Verification of an Ethereum Auction Contract through Formal Methods

Valerie Chen
Advisers: Zhong Shao, Vilhelm Sjöberg

May 2018

Abstract

The development of smart contracting technology promises greater efficiency and less overhead for transaction execution than traditional methods involving third parties. However, many smart contracts in circulation feature security vulnerabilities that could expose millions of dollars to the risk of theft, exacerbated by the code transparency inherent to smart contract design. This project explores methods for the certification of Ethereum smart contracts using formal methods in the Coq proof assistant. We consider a generic auction contract and model its bidding system, which allows users to submit bids within the designated auction time frame. We outline several conditions for correctness on the model, such as the highest bidder in the history of the bids is the winner at the conclusion of the auction, and the bids always increase in value with the progression of time. While we produced a comprehensive statement of the terms of correctness for the model, the proofs remain in progress. We will be pursuing further work towards this end, with the goal of extending the proof for contracts of greater complexity.

1 Introduction

With the rising popularity of bitcoin, the underlying technologies supporting the e-currency have been revitalized as well. The transparent and distributed ledger for maintaining transactional data (known as blockchain) as well as the decentralized consensus protocol used for updating such ledgers carry over to the recent development of smart contracts. This concept originated with work from Nick Szabo in 1996 (“Smart Contracts: Building Blocks for Digital Markets”) and is most notably implemented by Ethereum, a platform that provides a virtual machine (the Ethereum Virtual Machine, or EVM) which may execute scripts using a globally distributed network of nodes.

However, while the excitement surrounding blockchain and smart contracts continues to rise (reaching the realms of information technology, finance, and even national governance), significant vulnerabilities in the design of such software have become apparent. Some lead to significant losses, such as the case of “The DAO Attack”[1], in which an Ethereum-supported crowdfunding service lost roughly 50 million USD worth of investments due to a bug in the program which was exploited by hackers.
Thus, as these technologies work their way into finance, transportation, healthcare, and numerous other global industries and sectors, it is imperative that smart contract platforms are designed to combat existing bugs as well as broader categories of software vulnerabilities. This project keys in on a single type of auction contract, a straightforward and common category of smart contracts involving multiple parties. By translating the contract into a program written in the specification language Gallina, we are able to formally verify a number of assertions on the consistency of the contract using Coq, a proof assistant frequently utilized by Yale’s FLINT Group and the Yale-affiliated CertiK platform[2].

2 The Simple Auction Contract

2.1 Ethereum Smart Contracts

At the highest level, smart contracts in Ethereum manage customizable agreements between users. Each contract enumerates parameters, states, functions, and events, which together allow a series of transactions (a state change or a relocation of resources) to take place. These transactions depend on gas, the resource expended by operations on the blockchain.

The contract itself manages state changes as well as function execution, and may send and receive Ether (the Ethereum virtual currency). This capital management feature is foundational for the practical use of smart contracts.

2.2 Source Contract

The “Simple Auction Contract” (SAC) is an Ethereum contract developed by Julien Bouteloup (GitHub: /bneiluj) [3]. The SAC facilitates an absolute auction (i.e. highest bidder wins, no reserve price) between an unlimited number of bidding parties, with a single listed beneficiary. Participants bid in Ether, and may call a withdrawal function when they are outbid, which returns the payment in Ether to their user address. The auction is constrained by time; bidders can only participate during the specified bidding window, at the end of which the winner is determined (supposing there was at least one bid).

2.3 Code Structure

Parameters Fixed values describing a specific instance of an auction.

- beneficiary: address (20 byte value)[5].
- auctionStart: unsigned int (256 bits).
- biddingTime: unsigned int.

States Subject to change at different points in the execution of the contract (time-variant).

- highestBidder: address.
- highestBid: unsigned int.
• **pendingReturns**: mapping from addresses to unsigned ints. Keeps track of failed bids that have yet to be returned to the bidder via `withdraw()`.

• **ended**: bool. Flagged once, at the end of the auction, and prevents repeat calls of the `auctionEnd()` function.

**Functions**

• **SimpleAuction()**: initializer function.

• **bid()**: called by bidders to submit resources. Does not contain a check to see if the sender matches previous `highestBidder`; thus, a user may outbid their own bid.

