attestation report for the bidders $\mathbb{B}$. The bidders retrieves the pk_E and $Cert_{\text{E}}$ from enclave. Then they all retrieve the agg_{nonce} individually from *SC* and re-compute ($Cert_E \| agg_{nonce}$). Bidders then compare the value in the *userData* field of the report to the digest. If the values are equal, the remote attestation step is now completed. Each bidder encrypts their bid and calls the SendBid function of *SC* with (id_A, $cbid_i$, $deposit$), where $cbid_i$ represents the encrypted bid of the i'th bidder. Once the bid submission period is over, $\mathcal{M}$ retrieves the bids of auction id_A and sends them to E for the decryption. The view of $\mathcal{Z}$ is (pk_{E}, $Cert_{\text{E}}$, $report$, agg_{nonce}, $\{nonce_i, pk_i, cbid_i, deposit_i\}_{i\in\mathbb{B}}$).

b) **Ideal-world**: In the ideal world, $\mathcal{S}$ sends (reportGen, h) to $\mathcal{F}_{\text{auction}}$ and receives ($agg_{nonce}, ecert, ereport, h$). Then, to simulate the actions of bidders, $\mathcal{F}_{\text{auction}}$ sends (reportVer, id_A, $report$) for each $b_i \in \mathbb{B}$ to $\mathcal{F}_{\text{auction}}$. If the verification is true, $\mathcal{S}$ receives ($verified$). Next, $\mathcal{S}$ simulates the bid submission operations of bidder n times each being unique for n different bidders by sending (bidSubmit, id_A, $ubid$, $udeposit$) to $\mathcal{F}_{\text{auction}}$. Since the bidders are honest, the view of $\mathcal{Z}$ is (pk_{E}, $Cert_{\text{E}}$, $report$, agg_{nonce}, $\{nonce_i, pk_i, cbid_i, deposit_i\}_{i\in\mathbb{B}}$) which is same as real-world.

2) Case 1: Malicious bidders.

a) **Real-world**: Similar to the previous scenario, the enclave produces $Cert_{\text{E}}$ and an attestation report. Bidders retrieve the public key (pk) of the enclave, the attestation report ($report$), the certificate, and the agg_{nonce} separately and use them to validate the report. If the verification is true, each bidder generates a random bid (bid_i) and encrypts it before submitting it to *SC*. Malicious bidders can discard the auction at this stage by claiming that the *report* is incorrect, but this does not affect the system's security. Next, bidders create a shared-secret key (ss) and produce a (k_{MAC}, k_{ENC}) pair using KDF and KAF algorithms. Each bidder encrypts their bid using k_{ENC} and submits it to *SC* along with their *deposit*. Malicious bidders can submit incorrect ciphertext instead of encrypting their bid with k_{ENC}. However, during the decryption process, the E will exclude those bids, and the *SC* code will prevent the malicious bidders from retrieving their *deposit*. Hence, there isn't any rational financial reason for acting malicious. The view of $\mathcal{Z}$ is (pk_{E}, $Cert_{\text{E}}$, $report$, agg_{nonce}, $\{nonce_i, pk_i, cbid_i, deposit_i\}_{i\in\mathbb{B}}$, $\{sk_j, pk_j\}_{j\in\mathbb{B}\setminus\mathbb{B}'}$) where $\mathbb{B}'$ is honest bidders.

