Succinctly Verifiable Sealed-Bid Auction Smart Contract

Hisham S. Galal(✉) and Amr M. Youssef

Concordia Institute for Information Systems Engineering, Concordia University, Montréal, QC, Canada
h_galal@encs.concordia.ca

Abstract. The recently growing tokenization process of digital and physical assets over the Ethereum blockchain requires a convenient trade and exchange mechanism. Sealed-bid auctions are powerful trading tools due to the advantages they offer compared to their open-cry counterparts. However, the inherent transparency and lack of privacy on the Ethereum blockchain conflict with the main objective behind the sealed-bid auctions. In this paper, we tackle this challenge and present a smart contract protocol for a succinctly verifiable sealed-bid auction on the Ethereum blockchain. In particular, we utilize various cryptographic primitives including zero-knowledge Succinct Non-interactive Argument of Knowledge (zk-SNARK), Multi-Party Computation (MPC), Public-Key Encryption (PKE) scheme, and commitment scheme for our approach. First, the proving and verification keys for zk-SNARK are generated via an MPC protocol between the auctioneer and bidders. Then, when the auction process starts, the bidders submit commitments of their bids to the smart contract. Subsequently, each bidder individually reveals her commitment to the auctioneer using the PKE scheme. Then, according to the auction rules, the auctioneer claims a winner and generates a proof off-chain based on the proving key, commitments which serve as public inputs, and their underlying openings which are considered the auctioneer's witness. Finally, the auctioneer submits the proof to the smart contract which in turn verifies its validity based on the public inputs, and the verification key. The proposed protocol scales efficiently as it has a constant-size proof and verification cost regardless of the number of bidders. Furthermore, we provide an analysis of the smart contract design, in addition to the estimated gas costs associated with the different transactions.

Keywords: Ethereum · Smart contract · Sealed-bid auction zk-SNARK

1 Introduction

The unprecedented growing deployment of assets on Ethereum has created a remarkable market for assets exchange [1] which imposes a high demand for various trading tools such as verifiable and secure auctions. Auctions are platforms for vendors to advertise their assets where interested buyers deposit competitive bids based on their own monetary valuation. Commonly, the auction winner is the bidder who submitted the highest price, however, there are a variety of other rules to determine the winner. Additionally, auctions have also been known to promote many economic advantages for the efficient trade of goods and services. According to [18], there exist two types of sealed-bid auctions: (i) First-price sealed-bid auctions (FPSBA) where the bidders submit bids in sealed envelops to the auctioneer. Subsequently, the auctioneer solely opens them to determine the winner who submitted the highest bid, and (ii) Vickrey auctions, which are similar to FPSBA with the exception that the winner pays the second highest bid instead.

Arguably, the main objective behind concealing the losing bids in sealed-bid auctions is to prevent the use of bidders' valuations against them in future auctions. Therefore, bidders are motivated to cast their bids without worrying about the misuse of their valuations. Nonetheless, when auctioneers collude with malicious bidders, the aforementioned advantage is easily broken. Consequently, the auctioneer has to be trusted to preserve bids' privacy and to correctly claim the auction winner. Therefore, various constructions of sealed-bid auctions utilize cryptographic protocols to ensure the proper and secure implementation without harming the privacy of bids.

Ethereum is the second most popular blockchain based on its market capitalization that exceeds $53 billion USD as of May 2018 [4]. Ethereum allows running decentralized applications in a global virtual machine environment known as Ethereum Virtual Machine (EVM) [28] without depending on any third-party. From a practical viewpoint, the EVM is a large decentralized computer with millions of objects (known as smart contracts) that can maintain an internal database, execute code, and interact with each other. As a result, the EVM substantially simplifies the creation of blockchain applications on its platform rather than building new application-specific blockchain from scratch.

The code executed in the EVM is commonly known as a smart contract which lies dormant and passive until its execution is triggered by transactions. It inherits strong integrity assurance from the blockchain, even its creator cannot modify it once it has been deployed. In Ethereum, computation is expensive as transactions are executed and verified by the full-nodes on Ethereum network. Therefore, Ethereum defines a gas metric to measure the computation efforts and storage cost associated with transactions. In other words, each transaction has a fee (i.e., consumed gas) that is paid by the transaction's sender in Ether (Ethereum currency). With the help of the consensus protocol, the smart contract is also guaranteed to execute precisely as its code dictates. Although many other blockchains such as Bitcoin [24] offer the capability to run smart contracts, they are often very limited to a specific set of instructions. Conversely, the instructions on EVM theoretically allow running any Turing-complete program. However, there is a block gas limit that defines the maximum amount of gas that can be consumed by all transactions combined in a single block. The current block gas limit is around is 8-million gas as of May 2018 [2]. Therefore, smart contracts cannot include very expensive computations that exceed the block gas limit.

In addition, despite the flexible programming capability in Ethereum smart contracts, they still lack transactional privacy. In fact, the details of every transaction executed in the smart contract are visible to the entire network. Moreover, these details are eventually stored in the Ethereum blockchain which also gives the ability to review past transactions as well. Consequently, the lack of transactional privacy is a major challenge towards the deployment of sensitive financial applications. Usually, individuals and organizations prefer to preserve the privacy of their transactions. For example, an organization may not want to post how much it spent on the purchase of some arbitrary assets.

