Automated Invariant Generation for Solidity Smart Contracts

Ye Liu, Chengxuan Zhang, and Yi Li, *Member, IEEE*

***Abstract*—Smart contracts are computer programs running on blockchains to automate the transaction execution between users. The absence of contract specifications poses a real challenge to the correctness verification of smart contracts. Program invariants are properties that are always preserved throughout the execution, which characterize an important aspect of the program behaviors. In this paper, we propose a novel invariant generation framework, InvCon+, for Solidity smart contracts. InvCon+extends the existing invariant detector, InvCon, to automatically produce *verified* contract invariants based on both dynamic inference and static verification. Unlike InvCon+, InvCon only produces *likely* invariants, which have a high probability to hold, yet are still not verified against the contract code. Particularly, InvCon+is able to infer more expressive invariants that capture richer semantic relations of contract code. We evaluate InvCon+on 361ERC20 and 10ERC721 real-world contracts, as well as common ERC20 vulnerability benchmarks. The experimental results indicate that InvCon+efficiently produces high-quality invariant specifications, achieving a recall of 80%, which can be used to secure smart contracts from 17 types of common vulnerabilities.**

***Index Terms*—Invariant detection, smart contract.**

I. Introduction

SMART contracts are computer programs that operate on blockchain networks. They are used to facilitate the management of substantial financial assets and the automated execution of agreements among multiple parties who lack inherent trust. Notably, blockchain networks such as Ethereum [1] and BSC [2] are widely recognized as leading platforms supporting smart contracts, with applications spanning diverse domains such as supply-chain management, finance, energy, games, and digital artworks. However, the immutable and autonomous nature of smart contracts makes them particularly vulnerable to programming errors and malicious exploitation. Once deployed, a smart contract cannot be modified, and therefore the costs for mitigation are much higher. While smart contracts hold promise for facilitating value transfer among users, those that deviate from their specifications may harbor bugs or vulnerabilities. Numerous implementations of ERC20 contracts diverge from common expectations, as exemplified by standard non-compliance of ERC20 [3], particularly concerning event emission, balance updates, and transaction fee mechanisms. Numerous real-world security incidents highlight these risks. The *BatchOverflow* [4] vulnerability allowed attackers to create an arbitrarily large number of tokens due to an unchecked arithmetic overflow, prompting major exchanges to halt ERC-20 transactions. In other cases, such as the *Bancor* [5], failure to return expected values in the transfer or approve functions caused compatibility issues with third-party applications, resulting in stuck funds.

Even well-established standard ERC20 implementations exhibit inconsistencies [6]. The root cause lies in the limited semantic specifications outlined in the ERC20 standard proposal document [7]. Take the `transfer` function as an illustration—it is designed to move a specified amount of tokens from the sender to the recipient while triggering the `Transfer` event and should throw an error if the sender lacks adequate tokens for the transfer. Nevertheless, the ERC20 proposal provides only simple textual descriptions of the function, leading to semantic disparities across various ERC implementations and even different versions of the same implementation. For instance, the widely used ERC20implementation from OpenZeppelininitially did not permit a return value for the `transfer` function until a later commit,[1] causing incompatibility issues with renowned tokens like BNB, as reported by the reputable security company SECBIT [6]. In cases where a contract necessitates checking the return value of an external call to a `transfer` function of ERC20 contracts, even if the `transfer` is successful, it may revert due to the absence of a return value, resulting in compatibility problems [8]. However, removing the return value check exposes contracts to a potential vulnerability known as the *fake deposit attack* [9].

Ensuring the correctness of smart contracts poses a significant challenge, especially in the absence of contract specifications. On the one hand, the documentation for most smart contracts is scant, with even widely recognized smart contract libraries like OpenZeppelin [10], [11] found to have errors and deficiencies in their documentation [12]. On the other hand, the absence of contract specifications hampers the widespread adoption of formal verification tools in the realm of smart contracts. To address this issue, the commercial formal verification company Certora[2] has adopted a crowd sourcing approach—they hosted numerous competitions on well-known bug bounty platforms,

Received 31 December 2023; revised 1 June 2025; accepted 1 July 2025. Date of publication 4 August 2025; date of current version 4 November 2025. This work was supported in part by the Singapore Ministry of Education Academic Research Fund Tier 2 under Grant T2EP20224-0003 and in part by the Nanyang Technological University Centre for Computational Technologies in Finance (NTU-CCTF). *(Corresponding author: Yi Li.)*

The authors are with the Nanyang Technological University, Singapore 637616 (e-mail: ye.liu@ntu.edu.sg; chengxua001@ntu.edu.sg; yi_li@ntu.edu.sg).

Digital Object Identifier 10.1109/TDSC.2025.3592705

[1]https://github.com/OpenZeppelin/openzeppelin-solidity/commit/6331dd125d8e8429480b2630f49781f3e1ed49~cd

[2]https://www.certora.com/

such as Code4Rena,[3] to engage third-party security experts in the formulation of contract specifications. Yet, manual creation of formal specifications for smart contracts remains costly and error-prone.

Many automated techniques [13], [14] have been proposed to generate formal specifications in various forms to support the testing, verification, and validation of software programs. Among them, program invariants, which are enduring properties maintained throughout program execution, inherently serve as excellent candidates for enhancing and reinforcing program specifications. Program invariants have been used for vulnerability detection [15], conformance checking [3], runtime protection [16], type checking [17], and formal verification [18], [19] for smart contracts. Established tools, such as Daikon [13], can identify *likely* program invariants for Java programs through the execution of their test cases. The process involves statistically inferring the invariants that hold based on predefined templates, while discarding those refuted by the data trace records. The complete historical transaction data of smart contracts is consistently stored on blockchains, encapsulating all execution data since contract deployment, serving as a valuable data source for mining invariants.

In our prior work, InvCon [20] utilized Daikon to identify *likely* invariants for smart contracts, all of which are primitive predicates hold throughout the existing transaction histories. Moreover, Liu et al. [21] employed reinforcement learning to learn contract invariants critical to safely performing arithmetic operations, with a focus on preventing integer overflow and underflow. Despite their usefulness, the correctness of such inferred invariants remains unverified. In particular, an invariant which holds in past transactions may not always hold in the future—this may be due to the limited contract interactions observed in the transaction histories so far.

In this paper, we expand upon InvConto generate *verified* contract invariants utilizing both dynamic inference and static verification. We introduce a specialized invariant specification language tailored for Solidity smart contracts and propose a novel approach for inferring high-quality verified invariants. Specifically, we design a Houdini-like [14] algorithm to generate verified invariants for smart contracts. To address the explosion problem in searching for richer invariant candidates, such as implications that prevail in ERC20 and ERC721 [22], [23], [24] specifications, we introduce an iterative and incremental process for exploring these candidates on demand. We also apply control- and data-flow analyses to eliminate meaningless candidates and further improve the invariant generation efficiency. Our approach is implemented as an automated tool called InvCon+. Through evaluation on 361ERC20 contracts and 10ERC721 real-world Solidity contracts, we demonstrate that InvCon+produces comprehensive contract invariant specifications with no false positives. Furthermore, our analysis of real-world vulnerable ERC20 contracts underscores the potential of InvCon+in safeguarding these contracts through the application of mined invariant specifications.