• **withdraw()**: called by bidders who have been outbid to retrieve proffered ether.

• **auctionEnd()**: ends the instance.

**Events** Fired by the contract within transactional functions to signal a change in state. Fired events append an entry to log associated with the receipt in each transaction.

• **HighestBidIncreased(address bidder, uint amount)**: announces that the previous highest bid has been outbid.

• **AuctionEnded(address winner, uint amount)**: announces that the auction is complete, and no further bids may be submitted. Immediately following, the contract sends the bid resources (which it has been managing) to the beneficiary.

Within each of these functions, the contract will throw an exception if any clauses of the auction contract are violated. (For example, if a user submits a bid that doesn’t exceed the value of the current `highestBid`.) This prevents further execution and reverts any side effects associated with the current function call.

**2.4 Comparison with Similar Contracts**

There are many different Ethereum auction contracts available in the public domain, representing a wide range of auction formats and complexities. A few others we considered for this project include Bryn Bellomy’s “Solidity Auction”[6] and Ethereum’s example auction contract, “One Phase Auction”[7].

While both contracts share a similar organization of static and state variables (as outlined above), both implement many additional metadata fields. Ultimately, these seemed peripheral to the fundamental goals of our proof, so SAC was chosen mostly for simplicity’s sake. Additionally, “Solidity Auction” and “One Phase Auction” both embedded ether transfer calls into the logic of their bid functions (whereas the bid function in SAC only modifies a log of expected transactions, and `withdraw()` is only function that moves funds). “One Phase Auction” also included a safeguard to make sure that the gas expenditure remains below a reasonable threshold. These features are attractive for efficiency and security purposes, respectively, but complicate the nature of the proof for the given auction model. In the future, it may be beneficial to
explore modified versions of SAC that involve ether transfer in its state modification functions, therefore having a higher degree of interconnectedness with other contracts in the ecosystem. Since smart contracts are in practice highly interdependent, a complete proof on interactive contracts would be particularly useful for the progression towards more hack-resistant contracts.

It is notable that SAC only keeps track of the highest bidder/bid pairing at any given time, unlike “One Phase Auction,” which keeps a comprehensive ledger of bids that is updated throughout the bidding process. Ultimately, this resulted in a simpler logical model for the contract state, but required a more rigorous model for the proof that could account for all events in the history of the contract execution. This is described further in section 4.

3 The Coq Theorem Prover

3.1 Summary

Coq is a verification tool based on the Calculus of Inductive Constructions, a powerful logic system that is able to represent both functional programs (in the style of ML, a typed functional language) and proofs in higher-order logic. Basic data structures like sets, trees, and lists may be expressed inductively using Gallina, the specific language for Coq. In developing a Coq proof, we write to a proof script (the code itself) and observe the proof state, which is compiled progressively. The proof state at any given point in the script enumerates all subgoals currently needed to prove the axiom, as well the working hypotheses associated with the axiom.

Proving theorems in Coq requires the use of tactics. Tactics are commands (either built-in or imported) in the Coq environment that can solve or transform goals and hypotheses (often based on proven implications). Fundamental to the assistive power of Coq lies in the strength of its library of tactics, which allows parts of complicated proofs to be immediately parsed and simplified.

3.2 Example Proof Session

We demonstrate a simple example of a proof in CoqIDE, a development tool tailored for Coq.

3.3 Value Discussion and Applicability for Smart Contracts

Coq implements a rigorous type checker, which ensures that a given expression is well formed, i.e. conforms to the set of declarations associated with its type. While verifying a Solidity contract, this allows for efficient and accurate type judgments to support a wide variety of data structures represented in the contract model. For example, suppose we have a set of declarations:

```coq
H : list positive -> positive -> Prop
fn : list positive -> positive
rel : positive -> positive -> positive
user_list : list positive
user : positive
```
In this context, we see that \texttt{rel (fn user list) user} is of type \texttt{positive}, and \texttt{H fn} is of type \texttt{Prop} (a propositional statement). Furthermore, \texttt{H user list} is not well formed. The Coq type checker is able to come to these conclusions as well, verifying whether an expression is well formed in any given context. Note that \texttt{H} is a higher order function, taking other functions as arguments; Coq is able to model and manage complicated webs of higher order functions, which is extremely useful for proving assertions in contracts where numerous parties (users, other contracts) may affect the contract state.

