Making Smart Contracts Smarter

Loi Luu, Duc-Hiep Chu
National University of Singapore
{loiluu, hiepcd}@comp.nus.edu.sg

Hrishi Olickel
Yale-NUS College
hrishi.olickel@yale-nus.edu.sg

Prateek Saxena
National University of Singapore
prateeks@comp.nus.edu.sg

Aquinas Hobor
Yale-NUS College
& National University of Singapore
hobor@comp.nus.edu.sg

ABSTRACT

Cryptocurrencies record transactions in a decentralized data structure called a blockchain. Two of the most popular cryptocurrencies, Bitcoin and Ethereum, support the feature to encode rules or scripts for processing transactions. This feature has evolved to give practical shape to the ideas of smart contracts, or full-fledged programs that are run on blockchains. Recently, Ethereum's smart contract system has seen steady adoption, supporting tens of thousands of contracts, holding tens of millions dollars worth of virtual coins.

In this paper, we investigate the security of running Ethereum smart contracts in an open distributed network like those of cryptocurrencies. We introduce several new security problems in which an adversary can manipulate smart contract execution to gain profit. These bugs suggest subtle gaps in the understanding of the distributed semantics of the underlying platform. As a refinement, we propose ways to enhance the operational semantics of Ethereum to make contracts less vulnerable. For developers writing contracts for the existing Ethereum system, we build a symbolic execution tool called Oyente to find potential security bugs. Among 19,366 existing Ethereum contracts, Oyente flags 8,519 of them as vulnerable. We discuss the severity of attacks for several case studies which have source code available and confirm the attacks (which target only our accounts) in the main Ethereum network.

1. INTRODUCTION

Decentralized cryptocurrencies have gained considerable interest and adoption since Bitcoin was introduced in 2009 [1]. At a high level, cryptocurrencies are administered publicly by users in their network without relying on any trusted parties. Users in a cryptocurrency network run a consensus protocol to maintain and secure a shared ledger of data (the *blockchain*). Blockchain technology was initially used for peer-to-peer Bitcoin payments [1], but more recently, it has been used more broadly [2–4]. One prominent new use for blockchain technology is to enable *smart contracts*.

A smart contract is a program that runs on the blockchain and has its correct execution enforced by the consensus protocol [5]. A contract can encode any set of rules represented in its programming language—for instance, a contract can execute transfers when certain events happen (e.g. payment of security deposits in an escrow system). Accordingly, smart contracts can implement a wide range of applications, including financial instruments (*e.g.*, sub-currencies, financial derivatives, savings wallets, wills) and self-enforcing or autonomous governance applications (*e.g.*, outsourced computation [6], decentralized gambling [7]).

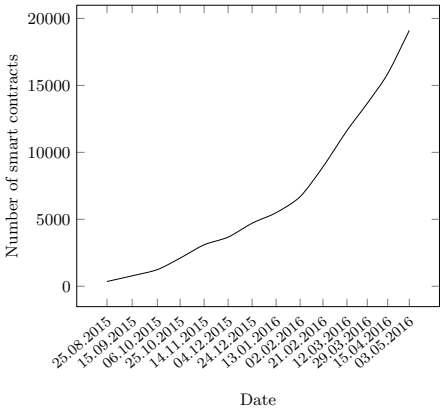

Figure 1: Number of smart contracts in Ethereum has increased rapidly.

A smart contract is identified by an address (a 160-bit identifier) and its code resides on the blockchain. Users invoke a smart contract in present cryptocurrencies by sending transactions to the contract address. Specifically, if a new transaction is accepted by the blockchain and has a contract address as the recipient, then all participants on the mining network execute the contract code with the current state of the blockchain and the transaction payloads as inputs. The network then agrees on the output and the next state of the contract by a consensus protocol. Ethereum, a more recent cryptocurrency, is a prominent Turing-complete smart contract platform [2]. Unlike Bitcoin, Ethereum supports stateful contracts in which values can persist on the blockchain to be used in multiple invocations. In the last six months alone, roughly 15,000 smart contracts have been deployed in the Ethereum network, suggesting a steady growth in the usage of the platform (see Figure 1). As Ethereum receives more public exposure and other similar projects like Rootstock [8] and CounterParty [9] emerge on top of the Bitcoin blockchain, we expect the number of smart contracts to grow.

Security problems in smart contracts. Smart contracts can handle large numbers of virtual coins worth hundreds of dollars apiece, easily making financial incentives high enough to attract adversaries. Unlike traditional distributed application platforms, smart contract platforms such as Ethereum operate in open (or permissionless) networks into which arbitrary participants can join. Thus, their execution is vulner-

able to attempted manipulation by arbitrary adversaries—a threat that is restricted to accidental failures in traditional permissioned networks such as centralized cloud services [10, 11]. Although users in Ethereum have to follow a predefined protocol when participating in the network, we show that there is considerable room for manipulation of a smart contract's execution by the network participants. For example, Ethereum (and Bitcoin) allow network participants (or miners) to decide which transactions to accept, how to order transactions, set the block timestamp and so on. Contracts which depend on any of these sources need to be aware of the subtle semantics of the underlying platform and explicitly guard against manipulation.

Unfortunately, the security of smart contracts has not received much attention, although several anecdotal incidents of smart contracts malfunctioning have recently been reported, including contracts that do not execute as expected [7, 12, 13] and/or that have locked away thousands of dollars worth of virtual coins [7, 13]. In contrast to classical distributed applications that can be patched when bugs are detected, smart contracts are irreversible and immutable. There is no way to patch a buggy smart contract, regardless of its popularity or how much money it has, without reversing the blockchain (a formidable task). Accordingly, reasoning about the correctness of smart contracts before deployment is critical, as is designing a safe smart contract system.

In this paper, we document several new security flaws of Ethereum smart contracts and give examples of real-world instances for each problem. These security flaws make contracts susceptible to abuse by several parties (including miners and contracts' users). We believe that these flaws arise in practice because of a *semantic gap* between the assumptions contract writers make about the underlying execution semantics and the actual semantics of the smart contract system. Specifically, we show how different parties can exploit contracts which have differing output states depending on the order of transactions and input block timestamp. To our knowledge, these semantic gaps have not been previously discussed or identified. We also document other serious but known problems such as improperly handled aborts/exceptions and logical flaws. Previous work has discussed these conceptually, often with simple self-constructed examples [14]. In our work, we study their impact on tens of thousands of real-life contracts, showing how these vulnerabilities can be used to sabotage or steal coins from benign users.

More importantly, our work emphasizes the subtle and/or missing abstractions in smart contract semantics that lead developers to a false sense of security. We propose refinements to Ethereum's protocol that do not require changes to existing smart contracts. However, such solutions do require all clients in the network to upgrade, thus running the risk of not seeing real deployment. If such a requirement is unacceptable, we provide a tool called Oyente for users to detect bugs in their contracts as a pre-deployment mitigation. Oyente is a symbolic execution tool exclusively designed to analyze Ethereum smart contracts. It follows closely the execution model of Ethereum smart contracts [15] and directly works with Ethereum virtual machine (EVM) code without access to the high level representation (*e.g.*, Solidity [16], Serpent [17]). This design choice is vital because the Ethereum blockchain only stores the EVM code of contracts, not their source.

Evaluation. We ran Oyente on 19,366 smart contracts from the first 1,460,000 blocks in Ethereum network and found that 8,519 contracts have the documented bugs. These contracts currently have a total balance of 6,169,802 Ethers, approximately equivalent to 62 million USD at the time of writing. We further discuss our results and our verified attack with one of the most active contracts of Ethereum (effecting only our own accounts), in Section 6.

Although we use Ethereum's smart contracts throughout this paper, our security problems and Oyente are largely platform agnostic. We suspect these kinds of problems exist in CounterParty's or Rootstock's networks and, once these blockchains are publicly available, can extend Oyente to them as well.

Contributions. This paper makes the following contributions.

- We document several new classes of security bugs in Ethereum smart contracts.
- We formalize the semantics of Ethereum smart contracts and propose recomendations as solutions for the documented bugs.
- We provide Oyente, a symbolic execution tool which analyses current Ethereum smart contracts and detect bugs.
- We run Oyente on real Ethereum smart contracts confirmed the attacks in the real Ethereum network.

2. BACKGROUND

We give a brief introduction to smart contracts and their execution model. Our discussion is restricted to most popular smart contract platform called Ethereum, but the security problems discussed in this paper are of wider application to other open distributed application platforms.

2.1 Consensus Protocol

Decentralized cryptocurrencies secure and maintain a shared ledger of facts between a set of peer-to-peer network operators (or miners). Miners run a peer-to-peer consensus protocol called the *Nakamoto consensus* protocol. The shared ledger is called a blockchain and is replicated by all miners. The ledger is organized as a hash-chain of blocks ordered by time, wherein each block has a set of facts, as shown in Figure 2. In every epoch, each miner proposes their own block to update the blockchain. Miners can select a sequence of new transactions to be included in the proposed block. At a high level, Nakamoto consensus works by probabilistically electing a leader among all the miners via a *proof-of-work* puzzle [1]. The leader then broadcasts its proposed block to all miners. If the proposed block obeys a certain predefined validity constraints, such as those ensuring mitigation of "double-spending" attacks, then all miners update their ledger to include the new block. We exclude certain details about the consensus protocol, such as the use of the longest-chain rule for resolving probabilistic discrepancies in leader election. Instead, we refer readers to the original Bitcoin or Ethereum paper for details [1, 2].

