Making Smart Contracts Smarter

Luu, Chu, Olickel et al.

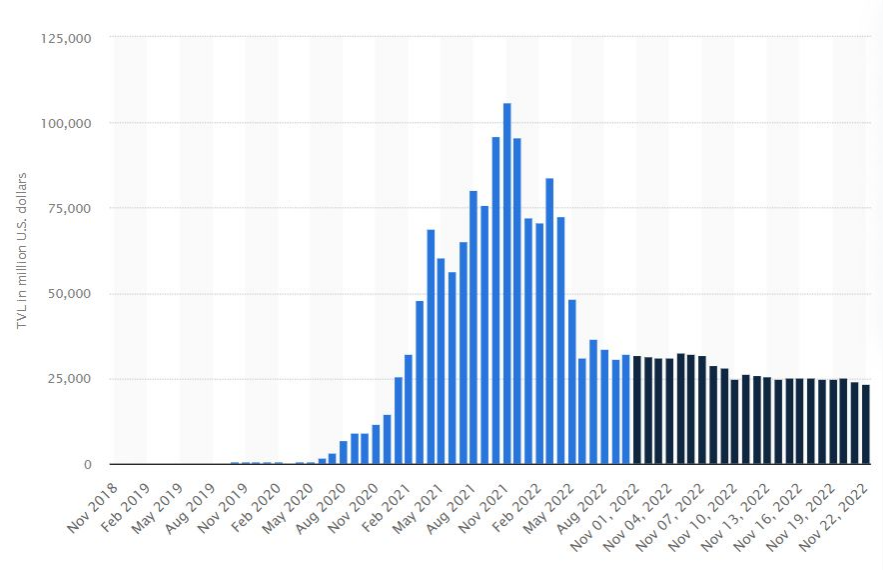
Oriel Zahavi
Noa Ziv

INTRODUCTION

INTRODUCTION

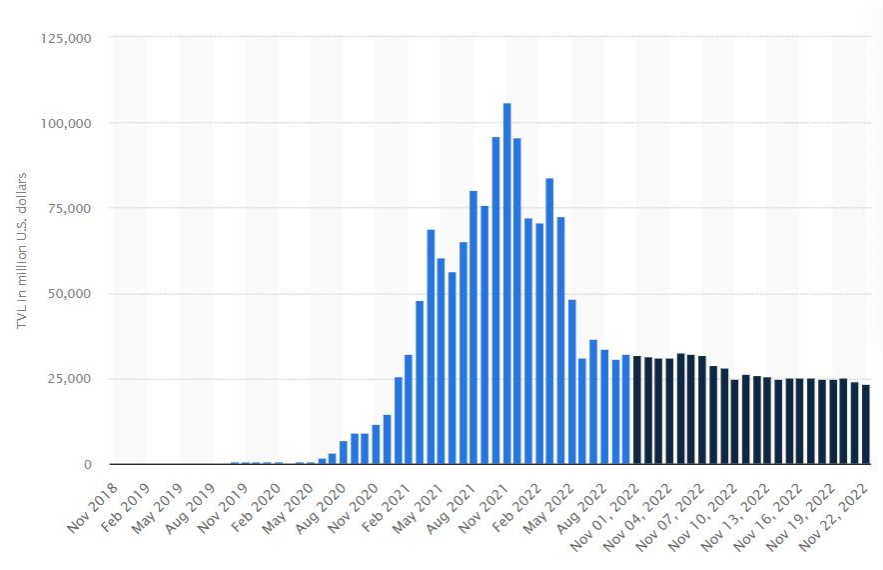
- In many cryptocurrencies today, there are smart contracts.
  We will mainly discuss the smart contracts on the Ethereum network.

- The value held in smart contracts on Ethereum today is enormous, and therefore the security of these contracts is crucial.



INTRODUCTION

- Even today, 9 years after the release of Ethereum, there are numerous smart contracts that do not take security seriously. As a result, there are still many vulnerabilities present in these smart contracts, and people may risk losing their funds.



Smart Contract

- Smart contracts are a type of program that allows for an agreement to be automatically executed without a third party.
  For example, there are smart contracts that enable loans without the need for banks

- Unlike traditional applications, any code that is uploaded to the blockchain cannot be changed. Therefore, before uploading a smart contract to the blockchain, it is important to ensure that it is safe for use.

- On the contrary to centralized financial services (such as banks), smart contracts are completely decentralized. This means that there is no entity capable of reversing a money transfer, for example, in the event of a security breach. Consequently, attackers can exploit this for their own benefit.

Ethereum

Ethereum is an open-source blockchain-based platform that enables the creation and use of distributed tokens, NFT tokens, and smart contracts.
These features facilitate the closure of transactions and agreements over the internet in a decentralized manner, meaning without a central entity managing them.

