Specification Mining for Smart Contracts with Automatic Abstraction Tuning

Florentin Guth
ETH Zurich, Switzerland
École Normale Supérieure, France
florentin.guth@ens.fr

Valentin Wüstholz
ETH Zurich, Switzerland
wuestholz@gmail.com

Maria Christakis
MPI-SWS, Germany
maria@mpi-sws.org

Peter Müller
ETH Zurich, Switzerland
peter.mueller@inf.ethz.ch

ABSTRACT

Smart contracts are programs that manage digital assets according to a certain protocol, expressing for instance the rules of an auction. Understanding the possible behaviors of a smart contract is difficult, which complicates development, auditing, and the post-mortem analysis of attacks.

This paper presents the first specification mining technique for smart contracts. Our technique extracts the possible behaviors of smart contracts from contract executions recorded on a blockchain and expresses them as finite automata. A novel dependency analysis allows us to separate independent interactions with a contract. Our technique tunes the abstractions for the automata construction automatically based on configurable metrics, for instance, to maximize readability or precision. We implemented our technique for the Ethereum blockchain and evaluated its usability on several real-world contracts.

1 INTRODUCTION

Smart contracts are programs that store and automatically move digital assets according to specified rules. While this idea was proposed over 20 years ago [61], it only gained traction when it was combined with blockchain technology to store an immutable record of all contract executions. Smart contracts have a wide range of applications, including fund raising, securities trading and settlement, supply-chain management, and electricity sourcing.

Despite their conceptual simplicity, understanding how to correctly interact with a smart contract is often challenging, not only for developers but also auditors. Specifically, legal invocations of contract operations typically need to satisfy implicit temporal ordering constraints, such as bidding only before an auction has ended. As an additional complication, contract executions may have subtle interactions (similar to data races) with each other when accessing the same state, especially since the smart-contract execution model provides no scheduling guarantees.

Specification mining [56] has been shown to help software engineers understand behaviors of complex systems. In this paper, we present the first application of specification mining to smart contracts. Our technique can be used to understand a contract of interest and its interactions with users and other smart contracts. We mine these specifications from contract executions recorded on a blockchain and express them as finite automata, describing the observed sequences of interactions with the contract.

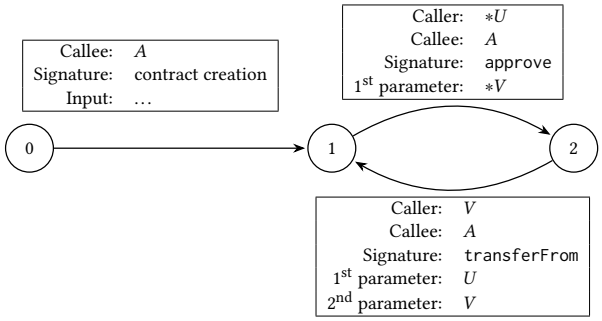

Figure 1: An automaton generated by our technique.

The automata produced by our technique not only characterize the protocol (API) for using a contract, but also shed light on temporal dependencies between different contract invocations that access the same state. This is useful for understanding the functionality of a smart contract, for debugging it, or even for a post-mortem analysis of an attack. For instance, a bug in the Parity wallet [3] allowed a function for setting the wallet owner to be called by anyone, even after the wallet was constructed. As a result, attackers managed to claim existing wallets by setting themselves as owners. In an automaton generated by our technique, the contract invocation for updating the wallet owner would appear as a transition that may be taken even after the wallet construction, which would help developers understand the attack.

Our approach goes beyond existing specification mining techniques for classical execution environments in two major ways. First, most techniques target interactions of sequential clients with a data structure [56]. In contrast, on a blockchain it is common that multiple clients interleave their interactions with a contract. To obtain precise specifications, our approach uses a novel dependency analysis to separate independent interactions with a contract and extract self-contained sequential traces from the blockchain.

Second, existing mining techniques use fixed, built-in abstractions to extract automata from sets of traces, e.g., by only considering method signatures. To support a wide range of use cases, our approach automatically adjusts its abstractions to maximize configurable metrics. These metrics can, for instance, favor coarse abstractions to obtain simple automata that are easy to grasp by a contract client, or precise abstractions to facilitate accurate understanding by a human auditor.

An example automaton that is generated by our technique is shown in Fig. 1. This automaton represents part of the functionality of the ERC20 token standard [1], an API that allows clients to create and manage their own crypto-currency using smart contracts. The edge labels of the automaton correspond to contract invocations. Specifically, function `approve` of contract instance A is invoked by user U to give user V permission to withdraw assets from U's account. Function `transferFrom` of the same contract instance A is invoked by V to withdraw assets from U's account after U's permission has been granted. This functionality is depicted in the automaton of Fig. 1 through relational abstractions that, for example, express that the first parameter of `transferFrom` needs to match the caller of `approve`. The notation $*U$ and $*V$ means that U and V are assigned fresh values. Although there may be multiple concrete instances of U and V, the automaton expresses that `transferFrom` still expects the same instance of U and V as in the previous invocation of `approve`, similarly to named capture groups in regular expressions.

The automaton is readable because our dependency analysis extracts self-contained interactions with the smart contract even though the blockchain contains many (partly overlapping) interactions. The abstractions applied here preserve relations between parameters of the above contract invocations, but omit other information, such as the value being approved and transferred or the return values of the invocations.

Our work makes the following technical contributions:

1. We present the first application of specification mining to smart contracts.
2. We propose a novel dependency analysis that handles multiple interleaving interactions with a smart contract to extract self-contained sequential traces (Sect. 4).
3. We present a novel automaton construction technique that automatically tunes its abstractions to optimize configurable metrics (Sect. 5).
4. We implemented our approach for the Ethereum blockchain and demonstrated its usefulness on several real-world contracts (Sect. 6).

2 BACKGROUND ON SMART CONTRACTS

A *blockchain* [54, 60, 62] is a decentralized consensus mechanism that was introduced by Bitcoin [10, 27, 50]. More specifically, a blockchain is a Byzantine fault-tolerant distributed database that is replicated across a peer-to-peer network of nodes and stores an ever-growing sequence of *blocks*, each uniquely identified by an increasing *block number*. A subset of the network nodes act as *miners*; they collect, in a block, a sequence of *transactions*, which are broadcast to the network but have not yet been stored in the blockchain. Transactions are created by user nodes of the network, for example, to transfer crypto-assets between different parties, and are communicated to the network. To earn the right to append a block to the blockchain, a miner needs to solve a mathematical challenge, in which case it permanently stores the block in the blockchain with a link to the previous block.

In the last few years, there have emerged several general-use, blockchain-based, distributed-computing platforms [13], the most popular of which is *Ethereum* [6]. Ethereum is open source [2], and its underlying crypto-currency is called *ether*. A key feature of Ethereum is its support for *contract accounts* in addition to *user accounts*. Like normal bank accounts, both contract and user accounts store a balance in ether and are owned by a user. Both types of accounts publicly reside on the Ethereum blockchain. A contract account, however, is not directly managed by users, but rather through code that is associated with it. Such code expresses contractual agreements between users, for instance, to implement and enforce an auction protocol. A contract account can also store persistent state (in a dictionary) that the code may access, for instance, to store auction bids. To better understand the process, imagine that a user issues a transaction with the auction contract account to place a bid. When this transaction is collected by a miner, the code of the contract account is automatically executed and the bid is recorded in the state of the account.

Contract accounts with their associated code and state are called *smart contracts*. The code is written in a Turing-complete bytecode, which is executed on the Ethereum Virtual Machine (EVM) [67]. Of course, programmers do not typically write EVM code. They can instead program in a variety of high-level languages, such as Solidity, Serpent, or Vyper, which compile to EVM bytecode.

3 GUIDED TOUR

In this section, we illustrate the workflow and architecture of our specification mining approach for smart contracts. Through a running example, we discuss the motivation behind the approach and the stages of our technique.