In summary, we make the following contributions:

[3]https://code4rena.com/

$$
\begin{aligned}
a, v \in Variable ::= {} & address \mid uint \mid int \mid string \mid bytes \mid \\
& byte \mid bool \mid array \mid mapping \mid struct\{\vec{v}\} \\
f \in Function ::= {} & func(\vec{a})\ \{\vec{s}\} \\
s \in Statement ::= {} & v \mid v := e \mid \textbf{if}\ (e)\ \{\vec{s}\}\ \textbf{else}\ \{\vec{s}\} \mid \\
& \textbf{call}(\vec{e}) \mid \textbf{return}\ e \mid \\
& \textbf{require}(e) \mid \textbf{assert}(e) \mid \textbf{revert} \\
e \in Expr ::= {} & v \mid const \mid e[e] \mid e.v \mid e \bowtie e
\end{aligned}
$$

Fig. 1. The core grammar of the Solidity language.

- We introduce a comprehensive invariant specification language designed for expressing operational semantics in Solidity smart contracts. This language enables logical operations on variables of primitive types and commonly used data structures like structs, arrays, and mappings in Solidity.
- We present a unified framework for generating *verified* invariants in Solidity smart contracts, combining dynamic invariant detection and static invariant verification. Specifically, we develop a custom algorithm inspired by the Houdini algorithm to verify invariants for smart contracts and introduce an iterative process to derive a richer class of invariants.
- Our proposed approach is implemented in InvCon+, and its effectiveness is evaluated on 361ERC20 contracts and 10ERC721 contracts, along with vulnerable ERC20 contracts involving 24 types of vulnerabilities. The results demonstrate that InvCon+can generate high-quality and comprehensive invariant specifications for smart contracts, achieving a recall of 0.80, which is able to prevent 17 types of common vulnerabilities. The dataset, raw results, and the prototype used in our experiments are available online at: https://sites.google.com/view/invconplus/.

Organization: The rest of the paper is organized as follows. Section II provides the background about smart contracts and invariant inference. Section III defines the invariant specification language. Then, Section IV introduces our invariant generation approach. Section V describes our implementation framework, InvCon+, and Section VI demonstrates our evaluation results. The related work is discussed in Section VII and we conclude the paper in Section VIII.

II. Background

A. Solidity Smart Contracts

Fig. 1 presents the foundational grammar of the Solidity language, with certain features, such as event emission, intentionally excluded for the sake of clarity. Solidity encompasses various primitive data types, including integer, string, and boolean. Distinguishing itself from other programming languages like Java, Solidity does not permit floating-point numbers and incorporates a distinctive address type. This design choice is rooted in the interaction pattern between contracts and blockchain users, each possessing a unique address. Moreover, the majority of contracts are developed with the primary goal of tokenizing digital assets.

A Solidity smart contract comprises a collection of state variables and a set of functions. Statements within each function can take the form of variable assignments, conditional statements, internal or external function calls, requirement or assertion statements, and reversion or return statements. Notably, the *require* and *assert* statements can be employed to enforce program invariants at runtime. In the realm of expressions, $\bowtie$ denotes a binary operator encompassing $\{+, -, *, /, >, <, \geq, \leq, =, \neq, \wedge, \vee\}$.

Smart Contract Execution: The execution of a smart contract function can be triggered by sending a blockchain transaction to the contract address. Typically, each transaction incorporates one or more contract calls, potentially leading to alterations in contract state variables unless the transaction undergoes a reversion. To ease the discussion in this paper, we model a smart contract SC as a tuple $(\vec{v}, \vec{f})$, where $\vec{v}$ is a vector of state variables and $\vec{f}$ is a list of public functions.

Definition 2.1 (Contract Execution): Let $Dom(v)$ be the domain of a variable v and $Dom(\vec{v}) = \prod_{v \in \vec{v}} Dom(v)$. Then, $\delta, \delta' \in Dom(\vec{v})$ represent two reachable contract states. For a function invocation $f(\vec{a})$, calling function f with parameters values $\vec{a}$, we define its high-level execution semantics as a state transition $\delta \xrightarrow{f(\vec{a})} \delta'$.

Note that since a contract execution is triggered by a transaction recorded into a specific block of the blockchain, the parameter values $\vec{a}$ also includes implicit transaction and block parameters, e.g., `msg.sender` and `block.number`.

Transaction Histories: The execution of a smart contract is intricately linked to its transaction histories on the blockchain. The transaction histories record every contract execution, capturing function calls, state transitions, and modification to state variables from the contract deployment onward. It encapsulates the evolution of the contract state, reflecting the cumulative effect of all transactions. This historical traceability is fundamental for auditing, debugging, and understanding the operational dynamics of smart contracts on the blockchain.

B. Invariant Inference

In this paper, we aim to mine *contract-level* and *function-level* invariant specifications.

Definition 2.2 (Function Pre/Post-conditions): Let f be a contract function, and predicates p and q be the pre/post-conditions of f, respectively, which can be represented as a Hoare triple $\{p\}f\{q\}$. Then the following condition should be satisfied.

$$\forall \delta, \forall \vec{a} \cdot \delta \models p \wedge \delta \xrightarrow{f(\vec{a})} \delta' \Rightarrow \delta' \models q \tag{1}$$

Definition 2.3 (Contract Invariant): Given a smart contract *SC*, its contract invariant I is a predicate that must hold for any contract function execution. More formally, we have $\forall f \in SC \cdot \{I\}f\{I\}$.

Invariant inference techniques can be broadly categorized as static and dynamic. Static invariant inference (e.g., Houdini [14]) identifies function pre/post-conditions and contract invariants that hold for any program execution. On the other hand, dynamic invariant inference (e.g., Daikon [13]) identifies *likely* invariants that hold for specific contract executions (e.g., executions of a test case).

Let Δ denotes a set of program executions $\{(\delta, f(\vec{a}), \delta')\}$, which bring the contract state from δ to δ'. The *likely* function pre/post-conditionsof f, i.e., $\{\hat{p}\}f\{\hat{q}\}$, hold for Δ if $\forall(\delta, f(\vec{a}), \delta') \in \Delta, \delta \models \hat{p} \wedge \delta \xrightarrow{f(\vec{a})} \delta' \Rightarrow \delta' \models \hat{q}$. The *likely* contract invariants of a smart contract is defined in a similar way, which is omitted here for brevity.

Algorithm 1: The Basic HOUDINI Algorithm.

Inputs : P, program under the analysis;
: $Candiates$, invariant annotation candidates.
Outputs : A, a set of verified invariant annotations.

```
1 A := Candidates
2 while true do
3 |   rft := Verify(P, A) ; //a set of refuted
  |    candidates failing the verification.
4 |   if rft = ∅ then
5 |   |   break
6 |   A ← A \ rft
7 end while
8 return A
```

C. Modular Verification

Modular verification [25] is a verification technique that analyzes each function or module of a program in isolation, using assume-guarantee reasoning. Instead of verifying the entire program as a whole, it checks whether each function satisfies its specification (e.g., preconditions and postconditions), assuming that other functions it calls also meet theirs. This decomposition improves scalability and enables reusability of verification results across different contexts, which is particularly valuable for large and complex programs. To improve the scalability of smart contract invariant analysis, our approach employs modular verification, a technique that analyzes individual smart contract functions in isolation, rather than directly verifying the entire smart contract.

