Quantitative Analysis of Smart Contracts*

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

[1]IST Austria (Institute of Science and Technology Austria)
[2]Hebrew University of Jerusalem
{krishnendu.chatterjee, amir.goharshady}@ist.ac.at
yaron.welner@mail.huji.ac.il

Abstract. Smart contracts are computer programs that are executed by a network of mutually distrusting agents, without the need of an external trusted authority. Smart contracts handle and transfer assets of considerable value (in the form of crypto-currency like Bitcoin). Hence, it is crucial that their implementation is bug-free. We identify the utility (or expected payoff) of interacting with such smart contracts as the basic and canonical quantitative property for such contracts. We present a framework for such quantitative analysis of smart contracts. Such a formal framework poses new and novel research challenges in programming languages, as it requires modeling of game-theoretic aspects to analyze incentives for deviation from honest behavior and modeling utilities which are not specified as standard temporal properties such as safety and termination. While game-theoretic incentives have been analyzed in the security community, their analysis has been restricted to the very special case of stateless games. However, to analyze smart contracts, stateful analysis is required as it must account for the different program states of the protocol. Our main contributions are as follows: we present (i) a simplified programming language for smart contracts; (ii) an automatic translation of the programs to state-based games; (iii) an abstraction-refinement approach to solve such games; and (iv) experimental results on real-world-inspired smart contracts.

1 Introduction

In this work we present a quantitative stateful game-theoretic framework for formal analysis of smart-contracts.

Smart contracts. Hundreds of crypto-currencies are in use today, and investments in them are increasing steadily [24]. These currencies are not controlled by any central authority like governments or banks, instead they are governed by the *blockchain* protocol, which dictates the rules and determines the outcomes, e.g., the validity of money transactions and account balances. Blockchain was initially used for peer-to-peer Bitcoin payments [43], but recently it is also used for running programs (called smart contracts). A *smart contract* is a program that runs on the blockchain, which enforces its correct execution (i.e., that it is running as

* A longer version of this article is available in [19].

originally programmed). This is done by encoding semantics in crypto-currency transactions. For example, Bitcoin transaction scripts allow users to specify conditions, or contracts, which the transactions must satisfy prior to acceptance. Transaction scripts can encode many useful functions, such as validating that a payer owns a coin she is spending or enforcing rules for multi-party transactions. The Ethereum crypto-currency [16] allows arbitrary stateful Turing-complete conditions over the transactions which gives rise to smart contracts that can implement a wide range of applications, such as financial instruments (e.g., financial derivatives or wills) or autonomous governance applications (e.g., voting systems). The protocols are globally specified and their implementation is decentralized. Therefore, there is no central authority and they are immutable. Hence, the economic consequences of bugs in a smart contract cannot be reverted.

Types of Bugs. There are two types of bugs with monetary consequences:

1. *Coding errors.* Similar to standard programs, bugs could arise from coding mistakes. At one reported case [33], mistakenly replacing `+=` operation with `=+` enabled loss of tokens that were backed by $800,000 of investment.
2. *Dishonest interaction incentives.* Smart contracts do not fully dictate the behavior of participants. They only specify the outcome (e.g., penalty or rewards) of the behaviors. Hence, a second source for bugs is the high level *interaction aspects* that could give a participant unfair advantage and incentive for dishonest behavior. For example, a naive design of rock-paper-scissors game [29] allows playing sequentially, rather than concurrently, and gives advantage to the second player who can see the opponent's move.

DAO attack: interaction of two types of bugs. Quite interestingly a coding bug can incentivize dishonest behavior as in the famous DAO attack [48]. The Decentralized Autonomous Organization (DAO) [38] is an Ethereum smart contract [51]. The contract consists of investor-directed venture capital fund. On June 17, 2016 an attacker exploited a bug in the contract to extract $80 million [48]. Intuitively, the root cause was that the contract allowed users to first get hold of their funds, and only then updated their balance records while a semantic detail allowed the attacker to withdraw multiple times before the update.

Necessity of formal framework. Since bugs in smart contracts have direct economic consequences and are irreversible, they have the same status as safety-critical errors for programs and reactive systems and must be detected before deployment. Moreover, smart contracts are deployed rapidly. There are over a million smart contracts in Ethereum, holding over 15 billion dollars at the time of writing [31]. It is impossible for security researchers to analyze all of them, and lack of automated tools for programmers makes them error prone. Hence, a formal analysis framework for smart contract bugs is of great importance.

Utility analysis. In verification of programs, specifying objectives is non-trivial and a key goal is to consider specification-less verification, where basic properties are considered canonical. For example, termination is a basic property in program analysis; and data-race freedom or serializability are basic properties in concurrency. Given these properties, models are verified wrt them without considering any other specification. For smart contracts, describing the correct specification that prevents dishonest behavior is more challenging due to the presence of game-like interactions. We propose to consider the expected user utility (or payoff) that is guaranteed even in presence of adversarial behavior of other agents as a canonical property. Considering malicious adversaries is standard in game theory. For example, the expected utility of a fair lottery is 0. An analysis reporting a different utility signifies a bug.

New research challenges. Coding bugs are detected by classic verification, program analysis, and model checking tools [23, 39]. However, a formal framework for incentivization bugs presents a new research challenge for the programming language community. Their analysis must overcome two obstacles: (a) the framework will have to handle game-theoretic aspects to model interactions and incentives for dishonest behavior; and (b) it will have to handle properties that cannot be deduced from standard temporal properties such as safety or termination, but require analysis of monetary gains (i.e., quantitative properties).

While game-theoretic incentives are widely analyzed by the security community (e.g., see [13]), their analysis is typically restricted to the very special case of one-shot games that do not consider different states of the program, and thus the consequences of decisions on the next state of the program are ignored. In addition their analysis is typically ad-hoc and stems from brainstorming and special techniques. This could work when very few protocols existed (e.g., when bitcoin first emerged) and deep thought was put into making them elegant and analyzable. However, the fast deployment of smart contracts makes it crucial to automate the process and make it accessible to programmers.

Our contribution. In this work we present a formal framework for quantitative analysis of utilities in smart contracts. Our contributions are as follows:

1. We present a simplified (loop-free) programming language that allows game-theoretic interactions. We show that many classical smart contracts can be easily described in our language, and conversely, a smart contract programmed in our language can be easily translated to Solidity [30], which is the most popular Ethereum smart contract language.
2. The underlying mathematical model for our language is stateful concurrent games. We automatically translate programs in our language to such games.
3. The key challenge to analyze such game models automatically is to tackle the state-space explosion. While several abstraction techniques have been considered for programs [45, 35, 14], they do not work for game-theoretic models with quantitative objectives. We present an approach based on interval-abstraction for reducing the states, establish soundness of our abstraction, and present a refinement process. This is our core technical contribution.
4. We present experimental results on several classic real-world smart contracts. We show that our approach can handle contracts that otherwise give rise to games with up to 10^{23} states. While special cases of concurrent games (namely, turn-based games) have been studied in verification and reactive synthesis, there are no practical methods to solve general concurrent quantitative games. To the best of our knowledge, there are no tools to solve quantitative concurrent games other than academic examples of few states,

and we present the first practical method to solve quantitative concurrent games that scales to real-world smart contract analysis.

In summary, our contributions range from (i) modeling of smart contracts as state-based games, to (ii) an abstraction-refinement approach to solve such games, to (iii) experimental results on real-world smart contracts.

2 Background on Ethereum smart contracts

2.1 Programmable smart contracts

Ethereum [16] is a decentralized virtual machine, which runs programs called contracts. Contracts are written in a Turing-complete bytecode language, called Ethereum Virtual Machine (EVM) bytecode [53]. A contract is invoked by calling one of its functions, where each function is defined by a sequence of instructions. The contract maintains a persistent internal state and can receive (transfer) currency from (to) users and other contracts. Users send transactions to the Ethereum network to invoke functions. Each transaction may contain input parameters for the contract and an associated monetary amount, possibly 0, which is transferred from the user to the contract.

Upon receiving a transaction, the contract collects the money sent to it, executes a function according to input parameters, and updates its internal state. All transactions are recorded on a decentralized ledger, called blockchain. A sequence of transactions that begins from the creation of the network uniquely determines the state of each contract and balances of users and contracts. The blockchain does not rely on a trusted central authority, rather, each transaction is processed by a large network of mutually untrusted peers called miners. Users constantly broadcast transactions to the network. Miners add transactions to the blockchain via a proof-of-work consensus protocol [43].

Subtleties. In this work, for simplicity, we ignore some details in the underlying protocol of Ethereum smart contract. We briefly describe these details below:

- *Transaction fees.* In exchange for including her transactions in the blockchain, a user pays transaction fees to the miners, proportionally to the execution time of her transaction. This fact could slightly affect the monetary analysis of the user gain, but could also introduce bugs in a program, as there is a bound on execution time that cannot be exceeded. Hence, it is possible that some functions could never be called, or even worse, a user could actively give input parameters that would prevent other users from invoking a certain function.
- *Recursive invocation of contracts.* A contract function could invoke a function in another contract, which in turn can have a call to the original contract. The underling Ethereum semantic in recursive invocation was the root cause for the notorious DAO hack [27].
- *Behavior of the miners.* Previous works have suggested that smart contracts could be implemented to encourage miners to deviate from their honest behavior [50]. This could in theory introduce bugs into a contract, e.g., a contract might give unfair advantage for a user who is a big miner.