Example. Fig. 2 shows a smart contract, written in Ethereum's Solidity, that implements multiple, concurrent rock-paper-scissors games. The contract provides a public API consisting of functions `StartGame`, `Bet`, and `Claim`.

Function `StartGame` initializes a game (line 13), specifying a period of 4 blocks (via duration `d`), during which players may place their bets, and returns the identifier of the new game (line 15). Function `Bet` requires the player to specify a game identifier `gid`, their position `p` in the game (when `p` is 0, the player is requesting to be `pA`, i.e., player A, and when `p` is 1, they want to be `pB`), and their hand `h` (a hand of 1 is rock, 2 is paper, and 3 is scissors). Additionally, `Bet` is a payable function that expects players to pay the bet amount (line 20) within the first 4 blocks of the game (line 22). Function `Claim` allows players to claim their winnings for a period of 4 blocks following the betting period (lines 32–33), and transfers money to the winner (lines 36, 39, 42, 46, and 49). Line 34 disallows players from claiming their winnings multiple times by manipulating the start of claims such that, after the first claim, it appears that the claiming period is already over.

Workflow. Simple contracts such as our running example can be understood by reading the source code. However, smart contracts are typically much more complicated and their source code is often not available. To facilitate the understanding of even complex contracts—and thereby their development, use, and auditing—we mine specifications that characterize how contracts interact with users and other contracts. For a given smart contract (the so-called *target contract*), our technique determines these interactions by

```
contract RockPaperScissors {
  struct Game {
    address pA; address pB; // players
    uint hA; uint hB;       // hands
    uint cS;                // claim start
  }
  uint constant d = 4; uint constant amnt = 42;
  uint gC = 0;              // game count
  mapping(uint => Game) public games;

  function StartGame() public returns (uint) {
    gC++;
    var g = Game(0, 0, 0, 0, block.number + d);
    games[gC] = g;
    return gC;
  }

  function Bet(uint gid, uint p, uint h) public payable {
    require(0 < h && h < 4 && p < 2);
    require(msg.value == amnt);
    var g = games[gid];
    require(0 < g.cS && block.number < g.cS);
    if (g.hA == 0 && p == 0) {
      g.pA = msg.sender; g.hA = h;
    } else if (g.hB == 0 && p == 1) {
      g.pB = msg.sender; g.hB = h;
    } else { require(false); }
  }

  function Claim(uint gid) public {
    var g = games[gid];
    require(0 < g.cS && g.cS <= block.number);
    require(block.number < g.cS + d);
    g.cS = 0; // disallows multiple claims
    if (g.hA == 0 && g.hB != 0) { // no player A
      g.pB.transfer(amnt); return;
    }
    if (g.hB == 0 && g.hA != 0) { // no player B
      g.pA.transfer(amnt); return;
    }
    if (g.hA == g.hB) {           // draw
      g.pA.transfer(amnt); g.pB.transfer(amnt);
      return;
    }
    if (winningHand(g.hA, g.hB) == g.hB) {
      g.pB.transfer(2 * amnt); return;
    }
    if (winningHand(g.hA, g.hB) == g.hA) {
      g.pA.transfer(2 * amnt); return;
    }
  }
}
```

Figure 2: Running example written in Solidity.

mining actual executions that are recorded on the blockchain (either the actual Ethereum blockchain or in a testing environment), and describes them through a finite automaton. As shown in Fig. 3, this process consists of two main steps, which we explain next.

Mining histories. Our technique captures all interactions with the target contract starting from a given *seed transaction*, which contains an invocation of the contract. In practice, the seed transaction typically instantiates the target contract—similarly to how objects are instantiated in object-oriented programs—such that all interactions with the contract are captured. Our technique locates

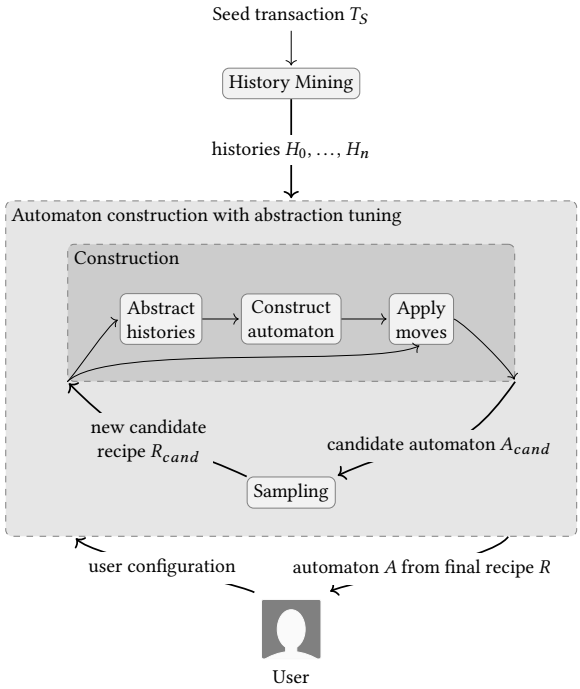

Figure 3: Overview of the workflow and tool architecture.

the seed transaction on the blockchain and collects all subsequent transactions. It then determines temporal dependencies among these transactions, resulting in a directed *dependency graph*. For example, a transaction T_2 temporally depends on a transaction T_1 if T_1 occurs before T_2 on the blockchain and the execution of T_1 affects the behavior of T_2, say, by writing a variable that is read by T_2. Transactions that (transitively) depend on the seed transaction should be captured in the generated automaton. All other transactions are unrelated to understanding the target contract. Tab. 1 shows a list of relevant transactions for the smart contract from Fig. 2, starting with the instantiation of the contract. As an example, observe that T_2 from Tab. 1 affects the behavior of T_5 by writing to location `games[g1]` that is read by T_5.

It is common that several clients interact simultaneously with an instance of a smart contract. In fact, the transactions in Tab. 1 come from four overlapping rock-paper-scissors games. In order to obtain precise specifications, it is important to separate these games into self-contained traces; otherwise, transactions T_4 to T_6 would, for instance, suggest that it is possible to call `Bet` three times in a row and modify the hand of a player after it has been set.

To obtain this separation, we use the dependency graph to cluster the relevant transactions into self-contained *sessions*. A session is the longest sequence of transactions such that the transactions in the session depend only on other (earlier) transactions in the same session and on the seed transaction. In our example, we identify a session for each of the four rock-paper-scissors games in Tab. 1.

The dependency graph and sessions are defined on the granularity of transactions since these are atomic operations on the blockchain. Each transaction may include several contract invocations. In order to obtain a precise automaton that also shows

Table 1: A sequence of transactions with the `RockPaperScissors` contract (of Fig. 2) starting from the contract-creation transaction.

Transaction Identifier	Contract Invocation	Block Number
1	contract creation	7
2	`g1 = StartGame()`	9
3	`g2 = StartGame()`	10
4	`Bet(g2, 1, 3)`	10
5	`Bet(g1, 0, 1)`	11
6	`Bet(g1, 1, 2)`	12
7	`Claim(g1)`	13
8	`g3 = StartGame()`	14
9	`Claim(g2)`	14
10	`Bet(g3, 0, 3)`	15
11	`Bet(g3, 1, 1)`	15
12	`g4 = StartGame()`	16
13	`Bet(g4, 1, 1)`	17
14	`Bet(g4, 0, 1)`	17
15	`Claim(g4)`	20
16	`Claim(g4)`	20

the contract interactions within a transaction, we decompose each transaction into its individual invocations. Performing this decomposition on a session yields a sequence of invocations, called *history*. In Tab. 1, each transaction contains a single contract invocation, so the transformation of sessions into histories is straightforward.

Constructing and optimizing the automaton. Once we have extracted a set of histories from the blockchain, we represent them as a finite automaton. Histories here are sequences of contract invocations, but our approach for constructing the automaton is more general; it works for sequences of arbitrary *events* such as contract invocations, internal function calls, or statement executions.

Each history can be represented trivially by an acyclic automaton that represents a sequence of n events by a chain of $n + 1$ states. In order to obtain more concise specifications, it is necessary to apply abstractions, which may merge different events and, thereby, facilitate the construction of smaller, cyclic automata.

