Specification Mining for Smart Contracts with Trace Slicing and Predicate Abstraction

Ye Liu[1]
Nanyang Technological University
Singapore
li0003ye@e.ntu.edu.sg

Yixuan Liu
Nanyang Technological University
Singapore
liuy0255@e.ntu.edu.sg

Yi Li
Nanyang Technological University
Singapore
yi_li@ntu.edu.sg

Cyrille Artho
KTH Royal Institute of Technology
Sweden
artho@kth.se

***Abstract*—Smart contracts are computer programs running on blockchains to implement Decentralized Applications. The absence of contract specifications hinders routine tasks, such as contract understanding and testing. In this work, we propose a specification mining approach to infer contract specifications from past transaction histories. Our approach derives high-level behavioral automata of function invocations, accompanied by program invariants statistically inferred from the transaction histories. We implemented our approach as tool SMCON and evaluated it on eleven well-studied Azure benchmark smart contracts and six popular real-world DApp smart contracts. The experiments show that SMCON mines reasonably accurate specifications that can be used to enhance symbolic analysis of smart contracts achieving higher code coverage and up to 56 % speedup, and facilitate DApp developers in maintaining high-quality documentation and test suites.**

I. INTRODUCTION

Blockchain technology has developed rapidly in recent years, since the introduction of Bitcoin [1] by Nakamoto in 2008. Blockchain itself is a distributed ledger maintained and shared by a peer-to-peer (P2P) network, and it evolved into a platform which supports the deployment and execution of smart contracts, popularized by Ethereum [2]. Smart contracts are self-executing computer programs used to implement Decentralized Applications (DApps). Users interact with smart contracts by executing transactions on the blockchain. Ethereum, the most prominent smart contract platform, is empowering many DApps, spanning areas such as finance, health, governance, games, etc. [3]. As of May 2023, there are more than 50 million smart contracts deployed on Ethereum, and these smart contracts have supported 13,968 DApps [4], [3].

Despite the high stakes involved, smart contracts are often developed in an undisciplined way. The existence of bugs and vulnerabilities compromises the reliability and security of smart contracts and endangers the trust of users. Durieux et al. [5] reported that nearly 10% of the smart contracts may contain security vulnerabilities related to access controls. ERC-20 [6] is the most popular smart contract standard on Ethereum, yet 13% of the ERC-20 token contracts do not conform to the standard specification [7]. Moreover, Qin et al. [8] demonstrated how economic behavior models can be exploited to attack the DeFi ecosystem with flash loans. A major difficulty in validating the conformance of smart contracts, i.e., whether the contract implementation adheres to the expected behaviors, is the lack of documented formal specifications.

Formal specifications capture the expected contract behaviors, in terms of formal languages, based on a formal model [9] with precise semantics. Specifications of a smart contract play a central role in describing, understanding, reasoning about contract behaviors, and detecting, through testing and verification, non-conformance issues such as functional bugs and security vulnerabilities.

Similar to traditional formal specifications, two forms of smart contract specifications have been studied in past work: (1) function-level program invariants [10], which are used in testing [11], verification [12], [13], and runtime validation [14] of smart contracts; and (2) contract-level behavioral specifications in the form of automata [15], which can be used to support contract synthesis [16], model-based testing [17], design verification [18], and workflow verification [19]. Specifically, Wang et al. [19] performed workflow verification via semantic conformance checking between state machine-based workflow specifications and smart contracts from the *Azure Blockchain Workbench*, an enterprise blockchain from Microsoft.

In this paper, we focus on mining high-level automata-based specifications automatically for smart contracts. Many approaches have been proposed for this task on traditional program traces: for example, grammar inference techniques [20], [21], [22] and deep learning-based techniques [23] have been used to learn automata from a set of program execution traces. The k-tail algorithm and its variants [24], [25], [26] merge states if the same set of "tail" invocation sequences are observed.

However, the way smart contracts behave poses new challenges for mining automata-based behavioral models. As they are usually deployed on public blockchain networks, smart contracts handle multiple user interactions simultaneously. Therefore, the execution traces recorded in contract transaction

This work was done while Ye Liu was a student at Nanyang Technological University.

histories consist of interleaving events triggered by different user interactions and may belong to different sessions. Since there does not exist a standard approach for managing user sessions, the execution traces cannot be easily separated for independent interactions. Moreover, predicate abstraction is crucial in deriving compact but accurate automata. Yet, the choice of predicates remains challenging and is often tightly tied with the specific analysis tasks. The predicate abstraction techniques used in computing state abstractions must be tailored to take into account the specific data structures and runtime environments of smart contracts.

To mine more accurate automata specification efficiently for smart contracts, we propose a specification mining algorithm powered by trace slicing and predicate abstraction [27]. The contract specification mining process is preceded by a slicing of the transaction histories. We perform trace slicing on the transaction histories via a parametric binding learned from the existing test suites. A slice of history is a sequence of inter-related transactions, e.g., all transactions related to one specific trade session. Smart contract transaction histories, being stored persistently on blockchain, record all past function executions since the contract deployment. To find suitable predicate candidates for state abstraction, we use a statistical inference technique [28], [10] to generate a set of dynamic invariants, based on the transaction histories. Then, we follow the counterexample-guided abstraction refinement (CEGAR) approach [21] to perform a lazy state abstraction, and introduce minimal existential abstraction to ensure the automata specification is accurate and simple. Finally, our automata specification subsumes all observed invocation sequences and at the same time preserves its generality.

In summary, we make the following contributions. First, we formalize the specification mining problem for smart contracts. Second, we propose a CEGAR-based specification mining algorithm, powered by trace slicing and predicate abstraction. Third, we implement our approach in tool SMCON and evaluate it on eleven well-studied Azure benchmark smart contracts and six popular real-world DApp smart contracts. The experiments indicate that the mined specifications are precise and useful for DApp development and can enhance symbolic analysis of smart contracts in achieving higher code coverage and detecting more issues within smaller number of function call sequences and speeding up symbolic execution by up to 56 % by enforcing trace slicing. The benchmarks, raw results, and source code are available at: https://sites.google.com/view/smcon/.

Organization. The rest of the paper is organized as follows. Section II provides the background. Section III illustrates our approach through an example. Section IV introduces our specification mining algorithm, followed by the implementation and evaluation in Sect. V. We compare with the related work in Sect. VI and conclude the paper in Sect. VII.

II. BACKGROUND

We borrow terminology about (non-)parametric events and traces from [29].

Definition 1 (Non-Parametric Events and Traces)**.** *Let ξ be a set of (non-parametric) events, called base events or simply events. An ξ-trace, or simply a (non-parametric) trace is any finite sequence of events in ξ, that is, an element in $\xi^\star$. If event $e \in \xi$ appears in trace $w \in \xi^\star$ then we write $e \in w$.*

Definition 2 (Parametric Events and Traces)**.** *Let X be a set of parameters and let V be a set of corresponding parameter values. If ξ is a set of base events as in Definition 1, then let $\xi(X)$ be the set of corresponding parametric events $e(\theta)$, where e is a base event in ξ and θ is a partial function in $[X \rightharpoonup V]$. A parametric trace is a trace with events in $\xi(X)$, that is, a word in $\xi(X)^\star$.*

From a user's perspective, a smart contract is a set of interface functions which can be invoked to execute contract code. Let these interface functions be represented as base events: ξ is the set of interface function names and $e \in \xi$ corresponds to a contract function. The execution of e accepts parameters (denoted as X), including the user-provided function inputs (X_1) and the contract state variables (X_2) stored on the blockchain. Let V be the corresponding values of X in parametric traces. Let D_X, D_{X_1}, and D_{X_2} be the corresponding domains. Finally, given a smart contract, let $\xi(X)$ be the set of all function executions, and any function invocation sequence can be represented as a parametric trace (word) in $\xi(X)^\star$. The behaviors of a smart contract can be captured by a labeled transition system that accepts all its function invocation sequences.