2.2 Tokens and user utility

A user's utility is determined by the Ether she spends and receives, but could also be affected by the state of the contract. Most notably, smart contracts are used to issue *tokens*, which can be viewed as a stake in a company or an organization, in return to an Ether (or tokens) investment (see an example in Figure 1). These tokens are *transferable* among users and are traded in exchanges in return to Ether, Bitcoin and Fiat money. At the time of writing, smart contracts instantiate tokens worth billions of dollars [32]. Hence, gaining or losing tokens has clear utility for the user. At a larger scope, user utility could also be affected by more abstract storage changes. Some users would be willing to pay to have a contract declare them as Kings of Ether [4], while others could gain from registering their domain name in a smart contract storage [40]. In the examples provided in this work we mainly focus on utility that arises from Ether, tokens and the like. However, our approach is general and can model any form of utility by introducing auxiliary utility variables and definitions.

```
contract Token {
    mapping(address=>uint) balances;
    function buy() payable {
        balances[msg.sender] += msg.value;
    }
    function transfer( address to, uint amount ) {
        if(balances[msg.sender]>=amount) {
            balances[msg.sender] -= amount;
            balances[to] += amount;
    }}}
```

Fig. 1: Token contract example.

3 Programming Language for Smart Contracts

In this section we present our programming language for smart contracts that supports concurrent interactions between parties. A party denotes an agent that decides to interact with the contract. A contract is a tuple $C = (N, I, M, R, X_0, F, T)$ where $X := N \cup I \cup M$ is a set of variables, R describes the range of values that can be stored in each variable, X_0 is the initial values stored in variables, F is a list of functions and T describes for each function, the time segment in which it can be invoked. We now formalize these concepts.

Variables. There are three distinct and disjoint types of variables in X:

- N contains "numeric" variables that can store a single integer.
- I contains "identification" ("id") variables capable of pointing to a party in the contract by her address or storing Null. The notion of ids is quite flexible in our approach: The only dependence on ids is that they should be distinct and an id should not act on behalf of another id. We simply use different

integers to denote distinct ids and assume that a “faking of identity” does not happen. In Ethereum this is achieved by digital signatures.

- M is the set of “mapping” variables. Each $m \in M$ maps parties to integers.

Bounds and Initial values. The tuple $R = (\underline{R}, \overline{R})$ where $\underline{R}, \overline{R} : N \cup M \to \mathbb{Z}$ represent lower and upper bounds for integer values that can be stored in a variable. For example, if $n \in N$, then n can only store integers between $\underline{R}(n)$ and $\overline{R}(n)$. Similarly, if $m \in M$ is a mapping and $i \in I$ stores an address to a party in the contract, then $m\,[i]$ can save integers between $\underline{R}(m)$ and $\overline{R}(m)$. The function $X_0 : X \to \mathbb{Z} \cup \{\textsc{Null}\}$ assigns an initial value to every variable. The assigned value is an integer in case of numeric and mapping variables, i.e., a mapping variable maps everything to its initial value by default. Id variables can either be initialized by Null or an id used by one of the parties.

Functions and Timing. The sequence $F =< f_1, f_2, \ldots, f_n >$ is a list of functions and $T = (\underline{T}, \overline{T})$, where $\underline{T}, \overline{T} : F \to \mathbb{N}$. The function f_i can only be invoked in time-frame $T(f_i) = \left[\underline{T}(f_i), \overline{T}(f_i)\right]$. The contract uses a global clock, for example the current block number in the blockchain, to keep track of time.

Note that we consider a single contract, and interaction between multiple contracts is a subject of future work.

3.1 Syntax

We provide a simple overview of our contract programming language. Our language is syntactically similar to Solidity [30], which is a widely used language for writing Ethereum contracts. A translation mechanism for different aspects is discussed in [19]. An example contract, modeling a game of rock-paper-scissors, is given in Figure 2. Here, a party, called `issuer` has issued the contract and taken the role of `Alice`. Any other party can join the contract by registering as `Bob` and then playing rock-paper-scissors. To demonstrate our language, we use a bidding mechanism.

Declaration of Variables. The program begins by declaring variables[1], their type, name, range and initial value. For example, `Bids` is a map variable that assigns a value between 0 and 100 to every id. This value is initially 0. Line numbers (labels) are defined in Section 3.2 below and are not part of the syntax.

Declaration of Functions. After the variables, the functions are defined one-by-one. Each function begins with the keyword `function` followed by its name and the time interval in which it can be called by parties. Then comes a list of input parameters. Each parameter is of the form `variable : party` which means that the designated party can choose a value for that variable. The chosen value is required to be in the range specified for that variable. The keyword `caller` denotes the party that has invoked this function and `payable` signifies that the party should not only decide a value, but must also pay the amount she decides.

[1] For simplicity, we demonstrate our method with global variables only. However, the method is applicable to general variables as long as their ranges are well-defined at each point of the program.

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

(1) function registerBob[1,10]
     (payable bid : caller) {
(2)    if(Bob==null) {
(3)      Bob = caller;
(4)      Bids[Bob]=bid;
       }
       else{
(5)      payout(caller, bid);
       }
(6) }
(7) function play[11, 15]
    (AlicesMove:Alice = 0,
    BobsMove:Bob = 0,
    payable Bids[Alice]: Alice){
(8)  if(played==1)
(9)     return;
     else
(10)    played = 1;
```

```
(11) if(BobsMove==0 and AlicesMove!=0)
(12)      AliceWon = 1;
(13) else if(AlicesMove==0 and BobsMove!=0)
(14)      BobWon = 1;
(15) else if(AlicesMove==0 and BobsMove==0)
      {
(16)      AliceWon = 0;
(17)      BobWon = 0;
      }
(18) else if(AlicesMove==BobsMove+1 or
        AlicesMove==BobsMove-2)
(19)      AliceWon = 1;
      else
(20)      BobWon = 1;
(21) }