Our contribution, we present a protocol for a sealed-bid auction smart contract that utilizes a set of cryptographic primitives to provide the following properties:

1. **Bids' Privacy.** The submitted bids are not visible to competitors during the bidding phase of the auction in the presence of malicious adversaries.
2. **Posterior privacy.** The losing bids are not revealed to the public assuming a semi-honest auctioneer.
3. **Bids' Binding.** Bidders cannot deny or change their bids once they are committed.
4. **Public Verifiability.** Any individual can verify the correctness of the auction winner proof.
5. **Fairness.** Rational parties are obligated to follow the proposed protocol to avoid being financially penalized.
6. **Non-Interactivity.** The smart contract, on behalf of the bidders, verifies the auction winner proof submitted by the auctioneer.
7. **Scalability.** The verification cost of the auction winner is nearly constant regardless of the number of bidders.

We have also created an open-source prototype for a Vickrey auction smart contract and made it available on Github [1] for researchers and community to review it. The rest of this paper is organized as follows. Section 2 provides a review of privacy-preserving protocols and sealed-bid auctions on the blockchain. In Sect. 3, we present the cryptographic primitives and tools utilized in designing the proposed Vickrey auction smart contract. Then, In Sect. 4, we provide the design of the auction contract together with an analysis of the estimated gas cost of relevant transactions. Finally, we present our conclusions and suggestions for future work in Sect. 5.

2 Related Work

Our proposal depends on utilizing zk-SNARK and distributed ledger (blockchain) technology to build an efficient (i.e., succinct proof with a relatively small verification cost) sealed-bid auction on Ethereum. Therefore, we

[1] https://github.com/hsg88/vickreyauction.

provide a review of state-of-the-art research papers that utilize zk-SNARK in building different cryptographic protocols on the blockchain, besides to papers that provide solutions to building sealed-bid auctions on top of blockchains.

A variety of privacy-preserving protocols are built on top of blockchain technology [5,10,11,15–17,19,20,22]. They combine cryptocurrency with cryptographic primitives such as MPC protocols, commitment schemes, and ZK proofs to achieve fairness in different adversary models. In a nutshell, initially, the protocol participants locks an arbitrary amount of cryptocurrency in an escrow smart contract. Subsequently, they proceed to engage in the various steps of the protocol. Finally, once the protocol reaches its final state, the escrow smart contract refunds the deposits back to the honest participants. Consequently, financially rational participants are obligated to adhere to the protocol rules in order to avoid the financial penalty.

One prominent example of the privacy-preserving protocol that has been deployed on Bitcoin is Zero-Knowledge Contingent Payment [22]. It allows a buyer and a seller to fairly trade an arbitrary digital good in exchange for bitcoins payment. Fairness is achieved without the need for a trusted party. In essence, by the end of the protocol, either the exchange completes with every participant receiving what they are expecting, or none of the participants gains an advantage over another one. Despite the limited flexibility of Bitcoin scripting language, the authors managed to provide a solution by depending on *hash-lock* transactions that allow someone to pay an arbitrary amount of bitcoins to anyone who can provide a preimage x such that $y =$SHA-256(x), for a publicly known value y. We describe a simple version of ZKCP for the sake of illustration purposes. Suppose that a seller Bob wants to trade a digital item p in exchange for v bitcoins. First, Bob encrypts the item p using a symmetric encryption algorithm to obtain $c = Enc_x(p)$ using a key x. Then, Bob computes the hash value of the key $y =$SHA-256(x). Subsequently, he sends (c, y) along with a ZK proof that claims $c = \text{ENC}_k(p)$ and $y =$SHA-256(x). After that, if Alice is interested in that item, she creates a *hash-lock* transaction to pay v bitcoins to anyone who reveals the preimage x such that $y = SHA256(x)$. Finally, Bob receives the payment v bitcoins by revealing x which also means that Alice can decrypt c to get the digital good p.

Campanelli et al. [15] took a step further to propose Zero-Knowledge Contingent Service Payments (ZKCSP) on top of Bitcoin. The main goal is to permit a fair exchange of services and payments over the Bitcoin blockchain. The authors argue that previous constructions of ZKCP [22] are not suitable for the exchange of digital service and payments. They utilized zk-SNARK proof systems [7] to build practical proofs for complex arguments. As an example, they built a prototype for Proofs of Retrievability (PoR) where a client Bob has stored some data on a cloud server and he wants to verify whether the server still keeps and stores his data correctly. In this case, the server offers a digital service rather than a digital good where the server's owner Alice wants to be certain that there is a payment at the end of successful verification of PoR. Moreover, Bob does not want to pay in advance. Therefore, ZKCSP tackles this situation. Also, ZKCSP can be viewed to be more general than ZKCP as it allows for the trade of goods as well as services.

On the area of smart contract frameworks, Kosba et al. [17], presented Hawk, a framework for writing smart contracts that preserves the privacy of financial transactions on the blockchain. The main advantage is to allow programmers without knowledge of cryptographic protocols to build a secure and privacy-preserving smart contract. To this end, the framework includes a compiler that utilizes various cryptographic primitives in generating the smart contracts. A Hawk program source code is composed of public and private parts. The public part is responsible for the logic that does not deal with the sensitive data or the money flow. On the other hand, the private part is responsible for hiding the information about data and input currency units. The compiler translates the Hawk program into three pieces that define the cryptographic protocol between users, manager, and the blockchain nodes. Up to our knowledge, the framework has not been released yet and we could not find a deployed smart contract on Ethereum blockchain built by Hawk.