Definition 3 (Labeled Transition System (LTS) [30])**.** *A smart contract is a labeled transition system (S, s_0, Σ, δ) where S is a set of possibly-infinite states, $S \subseteq D_{X_2}$ $s_0 \in S$ is an initial state, Σ is a possibly-infinite alphabet, $\Sigma \subseteq \xi(X)^\star$, and $\delta \subseteq S \times \Sigma \times S$ is a set of transitions.*

An LTS can be represented more compactly by abstracting it into an $EFSM$.

Definition 4 (Extended Finite State Machine ($EFSM$) [31])**.** *$EFSM$ is defined as a 6-tuple $(Q, q_0, \Sigma', G, U, T)$ where,*

- *Q is a finite set of symbolic states under a predicate abstraction $\alpha : S \to Q$,*
- *$q_0 \in Q$ is the initial symbolic state,*
- *Σ' is a finite alphabet defined, $\Sigma' \subseteq \xi^\star$,*
- *G is a set of guarding function g_i such that g_i: $D_X \to \{True, False\}$,*
- *U is a set of update function u_i such that u_i: $D_X \to D_X$,*
- *T is a transition relation, $T : Q \times G \times \Sigma \to U \times Q$.*

To compute state abstractions, predicate abstraction [21] is typically used, which is a function to create a partition of the domains of data types. For example, the widely used predicate abstraction for integer domain is $\{neg,\ zero,\ pos\}$ which represent negative, zero and positive numbers respectively. However, there could be many $EFSM$ candidates that an LTS can be abstracted into. In this paper, we borrow the concept of *minimal existential abstraction* [32] and later use it to obtain a compact $EFSM$.

Definition 5 (Minimal Existential Abstraction [32]). *EFSM* $= (Q, q_0, \Sigma', G, U, T)$ *is the minimal existential abstraction of LTS* $= (S, s_0, \Sigma, \delta)$ *with respect to* $\alpha : S \to Q$ *iff,*

$$\exists s_0 \in S \cdot \alpha(s_0) = q \iff q = q_0 \tag{1}$$

$$\begin{aligned} &\exists (s_0, e_0(\theta_0), s_1), \ldots, (s_{n-1}, e_{n-1}(\theta_{n-1}), s_n) \in \delta \cdot \\ &\alpha(s_0) = q_0 \wedge \alpha(s_1) = q_1 \wedge \cdots \wedge \alpha(s_{n-1}) = q_{n-1} \wedge \alpha(s_n) = q_n \\ &\iff (q_0, g_i, e_0, u_i, q_1), \ldots, (q_{n-1}, g_j, e_{n-1}, u_j, q_n) \in T \end{aligned} \tag{2}$$

Intuitively, the minimal existential abstraction implies that: (1) the initial concrete state can be mapped to the initial symbolic state in the extended finite state machine, and vice versa; (2) every concrete path is preserved in the extended finite state machine, and every symbolic path in the extended finite state machine has at least a corresponding concrete path.

III. Approach at a Glance

We illustrate our approach using the `GameChannel` contract from a DApp called Dicether. Dicether is a decentralized casino application on Ethereum, relying on a smart contract to provide an open, secure, and fair gaming experience. A new game is created by calling the contract function `createGame`. When approaching the end of a game, an admin user may invoke the `serverEndGame` function to close the game. More details can be found in its development documentation [33].

Figure 1 overviews how we separate interleaving interactions from past transaction histories. A transaction history is a sequence of transactions, where each transaction can be decoded as a contract function invocation. We apply a slicing function, which is determined by interaction patterns observed in test suites, on the transaction history, to produce a set of independent invocation sequences. For `GameChannel`, there are six game interaction sequences, corresponding to six user sessions. For instance, $user_1$: `A(gameId:1)` indicates the invocation of `createGame` by $user_1$ for creating a game with index 1. For simplicity, we omit the values of the other function parameters and the transaction environment variables. A function invocation may change the values of state variables, thus updating contract states. In Fig. 1, the first game is created by $user_1$ and after a while ended by $user_2$. Three game states, s_0, s_1, and s_2, are involved. The second game is created by $user_3$, and later canceled by $user_2$ and $user_3$ via `serverCancelActiveGame` and `userCancelActiveGame`, respectively.

From these invocation sequences, we can construct an extended finite state machine annotated with function pre-/post-conditions, as a specification of the observed contract behaviors. Specifically, in `GameChannel`, each function pre-/post-condition consists of a set of predicates either relevant to game state variables or function input parameters. Figure 3a shows the data structure used in `GameChannel`, where `server`, `gameIdCntr`, and `gameIdGame` maintain information about the game manager, the number of created games, and all game state information, respectively. The game state variables include `status`, `roundId`, `endInitiatedTime`, and `stake`. The variable `status` being ENDED (0) indicates that a game either has not been created or has already been terminated; `roundId` is an unsigned integer used to record the current game round; `endInitiatedTime` records when a game is required to terminate itself as per users' requests; and `stake` keeps the amount of fund that a player deposits into the contract when creating a game. Since all parameter values, including contract state variables and user-provided function inputs, can be decoded from blockchain transactions, we can infer dynamic invariants to be candidates of predicates on function pre-/post-conditions. Figure 2 shows the 11 resulting predicates.

Assume that all contract state variables are initialized to zero, so s_0 can be represented by $P_1 \wedge P_6 \wedge P_8 \wedge P_{10}$. These predicates also form the pre- and post-conditions in Table I, where some other parameter predicates in the pre- and post-conditions are over function input parameters, i. e., "`_roundId`" and caller of the function, i. e., "`caller`". The precondition of the *createGame* function is that all variable values, namely, `status`, `stake`, `roundId`, and `endInitiatedTime`, are zero; and its postcondition is that when *createGame* finishes, the variable `status` is set to `ACTIVE` (1) and the deposited `stake` is greater than zero and equals to the transferred fund, i. e., $msg.value$.[1]

Figure 3b shows our mined automaton, of which we have confirmed the correctness using the ground truth specification of `GameChannel`. The mined automaton has seven symbolic states. Only `createGame` can be called at the initial state (q_0). Furthermore, when the caller is `server`, he/she is allowed to call `serverEndGame` to terminate the game and move towards the final state (q_6) where `status` changes to be `ENDED` (0). Such an automaton captures the common usages of `GameChannel` and its permission policies, thus being a *likely* contract specification.

As for automata construction, SMCon uses a CEGAR-like approach, which will be detailed in Sect. IV. Briefly, we perform a lazy abstraction, i. e., we do not refine predicate abstraction unless we have to. To obtain an extended finite state machine, SMCon takes the sliced independent invocation sequences and the inferred function pre-/post-conditions as input. Initially, we construct an automaton containing only two states and then revisit the automaton to recognize the spurious symbolic paths that have no support, i. e., a corresponding concrete invocation sequence in the past observations. Then we refine the automaton to eliminate the spurious paths via either splitting larger states or removing unreachable transitions. We repeat this process until no spurious path is included in the resulting automaton.

IV. Contract Specification Mining

In this section, we introduce the specification mining problem for smart contracts and present our proposed algorithm.

Smart Contract Specification Mining. Given a contract's transaction histories, where all the past contract behaviors are

[1] In Solidity smart contracts, $msg.value$ refers to the amount of transferred native cryptocurrency, e.g., ETH on Ethereum, during contract function execution.

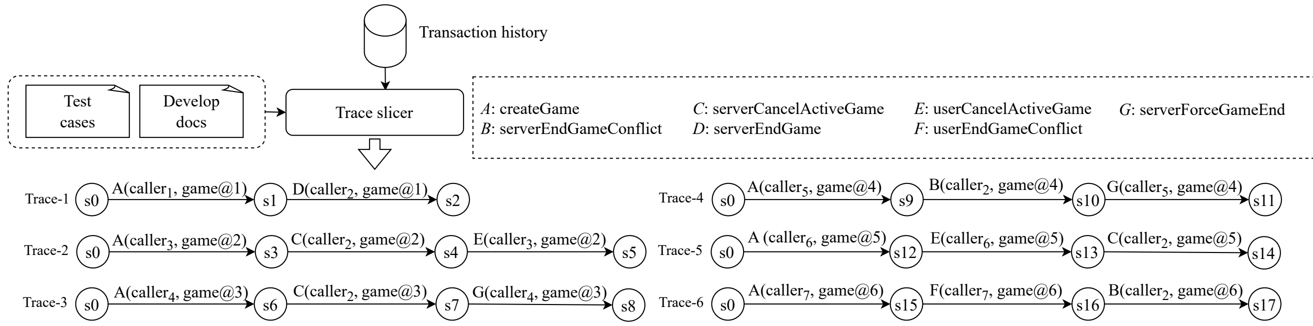


Fig. 1: Six game invocation sequences for GameChannel.