(22) function getReward[16,20]() {
(23)  if(caller==Alice and AliceWon==1
      or caller==Bob and BobWon==1)
      {
(24)    payout(caller,
               Bids[Alice] + Bids[Bob]);
(25)    Bids[Alice] = 0;
(26)    Bids[Bob] = 0;
      }
(27) }
    }
```

Fig. 2: A rock-paper-scissors contract.

For example, registerBob can be called in any time between 1 and 10 by any of the parties. At each such invocation the party that has called this function must pay some amount which will be saved in the variable bid. After the decisions and payments are done, the contract proceeds with executing the function.

Types of Functions. There are essentially two types of functions, depending on their parameters. *One-party functions*, such as registerBob and getReward require parameters from caller only, while *multi-party functions*, such as play ask several, potentially different, parties for input. In this case all parties provide their input decisions and payments concurrently and without being aware of the choices made by other parties, also a default value is specified for every decision in case a relevant party does not take part.

Summary. Putting everything together, in the contract specified in Figure 2, any party can claim the role of Bob between time 1 and time 10 by paying a bid to the contract, if the role is not already occupied. Then at time 11 one of the parties calls play and both parties have until time 15 to decide which choice (rock, paper, scissors or none) they want to make. Then the winner can call getReward and collect her prize.

3.2 Semantics

In this section we present the details of the semantics. In our programming language there are several key aspects which are non-standard in programming languages, such as the notion of time progress, concurrency, and interactions of several parties. Hence we present a detailed description of the semantics. We start with the requirements.

Requirements. In order for a contract to be considered valid, other than following the syntax rules, a few more requirements must be met, which are as follows:

- We assume that no division by zero or similar undefined behavior happens.
- To have a well-defined message passing, we also assume that no multi-party function has an associated time interval intersecting that of another function.
- Finally, for each non-id variable v, it must hold that $\underline{R}(v) \leq X_0(v) \leq \overline{R}(v)$ and similarly, for every function f_i, we must have $\underline{T}(f_i) < \overline{T}(f_i)$.

Overview of time progress. Initially, the time is 0. Let F_t be the set of functions executable at time t, i.e., $F_t = \{f_i \in F | t \in T(f_i)\}$, then F_t is either empty or contains one or more one-party functions or consists of a single multi-party function. We consider the following cases:

- *F_t empty.* If F_t is empty, then nothing can happen until the clock ticks.
- *Execution of one-party functions.* If F_t contains one or more one-party functions, then each of the parties can call any subset of these functions at time t. If there are several calls at the same time, the contract might run them in any order. While a function call is being executed, all parties are able to see the full state of the contract, and can issue new calls. When there are no more requests for function calls, the clock ticks and the time is increased to $t + 1$. When a call is being executed and is at the beginning part of the function, its caller can send messages or payments to the contract. Values of these messages and payments will then be saved in designated variables and the execution continues. If the caller fails to make a payment or specify a value for a decision variable or if her specified values/payments are not in the range of their corresponding variables, i.e. they are too small or too big, the call gets canceled and the contract reverts any changes to variables due to the call and continues as if this call had never happened.
- *Execution of multi-party functions.* If F_t contains a single multi-party function f_i and $t < \overline{T}(f_i)$, then any party can send messages and payments to the contract to specify values for variables that are designated to be paid or decided by her. These choices are hidden and cannot be observed by other participants. She can also change her decisions as many times as she sees fit. The clock ticks when there are no more valid requests for setting a value for a variable or making a payment. This continues until we reach time $\overline{T}(f_i)$. At this time parties can no longer change their choices and the choices become visible to everyone. The contract proceeds with execution of the function. If a party fails to make a payment/decision or if NULL is asked to make a payment or a decision, default behavior will be enforced. Default value for payments is 0 and default behavior for other variables is defined as part of the syntax. For example, in function `play` of Figure 2, if a party does not

choose, a default value of 0 is enforced and given the rest of this function, this will lead to a definite loss.

Given the notion of time progress we proceed to formalize the notion of "runs" of the contract. This requires the notion of labels, control-flow graphs, valuations, and states, which we describe below.

Labels. Starting from 0, we give the contract, beginning and end points of every function, and every command a label. The labels are given in order of appearance. As an example, see the labels in parentheses in Figure 2.

Entry and Exit Labels. We denote the first (beginning point) label in a function f_i by $\square_i$ and its last (end point) label by $\blacksquare_i$.

Control Flow Graphs (CFGs). We define the control flow graph CFG_i of the function f_i in the standard manner, i.e. $CFG_i = (V, E)$, where there is a vertex corresponding to every labeled entity inside f_i. Each edge $e \in E$ has a condition $cond(e)$ which is a boolean expression that must be true when traversing that edge. For more details see [19].

Valuations. A valuation is a function *val*, assigning a value to every variable. Values for numeric variables must be integers in their range, values for identity variables can be party ids or Null and a value assigned to a map variable m must be a function $val(m)$ such that for each identity i, we have $\underline{R}(m) \leq val(m)(i) \leq \overline{R}(m)$. Given a valuation, we extend it to expressions containing mathematical operations in the straight-forward manner.

States. A state of the contract is a tuple $s = (t, b, l, val, c)$, where t is a time stamp, $b \in \mathbb{N} \cup \{0\}$ is the current balance of the contract, i.e., the total amount of payment to the contract minus the total amount of payouts, l is a label (that is being executed), *val* assigns values to variables and $c \in P \cup \{\perp\}$, is the caller of the current function. $c = \perp$ corresponds to the case where the caller is undefined, e.g., when no function is being executed. We use S to denote the set of all states that can appear in a run of the contract as defined below.

Runs. A run ρ of the contract is a finite sequence $\{\rho_j = (t_j, b_j, l_j, val_j, c_j)\}_{j=0}^{r}$ of states, starting from $(0, 0, 0, X_0, \perp)$, that follows all rules of the contract and ends in a state with time-stamp $t_r > \max_{f_i} \overline{T}(f_i)$. These rules must be followed when switching to a new state in a run:

- The clock can only tick when there are no valid pending requests for running a one-party function or deciding or paying in multi-party functions.
- Transitions that happen when the contract is executing a function must follow its control flow graph and update the valuation correctly.
- No variable can contain an out-of-bounds value. If an overflow or underflow happens, the closest possible value will be saved. This rule also ensures that the contract will not create new money, given that paying more than the current balance of the contract results in an underflow.
- Each party can call any set of the functions at any time.

Remark 1. Note that in our semantics each function body completes its execution in a single tick of the clock. However, ticks might contain more than one function call and execution.

Run prefixes. We use H to mean the set of all prefixes of runs and denote the last state in $\eta \in H$ by $\mathit{end}(\eta)$. A run prefix η' is an extension of η if it can be obtained by adding one state to the end of η.

Probability Distributions. Given a finite set $\mathcal{X}$, a probability distribution on $\mathcal{X}$ is a function $\delta : \mathcal{X} \to [0,1]$ such that $\sum_{x \in \mathcal{X}} \delta(x) = 1$. Given such a distribution, its support, $\mathit{Supp}(\delta)$, is the set of all $x \in \mathcal{X}$ such that $\delta(x) > 0$. We denote the set of all probability distributions on $\mathcal{X}$ by $\Delta(\mathcal{X})$.

Typically for programs it suffices to define runs for the semantics. However, given that there are several parties in contracts, their semantics depends on the possible choices of the parties. Hence we need to define policies for parties, and such policies will define probability distribution over runs, which constitute the semantics for contracts. To define policies we first define moves.

Moves. We use $\mathcal{M}$ for the set of all moves. The moves that can be taken by parties in a contract can be summarized as follows:

- Calling a function f_i, we denote this by $\mathit{call}(f_i)$.
- Making a payment whose amount, y is saved in x, we denote this by $\mathit{pay}(x, y)$.
- Deciding the value of x to be y, we denote this by $\mathit{decide}(x, y)$.
- Doing none of the above, we denote this by $\boxtimes$.

Permitted Moves. We define $P_i : S \to \mathcal{M}$, so that $P_i(s)$ is the set of permitted moves for the party with identity i if the contract is in state $s = (t, b, l, \mathit{val}, p_j)$. It is formally defined as follows:

- If f_k is a function that can be called at state s, then $\mathit{call}(f_k) \in P_i(s)$.
- If $l = \Box_q$ is the first label of a function f_q and x is a variable that can be decided by i at the beginning of the function f_q, then $\mathit{decide}(x, y) \in P_i(s)$ for all permissible values of y. Similarly if x can be paid by i, $\mathit{pay}(x, y) \in P_i(s)$.
- $\boxtimes \in P_i(s)$.

Policies and Randomized Policies. A policy π_i for party i is a function $\pi_i : H \to A$, such that for every $\eta \in H$, $\pi_i(\eta) \in P_i(\mathit{end}(\eta))$. Intuitively, a policy is a way of deciding what move to use next, given the current run prefix. A policy profile $\pi = (\pi_i)$ is a sequence assigning one policy to each party i. The policy profile π defines a unique run ρ^π of the contract which is obtained when parties choose their moves according to π. A randomized policy ξ_i for party i is a function $\xi_i : H \to \Delta(\mathcal{M})$, such that $\mathit{Supp}(\xi_i(s)) \subseteq P_i(s)$. A randomized policy assigns a probability distribution over all possible moves for party i given the current run prefix of the contract, then the party can follow it by choosing a move randomly according to the distribution. We use Ξ to denote the set of all randomized policy profiles, Ξ_i for randomized policies of i and Ξ_{-i} to denote the set of randomized policy profiles for all parties except i. A randomized policy profile ξ is a sequence (ξ_i) assigning one randomized policy to each party. Each such randomized policy profile induces a unique probability measure on the set of runs, which is denoted as $\mathsf{Prob}^\xi[\cdot]$. We denote the expectation measure associated to $\mathsf{Prob}^\xi[\cdot]$ by $\mathbb{E}^\xi[\cdot]$.

3.3 Objective function and values of contracts

As mentioned in the introduction we identify expected payoff as the canonical property for contracts. The previous section defines expectation measure given randomized policies as the basic semantics. Given the expected payoff, we define values of contracts as the worst-case guaranteed payoff for a given party. We formalize the notion of objective function (the payoff function).

Objective Function. An objective o for a party p is in one of the following forms:

- $(p^+ - p^-)$, where p^+ is the total money received by party p from the contract (by "payout" statements) and p^- is the total money paid by p to the contract (as "payable" parameters).
- An expression containing mathematical and logical operations (addition, multiplication, subtraction, integer division, and, or, not) and variables chosen from the set $N \cup \{m\,[i]\,|m \in M, i \in I\}$. Here N is the set of numeric variables, $m[i]$'s are the values that can be saved inside maps.[2]
- A sum of the previous two cases.

Informally, p is trying to choose her moves so as to maximize o.

Run Outcomes. Given a run ρ of the program and an objective o for party p, the outcome $\kappa(\rho, o, p)$ is the value of o computed using the valuation at $\mathit{end}(\rho)$ for all variables and accounting for payments in ρ to compute p^+ and p^-.

Contract Values. Since we consider worst-case guaranteed payoff, we consider that there is an objective o for a single party p which she tries to maximize and all other parties are adversaries who aim to minimize o. Formally, given a contract C and an objective o for party p, we define the value of contract as:

$$\mathsf{V}(C, o, p) := \sup_{\xi_p \in \Xi_p} \inf_{\xi_{-p} \in \Xi_{-p}} \mathbb{E}^{(\xi_p, \xi_{-p})} \left[\kappa(\rho, o, p)\right],$$

This corresponds to p trying to maximize the expected value of o and all other parties maliciously colluding to minimize it. In other words, it provides the worst-case guarantee for party p, irrespective of the behavior of the other parties, which in the worst-case is adversarial to party p.

3.4 Examples

One contribution of our work is to present the simplified programming language, and to show that this simple language can express several classical smart contracts. To demonstrate the applicability, we present several examples of classical smart contracts in this section. In each example, we present a contract and a "buggy" implementation of the same contract that has a different value. In Section 6 we show that our automated approach to analyze the contracts can compute contract values with enough precision to differentiate between the correct and the buggy implementation. All of our examples are motivated from well-known bugs that have happened in real life in Ethereum.

[2] We are also assuming, as in many programming languages, that TRUE = 1 and FALSE = 0.

Rock-Paper-Scissors. Let our contract be the one specified in Figure 2 and assume that we want to analyze it from the point of view of the issuer p. Also, let the objective function be $(p^{+} - p^{-} + 10 \cdot \texttt{AliceWon})$. Intuitively, this means that winning the rock-paper-scissors game is considered to have an additional value of 10, other than the spending and earnings. The idea behind this is similar to the case with chess tournaments, in which players not only win a prize, but can also use their wins to achieve better "ratings", so winning has extra utility.

A common bug in writing rock-paper-scissors is allowing the parties to move sequentially, rather than concurrently [29]. If parties can move sequentially and the issuer moves after `Bob`, then she can ensure a utility of 10, i.e. her worst-case expected reward is 10. However, in the correct implementation as in Figure 2, the best strategy for both players is to bid 0 and then Alice can win the game with probability 1/3 by choosing each of the three options with equal probability. Hence, her worst-case expected reward is 10/3.

Auction. Consider an open auction, in which during a fixed time interval everyone is allowed to bid for the good being sold and everyone can see others' bids. When the bidding period ends a winner emerges and every other participant can get their money back. Let the variable `HighestBid` store the value of the highest bid made at the auction. Then for a party p, one can define the objective as:

$$p^{+} - p^{-} + (\texttt{Winner==}p) \times \texttt{HighestBid}.$$

This is of course assuming that the good being sold is worth precisely as much as the highest bid. A correctly written auction should return a value of 0 to every participant, because those who lose the auction must get their money back and the party that wins pays precisely the highest bid. The contract in Figure 3 (left) is an implementation of such an auction. However, it has a slight problem. The function bid allows the winner to reduce her bid. This bug is fixed in the contract on the right.

Three-Way Lottery. Consider a three-party lottery contract issued by a party p. The other two players can sign up by buying tickets worth 1 unit each. Then each of the players is supposed to randomly and uniformly choose a nonce. A combination of these nonces produces the winner with equal probability for all three parties. If a person does not make a choice or pay the fees, she will certainly lose the lottery. The rules are such that if the other two parties choose the same nonce, which is supposed to happen with probability $\frac{1}{3}$, then the issuer wins. Otherwise the winner is chosen according to the parity of sum of nonces. This gives everyone a winning probability of $\frac{1}{3}$ if all sides play uniformly at random. However, even if one of the sides refuses to play uniformly at random, the resulting probabilities of winning stays the same because each side's probability of winning is independent of her own choice assuming that others are playing randomly. We assume that the issuer p has objective $p^{+} - p^{-}$. This is because the winner can take other players' money. In a bug-free contract we will expect