A blockchain state σ is a mapping from addresses to accounts; thus the state of an account at address γ is $\sigma[\gamma]$. While Bitcoin only has normal accounts which hold some coins, Ethereum additionally supports smart contract ac-

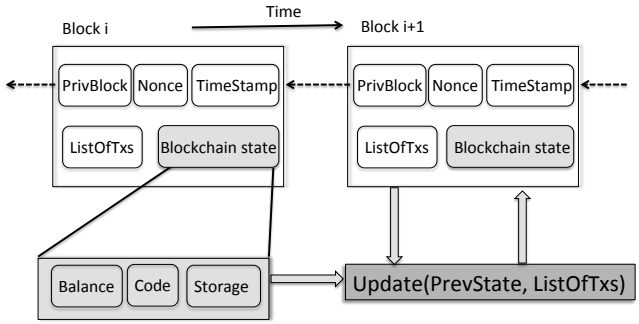

Figure 2: The blockchain's design in popular cryptocurrencies like Bitcoin and Ethereum. Each block consists of several transactions.

counts which have coins, executable code and persistent (private) storage. Ethereum supports its own currency called Ether; users can transfer coins to each other using normal transactions as in Bitcoin, and *additionally* can invoke contracts using contract-invoking transactions. Conceptually, Ethereum can be viewed as a transaction-based state machine, where its state is updated after every transaction. A valid transition from σ to σ', via transaction T is denoted as $\sigma \xrightarrow{T} \sigma'$.

2.2 Smart Contracts in Ethereum

A *smart contract* (or contract for short) is an "autonomous agent" stored in the blockchain, encoded as part of a "creation" transaction that introduces a contract to the blockchain. Once successfully created, a smart contract is identified by a *contract address*; each contract holds some amount of of virtual coins (Ether), has its own private storage, and is associated with its predefined executable code. A contract *state* consists of two main parts: a private storage and the amount of virtual coins (Ether) it holds (called *balance*). Contract code can manipulate variables like in traditional imperative programs. The code of an Ethereum contract is in a low-level, stack-based bytecode language referred to as Ethereum virtual machine (EVM) code. Users define contracts using high-level programming languages, *e.g.*, Solidity [16] (a JavaScript-like language), which are then compiled into EVM code. To invoke a contract at address γ, users send a transaction to the contract address. A transaction typically includes: payment (to the contract) for the execution (in Ether) and/ or input data for the invocation.

An Example. Figure 3 is a simple contract, defined in Solidity, which rewards anyone who solves a computational puzzle and submits the solution to the contract. The creator of the contract includes the compiled (EVM) code of Puzzle in a "contract creation" transaction. When such transaction is accepted to the blockchain, all miners will unanimously modify the blockchain state σ, adding a new contract account in three following steps. First, a unique address for the new contract is prepared. Then the contract's private storage is allocated and initialized by running the constructor (*i.e.*, Puzzle() function in Line 8). Finally, the executable EVM code portion that corresponds to the anonymous function (Line 15 onwards) is associated with the contract.

Any transaction invoking this contract will execute the anonymous `function()` (Line 15) by default. The information of the sender, the value (amount of Ether sent to the contract) and the included data of the invocation transaction is stored in a default input variable called `msg`. For example,

```
1 contract Puzzle{
2   address public owner;
3   bool public locked;
4   uint public reward;
5   bytes32 public diff;
6   bytes public solution;
7
8   function Puzzle() //constructor{
9     owner = msg.sender;
10    reward = msg.value;
11    locked = false;
12    diff = bytes32(11111); //pre-defined difficulty
13  }
14
15  function(){ //main code, runs at every invocation
16    if (msg.sender == owner){ //update reward
17      if (locked)
18        throw;
19      owner.send(reward);
20      reward = msg.value;
21    }
22    else
23      if (msg.data.length > 0){ //submit a solution
24        if (locked) throw;
25        if (sha256(msg.data) < diff){
26          msg.sender.send(reward); //send reward
27          solution = msg.data;
28          locked = true;
29        }}}}
```

Figure 3: A contract that rewards users who solve a computational puzzle.

the contract owner updates the `reward` variable by invoking a transaction T_o with some Ether (*i.e.*, in `msg.value`). Before updating the `reward` variable to the new value (Line 20), the contract indeed sends back to the owner, an amount of Ether that is equal to the current `reward` value (Line 19). The result of T_o is a new state of Puzzle which has a different `reward` value. Similarly, users can submit their solution in a different transaction payload (*e.g.*, `msg.data`) to claim the reward (Line 22–29). If the solution is correct, the contract sends the reward to the submitter (Line 26).

Gas system. By design, smart contract is a mechanism to execute code distributively. To ensure fair compensation for expended computation effort by the mining network, Ethereum pays miners some fees proportional to the required computation. Specifically, each instruction in the Ethereum bytecode has a pre-specified amount of *gas*. When a user sends a transaction to invoke a contract, she has to specify how much gas she is willing to provide for the execution (called `gasLimit`) as well as the price for each gas unit (called `gasPrice`). A miner who includes the transaction in his proposed block subsequently receives the transaction fee corresponding to the amount of gas the execution actually burns multiplied by `gasPrice`. If some execution requires more gas than `gasLimit`, the execution is terminated with an exception, the state σ is reverted to the initial state as if the execution does not happen. In case of such aborts, the sender still has to pay all the `gasLimit` to the miner though, as a counter-measure against resource-exhaustion attacks [6].

3. SECURITY BUGS IN CONTRACTS

We discuss several security bugs which allow malicious miners or users to exploit and gain profit.

3.1 Transaction-Ordering Dependence

As discussed in Section 2, a block includes a set of transactions, hence the blockchain state is updated several times in

```
1 contract MarketPlace{
2   uint public price;
3   uint public stock;
4   /.../
5   function updatePrice(uint _price){
6     if (msg.sender == owner)
7       price = _price;
8   }
9   function buy (uint quant) returns (uint){
10    if (msg.value < quant * price || quant > stock)
11      throw;
12    stock -= quant;
13    /.../
14  }}
```

Figure 4: A contract that acts as a market place where users can buy/ sell some tokens. Due to TOD, some order may or may not go through.

each epoch. Let us consider a scenario where the blockchain is at state σ and the new block includes two transactions (*e.g.*, T_i, T_j) invoking the same contract. In such a scenario, users have uncertain knowledge of which state the contract is at when their individual invocation is executed. For example, T_i is applied when the contract is at either state $\sigma[\alpha]$ or state $\sigma'[\alpha]$ where $\sigma \xrightarrow{T_j} \sigma'$, depending on the order between T_i and T_j. Thus, there is a discrepancy between the state of the contract that users may intend to invoke at, and the actual state when their corresponding execution happens. Only the miner who mines the block can decide the order of these transactions, consequently the order of updates. As a result, the final state of a contract really depends on how the miner orders the transactions invoking it. We call such contracts as transaction-ordering-dependent (or TOD) contracts.

3.1.1 Attacks

It may be not obvious to the readers why having dependence on the transaction ordering is problematic for smart contracts. The reasons are twofold. First, even a benign invocation to the contract may yield an unexpected result to users if there are concurrent invocations. Second, a malicious user can exploit the TOD contracts to gain more profits, even steal users' money. We explain the two scenarios below by using the Puzzle contract in Figure 3.

Benign scenario. We consider two transactions T_o and T_u sending to Puzzle at roughly the same time. T_o is from the contract owner to update the reward and T_u is from a user who submits a valid solution to claim the reward. Since T_o and T_u are broadcast to the network at roughly the same time, the next block will most likely include both the transactions. The order of the two transactions decides how much reward the user receives for the solution. The user expects to receive the reward that he observes when submitting his solution, but he may receive a different reward if T_o is executed first. The problem is more significant if the contract serves as a decentralized exchange or market place [18, 19]. In these contracts, sellers frequently update the price, and users send their orders to buy some items (Figure 4). Depending on the transaction ordering, users' buy requests may or may not go through. Even worse, buyers may have to pay much higher than the observed price when they issue the buy requests.

Malicious scenario. The above scenario may just be an accident because the owner of Puzzle does not know when a solution is submitted. However, a malicious owner can exploit transaction-ordering dependence to gain financial advantage. Note that there is a time gap between when transaction T_u is broadcast and when it is included in a block. In Ethereum, the time to find a new block is around 12 seconds. Thus, a malicious owner of the Puzzle contract can keep listening to the network to see if there is a transaction which submits a solution to his contract. If so, he sends his transaction T_o to update the reward and make it as small as zero. With a certain chance, both T_o and T_u are included in the new block and his T_o is placed (so executed) before the T_c. Thus the owner can enjoy a free solution for his puzzle. The owner can even bias the chance of his transaction winning the race (*i.e.*, to be executed first) by participating directly in the mining, setting higher `gasPrice` for his transaction (*i.e.*, to incentivize miners to include it in the next block) and/or colluding with other miners.

```
1 contract theRun {
2   uint private Last_Payout = 0;
3   uint256 salt =  block.timestamp;
4   function random returns (uint256 result){
5     uint256 y = salt * block.number/(salt%5);
6     uint256 seed = block.number/3 + (salt%300)
7                  + Last_Payout +y;
8     //h = the blockhash of the seed-th last block
9     uint256 h = uint256(block.blockhash(seed));
10    //random number between 1 and 100
11    return uint256(h % 100) + 1;
12  }}
```

Figure 5: A real contract which depends on block timestamp to send out money [20]. This code is simplified from the original code to save space.