TABLE I: The function pre-/post-conditions of `GameChannel`.

Functions	Preconditions	Post-conditions
createGame	$P_1 \wedge P_6 \wedge P_8 \wedge P_{10}$	$P_2 \wedge P_6 \wedge P_8 \wedge P_{11} \wedge (stake = msg.value)$
serverEndGameConflict	$(P_2 \vee P_3) \wedge (\_roundId > 0) \wedge (caller = server)$	$P_1 \vee (P_4 \wedge P_7 \wedge P_9) \wedge (roundId = \_roundId)$
serverCancelActiveGame	$(P_2 \vee (P_3 \wedge P_6)) \wedge (caller = server)$	$P_1 \vee (P_4 \wedge P_9)$
serverEndGame	$P_2 \wedge (caller = server)$	P_1
userCancelActiveGame	$P_2 \vee (P_4 \wedge P_6)$	$P_1 \vee (P_3 \wedge P_9)$
userEndGameConflict	$(P_2 \vee P_4) \wedge (\_roundId > 0)$	$P_1 \vee (P_3 \wedge P_7 \wedge P_9) \wedge (roundId = \_roundId)$
serverForceGameEnd	$P_4 \wedge (caller = server)$	P_1

$$P_1 : status = 0 \quad P_2 : status = 1$$
$$P_3 : status = 2 \quad P_4 : status = 3 \quad P_5 : status > 3$$
$$P_6 : roundId = 0 \quad P_7 : roundId > 0$$
$$P_8 : endInitiatedTime = 0 \quad P_9 : endInitiatedTime > 0$$
$$P_{10} : stake = 0 \quad P_{11} : stake > 0$$

Fig. 2: The 11 predicates that partition the game state.

captured by LTS_h, the *specification mining problem* is to mine an $EFSM$ as the likely specification of the smart contract. To solve the specification mining problem, we first perform a *trace slicing* on the input transaction histories, to obtain multiple independent invocation traces. Next, we find predicates that belong to preconditions or post-conditions of the smart contract's functions. Finally, we implement a counterexample-guided abstraction refinement loop to produce an extended finite state machine, satisfying the minimal existential abstraction property (see Definition 5).

A. Trace Slicing

Smart contracts are public-facing, and, by their nature, simultaneously accept inputs from multiple users. Contract executions in such a setting result in a linear transaction history, which consists of interleaving execution traces triggered through multiple user interactions/sessions. To record data owned by different users, most smart contracts supporting DApps, maintain a collection of custom data objects, indexed by user(session)-specific parameters. For example, the `GameChannel` contract maintains many concurrent game instances as state variables. To interact with a particular game instance, a user needs to specify the value of its `gameId`, through input parameters of the transaction (see Fig. 1). To mine meaningful contract specifications from transaction histories with mixed interactions, one has to slice them into independent traces for each game instance.

Definition 6 (Trace Slicing [29])**.** *Given a parametric trace $\tau \in \xi(X)^\star$ and a parametric binding θ in $[X \rightharpoonup V]$, let the θ-trace slice $\tau \restriction_\theta \in \xi^\star$ be the non-parametric trace defined as:*

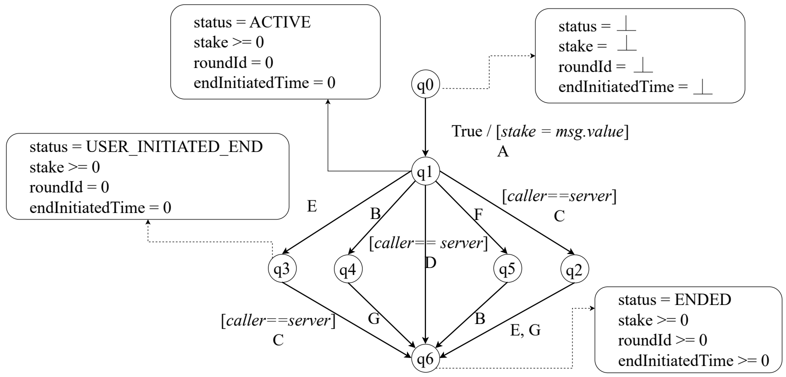
- *$\epsilon \restriction_\theta = \epsilon$, where ϵ is the empty trace/word, and*
- $(\tau e(\theta')) \restriction_\theta = \begin{cases} (\tau \restriction_\theta)e, & \text{if } \theta' \sqsubseteq \theta \\ \tau \restriction_\theta, & \text{otherwise} \end{cases}$

where we say that θ' is less informative than θ, written $\theta' \sqsubseteq \theta$ iff for any $x \in X$, if $\theta'(x)$ is defined then $\theta(x)$ is also defined and $\theta'(x) = \theta(x)$.

A transaction history of smart contract can be seen as a parametric trace, and *trace slicing* slices the history into a set of independent invocation sequences via certain parametric bindings (e.g., θ) [29]. A trace slice $\tau \restriction_\theta$ first filters out all the parametric events that are irrelevant to the parameter instance θ. A trace slice also forgets the parameter bindings of parametric events. As a result, a trace slice is non-parametric and merely a list of base events. To find parametric bindings, we should first ascertain the relation between different events, or say function invocations in smart contracts. Such parametric bindings can be inferred from the existing DApp test suites, which demonstrate typical usage scenarios and user interaction patterns. Specifically, we may observe a group of related functions and what parameter values they share in a unit test. For example, the test suites for `GameChannel` contain many well-written test cases where game objects are explicitly specified by the "`gameId`" variable in each contract function. Therefore, we can use such relations as a configuration to instruct how to automatically slice the transaction history

```
contract GameChannel{
  enum GameStatus {
    ENDED, ACTIVE, USER_INITIATED_END,
    ↪  SERVER_INITIATED_END
  }
  struct Game {
    GameStatus status;
    uint128 stake;
    uint32 roundId;
    uint endInitiatedTime;
  }
  // @dev Game id counter.
  uint public gameIdCntr = 0;
  address public server;
  mapping (uint => Game) public gameIdGame;
  // functions ...
}
```

(a) State variables of GameChannel.



(b) The mined automaton.

Fig. 3: The core data structure and the mined automaton of GameChannel.

according to the corresponding values of “`gameId`” to generate a set of independent game invocation sequences.

B. Predicate Discovery from Dynamic Invariants

The choice of predicates is crucial for computing good state abstractions. In this paper, we use *likely* pre- and post-conditions of contract functions as candidates. Because of the blockchain transparency, we may decode the values of contract state variables and user-provided function inputs, before and after each function invocation. Then we statistically infer dynamic invariants for each function, which hold for all observed invocations in the past transaction histories. But since the transaction history may be limited, the inferred pre- and post-conditions are *likely* to hold, which is good enough to serve as predicate candidates.

More specifically, we define a *predicate template* as “$x \bowtie y$”, where $x \in X$ is a parameter, and $y \in X \cup K$ is either a parameter or constant, and $\bowtie$ is an operator from the set $\{=, !=, >, <, <=, >=\}$. The template is instantiated on all successful transactions, which are not reverted during executions, and the instances which always hold are kept as predicate candidates for either function pre- or post-conditions. The predicates defined over state variables are used in constructing the symbolic states Q in EFSM.

The inference process is similar to how dynamic invariants are detected in Daikon-like systems [28] through a set of predefined invariant templates. However, smart contracts are usually writting in Turing-complete programming languages such as Solidity, supporting complex data structures including array, mapping and custom struct. Thus, we built our invariant inference on a tool called InvCon+ [34], [10] capable of invariant detection for smart contracts.