On the subject of sealed-bid auctions, Blass and Kerschbaum [11] proposed *Strain*, a protocol to build sealed-bid auctions, on top of blockchain technology, that preserve bids privacy against malicious participants. Strain uses a two-party comparison protocol to compare bids between pairs of bidders. Then, the comparison's outcome is submitted to the blockchain which serves as a secure bulletin board. Additionally, since bidders initially submit commitments to their bids, Strain utilizes ZK proof to verify that the submitted comparison's result corresponds to the committed bids. Furthermore, Strain uses reversible commitment scheme such that a group of bidders can jointly open the bid commitment. The objective of this scheme is to achieve fairness against malicious participants who prematurely abort or deviate from the protocol. As the authors reported in their work, Strain has an obvious flaw that reveals the order of bids, similar to Ordered Preserving Encryption (OPE). Furthermore, running protocols involving MPC on blockchain is not efficient due to extensive computations and the number of rounds involved. Meanwhile, our protocol does not suffer from Strains flaws, and it utilizes zk-SNARK to generate a proof that can be efficiently verified with a feasible cost on Ethereum.

Furthermore, Galal and Youssef [16] presented a protocol for running sealed-bid auctions on Ethereum. The protocol ensures the public verifiability, privacy of bids, and fairness. Initially, bidders submit Pedersen commitments of their bids to a smart contract. Subsequently, they reveal their commitments individually to the auctioneer using RSA encryption. Finally, the auctioneer determines the winning bid and claims the winner of the auction. There are two major issues in this protocol. First, for each losing bid, the auctioneer has to engage into a set of interactive commit-challenge-verify protocol to prove that the winning bid is greater than the losing bid. In other words, the number of interactions is proportional to the number of bidders. Second, current techniques for generating a secure random number on blockchains are not proven to be secure due to miners' influence; therefore, the random numbers used in a commit-challenge-verify proof can be compromised. The approach proposed in our paper overcomes these challenges by utilizing zk-SNARK which requires a single proof-verification for the whole auction process. Moreover, it is a non-interactive protocol that does not require random numbers to be generated on the blockchain.

3 Preliminaries

In this section, we briefly introduce the cryptographic primitives that are utilized in the design of our proposed protocol for the sealed-bid auction smart contract.

3.1 Commitment Scheme

Recall that in sealed-bid auctions, the bidders initially submit their bids in sealed envelops for a fixed period of time. Then, the auctioneer opens these envelopes to determine the winner. In other words, we need a tool to hide the bids temporarily, yet with the ability to reveal them later. This task can be easily fulfilled by a cryptographic primitive known as commitments schemes. Typically, a commitment scheme involves two parties: a sender (Alice) and a receiver (Bob). Additionally, it provides two security properties, namely, hiding and binding. Simply, let us denote for an abstract commitment scheme by the public algorithm $c = Com(x, r)$ which takes a value x, a random r, and produces a commitment c. In the reveal phase, Alice simply reveals the values x' and r', then Bob checks whether these two values produces the same original commitment c. The hiding property implies that it is infeasible for Bob to learn the value x given the commitment c. Likewise, the binding property implies that it is infeasible for Alice to reveal with different values x' and r' that produces the same commitment c. such that when Alice commits to an arbitrary value x and sends the commitment c to Bob. Although there are commitment schemes with strong security properties (e.g., information-theoretic hiding) such as Pedersen commitment, we instead use a relaxed one based on collision-resistant hash function due to its flexible integration with zk-SNARK.

To be precise, we choose SHA-256 to be the public algorithm for our commitment scheme. In order for Alice to commit to a bid x, she sends to Bob the commitment $c =$ SHA-256(s) where $s = (x||r)$, r is a k-bit randomness, and $||$ denotes the concatenation operation. Later on, to decommit c, she sends the value s' to him. Subsequently, Bob verifies that $c =$ SHA-256(s'). Then, on successful verification, he strips off the least significant k-bits from s' to recover the bid x.

3.2 zk-SNARK

ZK-SNARK is essentially a non-interactive zero-knowledge (NIZK) proof system. There are several constructions of zk-SNARK especially in the field of verifiable computations. In this paper, we follow the construction proposed by Sasson et al. [9] to verify computations compatible with Von Neumann architecture.

More precisely, to verify the correctness of the auctioneer's computations in determining the auction's winner.

Typically, any NIZK proof system about NP-language L consists of the following three main algorithms:

1. **Key generation**: $crs \leftarrow K(1^\lambda, L)$ which takes a security parameter λ, a description of the language L, and outputs the common reference string crs.
2. **Proof generation**: $\pi \leftarrow P(crs, s, w)$ which takes a crs, a statement s, a witness w such that $(s, w) \in L$, and outputs a proof π.
3. **Proof verification**: $\{0,1\} \leftarrow V(crs, \pi, s)$ which takes a proof π, the previously generated crs, a statement s, and outputs 0 or 1 to denote accept or reject.

