# Quantitative Analysis of Smart Contracts

Krishnendu Chatterjee[1] Amir Goharshady[1] Yaron Velner[2]

[1]IST Austria

[2]Hebrew University of Jerusalem

ESOP 2018

# Outline

# Outline

# What are Smart Contracts?

- Blockchain is used in Bitcoin to induce a consensus about who owns what

# What are Smart Contracts?

- Blockchain is used in Bitcoin to induce a consensus about who owns what
- This is actually a consensus about the results of a computation

# What are Smart Contracts?

- Blockchain is used in Bitcoin to induce a consensus about who owns what
- This is actually a consensus about the results of a computation
- Blockchain can be used to ensure consensus about the state and outputs of any well-defined machine (program)

# What are Smart Contracts?

- Blockchain is used in Bitcoin to induce a consensus about who owns what
- This is actually a consensus about the results of a computation
- Blockchain can be used to ensure consensus about the state and outputs of any well-defined machine (program)
- Programs run on the Blockchain are called Decentralized Applications (dapps) or Smart Contracts

# What are Smart Contracts?

- Blockchain is used in Bitcoin to induce a consensus about who owns what
- This is actually a consensus about the results of a computation
- Blockchain can be used to ensure consensus about the state and outputs of any well-defined machine (program)
- Programs run on the Blockchain are called Decentralized Applications (dapps) or Smart Contracts
- Ethereum supports arbitrary stateful Turing-complete smart contracts

# An Example Contract – Token Transfer

```
1     contract Token {
2       mapping(address=>uint) balances;
3
4       function buy_tokens() payable {
5         balances[msg.sender] += msg.value;
6       }
7
8       function transfer( address to, uint amount ) {
9         if(balances[msg.sender]>=amount) {
10          uint x = balances[msg.sender];
11          uint y = balances[to];
12          balances[msg.sender] = x - amount;
13          balances[to] = y + amount;
14    }}}
```

# Another Example – Three-way Lottery

```
 1  contract Lottery {
 2
 3  address a=0,b=0,c=0;
 4
 5  function register() payable {
 6    require(msg.value == 1);
 7    require(a == 0 || b == 0 || c == 0);
 8    require(msg.sender!=a && msg.sender!=b && msg.sender!=c);
 9    if(a==0)
10      a = msg.sender;
11    else if(b==0)
12      b = msg.sender;
13    else
14      c = msg.sender;
15  }
16
17  mapping(address => uint) hashedChoices;
18
19  function makeChoice(uint choice){
20    require(a!=0 && b!=0 && c!=0);
21    require(msg.sender==a||msg.sender== b||msg.sender==c);
22    require(hashedChoices[msg.sender] == 0);
23    hashedChoices[msg.sender] = choice;
24  }
```

# Another Example – Three-way Lottery

```
1   mapping(address => uint) actualChoices;
2
3   function revealChoice(uint choice)
4   {
5     require(msg.sender==a||msg.sender== b||msg.sender==c);
6     require(hashedChoices[a]!=0);
7     require(hashedChoices[b]!=0);
8     require(hashedChoices[c]!=0);
9     require(sha256(choice) == hashedChoices[msg.sender]);
10    actualChoices[msg.sender] = choice;
11  }
```

# Another Example – Three-way Lottery

```
 1 address winner = 0;
 2
 3 function claim()
 4 {
 5   require(actualChoices[a]!=0);
 6   require(actualChoices[b]!=0);
 7   require(actualChoices[c]!=0);
 8
 9   if(actualChoices[a]%3 == actualChoices[b]%3)
10     winner = a;
11   else if((actualChoices[b] + actualChoices[c])%2 == 0)
12     winner = c;
13   else
14     winner = b;
15
16   winner.send(this.balance);
17 }
```

# Outline

# The Two Types of Bugs

- Coding Errors

# The Two Types of Bugs

- Coding Errors
  - At one reported case (HKG Token), mistakenly replacing $+=$ operation with $=+$ led to a loss of $800,000.