C. Automata Construction

The over-generalization of the inferred function pre-/post-conditions is the main difficulty for their direct use in mining high-level automata specifications. To address this problem, we use a CEGAR-like approach to mine automata specifications with predicate abstraction.

Algorithm 1 SPLITREMOVE(q_n, t_{n+1}, *EFSM*)

Let $\langle q_0, Q, \Sigma, G, U, T\rangle = EFSM$
Let $t_{n+1} = (q_n, g_m, e_m, u_m, q_{n+1}) \in T$ ▷ a transition from state q_n to q_{n+1} by the invocation to function e_m where g_m, u_m are its precondition and post-condition, respectively.
$\hat{q_1} = q_n \wedge g_m$
$\hat{q_2} = q_n \wedge \neg g_m$
if $\mathrm{SAT}(\hat{q_1}) \wedge \mathrm{SAT}(\hat{q_2})$ **then** ▷ q_n is splittable with g_m.
  $Q \leftarrow (Q \setminus q_n) \cup \{\hat{q_1}, \hat{q_2}\}$ ▷ replace q_n with two new states.
  Removes transitions starting or ending with q_n in T
else
  if $\nexists\ \pi'_n \cdot \pi'_n \in EFSM\ \wedge \mathrm{concretize}(\pi'_n \oplus t_{n+1}) \in LTS_h$ **then**
    $T \leftarrow T \setminus t_{n+1}$ ▷ remove unreachable transition t_{n+1}
  else
    Let $S_{q_n|\pi'_n}$ be the set of all the concrete states of q_n in the history LTS_h, which are visited by the observed invocation sequences of π'_n.
    $Q \leftarrow (Q \setminus \{q_n\}) \cup \{\mathrm{Pred}(S_{q_n|\pi'_n}), q_n \wedge \neg \mathrm{Pred}(S_{q_n|\pi'_n})\}$
    Removes transitions starting or ending with q_n in T
return $\langle q_0, Q, \Sigma, G, U, T\rangle$ ▷ return the resulting automaton

Counterexample-guided abstraction refinement. To mine a precise specification, the key is to compute a precise state abstraction α, which partitions the contract state. The abstraction function α is implicitly computed following the paradigm of counterexample-guided abstraction refinement [21]. We define our specification mining algorithm by the three rules in Fig. 4. Our algorithm takes as input past observations of concrete invocation sequences and inferred function pre-/post-conditions. When the algorithm terminates, it produces an *EFSM* containing no spurious states and transitions.

The INIT rule initializes a preliminary extended finite state machine containing two states: q_0, referring to $\bigwedge_{x \in X_1} x = 0$ that all state variables are valued zero, and $\neg q_0$ for the remaining cases. The guard function G and update function U are directly instantiated by the inferred function pre- and postconditions, respectively. Also, the transition relation set T is initialized to be empty. Then, we apply the CONSTRUCT rule to add theoretically feasible state transitions to the automaton. A state transition is theoretically feasible if and

$$\frac{\bot}{\langle q_0 \leftarrow \bigwedge_{x\in X_1} x=0, Q\leftarrow\{q_0,\neg q_0\}, \Sigma, G\leftarrow\{g_m\}_m, U\leftarrow\{u_m\}_m, T\leftarrow\emptyset\rangle}\ \textsc{Init}$$

$$\frac{\langle q_0,Q,\Sigma,G,U,T\rangle \qquad \exists\, q_i,q_j\in Q\cdot(q_i\wedge g_m)\wedge(q_j\wedge u_m) \qquad \nexists\, t\cdot(q_i,g_m,e_m,u_m,q_j)\in T}{\langle q_0,Q,\Sigma,G,U,T\leftarrow T\cup\{t\}\rangle}\ \textsc{Construct}$$

$$\frac{\mathit{EFSM}\colon \langle q_0,Q,\Sigma,G,U,T\rangle \qquad \begin{array}{r} \exists\,\pi_n : q_0t_1t_2\cdots t_nq_n\in \mathit{EFSM}\ \exists\,\text{concretize}(\pi_n)\in \mathit{LTS}_h \\ \exists\,\pi_{n+1} : \pi_n\oplus t_{n+1}q_{n+1}\in \mathit{EFSM}\ \nexists\,\text{concretize}(\pi_{n+1})\in \mathit{LTS}_h \end{array}}{\mathit{EFSM} \leftarrow \textsc{SplitRemove}(q_n, t_{n+1}, \mathit{EFSM})}\ \textsc{RmPath}$$

Fig. 4: Specification mining rules.

only if it satisfies the logical conjunction of symbolic states and function preconditions or post-conditions. The resulting automaton could be over-generalized such that it includes spurious state transition paths. Therefore, we need to apply the RMPATH rule, following Algorithm 1 to eliminate those spurious state transition paths that are not supported in the concrete observations. Algorithm 1 rules out spurious paths by either state splitting or transition removal. These rules would be applied many times according to a fair scheduling. When the algorithm terminates, it produces an extended finite state machine, containing no spurious states or transitions. The illustration of a running example and the fair scheduling and its proof are available at https://sites.google.com/view/smcon/.

Loop transitions. The resulting automaton does not allow loop transitions according to the RMPATH rule. However, this kind of automaton may not be precise and useful contract specifications. Because many smart contracts have behavior cycles, it is preferred to have loop transitions in the resulting automaton. Therefore, we limit the range of path selection when applying RMPATH, i. e., a loop transition can only be covered once in any selected path. For example, a state transition path $q{-}Event_a{-}q{-}Event_a{-}q$ is not under our consideration when allowing loop transitions. With this minor modification to the RMPATH rule, the resulting automaton allows loop transitions so that it may express cycles.

V. IMPLEMENTATION AND EVALUATION

A. Implementation

We implement trace slice approach and specification mining algorithm as a tool named SMCON, written in around 3K lines of Python code. Specifically, we apply our trace slicing approach to retrieve independent user action traces from transaction histories according to the given trace slice configurations, and then we invoke InvCon+ to produce corresponding likely invariants. Based on these sequences and likely invariants, we are able to perform specification mining for smart contracts. Additionally, our algorithm relaxes the RMPATH rule to allow loops in the contract specifications for better generality (see Sect. IV-C). We used the Z3 SMT solver [35] for discharging satisfiability queries.

We generate function-level invariants for smart contracts from the past transaction histories and filter the generated invariants to keep those expressing parameter relations

TABLE II: The Azure smart contract benchmark.

Contract	Description	Formal Specifications	
		# States	# Transitions
AssetTransfer	Selling high-value assets	11	32
BasicProvenance	Keeping record of ownership	4	4
BazaarItemListing	Selling items	4	5
DefectCompCounter	Product counting	3	2
DigitalLocker	Sharing digital files	7	12
FreqFlyerRewards	Calculating flyer rewards	3	3
HelloBlockchain	Request and response	3	3
PingPongGame	Two-player games	4	2
RefrigTransport	IoT monitoring	5	8
RoomThermostat	Thermostat installation and use	3	4
SimpleMarketplace	Owner and buyer transactions	4	4
Average		4.64	7.18

(see Sect. IV-B). These invariants serve as the parameter predicates that we use for automata construction (see Sect. IV).

Through experiments, we evaluated SMCON to answer the following three research questions:

- **RQ1**: How effectively does SMCON mine smart contract specifications compared with the state-of-the-arts?
- **RQ2**: How effectively does SMCON mine automata from real-world DApp smart contracts, and with these automata, how is symbolic analysis for smart contracts enhanced?
- **RQ3**: What are the implications for DApp developers?

B. Methodology

To answer RQ1, we evaluate SMCON on parametric-free smart contracts from a well-studied benchmark used for Azure enterprise blockchain, where none of these contracts have index-related data structures so we do not perform trace slicing on their transactions. This benchmark includes 11 smart contracts exhibiting stateful behaviors, ranging over supply chain management, digital control, virtual games, etc. Each of these contracts is properly documented, and their specifications have been well formalized and examined by the previous work. Such ground truth specifications are deemed as the reference models in our evaluation. Because SMCON aims to dynamically infer specification models from past contract executions, we produce 10,000 transactions per contract using random test case generation. In detail, we deploy every contract 100 times to our testnet. Each contract instance is tested using 100 randomly generated transactions, which finally produce a trace, namely a sequence of contract executions. Subsequently, we perform SMCON on these contract traces to mine contract specification models.

