Verifying Declarative Smart Contracts

Haoxian Chen
ShanghaiTech University
Shanghai, China
hxchen@shanghaitech.edu.cn

Lan Lu
University of Pennsylvania
Philadelphia, PA, USA
lanlu@seas.upenn.edu

Brendan Massey
University of Pennsylvania
Philadelphia, PA, USA
masseybr@seas.upenn.edu

Yuepeng Wang
Simon Fraser University
Burnaby, BC, Canada
yuepeng@sfu.ca

Boon Thau Loo
University of Pennsylvania
Philadelphia, PA, USA
boonloo@seas.upenn.edu

ABSTRACT

Smart contracts manage a large number of digital assets nowadays. Bugs in these contracts have led to significant financial loss. Verifying the correctness of smart contracts is, therefore, an important task. This paper presents an automated safety verification tool, DCV, that targets declarative smart contracts written in DeCon, a logic-based domain-specific language for smart contract implementation and specification. DCV proves safety properties by mathematical induction and can automatically infer inductive invariants using heuristic patterns, without annotations from the developer. Our evaluation on 23 benchmark contracts shows that DCV is effective in verifying smart contracts adapted from public repositories, and can verify contracts not supported by other tools. Furthermore, DCV significantly outperforms baseline tools in verification time.

ACM Reference Format:
Haoxian Chen, Lan Lu, Brendan Massey, Yuepeng Wang, and Boon Thau Loo. 2024. Verifying Declarative Smart Contracts. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639203

1 INTRODUCTION

Smart contracts are programs that process transactions on blockchains – a type of decentralized and distributed ledgers. The combination of smart contracts and blockchains has enabled a wide range of innovations in many fields including banking [12], trading [13, 35], and financing [45], etc.

Nowadays, smart contracts are collectively managing a massive amount of digital assets [1]. However, alongside their widespread adoption, they have also suffered from security vulnerabilities [1–3], resulting in significant financial losses for users and organizations.

One of the key challenges with smart contracts is that once they are deployed and executed on blockchains, terminating their execution or updating the contracts becomes extremely difficult. This lack of flexibility can be particularly problematic when new vulnerabilities are discovered, as it limits the ability to rectify potential issues in deployed contracts.

Given these challenges and the potential financial risks involved, the need to formally verify the correctness of smart contracts before their deployment becomes increasingly crucial.

Existing formal verification approaches often directly verify the implementation of smart contracts by symbolically executing the compiled EVM bytecode [6, 26, 28, 32, 36]. While this approach is general, allowing it to be applied to all existing EVM-based smart contracts without modification, modeling the intricacies of the EVM stack introduces additional complexity, thus limiting the scalability of these approaches. Moreover, high-level properties are hard to be specified and checked on the bytecode as it lacks the high-level structure of the contracts.

In contrast, model-based verification approaches can achieve better scalability by specifying a formal model of the smart contract separately from its implementation. With this formal model and the implementation, two primary verification problems are addressed: (1) Does the formal model satisfy the desired properties [17, 33]? (2) Is the implementation consistent with the formal model [20]? While this verification approach is generally more efficient, as the formal model abstracts away implementation details irrelevant to the verification task, it does require additional effort from the user to specify the formal model. Additionally, the steep learning curves of formal specification languages may limit the adoption of such verification approach.

In this paper, we aim to achieve a good balance between the efficiency and the usability of smart contract verification, by leveraging the concept of executable specification for smart contracts. In particular, we target smart contracts written in DeCon [18], a domain-specific language for smart contract specification and implementation. A DeCon contract is a declarative specification for the smart contract by itself, making it more efficient to reason about than the low-level implementation in Solidity. It is also executable, in that it can be automatically compiled into a Solidity program which can be deployed and run on the Ethereum blockchain. Automatic code generation based on the verified specification can save developers the manual effort of implementing the contract. The

[1] According to Etherscan, as of the writing of this paper, the top ERC20 tokens are managing billions of dollars worth of tokens.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639203

high-level abstraction and executability of DeCon make it an ideal target for verifying contract-level properties.

We implement a prototype, DCV (DeCon Verifier), for verifying declarative smart contracts. Properties are specified as declarative queries for safety violations in the DeCon language. DCV verifies safety invariants using mathematical induction on the sequence of transactions. A typical challenge in induction is to infer inductive invariants that can help prove the target property. Our key insight for addressing this challenge is that the DeCon language exposes the exact logical predicates necessary for constructing inductive invariants, which makes inductive invariant inference tractable.

Another benefit of using DeCon is that it provides uniform interfaces for both contract implementation and property specification. Specifically, DeCon models the smart contract states as relational databases, and properties as violation queries against these databases. Thus, developers can specify both the contract logic and its properties in a declarative and succinct way, and finish the verification and implementation automatically.

This paper makes the following contributions.

- A verification method for smart contracts, targeting contract-level safety invariants based on a declarative specification language and the induction proof strategy (Sections 4, 5).
- A domain-specific adaptation of the Houdini algorithm [29] to infer inductive invariants for automated proof (Section 5).
- An open-source verification tool for future study and comparison [2].
- Evaluation that compares DCV with state-of-the-art verification tools, on 23 representative benchmark smart contracts. Specifically, DCV successfully verifies all benchmarks, including the ones not supported by other tools. Furthermore, it is significantly more efficient than other tools in terms of verification time (Section 6).

2 ILLUSTRATIVE EXAMPLE

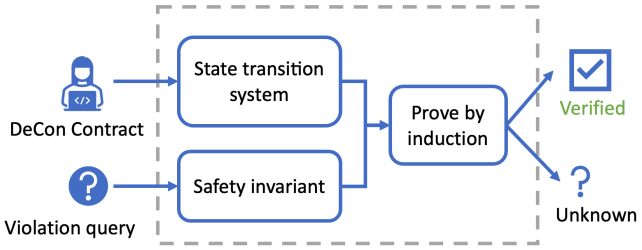

Figure 1: Overview of DCV.

Figure 1 presents an overview of DCV. It takes a smart contract and a property specification (in the form of a violation query) as input, both of which are written in the DeCon language (Section 3). The smart contract is then translated into a state transition system, and the property is translated into a safety invariant on the system states. DCV then verifies that the transition system preserves the safety invariant by mathematical induction. In our prototype, the verification is performed by Z3 [10], an automated theorem prover.

[2]Benchmarks are provided in supplementary materials. Source code will be released after publication.

If the verification succeeds, DCV guarantees that the smart contract is safe by ensuring that the violation query result is always empty, and returns an inductive invariant as a proof. However, if the verification fails, DCV returns "unknown", indicating that the smart contract may not satisfy the specified safety invariant.

In the rest of this section, we use a voting contract as an example to illustrate the workflow of DCV. This example is adapted from the voting example in Solidity [7], simplified for ease of exposition.