3.2 Timestamp Dependence

The next security problem that a contract may have uses the block timestamp as a triggering condition to execute some critical operations, *e.g.*, sending money. We call such contracts as timestamp-dependent contracts.

A good example of a timestamp-dependent contract is the theRun contract in Figure 5, which uses a homegrown random number generator to decide who wins the jackpot [20]. Technically TheRun uses the hash of some previous block as the random seed to select the winner (Line 9–10). The choice of block is determined based on the current block timestamp (Line 5–7).

Let us recall that when mining a block, a miner has to set the timestamp for the block (Figure 2). Normally, the timestamp is set as the current time of the miner's local system. However, the miner can vary this value by roughly 900 seconds, while still having other miners accept the block [21] [1]. Specifically, on receiving a new block and after checking other validity checks, miners check if the block timestamp is greater than the timestamp of previous block and is within 900 seconds from the timestamp on their local system. Thus, the adversary can choose different block timestamps to manipulate the outcome of timestamp-dependent contracts.

3.2.1 Attacks

A miner can set the block timestamp to be a specific value which influences the value of the timestamp-dependent condition and favor the miner. For example in the theRun

[1]The variation of block timestamp may now be less than 900 seconds according to the fact that Ethereum requires nodes in the network to have roughly "same" local timestamps. However, we could not find any updated document from Ethereum regarding the new possible variation, thus keeping 900 seconds.

```
1 contract KingOfTheEtherThrone {
2   struct Monarch {
3     // address of the king.
4     address ethAddr;
5     string name;
6     // how much he pays to previous king
7     uint claimPrice;
8     uint coronationTimestamp;
9     }
10  Monarch public currentMonarch;
11  // claim the throne
12  function claimThrone(string name) {
13    /.../
14    if (currentMonarch.ethAddr != wizardAddress)
15      currentMonarch.ethAddr.send(compensation);
16    /.../
17    // assign the new king
18    currentMonarch = Monarch(
19        msg.sender, name,
20        valuePaid, block.timestamp);
21  }}
```

Figure 6: A code snippet of a real contract which does not check the return value after calling other contracts [12].

contract, the hash of previous block and block number are known, other contract variables like last_payout which contribute to the generation of the random seed are also known. Thus the miner can precompute and select the timestamp so the function random produces an outcome that favors him. As a result, the adversary may completely bias the outcome of the random seed to be any value, thus awarding the jackpot to any player he pleases. Thus, theRun is vulnerable to any adversary who can manipulate the block timestamp.

While theRun uses timestamp as a insecure deterministic random seed, other contracts use block timestamp as global timestamp and perform some time-dependent computation. We show in Section 6 some contract which uses timestamp for this purpose and is vulnerable to manipulation. Unfortunately, there are many other contracts which are timestamp-dependent. As we show in Section 6, among the first 19,366 contracts, 83 of them depend on the block timestamp to transfer Ether to different addresses.

3.3 Mishandled Exceptions

In Ethereum, there are several ways for a contract to call another, *e.g.*, via send instruction or call a contract's function directly (*e.g.*,aContract.someFunction()). If there is an exception raised (*e.g.*, not enough gas, exceeding call stack limit) in the callee contract, the callee contract terminates, reverts its state and returns `false`. However, depending on how the call is made, the exception in the callee contract may or may not get propagated to the caller. For example, if the call is made via the send instruction, the caller contract should *explicitly* check the return value to verify if the call has been executed properly. This inconsistent exception propagation policy leads to many cases where exceptions are not handled properly. As we later show in Section 6, 27.9% of the contracts do not check the return values after calling other contracts via send. We discuss the threats of not validating return value of a contract call via our example in Figure 6, which is a code snippet of a real contract [12].

3.3.1 Attacks

The KingOfTheEtherThrone (KoET for short) contract in Figure 6 allows users to claim as "king of Ether" by paying some amount of Ether that the current king requires. A king gets profit (*i.e.*, compensation) from the difference between the price he paid to the king before him and the price other pays to be his successor. When a user claims the throne, the contract sends the compensation to the ceded king (Line 15), and assigns the user as the new king (Line 18—20).

The KoET contract does not check the result of the compensation transaction in Line 15 before assigning the new king. Thus, if for some reason, the compensation transaction does not finish properly, the current king loses his throne without any compensation. In fact, an instance of such a problem occurred and led to the termination of KoET [12]. The reason reported in [12] is that the address A_k of the current king is not a normal address, but a contract address. When sending a transaction (or a call) to A_k, some code will be executed, thus requiring more gas than a transaction to a normal address. However, KoET does not know what A_k executes internally beforehand to decide how much gas to give in the send instruction. Hence A_k runs out of gas and throws an exception. As a result, the state (including balance) of A_k remains unchanged, the compensation is returned to KoET and the current king loses his throne without any compensation.

The above problem often arises when a contract sends money to a dynamic address, since the sender does not know how much gas to allocate for the transaction. The contract should always check if the transaction succeeds before executing subsequent logic. The callee contract may throw an exception of any type (*e.g.*, division by zero, array index out of bound and so on).

In this scenario, the recipient (or callee) seems at fault, causing the send to fail. However, as we show next, a malicious user who invokes the caller contract can cause the send to fail deliberately, regardless of what the callee does.

Deliberately exceeding the call-stack's depth limit. The Ethereum Virtual Machine implementation limits the call-stack's depth to 1024 frames. The call-stack's depth increases by one if a contract calls another via the send or call instruction. This opens an attack vector to deliberately cause the send instruction in Line 15 of KoET contract to fail. Specifically, an attacker can prepare a contract to call itself 1023 times before sending a transaction to KoET to claim the throne from the current king. Thus, the attacker ensures that the call-stack's depth of KoET reaches 1024, causing the send instruction in Line 15 to fail. As the result, the current king will not receive any payment. This call-stack problem was identified in a previous report [22], but remains unfixed in the current Ethereum protocol.

Exploiting call stack limit to gain benefit. In the previous attack to KoET, the attacker does not receive any direct benefit besides causing other users to lose their entitlement. However, in many other examples, the attacker can exploit to directly benefit as well. In Ethereum, many contracts implement verifiable Ponzi (or pyramid) schemes [20,23,24]. These contracts pay investors interest for their investments from the subsequent investments by others. An attacker can invest his money, make payments to previous investors fail so he can receive his interest early. We discuss one of such contracts in Section 6. Specifically, in Section 6, we show that 5,411 contracts (27.9%) are vulnerable to this deliberately exceeding call-depth limit attack. We also conduct the attack in one of the most populars contract in Ethereum (with no harm to others, but our own accounts) to confirm our finding.

4. TOWARDS A BETTER DESIGN

We formalize a "lightweight" semantics for Ethereum in Section 4.1, and then build on this formalism in Section 4.2 to recommend solutions to the security issues identified in Section 3. Despite being lightweight, our formalism *rigorously* captures interesting features of Ethereum, exposing the subtleties in its semantics, which further enables us to state our proposed solutions precisely.

We use the following notation: $\leftarrow$ to denote assignment, $\bullet$ to denote an arbitrary element (when the specific value is important), $\Downarrow$ to denote big-step evaluation, and $\rightsquigarrow$ to denote small-step evaluation. Finally, $a[i \mapsto v]$ returns a new array identical to a, but on position i it contains the value v; this notation of array update also applies to nested arrays.

4.1 Operational Semantics of Ethereum

Recall (from Section 2) that a canonical state of Ethereum, denoted by σ, is a mapping between addresses and account states. We write a valid transition from σ to σ' via transaction T as $\sigma \xrightarrow{T} \sigma'$.

Formation and Validation of a Block. To model the formation of the blockchain and the execution of blocks, we define a global Ethereum state as a pair $\langle BC, \sigma \rangle$, where BC is the current block chain and σ is as before. Γ denotes the stream of incoming new transactions. For simplicity, we do not model miner rewards.

$$\textsc{Propose}\ \frac{\begin{array}{c}\texttt{TXs} \leftarrow \text{ Some transaction sequence } (T_1 \dots T_n) \text{ from } \Gamma \\ B \leftarrow \langle \texttt{blockid}\ ; \texttt{timestamp}\ ; \texttt{TXs}\ ; \dots \rangle \\ \texttt{proof-of-work}(B, BC) \\ \forall\, i, 1 \le i \le n : \sigma_{i-1} \xrightarrow{T_i} \sigma_i\end{array}}{\langle BC, \sigma_0 \rangle \Downarrow \langle B \cdot BC, \sigma_n \rangle}$$

Remove $T_1 \dots T_n$ from Γ and broadcast B

$$\textsc{Accept}\ \frac{\begin{array}{c}\text{Receive } B \equiv \langle \texttt{blockid}\ ; \texttt{timestamp}\ ; \texttt{TXs}\ ; \dots \rangle \\ \texttt{TXs} \equiv (T_1 \dots T_n) \\ \forall\, i, 1 \le i \le n : \sigma_{i-1} \xrightarrow{T_i} \sigma_i\end{array}}{\langle BC, \sigma_0 \rangle \Downarrow \langle B \cdot BC, \sigma_n \rangle}$$

Remove $T_1 \dots T_n$ from Γ and broadcast B

Figure 7: Proposing and Accepting a Block

The actions of the miners to form and validate blocks are given in Figure 7. Only one "elected leader" executes *successfully* the PROPOSE rule at a time. Other miners use the ACCEPT rule to "repeat" the transitions $\sigma_{i-1} \xrightarrow{T_i} \sigma_i$ after the leader broadcasts block B.

