Abstract

Ethereum is a public, open-source platform that allows users to create contracts that can store data on the Ethereum blockchain and send and receive messages to and from other contracts. Such contracts can be combined in complex ways to create powerful decentralized applications. Because of the nature of the Ethereum blockchain, bugs in contract code are difficult to rectify and may lead to large monetary losses if exploited. For this project, we propose to extend the compiler for DeepSEA, a certified programming language used in the development of mCertiKOS, to generate Coq specifications from DeepSEA specifications of Ethereum contracts written in Solidity. Last semester, we decided to first hand-write specifications for a payment channel and client code that models a system in which two players can make payments to each other with off-chain messages and then finalize the transaction using the on-chain payment channel smart contract. We propose to complete the proof of correctness of the payment channel system and then rewrite the Coq specifications in DeepSEA. Additional work will be needed to rework the proofs to work with the specifications generated by the DeepSEA compiler.

1 Introduction to Ethereum

Ethereum is a public, open-source platform that allows users to build and run decentralized applications that run on blockchain technology [3]. A blockchain is a distributed computing architecture where each node in the network executes and records transactions as blocks in the blockchain. Because only one block is added at a time and each block contains a cryptographic proof of its validity, each node in the network can agree upon the current state of the blockchain.
The blockchain forms the backbone of Bitcoin, a cryptocurrency released in 2009, which uses it to record transactions between users [4]. Ethereum, on the other hand, keeps track of the state of every account on the blockchain.

The account is the basic building block of Ethereum, of which there are two types: externally owned accounts (EOAs) and contract accounts (contracts) [1][2]. EOAs are human-controlled because they are controlled by private keys held by the accounts owner. Contracts (often also referred to as smart contracts), on the other hand, are a collection of code and data that reside at a specific address on the Ethereum blockchain in a binary format called Ethereum Virtual Machine (EVM) bytecode. Contracts, which are typically written in a higher-level language like Solidity and compiled into EVM bytecode to be deployed to the blockchain, can send and receive messages to and from each other and to accounts. In this manner, they can be put together to build decentralized applications backed by the Ethereum blockchain. Examples of such applications range from voting applications, to peer-to-peer trading markets, to video games.

2 Motivation

Contracts are such a young and developing technology that their interactions can lead to unintentional consequences and losses of large amounts of money. One such event was the Decentralized Autonomous Organization (DAO) hack, which occurred in June 2016 [5]. The DAO was a set of Ethereum contracts that enabled people to buy tokens representing voting rights during a funding period, called an initial coin offering (ICO) [6]. Once the ICO was over, investors could vote on proposals on how the DAO should spend the money. By May 2016, DAO had raised $160 million in Ether, the token used by Ethereum clients to pay for computations on the EVM, from thousands of anonymous investors. But just one month later, a hacker was able to exploit a bug in the DAO contract code to siphon away 3.6 million Ether ($50 million)—more than a third of the 11.5 million Ether in the DAO at the time.

3 Proposal

We propose to use DeepSEA, a certified programming language developed to allow developers to write specifications for program behavior [7], to write formally verified smart contracts. This way, we can guarantee that the behavior of such contracts will be well-defined and bug-free. DeepSEA was originally used to aid in the development of mCertiKOS, a
fully verified hypervisor that can boot a version of Linux as a guest. Edsger, the DeepSEA compiler frontend, and a Coq-CompCertX backend are used to compile DeepSEA specifications into certified executables. Coq is a formal proof assistant that provides a language to write programs and prove properties about them, and CompCertX is a variant of CompCert, a formally verified C compiler that can generate code for x86, PowerPC, and ARM processors.

Last semester, we decided to model a payment channel system in which two players can make payments to each other with off-chain messages and then finalize the transaction by presenting evidence of transactions to an on-chain smart contract. More concretely, we have a Player specification that defines a Record containing the state of the Player and various functions that represents actions that the Player may make (Figure 1). Similarly, we have a PmtChan specification that defines the payment channel contract state and actions that players can take on the contract (Figure 2). For an example usage of these two modules together, see Figure 3.

To prove the correctness of our system, we assume that we have a “left” player who is honest and only performs actions defined by the Player specification, and a “right” player, who is able to send any message to either the left player or the contract. We first define an inductive step relation that describes the different changes in player and contract state that can occur (Figure 4). The ultimate property about the system that we wish to prove is that the left player must be able to withdraw the amount of money that the right player owes him at all times, regardless of the steps that the right player takes. We call this property can_force (Figure 5). Currently, we are in the process of proving the can-force theorem.

The existing code can be found at https://github.com/CertiKOS/DeepSEA/tree/chrisf/ethereum/ethereum
Module Player.
Record State := { addr : Address;
            other_addr : Address;
            side : Side.Side;
            status : Status.Status;
            round_state : RoundState;
            other_rd_st_sig : option (RoundState * Sig)
            (* (round state, sig) from other player *) }.