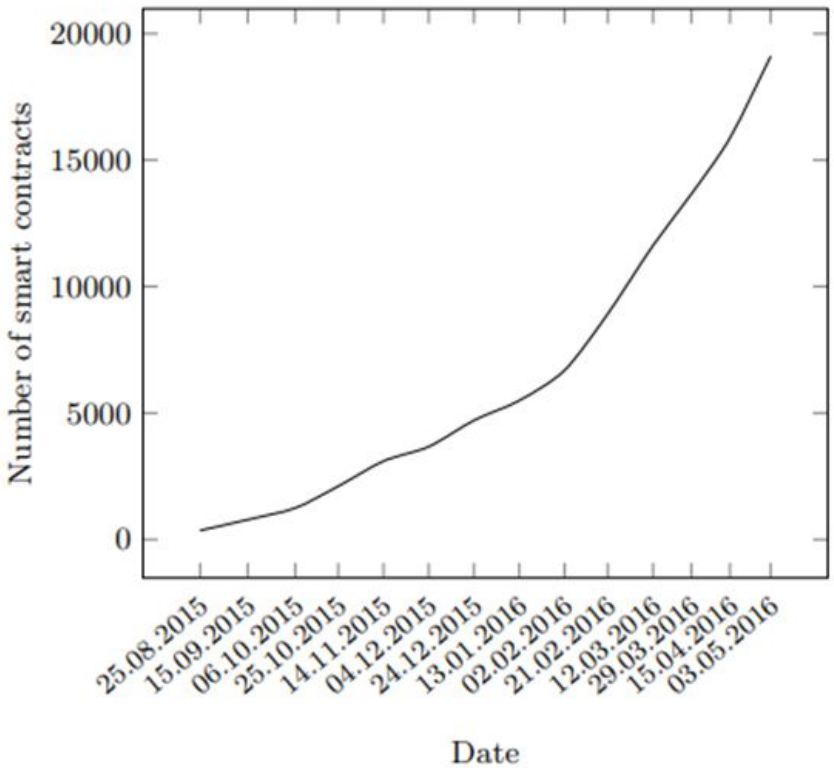


Figure 1: Number of smart contracts in Ethereum has increased rapidly.

EVM and Solidity

- Solidity is the high-level language used to write smart contracts for Ethereum
- The code of the smart contract is compiled into bytecode that runs on the EVM, in order to run the code of the smart contract.

Blockchain

- Decentralized cryptocurrencies operate on a peer-to-peer (P2P) system, without a third-party intermediary. Transaction tracking is performed through a "ledger" called the blockchain.

- We will denote the state-transition function as σ.
  For example, given an address γ, so σ(γ) represents the state of the account.

- A valid transition from σ to σ', via transaction T is denoted as: $\sigma \xrightarrow{T} \sigma'$.

Blockchain

The blockchain is a linked list of blocks and each block contains the following fields:

- **prevBlock:** Points to the previous block.
- **TimeStamp:** The time when the block was mined (created).
- **ListOftxs:** The list of transactions.
- **Blockchain State:** Maps addresses to accounts, indicating the state of each account after transaction operations.

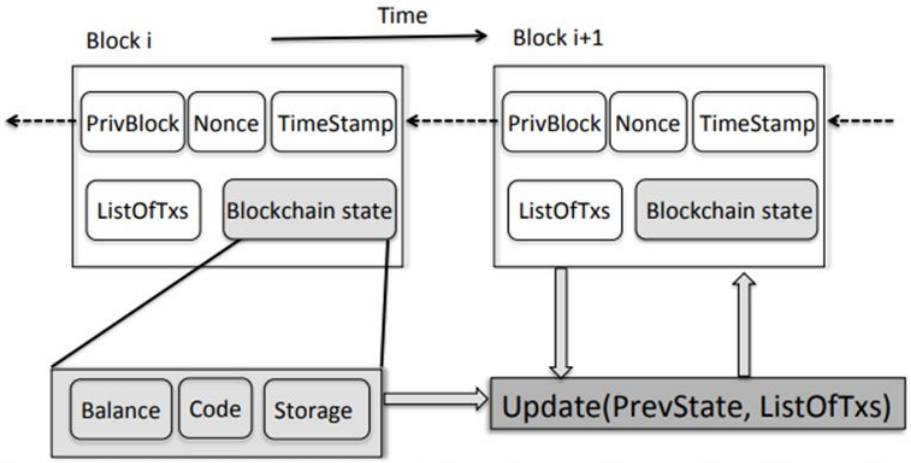


Figure 2: The blockchain's design in popular cryptocurrencies like Bitcoin and Ethereum. Each block consists of several transactions.