Security Issues: As discussed earlier, the issue of *timestamp-dependence* arises because the elected leader has some slack in setting the timestamp, yet other miners still accept the block. On the other hand, the issue of *transaction-ordering-dependence* exists because of some inevitable order among T_i; yet we have shown that when dealing with Ether (or money), this might lead to undesirable outcomes.

Transaction Execution. A transaction can activate the code execution of a contract. In Ethereum, the execution can access to three types of space in which to store data: (1) an operand LIFO stack s; (2) an auxiliary memory l, an infinitely expandable array; and (3) the contract's long-term storage str, which is part of $\sigma[id]$ for a given contract address id. Unlike stack and auxiliary memory, which reset after computation ends, storage persists as part of σ.

We define a virtual machine's execution state μ as a configuration $\langle A, \sigma \rangle$, where A is a call stack (of activation records) and σ is as before. The activation record stack is defined as:

$$\begin{array}{rcl} A & \triangleq & A_{normal} \mid \langle e \rangle_{exc} \cdot A_{normal} \\ A_{normal} & \triangleq & \langle M, pc, l, s \rangle \cdot A_{normal} \mid \epsilon \end{array}$$

where ϵ denotes an empty call stack; $\langle e \rangle_{exc}$ denotes that an exception has been thrown; and each part of an activation record $\langle M, pc, l, s \rangle$ has the following meaning:

M : the contract code array
pc : the address of the next instruction to be executed
l : an auxiliary memory (e.g. for inputs , outputs)
s : an operand stack.

Though a transaction in Ethereum is a complex structure and specifies a number of fields, we abstract it to a triple $\langle id, v, l \rangle$ where id is the identifier of the to-be-invoked contract, v is the value to be deposited to the contract, and finally l is an data array capturing the values of input parameters. Thus a transaction execution can be modeled with the rules in Figure 8: the first rule describes an execution that terminates successfully (or "normal halting") while the second rules describes one that terminates with an exception.

Note that the execution of a transaction is intended to follow the "transactional semantics" of which two important properties are: (1) *Atomicity*, requiring that each transaction be "all or nothing". In other words, if one part of the transaction fails, then the entire transaction fails and the state is left unchanged; and (2) *Consistency*, ensuring that any transaction will bring the system from one valid state to another. We will show, later in this section, how these properties might be violated, when we discuss the operational semantics of EVM instructions.

$$\text{TX-SUCCESS}\ \frac{\begin{array}{c}T \equiv \langle id, v, l \rangle \qquad M \leftarrow \texttt{Lookup}(\sigma, id) \\ \sigma' \leftarrow \sigma[id][bal \mapsto (\sigma[id][bal] + v)] \\ \langle \langle M, 0, l, \epsilon \rangle \cdot \epsilon, \sigma' \rangle \rightsquigarrow^* \langle \epsilon, \sigma'' \rangle\end{array}}{\sigma \xrightarrow{T} \sigma''}$$

$$\text{TX-EXCEPTION}\ \frac{\begin{array}{c}T \equiv \langle id, v, l \rangle \qquad M \leftarrow \texttt{Lookup}(\sigma, id) \\ \sigma' \leftarrow \sigma[id][bal \mapsto (\sigma[id][bal] + v)] \\ \langle \langle M, 0, l, \epsilon \rangle \cdot \epsilon, \sigma' \rangle \rightsquigarrow^* \langle \langle e \rangle_{exc} \cdot \epsilon, \bullet \rangle\end{array}}{\sigma \xrightarrow{T} \sigma}$$

Figure 8: Rules for Transaction Execution. $\texttt{Lookup}(\sigma, id)$ finds the associated code of contract address id in state σ; $\sigma[id][bal]$ refers to the balance of the contract at address id in state σ.

Execution of EVM Instructions. We have distilled EVM into a language ETHERLITE, which is a stack machine augmented with a memory and some Ethereum-like features. The instructions $ins \in$ instruction of ETHERLITE are:

$$\begin{array}{rcl} ins & \triangleq & \texttt{push}\ v \mid \texttt{pop} \mid \texttt{op} \mid \texttt{bne} \mid \\ & & \texttt{mload} \mid \texttt{mstore} \mid \texttt{sload} \mid \texttt{sstore} \mid \\ & & \texttt{call} \mid \texttt{return} \mid \texttt{suicide} \mid \texttt{create} \mid \texttt{getstate} \end{array}$$

The push instruction takes an argument $v \in$ value which is either a numeric constant z, code label λ, memory address

Table 1: Operational Semantics of `call` and `return`. EXC stands for "Exception".

$M[pc]$	Conditions	μ	μ'
`call`	$id \leftarrow$ address of the executing contract $a' \leftarrow \langle M, pc, l, s\rangle$ $M' \leftarrow \texttt{Lookup}(\sigma, \gamma)$ $\sigma' \leftarrow \sigma[id][bal \mapsto \sigma[id][bal] - z]$ $\sigma'' \leftarrow \sigma'[\gamma][bal \mapsto \sigma[id][bal] + z]$	$\langle\langle M, pc, l, \gamma \cdot z \cdot st \cdot sz \cdot s\rangle \cdot A, \sigma\rangle$	$\langle\langle M', 0, l', \epsilon\rangle \cdot a' \cdot A, \sigma''\rangle$
`call`	$id \leftarrow$ address of the executing contract $\sigma[id][bal] < v$ or $\lvert A\rvert = 1023$	$\langle\langle M, pc, l, \bullet \cdot v \cdot \bullet \cdot \bullet \cdot \bullet \cdot \bullet \cdot s\rangle \cdot A, \sigma\rangle$	$\langle\langle M, pc+1, l, 0 \cdot s\rangle \cdot A, \sigma\rangle$
`return`		$\langle\langle M, pc, \bullet, \bullet\rangle \cdot \epsilon, \sigma\rangle$	$\langle \epsilon, \sigma\rangle$
`return`	$a' \equiv \langle M', pc', l'_0, st' \cdot sz' \cdot s'\rangle$ $n \leftarrow min(sz', sz)$ $0 \le i < n : l'_{i+1} \leftarrow l'_i[st' + i \mapsto l[st + i]]$	$\langle\langle M, pc, l, st \cdot sz \cdot s\rangle \cdot a' \cdot A, \sigma\rangle$	$\langle\langle M', pc'+1, l'_n, 1 \cdot s'\rangle \cdot A, \sigma\rangle$
EXC	exceptional halting of callee	$\langle e\rangle_{exc} \cdot \langle M, pc, l, st \cdot sz \cdot s\rangle \cdot A, \sigma\rangle$	$\langle\langle M, pc+1, l, 0 \cdot s\rangle \cdot A, \sigma\rangle$

α, or contract/recipient address γ and adds it to the top of the "operand stack". The `pop` instruction removes (forgets) the top element of the operand stack. The `op` instruction, representing all of the arithmetic and logical etc. operations, pops its arguments, performs the operation, and pushes the result. Conditional branch `bne` is a standard "branch if not equal to zero". It pops two elements z and λ from the top of the operand stack; if z is nonzero then the program counter is set to λ, otherwise it is the program counter is incremented. The load and store instructions respectively reads from and writes to memory in the natural way. However, here we have two types of load and store, dealing with two types of memory mentioned above. While `mload` and `mstore` deal with the auxiliary memory l, `sload` and `sstore` respectively assesses and updates the contract storage *str*, i.e., the state of the contract.

Let us now discuss more interesting instructions inspired from Ethereum. The key instructions are `call` and `return`, whose operational semantics are provided in Table 1[2]. Each row describes the conditions under which an execution can move from configuration μ to configuration μ', i.e. $\mu \rightsquigarrow \mu'$. The first column indicates the instruction form captured by the rule. If the instruction about to be executed matches that form and all the (side) conditions in the second column are satisfied, then a step may be made from a configuration matching the pattern in the third column to a configuration matching the pattern in the last column.

A `call` instruction is roughly analogous to a remote procedure call[3]. The arguments, placed on the operand stack, are the destination γ, amount of Ether to transfer z, and two values st and sz (stand for "start address" and "size") to specify a slice of memory which contains additional function-specific parameters. The next two values in the operand stack similarly specify a place for the return value(s); they are exposed (in the rules) when the call is returned, or if an exception has occurred. Note that unlike the operand stack, which has no fixed maximum size, the call stack has a maximum size of $1,024$. If the call stack is already full then the remote call will cause an exception (second rule for `call`). When the remote call `return`s, a special flag is placed onto the operand stack, with 1 indicating a successful call (second rule for `return`) and 0 indicating an unspecified exception (rule EXC).