D. Houdini Algorithm

Houdini algorithm [14] is a powerful technique in computer science used for program annotation. Algorithm 1 illustrates its basic workflow. The algorithm accepts invariant annotation candidates guessed by either static analysis that mines candidates from source code based on heuristics, or template-based dynamic analysis such as the well-known Daikon approach [13]. These annotations can be in the form of loop invariants and function pre/post-conditions, assisting in program comprehension, debugging, and correctness analysis. In essence, Houdini algorithm works by iteratively refining and filtering the annotations until there are no refuted candidates (Lines 2-6), yielded by a modular verification tool such as ESC/Java [26] and Boogie [27]. Houdini algorithm has found applications in various areas, including static checking [28], providing an efficient

$$const \in \textbf{Int}, \textbf{Bool}, \textbf{Addr}, \textbf{Str} \quad x \in \textbf{FreeVar} \quad v \in \textbf{Var}$$
$$e \in \textbf{Expr} ::= const \mid v \mid old(v) \mid len(v) \mid SumMap(v) \mid e.x \mid e[x] \mid e \bowtie e$$
$$p \in \textbf{Predicate} ::= \bot \mid e \mid e \implies e$$
$$Statement ::= \textbf{Requires}\ p \mid \textbf{Ensures}\ p \mid \textbf{ContractInv}\ p$$

Fig. 2. The invariant specification language.

and scalable approach to enhance program understanding and reliability. Recently, Houdini algorithm has also been applied to facilitate the formal verification [29] of smart contracts.

III. Invariant Specification Language

Fig. 2 introduces our invariant specification language designed for Solidity smart contracts. The language accommodates variables of four types: integer, Boolean, address, and string, encompassing all primitive Solidity types illustrated in Fig. 1. We facilitate two types of variables. The first, denoted as v, pertains to function input parameters or contract state variables maintained in the persistent storage of the blockchain. The second, denoted as x, is reserved for free variables exclusively utilized to index structure members or items within arrays and mappings. Each invariant predicate is expressed as either a primitive logical expression or an implication expression. Furthermore, valid specification statements encompass function-level precondition invariant predicates (**Requires**) and postcondition invariant predicates (**Ensures**), and contract-level invariant predicates (**ContractInv**).

The expressions within the language may take the form of constants, variables, structure members, array items, and binary expressions. The **old**$(\cdot)$ notation is employed to differentiate between the value of a variable before entering the function and its value upon exiting the function, while **len**$(\cdot)$ refers to the array length or mapping size. Additionally, the language incorporates the widely used **SumMap**$(\cdot)$ operator for computing the arithmetic sum over mapping items. The notation "$e \bowtie e$" represents arithmetic or logical binary operations, where the operator "$\bowtie$" corresponds to the set defined in Solidity as shown in Fig. 1.

Utilizing this invariant language, we can articulate a diverse range of function and contract invariants. To exemplify its application, we present a simple illustration. In Fig. 3, a basic ERC20contract is depicted, featuring three state variables—`totalSupply`, `balances`, `allows` (standing for allowances)—and a function, `transferFrom`. The purpose of the `transferFrom` function is to transfer a specified amount of tokens from the account addressed at `from` to another account at `to`. An extensively studied ERC20contract invariant of this example can be succinctly expressed as: "$SumMap(balances) = totalSupply$". This assertion signifies that the total sum of items within the mapping variable `balances` must be equal to the value of `totalSupply`. Additionally, the function pre/post-conditions can be articulated as follows.

```
1  contract ERC20 {
2   // state variables
3   uint totalSupply;
4   mapping(address=>uint) balances;
5   mapping(address=>mapping(address=>uint)) allows;
6   ...
7   function transferFrom(address from, address to,
    ↪  uint tokens) public returns (bool) {
8     if (to == address(0)){
9             return false;
10    }
11    allows[from][msg.sender] =
      ↪  allows[from][msg.sender].sub(tokens);
12    balances[from] = balances[from].sub(tokens);
13    balances[to] = balances[to].add(tokens);
14    return true;
15  }
16 }
```

Fig. 3. A simple ERC20 contract.

Requires $\bot$

① **Ensures** $to \neq 0 \Rightarrow allows[from][msg.sender] = old(allows[from][msg.sender]) - tokens$

② **Ensures** $to \neq 0 \wedge from \neq to \Rightarrow balance[from] = old(balance[from]) - tokens \ \wedge \ balance[to] = old(balance[to]) + tokens$

③ **Ensures** $to \neq 0 \wedge from = to \Rightarrow balance[from] = old(balance[from]) \wedge balance[to] = old(balance[to])$

In this instance, it is straightforward to ascertain that there are no preconditions for the `transferFrom` function, assuming that all function preconditions are primitive predicates. The function is characterized by three postconditions. The first postcondition ① specifies that `allows` will undergo an update (Line 11) when `to` is a non-zero address. Additionally, in cases where `from` and `to` represent distinct addresses, the second postcondition ② dictates that the balances should be adjusted accordingly (Lines 12–13). Conversely, when `from` and `to` are identical, the last postcondition ③ emphasizes that the net effect on balance changes should be nullified. A detailed exploration of how these invariants are mined will be provided in Section IV-E.

IV. Invariant Generation Approach

In this section, we present our algorithm for generating verified invariants in smart contracts and elaborate on the techniques employed to infer implication invariants. For simplicity in presentation, we use the term "invariants" to collectively denote both function pre/post-conditionsand contract invariants when explicit characterization is unnecessary.

A. Algorithm

Algorithm 2 outlines our approach to invariant generation. The algorithm takes a smart contract SC, a sequence of contract transactions T, and a set of invariant templates Q as input. The

output, denoted as $Invs$, comprises a set of *verified* invariants, encompassing both primitive and implication invariants.

In this algorithm, $Invs$ is initialized as an empty set (Line 1). Subsequently, we initialize a set C that encompasses all potential invariant candidates under the given input (Line 2), similar to Daikon's initialization process [13], which instantiates all the parameterized invariant templates with concrete contract state variables and function input variables. For example, "$X = Y$" is a binary equation template where X and Y are placeholders that can be filled by two concrete variables: v_x and v_y whenever $Dom(v_x) \equiv Dom(v_y)$. It is important to note that here C excludes implication invariant candidates due to the exponential complexity of traversing all implication candidates. Instead, implication invariants will be generated on demand. Moreover, the execution trace set Δ is initialized as an empty set (Line 3).

The algorithm processes the transaction histories to extract corresponding execution traces. For each transaction t_i, the algorithm parses it to extract the invoked function f and parameters values $\vec{a}$ (Line 5). Additionally, the old and present contract states (i.e., values of the contract state variables), denoted as δ and δ', respectively, are recorded. The tuple $(\delta, f(\vec{a}), \delta')$ is added to the execution trace set Δ (Line 6).

Next, the algorithm executes the dynamic invariant detection procedure INVDETECT (Line 8) to obtain two classes of invariant candidates:

- C_{likely}, likely invariant candidates that hold for the entire transaction histories.
- $C_{partial}$, partially supported invariant candidates that hold for a subset of transaction histories.

Subsequently, a primitive invariant inference technique, detailed in Section IV-B, is applied to infer the standing invariants out of C_{likely}, and all the verified invariants are included in $Invs$ (Line 9). The unverified likely invariant candidates, $C_{likely} \setminus Invs$, and $C_{partial}$ are used to derive implication candidates assigned to C_{imp} (Line 10) via FINDIMPLICATIONS, which will be detailed in Section IV-C. Additionally, it is important to note that the found implications may not always hold. An iterative process is in place to validate these implications (Line 12) or weaken these implications via WEAKENIMPLICATIONS (Line 13) to identify new ones. This iterative process continues until all valid candidates are examined (Line 11). Finally, the algorithm returns $Invs$, which includes all the correctly mined invariants from transaction histories (Line 15).

Algorithm 2: Contract Invariant Inference.

Inputs : $SC = \{\vec{v}, \vec{f}\}$, where each element $v_i \in \vec{v}$ is a contract state variable and each element $f_i \in \vec{f}$ is a public contract function;
: $T = \{t_i | 1 \leq i \leq n\}$, where each element t_i is a contract transaction;
: Q, a set of invariant templates.
Outputs : $Invs$, a set of verified invariants.