2.1 A Voting Contract

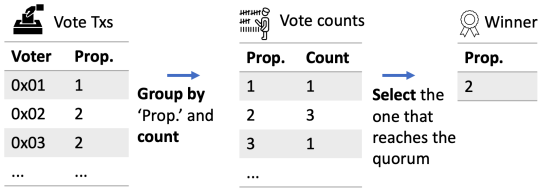

Figure 2: A voting contract

```
1  /* Declare relations. */
2  .decl recv_vote(proposal: uint)
3  .decl vote(p: address, proposal: uint)
4  .decl isVoter(v: address, b: bool)[0]
5  .decl votes(proposal: uint, c: uint)[0]
6  .decl wins(proposal: uint, b: bool)[0]
7  .decl voted(p: address, b: bool)[0]
8  .decl *winningProposal(proposal: uint)
9  .decl *hasWinner(b: bool)
10 .decl *quorumSize(q: uint)
11 .init isVoter
12
13 /* Voter v cast a vote to proposal p. */
14 vote(v,p) :- recv_vote(p), msgSender(v),
15              hasWinner(false), voted(v, false),
16              isVoter(v, true).
17
18 /* Count votes for each proposal p. */
19 votes(p,c) :- vote(_,p), c = count: vote(_,p).
20
21 /* A proposal wins by reaching a quorum. */
22 wins(p, true) :- votes(p,c), quorumSize(q),
23                  c >= q.
24 hasWinner(true) :- wins(_,b), b==true.
25 winningProposal(p) :- wins(p,b), b==true.
26 voted(v,true) :- vote(v,_).
27
28 /* Safety: at most one winning proposal. */
29 .decl inconsistency(p1: uint, p2: uint)[0,1]
30 .violation inconsistency
31 inconsistency(p1,p2) :- wins(p1,true),
32                         wins(p2,true),p1!=p2.
```

Listing 1: A smart contract for voting, written in DeCon [18].

Figure 2 illustrates a voting scenario in a declarative view (i.e., everything is represented as a relational table):

(1) Participants cast votes by sending `vote` transactions, and these transaction records are stored in the "Vote Txs" table (on the left), with the voter address and proposal ID ("Prop.") listed as columns.

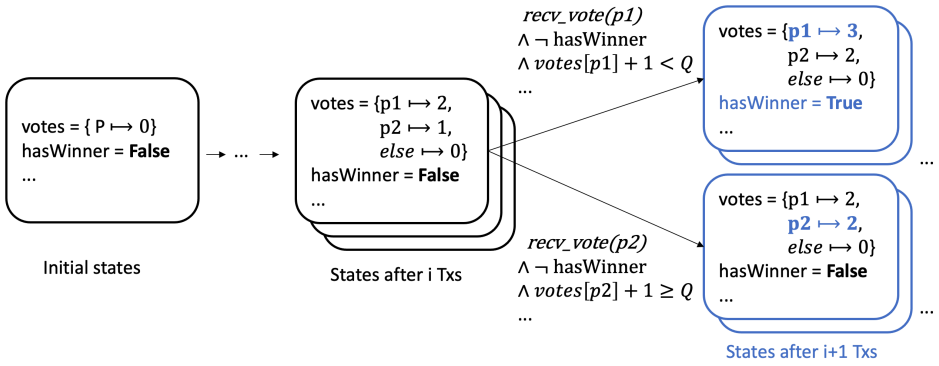

Figure 3: The voting contract as a state transition system.

(2) For each proposal p, its votes are counted by grouping the entries in the "Vote Txs" table by the "Prop." column, and then counting the number of entries within each group. The counting results are displayed in the "Vote counts" table (middle).

(3) The proposal that first reaches a quorum is declared as the winner. Suppose there are 5 participants and the quorum size is 3, proposal 2 is selected as the winner as it gets 3 votes.

2.2 Smart contract written in DeCon language

Listing 1 shows the implementation of this voting contract in DeCon [18], which consists of three major components:

(1) Relation declaration and annotation. The relations shown in Figure 2, along with other auxiliary relations, are declared in lines 1 to 10 of Listing 1. These declarations define the table schema in relational databases, where each schema consists of the table name followed by column names and types in parentheses. Optionally, a square bracket annotates the index of the primary key columns, indicating that these columns uniquely identify a row. For example, the relation `votes(proposal: uint, c: uint)[0]` on line 5 has the first column, proposal ID, as the primary key because votes are counted for each unique proposal. If no primary keys are annotated, all columns are interpreted as primary keys, meaning that the table is a set of tuples.

A special kind of relation is a singleton relation, annotated by $*$. Singleton relations only have one row, such as `winningProposal` in line 8.

By default all relational tables are initialized empty, except relations annotated by the `init` keyword (line 11). These relations are initialized by the constructor arguments passed during deployment.

(2) Relation definition in inference rules. Each relation is defined in the form of a rule, `head :- body`. Similar to the rules used in Datalog programs, the body consists of a list of relational literals, and is evaluated to true if and only if there exists a valuation of all variables such that each literal has the corresponding concrete entries in the table. If the body is true, the head is inserted into the corresponding table.

For instance, the rule in line 14 specifies that a `vote` transaction can be committed if there is no winner yet (`hasWinner(false)`), the message sender is a registered voter (`isVoter(v,true)`), and the voter has not voted before (`voted(v,false)`). The literal `recv_vote(p)` represents a transaction handler that evaluates to true upon receiving a `vote` transaction request. Rules that contain such transaction handlers (literal with a `recv_` prefix in the relation name) are referred to as transaction rules. Committing a transaction inserts a new entry into the transaction table ("Vote Txs" in Figure 2).

Inserting a new `vote(v,p)` literal also triggers updates to all its direct dependent rules. A rule is considered directly dependent on a relation R if and only if a literal of relation R is in its body. In this case, relation `votes` and `voted` are updated next. The chain of dependent rule updates continues until no further dependent rules can be triggered, and the transaction handling is finished. Using this mechanism, the votes for each proposal, as well as the winning proposal, are automatically updated when new votes are approved.

On the other hand, if the body of a transaction rule evaluates to false upon receiving a transaction request, the transaction request is rejected, and no updates are made to any of the affected relations.

(3) Properties as violation query. Line 31 specifies a safety property as another relation, which is further annotated as a violation query in line 32. This relation is defined by the rule in line 33. If the rule is evaluated to true, it means that there exists two different winning proposals, indicating a violation to the safety invariant that there is at most one winning proposal. Such violation query rule is expected to be always false during the execution of a correct smart contract.

2.3 Translating DeCon Contract to State Transition System

In order to perform formal verification, DeCon contracts are encoded as state transition systems. The state space comprises all possible valuations of the relational tables, and each successful transaction triggers an atomic state transition step (the transaction atomicity is guaranteed by the underlying Ethereum blockchain). Such encoding naturally captures the semantics of smart contracts: reactive programs that listen and respond to requests (transactions).

Figure 3 illustrates part of the transition system translated from the voting contract in Listing 1. The middle portion (labeled "States after i Txs") shows a state that is reached after i transactions from one of the initial states. At this point in the execution, proposal p_1 has received two votes, proposal p_2 has one, and no winner has

been declared yet. Two outgoing edges from this state are highlighted. The one on top represents a `vote(p1)` transaction, where p_1 receives an additional vote, thereby achieving the quorum and becoming the winning proposal. This transaction can be executed only if certain conditions are met, which are annotated on the edge (only part of the conditions are shown due to space limit). The edge is derived from the transaction rule in Listing 1 line 15 ($recv\_vote(p_1) \wedge \neg hasWinner \wedge ...$), and its dependent rules from line 20 to 28 ($votes[p_1] \geq Q \wedge ...$). This edge leads to a new state where proposal p_1's votes is incremented by one, and it becomes the winner, which is also translated from line 19 to 26.

Similarly, the bottom right shows another transaction where proposal p_2 gets a vote, but *hasWinner* remains *False* since no proposal has reached the quorum.

Section 4.2 formally describes the algorithm to translate a DeCon smart contract into a state transition system.

Property. The violation query rule (line 31) is translated into the following safety invariant:

$$\neg(\exists p_1, p_2.\ wins[p1] \wedge wins[p2] \wedge p1 \neq p2) \quad (1)$$

It states that there do not exist proposals $p1$ and $p2$ such that the violation query is true, which means there is at most one winning proposal. In the rest of the paper, we will represent predicates in logical form $wins[p]$ instead of the relational form `wins(p,true)` for conciseness.