[2]For completeness, operational semantics of other instructions are provided in the supplementary material

[3]Ethereum has several additional variants of call, including `CALLCODE` and `DELEGATECALL` which we do not model in EtherLite

There are *two* important points to note. First, an exception at a callee is not limited to (call) stack overflow. It could be due to various reasons such as gas exhaustion, division by zero, etc. Second, exceptions are *not propagated automatically*. Contract writers who wish to do so must explicitly check for the 0 and then raise a new exception, typically by jumping to an invalid label. For certain high-level commands in Solidity, a code snippet to perform these steps is inserted by the compiler.

Security Issues: Recall the security issues discussed in Section 3.3, in particular when exceptions are mishandled. The root cause of the problem is in the inconsistency of how exceptions influence the final state, depending whether a contract method is invoked as a transaction, or via the `call` instruction. In the former case, rule TX-EXCEPTION in Figure 8, the execution is *aborted*; while in the latter case, row EXC Table 1, a flag 0 is pushed into the operand stack of the caller. The way that an exception occurs at a callee is converted into a flag 0 (and the execution continues normally) indeed *breaks* the atomicity property. In other words, Ethereum transactions do not have *atomicity* in their semantics. This can lead to serious consequences, given that money transfers in Ethereum are mostly done using the `call` instruction.

There are three remaining instructions: `suicide`, `create`, and `getstate`. The `suicide` instruction transfers all of the remaining Ether to recipient γ and then terminates the contract; although somewhat similar to `call` in that Ether changes hands, it does not use the call stack. The `create` instruction creates a new contract account, taking three arguments from the operand stack. They are the amount of Ether to be put in the new contract, and two values to specify a slice of memory which contains the bytecode for the new contract. It proceeds in three steps:

1. Creating a new address and allocating storage for the new contract. The specified amount of Ether is also deposited into the contract.
2. Initializing the contract's storage.
3. Depositing the associated code body into the contract.

If the contract creation is successful, the address of new contract is pushed onto the operand stack; otherwise, a flag value of 0 is pushed. The three above-mentioned steps rely on certain helper procedures, which we will not attempt to capture with our formalism. Note that: (1) while the initialization code is executing, the newly created address exists but with no intrinsic body code; and (2) if the initialization

ends up with an exception then the state is left with a "zombie" account, and any remaining balance will be locked into the account forever. In other words, an unsuccessful contract creation might lead to an invalid contract residing in the system, breaking the *consistency* property of the "transactional semantics". This issue might not directly lead to some security attacks, it is clearly undesirable in the current design of Ethereum.

Lastly, `getstate` is an *abstract* instruction of which the concrete instance related to the security problem in 3.2 is to get the current block timestamp. A `getstate` instruction typically pushes certain "special" value onto the stack: in particular, the current timestamp, block id, remaining gas, current balance, and this contract's own address. Note that some of these values should be thought of as constants (e.g. current timestamp, block id), while others are updated in tandem with the execution of a transaction (e.g. remaining gas, current balance).

4.2 Recommendations for Better Semantics

We propose improvements to the operational semantics of Ethereum to fix the security problems discussed in Section 3. To deploy these proposals, all clients in the Ethereum network must upgrade.

4.2.1 Guarded Transactions (for TOD)

Recall that a TOD contract is vulnerable because users are uncertain about which state the contract will be in when their transactions is executed. This seems inevitable because miners can set arbitrary order between transactions (rule PROPOSE). To remove the TOD problem, we need to guarantee that an invocation of a contract code either returns the expected output or fails, even given the inherent non-deterministic order between selected transactions.

$$\text{TX-STALE}\ \frac{T \equiv \langle g, \bullet, \bullet, \bullet \rangle \qquad \sigma \not\vdash g}{\sigma \xrightarrow{T} \sigma}$$

$$\text{TX-SUCCESS}\ \frac{\begin{array}{c} T \equiv \langle g, id, v, l \rangle \qquad M \leftarrow \texttt{Lookup}(\sigma, id) \\ \sigma \vdash g \qquad \sigma' \leftarrow \sigma[id][bal \mapsto (\sigma[id][bal] + v)] \\ \langle \langle M, 0, l, \epsilon \rangle \cdot \epsilon, \sigma' \rangle \rightsquigarrow^* \langle \epsilon, \sigma'' \rangle \end{array}}{\sigma \xrightarrow{T} \sigma''}$$

$$\text{TX-EXCEPTION}\ \frac{\begin{array}{c} T \equiv \langle g, id, v, l \rangle \qquad M \leftarrow \texttt{Lookup}(\sigma, id) \\ \sigma \vdash g \qquad \sigma' \leftarrow \sigma[id][bal \mapsto (\sigma[id][bal] + v)] \\ \langle \langle M, 0, l, \epsilon \rangle \cdot \epsilon, \sigma' \rangle \rightsquigarrow^* \langle \langle e \rangle_{exc} \cdot \epsilon, \bullet \rangle \end{array}}{\sigma \xrightarrow{T} \sigma}$$

Figure 9: New Rules for Transaction Execution.

Guard Condition. Our new rules for transaction execution are given in Figure 9. A transaction T now additionally specifies a guard condition g; the current state σ needs to satisfy g for the execution of T to go through. If g is not satisfied, the transaction is simply dropped by the new rule TX-STALE. For transactions which do not provide g, we simply consider $g \equiv true$. This solution guarantees that either the sender gets the expected output or the transaction fails. The solution is also *backward-compatible* because we do not require changes in existing contract code: old transactions can simply take the default guard condition *true*.

To illustrate, let us revisit the Puzzle contract in Section 3.1. A user who submits a transaction T_u to claim the reward should specify the condition $g \equiv$ (`reward` == `R`), where `R` is the current reward stored in the contract. If the owner's transaction is executed first, g is invalidated and the user's transaction T_u will abort, meaning the owner will not get the solution[4]. Note that Puzzle is only one example of a more serious class of contracts serving as decentralized exchanges or market places (see Section 3.1). With our solution, buyers can easily avoid paying a price much higher than what they observe when issuing the buy orders.

Note that "guarded transactions" resemble the "compare-and-swap" (CAS) instruction supported by most modern processors. CAS is a key standard multithreaded synchronization primitive, and "guarded transactions" equips Ethereum with equivalent power.

4.2.2 Deterministic Timestamp

Allowing contracts to access to the block timestamp is essentially a *redundant* feature that renders contracts vulnerable to manipulation by adversaries. Typically, block timestamp is used for two purposes: serving as a deterministic random seed (*e.g.*, in theRun contract) and as a global timestamp in a distributed network (in [23, 24, 26]). Using block timestamp as a random seed is not wise since the entropy is low and the timestamp is easy to manipulate. There are ways to obtain better random seeds on the blockchain [27, 28].

Rather than using the easily-manipulable timestamp, contracts should use the block index—a new block is created approximately every 12 seconds in Ethereum—to model global time. The block index always increments (by one), removing any flexibility for an attacker to bias the output of contract executions.

A practical fix is to translate existing notions of timestamp into block numbers. The change can be implemented by returning the block id for the instruction `TIMESTAMP` and translating the associated expressions. For example, the condition timestamp - lastTime > 24 hours can be rewritten as blockNumber - lastBlock > 7,200. This implementation requires changes to only the `getstate` instruction from in Section 4.1.

4.2.3 Better Exception Handling

A straightforward solution is to check the return value whenever a contract calls another. Currently, the Solidity compiler inserts a code snippet to perform exception forwarding, except when the call is made via send or call, which are considered low-level instructions (from Solidity point of view). This half-way solution still leaves the "atomicity property" broken.

A better solution is to *automatically* propagate the exception at the level of EVM from callee to caller; this can be easily implemented but requires all clients to upgrade. We can *additionally* provide a mechanism for proper exception handling, *e.g.*, by having explicit `throw` and `catch` EVM instructions. If an exception is (implicitly or explicitly) thrown in the callee and not properly handled, the state of the caller can be reverted. This approach and has been used in many popular programming languages including C++, Java and

[4]Owners can read the solution from the transaction data if it is in plain-text, but solutions to this are well-studied [6, 14, 25].

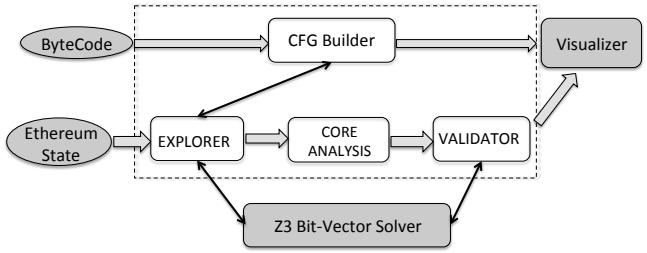

Figure 10: Overview Architecture of Oyente. Main components are within the dotted area. Shaded boxes are publicly available.

Python. Note that adding `throw` and `catch` instructions does not help when the contract owner/writer is malicious and deliberately plants a bug in the contract.

5. THE Oyente TOOL

Our solutions proposed in the previous Section do require all clients in the network to up-grade, thus running the risk of not seeing real deployment. As pre-deployment mitigation, we provide a tool, called Oyente, to help: (1) developers to write *better* contracts; and (2) users to avoid invoking *problematic* contracts. Importantly, *other* analyses can also be implemented as independent plugins, without interfering with our existing features. E.g., a straightforward extension of Oyente is to compute more precise estimations of worst-case gas consumption for contracts.