In general, any NIZK proof is simply a bulk of data that can be verified at any time without prior interactions between the prover and the verifier. A key requirement though is the proper generation of common reference string (CRS). If there is any trapdoor in the generation of CRS, then the prover is able to generate a fraudulent proof. Likewise, a malicious verifier can exploit the trapdoor to extract information about the witness. Therefore, the generation of CRS is of utmost importance to the security of NIZK proof. The zk-SNARK construction [9] provides the following security properties:

1. **Perfect Completeness.** An honest prover with a valid witness can always convince an honest verifier. More formally, given $(s, w) \in L, crs \leftarrow K(1^\lambda, L), \pi \leftarrow P(crs, s, w)$, then $V(crs, \pi) = 1$.
2. **Computational Soundness.** A polynomial-time adversary can convince a verifier that an invalid statement is true with a negligible probability. More formally, given $crs \leftarrow K(1^\lambda, L), \pi \leftarrow A(crs, s), s \notin L$, then $Pr[V(crs, \pi, s) = 1] \approx 0$.
3. **Computational Zero-Knowledge.** It is computationally infeasible for any polynomial-time adversary to reveal any information about the witness from the proof. More formally, there exists a simulator $S = (K\prime, P\prime)$ that outputs a transcript that is computationally indistinguishable from the one produced by (K, P) in a proof π without knowing a witness.
4. **Succinctness.** A NIZK is said to be succinct if an honestly generated proof has $Poly(\lambda)$- bits and the verification algorithm $V(crs, \pi, s)$ runs asymptotically in $O(|s| \cdot Poly(\lambda))$.

Recall that the key generation algorithm in zk-SNARK takes as an input a representation of the language L. Therefore, we want to find a suitable representation for the auction winner problem. Sasson et al. [9] proposed a general-purpose circuit generator that takes a C-code and translates it into an arithmetic circuit. Simply, arithmetic circuits are acyclic graphs with wires and mathematical operation gates as edges and node, respectively [23]. More precisely, an arithmetic circuit is a function $C : \mathbb{F}^m \times \mathbb{F}^n \rightarrow \mathbb{F}^l$ which essentially takes $(m+n)$-inputs and generates l-outputs. The arithmetic circuit C is said to have a valid assignment tuple $(a_1, ..., a_N)$ where $N = m + n + l$ when $C(a_1, ..., a_{x+y}) = (a_{x+y+1}, ..., a_N)$.

4 Auction Contract Design

In this section, we present the protocol for running a Vickrey auction as a smart contract on top of Ethereum. Our protocol is composed of six phases, where the first two phases are responsible for initializing the zk-SNARK proof system, while, the remaining four phases deal with the auctioning process itself.

4.1 Arithmetic Circuit Generation

Recall that before we can use the first algorithm in zk-SNARK, namely key generation, we need an arithmetic circuit that represents the function we want to provide a proof about its correct execution. Practically, creating an arithmetic circuit for complex arguments is a tedious and error-prone task, especially when the arguments involve operations that are intrinsically depending on logical operators such as the comparison operation and SHA-256 transformation. For this reason, we utilize the general-purpose circuit generator [3,9] to translate a program code into an arithmetic circuit.

Arguably, using a general-purpose arithmetic circuit generator often yields an inefficient circuit with a large number of gates. However, the computation problem of Vickrey auction, as shown in Algorithm 1, is not complex to the degree we worry about the performance of the generated circuit. Moreover, it is reported in [9] that the size of the generated arithmetic circuits scales additively rather than multiplicatively with respect to the size of the translated code.

Algorithm 1. Find highest and second-highest bids and verify commitments

```
 1: function AUCTION($C, U, V$)
 2:     $highest \leftarrow 0, secondHighest \leftarrow 0, status \leftarrow 0, i \leftarrow 0$
 3:     $success \leftarrow true$
 4:     while $i < N$ do                          ▷ N is the constant number of bidders
 5:         if $C[i] \neq$ SHA-256($U[i], V[i]$) then        ▷ check if commitment is valid
 6:             $success \leftarrow false$
 7:             return [success, highest, secondHighest]
 8:         end if
 9:         if $highest < bid$ then
10:             $secondHighest \leftarrow highest$
11:             $highest \leftarrow bid$
12:         end if
13:         $i \leftarrow i + 1$
14:     end while
15:     $success \leftarrow true$
16:     return [success, highest, secondHighest]
17: end function
```

While the auctioneer might be tempted to omit commitments to let a colluding bidder win the auction, doing so will result in a failed verification by the smart contract. In other words, the algorithm does not check whether the auctioneer supplies all commitments as part of the public inputs. On the other hand, the verification which is carried out by the smart contract does supply all commitments. As a consequence, there will be a difference between the public parameters used by the auctioneer to generate the proof, and the public parameters used by the smart contract to verify the proof. Therefore, the verification will fail, and the auctioneer will be penalized if he cannot supply a valid proof that uses the same public parameters as the smart contract.

4.2 Generation of CRS

The outputs of the CRS generation algorithm are the proving and verification keys which are used by the prover and verifier, respectively. It is a mandatory requirement in any NIZK proof system including zk-SNARK to ensure the proper generation of CRS in order to preserve the zero-knowledge and soundness properties. Commonly, the CRS is usually generated by a trusted party. However, this is against the whole premise of the blockchain as a decentralized platform that does not require a trusted party. Moreover, it is sufficient to generate the CRS only one-time as long as the problem statement does not change. In other words, we can initially generate the CRS for the Vickrey auction. Then, we can utilize the resultant CRS in multiple Vickrey auctions.