To answer RQ2, we evaluate SmCon on real-world parametric smart contracts running on Ethereum. We selected six popular Ethereum DApp smart contracts as shown in Table IV. We selected them from the Top-10 DApps covering different application domains [3], such as decentralized gaming, gambling, non-fungible token (NFT) usage, and an exchange market. For example, the DApp `SuperRare` has a total trading volume up to 557 million dollars contributed by more than 10,000 users in nearly 100,000 transactions [36]; and MoonCatRescue has a total trading volume up to 73 million dollars involving more than 11,000 users [37]. These DApps have been deployed and running for a long period, since as early as 2017, and their past transaction data can be downloaded from Ethereum. Most of these DApps (except 0xfair) maintain some form of design documentation on their websites or GitHub repositories; some also provide formal specifications, such as Dicether [33]. In addition, well-organized DApp projects, such as the studied ones, maintain test suites that exercise the core functionalities of the contracts with reasonable coverage. With these artifacts, we are able to construct ground models manually for DApp contracts. We collected their contract code and transaction data from Etherscan [4] and Ethereum archive node hosted by QuickNode [38]. Particularly, the number of transactions used for specification mining is also capped at 10,000 for all DApp smart contracts.

Evaluation Metrics. To evaluate SmCon, we use the accuracy metric recommended in [39] for automata specification mining evaluation. The accuracy metric measures the similarity between the mined automata specification and the ground truth, considering both precision and recall. Precision is defined as the percentage of sequences generated by the mined automata that are accepted by the ground truth, while recall is the percentage of sequences generated by the ground truth that are accepted by the mined automata. Following [23], we use the F_1-score to measure the overall accuracy, which is defined as: $F_1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$. Since automata may have infinite sequences when they have loop transitions, to obtain accurate precision, recall and F_1-score, we follow the similar strategies used in previous works [23], [40], [41] to generate the sequences. We set the maximum number of generated sentences to 10,000 with minimum coverage of each transition to be 20 in the generated traces [23] and restrict the length of the traces to twice the number of transitions [25] in the ground-truth models that have been formalized by the Azure benchmark or manually constructed by ourselves. In addition, for RQ1, we divide the transaction data into a training and a test set, where we mine the model from contract executions in the training set. We use another accuracy metric, denoted as Acc, to measure how many percentages of contract executions in testing set are accepted by the mined model.

C. Experiment Setup

All experiments were conducted on an Ubuntu 20.04.1 LTS desktop equipped with an Intel Core i7 16-core processor and 32 GB of memory. The ground truth, benchmark contracts, and raw results are available at: https://sites.google.com/view/smcon/.

D. RQ1. Effectiveness of SmCon

To answer RQ1, we compared SmCon with five baseline approaches. K-tail [24] learns an automaton from prefix trees of traces by merging nodes with the same 'tail' of length k. We evaluated its two settings, 1-Tail when $k = 1$ and 2-Tail when $k = 2$. SEKT [25] is a type of state-enhanced k-tail, which extends k-tail using program state information inferred from the full set of observed executions. We also evaluated its two settings, SEKT-1 when $k = 1$ and SEKT-2 when $k = 2$. Contractor++ [42], [25] creates finite state machine models exclusively based on program invariants inferred from the observed executions. To the best of our knowledge, there exists only one other approach to mine state machine models from smart contract executions, by Guth et al. [43]. However, their tool was not available for comparison at the time of writing. We will discuss this related work and compare with it in Sect. VI.

The original benchmark contracts do not always satisfy the specifications that come with them, which has also been revealed by a previous study [19]. For a fair comparison with the other approaches, we manually repaired these issues and also reported them to the developer [44], [45], [46], [47]. For instance, *SimpleMarketplace* is a contract application that implements a workflow for a simple transaction between an owner and a buyer in a marketplace. *SimpleMarketplace* has an *AcceptOffer* function to allow owner to accept the offer made by buyers. However, *AcceptOffer* even succeeds when there is no offer placed, thus violating its formal specification [47].

Evaluation results. Table III provides a detailed overview of the comparative performance of various tools, including our developed tool SmCon, in the domain of smart contract specification mining. Each row corresponds to a specific smart contract, with columns showcasing essential metrics such as the number of state machine models generated (# States), the F-score (F_1), and the accuracy (Acc). The evaluated tools, denoted as 1-Tail, 2-Tail, SEKT-1, SEKT-2, Contractor++, and SmCon, allow for a comprehensive analysis of their capabilities in extracting and representing contract specifications. The variety of contracts considered, ranging from *AssetTransfer* to *SimpleMarketplace*, ensures a diverse and thorough assessment of each tool's performance across different use cases. We do not compare with grammar inference and deep learning techniques since our preliminary experiments with the minimal-description-length grammar inference by LearnLib [48] indicate that grammar inference tends to overgeneralize, having very poor precision, while deep learning techniques demand a large volume of training data that is difficult to collect from real-world transactions.

Upon closer examination of the data, it is evident that SmCon consistently exhibits competitive performance metrics, followed by Contractor++. Notably, in the AssetTransfer contract, SmCon outperforms Contractor++ by generating a state machine model with 13 states, resulting an higher F_1

TABLE III: Experiement results on the Azure benchmark.

Contract	1-TAIL			2-TAIL			SEKT-1			SEKT-2			CONTRACTOR++			SMCON		
	# States	F_1	Acc	# States	F_1	Acc	# States	F_1	Acc	# States	F_1	Acc	# States	F_1	Acc	# States	F_1	Acc
AssetTransfer	24	0.52	0.93	40	0.47	0.77	24	0.52	0.93	40	0.47	0.77	13	0.2	1	13	0.34	0.97
BasicProvenance	4	0.72	1	6	0.67	1	4	0.67	1	6	0.7	1	3	0.63	1	3	0.8	1
BazaarItemListing	9	0.94	1	94	0.97	0.84	9	0.94	1	83	0.98	0.87	3	0.89	1	3	1	1
DefectCompCounter	3	1	1	3	1	1	3	1	1	3	1	1	3	1	1	3	1	1
DigitalLocker	18	0.57	0.95	29	0.34	0.94	18	0.57	0.95	29	0.34	0.94	9	0.95	1	10	0.87	1
FreqFlyerRewards	3	1	1	5	1	1	3	1	1	5	1	1	2	1	1	2	1	1
HelloBlockchain	4	1	1	5	1	1	4	1	1	5	1	1	3	1	1	3	1	1
PingPongGame	4	0.77	1	4	0.75	1	4	0.77	1	4	0.75	1	5	0.51	1	4	0.77	1
RefrigTransport	6	0.7	1	8	0.68	1	6	0.7	1	8	0.69	1	5	0.43	1	5	0.69	1
RoomThermostat	5	0.88	1	9	0.88	1	5	0.88	1	9	0.88	1	5	1	1	6	1	1
SimpleMarketplace	5	1	1	6	1	1	5	1	1	6	1	1	4	1	1	5	1	1
Average	7.73	0.83	0.99	19.00	0.80	0.96	7.73	0.82	0.99	18.00	0.80	0.96	5.00	0.78	1.00	5.18	0.86	1.00

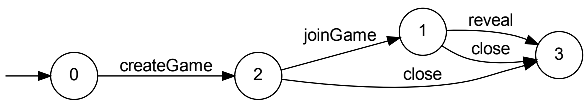
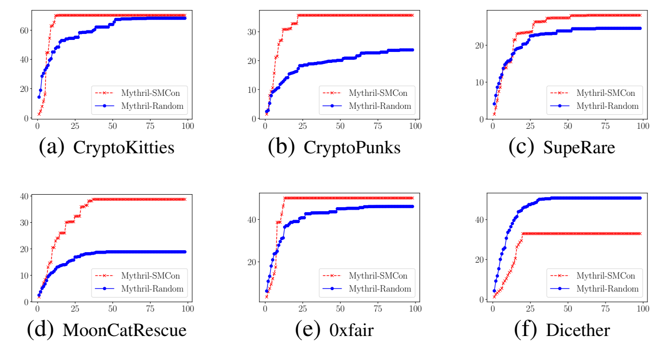


Fig. 5: The mined automaton for 0xfair. Note that we exclude parametric bindings for simplicity.