# The Two Types of Bugs

- Coding Errors
  - At one reported case (HKG Token), mistakenly replacing $+=$ operation with $=+$ led to a loss of $800,000.
  - Should be detected by standard verification

# The Two Types of Bugs

- Coding Errors
  - At one reported case (HKG Token), mistakenly replacing $+=$ operation with $=+$ led to a loss of $800,000.
  - Should be detected by standard verification
- Incentivization Bugs (Dishonest Interaction Incentives)

# The Two Types of Bugs

- Coding Errors
  - At one reported case (HKG Token), mistakenly replacing $+=$ operation with $=+$ led to a loss of $800,000.
  - Should be detected by standard verification
- Incentivization Bugs (Dishonest Interaction Incentives)
  - Due to game-theoretic interactions of contract parties

# The Two Types of Bugs

- Coding Errors
  - At one reported case (HKG Token), mistakenly replacing $+=$ operation with $=+$ led to a loss of \$800,000.
  - Should be detected by standard verification
- Incentivization Bugs (Dishonest Interaction Incentives)
  - Due to game-theoretic interactions of contract parties
  - Much harder to pin down

# The Two Types of Bugs

- Coding Errors
  - At one reported case (HKG Token), mistakenly replacing $+=$ operation with $=+$ led to a loss of $800,000.
  - Should be detected by standard verification
- Incentivization Bugs (Dishonest Interaction Incentives)
  - Due to game-theoretic interactions of contract parties
  - Much harder to pin down

Sometimes the two types coincide, i.e. a coding error leads to an incentive for dishonest interaction.

# Revisiting Token Transfer

```
1     contract Token {
2       mapping(address=>uint) balances;
3
4       function buy_tokens() payable {
5         balances[msg.sender] += msg.value;
6       }
7
8       function transfer( address to, uint amount ) {
9         if(balances[msg.sender]>=amount) {
10          uint x = balances[msg.sender];
11          uint y = balances[to];
12          balances[msg.sender] = x - amount;
13          balances[to] = y + amount;
14    }}}
```

# Revisiting the Lottery

```
 1 address winner = 0;
 2
 3 function claim()
 4 {
 5   require(actualChoices[a]!=0);
 6   require(actualChoices[b]!=0);
 7   require(actualChoices[c]!=0);
 8
 9   if(actualChoices[a]%3 == actualChoices[b]%3)
10     winner = a;
11   else if((actualChoices[b] + actualChoices[c])%2 == 0)
12     winner = c;
13   else
14     winner = b;
15
16   winner.send(this.balance);
17 }
```

# Outline

# Common Practices in Designing Contracts

- No loops
  - Due to “Gas” costs

# Common Practices in Designing Contracts

- No loops
  - Due to “Gas” costs
- Well-defined Phases

# Common Practices in Designing Contracts

- No loops
  - Due to “Gas” costs
- Well-defined Phases
- Concurrent Moves using Commitment Schemes

- We designed a programming language for writing contracts

- We designed a programming language for writing contracts
  - It has no loops

- We designed a programming language for writing contracts
  - It has no loops
  - Each function is assigned a time interval

- We designed a programming language for writing contracts
  - It has no loops
  - Each function is assigned a time interval
  - There is native support for commitment schemes, i.e. some functions get their parameters from different parties

- We designed a programming language for writing contracts
  - It has no loops
  - Each function is assigned a time interval
  - There is native support for commitment schemes, i.e. some functions get their parameters from different parties
- We showed that many real-world contracts can be written in our language pretty easily

# How Our Language Looks

```
contract RPS {
id Alice = issuer;
id Bob = null;
numeric bid[0,100] = 0;
numeric AliceWon[0,1] = 0;
numeric BobWon[0,1] = 0;
//0 denotes no choice,
//1 rock, 2 paper,
//3 scissors

function registerBob[1,10]
(payable _bid[0,100] : caller) {
      if(Bob==null) {
            Bob = caller;
            bid=_bid;
      }
      else
            payout(caller, bid);
}
```

```
function play[11,20]
  (numeric AlicesMove[0,3]=0: Alice,
   numeric BobsMove[0,3]=0: Bob,
   payable AlicesBid[0,100]=0: Alice)
   {
      id winner = null;
      if(AlicesBid!=bid)
        winner = Bob;
      else
        //set winner according to RPS rules
      if(winner==null)
      {
        payout(Alice, bid);
        payout(Bob, bid);
      }
      else
        payout(winner, 2*bid);
      //set the values of AliceWon and BobWon accordingly
   }
```

# Objectives

# Objectives

- We define an objective function $o$ for party $p$ and assume that she wants to maximize this objective. We assume that other parties are colluding to minimize it.

# Objectives

- We define an objective function $o$ for party $p$ and assume that she wants to maximize this objective. We assume that other parties are colluding to minimize it.
- The objective function can include not only monetary gains and losses, but also mathematical and logical expressions over the value of global variables at the end of the contract.
- For example, for a party $p$, her objective in a lottery can be:

$$p^+ - p^- + 10 \times [\texttt{winner} == p]$$

where $p^+$ is the amount she received from the contract and $p^-$ is the amount she paid. In a correct implementation of the three-way lottery, we expect the value of the contract to be $10/3$.