```
contract BuggyAuction {
map Bids[0,1000] = 0;
numeric HighestBid[0,1000] = 0;
id Winner = null;
numeric bid[0,1000] = 0;

function bid[1,10]
(payable bid : caller) {
   payout(caller, Bids[caller]);
   Bids[caller]=bid;
   if(bid>HighestBid)
   {
      HighestBid = bid;
      Winner = caller;
   }
}

function withdraw[11,20]()
{
  if(caller!=Winner)
  {
    payout(caller, Bids[caller])
    Bids[caller]=0;
  }
}}
```

```
contract Auction {
map Bids[0,1000] = 0;
numeric HighestBid[0,1000] = 0;
id Winner = null;
numeric bid[0,1000] = 0;

function bid[1,10]
(payable bid : caller) {
  if(bid<Bids[caller])
    return;
  payout(caller, Bids[caller]);
  Bids[caller]=bid;
  if(bid>HighestBid)
  {
    HighestBid = bid;
    Winner = caller;
  }
}

function withdraw[11,20]()
{
  if(caller!=Winner)
  {
  payout(caller, Bids[caller]);
  Bids[caller]=0;
  }
}}
```

Fig. 3: A buggy auction contract (left) and its fixed version (right).

the value of this objective to be 0, given that winning has a probability of $\frac{1}{3}$. However, the bug here is due to the fact that other parties can collude. For example, the same person might register as both players and then opt for different nonces. This will ensure that the issuer loses. The bug can be solved by ensuring one's probability of winning is $\frac{1}{3}$ if she honestly plays uniformly at random, no matter what other parties do. For more details about this contract see [19].

Token Sale. Consider a contract that sells *tokens* modeling some aspect of the real world, e.g. shares in a company. At first anyone can buy tokens at a fixed price of 1 unit per token. However, there are a limited number of tokens available and at most 1000 of them are meant to be sold. The tokens can then be transferred between parties, which is the subject of our next example. For now, Figure 4 (left) is an implementation of the selling phase. However, there is a big problem here. The problem is that one can buy any number of tokens as long as there is at least one token remaining. For example, one might first buy 999 tokens and then buy another 1000. If we analyze the contract from the point of view of a solo party p with objective $\texttt{balance}[p]$, then it must be capped by 1000 in a bug-free contract, while the process described above leads to a value of 1999. The fixed contract is in Figure 4 (right). This bug is inspired by a very similar real-world bug described in [52].

Token Transfer. Consider the same bug-free token sale as in the previous example, we now add a function for transferring tokens. An owner can choose a recipient and an amount less than or equal to her balance and transfer that many tokens to the recipient. Figure 5 (left) is an implementation of this concept. Taking the same approach and objective as above, we expect a similar result. However, there is again an important bug in this code. What happens if a party transfers tokens to herself? She gets free extra tokens! This has been fixed in the contract on the right. This example models a real-world bug as in [42].

```
contract BuggySale {
map balance[0,2000] = 0;
numeric remaining[0,2000] = 1000
numeric payment[0,2000] = 0;

function buy[1,10]
  (payable payment:caller)
{
  if(remaining<=0){
    payout(caller, payment);
    return;
  }
  balance[caller] += payment;
  remaining -= payment;
}}
```

```
contract Sale {
map balance[0,2000] = 0;
numeric remaining[0,2000] = 1000;
numeric payment[0,2000] = 0;

function buy[1,10]
  (payable payment:caller)
{
  if(remaining-payment<0){
    payout(caller, payment);
    return;
  }
  balance[caller] += payment;
  remaining -= payment;
}}
```

Fig. 4: A buggy token sale (left) and its fixed version (right).

Translation to Solidity. All aspects of our programming language are already present in Solidity, except for the global clock and concurrent interactions. The global clock can be modeled by the number of the current block in the blockchain and concurrent interactions can be implemented using commitment schemes. For more details see [19].

4 Bounded Analysis and Games

Since smart contracts can be easily described in our programming language, and programs in our programming language can be translated to Solidity, the main aim to automatically compute values of contracts (i.e., compute guaranteed payoff for parties). In this section, we introduce the bounded analysis problem for our programming language framework, and present concurrent games which is the underlying mathematical framework for the bounded analysis problem.

4.1 Bounded analysis

As is standard in verification, we consider the bounded analysis problem, where the number of parties and the number of function calls are bounded. In standard program analysis, bugs are often detected with a small number of processes, or a

```
contract BuggyTransfer {
map balance[0,2000] = 0;
numeric remaining[0,2000] = 1000;
numeric payment[0,2000] = 0;
numeric amount[0,2000] = 0;
numeric fromBalance[0,2000] = 0;
numeric toBalance[0,2000] = 0;
id recipient = null;

function buy[1,10]...

function transfer[1,10](
  recipient : caller
  amount : caller) {
    fromBalance = balance[caller];
    toBalance = balance[recipient];
    if(fromBalance<amount)
      return;
    fromBalance -= amount;
    toBalance += amount;
    balance[caller] = fromBalance;
    balance[recipient] = toBalance;
  }}
```

```
contract Transfer {
map balance[0,2000] = 0;
numeric remaining[0,2000] = 1000;
numeric payment[0,2000] = 0;
numeric amount[0,2000] = 0;


id recipient = null;

function buy[1,10]...

function transfer[1,10](
  recipient : caller
  amount : caller) {


    if(balance[caller]<amount)
      return;
    balance[caller] -= amount;
    balance[recipient] += amount;