1 $Invs := \emptyset$;
2 $C :=$ INITIALIZECANDIDATES$(\vec{v}, \vec{f}, Q)$; `//primitive candidates`
3 $\Delta := \emptyset$; `//execution trace set`
4 **foreach** $t_i \in T$ **do**
5 $(\delta, f(\vec{a}), \delta') \leftarrow$ PARSE(t_i) ;
6 $\Delta \leftarrow \Delta \cup (\delta, f(\vec{a}), \delta')$;
7 **end foreach**
8 $C_{likely}, C_{partial} \leftarrow$INVDETECT ($\Delta$, C);
9 $Invs \leftarrow$ STATICINFER(C_{likely}) ;
10 $C_{imp} \leftarrow$ FINDIMPLICATIONS$(C_{likely} \setminus Invs, C_{partial})$; `//implication candidates`
11 **while** $C_{imp} \neq \emptyset$ **do**
12 $Invs \leftarrow Invs \cup$ STATICINFER(C_{imp}) ;
13 $C_{imp} \leftarrow$ WEAKENIMPLICATIONS$(C_{imp} \setminus Invs)$;
14 **end while**
15 **return** $Invs$

Algorithm 3: STATICINFER(Candidates).

1 Instrument SC to **enable** each candidate from Candidates;
2 **while** *true* **do**
3 result = MODULARVERIFY(SC) ;
4 **if** *result* = *CORRECT* **then**
5 **return** enabled candidates ; `//verified invariants`
6 **else if** *result* = *INCORRECT due to failed candidate c* **then**
7 **disable** c in SC;
8 **else**
9 **raise** Error ; `//INCORRECT due to failed assertion in SC`
10 **end if**
11 **end while**

B. Primitive Invariant Inference

Algorithm 3 illustrates our Houdini-like algorithm to infer verified primitive invariants from the candidates mined from contract transaction histories. First, we enable all the candidates in SC via contract instrumentation (Line 1); each candidate is explicitly labeled by the added keywords, e.g., **ContractInv** for contract invariant, **Requires** for function precondition, and **Ensures** for function postcondition. Next, we invoke a modular verifier to statically verify these enabled candidates (Line 3), i.e., verifying each function in isolation where all the corresponding candidates are examined against the function implementation. When there is a failed invariant candidate c violating the verification condition, c will be disabled in SC (Line 7). This process will continue until all the enabled candidates are verified successfully (Line 4) and then returned (Line 5). Particularly, whenever there is a failed assertion in SC, i.e., a violated condition e in the **assert**(e) statement, the algorithm terminates with an error raised (Line 9). This happens in Solidity contracts, because **assert**(e) is often misused to replace **require**(e) that enforces program requirements due to their similar effects on transaction reversion. For smart contracts without failed assertions, the verified invariants is a maximal subset of the candidates whose conjunction is an inductive invariant.

C. Implication Invariant Inference

Figs. 4 and 5 illustrate the two procedures for identifying implication candidates in the form $e \Rightarrow e$, respectively. The high-level idea is that useful implication invariants can be identified from the cascading combinations of historical invariants that satisfy data/control flow of smart contracts.

$$\frac{\bot}{C_{imp} := \{(\eta \implies \tau) \mid \eta, \tau \in C_{likely} \setminus Invs \cup C_{partial}, \eta \neq \tau\}} \text{ Init}$$

$$\frac{(\eta \implies \tau) \in C_{imp} \qquad \begin{array}{c}\forall a \in vars(\eta), \forall b \in vars(\tau). \\ \neg\, dep(a,b)\end{array}}{C_{imp} \leftarrow C_{imp} \setminus (\eta \implies \tau)} \text{ Delete}$$

Fig. 4. FINDIMPLICATIONS.

$$\frac{\bot}{\hat{C_{imp}} := \emptyset} \text{ Init}$$

$$\frac{(\eta_1 \implies \tau), (\eta_2 \implies \tau) \in C_{imp} \setminus Invs \qquad \eta_1 \wedge \eta_2 \not\equiv false}{\hat{C_{imp}} \leftarrow \hat{C_{imp}} \cup (\eta_1 \wedge \eta_2 \implies \tau)} \text{ Append-1}$$

$$\frac{(\eta \implies \tau_1), (\eta \implies \tau_2) \in C_{imp} \setminus Invs \qquad \tau_1 \vee \tau_2 \not\equiv true}{\hat{C_{imp}} \leftarrow \hat{C_{imp}} \cup (\eta \implies \tau_1 \vee \tau_2)} \text{ Append-2}$$

Fig. 5. WEAKENIMPLICATIONS.

In Fig. 4, FINDIMPLICATIONS employs two straightforward inference rules. The first rule enumerates all the combinations of historical invariants while the second rule filters those meaningless combination candidates. *Init* explores all the potential implication candidates from the unverified likely invariants $C_{likely} \setminus Invs$ and partial invariant candidates $C_{partial}$, including them in C_{imp}. An implication invariant takes the form of $\eta \Rightarrow \tau$, where η and τ comes from the existing the unverified and partial invariant candidates. However, not all of the implication candidates constructed this way are relevant in terms of the contract semantics. An implication is potentially useful (i.e., relevant) if its precondition and postcondition align with the data/control-flow of the contracts, and irrelevant/meaningless implications should be discarded. The notation $vars(p)$ represents variables appearing in an invariant predicate p; for instance, $vars(p) = \{from, to\}$ when p is "$from \neq to$". Additionally, $\mathbf{dep}(a, b)$ denotes whether variable a depends on variable b in terms of control-flow or data-flow in smart contract functions. To determine the valid implications, we leverage the well-known static analysis tool Slither [30] to trace data-flow and control-flow in smart contract functions. Therefore, in Fig. 4, a *Delete* rule is applied to eliminate implications that do not adhere to the data-flow and control-flow relationship. This rule is iteratively applied until no further implications can be eliminated.

Some implication candidates may be too strong and cannot be proved. Fig. 5 illustrates how we derive a weaker set of implication candidates $\hat{C_{imp}}$ from those unverified implication candidates denoted as $C_{imp} \setminus Invs$. In Fig. 5, WEAKENIMPLICATIONS comprises three inference rules. It initially sets $\hat{C_{imp}}$ to an empty set. The basic idea behind the remaining two rules is that implication can be weakened by strengthening its precondition or weakening its post-condition. The rules *Append-1* and *Append-2* generate weaker implications by combing two unverified implication candidates. In essence, $\eta_1 \wedge \eta_2 \Rightarrow \tau$ is weaker than either $\eta_1 \Rightarrow \tau$ or $\eta_2 \Rightarrow \tau$. Similarly, $\eta \Rightarrow \tau_1 \vee \tau_2$ is weaker than both $\eta \Rightarrow \tau_1$ and $\eta \Rightarrow \tau_2$. To eliminate useless implications that are tautologies, we impose restrictions on the original implications, such as $\eta_1 \wedge \eta_2 \not\equiv$ false and $\tau_1 \vee \tau_2 \not\equiv$ true. It is evident that the weaker implications are also meaningful as they still satisfy the same control/data-flow dependencies as the original ones. Then, these weakened implication invariants will be formally verified. Consequently, the proved weakened invariants are included in the set of verified invariants.