2.4 Proof by Induction

To prove safety invariants of a smart contract against an infinite sequence of transactions, DCV adopts the mathematical induction approach. Given a state transition system, and a safety invariant, the proof consists of two steps:

(1) Base case: all initial states satisfy the safety invariant.
(2) Induction step: if the safety invariant holds for some state s, then it also holds for all possible next states s'.

One of the biggest challenges in automatic induction proof is finding inductive invariants. In some cases, a true safety invariant may not be inductive, which means that although the safety invariant is true for all possible states, it can still fail the induction proof step. To successfully complete the induction step, an inductive invariant $inv(s)$ needs to be found, such that $inv(s) \wedge prop(s)$ is inductive, where s is the state variable.

For example, the safety invariant in Equation 1 cannot be proved inductively on its own. Because the verifier cannot eliminate a spurious counterexample in the induction step: after a proposal is declared winner, another proposal receives a new vote and becomes another winner. We need extra inductive invariants to rule out such spurious counterexamples:

$$\forall u \in Proposal.\ wins[u] \implies hasWinner \quad (2)$$

which asserts that if any proposal $u \in Proposal$ is marked as the winner, the predicate *hasWinner* must also be true. Since the vote rule requires $\neg hasWinner$, such counterexample is unreachable and the induction proof is successful.

Inductive invariants are typically inferred in a guess-and-check manner [29], where a set of candidate invariants are enumerated until an inductive one is found. However, such approaches heavily rely on good heuristics to generate a set of candidate invariants.

Decl	:=	*.decl R*
Annot	:=	*.(init | violation | public) R*
(Relation) R	:=	*SR | SG | TR*
(Simple) SR	:=	*.decl Str(Schema)[K]*
(Singleton) SG	:=	*.decl * Str(Schema)*
(Transaction) TR	:=	*.decl * recv_Str(Schema)*
(Primary keys) K	:=	*[k1, k2, ...]*
Schema	:=	*Str : T1, Str : T2, ...*
(Type) T	:=	*address | uint | int | bool*

Figure 4: Syntax of DeCon relation declaration and annotation.

The insight of DCV is that, rules in DeCon contracts provide a concise and high-quality source of logical predicates for constructing inductive invariants. Since they only concern high-level logic and do not include implementation details, the extracted predicates are much smaller in size than those extracted from lower-level implementations, greatly speeding up the invariant search process. For instance, the predicates in Equation 2 ($wins[u]$, *hasWinner*), are presented in the rules in Listing 1. We describe the details of predicate extraction and inductive invariant generation in Section 5.

3 THE DECON LANGUAGE

A DeCon contract consists of three main blocks: (1) Relation declarations, (2) Relation annotations, and (3) Rules.

(Contract) P := Decl | Annot | Rule

Relation declarations. As shown in Figure 4, there are three kinds of relation declaration syntax:

- Simple relations (*SR*) have a string for a relation name, followed by a schema in parenthesis, and optional primary key indices in a square bracket. The schema consists of a list of column names and types. When inserting a new tuple to a table, if a row with the same primary keys exists, then the row is replaced by the new tuple.
- Singleton relations (*SG*) are relations annotated with a `*` symbol. These relations have only one row. Row insertion is also an update for singleton relations.
- Transaction relations (*TR*) are relations with prefix *recv_*, interpreted as an event trigger for incoming transaction requests. For example, in line 15 Listing 1, the literal `recv_vote(p)` is triggered to be true when the contract receives a `vote` transaction, with parameter `p` being the proposal ID.

In addition, special relations are reserved for blockchain environment. For example, relation `*msgSender(p: address)` stores incoming message sender address. DeCon also reserves relations for message values, current block number, contract constructors, etc. Reserved relations cannot be declared by programmers.

Relation annotations. Three kinds of relation annotations are supported: (1) `init` indicates that the relation is initialized by a constructor argument passed during deployment, (2) `violation` means that the relation represents a safety violation query, and

$$
\begin{array}{rl}
\mathit{Rule} & := H(\bar{x}) :\text{-}\ \mathit{Body} \\
\mathit{Body} & := \mathit{Join} \mid R(\bar{x}), y = \mathit{Agg} : R(\bar{y}) \\
\mathit{Join} & := R(\bar{x}) \mid \mathit{Pred}, \mathit{Join} \\
\mathit{Agg} & := \mathit{sum}\ n \mid \mathit{max}\ n \mid \mathit{min}\ n \mid \mathit{count} \\
\mathit{Pred} & := R(\bar{x}) \mid C(\bar{x}) \mid y = F(\bar{x}) \\
(\mathit{Condition})\ C & :=> \mid < \mid \geq \mid \leq \mid \neq \mid == \\
(\mathit{Function})\ F & := + \mid - \mid \times \mid /
\end{array}
$$

Figure 5: Syntax of DeCon rules. $H(\bar{x})$ and $R(\bar{x})$ are relational literals, with H and R being the relation name, and $\bar{x}$ is an array of variables or constants. For the *max*, *min*, and *sum* aggregators, n is a variable in a numerical domain.

(3) `public` generates a public interface to read the contents of the corresponding relational table.

Rules. Figure 5 shows the syntax of DeCon rules. A DeCon rule is of the form *head* :- *body*, which is interpreted from right to left: if the body is true, then it inserts the head tuple into the corresponding relational table.

A rule body is a conjunction of literals, and is evaluated to true if there exists a valuation of variables $\pi : V \mapsto D$ such that all literals are true. π maps a variable $v \in V$ to its concrete value in domain D. Given a variable valuation π, a relational literal is evaluated to true if and only if there exists a matching row in the corresponding relational table. Other kinds of literals, including conditions, functions, and aggregations, are interpreted as constraints on the variables.

In particular, DeCon supports three kinds of rules. Their differences in syntax and semantics are described as follows:

(1) Join rules are rules that have a list of predicates in the rule body, and contain at least one relational literal. A predicate can be either a relational literal, a condition, or a function,

(2) Transaction rules are a special kind of join rules that have one special literal in the body: transaction handlers. A transaction handler literal has `recv_` prefix in its relation name, and is evaluated to true when the corresponding transaction request is received. The rest of the rule body specifies the approving condition for the transaction,

(3) Aggregation rules are rules that contain a relational literal $R(\bar{x})$ and an aggregator $y = \mathit{Agg}\ n : R(\bar{y})$, where *Agg* can be either *max*, *min*, *count*, or *sum*. For each valid valuation of variables in $R(\bar{x})$, it computes the aggregate on the matching rows in $R(\bar{y})$. Take the following rule from the voting contract as an example.

```
votes(p,c):-vote(_,p),c=count:vote(_,p).
```

For each unique value p in the second column of table `vote`, the aggregator `c = count: vote (_,p)`, counts the number of rows in table `vote` whose second column equals p.

DeCon contracts are executable on the Ethereum blockchain [18]. To execute, they are first compiled into Solidity [8], which is further compiled to bytecode for the Ethereum blockchain.

Expressiveness. DeCon is able to express a wide range of smart contracts (Table 1), including the most popular open standards like ERC20 and ERC721. Certain functions, such as cryptographic hash functions and randomized functions, fall beyond the scope of relational logic, and are thus not supported by DeCon. Verifying such smart contracts with DCV's approach would require a modeling of these computations that is precise and practically verifiable, which is an intriguing direction for future research. DeCon also explicitly disallows recursion to ensure predictable and cost-effective gas consumption, but recursion is rare and not recommended as a good practice in smart contract development [18].

Algorithm 1 EncodeRule(r, R, Γ, τ).

Input: (1) A DeCon rule r, (2) the set of all DeCon rules R, (3) a map from relation to its modeling variable Γ, (4) a trigger τ, the newly inserted literal that triggers r's update.
Output: A formula over $S \times S$, encoding r's body condition, and all state updates triggered by inserting r's head literal.