Blockchain

The blockchain is a linked list of blocks and each block contains the following fields:

- **Balance:** Each account can hold an Ethereum balance.
- **Code:** For addresses holding smart contracts, a significant code is stored.
- **Storage:** The space where the code is stored.

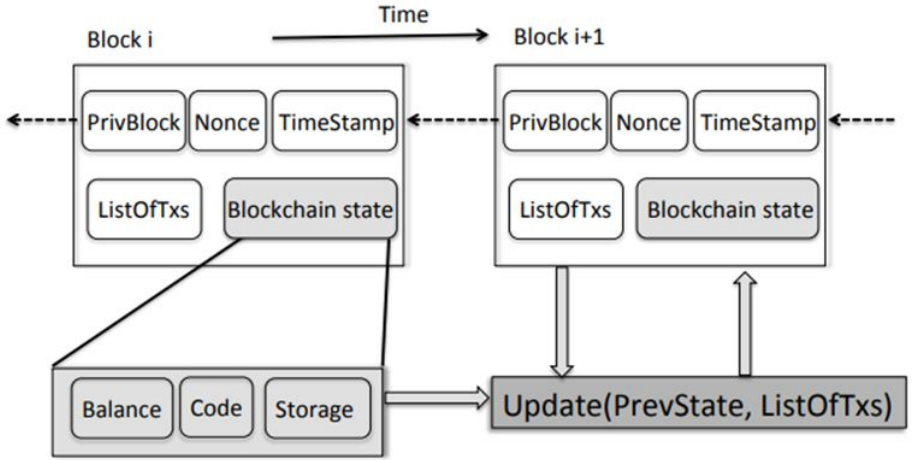


Figure 2: The blockchain's design in popular cryptocurrencies like Bitcoin and Ethereum. Each block consists of several transactions.

Blockchain

The blockchain is a linked list of blocks and each block contains the following fields:.

- **Nonce:** The nonce value ensure that transactions from the same account are processed in the correct order and are not duplicated.
  The nonce is also used as a counter to ensure that a block is unique and has not been created before.

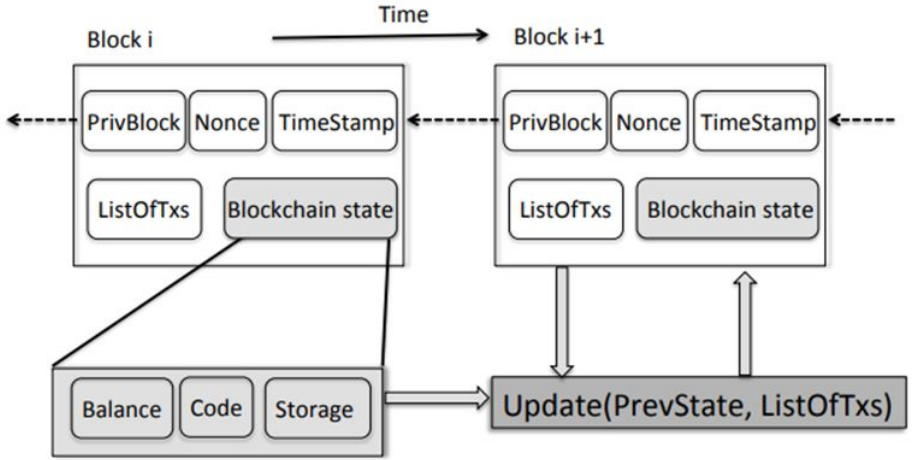


Figure 2: The blockchain's design in popular cryptocurrencies like Bitcoin and Ethereum. Each block consists of several transactions.

GAS SYSTEM

- Every operation in Ethereum has a predefined amount of gas required to execute it.
  Only for the example, the POP command requires 5 gas units.

- For each gas unit, we can specify a price called the ***gasPrice***.
  Additionally, there is a ***gasLimit***, which serves as an upper limit on the amount of gas in a transaction.
  If the gas used exceeds the gasLimit, the transaction terminates with an exception, and the blockchain state remains unchanged.

SECURITY BUGS IN CONTRACTS

TOD - Transaction-Ordering Dependence

- Let us consider a scenario where the blockchain is at state σ and the new block includes two transactions (e.g., Ti, Tj) 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

Question: What is the security issue that arises from TOD in this contract?

```
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.

Reminder

Block.timestamp value is also checked to validate the block to be added to the blockchain network. This value should be equal to or less than the actual value plus 900 seconds.

- The timestamp value should be higher or equal to the parent.timestamp value, otherwise the next block will be automatically rejected.
- Here, the nodes have the opportunity to change the timestamp in such a way that is no lower than the parent.timestamp and not higher than the actual time plus 900.

Timestamp Dependence

- The timestamp of the block can also pose a security issue when using the block timestamp as a triggering condition to execute some critical operations, e.g., sending money.
- We call such contracts as timestamp-dependent contracts.

Question: What is the security issue that arises from TOD in this contract?