Choosing suitable abstractions for the automaton construction is difficult for two reasons. First, it depends on the intended purpose of the generated automaton. A user of the target contract might prefer a simple automaton that is easy to read and, thus, favor coarse abstractions. On the other hand, an auditor might require an increased precision when examining more subtle contract interactions. Second, the space of possible abstractions is huge. For instance, even for each argument of an event (such as parameter and result values), there are numerous possible abstractions that strike different cost-benefit ratios.

Instead of using fixed abstractions, we address this challenge by optimizing the automaton according to a user-defined configuration. This configuration could, for instance, favor readability (e.g., by penalizing large automata) or precision (e.g., by penalizing large information loss). Our technique iteratively adjusts the applied abstractions to obtain a useful, or even optimal, result. This process is depicted by the outer gray box in Fig. 3 and explained below.

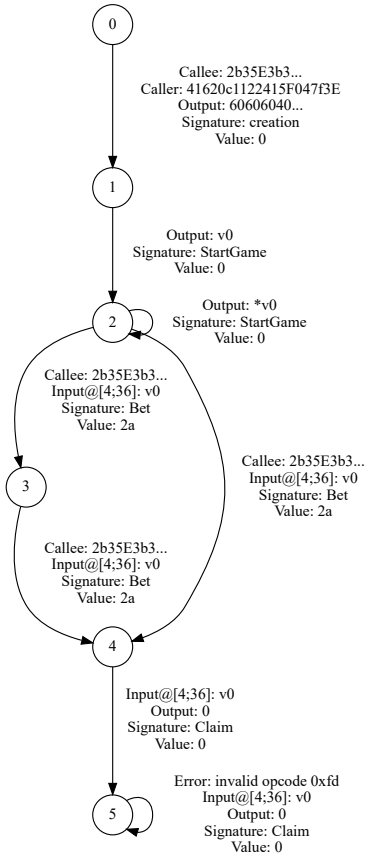

Figure 4: The output automaton for the `RockPaperScissors` contract (of Fig. 2), based on the transactions of Tab. 1.

The automaton construction proceeds in three steps (see inner gray box in Fig. 3). First, starting from the set of mined histories, we apply *event abstractions* to each event in the histories, resulting in a set of *abstract histories*. Event abstractions omit details of concrete events, for instance, they might ignore certain result values or abstract numerical parameters using standard abstract domains. Second, the resulting set of abstract histories is represented by a finite automaton. Third, this automaton is further simplified by applying *automaton moves*, which express local, automaton-specific abstractions such as merging two states.

To optimize the resulting automaton, we apply this construction process iteratively, as follows. The event abstractions and automaton moves are described by a so-called *recipe*; the initial recipe uses the identity event abstraction and contains no automata moves. In each iteration, we construct a candidate automaton A_{cand} according to the current candidate recipe. The subsequent sampling step then computes a cost for the candidate automaton A_{cand} based on a user-defined configuration and remembers the best automaton constructed so far. In search for an even better automaton, the sampling applies a random variation to the recipe and proceeds using this new candidate recipe. This process iterates until it reaches a user-defined exploration bound, and returns the best automaton so far. The user may then inspect the automaton and, if necessary,

restart the process with an adjusted configuration to obtain a more concise or a more detailed result.

For the running example, our tool chain produces the automaton of Fig. 4 using a configuration that strikes a good balance between readability and precision. This automaton is constructed based on the transactions of Tab. 1; the events appear as edge labels.

The automaton shows that multiple games may be started concurrently. The first `StartGame` event returns a variable v_0 that represents the game identifier. The notation $*v_0$ of the subsequent `StartGame` events expresses that each new game has a different identifier. Following the `StartGame` events, we have one or two consecutive invocations of `Bet` for a game v_0, along the right and left paths of the automaton, respectively. We then find a single successful `Claim` event for v_0. Here, the automaton shows that it is possible to invoke `Claim` after only one `Bet`; this functionality is needed to refund the bet amount if no second player participates in the game and emerges because of `g2` of Tab. 1. Observe that any further invocations of `Claim` for the same game v_0 result in an error, which is denoted by a self-loop in the automaton.

The automaton abstracts over the concrete result of `StartGame` and the first parameters of `Bet` and `Claim`. It also abstracts away the second and third parameters of `Bet`, while not abstracting the transferred amounts of Ether (e.g., value 42 (0x2a) for invocations of `Bet`). A characteristic of the automata that our technique generates is that each state is an accepting state, expressing that an interaction with the target contract may terminate at any state. For example, it is not necessary that a `StartGame` event is followed by an invocation of `Bet` and `Claim`. Our construction ensures that the resulting automaton over-approximates the histories extracted from the blockchain. That is, each of these histories is accepted by a run of the automaton.

In the following sections, we describe the main components of our architecture in more detail and explain precisely how we generate the automaton of Fig. 4.

4 MINING HISTORIES

The process of extracting histories from the blockchain starts by locating the given seed transaction T_S on the blockchain and collecting all subsequent transactions. These transactions are then processed in four major steps. First, to analyze the dependencies among these transactions, we determine the read and write effects for each of them (Sect. 4.1). Second, we use these effects to build a dependency graph, which describes temporal dependencies between transactions that access the same state (Sect. 4.2). Third, we remove any transactions that are unrelated to T_S and, consequently, to the target contract (Sect. 4.3). Fourth, we identify independent sessions and then decompose the transactions of each session to generate the histories of events that should be represented in the constructed automaton (Sect. 4.4).

4.1 Collecting transaction effects

The automata we produce summarize those transactions on the blockchain that are related to a given seed transaction; other transactions are irrelevant and should be omitted to obtain concise specifications. Two transactions are related if they access the same memory locations. Consequently, our analysis starts by collecting the read and write effects of the seed and all subsequent transactions. For this purpose, we execute each transaction on a local copy of the blockchain and use hooks into the virtual machine to record all memory accesses. This process obtains precise results, but is time consuming. However, it needs to be performed only once for each transaction on the blockchain; the results can be cached and extended incrementally as the blockchain grows. Moreover, it is often useful to mine only a small portion of the blockchain, for instance, when an auction lasts at most one week.

In general a location is any persistent state that a transaction may access. However, many transactions access the balance of accounts. These accesses introduce relationships between transactions that are otherwise unrelated and should be ignored in the mined specifications. For instance, they relate a game a user plays with an investment they make. To avoid such spurious connections, we exclude account balances from the locations we consider in the following. Technically, this means that we (slightly) under-approximate the effects of each transaction.

Definition 4.1 (Location). A *location* $l \in \mathcal{L}$ is any persistent state that the execution of a transaction may access (i.e., read or write) and that is not the balance of an account.

A location may, for instance, be an index to the persistent dictionary of a smart contract or an attribute of the block in which a transaction is contained, e.g., the block number or timestamp. For instance, in the running example, `g.pA` and **`block.number`** are locations accessed by transaction T_5 (from Tab. 1). Using the above definition, we can now define the effects of a transaction.

Definition 4.2 (Write effect). The *write effect* $w(T) \subseteq \mathcal{L}$ of a transaction T is the set of locations that are written to during execution of T.

For instance, the write effect $w(T_2)$ contains locations `gC`, `g.pA`, `g.pB`, `g.hA`, `g.hB`, `g.cS`, and `games[gC]`.

Definition 4.3 (Read effect). The *read effect* $r(T) \subseteq \mathcal{L}$ of a transaction T is the set of locations that are read from during execution of T, *without having previously been written to by* T.

For instance, the read effect $r(T_6)$ contains `games[gid]`, `g.cS`, **`block.number`**, `g.hA`, and `g.hB`. Since we will use effects to determine dependencies between different transactions, Def. 4.3 ignores reads from a location that is previously written to by the same transaction. Consequently, if $l \in w(T) \cap r(T)$, then a read from l occurs before a write to l during execution of T, e.g., `g.hB` $\in w(T_6) \cap r(T_6)$.