1: $\mathit{Body} \leftarrow$ EncodeRuleBody(Γ, τ, r)
2: $\mathit{Dependent} \leftarrow$ {EncodeRule(dr, R, Γ, τ') | $(dr, \tau') \in$ DependentRules(r, R, τ)}
3: $(H, H') \leftarrow$ GetStateVariable($\Gamma, r.\mathit{head}$)
4: $\mathit{Update} \leftarrow$ GetUpdate(H, r, tau)
5: $\mathit{TrueBranch} \leftarrow \mathit{Body} \wedge (H' =$ Update$) \wedge (\bigwedge_{d \in \mathit{Dependent}} d)$
6: $\mathit{FalseBranch} \leftarrow \neg \mathit{Body} \wedge (H' = H)$
7: **return** $\mathit{TrueBranch} \oplus \mathit{FalseBranch}$

4 PROGRAM TRANSFORMATION

4.1 Declarative Smart Contracts as Transition Systems

This section introduces the algorithm to translate a DeCon smart contract into a state transition system $\langle S, I, E, Tr \rangle$ where

- S is the state space: the set of all possible valuations of all relational tables in DeCon.
- $I \subseteq S$ is the set of initial states that satisfy the initial constraints of the system. All relations are by default initialized to zero, or unconstrained if they are annotated to be initialized by constructor arguments.
- E is the set of transaction types. Each element in E corresponds to a type of transaction in DeCon (analogous to a transaction function definition in Solidity).
- $Tr \subseteq S \times E \times S$ is the transition relation, generated from DeCon rules. $Tr(s, e, s')$ means that state s can transit to state s' via transaction e.

In the rest of this section, we introduce the algorithm to generate the transition relation from a DeCon smart contract.

4.2 Transition Relation

The transition relation Tr is defined by a formula $tr : S \times E \times S \mapsto \mathit{Bool}$. Given $s, s' \in S, e \in E$, s can transition to s' in one step via transaction type e if and only if $tr(s, e, s')$ is true. Equation 3 defines tr as a disjunction over the set of formulas encoding each transaction rule. The procedure EncodeRule is defined by Algorithm 1.

$$tr \triangleq \bigvee_{r \in TR} [\text{EncodeRule}(r, R, \Gamma, r.\mathit{trigger}) \wedge e = r.\mathit{TxName}] \quad (3)$$

The EncodeRule procedure takes four inputs: (1) a DeCon rule r, (2) the set of all DeCon rules R, (3) a map from relation to its modeling variable Γ, (4) and a trigger τ , the newly inserted literal

$$\frac{\Gamma, \tau \vdash R(\bar{x}) \rightsquigarrow \phi}{\Gamma, \tau \vdash H(\bar{y}) :\text{-} R(\bar{x}) \hookrightarrow \phi} \ (Join1) \qquad \frac{\Gamma, \tau \vdash Pred \rightsquigarrow \phi_1 \quad \Gamma, \tau \vdash H(\bar{y}) :\text{-} Join \hookrightarrow \phi_2}{\Gamma, \tau \vdash H(\bar{y}) :\text{-} Pred, Join \hookrightarrow \phi_1 \wedge \phi_2} \ (Join2)$$

$$\frac{\tau = insert\ R(\bar{z}) \quad s' = \Gamma(H)[\bar{k}].value + n}{\Gamma, \tau \vdash H(\bar{y}) :\text{-} R(\bar{x}), s = sum\ n : R(\bar{z}) \hookrightarrow s = s'} \ (Sum+) \qquad \frac{\tau = delete\ R(\bar{z}) \quad s' = \Gamma(H)[\bar{k}].value - n}{\Gamma, \tau \vdash H(\bar{y}) :\text{-} R(\bar{x}), s = sum\ n : R(\bar{z}) \hookrightarrow s = s'} \ (Sum-)$$

$$\frac{\tau = insert\ R(\bar{z}) \quad \phi \coloneqq c = \Gamma(H)[\bar{k}].value + 1}{\Gamma, \tau \vdash H(\bar{y}) :\text{-} R(\bar{x}), c = count : R(\bar{z}) \hookrightarrow \phi} \ (Count+) \qquad \frac{\tau = delete\ R(\bar{z}) \quad \phi \coloneqq c = \Gamma(H)[\bar{k}].value - 1}{\Gamma, \tau \vdash H(\bar{y}) :\text{-} R(\bar{x}), c = count : R(\bar{z}) \hookrightarrow \phi} \ (Count-)$$

$$\frac{\tau = insert\ R(\bar{z}) \quad m' = \Gamma(H)[\bar{k}].value}{\Gamma, \tau \vdash H(\bar{y}) :\text{-} R(\bar{x}), m = max\ n : R(\bar{z}) \hookrightarrow ite(n > m', m = n, m = m')} \ (Max)$$

$$\frac{\tau = insert\ R(\bar{z}) \quad m' = \Gamma(H)[\bar{k}].value}{\Gamma, \tau \vdash H(\bar{y}) :\text{-} R(\bar{x}), m = min\ n : R(\bar{z}) \hookrightarrow ite(n < m', m = n, m = m')} \ (Min)$$

Figure 6: Inference rules for the EncodeRuleBody procedure.

that triggers r's update. In particular, a trigger τ takes the form `insert [literal]` or `delete [literal]`. This procedure is invoked recursively to encode all dependent rules of a transaction into a constraint. The initial inputs are the transaction rules, and the trigger `insert [recv_tx]` representing a new incoming transaction request. R and Γ remain unchanged across invocations. The procedure works as follows.

(1) Encode individual rules. In step 1, r's body is encoded as a boolean formula, *BodyConstraint*, by calling a procedure *EncodeRuleBody* (Section 4.3). Take the rule for `vote` transaction in line 15 of Listing 1 as an example. Its body is encoded as:

$$\neg hasWinner \wedge \ \neg hasVoted[v] \wedge \ isVoter[v]$$

(2) Encode dependent rules. Step 2 first selects direct dependent rules of r from the set of all DeCon rules R, by calling a subroutine *DependentRules*(r, R, τ). It returns a set of tuple (dr, τ'), where dr is a direct dependent rule of r, and τ' is the corresponding trigger for dr. A rule dr is directly dependent on rule r if and only if r's head relation appears in dr's body. Following the example in Listing 1, rules for `votes` (line 19) and `voted` (line 26) both depend on `vote`, because they contain `vote` in the body.

Triggers for dependent rules are generated as follows. If τ is insertion, the next trigger τ' is also insertion of the rule head literal. For instance, τ =`insert recv_vote(p)` results in τ' =`insert vote(v,p)`, by the `vote` rule (line 14).

In addition, when inserting a new literal with primary keys, existing literals with the same primary key need to be deleted. For example, relation `votes` has the proposal as the primary key, when its count is incremented, both `insert votes(p,n+1)` and `delete votes(p,n)` will be returned as τ'.

If τ is deletion, then τ' is deletion of the head literal.

(3) Generate update constraints. Step 3 generates state variables for head relation, where H and H' are for the current and next step respectively. Step 4 generates the head relation update constraint:

$$\text{GetUpdate}(H, r, \tau) = \begin{cases} H.insert(r.head), & \text{if } \tau = \text{insert } \_ \\ H.delete(r.head), & \text{if } \tau = \text{delete } \_ \end{cases} \quad (4)$$

where concrete forms of `H.insert` and `H.delete` methods depend on the modeling variable's type. For instance, relation `votes` is modeled as mapping `Proposal => uint`. By the `votes` rule (line 19), `insert vote(p,v)` results in the update: $Store(votes, p, votes[p] + 1)$.

Step 5 and 6 get constraints for the true and false branches of the rule derivation, respectively. Step 7 returns the final formula as an exclusive-or of the true and false branches, which encodes r's body and how its update affects other relations in the contract.