```
contract TimeStamp {
    uint public prevBlockTime;

    constructor() payable {}

    function guess() external payable {
        require(msg.value == 2 ether);
        require(block.timestamp != prevBlockTime);

        prevBlockTime = block.timestamp;

        if (block.timestamp % 7 == 0) {
            (bool sent, ) = msg.sender.call{value: address(this).balance}("");
            require(sent, "Failed to send Ether");
        }
    }
}
```

Mishandled Exceptions

- The possibility of an attack in this case is when there is no consideration for handling and/or catching exceptions, meaning the calling function does not check if an exception occurred in the called function.

Question: Where is the vulnerability in the following contract?

```
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].

KOE attack example

Reentrancy Vulnerability

- A reentrancy attack is a type of vulnerability in smart contracts that allows attackers to execute a function multiple times before the previous call completes. This can lead to unexpected and harmful behavior, such as the theft of funds or unauthorized access to data.

Question: Where is the vulnerability in the following contract?

```
1 contract SendBalance {
2  mapping (address => uint) userBalances;
3  bool withdrawn = false;
4  function getBalance(address u) constant returns(uint){
5   return userBalances[u];
6  }
7  function addToBalance() {
8   userBalances[msg.sender] += msg.value;
9  }
10 function withdrawBalance(){
11  if (!(msg.sender.call.value(
12    userBalances[msg.sender])())) { throw; }
13  userBalances[msg.sender] = 0;
14  }}
```

Figure 7: An example of the reentrancy bug. The contract implements a simple bank account.

Reentrancy attack example

TOWARDS A BETTER DESIGN

TOWARDS A BETTER DESIGN

So let's start by defining symbols in semantics:

- ← - Assignment
- • - Arbitrary element
- ⇓ - Big evaluation step
- ⇝ - Small evaluation step
- a[i→v] - a is a new array where the element at index i receives the value v.
- <BC,σ> - Global Ethereum state is a pair where BC is the current blockchain and σ is the state-transition function
- Γ - The stream of incoming new transactions.

TOWARDS A BETTER DESIGN

- The Propose rule - is executed only by one selected miner who determines which transactions will be included in the next block

- The Accept rule - is executed by the other miners in the network who accept the proposed block and add it to the blockchain.

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

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

Figure 8: Proposing and Accepting a Block

TOWARDS A BETTER DESIGN

What problems arise from the semantics rules?

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

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

Figure 8: Proposing and Accepting a Block

TOWARDS A BETTER DESIGN

What problems arise from the semantics rules?

1. The selected miner determines the timestamp.
2. The selected miner decides on the order of transactions.

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

$$\frac{\begin{array}{c}\text{Receive } B \equiv \langle \texttt{blockid} \,;\, \texttt{timestamp} \,;\, \texttt{TXs} \,;\, \ldots \rangle \\ \texttt{TXs} \equiv (T_1 \ldots T_n) \\ \forall\, i, 1 \leq i \leq n : \sigma_{i-1} \xrightarrow{T_i} \sigma_i \end{array}}{\begin{array}{c}\langle BC, \sigma_0 \rangle \Downarrow \langle B \cdot BC, \sigma_n \rangle \\ \text{Remove } T_1 \ldots T_n \text{ from } \Gamma \text{ and broadcast } B \end{array}} \text{ ACCEPT}$$

Figure 8: Proposing and Accepting a Block

The activation record stack

- In Ethereum, an activation record stack is a data structure that is used to keep track of the current state of a contract during execution. It is a stack of records that contain information about the current state of the contract, including the values of its variables, the address of the contract, and the current function being executed.

- Each time a function is called within a contract, a new activation record is pushed onto the stack, containing information about the function call. When the function returns, the activation record is popped from the stack and the contract returns to the previous state before the function was called.

- The activation record stack is important because it allows the Ethereum Virtual Machine (EVM) to keep track of the current state of a contract, even as it executes complex operations involving multiple function calls. By maintaining a record of the state of the contract at each point in the execution, the EVM can ensure that the contract is executed correctly and that the results are consistent with the rules of the Ethereum network.

The activation record stack

- According to the following derivation rules, we have defined how stack operations should appear, and through them, we can represent all possible state snapshots of the stack. We can say that every program can either terminate successfully or end with an exception.
- When this quadruple represents the state of the called function.

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

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.