D. Termination

The termination of Algorithm 2 can be ensured by the fact that INVCON+can only produce a finite set of primitive invariant predicates. The conclusion regarding the termination of Algorithm 2 hinges on whether the loop (Lines 11-13) comes to an end. In each iteration of the loop, we possess at least one implication candidate, constructed by WEAKENIMPLICATIONS (refer to Section IV-C). Regarding WEAKENIMPLICATIONS, it consistently generates weaker implication candidates than the previous ones, utilizing conjunctions over premises or disjunctions over consequences. Assuming INVDETECT yields n primitive invariant predicates $C_{likely} \cup C_{partial} = \{p_1, \ldots, p_n\}$, then the weakest implication will be at least as strong as $p_1 \wedge \cdots \wedge p_n \Rightarrow p_1 \vee \cdots \vee p_n$. Consequently, the loop will finish in no more than $2 \times n$ iterations, establishing the termination of this algorithm.

E. Running Example

We illustrate our algorithm using the example presented in Fig. 3. The details regarding our transaction parsing and invariant detection will be elaborated in Section V. For the sake of simplicity in the illustration, assume that we have already acquired a set of likely and partially supported invariants through invariant detection on the transaction histories. In Table I, the invariants labeled with $\surd$are successfully verified by the static verifier, while the ones with ✗are unverified. In Step ①, we perform a Houdini-like static inference on these detected invariant candidates. Consequently, three likely invariants are verified, excluding $to \neq 0$. In the subsequent step (Step ②), nine additional implication invariant candidates are generated from the previously unverified likely invariants and partially supported invariants, according to the rules in FINDIMPLICATIONS (see Fig. 4). However, after the modular verification, only one implication is confirmed. Furthermore, we weaken these unverified implication invariants in Step ③ using WEAKENIMPLICATIONS (see Fig. 5) to derive four new implication candidates for further validation. Eventually, all the invariants listed in Section III are successfully recovered (in a logically equivalent form). Moreover, two other invariants, $balances[to] \geq old(balances[to])$ and $balances[from] \leq old(balances[from])$, are verified, which provide additional insights on how the balances of the sender and the receiver should change when `transferFrom` is called, beyond the standard specifications.

V. IMPLEMENTATION

A. Overview

Fig. 6 demonstrates the high-level architecture of INVCON+, our automated invariant detection tool for Solidity smart contracts. The inputs to INVCON+include a set of historical

TABLE I
ILLUSTRATION EXAMPLE OF INVARIANT VERIFICATION

Step	Invariants	
①	**Likely Contract Invariants**: totalSupply = SumMap(balances) ✓ **Likely Function Pre/post-conditions**: to $\neq$0 ✗ balances[to] $\geq$ old(balances[to]) ✓ balances[from] $\leq$ old(balances[from]) ✓	**Partially Supported Function Pre/post-conditions**: from $\neq$ to from = to balances[from] = old(balances[from]) - tokens balances[to] = old(balances[to]) + tokens allows[from][msg.sender] = old(allows[from][msg.sender]) - tokens balances[from] = old(balances[from]) balances[to] = old(balances[to])
②	**Implication Invariant Candidates:** to $\neq$0 $\Longrightarrow$ allows[from][msg.sender] = old(allows[from][msg.sender]) - tokens ✓ to $\neq$0 $\Longrightarrow$ balances[from] = old(balances[from]) - tokens ✗ to $\neq$0 $\Longrightarrow$ balances[to] = old(balances[to]) + tokens ✗ to $\neq$0 $\Longrightarrow$ balances[from] = old(balances[from] ✗ to $\neq$0 $\Longrightarrow$ balances[to] = old(balances[to]) ✗ from $\neq$to $\Longrightarrow$ balances[from] = old(balances[from]) - tokens ✗ from $\neq$to $\Longrightarrow$ balances[to] = old(balances[to]) + tokens ✗ from = to $\Longrightarrow$ balances[from] = old(balances[from] ✗ from = to $\Longrightarrow$ balances[to] = old(balances[to]) ✗	
③	**Weakened Implication Invariant Candidates:** to $\neq$0 $\wedge$ from $\neq$to $\Longrightarrow$ balances[from] = old(balances[from]) - tokens ✓ to $\neq$0 $\wedge$ from $\neq$to $\Longrightarrow$ balances[to] = old(balances[to]) + tokens ✓ to $\neq$0 $\wedge$ from = to $\Longrightarrow$ balances[from] = old(balances[from]) ✓ to $\neq$0 $\wedge$ from = to $\Longrightarrow$ balances[to] = old(balances[to]) ✓	

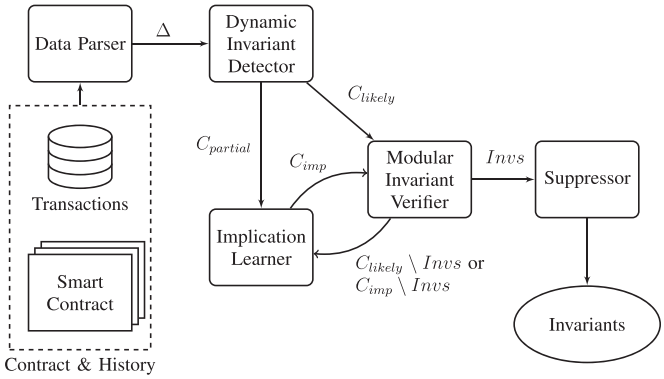

Fig. 6. The architecture overview of INVCON+.

transactions and the corresponding contract source code, while its output is a collection of smart contract invariant specifications or the accordingly annotated contract code. INVCON+comprises four modules: (1) a *data parser* that decodes contract code and transaction histories to extract concrete execution trace set; (2) a *dynamic invariant detector* that generates a set of likely and partially supported invariants; (3) a *modular invariant verifier* and an *implication learner* that verify and learn contract invariants, respectively; and (4) a *suppressor* that simplifies the results by removing redundant invariants. Notably, the implication learner has already been detailed in Section IV-C.

B. Data Parser of Smart Contract Transactions

Given a contract, we first collect all of its historical transactions. For each transaction, we decode the specific function input based on the contract's Application Binary Interface (ABI), and we interpret the transaction output in accordance with the contract's storage layout specifications. This layout dictates where each state variable is stored in the blockchain database. For instance, as shown in Fig. 1, the first declared state variable `totalSupply` is stored at the first slot (`0x0`) in the contract's blockchain database.

The input of a contract transaction is represented as a tuple *(sender, function, parameters)*, which encapsulates the transaction's sender, the invoked function's name, and the corresponding input parameters. Conversely, the transaction's output is denoted as *(status, storageChanges)*. Here, *status* signifies the transaction's success or failure, while *storageChanges* details the alterations in the contract's storage across various slots. By aligning storage slots with the contract's storage layout, one can effectively interpret these storage modifications as changes in the values of the contract's state variables. Employing the previously described preprocessing technique enables the extraction of a sequence of data triples (i.e., execution traces). These triples consist of the actual values of state variables and function input variables at the point of function entry, as well as the most recent values of state variables at the point of function exit. It is important to note that any misrecognition of variables can lead to incorrect invariant results. We have implemented measures to ensure the accuracy of variable recognition. For state variables of primitive types, we directly ascertain their values, as the storage layout for these variables remains constant during runtime. In the case of non-primitive, dynamic state variables, to reduce computational cost, we initially utilize the known variable values to hypothesize a correlation between the altered storage slots and the dynamic state variables. However, if this approach fails to produce an accurate mapping, it becomes necessary to replay the entire transaction. This replay process enables us to track the comprehensive execution information, including storage modifications, thus allowing for the accurate determination of the correct mapping.