4.3 Encoding Rule Bodies

The procedure EncodeRuleBody is defined by two sets of inference rules:

- $\Gamma, \tau \vdash r \hookrightarrow \phi$ states that a DeCon rule r is encoded by a boolean formula ϕ under context Γ and τ.
- $\Gamma, \tau \vdash Pred \rightsquigarrow \phi$ states that a predicate *Pred* is encoded by a formula ϕ under context Γ and τ.

The contexts (Γ and τ) of both judgments are defined in the same way as the input of Algorithm 1.

Figure 6 shows the inference rules that define the first judgment $\Gamma, \tau \vdash r \hookrightarrow \phi$. They are interpreted as follows.

Join rules are encoded as conjunctions of body predicates, each of which is encoded from a literal in the rule body. The encoding of individual literals is introduced later in this section.

Sum and Count have separate inference rules for tuple insertion (+) and deletion (−), where $\bar{k}$ represents the primary keys of relation H, extracted from the array $\bar{y}$, and $\Gamma(H)[\bar{k}].value$ reads the current aggregate result. Take this rule for instance:

```
votes(p,c) :- vote(_,p), c = count: vote(_,p).
```

with τ = `insert vote(v,p)`. It is encoded as $c = votes[p].value + 1$.

Max and Min are encoded as conditional constraints. If the matching field n in the inserted tuple $R(\bar{x})$ is greater (resp. smaller) than the current maximum (resp. minimum) m', then the new maximum (resp. minimum) is n. Otherwise it remains the same.

Note that they are only encoded for tuple insertions, based on the assumption that they only apply to transaction relations (tables that stores the transaction records), which are append only and have no primary keys. In other words, they have no tuple deletion.

This assumption is made for two reasons. First, updating *Max* and *Min* for tuple deletion is complicated, because if the current maximum or minimum is deleted, the second largest or smallest element needs to be fetched and become the new aggregation result. Such update requires storing the whole table and even maintaining

sorted table entries. Second, Ethereum has strict limits on the computation and storage of each smart contract and its transactions. Maintaining maximum and minimum for tables with delete operation is very expensive to be executed on Ethereum. We survey smart contracts in public repositories and find no contract with such logic. Therefore, DCV adds such assumption and greatly simplifies the rule encoding.

Encoding individual literals. Following are the inference rules for judgment: $\Gamma, \tau \vdash Pred \rightsquigarrow \phi$, which encodes individual literals.

$$\frac{\tau.rel = R}{\Gamma, \tau \vdash R(\bar{x}) \rightsquigarrow True}\ (Lit1) \qquad \frac{\tau.rel \neq R}{\Gamma, \tau \vdash R(\bar{x}) \rightsquigarrow \Gamma(R)[\bar{k}] = \bar{v}}\ (Lit2)$$

where $\bar{k}$ represents the primary keys in relational literal $R(\bar{x})$, extracted from $\bar{x}$, and $\bar{v}$ represents the remaining fields in $\bar{x}$. When $R(\bar{x})$ is the inserted literal (Lit1), it is encoded as $True$ without any constraints. Otherwise, it is interpreted as a constraint where the value $R[\bar{k}]$ matches $\bar{v}$ (Lit2).

$$\frac{}{\Gamma, \tau \vdash C \rightsquigarrow C}\ (Condition) \qquad \frac{}{\Gamma, \tau \vdash y = F(\bar{x}) \rightsquigarrow y = F(\bar{x})}\ (Function)$$

Conditions and functions are directly encoded as they are, as shown in the above rules.

Rule derivation and recursion. Rule recursion means that a rule is dependent on itself. A rule r_a is dependent to another rule r_b ($r_a \rightarrow r_b$) if r_b's head relation appears in r_a's body. This dependency relation is transitive: $r_a \rightarrow r_b \wedge r_b \rightarrow r_c \implies r_a \rightarrow r_c$. Using this dependency annotation ($\rightarrow$), rule recursion means $r_a \rightarrow \ldots \rightarrow r_a$.

Different from traditional Datalog, where recursion is a powerful feature to concisely express sophisticated queries, DeCon prohibits recursion for gas efficiency reasons [18]. Therefore, DCV only considers non-recursive rules. The absence of recursion keeps the size of the transition constraint linear to the number of rules in the DeCon contract, thus making the safety verification tractable.

Blockchain environment variables, including sender address and value of transactions, block number, address of the contracts, are modeled as symbolic constants. Since DCV focuses on verifying contract logic designs, addresses and integers are modeled as mathematical integers (unbounded), which allows more efficient reasoning with Z3's integer theory.

Multi-contract Interactions are specified implicitly by DeCon rules that join relations from different contracts. Such interactions are performed via message passing. Unlike prior work checking for message handling errors, DCV assumes message delivery and handling are always successful, and instead focuses on the functional correctness. Note that such interactions are limited to functions without mutual recursions. Mutual recursions are not supported because it breaks the atomicity assumption of a transaction.

4.4 Safety Invariant Generation

Each violation query rule qr in a DeCon contract is first encoded as a formula ϕ such that $\Gamma, \tau \vdash qr \hookrightarrow \phi$. Note that the context Γ is the same mapping used in the transition system encoding process. The second context, trigger τ, is a reserved literal $check()$, which triggers the violation query rule after every transaction.

Next, the safety invariant is generated from ϕ as follows:

$$Prop \triangleq \neg(\exists x \in X.\ \phi(s, x))$$

Algorithm 2 Procedure to find inductive invariants.

Input: a transition system ts, a map from relation to its modeling variable Γ, and a set of DeCon transaction rules R.
Output: an inductive invariant of ts.

```
function FindInductiveInvariant(C,ts)
    for inv in C do:
        if refuteInv(inv, C, ts) then
            return FindInductiveInvariant(C \ inv, ts)
        end if
    end for
    return ⋀_{c_i ∈ C} c_i
end function
P ← ⋃_{r ∈ R} ExtractPredicates(r, Γ)
C ← GenerateCandidateInvariants(P)
return FindInductiveInvariant(C, ts)
```

where X is the state space for the set of non-state variables in ϕ. The property states that there exists no valuations of variables in X such that the violation query is non-empty. In other words, the system is safe from such violation.

5 VERIFICATION METHOD

5.1 Proof by Induction

Given the state transition system translated from the DeCon smart contract, the target property $prop(s)$, which is translated from the violation query, is proven by mathematical induction. In particular, let S be the set of states in the transition system, and E be the set of transaction types (`vote` is the only transaction type in the example in Listing 1). Given $s, s' \in S, e \in E$, let $init(s)$ indicate whether s is in the initial state, and $tr(s, e, s')$ indicate whether s can transition to s' via transaction type e. The mathematical induction is as follows:

$$\begin{aligned}
\text{ProofInd}(init, tr, prop) &\triangleq \text{Base}(init, prop) \\
&\quad \wedge \text{Induction}(tr, prop) \\
\text{Base}(init, prop) &\triangleq \forall s \in S.\ init(s) \\
&\quad \implies inv(s) \wedge prop(s) \\
\text{Induction}(tr, prop) &\triangleq \forall s, s' \in S, e \in E. \\
&\quad inv(s) \wedge prop(s) \wedge tr(s, e, s') \\
&\quad \implies inv(s') \wedge prop(s')
\end{aligned} \tag{5}$$

where $inv(s) \wedge prop(s)$ is an inductive invariant inferred by DCV such that $prop(s)$ is proved to be an invariant of the transition system.

Algorithm 2 presents the procedure to infer inductive invariants. It first extracts a set of predicates P from the set of transaction rules R (Section 5.2). Then it generates a set of candidate invariants using predicates in P, following two heuristic patterns (Section 5.3). Finally, it invokes a recursive subroutine FindInductiveInvariant to find an inductive invariant.