Rules for Transaction Execution

- We will say that a transaction has been successfully received if we transition from state σ to state σ'.
- We will say that a transaction has failed if an exception is thrown during the execution of the code, in which case we remain in state σ.
- The money is transferred to the contract as a condition at the beginning of the code's execution as MSG.VALUE, and only if the transaction completes without an exception, the money remains in the contract. If not, the money will not be received by the contract.

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

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

Figure 9: Rules for Transaction Execution. **Lookup**(σ,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 will present some of the commands that are possible in the EVM, which is the execution environment of Ethereum in the EtherLite language.

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

- **op** - denotes all arithmetic and logical operations (addition, subtraction, OR, AND, etc.).
- **bne** - pops two items from the top of the stack - if the first item is not zero, the program counter becomes the second item; otherwise, increment the program counter by 1 (i.e., proceed to the next instruction).

Execution of EVM Instructions

We will present some of the commands that are possible in the EVM, which is the execution environment of Ethereum in the EtherLite language.

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

- **mload and mstore** - instructions for memory operations.
- **sload and sstore** - instructions for contract storage operations.
- **call** - function call.
- **return** - return from a function.

Execution of EVM Instructions

We will present some of the commands that are possible in the EVM, which is the execution environment of Ethereum in the EtherLite language.

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

- **suicide** - Transfer of all funds to user γ and then terminate the contract (storage is released, and the contract becomes inactive and unusable), without using the stack operation "call".

Execution of EVM Instructions

We will present some of the commands that are possible in the EVM, which is the execution environment of Ethereum in the EtherLite language.

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

- **create** - creation of a new smart contract, receiving three arguments: the initial amount with which the contract starts, the address of the smart contract code, and the memory to allocate to the code.
- **getstate** - pushes the timestamp of the last block and several other insignificant variables to the stack.

Recommendations for Better Semantics

1. **Guarded Transactions (for TOD)**

   - The article's authors propose an improvement to the semantics of accepting or rejecting a transaction for TOD.

   - Recall that the TOD contract is a vulnerable contract since users are unaware of the contract's state when their transaction is executed.
     Therefore, we need to ensure that users' transactions are not affected by TOD.

   - To achieve this, the caller of the transaction will add a condition. If the condition is not met, the transaction will not be executed.

Recommendations for Better Semantics

1. **Guarded Transactions (for TOD)**

- the first rule states that if the condition is not met, the blockchain state remains unchanged.
- We can see that the other two rules are very similar to what we saw previously only that the difference here is that the state function is updated according to the additional condition.

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

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

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

Figure 10: New Rules for Transaction Execution.

- In the puzzle example, we can now demand by using a **guard condition** that if the prize is as we see it when we check the contract (before the transaction is executed), we compare it to that value before the transaction. If the condition is met, then the transaction will indeed be executed.