A transaction may read attributes of the block in which it is contained (such as the block number). These attributes are updated automatically when a block is appended to the blockchain. In order to reflect these implicit write operations, our technique inserts a *ghost transaction* right before the first transaction of every block. The write effect of this ghost transaction contains all block attributes. For instance, the ghost transaction for block 11, denoted B_{11}, writes to location **`block.number`**, which is then read by T_5.

4.2 Building the dependency graph

We use read and write effects to compute two kinds of dependencies between transactions, called strong and weak dependencies.

Strong dependencies are used to determine the relevant transactions, that is, the transactions that are related to the seed transaction and, therefore, need to be reflected in the constructed automaton. Weak dependencies help define the order in which the relevant transactions may occur.

Definition 4.4 (Strong dependency). A transaction T_j *strongly depends* on transaction T_i, denoted by $T_i \longrightarrow T_j$, if and only if (1) T_i occurs before T_j on the blockchain and (2) T_i writes to a location that T_j reads, and no transaction between T_i and T_j writes to that location: $(w(T_i) \cap r(T_j)) \setminus B \neq \emptyset$, where $B = \bigcup_{k=i+1,\ldots,j-1} w(T_k)$.

Intuitively, a strong dependency reflects that the execution of T_i affects the execution of T_j. Consequently, it indicates that T_i should be included in the constructed automaton if T_j is (we will explain the details in the next subsection). Moreover, if they are included, the automaton must reflect that T_i occurs before T_j.

A weak dependency alone does not determine which transactions to include in the automaton, but expresses an ordering constraint that needs to be reflected; changing the order of two weakly dependent transactions may influence the result of the execution.

Definition 4.5 (Weak dependency). A transaction T_j *weakly depends* on transaction T_i, denoted by $T_i \dashrightarrow T_j$, if and only if (1) T_i occurs before T_j on the blockchain and (2) T_i writes to or reads from a location that T_j writes, and no transaction between T_i and T_j writes to that location: $(w(T_i) \cap w(T_j)) \setminus B \neq \emptyset$ or $(r(T_i) \cap w(T_j)) \setminus B \neq \emptyset$, where B is defined as above.

We refer to strong and weak dependencies as *temporal dependencies*; we use them to build a transaction dependency graph:

Definition 4.6 (Dependency graph). A *dependency graph* is a directed graph, in which each node represents a transaction and each edge a strong or weak dependency between two transactions.

The dependency graph generated by our tool chain for the running example is shown in Fig. 5. Notice that there is a strong dependency $T_5 \longrightarrow T_6$ because function `Bet` of T_5 writes to `g.hA`, which is read by `Bet` of T_6. However, there is a weak dependency $T_{13} \dashrightarrow T_{14}$ since `Bet` of T_{13} reads from `g.hA`, which is written to by `Bet` of T_{14}. There is also a weak dependency $T_9 \dashrightarrow B_{15}$ (the ghost transaction that is inserted by our technique right before the first transaction of block 15), which indicates that function `Claim` of T_9 reads from location `block.number`, which is written to by B_{15}.

4.3 Filtering irrelevant transactions

Intuitively, we consider a transaction to be relevant if: (a) It is (directly or transitively) affected by the execution of the seed transaction, for example, a bid in an auction that was started by the seed transaction. (b) It occurs after the seed transaction and it (directly or transitively) affects the execution of an already relevant transaction according to case (a), for instance, the transaction that initializes the minimum price of an auctioned item before a bid is placed in an auction that was started by the seed transaction. In order to formalize this intuition, we first define transitive dependencies.

Definition 4.7 (Strong dependency path). A *strong dependency path* from transaction T_i to T_j is denoted $T_i \longrightarrow^* T_j$ and represents a non-empty path from T_i to T_j consisting only of strong dependencies.

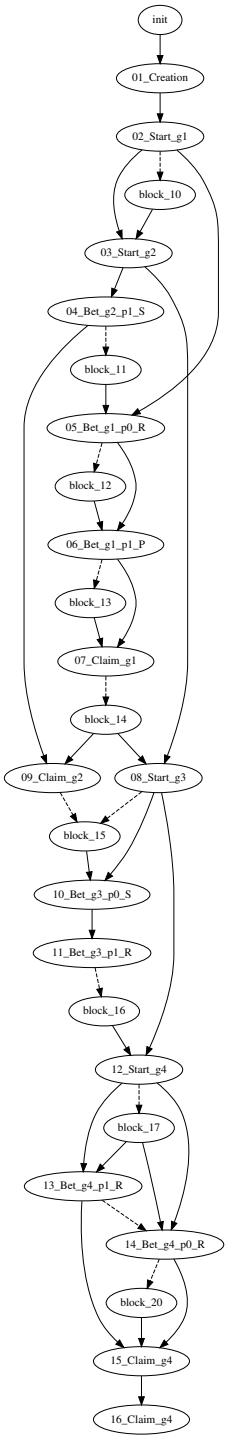

Figure 5: The dependency graph for the `RockPaperScissors` contract (of Fig. 2), generated from the transactions of Tab. 1. Each (non-ghost) node in the graph is annotated with the corresponding transaction identifier followed by the contract invocation with its arguments. The graph does not show edges between nodes when these are implied by transitivity of the dependencies.

Definition 4.8 (Weak dependency path). A *weak dependency path* from transaction T_i to T_j is denoted $T_i \dashrightarrow^* T_j$ and represents a non-empty path from T_i to T_j consisting of any number of strong dependencies and at least one weak dependency.

We remove all irrelevant transactions from the dependency graph as described by the following definition.

Definition 4.9 (Filtered dependency graph). Given a dependency graph G, a *filtered dependency graph* is generated from G by removing any transaction T that does not fulfill either of the following two conditions:

a. There exists a strong dependency path $T_S \rightarrow^* T$.
b. There exists a weak dependency path $T_S \dashrightarrow^* T$, and there exists a transaction T' such that there exist strong dependency paths $T \rightarrow^* T'$ and $T_S \rightarrow^* T'$.

This definition closely matches the intuition given at the beginning of this subsection. The requirement of a weak dependency path in case (b) is motivated by the fact that without at least a weak path, transactions T_S and T would be unordered. If they were unordered, T might as well have occurred *before* the seed transaction. Therefore, considering T to be a relevant transaction would contradict the role of the seed transaction as the "beginning of time". However, this requirement could in principle be dropped if a more relaxed interpretation of the seed transaction was desired.

In the dependency graph of Fig. 5, we do not filter out the ghost transaction B_{11} because it satisfies case (b) of the above definition: It has a weak dependency on the seed T_1 and the strong dependency paths $B_{11} \rightarrow T_5$ and $T_1 \rightarrow^* T_5$. If `Bet` was not required to be invoked within the betting period (line 22 of Fig. 2), there would be no edge between B_{11} and T_5, in which case transaction B_{11} would be filtered out. As another example, imagine that we create a second instance of the rock-paper-scissors contract. Even if the same players participated in games of both contracts, the accessed locations would be disjoint because we exclude account balances when determining read and write effects. As a result, all transactions with the second instance of the contract would not depend on the seed T_1 and would be removed.

4.4 Generating histories of events

In the final phase of the history mining, our technique traverses the filtered dependency graph to cluster its transactions into self-contained sessions. The transactions in these sessions are then decomposed to obtain histories of events. The events (that is, contract invocations) are collected when our technique executes the transactions on a local copy of the blockchain to compute their read and write effects (see Sect. 4.1).

Sessions represent independent interactions with a contract, for instance, different rock-paper-scissors games. Since strong dependencies indicate that two transactions are related by influencing each other's execution, strongly dependent transactions belong to the same session:

Definition 4.10 (Session). A *session* is the longest sequence of transactions that have a strong dependency only on transactions in the same session.

Intuitively, each transaction that is a sink in the graph with respect to strong dependencies (called *final transaction* below) can be placed in a separate session because no other transaction strongly depends on it. A session containing a final transaction T_F contains all transactions T such that $T \rightarrow^* T_F$.