  }}
```

Fig. 5: A buggy transfer function (left) and its fixed version (right).

small number of context switches between concurrent threads. In the context of smart contracts, we analogously assume that the number of parties and function calls are bounded.

Contracts with bounded number of parties and function calls. Formally, a contract with bounded number of parties and function calls is as follows:

- Let C be a contract and $k \in \mathbb{N}$, we define C_k as an equivalent contract that can have at most k parties. This is achieved by letting $\mathbb{P} = \{\mathbb{p}_1, \mathbb{p}_2, \ldots, \mathbb{p}_k\}$ be the set of all possible ids in the contract. The set $\mathbb{P}$ must contain all ids that are in the program source, therefore k is at least the number of such ids. Note that this does not restrict that ids are controlled by unique users, and a real-life user can have several different ids. We only restrict the analysis to bounded number of parties interacting with the smart contract.
- To ensure runs are finite, number of function calls by each party is also bounded. Specifically, each party can call each function at most once during each time frame, i.e. between two consecutive ticks of the clock. This closely resembles real-life contracts in which one's ability to call many functions is limited by the capacity of a block in the blockchain, given that the block must save all messages.

4.2 Concurrent Games

The programming language framework we consider has interacting agents that act simultaneously, and we have the program state. We present the mathematical framework of concurrent games, which are games played on finite state spaces with concurrent interaction between the players.

Concurrent Game Structures. A concurrent two-player game structure is a tuple $G = (S, s_0, A, \Gamma_1, \Gamma_2, \delta)$, where S is a finite set of states, $s_0 \in S$ is the start state, A is a finite set of actions, $\Gamma_1, \Gamma_2 : S \to 2^A \setminus \emptyset$ such that Γ_i assigns to each state $s \in S$, a non-empty set $\Gamma_i(s) \subseteq A$ of actions available to player i at s, and finally $\delta : S \times A \times A \to S$ is a transition function 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 $\delta(s, a_1, a_2) \in S$.

Plays and Histories. 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. We denote the set of all plays by $\mathscr{P}$. Every finite prefix $p[..r] := \left((s_0, a_1^0, a_2^0), (s_1, a_1^1, a_2^1), \ldots, (s_r, a_1^r, a_2^r)\right)$ of a play is called a *history* and the set of all histories is denoted by $\mathscr{H}$. If $h = p[..r]$ is a history, we denote the last state appearing according to h, i.e. $s_{r+1} = \delta(s_r, a_1^r, a_2^r)$, by $last(h)$. We also define $p[.. -1]$ as the empty history.

Strategies and Mixed strategies. A strategy is a recipe that describes for a player the action to play given the current game history. Formally, a strategy φ_i for player i is a function $\varphi_i : \mathscr{H} \to A$, such that $\varphi_i(h) \in \Gamma_i(last(h))$. A pair $\varphi = (\varphi_1, \varphi_2)$ of strategies for the two players is called a strategy profile. Each such φ induces a unique play. A mixed strategy $\sigma_i : \mathscr{H} \to \Delta(A)$ for player i given the history of the game. Intuitively, such a strategy suggests a distribution of actions to player i at each step and then she plays one of them randomly according to that distribution. Of course it must be the case that $Supp(\sigma_i(h)) \subseteq \Gamma_i(last(h))$. A pair $\sigma = (\sigma_1, \sigma_2)$ of mixed strategies for the two players is called a mixed strategy profile. Note that mixed strategies generalize strategies with randomization. Every mixed strategy profile $\sigma = (\sigma_1, \sigma_2)$ induces a unique probability measure on the set of plays, which is denoted as $\mathsf{Prob}^{\sigma}[\cdot]$, and the associated expectation measure is denoted by $\mathbb{E}^{\sigma}[\cdot]$.

State and History Utilities. In a game structure G, a state utility function u for player 1 is of the form $u : S \to \mathbb{R}$. Intuitively, this means that when the game enters state s, player 1 receives a reward of $u(s)$. State utilities can be extended to history utilities. We define the utility of a history to be the sum of utilities of all the states included in that history. Formally, if $h = \left(s_i, a_1^i, a_2^i\right)_{i=0}^{r}$, then $u(h) = \sum_{i=0}^{r} u(s_i)$. Given a play $p \in \mathscr{P}$, we denote the utility of its prefix of length L by $u_{\mathsf{L}}(p)$.

Games. 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's goal is to minimize it.

Values. The L-step finite-horizon value of a game (G, u) is defined as

$$\upsilon_{\mathsf{L}}(G, u) := \sup_{\sigma_1} \inf_{\sigma_2} \mathbb{E}^{(\sigma_1, \sigma_2)} \left[u_{\mathsf{L}}(p)\right], \tag{1}$$

where σ_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. The values of games can be computed using the value-iteration algorithm or dynamic programming, which is standard. A more detailed overview of the algorithms for games is provided in [19].

Remark 2. Note that in (1), limiting player 2 to pure strategies does not change the value of the game. Hence, we can assume that player 2 is an arbitrarily powerful nondeterministic adversary and get the exact same results.

4.3 Translating contracts to games

The translation from bounded smart contracts to games is straightforward, where the states of the concurrent game encodes the states of the contract. Correspondences between objects in the contract and game are as follows: (a) moves in contracts with actions in games; (b) run prefixes in contracts with histories in games; (c) runs in contracts with plays in games; and (d) policies (resp., randomized policies) in contracts with strategies (resp., mixed strategies) in games. Note that since all runs of the bounded contract are finite and have a limited length, we can apply finite horizon analysis to the resulting game, where L is the maximal length of a run in the contract. This gives us the following theorem:

Theorem 1 (Correspondence). *Given a bounded contract C_k for a party $\mathbb{p}$ with objective o, a concurrent game can be constructed such that value of this game, $\mathfrak{v}_{\mathsf{L}}(G, u)$, is equal to the value of the bounded contract, $\mathsf{V}(C_k, o, \mathbb{p})$.*

For details of the translation of smart contracts to games and proof of the theorem above see [19].

Remark 3. In standard programming languages, there are no parties to interact and hence the underlying mathematical models are graphs. In contrast, for smart contracts programming languages, where parties interact in a game-like manner, we have to consider games as the mathematical basis of our analysis.

5 Abstraction for Quantitative Concurrent Games

Abstraction is a key technique to handle large-scale systems. In the previous section we described that smart contracts can be translated to games, but due to state-space explosion (since we allow integer variables), the resulting state space of the game is huge. Hence, we need techniques for abstraction, as well as refinement of abstraction, for concurrent games with quantitative utilities. In this section we present such abstraction refinement for quantitative concurrent games, which is our main technical contribution in this paper. We show the soundness of our approach and its completeness in the limit. Then, we introduce a specific method of abstraction, called interval abstraction, which we apply to the games obtained from contracts and show that soundness and refinement are inherited from the general case. We also provide a heuristic for faster refining of interval abstractions for games obtained from contracts.

5.1 Abstraction for quantitative concurrent games

Abstraction considers a partition of the state space, and reduces the number of states by taking each partition set as a state. In case of transition systems (or graphs) the standard technique is to consider existential (or universal) abstraction to define transitions between the partition sets. However, for game-theoretic interactions such abstraction ideas are not enough. We now describe the key intuition for abstraction in concurrent games with quantitative objectives and formalize it. We also provide a simple example for illustration.

Abstraction idea and key intuition. In an abstraction the state space of the game (G, u) is partitioned into several abstract states, where an abstract state represents a set of states of the original game. Intuitively, an abstract state represents a set of similar states of the original game. Given an abstraction our goal is to define two games that can provide lower and upper bound on the value of the original game. This leads to the concepts of lower and upper abstraction.

- *Lower abstraction.* The lower abstraction $(G^{\downarrow}, u^{\downarrow})$ represents a lower bound on the value. Intuitively, the utility is assigned as minimal utility among states in the partition, and when an action profile can lead to different abstract states, then the adversary, i.e. player 2, chooses the transition.
- *Upper abstraction.* The upper abstraction $(G^{\uparrow}, u^{\uparrow})$ represents an upper bound on the value. Intuitively, the utility is assigned as maximal utility among states in the partition, and when an action profile can lead to different abstract states, then player 1 is chooses between the possible states.

Informally, the lower abstraction gives more power to the adversary, player 2, whereas the upper abstraction is favorable to player 1.