- User who submits a transaction Tu to claim the reward should specify the condition g ≡ (reward == R), where R is the current reward stored in the contract

```
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.

Recommendations for Better Semantics

2. Deterministic Timestamp

We will recall that a timestamp has two purposes:

1. To serve as a random seed
2. To serve as a global timestamp signature.

Recommendations for Better Semantics

2. Deterministic Timestamp

Now let's see an example for when the timestamp is served as a global timestamp signature.

If we know that a block in Ethereum is generated on average every 12 seconds, what is the issue with the following condition:
timestamp - lastTime > 24 hours?

Recommendations for Better Semantics

2. Deterministic Timestamp

If we know that a block in Ethereum is generated on average every 12 seconds, what is the issue with the following condition:
timestamp - lastTime > 24 hours?

So if we recall the vulnerabilities associated with using a timestamp, we remember that the timestamp can be manipulated by the caller. Therefore, a better and equivalent condition would be: blockNumber - lastBlock > 7,200.

Since the block number increases by one each time, if someone wants to check if 24 hours have passed, they would need to check that 7,200 blocks have passed, which, if calculated, equals: 7200 * (12/60) / 60 = 24.

Recommendations for Better Semantics

3. Better Exception Handling

- If the send and call commands fail for any reason, no error is thrown. Therefore, if the contract writer does not check the return value of these calls, atomicity is not achieved.

- One possible solution is to add throw and catch commands to the EVM language. This ensures that if an error occurs, it propagates upward until it is caught or the transaction fails. In this way, we maintain atomicity.

THE *Oyente* TOOL

Oyente

- In fact, the purpose of the article is to publish the researchers' solution, which is a testing tool called "***Oyente***." All the solutions presented so far require an update of the Ethereum network.

- Therefore, an external testing tool is proposed that does not require an update of the network.

- Its goal is to assist in two ways: to write safer contracts and to prevent users from interacting with problematic (malicious) contracts.
- We will see an example of its usage further on.

Oyente