b) **Ideal-world**: In an ideal world, $\mathcal{S}$ would send (reportGen, h) to $\mathcal{F}_{\text{auction}}$ and receive (agg_{nonce}, $ecert$, $ereport$, h). Then, to simulate the actions of bidders, $\mathcal{F}_{\text{auction}}$ would send (reportVer, id_A, $report$) for each $b_i \in \mathbb{B}$ to $\mathcal{F}_{\text{auction}}$. If the verification is true, $\mathcal{S}$ receives ($verified$). Once the remote attestation period is over, $\mathcal{S}$ generates a key pair (pk_i, sk_i) for each bidder. For the malicious bidders, $\mathcal{S}$ sends their key pairs to $\mathcal{Z}$. Both $\mathcal{S}$ and $\mathcal{Z}$ use the same KAF and KDF functions, resulting in k_{MAC} and k_{ENC} variables being the same. For the honest bidders, $\mathcal{S}$ simulates bid generation and encryption operations, while for the corrupted bidders, $\mathcal{Z}$ samples bid_i and encrypts their bids, which are then sent to $\mathcal{S}$. If $\mathcal{Z}$ uses different k_{ENC}, $\mathcal{S}$ will discard them during decryption, resulting in the failure of the attack. Eventually, all the bids will get decrypted, and $\mathcal{S}$ will identify the winning bid. Hence, the view of $\mathcal{Z}$ is (pk_{E}, $Cert_{\text{E}}$, $report$, agg_{nonce}, $\{nonce_i, pk_i, cbid_i, deposit_i\}_{i\in\mathbb{B}}$, $\{sk_j, pk_j\}_{j\in\mathbb{B}\setminus\mathbb{B}'}$) which is same as real-world.

Part 3: We now consider winner announcement protocol that facilitates the setting of the winner address and involves interactions between the enclave and the manager, as outlined in Protocol 5.

1) Case 0: Honest Manager.

a) **Real-world**: The manager $\mathcal{M}$ retrieves the signature of the enclave E, the winner bid *minBid*, and the address of the winner $addr_{winner}$ from the manager. $\mathcal{M}$ constructs transaction tx_M and calls the SetWinner function. The *SC* first verifies if the σ_{E} is signed by the enclave. If so, the *SC* updates the winner address and the winner bid on the *SC*. Hence, the view of $\mathcal{Z}$ will be (*minBid*, $addr_{winner}$, $addr_{\mathrm{E}}$, σ_{E}).

b) **Ideal-world**: In the ideal world, $\mathcal{S}$ receives (σ_{E}, *minBid*, $addr_{winner}$) from the enclave. The $\mathcal{S}$ sends (setWinner, id_A, σ_{E}, *minBid*, $addr_{winner}$) to $\mathcal{F}_{\text{auction}}$. Upon receiving it, $\mathcal{F}_{\text{auction}}$ first retrieves the $addr_E$ from scTable and sends (Sign, *uid*, *minBid*, $addr_{winner}$, $addr_{\mathrm{E}}$) to $\mathcal{F}_{\text{sig}}$. Upon receiving (Signature, *uid*, *minBid*, $addr_{winner}$, $addr_{\mathrm{E}}$, σ'_{E}), $\mathcal{F}_{\text{auction}}$ checks if σ_{E} equals σ'_{E}. If so, it updates the winner address to $addr_{winner}$. Thus, the view of $\mathcal{Z}$ will be (*minBid*, $addr_{winner}$, $addr_{\mathrm{E}}$, σ_{E}), which is the same as the real world.

2) Case 1: Malicious Manager .

a) **Real-world**: As in the previous case, the $\mathcal{M}$ retrieves (σ_{E}, *minBid*, $addr_{winner}$) from the E. Instead of assigning the winner address to $addr_{winner}$, the manager may attempt to assign it to another entity. However, since the σ_{E} is verified on-chain, this attempt will fail during verification. As in the previous case, the $\mathcal{M}$ sends the transaction to the *BC*, and if the signature is valid, the winner address is updated on the *SC*. Thus, the view of $\mathcal{Z}$ is (*minBid*, $addr_{winner}$, $addr_{\mathrm{E}}$, σ_{E}).

b) **Ideal-world**: As in the previous case's ideal-world, $\mathcal{S}$ receives (σ_{E}, *minBid*, $addr_{winner}$). $\mathcal{Z}$ may attempt to change the $addr_{winner}$ to another entity's address. However, $\mathcal{F}_{\text{auction}}$ will first check if the σ_{E} is legitimate by interacting with $\mathcal{F}_{\text{sig}}$. Hence, the attempt of $\mathcal{Z}$ will be caught. Thus, the view of $\mathcal{Z}$ is (*minBid*, $addr_{winner}$, $addr_{\mathrm{E}}$, σ_{E}), which is the same as in the real-world.