The procedure FindInductiveInvariants is adopted from the Houdini algorithm [29]. It iteratively refutes candidate invariants in C, until there is no candidate that can be refuted, and returns the conjunction of all remaining invariants. The subroutine refuteInv is defined in Equation 6, which refutes a candidate invariant if it is

Algorithm 3 ExtractPredicate(r, Γ).

Input: a transaction rule r, a map from relation to its modeling variable Γ.
Output: a set of predicates P.

1: $\tau \leftarrow r.trigger$
2: $P_0 \leftarrow \{p \mid l \in r.body, \Gamma, \tau \vdash l \rightsquigarrow p\}$
3: $P_1 \leftarrow \{p \wedge q \mid p \in P_0, q \in \text{MatchingPredicates}(p, r)\}$
4: **return** $P_0 \cup P_1$

not inductive:

$$\begin{aligned} \text{refuteInv}(inv, C, ts) \triangleq \quad & \vee\neg(ts.init \implies inv) \\ & \vee\neg[(\bigwedge_{c \in C} c) \wedge ts.tr \implies inv'] \end{aligned} \tag{6}$$

where inv' is adopted by replacing all state variables in inv with their corresponding variable in the next transition step.

Given a set of candidate invariants C, this algorithm guarantees to find the strongest inductive invariant that can be constructed in the form of conjunction of the candidates in C [29].

5.2 Predicate Extraction

Algorithm 3 presents the predicate extraction procedure. It first transforms each literal in the transaction rule into a predicate, and puts them into a set P_0. Some predicates in P_0 do not contain enough information on their own, e.g., predicates that contain only free variables, because the logic of a rule is established on the relation among its literals, e.g., two literals sharing the same variable v means joining on the corresponding columns. On the contrary, predicates that contain constants, e.g. hasWinner == true, convey the matching of a column to a certain concrete value, and can thus be used directly in candidate invariant construction.

Therefore, in the next step, each predicate p in P_0 is augmented by one of its matching predicates in $matchingPredicates(p, r)$, which is the set of predicates in rule r that share at least one variable with predicate p. This set of augmented predicates is P_1. Finally, the union of P_0 and P_1 is returned.

5.3 Candidate Invariant Generation

This section introduces the candidate invariant generation algorithm and the design rationale. The goal of the algorithm is to find invariants that, when combined with the property we want to verify, are inductive and sufficient to imply the target property.

A property is inductive if it can be proven using the induction proof in Equation 5. To prove a property by induction, we need to find invariants that can eliminate spurious counterexamples that the verifier might find. Spurious counterexamples are assignments to contract states that can never be reached from the initial state.

Consider the voting contract presented in Listing 1, where the objective is to establish through mathematical induction that there is at most one winner. During the induction step, the verifier may present a spurious counterexample: a participant is declared the winner (`wins[u]==True`), and subsequently, another vote is cast for a different participant, providing enough votes to declare it as a winner as well. One way to suppress this counterexample is to establish the invariant that, when a participant has been declared winner, the variable $hasWinner$ is always true:

$$wins[u] \implies hasWinner$$

Recall that the vote transaction requires `hasWinner` to be false, so this spurious counterexample is prevented from happening.

Invariant template. In the above implication, it is important to note that the premise constitutes a predicate within the target property, while the conclusion is a negation of one of the predicates in transaction rules. This observation is generalized into the following invariant pattern:

$$p \implies \neg q$$

Here, q is instantiated by predicates extracted from transaction rules, and p is by (1) predicates extracted from property specification rules, and (2) negated predicates extracted from initialization rules. Predicates are extracted using Algorithm 3.

For predicates containing primary keys, such as the $wins[u]$ predicate where u represents the primary key of the relation $wins$, DCV incorporates a universal quantifier for the variable u. Thus, the previously mentioned invariant is of the comprehensive form:

$$\forall u \in Proposal.\ wins[u] \implies hasWinner$$

DCV then unions all possible template instantiations into the set of candidate invariants.

6 EVALUATION

Table 1: Benchmark properties for group one.

Benchmarks	Properties
ERC20	Account balances add up to totalSupply.
ERC721	All existing tokens have an owner.
ERC777	No default operator is approved for individual account.
ERC1155	Each token's account balances add up to that token's totalSupply.
wallet	No negative balance.
crowFunding	(1) No missing fund. (2) Mutual exclusion of refund and withdraw. (3) Cannot refund after target amount is raised.
brickBlockToken	(1) No transfer before unpause. (2) No distribute token before sale finalized. (3) No evacuate before upgrade. (4) Always be puased after upgrade.
finalizableCrowdSale	(1) No token sale after finalization. (2) No premature finalization.
cappedCrowdSale	(1) No illegal finalization. (2) No token sale after finalization.
controllableToken	Account balances add up to totalSupply.
partitionToken	Account balances add up to totalSupply in each partition.
paymentSplitter	No overpayment.
vestingWallet	No early release.
voting	At most one winning proposal.
auction	Each participant can withdraw at most once.

Table 2: Verification efficiency measured in time (seconds). TO stands for time-out after 1 hour, OM stands for out of memory, Unknown (?) means the verifier cannot verify the property, and Errors (×) means the verifier exits due to internal errors.

Group	Name	#Rules	LOC	DCV	Inv.?	Solc		Solc-verify		VeriSmart	
						ref.	DeCon	ref.	DeCon	ref.	DeCon
Open standards and examples	ERC20	19	389	0.78		25.07	x	x	64.03	?	0.56
	ERC721	13	520	0.88		TO	x	x	57.21	0.31	?
	ERC777	31	562	0.90		TO	?	22.06	?	?	?
	ERC1155	18	645	0.97		11.14	TO	15.10	64.83	?	?
	wallet	12	67	0.82		0.16	?	4.87	?	?	0.56
	crowFunding-p1	14	85	0.85		1.00	x	?	21.00	TO	?
	crowFunding-p2			1.47	✓	?	?	?	5.00	TO	?
	crowFunding-p3			1.24	✓	?	?	?	?	TO	?
	BrickBlockToken-p1	36	595	0.86	✓	?	?	?	?	1.35	1.15
	BrickBlockToken-p2			1.61	✓	?	?	?	?	1.36	?
	BrickBlockToken-p3			2.18	✓	?	?	?	?	TO	1.26
	BrickBlockToken-p4			2.25		?	?	5.83	1.00	TO	?
	FinalizableCrodSale-p1	22	457	1.29	✓	?	?	?	?	TO	?
	FinalizableCrodSale-p2			0.80		?	?	?		TO	?
	CappedCrowdSalea-p1	25	435	11.13	✓	?	?	?	?	TO	?
	CappedCrowdSalea-p2			1.36	✓	?	?	?	?	TO	?
	paymentSplitter	6	166	1.22		TO	13.94	8.51	?	TO	?
	vestingWallet	7	113	0.82		TO	?	21.62	10.45	?	?
	voting	6	36	0.86	✓	x	TO	?	?	?	?
	auction	13	146	2.27	✓	x	TO	?	?	?	?
	controllableToken	23	55	0.90		43.26	2.72	x	56.00	TO	0.51
	partitionToken	16	70	0.79		0.41	0.31	5.84	6.91	0.16	0.36
Top ERC20 tokens	bnb	24	172	0.86		3.27	0.66	10.06	28.68	?	?
	link	20	308	0.84		0.51	x	64.63	25.08	0.26	0.86
	ltcSwapAsset	25	655	0.80		TO	x	x	50.70	?	1.97
	matic	25	510	0.85		2.26	x	67.37	26.84	?	0.91
	shib	22	508	0.86		0.99	x	70.26	22.41	244.99	0.56
	tether	27	474	0.81		51.00	x	x	30.75	OM	?
	theta	21	213	0.77		321.65	0.91	20.66	19.10	TO	?
	wbtc	28	731	0.82		TO	x	x	59.56	?	0.86
Count	30			30	10	12	5	12	17	6	11

Implementation. We implement the smart contract transformation and inductive invariant generation algorithms in Scala and use Z3 [10] to check the satisfiability of generated formulas. Quantified formulas are handled by Z3's default heuristics.