C. Dynamic Invariant Detector

The effectiveness of dynamic invariant detection largely depends on the diversity and scale of the customized invariant templates used. In our methodology, these invariant templates are required to conform to the invariant specification language outlined in Fig. 2. However, it is both impossible and impractical to cover every conceivable invariant template. Our approach, akin to that of InvCon [20], limits the scope to unary, binary, and ternary invariant templates. Unlike InvCon, our templates are specifically designed for Solidity smart contracts, which are predominantly used for financial applications. These contracts often entail intricate scientific computations on scalar variables. Furthermore, Solidity features an array of complex data structures, such as *mapping* and *struct*. To effectively infer invariants related to these structures, we have incorporated several derivation templates, such as *MemberItem* and *MappingItem*, which facilitate access to elements within these data structures. Additionally, drawing inspiration from the significance of balance invariants as highlighted by Wang et al. [31], we have introduced a *SumMap* derivation template. This template is specifically designed to aggregate the values contained within a mapping variable.

Dynamic invariant detection employs a statistical methodology to generate *likely* primitive invariants with a certain degree of statistical confidence. Contrasting with the approach of InvCon, our method retains invariants that are refuted by certain transactions in the final results. This is because less stringent forms of these invariants, expressed as implications, may still hold true for certain contracts. Both the likely invariants and the falsified ones constitute a high-quality set of primitive predicates. Each of these predicates has been empirically verified through historical transaction data of smart contracts. In our evaluation setting, each valid primitive invariant must be supported by at least three historical transactions.

Note that InvCon+supports not only the ERC20 and ERC721 standards, but also other customized smart contracts from DApp developers [32]. Currently, InvCon+does not support proxy-like smart contracts since their functionality and data storage are dynamically-linked from external contracts, posing threats to the validity of our invariant analysis. We plan to address this in future works.

D. Modular Invariant Verifier

The Houdinialgorithm [14] is a widely recognized technique commonly used in program annotation and validation processes. Its primary objective is to automatically generate invariant annotations from a group of candidates. To adapt Houdinialgorithm for Solidity contracts, we initially instrument the contracts with the mined invariants. This entails converting the invariants into a compatible format and then embedding them into the contract. The annotations are strategically placed at the beginning of functions to align with their specific names and arguments. Subsequently, we transform the instrumented contracts into *Boogie* [33] programs, leveraging the existing formal verification tool VeriSol [29] for Solidity smart contracts. We have refined the Boogie translator in VeriSolto better accommodate contracts with inheritance and polymorphism features. For instance, the original translator lacked support for unnamed parent contract calls using the "super" keyword in Solidity, and it did not handle function overloading where a contract includes multiple functions with the same name. We have enhanced its translation rules to effectively translate these complex contracts into Boogie programs. Finally, we utilize Boogie's own Houdinimodular verifier to infer among the aforementioned invariant annotations, resulting in a set of verified invariants.

E. Suppressor

An invariant is deemed redundant if it can be derived from another invariant. The invariants verified by InvCon+may contain such redundancies. Instead of eliminating redundancies in the dynamically detected invariants (as what Daikon [13] does), we only remove redundancies from the invariants that are successfully verified. This design leaves more choices to the implication learner, when synthesizing implication invariants. Among the verified invariants, we utilize the Z3 solver [34] to determine if one invariant predicate can be deduced from another. Following this analysis, we retain only the strongest invariant predicates in our final results.

In the next section, we will evaluate the effectiveness and performance of our implementation in invariant inference.

VI. Evaluation

In this section, we evaluate InvCon+to answer the following research questions:

1) *RQ1:* How effectively does InvCon+generate invariants for smart contracts?
2) *RQ2:* How does the length of transaction histories used affect the performance of InvCon+?
3) *RQ3:* How effective are the invariants detected by InvCon+in preventing real-world security attacks?

A. Methodology

Benchmark: To answer RQ1 and RQ2, we collected real-world smart contracts implementing the most popular ERC20and ERC721 standards, which have been studied extensively in previous works [3], [6], [16], [35], [36]. The most important reason of choosing smart contracts implementing common standards is that their invariant specifications are better understood, making it easier to obtain the ground truth. First, we queried the public Ethereum ETL dataset hosted on the Google BigQuery platform [37] and then identified 13,116 contract addresses flagged as ERC20 deployed between 2021 and 2022. Next, we identified 2,689 ERC721 contract addresses deployed between 2020 and 2022. To facilitate our analysis, we kept only open-source contracts written in Solidity versions ranging between 0.5.0 and 0.5.17, which are currently supported by VeriSol. Finally, we obtained 361ERC20 contracts and 10ERC721 contracts for the experimental evaluation, where each contract has at least 50 historical transactions as of June 2023. Note that InvCon+is able to detect *likely* invariants for smart contracts of Solidity versions beyond 0.5. The resulting

TABLE II
COMMON ERC20 INVARIANTS

Categories	Preconditions	Postconditions
transfer(to, amount)	**[a1]** msg.sender $\neq$ 0 **[a2]** to $\neq$ address(0) **[a3]** amount $\geq$ 0 **[a4]** amount $\leq$ balances[msg.sender] **[a5]** balances[to] + amount $\leq$ MAXVALUE	**[b1]** to $\neq$ msg.sender $\implies$ balances[msg.sender] = old(balances[msg.sender]) - amount **[b2]** to $\neq$ msg.sender $\implies$ balances[to] = old(balances[to]) + amount **[b3]** to = msg.sender $\implies$ balances[to] = old(balances[to]) **[b4]** to = msg.sender $\implies$ balances[msg.sender] = old(balances[msg.sender]) **[b5]** totalSupply = old(totalSupply)
transferFrom (from, to, amount)	**[a6]** from $\neq$ address(0) **[a7]** to $\neq$ address(0) **[a8]** amount $\geq$ 0 **[a9]** amount $\leq$ balances[from] **[a10]** amt $\leq$ allowed[from][msg.sender] **[a11]** balances[to] + amount $\leq$ MAXVALUE	**[b6]** allowed[from][msg.sender] = old(allowed[from][msg.sender]) - amount **[b7]** from $\neq$ to $\implies$ balances[from] = old(balances[from]) - amount **[b8]** from $\neq$ to $\implies$ balances[to] = old(balances[to]) + amount **[b9]** from = to $\implies$ balances[from] = old(balances[from]) **[b10]** allowed[from][msg.sender] = old(allowed[from][msg.sender]) - amount **[b11]** totalSupply = old(totalSupply)
approve(spender, amount)	**[a12]** amount $\geq$ 0 **[a13]** spender $\neq$ address(0)	**[b12]** allowed[msg.sender][spender] = amount **[b13]** totalSupply = old(totalSupply)
increaseAllowance(spender, amount)	**[a14]** spender $\neq$ address(0) **[a15]** amount $\geq$ 0 **[a16]** allowed[msg.sender][spender] + amount $\leq$ MAXVALUE	**[b14]** allowed[msg.sender][spender] = old(allowed[msg.sender][spender]) + amount **[b15]** totalSupply = old(totalSupply)
decreaseAllowance(spender, amount)	**[a17]** spender $\neq$ address(0) **[a18]** amount $\geq$ 0 **[a19]** allowed[msg.sender][spender] $\geq$ amount	**[b16]** allowed[msg.sender][spender] = old(allowed[msg.sender][spender]) - amount **[b17]** totalSupply = old(totalSupply)
mint(account, amount)	**[a20]** account $\neq$ address(0) **[a21]** amount $\geq$ 0 **[a22]** balances[account] + amount $\leq$ MAXVALUE	**[b18]** balances[account] = old(balances[account]) + amount **[b19]** totalSupply = old(totalSupply) + amount
burn(from, amount)	**[a23]** from $\neq$ address(0) **[a24]** amount $\geq$ 0 **[a25]** balances[from] $\geq$ amount	**[b20]** balances[from] = old(balances[from]) - amount **[b21]** totalSupply = old(totalSupply) + amount
pause()	**[a26]** paused = false	**[b22]** paused = true
unpause()	**[a27]** paused = true	**[b23]** paused = false
Contract Invariant	**[c1]** totalSupply = SumMap(balances)	