Definition init (addr other_addr: Address) (side: Side.Side) :
    State := (* ... *)

Definition deposit (st : State) (cst : PmtChan.State) (amt : Z)
    : option PmtChan.State := (* ... *)

Definition pay (st : State) (cst : PmtChan.State) (amt : Z) :
    option (RoundState * Sig * State) := (* ... *)

Definition recv (st : State) (cst : PmtChan.State) (rd_st_sig :
    RoundState * Sig) : option State := (* ... *)

Definition trigger (st : State) (cst : PmtChan.State) : option
    (State * PmtChan.State) := (* ... *)

Definition finalize (st : State) (cst : PmtChan.State) : option
    (State * PmtChan.State) := (* ... *)

Definition update_pending_state (st : State) (cst : PmtChan.
    State) : option PmtChan.State := (* ... *)
End Player.

Figure 1: Player module with associated state and actions
Module PmtChan.

Record State := { left_player : Address;
    right_player : Address;
    left_deposit : Z;
    right_deposit : Z;
    left_withdrawal : Z;
    right_withdrawal : Z;
    left_withdrawn : Z;
    right_withdrawn : Z;
    status : Status.Status;
    pending_state : RoundState (* round number, left delta *) }.

Definition deposit (msg : Msg) (st : State) : option State :=
(* ... *)

(* Sends withdrawal - withdrawn to the sender. *)
Definition withdraw (msg : Msg) (st : State): option State :=
(* ... *)

Definition trigger (msg : Msg) (st : State) : option State :=
(* ... *)

Definition update_pending_state (msg : Msg) (st : State) (new_state_sig : RoundState * Sig) : option State := (* ... *)

Definition finalize (msg : Msg) (st : State) : option State :=
(* ... *)
End PmtChan.

Figure 2: Payment channel module with associated state and actions
Definition test :=
  let left_player := AddressMod.repr 100 in
  let right_player := AddressMod.repr 101 in
  match PmtChan.init left_player right_player with
  | Some cst =>
    let lst := init left_player right_player Side.Left in
    let rst := init right_player left_player Side.Right in
    cst <- deposit lst cst 20;
    cst <- deposit rst cst 20;
    '(rd_st0, sig0, rst) <- pay rst cst 20;
    lst <- recv lst cst (rd_st0, sig0);
    '(rd_st1, sig1, lst) <- pay lst cst 10;
    rst <- recv rst cst (rd_st1, sig1);
    '(rst, cst) <- Player.trigger rst cst;
    'cst <- Player.update_pending_state lst cst;
    'cst <- PmtChan.update_pending_state (Build_Msg (AddressMod.repr 101) 0) cst (rd_st1, sig1);
    Some cst
  | None => None
end.

Eval compute in test.

Figure 3: Using Player and PmtChan together

Inductive step : Side.Side ->
  (PlayerState * ContractState * list (RoundState * Sig)) ->
  (PlayerState * ContractState * list (RoundState * Sig)) ->
  Prop :=
  | left_pay : (* ... *)
  | right_pay : (* ... *)
  | left_trigger : (* ... *)
  | right_trigger : (* ... *)
  | left_update_pending_state : (* ... *)
  | right_update_pending_state1 : (* ... *)
  | right_update_pending_state2 : (* ... *)
  | left_finalize : (* ... *)
  | right_finalize : (* ... *).

Figure 4: Inductive step relation describing the possible changes in the system
Inductive can_force (amt : Z) (st : PlayerState * ContractState * list (RoundState * Sig)):
  Prop :=
  | finished : (let '(_, cst, _) := st in PmtChan.left_withdrawal cst >= amt) -> can_force amt st
  | left_step : forall st', step Side.Left st st' ->
    (forall st'', multistep_right st' st'' -> can_force amt st'') ->
    can_force amt st.

Theorem can_force_thm : forall left_addr right_addr st st' pst cst,
  Some cst = PmtChan.init left_addr right_addr ->
  pst = Player.init left_addr right_addr Side.Left ->
  multistep st st' -> forall pst' cst' rd_sts',
  st = (pst, cst, [I]) ->
  st' = (pst', cst', rd_sts') ->
  can_force (PmtChan.left_deposit cst' + snd (Player.
    round_state pst')) (pst', cst', rd_sts').

Figure 5: Can-force property and theorem
After we complete this proof, we want to rewrite the Coq specifications in DeepSEA and then rework the proof to work with the generated specifications from DeepSEA.

The ultimate goal is to be able to compile DeepSEA specifications into an EVM executable as well. This would require porting the Coq-CompCertX backend to generate EVM bytecode. This is outside the scope of our project, and we will only be focusing on extending the frontend of the DeepSEA compiler.

4 Deliverables

(a) Proof of correctness of the payment channel system
(b) Rewrite the Coq specifications in DeepSEA
(c) Modify the proof to work with the output from the DeepSEA compiler

References


