1 Introduction

Bitcoin [1] introduced the notion of a blockchain as a way of enabling distributed consensus among a set of mutually distrusting parties. In the general case, however, a blockchain can be thought of as a state transition system. In Bitcoin, the state is the collection of all coins that have been mined but not yet spent, mapped to the public keys of their owners and a state transition function takes a state and a transaction and outputs a new state. For example, if Alice wants to transfer $X to Bob, the initial state is the collection of all unspent coins, and the state transition function reduces the value in Alice’s account by $X and increases the value in Bob’s account by $X leading to a new state. More formally, the state transition function can be defined as APPLY(S, TX) ⇝ S’ or ERROR.

The use of blockchain in Bitcoin was limited to managing transactions that transfer coins from one account to another, but this general notion of blockchain as a state-transition system has been utilized to manage both digital [2, 3] and physical [4] assets on top of separate blockchains. In Ethereum [5], the authors intend to provide a platform for building diverse decentralized applications on top of a blockchain without the need to deploy an independent network for each application.

2 Ethereum

Ethereum achieves its goal of enabling diverse applications on top of a common ledger by building an abstraction of a blockchain with a built-in Turing-complete programming language. This allows a developer to write decentralized applications (called smart contracts) which programatically encode the rules of a multi-party interaction. In §2.1, §2.2 and §2.3 we describe the various components of the state machine underlying the Ethereum blockchain platform. We discuss the execution of the smart contracts in §2.4.

2.1 Accounts

In Ethereum, the state consists of objects called accounts which are uniquely identified by a 20-byte address. An Ethereum account is made up of four fields: (a) the nonce which is a counter that prevents replay attacks, (b) the account’s balance, specified in ether (the currency of Ethereum), (c) the account’s contract code, if present, and (d) the account’s storage, which is empty by default.

Ethereum defines two kinds of accounts based on their ownership: (a) externally owned accounts, which are owned by a person or a private entity and are controlled by a public-private keypair, and (b) contract accounts, which are controlled by the code of a smart contract.

2.2 Messages and Transactions

A state machine requires a way to transition between the various possible states. In Ethereum, this transition is achieved through the use of messages and transactions, which only differ conceptually in their initiator.
2.2.1 Transactions

A transaction refers to the signed data package that is sent from an externally owned account. A transaction consists of the following six fields: (a) a recipient of the message identified by the 20-byte account address, (b) a signature identifying the sender, (c) the amount of ether to transfer from the sender to the recipient, (d) an optional data field which can be used to pass input to the smart contract, (e) a STARTGAS value which represents the maximum number of computational steps that the transaction execution is allowed to take, and (f) a GASPRICE value which represents the fee that the sender pays per computational step.

The STARTGAS and the GASPRICE are together used to restrict the amount of computation that can be executed inside a smart contract, which is crucial to prevent infinite loops and other computational wastage inside the code. Each step executed inside a smart contract and the storage utilized has a cost attached to it. This cost is specified in a unit called gas, which is intentionally separated from Ethereum’s currency ether to ensure that the cost of each computational step is fixed and does not fluctuate with the market.

2.2.2 Messages

A message refers to a virtual object that is sent from a contract account to another contract account. A message only exists inside the execution environment and is never serialized and sent on the wire. A message consists of the following five fields: (a) the sender of the message, (b) the recipient of the message, (c) the amount of ether to transfer, (d) an optional data field and (e) a STARTGAS value.

2.3 State Transition Function

Ethereum’s state transition function takes in the transaction and the most recent state of the blockchain as an input. The output of the state transition function is a new state which gets appended to the underlying blockchain. More concretely, the state transition function APPLY(S, TX) ⇝ S’ can be described as follows:

1. Check if the transaction is well-formed, the signature is valid, and the nonce matches the nonce in the sender’s account. If not, return error.

2. Calculate the transaction fee as \( \text{STARTGAS} \times \text{GASPRICE} \), and determine the sending address from the signature. Subtract the fee from the sender’s account balance and increment the sender’s nonce. If there is not enough balance to spend, return an error.