TABLE III
COMMON ERC721 INVARIANTS

Categories	Preconditions	Postconditions
(safe)-transferFrom(from, to, tokenId)	**[a28]** from = _tokenOwner[tokenId] **[a29]** from $\neq$ address(0) **[a30]** to $\neq$ address(0) **[a31]** (msg.sender = from $\vee$ msg.sender = _tokenApprovals[tokenId] $\vee$ _operatorApprovals[from][msg.sender] = true)	**[b24]** from $\neq$ to $\implies$ _ownedTokensCount[from] = old(_ownedTokensCount[from]) - 1 **[b25]** from $\neq$ to $\implies$ _ownedTokensCount[to] = old(_ownedTokensCount[to]) + 1 **[b26]** from = to $\implies$ _ownedTokensCount[from] = old(_ownedTokensCount[from]) **[b27]** from = to $\implies$ _ownedTokensCount[to] = old(_ownedTokensCount[to]) **[b28]** _tokenOwner[tokenId] = to **[b29]** _tokenApprovals[tokenId] = address(0)
approve(to, tokenId)	**[a32]** _tokenOwner[tokenId] $\neq$ address(0) **[a33]** (msg.sender = _tokenOwner[tokenId] $\vee$ _operatorApprovals[_tokenOwner[tokenId]][msg.sender] = true)	**[b30]** _tokenApprovals[tokenId] = to
setApproveForAll(operator, _approved)	**[a34]** operator $\neq$ msg.sender	**[b31]** _operatorApprovals[msg.sender][operator] = _approved
Contract Invariant	**[c2]** len(_tokenOwner) = SumMap(_ownerTokenCount)	

likely invariants could be verified using other state-of-the-art verifiers such as Certora [38], which we leave as our future work.

To establish the ground truth for ERC20 and ERC721 contract specifications and ensure the included invariants are comprehensive, we investigated multiple external sources. These include the formal specifications referenced in the existing literature [6], [16], popular smart contract libraries, such as OpenZeppelin [10], and online documentations provided by smart contract formal verification companies. We list the collected ERC20 and ERC721 invariant specifications in Tables II and III, respectively. These specifications are mainly based on Certora [22], [23], [39], KEVM [24], [35], and OpenZeppelin API documentations [40], [41]. We analyzed each of the collected invariants and manually translated it into our own specification language (C.f. Fig. 2), which is a straightforward exercise in most cases. We categorized these invariant specification into contract invariants, function preconditions and postconditions in Tables II and III. The functions listed in each table are the most commonly used standard functions for ERC20and ERC721 contracts. Some specifications documented in external sources were omitted, e.g., "Emitting a `Transfer` event" for the `transfer` function, because the particular language features are not supported by our specification language.

Evaluation Metrics: We use two evaluation metrics to evaluate INVCON+on the ERC20 and ERC721 contracts. Particularly, we use **Precision** and $\mathbf{Recall_{ERC20}}$ ($\mathbf{Recall_{ERC721}}$) to measure

the effectiveness of the generated invariants, denoted as X_{proved}. We denote the ground truth invariants (e.g., ERC20) as Y. Formally,

$$\textbf{Precision} = \frac{|X_{proved}|}{|X|}, \tag{2}$$

$$\textbf{Recall}_{\textbf{ERC20}}(\textbf{Recall}_{\textbf{ERC721}}) = \frac{|X_{proved} \cap Y|}{|Y|}, \tag{3}$$

where *precision* refers to the proportion of the generated invariants which are correct and *recall* is the proportion of the ground truth invariants which can be successfully generated. Since the contract execution trace set Δ from transaction histories may only contain a subset of functions invocations, i.e., some functions are never invoked. For a fair comparison, let $Y \lfloor \Delta$ represent the ground truth invariants for the functions appeared in the histories, and we use the adjusted recall in our experiments: $\frac{|X_{proved} \cap Y|}{|Y \lfloor \Delta|}$.

Note that although the ground truth invariants are derived based on multiple external sources and widely deemed to be standard, they may still be incomplete, as there are infinitely many correct invariants in theory. The purpose of collecting the ground truth invariants is to include the list of common expectations that are needed for contract safety and reliability. On the other hand, certain smart contracts may not faithfully implement the ERC standards, and as a result, either some ground-truth invariants may not hold for them or they satisfy additional invariants not included in the ground truth. Nevertheless, an ideal invariant generation tool should be able to recover as many ground-truth invariants as possible, and meanwhile, recover other relevant invariants that are correct and useful in describing unique smart contract behaviors.

B. Experiment Setup

All the experiments were conducted on a desktop computer with the Ubuntu 20.04 OS, an Intel Core Xeon 3.50GHx processor, and 32 GB of RAM. To facilitate the evaluation, we have crawled and cached all transaction histories in advance for the contracts used in our experiments.

C. RQ1: Effectiveness of Invariant Generation

Baseline: To evaluate the performance of InvCon+, we used InvCon as our baseline. InvCon uses Daikon as the back-end invariant detection engine and more implementation details can be found in the previous work [20]. To the best of our knowledge, Cider [21] is the only automated invariant generation tool for smart contracts besides InvCon. We have contacted the authors of Cider recently and the authors indicated that the repository is no longer available. We will discuss and compare with this work later.

Additionally, we compared InvCon+with its two variants: InvCon+*Naive*, which performs only dynamic invariant detection tailored to Solidity contracts, and InvCon+*Primitive*, which employs the Houdinialgorithm to generate verified invariants based only on dynamically detected invariant candidates.

TABLE IV
THE COMPARISON RESULTS ON ERC20 CONTRACTS

Tool	#Inv	Prec.	$\textbf{Rec}_{\textbf{ERC20}}$	Avg.time (s)
InvCon	413.23	0.095	0.19	13.99
InvCon+ *Naive*	480.89	0.094	0.63	15.25
InvCon+ *Primitive*	22.49	1.000	0.61	20.57
InvCon+	46.12	1.000	0.80	250.25

Results: Table IV presents the comparison results for 361ERC20 contracts, with a constraint of utilizing a maximum of 200 transactions per contract. The first column displays the names of the tools, while the second column enumerates the averaged number of invariants generated by each respective tool per contract. The middle two columns showcase the overall $\textbf{Precision}$ and $\textbf{Recall}_{\textbf{ERC20}}$ scores, and the last column provides the averaged time usage for each tool.

InvCon+achieves the highest recall score, reaching 0.80, and generates approximately 46 invariants per contract, all of which are successfully verified by VeriSol. Notably, InvConperforms the least favorably in terms of the invariants generated, even when compared with InvCon+*Naive*. Specifically, InvConproduces the second-highest number of invariants, yet its recall score is significantly lower than that of InvCon+*Naive*, while maintaining a similar precision score of less than 0.1. The poor performance is primarily attributed to the fact that InvCon's underlying invariant detection engine, Daikon, supports only Boolean, integer/float, and string types native to Java. Consequently, the *address* type (20 bytes long) in the Solidity language cannot be seamlessly converted into a Java integer (8 bytes long). Its conversion to the Java string type discards semantic information, rendering the straightforward production of common invariants (e.g., **a1**, **a2** in Table II) unattainable for InvCon. Additionally, Daikon employs floating-point operations in arithmetic invariant templates (e.g., linear equation templates), which is not allowed in the Solidity semantics, leading to incorrect invariants for **b6**, **b10** in Table II.