# Contract Values

- Intuitively, a policy $\sigma_p$ for a party $p$ of the contract is a function that, given the sequence of states the contract has visited up until now, suggests an action (or a distribution over actions) for $p$.
  This action is always in the form of issuing a message, i.e. setting a value for a variable, paying an amount or calling a function. It should always follow the rules of the contract.

# Contract Values

- Intuitively, a policy $\sigma_p$ for a party $p$ of the contract is a function that, given the sequence of states the contract has visited up until now, suggests an action (or a distribution over actions) for $p$.
  This action is always in the form of issuing a message, i.e. setting a value for a variable, paying an amount or calling a function. It should always follow the rules of the contract.
- The value of the contract for a party $p$ is her guaranteed earnings/losses assuming that all other parties are adversarial.

# Contract Values

- Intuitively, a policy $\sigma_p$ for a party $p$ of the contract is a function that, given the sequence of states the contract has visited up until now, suggests an action (or a distribution over actions) for $p$.
  This action is always in the form of issuing a message, i.e. setting a value for a variable, paying an amount or calling a function. It should always follow the rules of the contract.
- The value of the contract for a party $p$ is her guaranteed earnings/losses assuming that all other parties are adversarial.
- Formally, if we let $u_p(\sigma_p, \sigma_{-p})$ be the earnings of party $p$ when she follows a policy $\sigma_p$ and others follow $\sigma_{-p}$, then her contract value is:

$$\sup_{\sigma_p} \inf_{\sigma_{-p}} u_p(\sigma_p, \sigma_{-p}).$$

# Outline

# Concurrent Two-player Game Structures

A concurrent two-player game structure $G = (S, s_0, A, \Gamma_1, \Gamma_2, \delta)$ consists of:

# Concurrent Two-player Game Structures

A concurrent two-player game structure $G = (S, s_0, A, \Gamma_1, \Gamma_2, \delta)$ consists of:

- A finite set $S$ of states,

# Concurrent Two-player Game Structures

A concurrent two-player game structure $G = (S, s_0, A, \Gamma_1, \Gamma_2, \delta)$ consists of:

- A finite set $S$ of states,
- An initial (start) state $s_0$,

# Concurrent Two-player Game Structures

A concurrent two-player game structure $G = (S, s_0, A, \Gamma_1, \Gamma_2, \delta)$ consists of:

- A finite set $S$ of states,
- An initial (start) state $s_0$,
- A finite set $A$ of actions,

# Concurrent Two-player Game Structures

A concurrent two-player game structure $G = (S, s_0, A, \Gamma_1, \Gamma_2, \delta)$ consists of:

- A finite set $S$ of states,
- An initial (start) state $s_0$,
- A finite set $A$ of actions,
- Two action assignment functions $\Gamma_i : S \to 2^A \setminus \{\emptyset\}$. Intuitively, $\Gamma_i$ decides which actions are available to player $i$ at each state;

# Concurrent Two-player Game Structures

A concurrent two-player game structure $G = (S, s_0, A, \Gamma_1, \Gamma_2, \delta)$ consists of:

- A finite set $S$ of states,
- An initial (start) state $s_0$,
- A finite set $A$ of actions,
- Two action assignment functions $\Gamma_i : S \to 2^A \setminus \{\emptyset\}$. Intuitively, $\Gamma_i$ decides which actions are available to player $i$ at each state;
- and a transition function $\delta : S \times A \times A \to S$ that assigns to every state $s \in S$ and action pair $a_1 \in \Gamma_1(s), a_2 \in \Gamma_2(s)$ a successor state $\delta(s, a_1, a_2) \in S$.

# Gameplay

- The game starts at state $s_0$. At each state $s_i \in S$, player 1 chooses an action $a_1^i \in \Gamma_1(s_i)$ and player 2 chooses an action $a_2^i \in \Gamma_2(s_i)$. The choices are made simultaneously and independently. The game subsequently transitions to the new state $s_{i+1} = \delta(s_i, a_1, a_2)$ and the same process continues.

# Gameplay

- The game starts at state $s_0$. At each state $s_i \in S$, player 1 chooses an action $a_1^i \in \Gamma_1(s_i)$ and player 2 chooses an action $a_2^i \in \Gamma_2(s_i)$. The choices are made simultaneously and independently. The game subsequently transitions to the new state $s_{i+1} = \delta(s_i, a_1, a_2)$ and the same process continues.
- This leads to an infinite sequence of tuples $p = \left(s_i, a_1^i, a_2^i\right)_{i=0}^{\infty}$ which is called a *play* of the game.