Furthermore, dependent typing in Coq allows for polymorphic types. This is generally useful for declaring types that inherit from standard Coq library types (e.g. \texttt{list}). However, it can also work in tandem with higher-order functionality to represent polymorphic contracts and inheritance relations, e.g.:

```solidity
pragma solidity ^0.4.16;
contract color {...}
contract mood {...}
contract red is color {...}
contract blue is color, mood {...}
```

In modeling these relationships, Coq’s type system can allow for definitions of properties and types in child contracts to be described in relation to that of their parent contracts.

Another important aspect of Coq is that the set of built-in features is very small. While data types like integers, booleans, and lists are declared in the standard library, the system is meant for customization. While this enables us to tailor the data representation in our model to the structures of the source program (in our case, Solidity contracts), it does pose the challenge of needing to define all these types and relationships from scratch. This project imports definitions from the CompCert library, which allows us to easily define and reason about Solidity data types (\texttt{uint}, \texttt{address}, \texttt{byte}, etc).

4 The Proof in Coq

To complete the proof, we first model the contract as a state transition system in Gallina. Then, using the same terms, we define axioms that describe the conditions of correctness. The last step involves proving these axioms via the methodology outlined in previous sections.

4.1 Modeling the Contract

SAC has a number of parameters and state variables. We observe that this list of fields defines the state of the contract at any given moment in time. Thus, we model the contract state using the \texttt{Record} function, a macro allowing the definition of records with fields (as commonly seen in many programming languages).

The relevant data structures in SAC are \texttt{address}, \texttt{uint}, and \texttt{bool}. In Solidity, type \texttt{address} is a 20-byte account identifier (comparable to a hash), and \texttt{uint} is a 256-bit unsigned integer. We abstract both these data types into the simple Coq type \texttt{positive}, which allows us to utilize CompCert’s library.
of axioms on positives (CoqLib.v). This type is more compatible for modeling unsigned integers than naturals (nat in Coq), but the concession is that there is no additive identity on the set of positives. Thus, in order to initialize highestBidder and highestBid to a reasonable value at the beginning of the auction, we wrap these fields in optionals.

We use the inductive type PTree from CompCert to model the mapping from previous bidders to their respective bids in pendingReturns. These fields describe a State; we also define StateList as a list type that describe the history of an auction.

---

Open Scope positive_scope.

Record State :=
{ beneficiary : positive;
auctionStart : positive;
biddingTime : positive;
highestBidder : option positive;
highestBid : option positive;
ended : bool;
pendingReturns : PTree.t positive (* Maps positive -> positive *) }.

Definition StateList := list State.

Exc. 1: Modeling the contract state as a collection of fields, and the progression of the auction as a list of states (i.e. a history).

We define functions in Auction.v that correspond to the contract initializer as well as transactional functions: init, bid, and sigEnd. init takes in auction parameters as arguments and returns a StateList. bid and sigEnd both take in a history and time of the function call, and bid additionally takes in information about the bid. Both return optional histories, as the functions in the Solidity contract may throw errors if the call is invalid, e.g. if the bid does not exceed the previous high bid or if the auction is being ended before the requisite amount of time has elapsed. If either of these functions return None, it signals that the function call has attempted an invalid operation and has no impact on the history of states.

Next, we define a world state in System.v which references types from Contract.v. Since SAC does not keep a comprehensive record of all bids (and memory of a bid that was defeated is lost once the bidder withdraws the ether), we must define our own log (BidLog) of all successful bids in the contract. Ultimately, we will assert invariants on the relationship between this log and the contract state. WorldState also includes a time variable, which is used by the bid() and auctionEnd() functions in the contract, but is not a state in the contract. Thus, we add the time field of each state change as a component external to the contract state.

Finally, we define an inductive step relation on world states. The step function asserts a relation between world states that describes whether or not one WorldState is reachable (via bid and sigEnd) from another. We include a
Definition init
  (beneficiary auctionStart biddingTime : positive) : StateList := ...

Definition bid (history : StateList) (newBidder newBid now: positive)
  : option StateList := ...

Definition sigEnd (history : StateList) (now : positive) : option StateList := ...

Exc. 2: Modeling functions of the contract.