(a) CryptoKitties (b) CryptoPunks (c) SupeRare

(d) MoonCatRescue (e) 0xfair (f) Dicether

Fig. 6: Opcode coverage achieved with the number of function call sequences used. The x-axis and y-axis indicate the number of function call sequence and the percentage of opcode coverage, respectively.

score of 0.34 with neglectable loss of precision. Across all contracts, SMCON maintains an average of 5.18 states per model, an impressive F_1 score of 0.86, and nearly perfect accuracy (1.00). These results highlight the efficacy of SMCON in accurately capturing the intricacies of smart contract behavior. The tool's robust performance, in terms of model compactness, accuracy and F_1 score, distinguishes it from other baseline approaches, emphasizing its potential as a reliable solution for specification mining tasks.

In summary, SMCON emerges as a promising tool for smart contract specification mining, achieving good precision, recall, and accuracy for around three minutes per contract. The presented results demonstrate its consistent ability to generate accurate state machine models across a diverse set of contracts. The high average F_1 score and accuracy substantiates effectiveness in capturing the intended behavior of smart contracts. These results establish SMCON as a valuable resource for researchers and practitioners in search of a dependable and adaptable tool for real-world smart contract analysis.

E. RQ2. Experiment Results on Real-world Smart Contracts

Table IV illustrates the automata mining results of SMCON on six real-world DApp contracts. The model complexity of specifications mined varies a lot. *CryptoKitties* has the simplest model with two states and three transitions. The model can be interpreted as a regular language "$(createAuction \rightarrow bid \mid cancelAuction)*$", where each active auction accepts only one bid. Figure 5 shows the specifications mined for *0xfair*,[2] which perfectly articulates the usage scenarios of a **R**ock-**P**aper-**S**cissor game. *0xfair* employs a seal mechanism to achieve fairness where nobody can cheat on others. First, the creator encrypts his choice and publicizes the choice proof, namely, the corresponding cryptographic signature, when creating a game via *createGame*. Naturally, the second player joins this game with an explicit choice via *joinGame*. Finally, the creator reveals his choice by decrypting the choice with the secret key, which is used to determine the game winner. In addition, a game should be closed when it expires, because no other players join or the creator fails to reveal the choice.

The remaining four DApps have more complex models, which we omit due to limited space. We manage to assess these mined models through their existing test suites. We first re-ran all the test cases for each DApp and found that many test cases failed. For example, *MooncatRescue* has 1,119 test cases, with 993 passing and 126 failing mostly due to VM error messages slightly unmatched with the expected. Next, we were able to construct their ground truth specifications manually, where two of the authors spent two hours per smart contract individually. In particular, we ensured that the tested behaviors must be included in the ground-truth specifications and added any additional untested behaviors clearly documented. Because the transaction data used is collected from an uncontrolled blockchain environment, the diversity of historical usage behaviors have a considerable impact on the mined model. Our study shows that GameChannel achieved the highest precision of 93.1 % and recall of 97.9 %, respectively, while SupeRare scored the second highest recall of 96.1 % for SupeRare, followed by CryptoPunks achieving 80.7 %, and MoonCatRescue achieved the second highest precision of 85.6 %. Therefore, we believe our mined models for real-

[2] https://etherscan.io/address/0xa8f9c7ff9f605f401bde6659fd18d9a0d0a802c5

TABLE IV: Experiment results on real-world DApp contracts.

	Description	Mined Specifications		Opcode Coverage						Number of Issues					
				Mythril-Random		Mythril-SMCon		Statistics		Mythril-Random		Mythril-SMCon		Statistics	
		#State	#Tran.	Avg.	Var.	Avg.	Var.	p-value	$\hat{A_{12}}$	Avg.	Var.	Avg.	Var.	p-value	$\hat{A_{12}}$
CyptoKitties	kitty auction	2	5	68.20%	0.0016	70.21%	0.0214	0.3785	0.6667	4.5	0.3	6.5000	7.5000	0.0677	0.6667
CryptoPunks	punk market	20	81	23.72%	0.0003	35.72%	0.0162	0.0345	0.8333	1	0.4	1.3333	0.6667	0.2243	0.6389
SupeRare	art market	15	88	24.58%	0	28.12%	0.0130	0.2415	0.3333	0	0	0.3333	0.2667	0.0873	0.6667
MoonCatRescue	cat adoption	18	70	18.88%	0.0003	38.76%	0.0028	0.0001	1	1	0	1	0	NA	0.5000
0xfair	RPS game	4	5	46.11%	0.0003	50.09%	0.0000	0.0011	1	4	0	5	0	NA	1
Dicether	bet game	8	16	50.83%	0.0022	32.99%	0	0.0001	0	4	0.8	3	0	0.0204	0.1667
Average				38.72%	0.0008	42.65%	0.0089	0.1093	0.6389	2.4167	0.25	2.8611	1.4056	0.1000	0.6065

world DApp contracts shall capture widely-used high-level program specifications, which can be used to enhance DApp development, e.g., uncovering issues of DApp document and test suites.

We study the effectiveness of the resulting automata for symbolic analysis of smart contracts. Unlike testing, symbolic analysis often yields a more comprehensive security report by effectively exploring multiple program paths at once. Nevertheless, symbolic analysis may face path explosion problem, which largely affects its performance. In Table IV, we compare two usages of the state-of-the-art industrial symbolic analysis tool named Mythril [49]—by providing randomly generated function call sequences, i.e., Mythril-Random, and by providing function call sequences generated from automata specifications minded by SMCON, i.e., Mythril-SMCON. For each contract, we cap the length of function call sequences to be five, the time budget to be one hour while the timeout of symbolic execution of a function call sequence is set to 10 minutes. For reliable comparison, we repeated such symbolic analysis process six times per contract. Notice, for contract functions absent in our mined automata, we perform random selection and insert the selected ones into the function call sequence generated by the model.

Smart contracts are compiled into opcodes executable on the Ethereum Virtual Machine. Table IV shows the opcode coverage achieved and the number of issues reported by Mythril-Random and Mythril-SMCON. At first glance, most of the code coverage statistics seem low. This is partly because we only test the public contract functions that could alter program states while leaving untested the other view functions that only access program states. The timeout setting also has an impact on this, and we will explain it later. Overall, Mythril-SMCON achieves 42.65 % code coverage and finds around 3 issues per contract, which is more than what Mythril-Random achieves. Note, the issues reported by Mythril are often considered as warnings for developers to check, which may not always reflect real vulnerabilities. In detail, Mythril-SMCON outperforms Mythril-Random in all cases except Dicether. Moreover, Mythril-SMCON is proved more likely to explore new program paths, since Mythril-SMCON displays a larger variance of the code coverage than Mythril-Random except 0xfair and Dicether. Our further investigation shows that Dicether has two preparation functions to execute before any game-related operations, which are not included in the mined automata, but such problem can be mitigated using dependency analysis [50]. For 0xfair, the program path constraints are too complicated to solve within the given timeout, which will be illustrated in Fig. 7. We also perform statistics analysis using Mann Whitney U-test to show the significance level of the experiment result and Vargha and Delaney's A12 statistical test to determine the extent to which Mythril-SMCON outperforms Mythril-Random. The results in Table IV indicate that in terms of code coverage or number of issues reported, Mythril-SMCON usually performs better than Mythril-Random in 4 out of 6 cases, with resulting $\hat{A_{12}}$ scores exceeding 0.6 and p-values being smaller than or close to a significance level of 0.05. We also delve into how SMCON promote efficiency of symbolic analysis in achieving good opcode coverage with much less function call sequences. As shown in Fig. 6, except Dicether, SMCON helps Mythril reach a higher opcode coverage with less number of function call sequences compared to its random counterpart, highlighting the usefulness of the specifications mined. We acknowledge that there are many studies that improve fuzzing effectiveness by incorporating valuable feedback information from static analysis [51] and dynamic analysis [50], [52]. The high-level behavior automata mined by SMCON align with this field and complement these existing fuzzing tools.