# Gameplay

- The game starts at state $s_0$. At each state $s_i \in S$, player 1 chooses an action $a_1^i \in \Gamma_1(s_i)$ and player 2 chooses an action $a_2^i \in \Gamma_2(s_i)$. The choices are made simultaneously and independently. The game subsequently transitions to the new state $s_{i+1} = \delta(s_i, a_1, a_2)$ and the same process continues.
- This leads to an infinite sequence of tuples $p = \left(s_i, a_1^i, a_2^i\right)_{i=0}^{\infty}$ which is called a *play* of the game.
- The notion of a policy (strategy) is defined in the usual way

# Utilities and Game Values

- A utility function is of the form $u : S \to \mathbb{R}$ and assigns a utility to each state.
- The utility function can be extended to finite plays by summing up over the states.

# Utilities and Game Values

- A utility function is of the form $u : S \to \mathbb{R}$ and assigns a utility to each state.
- The utility function can be extended to finite plays by summing up over the states.
- A *Game* is a pair $(G, u)$ where $G$ is a game structure and $u$ is a utility function for player 1. We assume that player 1 is trying to maximize $u$, while player 2 aims to minimize it.

# Utilities and Game Values

- A utility function is of the form $u : S \to \mathbb{R}$ and assigns a utility to each state.
- The utility function can be extended to finite plays by summing up over the states.
- A *Game* is a pair $(G, u)$ where $G$ is a game structure and $u$ is a utility function for player 1. We assume that player 1 is trying to maximize $u$, while player 2 aims to minimize it.
- The L-step finite-horizon value of a game $(G, u)$ is defined as

$$V_{\mathsf{L}}(G, u) := \sup_{\sigma_1} \inf_{\sigma_2} \mathbb{E}^{(\sigma_1, \sigma_2)} \left[ u_{\mathsf{L}}(p) \right],$$

where $\sigma_i$ iterates over all possible mixed strategies of player $i$. This models the fact that player 1 is trying to maximize the utility in the first L steps of the run, while player 2 is minimizing it.

# Translating Contracts to Games

We consider the bounded analysis problem, where

# Translating Contracts to Games

We consider the bounded analysis problem, where

- The number of parties is bounded

# Translating Contracts to Games

We consider the bounded analysis problem, where

- The number of parties is bounded
- The number of function calls by each party at each time frame is also bounded.

# Translating Contracts to Games

We consider the bounded analysis problem, where

- The number of parties is bounded
- The number of function calls by each party at each time frame is also bounded.
  - In real-life contracts one's ability to call many functions is limited by the capacity of a block in the blockchain.

# Huge State-Space

- The state-space of the resulting games are huge. There is one state for each possible valuation of the variables at each time frame.

# Huge State-Space

- The state-space of the resulting games are huge. There is one state for each possible valuation of the variables at each time frame.
- :(

# Huge State-Space

- The state-space of the resulting games are huge. There is one state for each possible valuation of the variables at each time frame.
- :(
- :(

# Abstraction

- We use a sound abstraction method to reduce the number of states.

# Abstraction

- We use a sound abstraction method to reduce the number of states.
- We first partition the states into several sets.

# Abstraction

- We use a sound abstraction method to reduce the number of states.
- We first partition the states into several sets.
- We then create two new games $(G^{\uparrow}, u^{\uparrow})$ and $(G^{\downarrow}, u^{\downarrow})$. These games are obtained by merging the states that are in the same partition to form a single "abstract" state.

# Abstraction

- We use a sound abstraction method to reduce the number of states.
- We first partition the states into several sets.
- We then create two new games $(G^{\uparrow}, u^{\uparrow})$ and $(G^{\downarrow}, u^{\downarrow})$. These games are obtained by merging the states that are in the same partition to form a single "abstract" state.
- In $u^{\uparrow}$, an abstract state's utility is the maximum utility among normal states that it corresponds to. In $u^{\downarrow}$, we take the minimum utility.
- When at an abstract state $s$ the players play actions $a_1$ and $a_2$, the game can transition to any abstract state that contains the result of a transition with the same actions from a normal state in $s$. In $G^{\uparrow}$ player 1 chooses the resulting transition (solves the nondeterminism), while in $G^{\downarrow}$ player 2 does this.