General abstraction for concurrent games. Given a game (G, u) consisting of a game structure $G = (S, s_0, A, \Gamma_1, \Gamma_2, \delta)$ and a utility function u, and a partition Π of S, the lower and upper abstractions, $(G^{\downarrow} = (S^{\mathsf{a}}, s_0^{\mathsf{a}}, A^{\mathsf{a}}, \Gamma_1^{\downarrow}, \Gamma_2^{\downarrow}, \delta^{\downarrow}), u^{\downarrow})$ and $(G^{\uparrow} = (S^{\mathsf{a}}, s_0^{\mathsf{a}}, A^{\mathsf{a}}, \Gamma_1^{\uparrow}, \Gamma_2^{\uparrow}, \delta^{\uparrow}), u^{\uparrow})$, of (G, u) with respect to Π are defined as:

- $S^{\mathsf{a}} = \Pi \cup D$, where $D = \Pi \times A \times A$ is a set of dummy states for giving more power to one of the players. Members of S^{a} are called abstracted states.
- The start state of G is in the start state of $G^{\uparrow}$ and $G^{\downarrow}$, i.e. $s_0 \in s_0^{\mathsf{a}} \in \Pi$.
- $A^{\mathsf{a}} = A \cup \Pi$. Each action in abstracted games either corresponds to an action in the original game or to a choice of the next state.
- If two states $s_1, s_2 \in S$, are in the same abstracted state $s^{\mathsf{a}} \in \Pi$, then they must have the same set of available actions for both players, i.e. $\Gamma_1(s_1) = \Gamma_1(s_2)$ and $\Gamma_2(s_1) = \Gamma_2(s_2)$. Moreover, s^{a} inherits these action sets. Formally, $\Gamma_1^{\downarrow}(s^{\mathsf{a}}) = \Gamma_1^{\uparrow}(s^{\mathsf{a}}) = \Gamma_1(s_1) = \Gamma_1(s_2)$ and $\Gamma_2^{\downarrow}(s^{\mathsf{a}}) = \Gamma_2^{\uparrow}(s^{\mathsf{a}}) = \Gamma_2(s_1) = \Gamma_2(s_2)$.
- For all $\pi \in \Pi$ and $a_1 \in \Gamma_1^{\downarrow}(\pi)$ and $a_2 \in \Gamma_2^{\downarrow}(\pi)$, we have $\delta^{\downarrow}(\pi, a_1, a_2) = (\pi, a_1, a_2) \in D$. Similarly for $a_1 \in \Gamma_1^{\uparrow}(\pi)$ and $a_2 \in \Gamma_2^{\uparrow}(\pi)$, $\delta^{\uparrow}(\pi, a_1, a_2) = (\pi, a_1, a_2) \in D$. This means that all transitions from abstract states in Π go to the corresponding dummy abstract state in D.
- If $d = (\pi, a_1, a_2) \in D$ is a dummy abstract state, then let $X_d = \{\pi' \in \Pi \mid \exists \quad s \in \pi \quad \delta(s, a_1, a_2) \in \pi'\}$ be the set of all partition sets that can be reached from π by a_1, a_2 in G. Then in $G^{\downarrow}$, $\Gamma_1^{\downarrow}(d)$ is a singleton, i.e., player 1

has no choice, and $\Gamma_2^{\downarrow}(d) = X_d$, i.e., player 2 can choose which abstract state is the next. Conversely, in $G^{\uparrow}$, $\Gamma_2^{\uparrow}(d)$ is a singleton and player 2 has no choice, while $\Gamma_1^{\uparrow}(d) = X_d$ and player 1 chooses the next abstract state.
- In line with the previous point, $\delta^{\downarrow}(d, a_1, a_2) = a_2$ and $\delta^{\uparrow}(d, a_1, a_2) = a_1$ for all $d \in D$ and available actions a_1 and a_2.
- We have $u^{\downarrow}(s^{\mathsf{a}}) = \min_{s \in s^{\mathsf{a}}}\{u(s)\}$ and $u^{\uparrow}(s^{\mathsf{a}}) = \max_{s \in s^{\mathsf{a}}}\{u(s)\}$. The utility of a non-dummy abstracted state in $G^{\downarrow}$, resp. $G^{\uparrow}$, is the minimal, resp. maximal, utility among the normal states included in it. Also, for each dummy state $d \in D$, we have $u^{\downarrow}(d) = u^{\uparrow}(d) = 0$.

Given a partition Π of S, either (i) there is no lower or upper abstraction corresponding to it because it puts states with different sets of available actions together; or (ii) there is a unique lower and upper abstraction pair. Hence we will refer to the unique abstracted pair of games by specifying Π only.

Remark 4. Dummy states are introduced for conceptual clarity in explaining the ideas because in lower abstraction all choices are assigned to player 2 and upper abstraction to player 1. However, in practice, there is no need to create them, as the choices can be allowed to the respective players in the predecessor state.

Example. Figure 6 (left) shows a concurrent game with (G, u) with 4 states. The utilities are denoted in red. The edges correspond to transitions in δ and each edge is labeled with its corresponding action pair. Here $A = \{\mathrm{a}, \mathrm{b}\}$, $\Gamma_1(s_0) = \Gamma_2(s_0) = \Gamma_2(s_1) = \Gamma_1(s_2) = \Gamma_2(s_2) = \Gamma_2(s_3) = A$ and $\Gamma_1(s_1) = \Gamma_1(s_3) = \{\mathrm{a}\}$. Given that action sets for s_0 and s_2 are equal, we can create abstracted games using the partition $\Pi = \{\pi_0, \pi_1, \pi_2\}$ where $\pi_1 = \{s_0, s_2\}$ and other sets are singletons. The resulting game structure is depicted in Figure 6 (center). Dummy states are shown by circles and whenever a play reaches a dummy state in $G^{\downarrow}$, player 2 chooses which red edge should be taken. Conversely, in $G^{\uparrow}$ player 1 makes this choice. Also, $u^{\uparrow}(\pi_0) = \max\{u(s_0), u(s_2)\} = 10, u^{\downarrow}(\pi_0) = \min\{u(s_0), u(s_2)\} = 0$ and $u^{\uparrow}(\pi_1)u^{\downarrow}(\pi_1) = u(s_1) = 10, u^{\uparrow}(\pi_2) = u^{\downarrow}(\pi_2) = u(s_3) = 0$. The final abstracted $G^{\downarrow}$ of the example above, without dummy states, is given in Figure 6 (right).

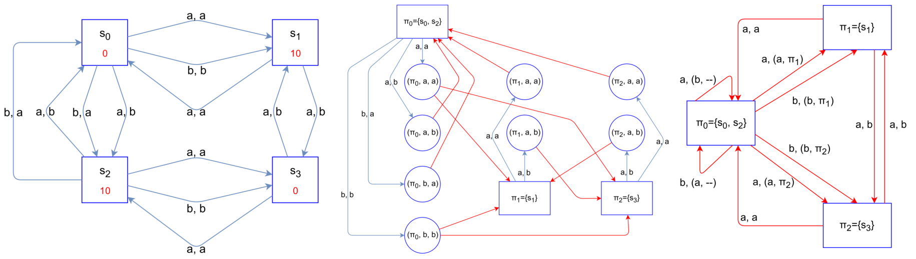

Fig. 6: An example concurrent game (left), abstraction process (center) and the corresponding $G^{\downarrow}$ without dummy states (right).

5.2 Abstraction: soundness, refinement, and completeness in limit

For an abstraction we need three key properties: (a) soundness, (b) refinement of the abstraction, and (c) completeness in the limit. The intuitive description is as follows: (a) soundeness requires that the value of the games is between the value of the lower and upper abstraction; (b) refinement requires that if the partition is refined, then the values of lower and upper abstraction becomes closer; and (c) completeness requires that if the partitions are refined enough, then the value of the original game can be approximated. We present each of these results below.

Soundness. Soundness means that when we apply abstraction, value of the original game must lie between values of the lower and upper abstractions. Intuitively, this means abstractions must provide us with some interval containing the value of the game. We expect the value of $(G^{\downarrow}, u^{\downarrow})$ to be less than or equal to the value of the original game because in $(G^{\downarrow}, u^{\downarrow})$, the utilities are less than in (G, u) and player 2 has more power, given that she can choose which transition to take. Conversely, we expect $(G^{\uparrow}, u^{\uparrow})$ to have a higher value than (G, u).

Formal requirement for Soundness. An abstraction of a game (G, u) leading to abstraction pair $(G^{\uparrow}, u^{\uparrow}), (G^{\downarrow}, u^{\downarrow})$ is sound if for every L, we have $\upsilon_{2\mathsf{L}}(G^{\downarrow}, u^{\downarrow}) \leq \upsilon_{\mathsf{L}}(G, u) \leq \upsilon_{2\mathsf{L}}(G^{\uparrow}, u^{\uparrow})$. The factor 2 in the inequalities above is due to the fact that each transition in the original game is modeled by two transitions in abstracted games, one to a dummy state and a second one out of it. We now present our soundness result.

Theorem 2 (Soundness, Proof in [19]). *Given a game (G, u) and a partition Π of its state space, if $G^{\uparrow}$ and $G^{\downarrow}$ exist, then the abstraction is sound, i.e. for all L, it is the case that $\upsilon_{2\mathsf{L}}(G^{\downarrow}, u^{\downarrow}) \leq \upsilon_{\mathsf{L}}(G, u) \leq \upsilon_{2\mathsf{L}}(G^{\uparrow}, u^{\uparrow})$.*

Refinement. We say that a partition Π_2 is a refinement of a partition Π_1, and write $\Pi_2 \sqsubseteq \Pi_1$, if every $\pi \in \Pi_1$ is a union of several π_i's in Π_2, i.e. $\pi = \bigcup_{i \in \mathcal{I}} \pi_i$ and for all $i \in \mathcal{I}$, $\pi_i \in \Pi_2$. Intuitively, this means that Π_2 is obtained by further subdividing the partition sets in Π_1. It is easy to check that $\sqsubseteq$ is a partial order over partitions. We expect that if $\Pi_2 \sqsubseteq \Pi_1$, then the abstracted games resulting from Π_2 give a better approximation of the value of the original game in comparison with abstracted games resulting from Π_1. This is called the refinement property.

Formal requirement for the Refinement Property. Two abstractions of a game (G, u) using two partitions Π_1, Π_2, such that $\Pi_2 \sqsubseteq \Pi_1$, and leading to abstracted games $(G_i^{\uparrow}, u_i^{\uparrow}), (G_i^{\downarrow}, u_i^{\downarrow})$ corresponding to each Π_i satisfy the refinement property if for every L, we have $\upsilon_{2\mathsf{L}}(G_1^{\downarrow}, u_1^{\downarrow}) \leq \upsilon_{2\mathsf{L}}(G_2^{\downarrow}, u_2^{\downarrow}) \leq \upsilon_{2\mathsf{L}}(G_2^{\uparrow}, u_2^{\uparrow}) \leq \upsilon_{2\mathsf{L}}(G_1^{\uparrow}, u_1^{\uparrow})$.

Theorem 3 (Refinement Property, Proof in [19]). *Let $\Pi_2 \sqsubseteq \Pi_1$ be two partitions of the state space of a game (G, u), then the abstractions corresponding to Π_1, Π_2 satisfy the refinement property.*

Completeness in the limit. We say that an abstraction is complete in the limit, if by refining it enough the values of upper and lower abstractions get as close together as desired. Equivalently, this means that if we want to approximate the value of the original game within some predefined threshold of error, we can do so by repeatedly refining the abstraction.

Formal requirement for Completeness in the limit. Given a game (G, u), a fixed finite-horizon L and an abstracted game pair corresponding to a partition Π_1, the abstraction is said to be complete in the limit, if for every $\epsilon \geq 0$ there exists $\Pi_2 \sqsubseteq \Pi_1$, such that if $(G_2^{\downarrow}, u_2^{\downarrow}), (G_2^{\uparrow}, u_2^{\uparrow})$ are the abstracted games corresponding to Π_2, then $\upsilon_{\mathsf{L}}(G_2^{\uparrow}, u_2^{\uparrow}) - \upsilon_{\mathsf{L}}(G_2^{\downarrow}, u_2^{\downarrow}) \leq \epsilon$.