To avoid the need for a trusted party, various MPC protocols have been proposed to generate the CRS. Bowe et al. [13] presented an MPC protocol to generate CRS for the Zcash cryptocurrency. For the sake of simplicity, let us consider that the CRS is composed of a single element $s \cdot g$ where $s \in \mathbb{F}_p^*$ and g is the generator for a group $\mathbb{G}$ written in the additive notation. Consider that, a prover Alice and a verifier Bob want to generate the CRS such that none of them has knowledge of its discrete log. The protocol runs as follows:

1. Alice chooses a uniform number $s_1 \in \mathbb{F}_p^*$ and sends the element $s_1 \cdot g$ to Bob.
2. Bob chooses a number $s_2 \in \mathbb{F}_p^*$ and sends the element $s_2 s_1 \cdot g$ to Alice.
3. Finally, Alice and Bob use the element $s_2 s_1 \cdot g$ as the CRS.

The problem with this simple protocol is that Bob can maliciously choose s_2 in a way that affects the final output of s. Therefore, the authors in [8,13] proposed a pre-commitment step where each participant first picks a secret number s_i then sends a commitment to it. Later on, they follow the same steps as above but with providing a ZK proof that they used the same secret number corresponding to their commitments. A major problem with this protocol is that the pre-commitment step requires a pre-selection of participants. Moreover, there is an overhead with generating the commitments and verifying the associated ZK proofs.

We follow the MPC protocol for CRS generation in [14] for a number of reasons. First, it does not require a pre-selection of parties to participate in the MPC protocol. Therefore, instead of trusting a specific group of people with the generation of the CRS, any individual can actively join to be assured that a valid CRS is generated even when the rest of participants are malicious. Second, It is more scalable and efficient than the previous construction as it is only a two-round protocol. The basic idea is to use random beacons to eliminate the step of pre-commitment in [13]. Fortunately, Ethereum is the second most popular blockchain, which implies that a large number (i.e., more than 51%) of the miners in the network are reasonably assumed to act honestly. Therefore, we can leverage the blockchain itself as a source of random beacons [12] without worrying about the influence of malicious miners. Hence, the MPC protocol for CRS generation proceeds as follows:

1. Alice chooses a uniform number $s_1 \in \mathbb{F}_p^*$ and sends the element $s_1 \cdot g$ to Bob.
2. Similarly, Bob chooses a uniform number $s_2 \in \mathbb{F}_p^*$ and sends the element $s_2 s_1 \cdot g$ to the smart contract.
3. A beacon s_3 is read from the blockchain, and the element $s_3 s_2 s_1 \cdot g$ is used as CRS.

Assuming that no malicious miner can affect the beacon s_3 in a way that makes finding the discrete log of $s_3 s_2 s_1 . g$ an easy problem, then we can safely say that the generated CRS has no trapdoor that compromises the security of a zk-SNARK proof. We also note that it is an optional task for the bidders to participate in the CRS generation. In other words, the integration of the random beacon in the MPC protocol is sufficient to ensure that the auctioneer cannot control the trapdoor of the generated CRS. Obviously, the achieved security is much stronger when at least one honest bidder participates in the MPC protocol.

1. T_1, T_2, T_3 define the periods using block numbers for the following phases: commitments of bids, opening the commitments, and proof generation and verification, respectively. Note that, T refers to the current block number.
2. N is the maximum number of bidders.
3. F defines the financial deposit of ethers to guarantee fairness against financial rational malicious participants.
4. A_{pk} is the auctioneer's public key of an asymmetric encryption scheme.
5. $Vkey$ is the verification key part of the generated CRS.

```
Create:    $\{T_1, T_2, T_3, N, F, A_{pk}, Vkey\}$ : from auctioneer $A$
           Set $state := INIT, bidders := \{\}$
           Set $highestBid := 0, secondHighestBid := 0$
           Store $N, F, A_{pk}, Vkey$
           Store $T_1, T_2, T_3$
           Assert $T < T_1 < T_2 < T_3 < T_4$
           Assert $ledger[A] >= F$
           Set $ledger[A] := ledger[A] - F$
           Set $deposit := deposit + F$
```

Fig. 1. Pseudocode for Vickrey auction smart contract constructor

4.3 Smart Contract Deployment

The auctioneer deploys the smart contract that runs the Vickrey auction on top of Ethereum. The deployment is basically a transaction that carries the compiled EVM bytecode as a payload and triggers the execution of the constructor. The constructor is responsible for initializing the state of variables in the smart contract. In Fig. 1, we show a pseudocode to illustrate the initialization of the state variables based on the parameters received from the auctioneer.