To speed up symbolic analysis, we could also enforce the trace slice setting by fixing trade session parameter, e.g., gameId, to a constant because most trade sessions are homogeneous, non-interleaving and symbolic analysis of a trade session should suffice. To investigate this impact, we sampled the function call sequences derived from the mined automata of the DApp contracts where each sequence represents a particular trade scenario. Figure 7 draws the overall time consumptions for default and trace slice setting, where for each setting, we symbolically execute each sequence five times. Trace slice setting takes smaller time for all cases except CryptoKitties, where MoonCatRescue has 56% speedup, followed by 0xfair's 36%.

In summary, the automata infered by SMCON about high-level program behaviors is critical to reduce the burden of symbolic analysis for complicated smart contracts, and it can complement existing speedup techniques, such as predicting unsatisfiable symbolic path with machine learning models [53].

F. RQ3. Implications in DApp Development

Outdated Documentation. Management of documentation and ensuring its consistency with contract implementation is often labor-intensive. For instance, *Dicether* is a gambling game

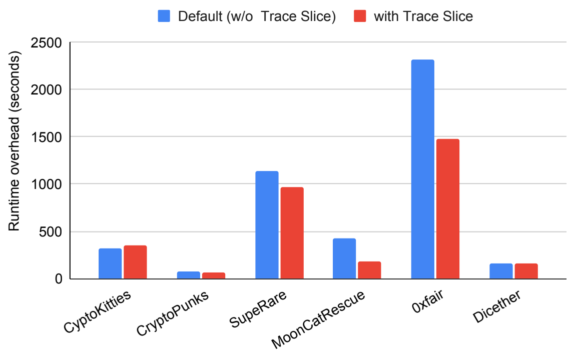


Fig. 7: Time consumption of symbolic analysis with or without enforcing trace slicing.

running on Ethereum, first launched in 2018. We investigated and collected eight contract versions of Dicether so far from DAppRadar [3] and Etherscan [4] where maintenance occurs the most in its first year and each contract version lived for about two months on average. By differentiating between these contract versions, we found some contract maintainence performs only routine tasks, e.g., minimal patching for security and reliability considerations. In contrast, some maintenance updates introduce substantial changes to business logics. When comparing its first contract version[3] with the seventh contract version,[4] we noticed function renaming changes, e.g., *playerCancelActiveGame* to *userCancelActiveGame*. Additionally, for the original contract version, the server user or normal player cannot perform any operation until a created game session is accepted. However, this business logic was removed in the seventh contract version. Yet, the only formal documentation [33] of the *Dicether* design is outdated and can no longer reflect the program behaviors of the recently used contract versions. To summarize, we believe SMCON could mine high-level automata for evolving smart contracts that can facilitate developers to track new changes easily and maintain high-quality documentation.

Test Suite Bias. Developing test suites for smart contracts is non-trivial since developers usually have little knowledge of how smart contracts are used after contract deployment to blockchains, thus crafting test cases for functions that are rarely used could waste human efforts and missing test cases for functions that are heavily used could leave a room for security risks. For example, *CryptoPunks* is one of the earliest examples of using Non-Fungibale Tokens (NFTs) on Ethereum, which inspired the ERC-721 standard to some extent. CryptoPunks has 10,000 unique collectible characters called punks, with proof of ownership stored on Ethereum [54]. To start with, function `getPunk` or `setInitialOwners` of CryptoPunks is called to assign punks to users. Users can transfer the ownership of a punk by calling `transferPunk`. Users can make a bid to a punk via `enterBidForPunk`. CryptoPunks has a set of test suites in its GitHub repository covering seven use scenarios such as setting the initial owner(s) of punks or opening a sale for punks [55]. However, in the existing test suites, there is only one test case for `setInitialOwners`, while the other test cases all focus on `setInitialOwner`. An interesting observation is that, based on transaction histories, the contract manager always use `setInitialOwners` to initialize a batch of punks for a group of owners instead of `setInitialOwner` for individual assignment. Our mined automata highlights this disproportional focus on rarely used functions, while inadequate tests were written for more frequently used functionalities. For example, it may be expected to test `setInitialOwners` whether a punk assigned to one owner can be wrongly overwritten by a succeeding owner in the same group, which is indeed not enforced in the current contract implementation.

[3]https://etherscan.io/address/0xc95d227a1cf92b6fd156265aa8a3ca7c7de0f28e

[4]https://etherscan.io/address/0xaec1f783b29aab2727d7c374aa55483fe299fefa

Threats to validity. An internal threat is potential errors in the manually derived ground truth for DApp contract specifications. To mitigate this, we collected well-documented smart contracts from popular DApp projects and re-ran their test suites. Additionally, our tool implementation and experimental scripts might contain bugs. Two authors closely collaborated on the tool and reviewed the code regularly. We also checked for outliers in the results, uncovering and fixing a few bugs. Externally, our findings may not generalize to all DApp smart contracts. To address this, we selected representative DApps from various application domains.

VI. Related Work

Smart Contract Specification Mining. Several tools have been developed for mining smart contract specifications, which can be categorized into low-level functional specifications [56], [57], [58], [10], [34] and high-level behavioral specifications [43]. SolType [56] focuses on Solidity smart contracts, allowing developers to add refinement type annotations for static analysis of arithmetic operations. While it effectively detects issues like integer overflows, it is limited to contract-level arithmetic invariants. Cider [57] extends SolType by using deep reinforcement learning to infer contract invariants, but these remain unverified. SmartInv [58] takes a multimodal learning approach to infer invariants that identify hard-to-detect bugs. InvCon [10] and InvCon+ [34] infer invariants from blockchain transactions, with VeriSol [19] verifying their correctness. However, these tools only infer invariants at function boundaries and do not capture higher-level state transitions.

Guth et al. [43] mine specifications by slicing transaction histories into independent sequences and constructing a finite state machine (FSM) based on data dependencies. Our approach differs in two key ways: (1) we use test suite-based contract interaction patterns for more precise slicing, and (2) we mine extended finite state machines (EFSM), which are more expressive than traditional FSMs.

Automata Mining. Automata mining has a rich history [22], [23], [20], [25], [59], [24], [26], [60]. Traditional approaches, such as grammar inference [61] and counterexample-guided abstraction refinement (CEGAR) [21], have been applied to learn behavioral models of systems. Aarts et al. [20] built on the

L* algorithm [62] to generate restricted EFSMs from dynamic execution traces. RPNI-MDL [22] merges states based on the minimum description length principle, but only works with positive traces. The k-tail algorithm [24] and its extensions [26], [25] merge states based on trace suffixes and incorporate input predicates to capture data relations. Krka et al. [25] developed TEMI to mine more complex EFSMs, while Synoptic [60] uses temporal invariants before applying k-tail.

Other methods such as CONTRACTOR++ [42], [25] infer FSMs from program invariants, and Le and Lo [23] use deep learning for automata specification generation. Despite the success of these techniques, smart contract specification mining presents unique challenges, particularly the dynamic, stateful environment of real-world smart contracts. The interaction between users and the system, reflected in transaction histories, requires specialized techniques like trace slicing to extract meaningful specifications. This complexity differentiates smart contract mining from traditional mining approaches.

VII. Conclusion

In this paper, we have formally defined the specification mining problem for smart contracts and proposed a CEGAR-like approach to mine automata specifications based on past transaction histories. The mined specifications capture not only the allowed function invocation sequences, but also the inferred program invariants describing contract semantics precisely. Such contract specifications are useful in contract understanding, testing, verification, and validation. Our evaluation results show that our tool, SMCON, mines specifications accurately and efficiently; it may also be used to enhance symbolic analysis for smart contracts and facilitate developer in maintaining high-quality document and test suites.