Sessions are sequences of transactions, that is, ordered. The mined specification should reflect all possible orderings of the transactions in a session that are consistent with the executions extracted from the blockchain. These orderings are reflected by the strong and weak dependencies in the filtered dependency graph. We capture them by introducing potentially multiple sessions per final transaction: one for each topological ordering of the transactions.

For instance, there are four final transactions in the dependency graph of Fig. 5, namely T_7, T_9, T_{11}, and T_{16}, giving rise to the following four sessions. Note that the sessions for different final transactions overlap, at least in the seed transaction.

$$
\begin{aligned}
S_1 &:= [T_1, T_2, B_{11}, T_5, B_{12}, T_6, B_{13}, T_7] \\
S_2 &:= [T_1, T_2, B_{10}, T_3, T_4, B_{14}, T_9] \\
S_3 &:= [T_1, T_2, B_{10}, T_3, B_{14}, T_8, B_{15}, T_{10}, T_{11}] \\
S_4 &:= [T_1, T_2, B_{10}, T_3, B_{14}, T_8, B_{16}, T_{12}, B_{17}, T_{13}, T_{14}, B_{20}, T_{15}, T_{16}]
\end{aligned}
$$

Although weak dependencies are not considered when computing which transactions should be included in a session, they are useful in determining the ordering of transactions in sessions. For instance, when computing session S_1, observe that both B_{12} and T_5 have a strong dependency path to T_6, but only the weak dependency from T_5 to B_{12} reflects in which order these transactions occur. Ignoring such weak dependencies would lead to more sessions, some of which might correspond to invalid executions.

It might be tempting to restrict the order of transactions in a session to exactly the order in which they occur on the blockchain. However, this approach would lead to sessions that reflect the accidental ordering of unrelated transactions and, thus, miss other orderings that would have been possible and, therefore, should be shown in the mined specification.

Even though transactions are the atomic units of execution on the blockchain, it is more informative to express specifications of smart contracts on the level of individual events, in particular, contract invocations. For this purpose, we decompose the transactions in each session into its constituent events to obtain a set of histories:

Definition 4.11 (History). A *history* is a sequence of events that occur during the execution of a session.

Note that we perform the filtering, clustering, and ordering on the level of transactions and only then decompose the transactions into events. Since the events within one transaction are typically strongly dependent, performing the processing on the level of individual events would not increase the precision of our specifications, but lead to a much higher computational effort.

For the running example, we generate four histories of events by replacing each transaction in the above sessions with the corresponding contract invocation (from Tab. 1). Note that a ghost transaction does not contain any contract invocations.

5 AUTOMATON CONSTRUCTION WITH AUTOMATIC ABSTRACTION TUNING

Finding suitable abstractions to apply during the automaton construction is difficult. As discussed earlier, not only is the space of abstractions huge, but also the suitability of an abstraction depends

Algorithm 1 Automaton construction with abstraction tuning.

```
1  procedure ConstructAndTuneAutomaton(H_0, ..., H_n)
2    R_cand ← IdentityEventAbstraction()
3    C_opt, C_lst ← ∞
4    R_opt, R_lst ← R_cand
5    while ¬BoundReached() do
6      A_cand ← ApplyAutomatonRecipe(H_0, ..., H_n, R_cand)
7      C_cand ← ComputeAutomatonCost(A_cand)
8      if C_cand < C_opt then
9        C_opt, R_opt ← C_cand, R_cand
10     if Accept(C_cand, C_lst) then
11       C_lst, R_lst ← C_cand, R_cand
12     R_cand ← ModifyRecipe(R_lst)
13   return ApplyAutomatonRecipe(H_0, ..., H_n, R_opt)
14 procedure ApplyAutomatonRecipe(H_0, ..., H_n, R)
15   H'_0, ..., H'_n ← AbstractHistories(H_0, ..., H_n, R)
16   A_tmp ← BuildAutomaton(H'_0, ..., H'_n)
17   return ApplyAutomatonMoves(A_tmp, R)
```

on the intended use of the automaton. To address these challenges, our technique automatically tunes its abstractions to maximize the user-defined configuration, like readability or precision.

Algorithm. Alg. 1 presents the automaton construction with abstraction tuning. As shown in Fig. 3, it takes as input a set of concrete histories and builds a candidate automaton A_{cand} from the input histories and a candidate recipe R_{cand}. Recall from Sect. 3 that a recipe is a sequence of event abstractions and automaton moves. Event abstractions and automaton moves are applied as pre- and post-processing steps of the actual automaton construction, respectively (see procedure ApplyAutomatonRecipe). The actual construction is straightforward and results in a tree-shaped automaton that precisely characterizes all (abstracted) input histories. Cycles are introduced later, when automaton moves are applied. The initial recipe consists only of the identity event abstraction.

Once the initial automaton has been constructed, our algorithm tries to optimize it, according to a user-provided configuration. For this purpose, it applies random variations to the recipe, hoping to achieve a lower cost. A key insight of our algorithm is that these variations are not applied to the best recipe seen so far, but to a recent one. This approach allows the algorithm to explore recipes that might temporarily increase the cost of the candidate automaton but eventually lead the exploration away from a local minimum.

The algorithm keeps track of the current recipe (R_{cand}), the best recipe so far (R_{opt}), and the recipe used for the next random variation (R_{lst}), together with their associated costs. After computing a candidate automaton, the algorithm compares its cost to the best recipe so far and updates the information if a new best automaton has been found. It then decides which recipe to accept as a basis for the next iteration. R_{cand} is chosen if its cost is lower than the cost for the previous R_{lst}; even if the cost is higher, it is accepted with a certain probability that decreases proportionally to how much the cost increases as well as to how much time has elapsed. This process allows the algorithm to steer away from local minima, but also ensures that the exploration stabilizes over time. This iterative process, inspired by *simulated annealing* [35, 48], is stopped when an exploration bound is reached, such as the maximum number of iterations or a timeout. When stopped, the user is presented with the best automaton so far.

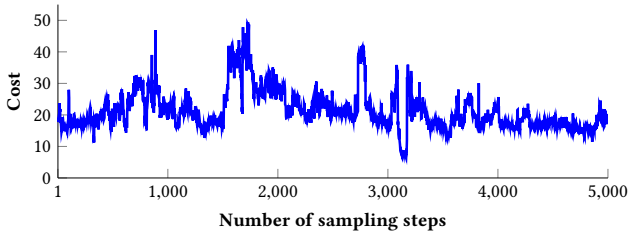

Figure 6: The cost of the automaton of Fig. 4 for the first 5000 sampling steps.

Fig. 6 plots the cost of the automaton of Fig. 4 over the number of sampling steps. Observe that our algorithm avoids several local minima by allowing the cost to increase up to ~50 before dropping to 6.7. As the number of steps increases, the cost is allowed to increase less. After 5,000 steps, the cost is ~20, which is higher than the best cost of 6.7 observed around step 3,100. As discussed earlier, our algorithm returns the automaton with the best cost.

Cost metrics. The metrics used by our algorithm are essentially a linear combination of readability, generality, and precision. In particular, it is possible to penalize automata with many states and edges to favor readability, to reward automata that describe more histories than those directly observed on the blockchain to favor generality, and to penalize automata that describe histories that were not observed on the blockchain to favor precision.

Fig. 7 shows an automaton generated by our technique for the rock-paper-scissors contract when the user configuration favors simplicity and generality over precision. The automaton still shows, for instance, that `Claim` events occur only after at least one `Bet` event. However, it now allows one or more bets instead of one or two, as in the more precise automaton from Fig. 4.

Event abstractions. Our technique represents events as records whose fields store information about their occurrence at execution time, for instance an `Output` field stores the return value of a contract invocation, as in Fig. 4. In order to obtain concise automata, we allow our algorithm to abstract from the details using the following event abstractions. A recipe determines the abstraction per function signature and event field.

- *Identity abstraction*: The value of the field is unchanged.
- *Variable abstraction*: The value is abstracted by a variable.
- *Top abstraction*: The value is abstracted away.