Theorem 4 (Completeness in the Limit, Proof in [19]). *Every abstraction on a game (G, u) using a partition Π is complete in the limit for all values of L.*

5.3 Interval Abstraction

In this section, we turn our focus to games obtained from contracts and provide a specific method of abstraction that can be applied to them.

Intuitive Overview. Let (G, u) be a concurrent game obtained from a contract as in the Section 4.3. Then the states of G, other than the unique dummy state, correspond to states of the contract C_k. Hence, they are of the form $s = (t, b, l, val, p)$, where t is the time, b the contract balance, l is a label, p is the party calling the current function and val is a valuation. In an abstraction, one cannot put states with different times or labels or callers together, because they might have different moves and hence different action sets in the corresponding game. The main idea in interval abstraction is to break the states according to intervals over their balance and valuations. We can then refine the abstraction by making the intervals smaller. We now formalize this concept.

Objects. Given a contract C_k, let $\mathcal{O}$ be the set of all objects that can have an integral value in a state s of the contract. This consists of the contract balance, numeric variables and $m[\mathbb{p}]$'s where m is a map variable and $\mathbb{p}$ is a party. More precisely, $\mathcal{O} = \{\upbeta\} \cup N \cup \{m[\mathbb{p}] | m \in M, \mathbb{p} \in \mathbb{P}\}$ where $\upbeta$ denotes the balance. For an $o \in \mathcal{O}$, the value assigned to o at state s is denoted by o_s.

Interval Partition. Let C_k be a contract and (G, u) its corresponding game. A partition Π of the state space of G is called an interval partition if:

- The dummy state is put in a singleton set π_d.
- Each $\pi \in \Pi$ except π_d has associated values, $t_\pi, l_\pi, \mathbb{p}_\pi$ and for each $o \in \mathcal{O}$, $\overline{o}_\pi, \underline{o}_\pi$, such that $\pi = \{s \in S | s = (t_\pi, b, l_\pi, val, \mathbb{p}_\pi)$ and for all $o \in \mathcal{O},\ \ \underline{o}_\pi \leq s_o \leq \overline{o}_\pi\}$. Basically, each partition set includes states with the same time, label and caller in which the value of every object o is in an interval $[\underline{o}_\pi, \overline{o}_\pi]$.

We call an abstraction using an interval partition, an interval abstraction.

Refinement Heuristic. We can start with big intervals and continually break them into smaller ones to get refined abstractions and a finer approximation of the game value. We use the following heuristic to choose which intervals to break: Assume that the current abstracted pair of games are $(G^{\downarrow}, u^{\downarrow})$ and $(G^{\uparrow}, u^{\uparrow})$

corresponding to an interval partition Π. Let $d = (\pi_d, a_1, a_2)$ be a dummy state in $G^\uparrow$ and define the skewness of d as $\upsilon(G^\uparrow_d, u^\uparrow) - \upsilon(G^\downarrow_d, u^\downarrow)$. Intuitively, skewness of d is a measure of how different the outcomes of the games $G^\uparrow$ and $G^\downarrow$ are, from the point when they have reached d. Take a label l with maximal average skewness among its corresponding dummy states and cut all non-unit intervals of it in more parts to get a new partition Π'. Continue the same process until the approximation is as precise as desired. Intuitively, it tries to refine parts of the abstraction that show the most disparity between $G^\downarrow$ and $G^\uparrow$ with the aim to bring their values closer. Our experiments show its effectiveness.

Soundness and Completeness in the limit. If we restrict our attention to interval abstractions, soundness is inherited from general abstractions and completeness in the limit holds because Π_* is an interval partition. Therefore, using interval abstractions is both sound and complete in the limit.

Interval Refinement. An interval partition Π' is interval refinement of a given interval partition Π if $\Pi' \sqsubseteq \Pi$. Refinement property is inherited from general abstractions. This intuitively means that Π' is obtained by breaking the intervals in some sets of Π into smaller intervals.

Conclusion. We devised a sound abstraction-refinement method for approximating values of contracts. Our method is also complete in the limit. It begins by converting the contract to a game, then applies interval abstraction to the resulting game and repeatedly refines the abstraction using a heuristic until the desired precision is reached.

6 Experimental Results

Implementation and Optimizations. The state-space of the games corresponding to the smart contracts is huge. Hence the original game corresponding to the contract is computationally too expensive to construct. Therefore, we do not first construct the game and then apply abstraction, instead we first apply the interval abstraction, and construct the lower and upper abstraction and compute values in them. We optimized our implementation by removing dummy states and exploiting acyclicity using backward-induction. More details are provided in [19].

Experimental Results. We present our experimental results (Table 1) for the five examples mentioned in Section 3.4. In each of the examples, the original game is quite large, and the size of the state space is calculated without creating them. In our experimental results we show the abstracted game size, the refinement of games to larger sizes, and how the lower and upper bound on the values change. We used an Ubuntu machine with 3.2GHz Intel i7-5600U CPU and 12GB RAM.

Interpretation of the experimental results. Our results demonstrate the effectiveness of our approach in automatically approximating values of large games and real-world smart contracts. Concretely, the following points are shown:

- *Refinement Property.* By repeatedly refining the abstractions, values of lower and upper abstractions get closer at the expense of a larger state space.

Rock-Paper-Scissors						
Size	Abstractions					
$> 2.5 \cdot 10^{14}$	Correct Program			Buggy Variant		
	states	$[l\ ,\ u]$	time	states	$[l\ ,\ u]$	time
	19440	[0.00 , 10.00]	367	25200	[0.00 , 10.00]	402
	135945	[1.47 , 6.10]	2644	258345	[8.01 , 10.00]	4815
	252450	[1.83 , 5.59]	3381			

Auction						
Size	Abstractions					
$> 5.2 \cdot 10^{14}$	Correct Program			Buggy Variant		
	states	$[l\ ,\ u]$	time	states	$[l\ ,\ u]$	time
	3360	[0 , 1000]	68	2880	[0 , 1000]	38
	22560	[0 , 282]	406	27360	[565 , 1000]	552
	272160	[0 , 227]	4237	233280	[748 , 1000]	3780

Lottery						
Size	Abstractions					
$> 2.5 \cdot 10^{8}$	Correct Program			Buggy Variant		
	states	$[l\ ,\ u]$	time	states	$[l\ ,\ u]$	time
	1539	$[-1\ ,\ 1]$	17	1701	$[-1\ ,\ 1]$	22
	2457600	[0 , 0]	13839	2457600	$[-1\ ,\ -1]$	13244

Sale						
Size	Abstractions					
$> 4.6 \cdot 10^{22}$	Correct Program			Buggy Variant		
	states	$[l\ ,\ u]$	time	states	$[l\ ,\ u]$	time
	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					
$> 10^{23}$	Correct Program			Buggy Variant		
	states	$[l\ ,\ u]$	time	states	$[l\ ,\ u]$	time
	1040	[0 , 2000]	20	6561	[0 , 2000]	237
	32880	[844 , 1793]	562	131520	[1716 , 2000]	3979
	148311	[903 , 1352]	3740			

Table 1: Experimental results for correct and buggy contracts. $l := \upsilon(G^{\downarrow}, u^{\downarrow})$ denotes the lower value and $u := \upsilon(G^{\uparrow}, u^{\uparrow})$ is the upper value. Times are in seconds.

- *Distinguishing Correct and Buggy Programs.* Values of the lower and upper abstractions provide an approximation interval containing the contract value. These intervals shrink with refinement until the intervals for correct and buggy programs become disjoint and distinguishable.
- *Bug Detection.* One can anticipate a sensible value for the contract, and an approximation interval not containing the value shows a bug. For example, in token sale, the objective (number of tokens sold) is at most 1000, while results show the buggy program has a value between 1741 and 2000.
- *Quantification of Economic Consequences.* Abstracted game values can also be seen as a method to quantify and find limits to the economic gain or loss of a party. For example, our results show that if the buggy auction contract is deployed, a party can potentially gain no more than 1000 units from it.

7 Comparison with Related Work

Blockchain security analysis. The first security analysis of Bitcoin protocol was done by Nakamoto [43] who showed resilience of the blockchain against double-spending. A stateful analysis was done by Sapirshtein et al. [47] and by Sompolinsky and Zohar [49] in which states of the blockchain were considered. It was done using MDPs where only the attacker decides on her actions and the victim follows a predefined protocol. Our paper is the first work that is using two-player and concurrent games to analyze contracts and the first to use stateful analysis on arbitrary smart contracts, rather than a specific protocol.

Smart contract security. Delmolino et al. [29] held a contract programming workshop and showed that even simple contracts can contain incentive misalignment bugs. Luu et al. [41] introduced a symbolic model checker with which they could detect specific erroneous patterns. However the use of model checker cannot be extended to game-theoretic analysis. Bhargavan et al. [9] translated solidity programs to F^* and then used standard verification tools to detect vulnerable code patterns. See [7] for a survey of the known causes for Solidity bugs that result in security vulnerabilities.