Benchmarks. We collect 23 benchmark contracts in two groups. The first group consists of 12 contracts from open libraries [5, 7] and examples from prior research [9]. To be included, each contract must meet two criteria: (1) expressible in DeCon language, and (2) has contract-level safety properties annotation, or interpretable documentation. These contract-level properties are invariants that hold across an infinite sequence of transactions. Table 1 shows the contract names and their target properties. The second group comprises eight of the most popular ERC20 contracts, ranked by circulating market cap maintained by Etherscan [12]. Filtering out contracts without source code or having unsupported features, the remaining are verified against the common ERC20 token property.

Baselines. We compare against solc [8], solc-verify [26] and VeriSmart [41]. Solc, the Solidity compiler maintained by the Ethereum community, features a built-in checker for verifying assertions in source programs. We use version 0.8.13 for this experiment. Solc-verify extends solc 0.7.6 with strategies like specification annotation and modular program verification. VeriSmart, a safety verifier, enhances verification efficiency by autonomously discovering transaction invariants. We employ the latest VeriSmart commit [15] compatible with our machine and the latest solc version (0.5.11) supported by the tool. While Verx [36] and Zeus [28] were considered, they are not publicly available.

Experiment setup. We adapted benchmark contracts for compatibility with all comparison tools. Modifications included removing recursion from the delegate vote function in the voting contract, substituting inline assembly in Solidity contracts with native code, and making minor syntax adjustments to meet specific Solidity version requirements. Additionally, VeriSmart underwent preprocessing steps for successful verification, including flattening contracts with external libraries and inlining long chains of function invocations from DeCon-generated contracts.

With these adjustments, we implemented DeCon counterparts for each reference Solidity contract. Verification tasks were performed on three contract versions: (1) DeCon contracts with DCV, (2) reference Solidity contracts with baseline tools, and (3) Solidity contracts generated from DeCon with baseline tools. Verification time for each task was measured within a one-hour time budget. Experiments were executed on a server with 20 3.7GHz cores and 250GB memory, operating in single-threaded mode.

Results: DCV is highly efficient. Table 2 shows the evaluation results. DCV verifies all contract properties within 10 seconds, and the majority finish in one second.

On the contrary, solc successfully verifies only 12 reference contracts, with one taking an extended 321 seconds. It times out on six contracts and encounters SMT solver invocation errors on two, a known issue documented on the GitHub repository issue tracker [4], sensitive to operating system and Z3 library versions.

Solc-verify verifies 12 reference contracts and reports unknown on 12. It encounters errors on six contracts due to challenges in analyzing certain parts of the OpenZepplin libraries.

VeriSmart verifies six contracts, times out on 12, runs out of memory on one, and reports unknown for the rest of the contracts.

For Solidity contracts generated from DeCon, solc verifies five, solc-verify verifies 17, and VeriSmart verifies 11. The performance difference between the reference version and the DeCon-generated version is potentially caused by the fact that DeCon generates standalone contracts that implement all functionalities without external libraries. On the other hand, DeCon implements contract states (relations) as mappings from primary keys to tuples, which may incur extra analysis complexity compared to the reference version.

The effects of inductive invariant inference. As indicated by the "inv.?" column in Table 2, among the 30 properties we tested, 10 were not inductive and required inductive invariant inference. DCV is able to discover inductive invariants that prove the target properties for all contracts.

In summary, DCV is highly efficient in verifying contract-level safety invariants, and can handle a wider range of smart contracts compared to other tools. Baseline tools face challenges due to their overly precise modeling of contract implementation and their inability to discover inductive invariants. By taking advantage of the high-level abstractions of the DeCon language, DCV achieves significant speedup over the baseline tools.

Qualitative comparison with bounded model checking (BMC) tools. BMC tools unroll loops up to a certain times and then verify safety within that bound. For example, ESBMC-Solidity [42] verifies individual transactions and can find vulnerabilities like integer overflow in specific functions. However, it cannot verify invariants across multiple transaction executions. EthBMC [23] also uses BMC to verify smart contracts, but it only focuses on a specific set of vulnerabilities. Unlike these BMC tools, DCV can prove any property that a user specifies, for infinite transaction traces.

Threats to validity. The validity of our findings is subject to certain threats: (1) benchmark selection: while DeCon is flexible enough to specify most contracts in the financial domains, it does not support some features found in other domains, such as cryptographic algorithms, or low-level constructs like checking interfaces of another contract. A few contracts requiring these features are excluded from the benchmarks. However, these features can usually be abstracted away in separate user-defined functions that are verified – an avenue of future work; (2) property types: DCV only supports safety invariants, which should hold across an infinite sequence of transactions. Other forms of properties, e.g., arithmetic safety, and constraints on states before and after a transaction, is out of the scope of DCV, and thus is not included in the benchmark properties; and (3) DeCon-Solidity compiler: the correctness of the DeCon-Solidity compiler has not been formally verified. Potential discrepancies between the DeCon specification and the generated Solidity code could affect the evaluation results. Formally verifying DeCon compiler is an important avenue for future work.

7 RELATED WORK

Smart contract verification. The challenge of verifying smart contracts has been extensively addressed in the literature [26, 28, 32, 36, 38, 41, 46]. Several methods, including VeriSmart [41], SmartACE [47], and VetSC [22], focus on safety verification, with some capable of generating counterexamples as sequences of transactions to disprove safety properties.

DCV distinguishes itself by employing a high-level executable specification language, DeCon, as the verification target. While this choice improves verification efficiency, DCV is limited to contracts written in DeCon, in contrast to other tools that operate on existing contracts in Solidity or Move.

Formal semantics of smart contracts. KEVM [20] and ACT [11] introduces formal semantics for smart contracts, and can automatically verify that a Solidity program (the compiled EVM bytecode) implements its formal specification. ACT can also prove contract invariants, but it relies on users to provide inductive invariants.

Formal semantics of EVM bytecode have also been formalized in F* [25] and Isabelle/HOL [16]. Scilla [39] is a type-safe intermediate language for smart contracts that also provides formal semantics. They offer precise models of the smart contract behaviors, and support deductive verification via proof assistants. However, working with a proof assistant requires non-trivial manual effort. On the contrary, DCV provides fully automatic verification.

Vulnerability detection. Securify [44] encodes smart contract semantic information into relational facts, and uses Datalog solver to search for property compliance and violation patterns in these facts. Oyente [31] uses symbolic execution to check generic security vulnerabilities, including reentrancy attack, transaction order dependency, etc. Maian [34] detects vulnerabilities by analyzing transaction traces. Unlike the sound verification tools, which require some amount of formal specification from the users, these work require no formal specification and can be directly applied to any existing smart contracts without modification, offering a quick and light-weight alternative to sound verification, although may suffer from false positives or negatives.

Fuzzing and testing. Fuzzing and testing techniques [14, 21, 24, 27, 40] can complement deductive verification tools in several ways. Firstly, they operate in a black-box mode, suitable for contracts without source code access. Secondly, while they may not guarantee the absence of bugs, they offer better scalability, which can be an advantage for complex properties that formal verification tools struggle with. Lastly, they provide concrete counter-examples,

which offers valuable insights for debugging. By combining deductive verification with fuzzing and testing, developers can validate smart contracts in a more comprehensive and robust way.