Our analysis tool is based upon symbolic execution [29]. Symbolic execution represents the values of program variables as symbolic expressions of the input symbolic values. Each symbolic path has a path condition which is a formula over the symbolic inputs built by accumulating constraints which those inputs must satisfy in order for execution to follow that path. A path is *infeasible* if its path condition is unsatisfiable. Otherwise, the path is *feasible*.

We choose symbolic execution because it can *statically* reason about a program path-by-path. On one hand, this is superior to *dynamic* testing, which reasons about a program input-by-input. For Ethereum, dynamic testing would even require much more effort to simulate the execution environment. As an example, to detect the transaction-ordering dependence, we must compare the outcomes of the interleaving of different execution paths. It is almost impossible to approach this problem with dynamic testing, given the non-determinism and complexity of the blockchain behaviors.

On the other hand, by reasoning about one path at a time, symbolic execution can achieve better precision (or less false positives) compared to traditional approaches using static taint analysis or "Abstract Interpretation" [30]. In Abstract Interpretation, abstract program states are often merged, admitting states that never happen in a real execution, and eventually lead to high false positives.

5.1 Design Overview

Figure 10 depicts the overview architecture of Oyente. It takes two inputs including bytecode of a contract to be analyzed and the current Ethereum global state. It answers whether the contract has any security problems (*e.g.*, TOD, timestamp-dependence, mishandled exceptions), outputting "problematic" symbolic paths to the users. One by-product of our tool is the Control Flow Graph (CFG) of the contract bytecode. We plan that in the future Oyente will be able to work as an *interactive debugger*, thus we feed the CFG and the problematic paths into a Graph Visualizer.

The bytecode is publicly available on the blockchain and Oyente interprets EVM instruction set to faithfully maps instructions to constraints, *i.e.*, bit-level accuracy. The Ethereum global state provides the initialized (or current) values of contract variables, thus enabling more precise analysis. All other variables including value, data of message call are treated as input symbolic values.

Oyente follows a modular design. It consists of four main components, namely CFGBuilder, Explorer, CoreAnalysis and Validator. CFGBuilder constructs a Control Flow Graph of the contract, where nodes are basic execution blocks, and edges represent execution jumps between the blocks. Explorer is our main module which symbolically executes the contract. The output of Explorer is then fed to the CoreAnalysis where we implement our logic to target the vulnerabilities identified in Section 3. Finally, Validator filters out some false positives before reporting to the users.

5.2 Implementation

We implement Oyente in Python with roughly 4,000 lines of code. Currently, we employ Z3 [31] as our solver to decide *satisfiability*. Oyente faithfully simulates Ethereum Virtual Machine (EVM) code which has 64 *distinct* instructions in its language. Oyente is able to detect all the three security problems discussed in Section 3. We describe each component below.

CFG Builder. CFGBuilder builds a *skeletal* control flow graph which contains all the basic blocks as nodes, and some edges representing jumps of which the targets can be determined by locally investigating the corresponding source nodes. However, some edges cannot be determined statically at this phase, thus they are constructed on the fly during symbolic execution in the later phase.

Explorer. Our Explorer starts with the entry node of the skeletal CFG. At any one time, Explorer may be executing a number of symbolic states. The core of Explorer is an interpreter loop which gets a state to run and then symbolically executes a single instruction in the context of that state. This loop continues until there are no states remaining, or a user-defined timeout is reached.

A conditional jump (JUMPI) takes a boolean expression (branch condition) and alters the *program counter* of the state based on whether the condition is *true* or *false*. Explorer queries Z3 to determine if the branch condition is either provably true or provably false along the current path; if so, the program counter is updated to the appropriate target address. Otherwise, both branches are possible: we then explore both paths in Depth First Search manner, updating the program counter and path condition for each path appropriately. (Note that edges might be added to the skeletal CFG.)

At the end of the exploration phase, we produce a set of symbolic traces. Each trace is associated with a path constraint and auxiliary data that the analyses in later phase require. The employment of a constraint solver, Z3 in particular, helps us eliminate *provably* infeasible traces from consideration.

Core Analysis. CoreAnalysis contains analysis sub-components to detect contracts which are TOD, timestamp-dependent or mishandled exceptions. Currently Explorer collects only

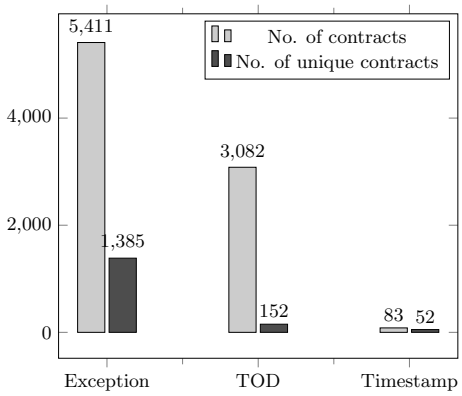

Figure 11: Number of buggy contracts per each security problem reported by OYENTE.

paths which exhibit *distinct* flows of Ether. Thus, we detect if a contract is TOD if it sends out Ether differently when the order of transactions changes. Similarly, we check if a contract is timestamp-dependent if the condition to send includes the block timestamp. We describe how we implement our analyses as below.

- *TOD detection.* Explorer returns a set of traces and the corresponding Ether flow for each trace. Our analysis thus checks if two different traces have different Ether flows. If a contract has such pairs of traces, OYENTE reports it as a TOD contract.
- *Timestamp dependence detection.* We use a special symbolic variable to represent the block timestamp. Note that the block timestamp stays constant during the execution. Thus, given a path condition of a trace, we check if this symbolic variable is included. A contract is flagged as timestamp-dependent if any of its traces depends on this symbolic variable.
- *Mishandled exceptions.* Detecting a mishandled exception is straightforward. Recall that if a callee yields an exception, it pushes 0 to the caller's operand stack. Thus we only need to check if the contract executes the ISZERO instruction (which checks if the top value of the stack is 0) after every call. If it does not, any exception occurred in the callee is ignored. Thus, we flags such contract as a contract that mishandles exceptions.

Validation. The last component is Validator which attempts to remove false positives. For instance, given a contract flagged as TOD by CoreAnalysis and its two traces t_1 and t_2 exhibiting different Ether flows, Validator queries Z3 to check if both ordering (t_1, t_2) and (t_2, t_1) are feasible. If no such t_1 and t_2 exist, the case is considered as a false positive. However, because we have not fully simulated the execution environment of Ethereum, Validator is far from being complete. For the results presented in Section 6, we resort to manual analysis to confirm the security bugs. In other words, the current main usage of OYENTE is to flag potentially vulnerable contracts; full-fledged false positive detection is left for future work.

6. EVALUATION

In this Section, we measure the efficacy of OYENTE via quantitative and qualitative analyses. We run OYENTE on all contracts in the first 1,459,999 blocks of Ethereum. Our goals are threefold. First, we aim to measure the prevalence of the security bugs discussed in Section 3 in the real Ethereum contracts. Second, we highlight that our design and implementation choices are driven by the characteristics of real-life smart contracts, and that OYENTE is robust enough to handle them. Lastly, we present several case studies demonstrating misunderstandings that many contract developers have about the subtle semantics of Ethereum.

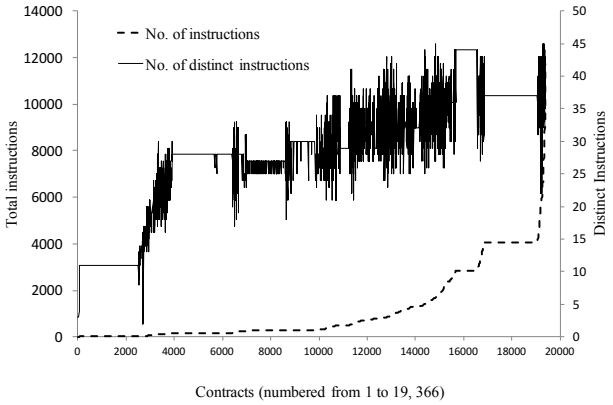

Figure 12: Number of instructions in each contract

6.1 Benchmarks and Tool Robustness

We collected 19,366 smart contracts from the blockchain as of May 5, 2016. These contracts currently hold a total balance of 6,169,802 Ether, or 62 Million US dollars at the time of writing. The balance in contracts vary significantly: most of contracts do not hold any Ether (*e.g.*, balance is zero), 10% of them have at least 1 Ether, while the highest balance (2,401,557 Ether) accounts for 38.9% of the total balance in all contracts. On an average, a contract has 318.5 Ether, or equivalently 4523 US dollars. This suggests that attackers are well-incentivized to target and exploit flaws in smart contracts to gain profit.

Ethereum contracts vary from being simple to fairly complex. Figure 12 shows that the number of instructions in a contract ranges from 18 to 23,609, with an average of 2,505 and a median of 838. The number of distinct instructions used in a single contract is shown in Figure 12. It shows that to handle these real-world contracts, OYENTE needs to correctly handle the logic of 63 instructions. We choose to build OYENTE on EVM bytecode rather than the source code (*e.g.*, Solidity [16]) because only 184 out of 19,366 contracts have source code publicly available on public repositories [32,33]. OYENTE finds a total number of 366,213 feasible execution paths which took a total analysis time of roughly 3,000 hours on Amazon EC2.