Acknowledgement

We thank all the anonymous reviewers for their constructive feedback on this work. This work was supported by the Nanyang Technological University Centre for Computational Technologies in Finance (NTU-CCTF). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NTU-CCTF.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.

[2] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[3] "Dappradar," https://dappradar.com/, 2023.

[4] "Etherscan," https://etherscan.io, 2023.

[5] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, 2020, pp. 530–541.

[6] "EIP-20: A standard interface for tokens," https://eips.ethereum.org/EIPS/eip-20, 2015.

[7] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang, "TokenScope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 1503–1520.

[8] K. Qin, L. Zhou, B. Livshits, and A. Gervais, "Attacking the DeFi ecosystem with flash loans for fun and profit," in *International Conference on Financial Cryptography and Data Security*. Springer, 2021, pp. 3–32.

[9] J. Jiao, S. Kan, S.-W. Lin, D. Sanan, Y. Liu, and J. Sun, "Semantic understanding of smart contracts: Executable operational semantics of Solidity," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1695–1712.

[10] Y. Liu and Y. Li, "InvCon: A dynamic invariant detector for Ethereum smart contracts," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct. 2022.

[11] H. Wang, Y. Li, S.-W. Lin, L. Ma, and Y. Liu, "VULTRON: Catching vulnerable smart contracts once and for all," in *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE Press, 5 2019, pp. 1–4.

[12] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE symposium on security and privacy (SP)*. IEEE, 2020, pp. 1661–1677.

[13] Y. Liu, Y. Li, S.-W. Lin, and R. Zhao, "Towards automated verification of smart contract fairness," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 666–677.

[14] A. Li, J. A. Choi, and F. Long, "Securing smart contract with runtime validation," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 438–453.

[15] A. Mavridou and A. Laszka, "Designing secure ethereum smart contracts: A finite state machine based approach," in *International Conference on Financial Cryptography and Data Security*. Springer, 2018, pp. 523–540.

[16] ——, "Tool demonstration: FSolidM for designing secure Ethereum smart contracts," in *International conference on principles of security and trust*. Springer, 2018, pp. 270–277.

[17] Y. Liu, Y. Li, S.-W. Lin, and Q. Yan, "ModCon: A model-based testing platform for smart contracts," in *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, Nov. 2020.

[18] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "VeriSolid: Correct-by-design smart contracts for Ethereum," in *International Conference on Financial Cryptography and Data Security*. Springer, 2019, pp. 446–465.

[19] Y. Wang, S. K. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, I. Naseer, and K. Ferles, "Formal verification of workflow policies for smart contracts in azure blockchain," in *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 2019, pp. 87–106.

[20] F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F. Vaandrager, "Automata learning through counterexample guided abstraction refinement," in *International Symposium on Formal Methods*. Springer, 2012, pp. 10–27.

[21] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *International Conference on Computer Aided Verification*. Springer, 2000, pp. 154–169.

[22] C. De la Higuera, *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010, vol. 24, no. 3-4.

[23] T.-D. B. Le and D. Lo, "Deep specification mining," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 106–117.

[24] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE transactions on Computers*, vol. 100, no. 6, pp. 592–597, 1972.

[25] I. Krka, Y. Brun, and N. Medvidovic, "Automatic mining of specifications from invocation traces and method invariants," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 178–189.

[26] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 501–510.

[27] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," in *Computer Aided Verification*, vol. 97, 1997, pp. 72–83.

[28] "Daikon," http://plse.cs.washington.edu/daikon/, 2021, the Daikon invariant detector.

[29] C. Lee, F. Chen, and G. Roşu, "Mining parametric specifications," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 591–600.

[30] S. M. Beillahi, G. Ciocarlie, M. Emmi, and C. Enea, "Behavioral simulation for smart contracts," in *Proceedings of the 41st ACM SIGPLAN*

Conference on Programming Language Design and Implementation, 2020, pp. 470–486.
[31] K.-T. Cheng and A. S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *30th ACM/IEEE Design Automation Conference*. IEEE, 1993, pp. 86–91.
[32] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang, "Automated abstraction refinement for model checking large state spaces using sat based conflict analysis," in *International Conference on Formal Methods in Computer-Aided Design*. Springer, 2002, pp. 33–51.
[33] "Dicether: A secure dice game," https://dicether.github.io/paper/paper.pdf, 2018.
[34] Y. Liu, C. Zhang *et al.*, "Automated invariant generation for solidity smart contracts," *arXiv preprint arXiv:2401.00650*, 2024.
[35] L. d. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.
[36] "SuperRare," https://www.dapp.com/app/SuperRare, 2022.
[37] "MoonCatRescue," https://dappradar.com/ethereum/games/mooncatrescue, 2022.
[38] "Quicknode," https://www.quicknode.com/, 2023.
[39] D. Lo and S.-C. Khoo, "Quark: Empirical assessment of automaton-based specification miners," in *2006 13th Working Conference on Reverse Engineering*. IEEE, 2006, pp. 51–60.
[40] T.-D. B. Le, X.-B. D. Le, D. Lo, and I. Beschastnikh, "Synergizing specification miners through model fissions and fusions (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 115–125.
[41] D. Lo, L. Mariani, and M. Santoro, "Learning extended fsa from software: An empirical assessment," *Journal of Systems and Software*, vol. 85, no. 9, pp. 2063–2076, 2012.
[42] G. De Caso, V. Braberman, D. Garbervetsky, and S. Uchitel, "Automated abstractions for contract validation," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 141–162, 2010.
[43] F. Guth, V. Wüstholz, M. Christakis, and P. Müller, "Specification mining for smart contracts with automatic abstraction tuning," *arXiv preprint arXiv:1807.07822*, 2018.
[44] "Bug report in defective-component-counter smart contract," https://github.com/Azure-Samples/blockchain/issues/278, 2024.
[45] "Bug report in digital-locker smart contract," https://github.com/Azure-Samples/blockchain/issues/279, 2024.
[46] "Bug report in hello-blockchain smart contract," https://github.com/Azure-Samples/blockchain/issues/280, 2024.
[47] "Bug report in simple-marketplace smart contract," https://github.com/Azure-Samples/blockchain/issues/281, 2024.
[48] "LearnLib–an open framework for automata learning," https://learnlib.de/, 2022.
[49] "Mythril," https://github.com/ConsenSys/mythril, 2019, a Security Analysis Tool for EVM Bytecode.
[50] H. Wang, Y. Liu, Y. Li, S.-W. Lin, C. Artho, L. Ma, and Y. Liu, "Oracle-supported dynamic exploit generation for smart contracts," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1795–1809, 2020.
[51] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 557–560.
[52] C. Shou, S. Tan, and K. Sen, "Ityfuzz: Snapshot-based fuzzer for smart contract," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 322–333.
[53] M. Yang, D. Lie, and N. Papernot, "Exploring strategies for guiding symbolic analysis with machine learning prediction," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2024, pp. 659–669.
[54] "Cryptopunks," https://www.larvalabs.com/cryptopunks/, 2023.
[55] "CryptoPunks: Collectible characters on the Ethereum blockchain," https://github.com/larvalabs/cryptopunks/tree/master/test, 2017.
[56] B. Tan, B. Mariano, S. K. Lahiri, I. Dillig, and Y. Feng, "Soltype: refinement types for arithmetic overflow in solidity," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–29, 2022.
[57] J. Liu, Y. Chen, B. Tan, I. Dillig, and Y. Feng, "Learning contract invariants using reinforcement learning," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–11.
[58] S. J. Wang, K. Pei, and J. Yang, "Smartinv: Multimodal learning for smart contract invariant inference," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 126–126.
[59] N. Walkinshaw, R. Taylor, and J. Derrick, "Inferring extended finite state machine models from software executions," *Empirical Software Engineering*, vol. 21, no. 3, pp. 811–853, 2016.
[60] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 267–277.
[61] E. M. Gold, "Language identification in the limit," *Information and control*, vol. 10, no. 5, pp. 447–474, 1967.
[62] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and computation*, vol. 75, no. 2, pp. 87–106, 1987.