4.4 Submission of Bids

In this phase, the bidders create transactions to pay F-ethers and submit commitments of their bids as described in Sect. 3 to the function `Bid` which stores the commitments on the smart contract as shown in Fig. 2.

```
Bid:    $\{commit_B\}$ from a bidder $B$:
        Assert $T < T_1$
        Assert $ledger[B] > F$
        Set $ledger[B] := ledger[B] - F$
        Set $deposit := deposit + F$
        Set $bidders[B].Commit := commit_B$
```

Fig. 2. Pseudocode for the submission of bid commitment

We also note that at the end of this phase, if the number of the submitted commitments is less than the maximum number of bidders N, then the auctioneer has to submit the remaining number of commitments with an underlying bid value equals to zero. The reason behind this step is due to technical limitation in the general-purpose arithmetic circuit generator [9]. More precisely, the translated code is not allowed to have loops with variable number of iterations, so we have to fix the loop counter to the maximum number of bidders N. Basically, the circuit generator flattens and unrolls the loop iterations, therefore, the number of iterations has to be known in advance.

4.5 Opening the Commitments

In this phase, each bidder B_i encrypts the pair (x_i, r_i) by the public key A_{pk}, where x_i is the bid value, and r_i is a random number. Then, the ciphertext is sent to the function `Open` on the smart contract as shown in Fig. 3.

Arguably, paying for a transaction that stores the ciphertext on the smart contract seems to be an unnecessary task at the first glance. However, we chose to store the ciphertext rather than sending them off-chain to the auctioneer so we can avoid the following attack scenario. Suppose a malicious auctioneer pretends that an arbitrary bidder Bob has not submitted a ciphertext of the correct

```
Open:    ${ciphertext_B}$ from a bidder $B$:
         Assert $T_1 < T < T_2$
         Assert $B \in bidders$
         Set $bidders[B].Ciphertext := ciphertext_B$
```

Fig. 3. Pseudocode for the Open function

opening values of his associated commitment. In this case, Bob has no chance of denying this false claim. However, if Bob's ciphertext is stored on the smart contract, then the transaction that carried Bob's ciphertext of his commitment's opening values sufficiently defeats this false claim. Furthermore, as it is shown in the pseudocode Fig. 3, the ciphertext is stored in a mapping $bidders[Bob]$ as part of a structure that also contains the sealed-bid (commitment) with senders address as a key. When the auctioneer tries to claim that Bob's ciphertext does not contain the valid openings of his commitment, then Bob is alerted to submit the opening values as plaintext to the smart contract. Subsequently, based on these values, the smart contract recomputes the commitment and the ciphertext, then it compares them against the commitment and ciphertext which are stored in the mapping $bidders[Bob]$. In the case, they are found to be equal, then the protocol terminates by penalizing the auctioneer and refunding the initial deposit to all bidders. Otherwise, Bob is penalized and his commitments and ciphertext are discarded from further processing steps.

4.6 Proof Generation and Verification

In this phase, the auctioneer reads the ciphertext stored on the smart contract and decrypts them off-chain. As a result, the auctioneer has a knowledge of the necessary witness (i.e., openings of commitments), highest-bid, and second-highest bid to generate a proof. Hence, the auctioneer creates a transaction to pass the highest and second-highest bids as part of the public inputs, and the generated proof as shown in Fig. 4.

```
Prove:   $\{bid_1, bid_2, proof\}$ from auctioneer $A$:
         Assert $T_2 < T < T_3$
         Assert $state = init$
         Set $highestBid := bid_1$
         Set $secondHighestBid := bid_2$
         Set $params := \{bid_1, bid_2, bidders.commits, true\}$
         Assert $Verify(Vkey, proof, params)$
         Set $state := valid$
```

Fig. 4. Pseudocode for the Reveal function

The function `Verify` checks the validity of the submitted proof based on the verification key $Vkey$ and the public inputs $params$. It utilizes the pre-compiled contracts EIP-196 [27] and EIP-197 [26] to perform the necessary elliptic-curve pairing operations required for zk-SNARK proof verification [9]. Finally, it returns a boolean output to indicate whether to accept the proof or reject it.

Once the verification of zk-SNARK proof has passed successfully, the auctioneer creates a transaction to finalize the auctioning process. Basically, the auctioneer reveals to the smart contract the openings associated with the auction winner commitment as shown in Fig. 5. Then, the smart contract checks whether the winner's bid value equals to the highest bid, and it also checks that the commitment of the opening values is equivalent to one of the previously submitted commitments by the bidders. Finally, the smart contract determines the address associated with the auction winner, and it begins to refund the deposit F to the losing bidders and the auctioneer. However, the winner deposit is not refunded until she fulfills the payment of the second-highest bid.

```
Finalize:   ${bid_w, random_w}$ from auctioneer $A$:
            Assert $T_2 < T < T_3$
            Assert $state = valid$
            Assert $highestBid = bid_w$
            Set $commit_w := SHA256(bid_w, random_w)$
            Assert $commit_w \in bidders.commits$
            Set $winner := Find(bidders, commit_w)$
            Set $ledger[A] := ledger[A] + F$
            Set $deposit := deposit - F$
            For $b \in bidders - \{winner\}$
                Set $ledger[b] := ledger[b] + F$
                Set $deposit := deposit - F$
            Set $state := finalized$
```

Fig. 5. Pseudocode for the Finalize function

As shown in Fig. 6, there is one more function in the smart contract called *Dispute*. The objective of this function is to counter a possible malicious behavior by the auctioneer. Let us assume the auctioneer refuses to carry out the proof generation and verification task. This will certainly result in a permanent lock of the deposits paid by the bidders. Therefore, any bidder can call this function that checks if the *state* variable on the smart contract is still equal to *init* after the block number T_3 has been mined, then it refunds the deposits back to the bidders. Consequently, only the auctioneer ends up being financially penalized since there is no way to refund his deposit.

```
Dispute:    Assert $T > T_3$
            Assert $state = init$
            For $b \in bidders$
                Set $ledger[b] := ledger[b] + F$
                Set $deposit := deposit - F$
```