6.2 Quantitative analysis

Experimental setup. We run OYENTE on 19,366 contracts in our benchmark. All experiments are conducted on 4 Amazon EC2 `m4.10xlarge` instances, each has 40 Amazon vCPU with 160 GB of memory and runs 64-bit Ubuntu 14.04. We use Z3 v4.4.1 as our constraint solver [31]. We set a timeout for our symbolic execution (*e.g.*,Explorer component) of 30 mins per contract. The timeout for each Z3 request is set to 1 second.

Performance. On average, OYENTE takes 350 seconds to analyze a contract. 267 contracts require more than 30 minutes to analyze. The number of paths explored by OYENTE

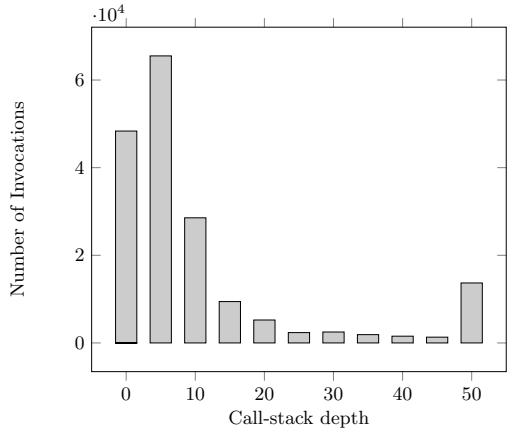

Figure 13: Call-stack depth statistics in 180,394 contract invocations in Ethereum network

ranges from 1 to 4613 with an average of 19 per contract and a median of 6. We observe that the running time depends near linearly on the number of explored paths, *i.e.*, the complexity of contracts.

6.2.1 Results

Figure 11 reports our results. OYENTE flags 8,519 contracts which have at least one security issue discussed in Section 3. Out of these, 1,556 are distinct (by direct comparison of the bytecode). Of these, we were able to collect source code for only 155 contracts to confirm the tool's correctness; we manually check for false positives. Among all contracts with source, OYENTE has a low false positive rate of 5.8%, *i.e.*, only 9 cases out of 155.

Mishandled exceptions. 5,411 contracts have mishandled exceptions flagged by OYENTE, which account for 27.9% of the contracts in our benchmark. Out of these 5,411 contracts, 1,385 are found to be distinct and 116 contracts have source code available. By manual analysis, we verify that all these 116 contracts are true positives. In order to confirm, we identify if any external calls (SEND, CALL instructions) are present and not followed by any checks for failures. These failure checks are implemented by verifying if the return value is non-zero.

The prevalence of this problem is explained by the following observation. In the first 1,459,999 blocks on the public blockchain, 180,394 cross-contract calls were processed. For each contract invocation, there may be additional calls to other contracts, thus increasing the call-stack depth. These can be due to function or library calls, external account transactions, or nested recursive contract calls. We plot the call-stack depths of these contract invocations in Figure 13, which shows that most of them involve some level of nesting (*e.g.*, invoking other contracts). Further, all contract invocations do not exceed the call-stack depth of 50 in benign runs, which is far below the call-stack depth's limit of 1,024. This explains why exceptions are commonly unexpected and unhandled in benign invocations.

Transaction-ordering dependence. The TOD contracts are less common with 3,082 contracts, or 15.9% of the contracts in our benchmarks. Of these contracts, there are 152 distinct contracts and 32 have source code available. We manually verify that 9 of them are false positives and 23 are true positives. In order to confirm, we look for different flows of Ether wherein the outcome depends on the order of the input transactions.

```
1 function enter(){
2     uint amount = msg.value;
3     if ((amount < contribution)...) {
4         msg.sender.send(msg.value);
5         return;
6     }
7     addParticipant(msg.sender, inviter);
8     address next = inviter;
9     uint rest = amount;
10    uint level = 1;
11    while ( (next != top) && (level < 7) ){
12        uint toSend = rest/2;
13        next.send(toSend);
14        /.../
15    }
16    next.send(rest);
17    Tree[next].totalPayout += rest;
18 }
```

Figure 14: A false positive TOD contract flagged by OYENTE. This contract implements a typical ponzi scheme.

Several true positive cases, where this dependence is exploitable, are discussed later. As an example of a false positive, Figure 14 shows a case where there are two separate flows of Ether, but the order of their execution does not change the outcome of the contract. The first flow (send in Line 4) returns Ether to the sender if some requirement is not met, while the latter (Line 7–17) pays out to previous participants. OYENTE recognized the two flows of Ether and flagged the contract as potentially buggy. However, the order in which these flows are executed by incoming transactions does not affect the intended payouts and recipients therein. The tool could involve more time-consuming analyses to resolve such cases in the future. The final state of the contract remains the same.

Timestamp dependence. OYENTE reports 83 timestamp-dependent contracts in our benchmark, out of which 53 are distinct. Only 7 of these have source code available, which we manually verified for false positives. To confirm, we check if timestamp is included in the path condition of a flow of Ether, such that manipulation of the block timestamp would result in a different payout or recipient from the contract.

6.3 Qualitative Analysis

We investigate several contracts flagged by OYENTE to show how it helps analyze Ethereum smart contracts.

6.3.1 Severity of Attacks

We have found vulnerable contracts which have different levels of damage severity reported in Section 6.2. For instance, the PonziGovernMental [24] contract with highest balance (1,099.5 Ether) flagged by OYENTE among the mishandled exception category. The contract operates as below.

- The contract accepts investments from users. A new investment pays to previous investors and adds to the jackpot.
- After 12 hours without no new investment, the last investor and the contract owner shares the jackpot.

The code to handle the last step is in Figure 16. Line 7 and 9 use send instructions to send Ether, which may not execute correctly as discussed in Section 3.3. The contract does not check if the operations were successful, leaving it vulnerable to attack.

Stealing Ether from investors. There is reason to believe this may be a back-door masquerading as a bug. If we consider the code snippet from Figure 16, Line 11 seems

```
1 contract Foo{
2   Foo public self;
3   EtherId public etherid;
4   uint256 public domain;
5   /../
6   function callself(int counter){
7     if (counter < 1023){
8       if (counter > 0)
9         self.callself.gas(msg.gas-2000)(counter+1);
10      else self.callself(counter+1);
11    }
12    else etherid.changeDomain.value(this.balance)\
13      (domain, 10000000, 1 ether/100, 0);
14  }}
```

Figure 15: A contract conducts the call-stack attack by calling itself 1023 times before sending a buy request to EtherID.

harmless in that it pays out the owners after all creditors have been paid, with all remaining amount. However, if the owners were to call the contract with a carefully constructed call-stack size of 1023, none of the send instructions would succeed, resulting in no Ether going out. A second call to the contract would result in the owners receiving the entirety of the contract's balance and forcing previous investors to lose all Ether invested in the contract. Similar attacks exist in other contracts as well (see [20, 23]).

Manipulation of contract outcome. PonziGovernMental is another example of timestamp-dependent contract. In Line 5, PonziGovernMental determines the current time by using the current block timestamp. If a user invokes PonziGovernMental when it is close to 12 hours since the last deposit, the miner can set the next block timestamp to make the condition in Line 5 either valid or invalid. Thus, miners can force the PonziGovernMental contract to finish the current round earlier by picking a timestamp value which is ahead of the current time. Alternatively, miners can extend the round for 12 hours by choosing a smaller timestamp. Thus, miners who have a stake in the contract will set the timestamp to a value which favors their outcome. The problem also exists in other contracts (see [26]).

Lock or Sabotage Others' Funds. Attackers can also prevent others from receiving their legitimate payments. One example of such an attack is in EtherID, which is one of the most active contracts in Ethereum with 57,738 received/sent transactions as of writing [34]. EtherID works as a name registrar for Ethereum network to allow users to create, buy and sell any ID (like a token). The code which handles users' requests to buy a registered ID is in Figure 17. The code checks if the buyer has enough Ethers (Line 2), sends the payment to the existing owner (Line 4) and change the ownership of the ID (Line 5–10). As described in Section 3.3, the send instruction in Line 4 may fail. As a result, the ID owners may not receive the payment and still have to transfer the ownership of their ID to the buyers. There is no way for the owners to claim the payment later on. The Ether value is locked in the contract forever.

6.4 Public Verification

Verifying the above attack on the public blockchain is feasible, but for ethical reasons we do not conduct our attack confirmation on contracts [20, 23, 24] where users may lose funds. Instead, we perform our verification on EtherID contract on which the attack has less severity. More importantly, EtherID allows us to target our own accounts, other accounts are not affected in the experiments.

We verify the problem of EtherID by creating our own IDs

```
1 function lendGovernmentMoney(address buddy)
2     returns (bool) {
3     uint amount = msg.value;
4     // check the condition to end the game
5     if (lastTimeOfNewCredit + TWELVE_HOURS >
6         block.timestamp) {
7         msg.sender.send(amount);
8         // Sends jacpot to the last creditor
9         creditorAddresses[creditorAddresses.length - 1]
10            .send(profitFromCrash);
11        owner.send(this.balance);
12
13        // Reset contract state
14        lastCreditorPayedOut = 0;
15        lastTimeOfNewCredit = block.timestamp;
16        profitFromCrash = 0;
17        creditorAddresses = new address[](0);
18        creditorAmounts = new uint[](0);
19        round += 1;
20        return false;
21    }}
```

Figure 16: PonziGovernmentMental contract, with over 1000 Ether, allows users to participate/profit from the creation/fall of a government.

and self-purchasing them. We show that the registers of our IDs do not receive the intended payments when the registers use contract wallets, or when the buyers are malicious and conduct the call-stack exceeding attack.