To obtain the automaton from Fig. 4, our technique applied, for instance, the variable abstraction on the `Output` field of all `StartGame` events. This means that the `Output` field of these events was each assigned a fresh variable. These variables were then replaced by a single variable v_0 using an automaton move, which we describe later. The top abstraction was applied on two input fields of all `Bet` events, namely the player position and their hand, which is why they are not shown in Fig. 4.

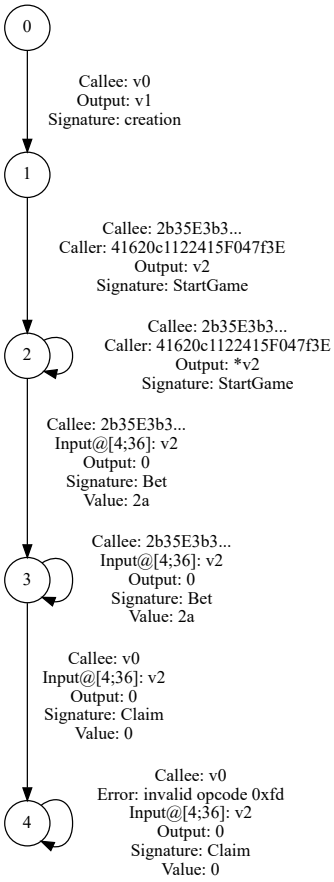

Figure 7: A general automaton for the contract of Fig. 2.

It is possible to apply other abstractions like the standard numerical abstractions of abstract interpretation or a byte-length abstraction that abstracts a value by its number of bytes, for example, to denote the order of magnitude of assets. For simplicity, we focus on the abstractions that we found most useful in our experiments.

Automaton moves. Event abstractions apply uniformly to all event fields for a given function signature, for instance, the result values of all invocations of `Bet`. It is also useful to apply abstractions more locally, depending on the context in the automaton. We call such automaton-specific abstractions *automaton moves*. We focus on two important automaton moves in the following, merging states and merging symbolic variables.

The automata constructed from abstract histories are tree-shaped and may include long chains of states. To obtain more readable automata, we merge states, which may join such chains and, more importantly, introduce cycles and substantially reduce the total number of states in an automaton.

Definition 5.1 (State merge). Given an automaton $\mathcal{A}$ and a function f from states to states, a *state merge* is a move that produces an automaton $\mathcal{A}'$, where every state q is replaced by $f(q)$ and every transition $q \xmapsto{e} r$ in $\mathcal{A}$ is replaced by $f(q) \xmapsto{e} f(r)$ in $\mathcal{A}'$.

Based on this definition, we present two instantiations of function f, which are based on the future of automaton states and are used by our technique:

- *Same bounded future*: Given a state q in an automaton $\mathcal{A}$, L_q denotes the set of words accepted by $\mathcal{A}$ starting from q. For $k \in \mathbb{N}$, L_q^k denotes the set of words that are prefixes of words in L_q with length at most k. We consider two automaton states q and r to have the same bounded future if $L_q^k = L_r^k$. To merge states with the same bounded future, our technique selects a random value for k and defines function f such that $f(q) = f(r)$ if and only if $L_q^k = L_r^k$.
- *Similar bounded future*: We consider two automaton states q and r to have a similar bounded future when $L_q^k \subset L_r^k$ or $L_r^k \subset L_q^k$. Our technique selects a random value for k and defines function f such that $f(q) = f(r)$ if and only if $L_q^k \subset L_r^k$ or $L_r^k \subset L_q^k$.

Note that both variations of this automaton move potentially make the resulting automaton very imprecise, for instance, if k is very small. However, the subsequent evaluation of the cost metric will detect such overly coarse abstractions and reject the recipe. In Fig. 4, state 2 is the result of merging several states that led to betting in the future.

As discussed above, an important event abstraction is to replace concrete values by symbolic variables. It is often useful to track relationships between different symbolic variables, especially, the equality of variables. Since we need to relate symbolic variables also across different histories, we cannot simply apply relational abstractions. Instead, we define an automaton move that expresses the equality of two symbolic variables by merging them.

Definition 5.2 (Variable merge). Given an automaton, a *variable merge* is an automaton move that:

a. selects two distinct symbolic variables, v_1 and v_2, that appear in event fields of the automaton;
b. replaces every occurrence of v_2 by v_1;
c. replaces v_1 by $*v_1$ in event fields where the variable is assigned a different value, according to the concrete histories.

The last step is necessary to ensure that each history of concrete events is still characterized by the corresponding path in the automaton. In our example, this move was repeatedly applied to yield the automaton of Fig. 4, in which all `StartGame` events return a unique game identifier and the subsequent `Bet` and `Claim` events refer to the same identifier.

6 EXPERIMENTAL EVALUATION

So far, we have shown the usability of our technique on the rock-paper-scissors contract. Here, we further evaluate its practicality on several real-world contracts deployed on the Ethereum blockchain.

Setup. For our experiments, we selected nine popular contracts from five categories that represent common applications of smart contracts. In particular, we chose contracts that implement auctions, gambling games, Ponzi schemes [12], tokens[1], and wallets. An overview of these contracts is shown in the first column of Tab. 2,

[1]Tokens allow clients to create and manage their own crypto-currency.

Table 2: Results for history mining.

Contract	LOC	Transactions		Histories	
		Total	Final	Total	Average Length
ENS	606	1,859,823	274	60	10
Esports	129	767,655	54	16	16
Etherdice	976	708,244	664	80	134
PiggyBank	89	628,585	150	123	71
PonziKing	–	1,163,028	45	7	14
BAT	175	26,276	69	30	61
REP	–	21,101	65	3	23
TheDao	1,236	3,535	188	33	81
EthDev	–	603,106	105	147	38

grouped by the above categories (in the same order). The second column shows the lines of code for each contract, where available.

For each contract, we identified an interesting seed transaction as input to our tool, in particular, a transaction that instantiates the contract or one that marks the beginning of interesting interactions, like the start of an auction. For all experiments, we set an exploration bound of 10,000 recipes.

We ran the experiments on an Intel Xeon CPU E5-4627 v2 @ 3.30GHz machine with 256GB of memory running the Ubuntu operating system with Linux 4.4 kernel.

History mining. In Tab. 2, we evaluate the history mining component of our tool chain. As shown in the table, the number of final transactions that remain after filtering the dependency graphs (fourth column) is only a small fraction of the total number of collected transactions when starting from the seed (third column). This shows the effectiveness of our technique in eliminating irrelevant transactions and, thereby, obtaining precise specifications. At the same time, the filtering retains sufficiently many transactions for our tool to generate several histories (fifth column) of reasonable average length (sixth column). On the other hand, the number of histories is still manageable even though there are histories that differ from each other only with respect to the order of their events.

As expected, the running time of the history mining is proportional to the time it takes to execute the transactions on our copy of the blockchain for collecting their effects. For the contracts shown in Tab. 2, this time ranges from a few seconds (e.g., 11s for TheDao) to several hours (e.g., 36h for ENS). On average, 50% of the total mining time is spent on executing the transactions and collecting their read and write effects, the other 50% on building the dependency graph, filtering it, and generating histories. The former part becomes more dominant as the number of transactions increases (e.g., 86% for ENS). As we explained earlier, it is possible to cache the results of this step and extend them incrementally as the blockchain grows.

Automaton construction and tuning. In our experiments, we used three user configurations: one that strikes a good balance between readability and precision (called the *default configuration*), one that favors generality (*general configuration*), and one that favors precision (*precise configuration*).

In Tab. 3, we evaluate the automaton construction and abstraction tuning component of our tool chain. Columns 2–5 of the table, labeled 'Transitions', show the number of transitions in the initial, default, general, and precise automata. As expected, the initial

Table 3: Results for automaton construction and tuning.