Fig. 6. Pseudocode for the Dispute function

4.7 Gas Cost Analysis

We have implemented a prototype for the proposed protocol and smart contract, We have also made it open-source and published it on Github repository [2]. To experiment with our prototype, we have created a local Ethereum blockchain using the *Geth* client version 1.8.10. The genesis file that is responsible for the initialization of the local blockchain contains the following attribute in order to support the pre-compiled contracts EIP-196 and EIP-197: $\{``byzantiumBlock'' : 0\}$. In our experiment, we created test-case with the number of bidders $N = 100$, respectively. Table 1 shows the gas cost and the corresponding price in *USD* for the deployment and functions calls on the smart contract. At the time of carrying out our experiment, May 30th, 2018, the ether exchange rate is 1 ether = 583$ and the median gas price is approximately 20 *Gwei* $= 20 \times 10^{-9}$ ether.

Table 1. Gas cost associated with the various functions in the smart contract

Function	Transaction cost	Price
Deployment	1346611	15.70
Bid	115583	1.35
Open	44176	0.52
Prove	3395077	39.58
Finalize	92362	1.07
Dispute	47112	0.55

The smart contract deployment cost is a little bit expensive as it involves the deployment of helper contracts and libraries that are responsible for verifying zk-SNARK proof. However, this cost can be significantly reduced if the helper contracts and libraries are already deployed beforehand. Also, the cost of the *Prove* function is constant due to the fixed cost of the elliptic-curve operations performed during the verification of the zk-SNARK proof. Arguably, this cost might be not convenient for relatively cheap auctioned assets. However, our work scales much better and more efficiently than previous auctions constructions on blockchain [11,16,25].

[2] https://github.com/hsg88/VickreyAuction.

Although we mimic the real-world behavior of sealed-bid auctions where the auctioneer gains knowledge of the sealed-bids, we still guarantee a verifiable computation of the auction winner. One may argue that the auctioneer may abuse the information she gained in future auctions to increase her revenues. We are currently investigating the integration of Trusted Execution Environment (TEE) such as Intel Software Guard eXtensions (SGX) into our protocol. Briefly speaking, the role of the auctioneer is replaced by a code that runs inside an enclave that protects it from modification and prevents disclosure of its state by the operating system or any other application. Hence, it provides the confidentiality of data which the blockchain lacks. Additionally, the enclave can issue proof which is also known as *attestation* of computation correctness.

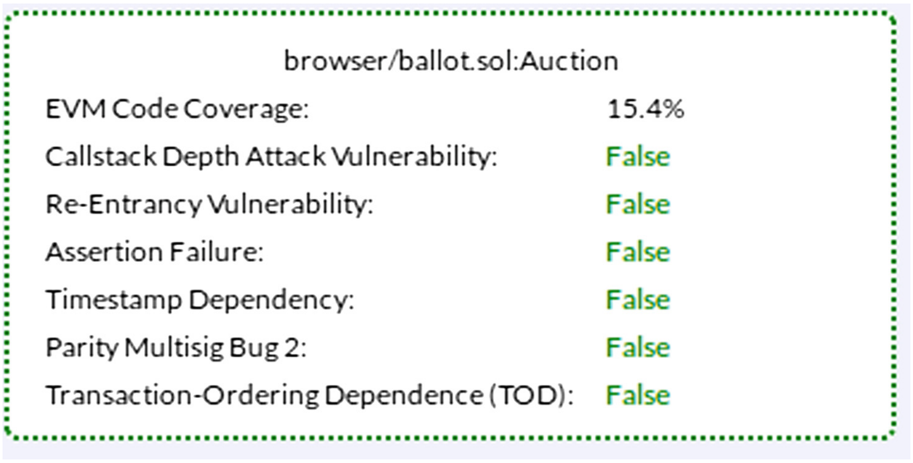
4.8 Checking for Potential Security Bugs

The Vickrey auction smart contracts are implemented in Solidity which is the de-facto programming language for writing smart contracts on Ethereum. Due to the sensitive nature of the smart contracts especially when they involve operations to send or receive money, we have to properly assess their security before deployment. Recall that, once a smart contract is deployed, there is no way to modify it to patch a bug. Therefore, we utilize security analysis tools to check for potential bugs such as the famous *Re-entry* [6] bug found in DAO smart contracts that eventually caused hard-fork on Ethereum to mitigate the attack. Precisely, we use Oyente[3] [21] to ensure that the Vickrey auction smart contract does not contain any vulnerabilities to known critical security issues. The results from running Oyente on the Vickrey auction protocol are shown in Fig. 7. Also, we have utilized unit-testing to detect errors in the Solidity code, and the applied tests are included in the repository for further inspection by the community.