(* In Contract.v *)
Definition Time := positive.
Definition address := positive.
Definition amount := positive.
Definition BidEntry := (address * amount)%type.
Definition BidLog := list (BidEntry).

(* In System.v *)

Exc. 3: Modeling the world state.

condition in each member of the inductive definition that formalizes the natural assumption of temporal progression: i.e. $tm' > tm$.

The \texttt{multistep} function builds on \texttt{step} to inductively describe reachability between a series of world states. The member \texttt{multistep.0} describes the reflexive relation on \texttt{step}, i.e. any world state is reachable from itself. \texttt{multistep.m} asserts that if there is a multi-step relations between world states $s_0$ and $s_1$, as well as a step relation between $s_1$ and $s_2$, then there exists a multi-step relation between $s_0$ and $s_2$.

Additionally, we prove a theorem \texttt{step_imp_multi} which states that any step relation is also a multi-step relation. This is useful in cases where it is easy to prove that some proposition $P$ implies a step relation; we may immediately conclude that $P$ proves a multi-step relation as well.

4.2 Defining Verification Goals

Reasoning intuitively on a simple auction, we have these basic goals: (1) the user who submits the highest bid over the course of the auction wins, and (2) all other bidders get their resources back. The challenge of the project lies in formalizing these intuitions into provable statements.

We write a \texttt{theorem} which asserts that if there exists a multi-step relation between world states $w$ and $w'$, where $w$ is an initial world state (defined in \texttt{init.ws}), then the highest bidder in the most recent entry of the contract
Inductive step : WorldState -> WorldState -> Prop :=
  | bid_step : forall ast bdr amt tm log ast' tm',
    Auction.bid ast bdr amt tm' = Some(ast') ->
    tm' > tm ->
    step (ast, tm, log) (ast', tm', (bdr, amt) :: log)
  | end_step : forall ast tm log ast' tm',
    Auction.sigEnd ast tm' = Some(ast') ->
    tm' > tm ->
    step (ast, tm, log) (ast', tm', log).

Inductive multistep : WorldState -> WorldState -> Prop :=
  | multistep_0 : forall s,
    multistep s s
  | multistep_m : forall s0 s1 s2,
    multistep s0 s1 ->
    step s1 s2 ->
    multistep s0 s2.

Theorem step_imp_multi : forall w w' ast t lg ast' t' lg',
  step w w' ->
  w = (ast, t, lg) ->
  w' = (ast', t', lg') ->
  multistep (ast, t, lg) (ast', t', lg').

Proof.

Exc. 4: Step relations.

history (Auction.getWinner) is equivalent to the highest bidder in the world state's log of bids (Contract.highestBidder). We also define a similar theorem for single steps. Once these theorems are proven, they can be used to reason that when the auction contract has reached its terminal state (i.e. the auction is complete, ended = true), this condition will also hold.

A couple of supporting lemmas for this theorem assert that given a step relation on two states, the highest bid (determined by Auction.getWinner and Contract.highestBidder, respectively) either increases or stays the same. The progressive nature of bids across state transitions is the invariant supporting our final assertion.

This project currently does not define a function corresponding to the process of retrieving funds from bids that were outbid; namely, withdraw(). However, given the pendingReturns field in the contract state and the BidLog in the world state, we hypothesize another type WithdrawalLog would provide the parameters necessary to reason about whether all bidders can successfully withdraw when they have been outbid. WithdrawalLog may be a part of the world state, and has a bidder address and bid amount appended to it every time withdraw() is successful. An outline of an invariant for the second proof goal, then, may be that the sum of the bid amount in pendingReturns and WithdrawalLog corresponding to each bidder must be equivalent to the sum of all bids associated with the given bidder in BidLog.
Theorem winner_is_hi_bidder_mlt_init : forall w w',
multistep w w' ->
forall bn aus bt t ast' t' lg' p, w = (init_ws bn aus bt t) ->
w' = (ast, t', lg') ->
Auction.getWinner ast' = p ->
Contract.highestBidder lg' = p.

Exc. 5: A theorem describing a high-level goal of this proof.

4.3 Proofs of Correctness
At the current stage of the project, a subset of the theorems we have stated have been completed. Some excerpts from the code are listed in Exc. 6.