Contract	Transitions				Accepted Recipes			Cost Reduction		
	I	D	G	P	D	G	P	D	G	P
ENS	87	20	20	20	74.7%	65.7%	83.4%	81.8%	86.1%	66.5%
Esports	65	21	20	32	73.3%	68.9%	82.5%	65.4%	83.7%	62.9%
Etherdice	463	37	81	460	78.0%	55.8%	86.6%	88.4%	91.3%	68.0%
PiggyBank	121	27	30	34	71.6%	60.0%	82.6%	77.2%	87.4%	49.3%
PonziKing	24	24	24	24	68.6%	51.0%	82.3%	30.7%	10.0%	58.4%
BAT	253	54	39	52	73.6%	74.8%	82.3%	86.8%	92.9%	59.5%
REP	66	20	35	35	53.9%	83.3%	75.7%	80.4%	73.5%	70.5%
TheDao	456	84	47	99	70.5%	69.2%	77.2%	78.5%	88.5%	77.7%
EthDev	121	67	46	21	80.8%	75.5%	61.2%	82.4%	82.1%	91.2%

automata are too large to be readable by a user (with up to 464 transitions for Etherdice), whereas the automata generated by the default configuration of our tool contain an average of 39 transitions. We can see that the number of transitions typically varies for different configurations since the number of transitions is one of the factors in the cost computation. For instance, for TheDao, the general automaton contains the fewest number of transitions and the precise automaton contains the most. However, since we also consider other cost factors, the same does not always hold for other smart contracts.

Columns 6–8, labeled 'Accepted Recipes', show the percentage of recipes that are accepted by our optimization algorithm. Observe that, even in the worst case, 51% of the generated recipes are accepted (for PonziKing) and, therefore, contributed to the exploration; in the best case, this percentage goes up to 88.4% for Etherdice. Columns 9–11, labeled 'Cost Reduction', show how much the cost is reduced between the initial and the best automaton. For most contracts, the cost reduction is high (on average 74.6% for the default configuration), which suggests that our technique is able to effectively tune the automata based on the given cost configuration.

With a bound of 10,000 recipes, the running time of this tool component is between 7secs (for PonziKing) and 551mins (for Etherdice). The latter is, however, an outlier, which we attribute to the large number of histories and their average length (Tab. 2). The final automata for the majority of contracts and configurations are generated within less than ten minutes, despite the high exploration bound we selected. We observed that, even with a much smaller bound, the results are often comparable and generated within only a few seconds for most contracts. In general, most of the running time of this component is spent on computing a cost for each candidate automaton.

To illustrate the effectiveness of our approach, we describe our experience from generating the general automaton for the ENS auction. Even though the mining process starts out with almost two million transactions, it produces a concise automaton with only 20 transitions. This automaton correctly shows that auctions are started with an invocation of startAuctions. Bids may be placed by calling newBid or startAuctionsAndBid, which can start an auction and place a bid at the same time. Bids are then unsealed by invoking unsealBid. The automaton shows also that, after an invocation of unsealBid, no more bids are placed which ensures fairness of the auction. After the bids are unsealed, the auction may be finalized by calling finalizeAuction. We believe the automaton provides a convenient way to extract and visualize such information even if there are hundreds of different transactions.

7 RELATED WORK

Specification mining. The problem of specification mining is fundamental and well studied in the literature [56]. To the best of our knowledge, this work is the first to apply specification mining in the context of smart contracts. Early work on specification mining [16] phrases the problem as learning finite-state machines from sets of input/output pairs that partially describe their behavior. This early work presents the k-tails heuristic, which merges states in order to generalize from the given examples. In our setting, this heuristic corresponds to the same-future state merge.

There are approaches that aim to simplify mining by restricting what constitutes an automaton state [21, 53] or by defining discoverable patterns a priori [30–32], like the alternating pattern $(ab)^*$ (e.g., locking and unlocking resources) or the resource usage pattern ab^*c (e.g., opening, reading, and closing files). Although scalable, these approaches can miss interesting interactions between states.

More generally, work on specification mining may be classified into dynamic and static approaches. Like our technique, dynamic specification mining relies on having run the target program with reasonable coverage. Applications of dynamic mining include software revision histories [39], heap properties of object-oriented programs [22], component interactions [47], library APIs [36], system logs [14], scenario-based system behaviors [26, 41–43], etc.

Static approaches are further subdivided into component- and client-side mining. In component-side techniques, a specification is mined by analyzing a component's implementation [7, 49], whereas client-side techniques generate a specification that reflects usage patterns in a code-base [24, 46, 51, 59, 64, 65]. The two kinds of approaches have been shown to complement each other [66].

Specification mining approaches may also be classified into automaton-based (e.g., [20, 44]) and non-automaton-based ones (e.g., [23, 25, 37, 38, 55, 68]). Process mining [63] is a non-automaton-based technique, which dynamically records system events and mines workflow graphs. These graphs, which are similar to petri-nets, typically encode interactions of concurrent processes.

In contrast to existing work, our technique obtains precise specifications with a novel dependency analysis that allows us to extract sequential and self-contained traces from the blockchain. It also gives users the flexibility to adjust the generated automata according to their specific needs. To achieve this flexibility, we phrase specification mining as an optimization problem by computing a cost for each generated automaton and automatically tuning its abstractions. We also introduce a novel abstraction that allows us to capture relations between values occurring in different automaton events and fields, like the value of the game identifier in the running example. Our technique could easily be extended with additional metrics. For instance, there could be scenarios where it is useful to reduce rare state transitions by increasing their cost [9, 40].

Simulated annealing. The algorithm for simulated annealing was first described by Metropolis et al. [48], but it was better detailed many years later [19]. Although simulated annealing has been applied to optimization problems for a few decades already [35], it only recently started being integrated with program analysis techniques (e.g., [28, 29, 58]). We use simulated annealing for automatically tuning automata that describe the functionality of smart contracts.

Program analysis for smart contracts. Smart contract code is susceptible to bugs just like any other program, with the additional hazard of losing crypto-assets [11]. As a consequence, the program-analysis and verification community has already developed several bug-finding techniques for smart contracts, including debugging, static analysis, symbolic execution, and verification [4, 5, 8, 15, 17, 18, 33, 34, 45, 52]. As mentioned earlier, analyzing smart contracts poses further challenges due to their execution model. For instance, users have no direct control over the order in which transactions are processed by the miners, making smart contracts susceptible to concurrency bugs [57]. These characteristics had to be considered when designing our specification mining technique (e.g., by exploring all possible orderings of transactions when computing sessions).

8 CONCLUSION

We presented the first specification mining technique for smart contracts. Our precise specifications provide important insights into the functionality and interactions between smart contracts, and are useful to develop, understand, and audit smart contracts. Unlike existing work, our technique gives users the flexibility to adjust the generated automata based on their needs during specific usage scenarios. We achieve this by phrasing the problem of finding useful abstractions for automaton construction as an optimization problem with user-adjustable costs.

As future work, we plan to use our technique to identify common design patterns and evolve the language design of smart contracts. We also plan to apply the idea of cost-guided abstraction tuning to other forms of program analysis, such as heap analyses, invariant inference, and process mining.

REFERENCES

[1] [n. d.]. ERC20 Token Standard. https://theethereum.wiki/w/index.php/ERC20_Token_Standard.
[2] [n. d.]. Ethereum. https://github.com/ethereum.
[3] [n. d.]. Parity Wallet Security Alert. http://paritytech.io/security-alert/.
[4] [n. d.]. Remix—Solidity IDE. https://remix.readthedocs.io/en.
[5] [n. d.]. Securify. http://securify.ch.
[6] 2014. Ethereum White Paper. https://github.com/ethereum/wiki/wiki/White-Paper.
[7] Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. 2005. Synthesis of Interface Specifications for Java Classes. In *POPL*. ACM, 98–109.
[8] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *CPP*. ACM, 66–77.
[9] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining Specifications. In *POPL*. ACM, 4–16.
[10] Andreas M. Antonopoulos. 2014. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies.* O'Reilly Media.
[11] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts. In *POST (LNCS)*, Vol. 10204. Springer, 164–186.
[12] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. 2017. Dissecting Ponzi Schemes on Ethereum: Identification, Analysis, and Impact. *CoRR* abs/1703.03779 (2017).
[13] Massimo Bartoletti and Livio Pompianu. 2017. An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns. In *FC (LNCS)*, Vol. 10323. Springer, 494–509.
[14] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. 2013. Unifying FSM-Inference Algorithms through Declarative Specification. In *ICSE*. IEEE Computer Society, 252–261.
[15] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *PLAS*. ACM, 91–96.