# Soundness, Refinement and Completeness in the Limit

It is easy to check that our abstraction has the following three properties (for proof and formal treatment see the paper):

- **Soundness.** The value of $(G^{\downarrow}, u^{\downarrow})$ is always a lower-bound for that of $(G, u)$ and the value of $(G^{\uparrow}, u^{\uparrow})$ is always an upper-bound. Hence, solving each abstracted game pair gives us an interval that contains the value of the original game.
- **Refinement.** If we refine our partitions by breaking each set into smaller sets, the interval shrinks.
- **Completeness in the Limit.** For any $\epsilon > 0$, we can refine our partitions such that the length of the interval becomes less than $\epsilon$.

# Outline

# Experimental Results

# Experimental Results

- We implemented the approach with some heuristics for refining abstractions

# Experimental Results

- We implemented the approach with some heuristics for refining abstractions
- We tested our approach on several real-world bugs (including the two examples shown before).

# Experimental Results

- We implemented the approach with some heuristics for refining abstractions
- We tested our approach on several real-world bugs (including the two examples shown before).
- In each case, we coded both the correct and the buggy variant of the contract.

- The original contracts had up to more than $10^{23}$ concrete states.

| **Sale** | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Size** | Abstractions | | | | | | |
| | Correct Program | | | | Buggy Variant | | |
| | states | $[l$ , | $u]$ | time | states | $[l$ , | $u]$ | time |
| $> \mathbf{4.6 \cdot 10^{22}}$ | 17010 | [0 , | 2000] | 226 | 17010 | [0 , | 2000] | 275 |
| | 75762 | [723 , | 1472] | 1241 | 81202 | [1167 , | 2000] | 1733 |
| | 131250 | [792 , | 1260] | 2872 | 124178 | [1741 , | 2000] | 2818 |

| **Transfer** | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Size** | Abstractions | | | | | | |
| | Correct Program | | | | Buggy Variant | | |
| | states | $[l$ , | $u]$ | time | states | $[l$ , | $u]$ | time |
| $> \mathbf{10^{23}}$ | 1040 | [0 , | 2000] | 20 | 6561 | [0 , | 2000] | 237 |
| | 32880 | [844 , | 1793] | 562 | 131520 | [1716 , | 2000] | 3979 |
| | 148311 | [903 , | 1352] | 3740 | | | | |

- We refined our abstraction to the point that the intervals for values of the two contracts became disjoint.

| **Sale** | | | | | | | |
|---|---|---|---|---|---|---|---|
| Size | Abstractions | | | | | | |
| | Correct Program | | | Buggy Variant | | | |
| | states | $[l$ , | $u]$ time | states | $[l$ , | $u]$ | time |
| $> 4.6 \cdot 10^{22}$ | **17010** | **[0** , | **2000]** 226 | **17010** | **[0** , | **2000]** | 275 |
| | **75762** | **[723** , | **1472]** 1241 | **81202** | **[1167** , | **2000]** | 1733 |
| | **131250** | **[792** , | **1260]** 2872 | **124178** | **[1741** , | **2000]** | 2818 |

| **Transfer** | | | | | | | |
|---|---|---|---|---|---|---|---|
| Size | Abstractions | | | | | | |
| | Correct Program | | | Buggy Variant | | | |
| | states | $[l$ , | $u]$ time | states | $[l$ , | $u]$ | time |
| $> 10^{23}$ | **1040** | **[0** , | **2000]** 20 | **6561** | **[0** , | **2000]** | 237 |
| | **32880** | **[844** , | **1793]** 562 | **131520** | **[1716** , | **2000]** | 3979 |
| | **148311** | **[903** , | **1352]** 3740 | | | | |

- The runtimes were a few hours in some cases, but we are happy with it given that each contract should only be checked once before its deployment.

| **Sale** | | | | | | |
|---|---|---|---|---|---|---|
| Size | Abstractions | | | | | |
| | Correct Program | | | Buggy Variant | | |
| | states | $[l, u]$ | time | states | $[l, u]$ | time |
| $> 4.6 \cdot 10^{22}$ | 17010 | [0, 2000] | 226 | 17010 | [0, 2000] | 275 |
| | 75762 | [723, 1472] | 1241 | 81202 | [1167, 2000] | 1733 |
| | 131250 | [792, 1260] | **2872** | 124178 | [1741, 2000] | **2818** |

| **Transfer** | | | | | | |
|---|---|---|---|---|---|---|
| Size | Abstractions | | | | | |
| | Correct Program | | | Buggy Variant | | |
| | states | $[l, u]$ | time | states | $[l, u]$ | time |
| $> 10^{23}$ | 1040 | [0, 2000] | 20 | 6561 | [0, 2000] | 237 |
| | 32880 | [844, 1793] | 562 | 131520 | [1716, 2000] | **3979** |
| | 148311 | [903, 1352] | **3740** | | | |