3. Initialize \( \text{GAS} = \text{STARTGAS} \), and remove a fixed quantity of gas per byte to pay for the bytes in the transaction.
Figure 2: A transaction changes the data at address bb75a90 and reduces the balance of 14c5f8ba.

4. Transfer the transaction value from the sender’s account to the receiving account. If the receiving account does not yet exist, create it. If the receiving account is a contract, run the contract’s code either to completion or until the execution runs out of gas.

5. If the value transfer failed because the sender did not have enough money, or the code execution runs out of gas, revert all state changes except the payment of the fees, and pay the fees to the miner.

6. Otherwise, refund the fees for all remaining gas to the sender, and pay the fees for the gas consumed to the miner.

2.4 Execution of Smart Contracts

The code in Ethereum contracts is written in a low-level, stack-based bytecode language called the Ethereum Virtual Machine (EVM). While EVM is executing a smart contract, its complete state can be defined by an eight-tuple: \( \langle \text{block state, transaction, message, code, memory, stack, program counter, gas} \rangle \), where block state is the global state containing all accounts and includes balance and storage. Since the process of executing code is part of the state transition function defined in \( \S 2.3 \), the code execution occurs at all nodes participating in the blockchain.

3 Attacks on Smart Contracts

The design of Ethereum allows developers to deploy arbitrary code which manages financial instruments on top of a blockchain. As a result, smart contracts have become a lucrative attack target and many vulnerabilities in the design of contracts have been exploited leading to considerable financial loss. \[6, 7\] One of the most famous attacks on a smart contract was TheDAO attack \[8\] which led to a loss of approximately \$60 million. In \( \S 3.1 \) and \( \S 3.2 \) we discuss decentralized autonomous organizations and the attack on TheDAO.

3.1 Decentralized Autonomous Organizations (DAOs)

A decentralized autonomous organization is a contract that people purchase shares of and use these shares to vote on how to spend the contract’s ether balance. This is similar to a company in the real-world except that the behavior of a DAO is enforced by code and not by law. TheDAO was a decentralized autonomous organization launched in 2016 which aimed to function as a venture capital fund. A simplified version of TheDAO code is shown in Figure \[3\]
contract SimpleDAO {
    mapping (address => uint) public credit;
    function donate(address to) {
        credit[to] += msg.value;
    }
    function queryCredit(address to) returns (uint) {
        return credit[to];
    }
    function withdraw(uint amount) {
        if (credit[msg.sender] >= amount) {
            msg.sender.call.value(amount)();
            credit[msg.sender] -= amount;
        }
    }
}

contract Mallory {
    SimpleDAO public dao = SimpleDAO(0x354...);
    address owner;
    function Mallory() {
        owner = msg.sender;
    }
    function () {
        credit = dao.queryCredit(this);
        dao.withdraw(credit);
    }
    function getJackpot(uint amount) {
        balance = this.balance;
        owner.send(balance);
    }
}

Figure 3: A simplified version of TheDAO code. Figure 4: The attacker code.

3.2 TheDAO Attack

The bug in TheDAO allowed an attacker to steal all the ether from the SimpleDAO contract shown in Figure 3. The attack starts with an adversary publishing the contract Mallory as shown in Figure 4. Then, the adversary sends some ether to the Mallory contract which invokes the fallback function in Figure 4 at line 8. The fallback function invokes the withdraw function of the SimpleDAO contract, which sends the amount of money requested to the Mallory contract through the call function in Figure 3 at line 12. This has the side-effect of invoking the fallback function of the Mallory contract again without updating the value inside the credit mapping. Thus, Mallory calls the withdraw function again and since the credit value isn’t updated yet, the check at line 11 in Figure 3 succeeds. Consequently, the SimpleDAO contract sends the amount to the contract Mallory for a second time, and this occurs in a loop until either of the following three conditions happen: (a) the gas is exhausted, (b) the call stack is full, or (c) the balance of the SimpleDAO contract becomes zero. Thus, the adversary is able to siphon off all the ether from the SimpleDAO contract.

References