Games and verification. Abstraction for concurrent games has been considered wrt qualitative temporal objectives [22, 44, 28, 3]. Several works considered concurrent games with only pure strategies [36, 37, 28]. Concurrent games with pure strategies are extremely restrictive and effectively similar to turn-based games. The min-max theorem (determinacy) does not hold for them even in special cases of one-shot games or games with qualitative objectives.

Quantitative analysis with games is studied in [12, 17, 21]. However these approaches either consider games without concurrent interactions or do not consider any abstraction-refinement. A quantitative abstraction-refinement framework has been considered in [18]; however, there is no game-theoretic interaction. Abstraction-refinement for games has also been considered [20, 36]; however, these works neither consider games with concurrent interaction, nor quantitative objectives. Moreover, [20, 36] start with a finite-state model without variables, and interval abstraction is not applicable to these game-theoretic frameworks. In contrast, our technical contribution is an abstraction-refinement approach for quantitative games and its application to analysis of smart contracts.

Formal methods in security. There is a huge body of work on program analysis for security; see [46, 1] for survey. Formal methods are used to create safe programming languages (e.g., [34, 46]) and to define new logics that can express security properties (e.g., [15, 6, 5]). They are also used to automatically verify security and cryptographic protocols, e.g., [2, 11] and [8] for a survey. However, all of these works aimed to formalize qualitative properties such as privacy violation and information leakage. To the best of our knowledge, our framework is the first attempt to use formal methods as a tool for reasoning about monetary loses and identifying them as security errors.

Bounded model checking (BMC). BMC was proposed by Biere et al. in 1999 [10]. The idea in BMC is to search for a counterexample in executions whose length is at most k. If no bug is found then one increases k until either a bug is found, the problem becomes intractable, or some pre-known upper bound is reached.

Interval abstraction. The first infinite abstract domain was introduced in [25]. This was later used to prove that infinite abstract domains can lead to effective static analysis for a given programming language [26]. However, none of the standard techniques is applicable to game analysis.

8 Conclusion

In this work we present a programming language for smart contracts, and an abstraction-refinement approach for quantitative concurrent games to automatically analyze (i.e., compute worst-case guaranteed utilities of) such contracts. This is the first time a quantitative stateful game-theoretic framework is studied for formal analysis of smart contracts. There are several interesting directions of future work. First, we present interval-based abstraction techniques for such games, and whether different abstraction techniques can lead to more scalability or other classes of contracts is an interesting direction of future work. Second, since we consider worst-case guarantees, the games we obtain are two-player zero-sum games. The extension to study multiplayer games and compute values for rational agents is another interesting direction of future work. Finally, in this work we do not consider interaction between smart contracts, and an extension to encompass such study will be a subject of its own.

Acknowledgments. The research was partially supported by Vienna Science and Technology Fund (WWTF) Project ICT15-003, Austrian Science Fund (FWF) NFN Grant No S11407-N23 (RiSE/SHiNE), and ERC Starting grant (279307: Graph Games).

References

1. Abadi, M.: Software security: A formal perspective. In: FM. (2012)
2. Abadi, M., Rogaway, P.: Reconciling two views of cryptography. In: Proceedings of the IFIP Conference on Theoretical Computer Science, Springer (2000) 3–22
3. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.Y.: Alternating refinement relations. In: CONCUR. (1998) 163–178
4. Anonymous Author: King of the ether. (2017) `www.kingoftheether.com`.
5. Arden, O., Liu, J., Myers, A.C.: Flow-limited authorization. In: CSF. (2015) 569–583
6. Arden, O., Myers, A.C.: A calculus for flow-limited authorization. In: CSF. (2016)
7. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts. IACR Cryptology ePrint Archive (2016) 1007
8. Avalle, M., Pironti, A., Sisto, R.: Formal verification of security protocol implementations: a survey. Formal Aspects of Computing **26**(1) (2014) 99–123
9. Bhargavan, K., et al.: Formal verification of smart contracts: Short paper. In: PLAS, ACM (2016)
10. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without bdds. TACAS (1999) 193–207
11. Blanchet, B., Chaudhuri, A.: Automated formal analysis of a protocol for secure file sharing on untrusted storage. In: SP, IEEE (2008) 417–431
12. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: CAV 2009. (2009) 140–156
13. Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J.A., Felten, E.W.: Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In: SP, IEEE (2015) 104–121
14. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.J.: Symbolic model checking: 1020 states and beyond. Information and Computation **98**(2) (1992)
15. Burrows, M., Abadi, M., Needham, R.M.: A logic of authentication. In: Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, The Royal Society (1989) 233–271
16. Buterin, V., et al.: Ethereum white paper. (2013)
17. Cerný, P., Chatterjee, K., Henzinger, T.A., Radhakrishna, A., Singh, R.: Quantitative synthesis for concurrent programs. In: CAV. (2011) 243–259
18. Cerný, P., Henzinger, T.A., Radhakrishna, A.: Quantitative abstraction refinement. In: POPL. (2013)
19. Chatterjee, K., Goharshady, A.K., Velner, Y.: Quantitative analysis of smart contracts. arXiv preprint arXiv:1801.03367 (2018)
20. Chatterjee, K., Henzinger, T.A., Jhala, R., Majumdar, R.: Counterexample-guided planning. In: UAI. (2005) 104–111
21. Chatterjee, K., Ibsen-Jensen, R.: Qualitative analysis of concurrent mean-payoff games. Information and Computation **242** (2015) 2–24
22. Church, A.: Logic, arithmetic, and automata. In: Proceedings of the International Congress of Mathematicians, Institut Mittag-Leffler (1962) 23–35
23. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
24. CoinMarketCap: Crypto-currency market capitalizations. (2017) `coinmarketcap.com`.
25. Cousot, P., Cousot, R.: Static determination of dynamic properties of generalized type unions. In: ACM Conference on Language Design for Reliable Software. Volume 12., ACM (1977) 77–94

26. Cousot, P., Cousot, R.: Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In: PLILP, Springer (1992) 269–295
27. Daian, P.: Analysis of the DAO exploit. (2016) `hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit`.
28. de Alfaro, L., Godefroid, P., Jagadeesan, R.: Three-valued abstractions of games: Uncertainty, but with precision. In: LICS, IEEE (2004)
29. Delmolino, K., Arnett, M., Kosba, A.E., Miller, A., Shi, E.: Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. IACR Cryptology ePrint Archive **2015** (2015) 460
30. Ethereum Foundation: Solidity language documentation. (2017)
31. Etherscan: Contract accounts. (2017) `etherscan.io/accounts/c`.
32. Etherscan: Token information. (2017) `etherscan.io/tokens`.
33. ETHNews: Hkg token has a bug and needs to be reissued. (2017) `ethnews.com/ethercamps-hkg-token-has-a-bug-and-needs-to-be-reissued`.
34. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: Automated security certification of android. Technical report (2009)
35. Godefroid, P., Van Leeuwen, J., Hartmanis, J., Goos, G., Wolper, P.: Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. Volume 1032. Springer Heidelberg (1996)
36. Henzinger, T.A., Jhala, R., Majumdar, R.: Counterexample-guided control. In: ICALP. (2003)
37. Henzinger, T.A., Majumdar, R., Mang, F., Raskin, J.F.: Abstract interpretation of game properties. In: SAS. (2000)
38. Jentzsch, C.: Decentralized autonomous organization to automate governance. (2016) `download.slock.it/public/DAO/WhitePaper.pdf`.
39. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4) (2009) 21:1–21:54
40. Johnson, N.: A beginner's guide to buying an ens domain (2017)
41. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: CCS. (2016) 254–269
42. Luu, L., Velner, Y.: Audit report for digix's smart contract platform. (2017)
43. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. (2008)
44. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL. (1989) 179–190
45. Queille, J., Sifakis, J.: Specification and verification of concurrent systems in cesar. In: International Symposium on programming, Springer (1982) 337–351
46. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1) (2003) 5–19
47. Sapirshtein, A., Sompolinsky, Y., Zohar, A.: Optimal selfish mining strategies in bitcoin. arXiv preprint arXiv:1507.06183 (2015)
48. Simonite, T.: $80 million hack shows the dangers of programmable money. (June 2016) `www.technologyreview.com`.
49. Sompolinsky, Y., Zohar, A.: Bitcoin's security model revisited. CoRR **abs/1605.09193** (2016)
50. Teutsch, J., Jain, S., Saxena, P.: When cryptocurrencies mine their own business? In: FC. (2016)
51. Toobin, A.: The DAO, ethereum's $150 million blockchain investment fund, has a logic problem. (2016) `www.inverse.com/article/16314-the-dao-ethereum-s-150-million-blockchain`.
52. Tran, V., Velner, Y.: Coindash audit report. (2017)
53. Wood, G.: Ethereum yellow paper. (2014)