- The tool is based on symbolic execution, which is a technique aimed at describing all the possible paths that a program can reach and checking the program's correctness and the presence of bugs.

- For example, in the case of TOD, we would need to compare the results of different paths, and if we observe a contradiction between two different paths, we can conclude that the contract is problematic in terms of TOD.

Let's discuss the design and algorithm of the tool:

- Oyente takes the bytecode of the contract and its blockchain state as input and determines whether there are any issues in the contract.

- **CFG (Control Flow Graph) Builder -**
  CFG constructs a graph representing the code (in bytecode) and captures all the possible program paths.

  The nodes represent the commands in the program, and each edge between two nodes represents a possible execution from command A to command B.

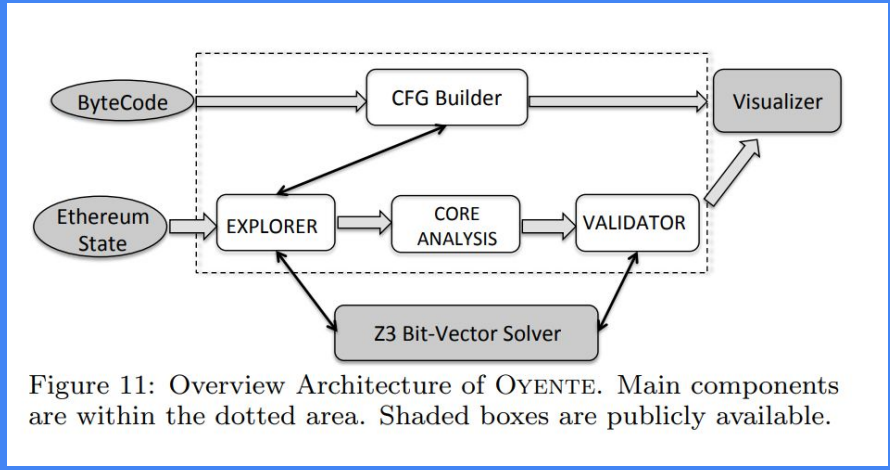


Figure 11: Overview Architecture of OYENTE. Main components are within the dotted area. Shaded boxes are publicly available.

- **EXPLORER -**
  The explorer starts from the initial node and generates all possible paths that can be traversed from the graph. It runs in parallel all the possible transitions from each state.

  When encountering a conditional jump (JUMPI) in the program, Explorer queries Z3 to check if the branch condition can be determined as either provably true or provably false along the current path.

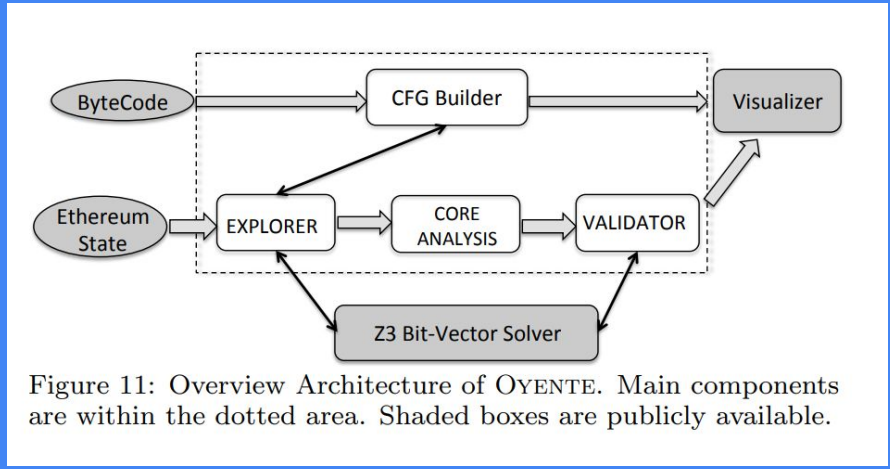


Figure 11: Overview Architecture of OYENTE. Main components are within the dotted area. Shaded boxes are publicly available.

- If the branch condition is true/false - explorer will update the PC accordingly.
  Else, Explorer employs a Depth First Search (DFS) approach to explore both paths. It updates the program counter and path condition for each path accordingly.

- By using Z3 in this manner, Explorer can eliminate provably infeasible traces during the exploration phase

- The output of the explorer presents, to the next component, all the unique paths within the contract and from the contract outward

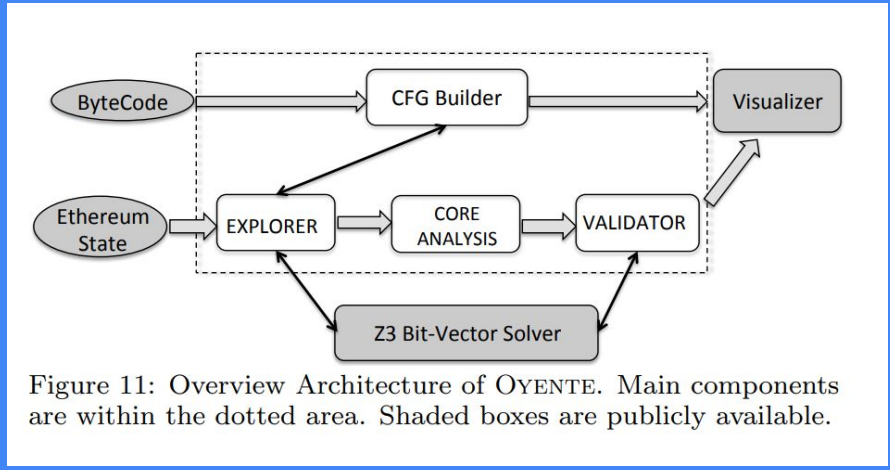


Figure 11: Overview Architecture of OYENTE. Main components are within the dotted area. Shaded boxes are publicly available.

- **CORE ANALYSIS -**
  It identifies the software behavior for many different scenarios and inputs. Ultimately, the output of the explorer