4.4 Discussion
Key to the process of developing these proofs were the Coq features that allow for proofs to be constructed from top down. Once a theorem is stated, we may use Admitted to temporarily assume it is provable. It is rather clear what the high level goals are, as they most closely resemble how we intuitively think about the desired features of a contract; what is less clear is the exact set of subproofs and lemmas required to support these overarching theorems. Thus, after stating the high-level theorem, we may attempt to transform its hypotheses into workable statements for its proof using (yet unproven) axioms. This process of trial and error (aided by Coq tacticals like try, all, repeat) was fundamental to the production of this project’s completed proofs.
Theorem step_imp_multi : \(\forall w \ w' \ ast \ t \ lg \ ast' \ t' \ lg',\)
\[\text{step } w \ w' \rightarrow w = (ast, t, lg) \rightarrow w' = (ast', t', lg') \rightarrow\]
\multistep (ast, t, lg) (ast', t', lg').
Proof.
intros.
apply multistep_m with (s1 := (ast, t, lg)) (s2 := (ast', t', lg')).
- apply multistep_0.
- rewrite <- H0. rewrite <- H1. assumption. Qed.

Lemma add_list_same_nil : \(\forall (x : \text{Contract.BidEntry}) (lst : \text{Contract.BidLog}), lst = x :: lst \rightarrow \text{False}.\)
Proof.
induction lst. discriminate 1. intros. apply IHlst. congruence. Qed.

Lemma step_length_inc_one : \(\forall w \ w', \text{step } w \ w' \rightarrow \forall a t lg a' t' lg', w = (a, t, lg) \rightarrow w' = (a', t', lg') \rightarrow \)
\(\text{length } a' = \text{length } a + 1.\)
Proof.
intros. destruct a.
- simpl. induction H.
  + unfold Auction.bid in H. destruct ast. discriminate H.
  + unfold Auction.sigEnd in H. destruct ast. discriminate H.
- induction H.
  + unfold Auction.bid in H. destruct ast. discriminate H. Open Scope positive_scope.
  destruct (Auction.auctionStart s0 + Auction.biddingTime s0 <=? tm').
  discriminate H. destruct (Auction.highestBid s0). destruct (amt <=? p).
  discriminate H. destruct ((Auction.pendingReturns s0) ! bdr).
  rewrite <- H7. simpl. omega.
  rewrite <- H7. simpl. omega.
  rewrite <- H7. simpl. omega.
  + unfold Auction.sigEnd in H. destruct ast. discriminate H.
  destruct ((tm' <=? Auction.auctionStart s0 + Auction.biddingTime s0) || Auction.ended s0).

Exc. 6: Completed proofs.
5 Conclusion / Future Work

While the proof in Coq for the Simple Auction Contract is yet incomplete, this project demonstrates that modeling smart contracts via state transition systems and step relations is a sound strategy for producing rigorous proofs of program correctness. Looking forward, we consider a number of strategies for the completion of this proof as well possible avenues for expansion. Firstly, we may consider separating the parameters of the contract from the states in the contract state type; this has the benefit of delineating contract constants from variables, which may simplify proofs involving assertions on a relation between the two groups. Furthermore, we may explore further the representation of timed intervals in Coq, a key element of auctions and a recurrent theme in many smart contracts. Lastly, in order to accurately represent the Solidity smart contract language, we may replace the simple data structures with representations of integral (limited) data types via CompCert libraries.

SAC is but one of many existing auction contracts, all of which present unique challenges for verification. We anticipate some contracts, e.g. simple Dutch auctions, are structurally similar to and thus might pose categorically similar challenges as SAC. However, some contracts may require private bids (e.g. Vickrey and other sealed-bid auctions) or apply a completely different bidder-to-beneficiary dynamic (e.g. Walrasian auctions). The variance in auction types and implementations provide rich ground for research on the applications of formal logic in smart contract and blockchain security.

6 Acknowledgements

Many thanks to Professor Zhong Shao and Dr. Vilhelm Sjöberg for their wisdom and guidance on this project, as well as introducing me to the possibilities of computer science research. Further gratitude towards Professor Ruzica Piskac for teaching me many of the foundational concepts in software verification that I utilized throughout the past semester. Lastly, I am grateful to my peers in the computer science major at Yale, with whom I have shared countless hours in the third floor of AKW, and without whom I would not have made it to this point today.

References