Our two IDs are dummywallet and foowallet registered by two addresses 0x33dc532ec9b61ee7d8adf558ff248542c2a2a62e and 0x62ec11a7fb5e35bd9e243eb7f867a303e0dfe08b respectively. The price to buy either of the ID is 0.01 Ether. The address 0x33dc532... is a contract address, which performs some computation (thus burning gas) on receiving any payment.

We then send two transactions from different addresses to buy the two IDs. The first transaction [5] purchases dummywallet. However, 0x33dc532... is a contract address, which is implemented to burn all the provided gas on receiving any payment without doing anything else. Thus the send function in Line 4 of EtherID to 0x33dc532... will fail. As a result, 0x33dc532... sells it ID without receiving any payment. The fund 0.01 Ethers is kept in the contract EtherID forever.

The second transaction [6] sent from a contract, which calls itself 1023 times before sending a buy request to EtherID to buy foowallet. The code snippet to perform such attack is in Figure 15. When EtherID executes send in Line 4, the call stack already has 1024 frames, so send fails regardless of how much gas is used. Hence, the address 0x62ec11a7... does not receive the payment of 0.01 Ethers as it should.

7. RELATED WORK

Smart Contract Security. Delmolino et al. [14] show that even a simple self-construct contract (*e.g.*, "Rock, Paper, Scissors") can contain several logic problems, including:

- *Contracts do not refund.* Some contracts proceed further only if users send a certain amount of Ether. However, these contracts sometimes "forget" to refund users if users send less than what is required.
- *Lack of cryptography to achieve fairness.* Some contracts perform computation based on users' inputs to decide the outcome (*e.g.*, rolling a die). However, those contracts store users' input in plaintext on the blockchain. Accord-

[5] TX hash: 0xb169b07c274a71727ecfe9d0610d09917c454c1216cd659350f83ef44ba071b4

[6] TX hash: 0x0c10fafe0cdbfff32abfe53d57ec861d09986cc1050c850481f79b1a862bb10a

```
1// ID on sale, and enough money
2if(d.price > 0 &&  msg.value >= d.price){
3    if(d.price > 0)
4        address(d.owner).send(d.price);
5    d.owner = msg.sender;// Change the ownership
6    d.price = price;     // New price
7    d.transfer = transfer;  // New transfer
8    d.expires = block.number + expires;
9    DomainChanged( msg.sender, domain, 0 );
10}
```

Figure 17: EtherID contract, which allows users to register, buy and sell any ID. This code snippet handles buy requests from users.

ingly, malicious users can submit inputs biased in their favor.

- *Incentive misalignment.* Some contracts do not incentivize users to follow intended behavior. Consider a gambling game that uses a commit-reveal scheme in which participants first submit their encrypted move along with a deposit before later revealing it. After the first move is revealed, the second user may realize his move will lose. Since his deposit lost, he may not be willing to spend gas to reveal his choice.

These security problems are more about logical flaws in the implementation of contracts. In contrast, our paper documents new security bugs stemming from semantic misunderstandings of smart contract developers. We suggest improvements to the semantics and introduce Oyente to detect these bugs in existing contracts in the Ethereum blockchain. Our evaluation showed that 8,519 existing contracts contain at least one of the new bugs. The call-stack problem of Ethereum was reported previously in a security audit by Miller *et al.* [22]. The bug, however, still remains unfixed.

Other work also studies security and/or privacy concerns in designing smart contracts [6, 25, 35, 36]. For instance, Hawk [25] provides confidential execution for contracts by leveraging cryptographic techniques and Town Crier [35] feeds reliable, trustworthy data from trusted web servers to smart contracts via hardware rooted trust.

Distributed Systems and Programming Languages. Security problems in smart contracts are often related to problems in traditional distributed systems. For example, concurrency control in multiuser distributed database systems (DBMS) [37] is superficially similar to the transaction-order dependency problem. However, transaction-ordering problems in permissionless distributed systems like cryptocurrencies are more complex than in traditional systems because adversaries can manipulate the order.

Many previous works attempt to build a global timestamp in distributed systems, in both asynchronous and synchronous settings [38–40]. Time in distributed systems traditionally forms a partial order rather than the the total order given by the blockchain. As we discussed in Section 3, having access to the block timestamp (in addition to the block id) is redundant and invites attack. Lastly, propagating exceptions is inspired by the exception handling mechanism in modern languages [41, 42].

8. ACKNOWLEDGMENT

We thank Brian Demsky, Vitalik Buterin, Yaron Welner, Gregory J. Duck, Christian Reitwiessner, Dawn Song, Andrew Miller, Jason Teutsch and Alex Zikai Wen for useful discussions and feedback on the early version of the paper. This work is supported by the Ministry of Education, Singapore under under Grant No. R-252-000-560-112. All opinions expressed in this work are solely those of the authors.

9. REFERENCES

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *bitcoin.org*, 2009.

[2] Ethereum Foundation. Ethereum's white paper. https://github.com/ethereum/wiki/wiki/White-Paper, 2014.

[3] A. Miller et al. Permacoin: Repurposing Bitcoin work for long-term data preservation. *IEEE Security and Privacy*, 2014.

[4] Use case for factom: The world's first blockchain operating system (bos). http://kencode.de/projects/ePlug/Factom-Linux-Whitepaper.pdf, Feb 2015.

[5] N. Szabo. The idea of smart contracts. http://szabo.best.vwh.net/smart_contracts_idea.html, 1997.

[6] L. Luu, J. Teutsch, R. Kulkarni, and R. Saxena. Demystifying incentives in the consensus computer. In *CCS*, pages 706–719, 2015.

[7] EtherDice smart contract is down for maintenance. https://www.reddit.com/r/ethereum/comments/47f028/etherdice_is_down_for_maintenance_we_are_having/.

[8] RSK Labs. Rootstock: Smart contracts platform powered by Bitcoin. http://www.rootstock.io/, 2015.

[9] Counterparty platform. http://counterparty.io/, 2015.

[10] J. C. Corbett et al. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, aug 2013.

[11] J. Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.

[12] KingOfTheEtherThrone smart contract. https://github.com/kieranelby/KingOfTheEtherThrone/blob/v0.4.0/contracts/KingOfTheEtherThrone.sol.

[13] GovernMental's 1100 ETH payout is stuck because it uses too much gas. https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/.

[14] D. Delmolino et al. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. Cryptology ePrint Archive, Report 2015/460, 2015. http://eprint.iacr.org/.

[15] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. http://gavwood.com/paper.pdf, 2014.

[16] Ethereum Foundation. The solidity contract-oriented programming language. https://github.com/ethereum/solidity.

[17] Ethereum Foundation. The serpent contract-oriented programming language. https://github.com/ethereum/serpent.

[18] EtherEx: A fully decentralized cryptocurrency exchange. https://etherex.org/.

[19] EtherOpt: A decentralized options exchange. http://etheropt.github.io/.

[20] The Run smart contract. https://etherscan.io/address/0xcac337492149bdb66b088bf5914bedfbf78ccc18.

[21] Ethereum Foundation. Block validation algorithm. https://github.com/ethereum/wiki/wiki/Block-Protocol-2.0#block-validation-algorithm.

[22] A. Miller, B. Warner, and N. Wilcox. Gas economics. https://github.com/LeastAuthority/ethereum-analyses/blob/master/GasEcon.md.

[23] Protect The Castle Contract. http://protect-the-castle.ether-contract.org/.

[24] GovernMental Smart Contract. http://governmental.github.io/GovernMental/.

[25] A. Kosba et al. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *SP*, 2016.

[26] Lottopolo smart contract. https://etherchain.org/account/0x0155ce35fe73249fa5d6a29f3b4b7b98732eb2ed.

[27] Random number generator contract. https://github.com/randao/randao.

[28] J. Bonneau, J. Clark, and S. Goldfeder. On Bitcoin as a public randomness source. Cryptology ePrint Archive, Report 2015/1015, 2015. http://eprint.iacr.org/.

[29] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394.

[30] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[31] Microsoft Corporation. The Z3 theorem prover. https://github.com/Z3Prover/z3.

[32] The Ethereum block explorer. https://etherscan.io/.

[33] The Ethereum network stats. https://etherchain.org/.

[34] A. Navernioukˇ. EtherID: Ethereum name registrar. http://etherid.org/.

[35] F. Zhang et al. Town crier: An authenticated data feed for smart contracts. Cryptology ePrint Archive, Report 2016/168, 2016. http://eprint.iacr.org/.

[36] A. Juels, A. Kosba, and E. Shi. The ring of Gyges: Investigating the future of criminal smart contracts. Cryptology ePrint Archive, Report 2016/358, 2016. http://eprint.iacr.org/.

[37] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[38] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226, 1989.

[39] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56âĂŞ66, 1988.

[40] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, pages 558–565, July 1978.

[41] A. Koening and B. Stroustrup. Exception handling for C++. *Journal of Object-Oriented Programming*, 3(2):16–33, 1990.

[42] R. Milner, M. Tofte, and D. MacQueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.