5 Conclusions

In this paper, we showed how zk-SNARK can be utilized to build Vickrey auction on top of Ethereum blockchain. The proposed protocol achieves many desirable properties such as bids privacy where the information about the bids is kept hidden from competitors. Additionally, the auction smart contract, on behalf of the bidders verifies the correctness of the auction winner claimed by the auctioneer. Moreover, the proposed protocol is scalable as the cost of the verification transaction is constant regardless of the number of bidders. Furthermore, the proposed protocol is in the favor of bidders with relatively low-processing power devices, since there are only two simple interactions with the smart contract involving the submission of commitments of bids, and revealing the underlying openings. For future work, we will investigate other approaches applicable to the blockchain where we can also protect the privacy of bids from all parties including the auctioneer by integrating TTE into our protocol. More precisely, we are

[3] We used the online Oyente tool available on https://oyente.melon.fund.



```
browser/ballot.sol:Auction
EVM Code Coverage:                                15.4%
Callstack Depth Attack Vulnerability:             False
Re-Entrancy Vulnerability:                        False
Assertion Failure:                                False
Timestamp Dependency:                             False
Parity Multisig Bug 2:                            False
Transaction-Ordering Dependence (TOD):            False
```

Fig. 7. Results from running Oyente online tool on the smart contract

exploring the feasibility of alternative approaches to building sealed-bid auction on top of the Enigma protocol [29]. Additionally, we will continue improving our prototype and apply formal verification techniques on our code to detect potential flaws and vulnerabilities.

References

1. Digital assets in ethereum blockchain. https://tokenmarket.net/blockchain/ethereum/assets/
2. Ethereum gaslimit history (2018). https://etherscan.io/chart/gaslimit
3. Python-based system for zk-snark based verifiable computations and smart contracts (2018). https://github.com/Charterhouse/pysnark
4. Top 100 cryptocurrencies by market capitalization (2018). https://coinmarketcap.com
5. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on bitcoin. In: IEEE Symposium on Security and Privacy (SP), pp. 443–458. IEEE (2014)
6. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (SoK). In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 164–186. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
7. Ben-Sasson, E., Chiesa, A., Genkin, D., Kfir, S., Tromer, E., Virza, M.: libsnark (2014). https://github.com/scipr-lab/libsnark
8. Ben-Sasson, E., Chiesa, A., Green, M., Tromer, E., Virza, M.: Secure sampling of public parameters for succinct zero knowledge proofs. In: IEEE Symposium on Security and Privacy (SP), pp. 287–304. IEEE (2015)
9. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Succinct non-interactive zero knowledge for a von neumann architecture. In: USENIX Security Symposium, pp. 781–796 (2014)

10. Bentov, I., Kumaresan, R.: How to use bitcoin to design fair protocols. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 421–439. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44381-1_24
11. Blass, E.-O., Kerschbaum, F.: Strain: A secure auction for blockchains. Cryptology ePrint Archive, Report 2017/1044 (2017). https://eprint.iacr.org/2017/1044
12. Bonneau, J., Clark, J., Goldfeder, S.: On bitcoin as a public randomness source. IACR Cryptology ePrint Archive 2015, 1015 (2015)
13. Bowe, S., Gabizon, A., Green, M.D.: A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. Technical report, TR 2017/602, IACR (2017)
14. Bowe, S., Gabizon, A., Miers, I.: Scalable multi-party computation for zk-snark parameters in the random beacon model (2017)
15. Campanelli, M., Gennaro, R., Goldfeder, S., Nizzardo, L.: Zero-knowledge contingent payments revisited: attacks and payments for services. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 229–243. ACM (2017)
16. Galal, H., Youssef, A.: Verifiable sealed-bid auction on the ethereum blockchain. In: International Conference on Financial Cryptography and Data Security, Trusted Smart Contracts Workshop. Springer (2018)
17. Kosba, A., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: the blockchain model of cryptography and privacy-preserving smart contracts. In: IEEE Symposium on Security and Privacy (SP), pp. 839–858. IEEE (2016)
18. Krishna, V.: Auction Theory. Academic Press (2009)
19. Kumaresan, R., Bentov, I.: Amortizing secure computation with penalties. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 418–429. ACM (2016)
20. Kumaresan, R., Vaikuntanathan, V., Vasudevan, P.N.: Improvements to secure computation with penalties. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 406–417. ACM (2016)
21. Luu, L., Chu, D.-H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 254–269. ACM (2016)
22. Maxwell, G.: Zero knowledge contingent payment (2011). https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment. Accessed 05 Jan 2016
23. Mayer, H.: zk-snark explained: Basic principles (2016). https://blog.coinfabrik.com/wp-content/uploads/2017/03/zkSNARK-explained_basic_principles.pdf
24. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
25. Snchez, D.C.: Raziel: private and verifiable smart contracts on blockchains. Cryptology ePrint Archive, Report 2017/878 (2017). https://eprint.iacr.org/2017/878
26. Reitwiessner, C., Buterin, V.: Elliptic curve pairing operations (2017). https://github.com/ethereum/EIPs/blob/master/EIPS/eip-197.md
27. Reitwiessner, C., Buterin, V.: Elliptic curve point addition and scalar multiplication operations (2017). https://github.com/ethereum/EIPs/blob/master/EIPS/eip-196.md
28. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum Proj. Yellow Paper **151**, 1–32 (2014)
29. Zyskind, G., Nathan, O., Pentland, A.: Enigma: decentralized computation platform with guaranteed privacy. arXiv preprint arXiv:1506.03471 (2015)