Run-time verification. Run-time verification has also been extensively explored in literature [18, 19, 30, 37]. DeCon [18], for instance, provides a run-time verifier with provenance support for visualizing counter-examples. Solythesis [30] proposes novel algorithm to minimize run-time monitoring overhead. While run-time verification is generally more scalable than static verification, it incurs run-time overhead and increases transaction fees. Moreover, the difficulties of updating smart contracts after their deployment undermines the importance of validation during development.

8 CONCLUSION

We present DCV, an automatic safety verification tool for declarative smart contracts written in the DeCon language. It leverages the high-level abstraction of DeCon to generate succinct models of the smart contracts, performs sound verification via mathematical induction, and applies domain-specific adaptations of the Houdini algorithm to infer inductive invariants. Evaluation shows that it is highly efficient, verifying all 23 benchmark smart contracts, with significant speedup over the baseline tools.

Our experience with DCV has also inspired interesting directions for future research. First, although DCV can verify a wide range of contracts in the financial domain, we find certain interesting applications that require non-trivial extensions to the modeling language, including contract inheritance, interaction between contracts, and functions that lie outside relational logic. Second, we aim to explore verification beyond safety invariants. Properties expressed in temporal logic [36, 43] and high-level semantics [22] represent promising areas. Last, the current approach involves rewriting smart contracts in DeCon. An intriguing direction is to infer the DeCon counterpart of an existing smart contract, thus potentially reducing the manual effort required in the verification process. By addressing these challenges and extensions, we aim to push the boundaries of smart contract verification and enhance the applicability of declarative programming languages in this domain.

9 ACKNOWLEDGEMENT

We thank anonymous reviewers for their insightful feedback. This work is supported by the ShanghaiTech Startup Fund, NSERC Discovery Grant, and NSF Grant CNS-2104882 and CNS-2107147.

REFERENCES

[1] King of the ether throne — post-mortem investigation. https://www.kingoftheether.com/postmortem.html, 2016.
[2] Understanding the dao attack. https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack/, 2016.
[3] Not a fair game – fairness analysis of dice2win. https://blogs.360.net/post/Fairness_Analysis_of_Dice2win_EN.html, 2018.
[4] Cannot replicate smtchecker example output. https://github.com/ethereum/solidity/issues/13073, 2022.
[5] Openzeppelin. https://github.com/OpenZeppelin/openzeppelin-contracts, 2022.
[6] Smtchecker and formal verification. https://docs.soliditylang.org/en/v0.8.17/smtchecker.html, 2022.
[7] Solidity by example. https://docs.soliditylang.org/en/v0.8.17/solidity-by-example.html, 2022.
[8] The solidity contract-oriented programming language. https://github.com/ethereum/solidity, 2022.
[9] Verx smart contract verification benchmarks. https://github.com/eth-sri/verx-benchmarks, 2022.
[10] Z3. https://github.com/Z3Prover/z3, 2022.
[11] Act. https://github.com/ethereum/act, 2023.
[12] Erc-20 top tokens. https://etherscan.io/tokens, 2023.
[13] Non-fungible tokens (nft). https://etherscan.io/tokens-nft, 2023.
[14] Scribble. https://github.com/ConsenSys/Scribble, 2023.
[15] Verismart commit on may 31, 2020. https://github.com/kupl/VeriSmart-public/commit/858af814fbdab0c9d85b758e4c4575402ebf2bdf, 2023.
[16] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 66–77, 2018.
[17] Franck Cassez, Joanne Fuller, and Horacio Mijail Antón Quiles. Deductive verification of smart contracts with dafny. In *International Conference on Formal Methods for Industrial Critical Systems*, pages 50–66. Springer, 2022.
[18] Haoxian Chen, Gerald Whitters, Mohammad Javad Amiri, Yuepeng Wang, and Boon Thau Loo. Declarative smart contracts. In *ESEC/FSE '22*, 2022.
[19] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, et al. Soda: A generic online detection framework for smart contracts. In *NDSS*, 2020.
[20] Xiaohong Chen, Daejun Park, and Grigore Roşu. A language-independent approach to smart contract verification. In *International Symposium on Leveraging Applications of Formal Methods*, pages 405–413. Springer, 2018.
[21] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 227–239. IEEE, 2021.
[22] Yue Duan, Xin Zhao, Yu Pan, Shucheng Li, Minghao Li, Fengyuan Xu, and Mu Zhang. Towards automated safety vetting of smart contracts in decentralized applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 921–935, 2022.
[23] Joel Frank, Cornelius Aschermann, and Thorsten Holz. ETHBMC: A bounded model checker for smart contracts. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2757–2774. USENIX Association, August 2020.
[24] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 557–560, 2020.
[25] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*, pages 243–269. Springer, 2018.
[26] Ákos Hajdu and Dejan Jovanović. solc-verify: A modular verifier for solidity smart contracts. In *Working conference on verified software: theories, tools, and experiments*, pages 161–179. Springer, 2019.
[27] Bo Jiang, Ye Liu, and Wing Kwong Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 259–269. IEEE, 2018.
[28] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: analyzing safety of smart contracts. In *Ndss*, pages 1–12, 2018.
[29] Shuvendu K Lahiri and Shaz Qadeer. Complexity and algorithms for monomial and clausal predicate abstraction. In *International Conference on Automated Deduction*, pages 214–229. Springer, 2009.
[30] Ao Li, Jemin Andrew Choi, and Fan Long. Securing smart contract with runtime validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 438–453, 2020.
[31] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269, 2016.
[32] Matteo Marescotti, Rodrigo Otoni, Leonardo Alt, Patrick Eugster, Antti EJ Hyvärinen, and Natasha Sharygina. Accurate smart contract verification through direct modelling. In *International Symposium on Leveraging Applications of Formal Methods*, pages 178–194. Springer, 2020.
[33] Zeinab Nehai, Pierre-Yves Piriou, and Frederic Daumas. Model-checking of smart contracts. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 980–987. IEEE, 2018.
[34] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 653–663, New York, NY, USA, 2018. Association for Computing Machinery.
[35] Benedikt Notheisen, Magnus Gödde, and Christof Weinhardt. Trading stocks on blocks-engineering decentralized markets. In *International Conference on Design Science Research in Information System and Technology*, pages 474–478. Springer, 2017.
[36] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*, pages 1661–1677. IEEE, 2020.

[37] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934*, 2018.

[38] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. ethor: Practical and provably sound static analysis of ethereum smart contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 621–640, 2020.

[39] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with scilla. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

[40] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. {SmarTest}: Effectively hunting vulnerable transaction sequences in smart contracts through language {Model-Guided} symbolic execution. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1361–1378, 2021.

[41] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. Verismart: A highly precise safety verifier for ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1678–1694. IEEE, 2020.

[42] Kunjian Song, Nedas Matulevicius, Eddie B de Lima Filho, and Lucas C Cordeiro. Esbmc-solidity: an smt-based model checker for solidity smart contracts. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 65–69, 2022.

[43] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. Smartpulse: automated checking of temporal properties in smart contracts. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 555–571. IEEE, 2021.

[44] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, 2018.

[45] Shuai Wang, Yong Yuan, Xiao Wang, Juanjuan Li, Rui Qin, and Fei-Yue Wang. An overview of smart contract: architecture, applications, and future trends. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 108–113. IEEE, 2018.

[46] Yuepeng Wang, Shuvendu K Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. Formal verification of workflow policies for smart contracts in azure blockchain. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 87–106. Springer, 2019.

[47] Scott Wesley, Maria Christakis, Jorge A Navas, Richard Trefler, Valentin Wüstholz, and Arie Gurfinkel. Verifying solidity smart contracts via communication abstraction in smartace. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 425–449. Springer, 2022.