- for example, for TOD, if the contract produces different results for the same transactions in a different order, it indicates a TOD issue.

- Another example is that we can determine if the contract is timestamp-dependent if the checked condition includes the timestamp of the block.

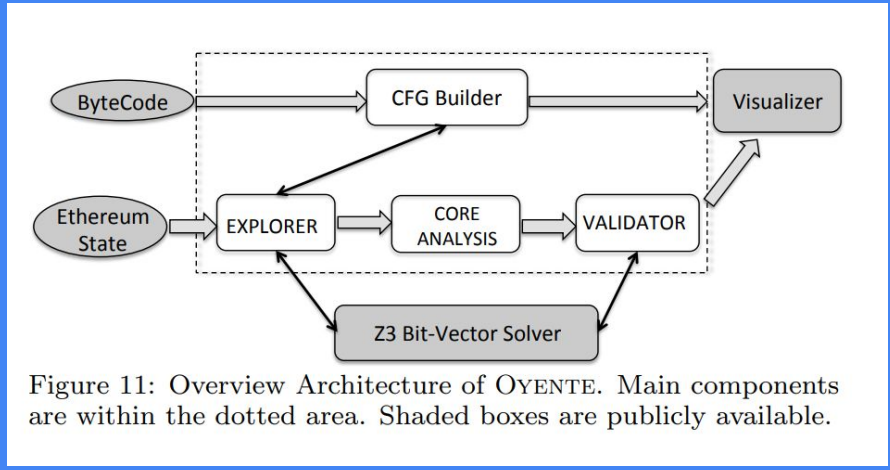


Figure 11: Overview Architecture of OYENTE. Main components are within the dotted area. Shaded boxes are publicly available.

- **VALIDATOR -**
  verifies that both orderings (t1, t2) and (t2, t1) are plausible for transactions t1 and t2, and if such orderings do not exist at all, the result of the core analysis is considered a false positive (FP).

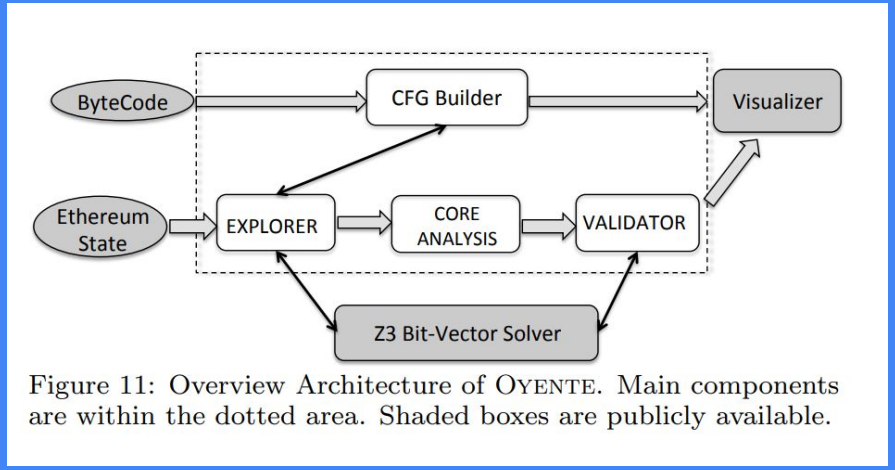


Figure 11: Overview Architecture of OYENTE. Main components are within the dotted area. Shaded boxes are publicly available.

EVALUATION

- The diagram illustrates the efficiency of the tool as of 2016.
- The tool was run on approximately 20,000 smart contracts in 2016. The combined value of these contracts was around 30 million dollars at the time the article was written.

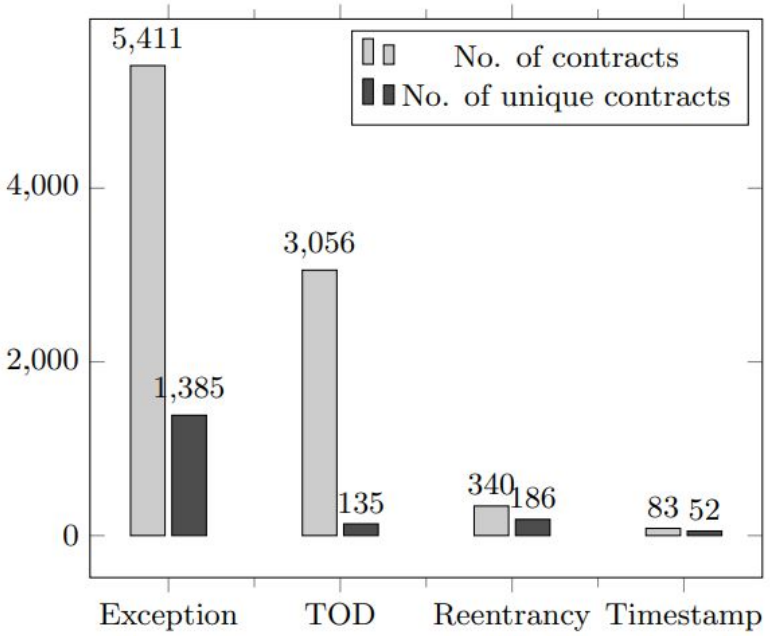


Figure 12: Number of buggy contracts per each security problem reported by OYENTE.

- We can see that most contracts use a stack depth of 0 to 50, which is well below the limit of 1024 stack-depth calls. This allows for an atmosphere of naivety where extreme cases are not thoroughly checked. It also makes it easier to attack the contract by pre-filling the stack so that an exception is only thrown on the 1024th call.

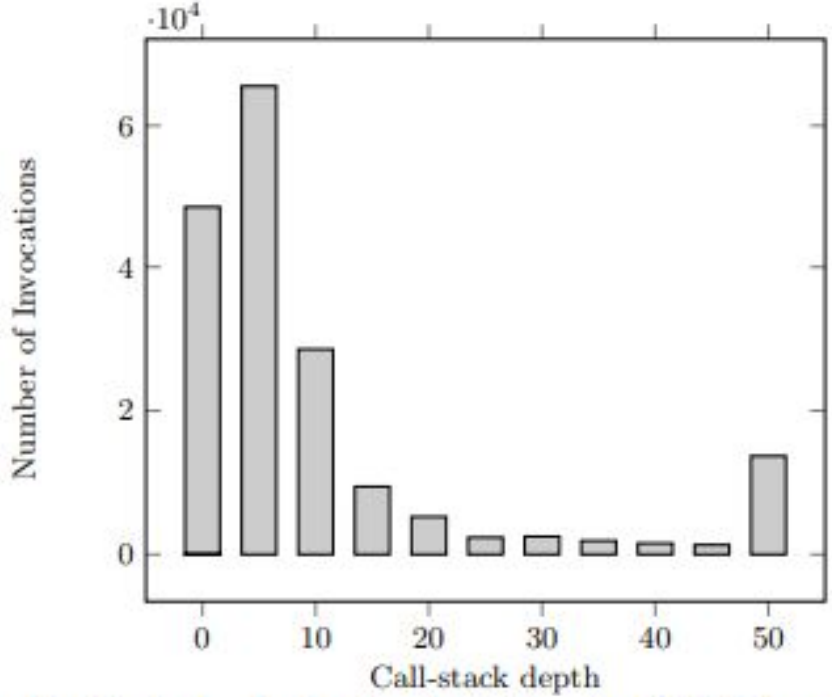


Figure 14: Call-stack depth statistics in 180,394 contract invocations in Ethereum network

- Let's take a look at the following contract, EtherID, which was very popular in 2016. There were 57,738 transactions executed, and using Oyente, we discovered an exception vulnerability by inspecting the stack.

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

Oyente example

SUMMARY

SUMMARY

What have we learned today?

- We learned about smart contracts and their importance in being secure for use.
- We discussed the potential issues that can arise when using smart contracts on the Ethereum network.
- We explored how to write smart contracts in a more secure manner.
- We introduced Oyente as a tool for identifying vulnerabilities in smart contracts.