[16] Alan W. Biermann and Jerome A. Feldman. 1972. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *TC* 21 (1972), 592–597. Issue 6.

[17] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Yaron Velner. 2018. Quantitative Analysis of Smart Contracts. In *ESOP (LNCS)*. Springer. To appear.

[18] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-Optimized Smart Contracts Devour your Money. In *SANER*. IEEE Computer Society, 442–446.

[19] Siddhartha Chib and Edward Greenberg. 1995. Understanding the Metropolis-Hastings Algorithm. *The American Statistician* 49 (1995), 327–335. Issue 4.

[20] Jonathan E. Cook and Alexander L. Wolf. 1998. Discovering Models of Software Processes from Event-Based Data. *TOSEM* 7 (1998), 215–249. Issue 3.

[21] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. 2006. Mining Object Behavior with ADABU. In *WODA*. ACM, 17–24.

[22] Brian Demsky and Martin C. Rinard. 2002. Role-Based Exploration of Object-Oriented Programs. In *ICSE*. ACM, 313–334.

[23] Mohammad El-Ramly, Eleni Stroulia, and Paul G. Sorenson. 2002. From Run-Time Behavior to Usage Scenarios: An Interaction-Pattern Mining Approach. In *KDD*. ACM, 315–324.

[24] Dawson R. Engler, David Yu Chen, and Andy Chou. 2001. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP*. ACM, 57–72.

[25] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.* 69 (2007), 35–45. Issue 1–3.

[26] Dirk Fahland, David Lo, and Shahar Maoz. 2013. Mining Branching-Time Scenarios. In *ASE*. IEEE Computer Society, 443–453.

[27] Pedro Franco. 2014. *Understanding Bitcoin: Cryptography, Engineering and Economics*. John Wiley and Sons.

[28] Zhoulai Fu and Zhendong Su. 2016. Mathematical Execution: A Unified Approach for Testing Numerical Code. *CoRR* abs/1610.01133 (2016).

[29] Zhoulai Fu and Zhendong Su. 2017. Achieving High Coverage for Floating-Point Code via Unconstrained Programming. In *PLDI*. ACM, 306–319.

[30] Mark Gabel and Zhendong Su. 2008. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *FSE*. ACM, 339–349.

[31] Mark Gabel and Zhendong Su. 2008. Symbolic Mining of Temporal Specifications. In *ICSE*. ACM, 51–60.

[32] Mark Gabel and Zhendong Su. 2010. Online Inference and Enforcement of Temporal Properties. In *ICSE*. ACM, 15–24.

[33] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. *PACMPL* 2 (2018), 48:1–48:28. Issue POPL.

[34] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *NDSS*. The Internet Society. To appear.

[35] Scott Kirkpatrick, C. Daniel Gelatt Jr., and Mario P. Vecchi. 1983. Optimization by Simulated Annealing. *Science* 220 (1983), 671–680. Issue 4598.

[36] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic Mining of Specifications from Invocation Traces and Method Invariants. In *FSE*. ACM, 178–189.

[37] Patrick Lam and Martin C. Rinard. 2003. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *ECOOP (LNCS)*, Vol. 2743. Springer, 275–302.

[38] David Lie, Andy Chou, Dawson R. Engler, and David L. Dill. 2001. A Simple Method for Extracting Models for Protocol Code. In *ISCA*. ACM, 192–203.

[39] Benjamin Livshits and Thomas Zimmermann. 2005. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In *ESEC/FSE*. ACM, 296–305.

[40] David Lo and Siau-Cheng Khoo. 2006. SMArTIC: Towards Building an Accurate, Robust and Scalable Specification Miner. In *FSE*. ACM, 265–275.

[41] David Lo and Shahar Maoz. 2008. Mining Scenario-Based Triggers and Effects. In *ASE*. IEEE Computer Society, 109–118.

[42] David Lo and Shahar Maoz. 2010. Scenario-Based and Value-Based Specification Mining: Better Together. In *ASE*. ACM, 387–396.

[43] David Lo, Shahar Maoz, and Siau-Cheng Khoo. 2007. Mining Modal Scenario-Based Specifications from Execution Traces of Reactive Systems. In *ASE*. ACM, 465–468.

[44] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2007. Towards Self-Protecting Enterprise Applications. In *ISSRE*. IEEE Computer Society, 39–48.

[45] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *CCS*. ACM, 254–269.

[46] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In *PLDI*. ACM, 48–61.

[47] Leonardo Mariani and Mauro Pezzè. 2007. Dynamic Detection of COTS Component Incompatibility. *IEEE Software* 24 (2007), 76–85. Issue 5.

[48] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. 1953. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics* 21 (1953), 1087–1092. Issue 6.

[49] Mangala Gowri Nanda, Christian Grothoff, and Satish Chandra. 2005. Deriving Object Typestates in the Presence of Inter-Object References. In *OOPSLA*. ACM, 77–96.

[50] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. 2016. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press.

[51] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-Based Mining of Multiple Object Usage Patterns. In *ESEC/FSE*. ACM, 383–392.

[52] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. *CoRR* abs/1802.06038 (2018).

[53] Michael Pradel and Thomas R. Gross. 2009. Automatic Generation of Object Usage Specifications from Large Method Traces. In *ASE*. IEEE Computer Society, 371–382.

[54] Siraj Raval. 2016. *Decentralized Applications: Harnessing Bitcoin's Blockchain Technology*. O'Reilly Media.

[55] Orna Raz, Philip Koopman, and Mary Shaw. 2002. Semantic Anomaly Detection in Online Data Sources. In *ICSE*. ACM, 302–312.

[56] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2013. Automated API Property Inference Techniques. *TSE* 39 (2013), 613–637. Issue 5.

[57] Ilya Sergey and Aquinas Hobor. 2017. A Concurrent Perspective on Smart Contracts. In *FC (LNCS)*, Vol. 10323. Springer, 478–493.

[58] Rahul Sharma and Alex Aiken. 2014. From Invariant Checking to Invariant Inference Using Randomized Search. In *CAV (LNCS)*, Vol. 8559. Springer, 88–105.

[59] Sharon Shoham, Eran Yahav, Stephen J. Fink, and Marco Pistoia. 651–666. Static Specification Mining Using Automata-Based Abstractions. *TSE* 34 (651–666), 2008. Issue 5.

[60] Melanie Swan. 2015. *Blockchain: Blueprint for a New Economy*. O'Reilly Media.

[61] Nick Szabo. 1996. Smart Contracts: Building Blocks for Digital Markets. http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.

[62] Don Tapscott and Alex Tapscott. 2016. *Blockchain Revolution: How the Technology Behind Bitcoin is Changing Money, Business, and the World*. Penguin.

[63] Wil M. P. van der Aalst, Boudewijn F. van Dongen, Joachim Herbst, Laura Maruster, Guido Schimm, and A. J. M. M. Weijters. 2003. Workflow Mining: A Survey of Issues and Approaches. *Data Knowl. Eng.* 47 (2003), 237–267. Issue 2.

[64] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting Object Usage Anomalies. In *ESEC/FSE*. ACM, 35–44.

[65] Westley Weimer and George C. Necula. 2005. Mining Temporal Specifications for Error Detection. In *TACAS (LNCS)*, Vol. 3440. Springer, 461–476.

[66] John Whaley, Michael C. Martin, and Monica S. Lam. 2002. Automatic Extraction of Object-Oriented Component Interfaces. In *ISSTA*. ACM, 218–228.

[67] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. http://gavwood.com/paper.pdf.

[68] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *ICSE*. ACM, 282–291.