InvCon+ *Primitive*exhibits a slightly lower recall score than InvCon+*Naive*, because some ground truth invariants that are inferred as likely invariants by InvCon+*Naive*may not be verified by InvCon+*Primitive*. This may be due to contract implementations slightly deviating from the standard. For example, many ERC20 tokens do not enforce the precondition **a2** of the transfer function in Table II, because transferring token to zero address could be used to implement the token burning functionality. The verified invariants are a more accurate reflection of the actual contract implementations, compared with the likely invariants. Leveraging an algorithm capable of producing implications that widely exist in ERC20 invariants (C.f. Table II), InvCon+outperforms all the baseline tools, yielding 100% precise invariant results.

Regarding the time usage, it is unsurprising that InvCon+ takes the most time, whereas InvConand InvCon+*Naive*finish the fastest. In our experiments, we observed that the static inference process consumes the majority of the time, constituting nearly 53% of the overall time usage, as depicted in Fig. 9. This is primarily due to the iterative application of static inference until

```
1  contract TokenMintERC20Token is ERC20 {
2  address Account;
3  uint256 _totalSupply;
4  mapping(address => uint256) _balances;
5
6  // other functions omitted
7  /** @dev Creates `amount` tokens and assigns them to
   ↪  `account`,
8  *   increasing the total supply.
9  * Requirements
10 * - `to` cannot be the zero address.*/
11 function _mint(address account, uint256 amount)
   ↪  internal {
12  require(account != address(0), "ERC20: mint to the
    ↪  zero address");
13  _totalSupply = _totalSupply.add(amount);
14  _balances[account] = _balances[account].add(amount);
15  _balances[Account] = _totalSupply/100;
16 }
17 }
```

Fig. 7. `TokenMintERC20Token` contract violating **c1**.

```
1  function _transfer(address sender, address recipient,
   ↪  uint256 amount) internal {
2  require(sender!=address(0), "zero address");
3  require(recipient!=address(0), "zero address");
4
5  _balances[sender]=_balances[sender].sub(amount);
6  _balances[recipient]=_balances[recipient].add(amount);
7  emit Transfer(sender, recipient, amount);
8  }
9  function _approve(address owner, address spender,
   ↪  uint256 value) internal {
10 require(owner!=address(0), "zero address");
11 require(spender!=address(0), "zero address");
12
13 _allowances[owner][spender] = value;
14 emit Approval(owner, spender, value);
15 }
16
17 function transferFrom(address sender, address recipient,
   ↪  uint256 amount) public returns (bool) {
18  _transfer(sender, recipient, amount);
19  _approve(sender, msg.sender,
    ↪  _allowances[sender][msg.sender].sub(amount));
20  return true;
21 }
```

Fig. 8. Illustration of the overprotected *transferFrom* function.

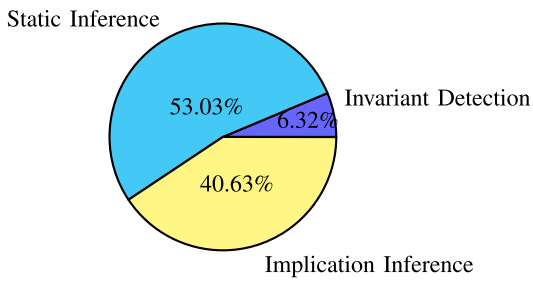

Fig. 9. The time usage by different components of InvCon+.

no more implication candidates are provided. Moreover, implication invariants generated in later iterations tend to be more intricate, resulting in more complicated SMT formulas which take more time to solve. To enhance the efficiency of InvCon+, we recommend capping the iterations used in the verification process to four; under such a setting, InvCon+demonstrates an averaged time savings of one minute in the entire verification process without compromising the quality of resulting invariants.

TABLE V
The Mutation Testing Results on ERC20 Contracts Against the Verified Invariants by InvCon+

Categories	approve	transfer	transferFrom
No. total mutants	1,539	1,141	297
No. killed mutants	998 (64.8 %)	624 (54.6 %)	101 (34.0 %)
P1. Contract invariants	245 (24.5 %)	344 (55.1 %)	55 (54.4 %)
P2. Function pre/post	763 (76.4 %)	465 (74.5 %)	61 (60.4 %)
P3. ERC20 standard	751 (75.2 %)	266 (42.6 %)	43 (42.5 %)
P4. Non-ERC20 standard	995 (99.7 %)	601 (96.3 %)	98 (97.0 %)

Additionally, we investigated further on the contracts for which InvCon+generated additional invariants deviating from the ground-truth ones. Many of these contracts are found to be non-compliant with ERC20 specifications. As illustrated in Fig. 7, we examined a real-world contract, `TokenMintERC20Token`,[4] where the `_mint` function deviates from the contract invariant **c1**—the sum of account balances always equals to the total supply—indicating non-compliance with the standard. This discrepancy arises because only 1% of the total supply tokens have been distributed to the `Account` (Line 15).

Invariant quality: Mutation testing is a technique that introduces small syntactic changes (mutants) to the program to simulate common faults. The premise is that a strong and precise set of invariants should cause many of these mutants to be rejected by the verifier–that is, the verification condition fails for mutated versions of the code. Thus, a high number of killed mutants indicates that the invariants successfully capture essential semantic properties of the program. Thus, to assess the significance of the invariants generated by InvCon+, we conducted mutation testing on the same benchmark and computed the corresponding mutation scores against these invariants. Table V presents the mutation testing results on ERC20 contracts, specifically focusing on the three most important functions: *approve*, *transfer*, and *transferFrom*. We introduced six mutation operators, such as *binary/unary-op-mutation* and *require-mutation*, along with the others, based on the mutation generator Gambit [42] developed by Certora.[5] This mutation-based approach was also adopted by Certora to evaluate the quality of smart contract specifications.[6] In total, we generated 1,539, 1,141, and 297 mutants for *approve*, *transfer*, and *transferFrom*, respectively. Table V shows that 64.8%, 54.6%, and 34.0% of mutants of approve, transfer, and transferFrom are successfully killed, respectively.

To delve into those killed mutants, in Table V, we use P1, P2, P3, and P4 to denote different types of invariants. The resulting invariants can be classified either into contract-level invariants (P1) and function-level pre/post-conditions (P2) according to context difference, or into ERC20 and non-ERC20 according to their standard compliance. The corresponding rows show the number of killed mutants by these invariants. Although the contract invariants (P1) account for 24.5% to 54.4% of the killed mutants, function pre/post-conditions (P2) demonstrate a more substantial impact occupying at most 76.4% of the killed

[4]https://etherscan.io/address/0x62c23c5f75940c2275dd3cb9300289dd30992e59
[5]https://www.certora.com/
[6]https://docs.certora.com/en/latest/docs/gambit/index.html

TABLE VI
ERC721 INVARIANTS GENERATED BY INVCON+

Category	Preconditions	Postconditions
(safe)-transferFrom	[a28, a29, a30]	[b24, b25, b26, b27, b28, b29]
approve	[a32]	[b30]
setApproveForAll	[a34]	[b31]
Contract Invariant		[c2]

mutants. Moreover, non-ERC20 standard invariants successfully eliminate nearly the entire set (96% more) of the total killed mutants. In contrast, ERC20 standard invariants eliminate a smaller set of mutants. This suggests that the invariants generated by INVCON+capture richer program semantics, contributing to a more comprehensive set of invariant specifications for smart contracts.

Interestingly, Table V reveals that only 34% of mutants related to the *transferFrom* function are successfully eliminated. Upon investigation, we discovered that *transferFrom* is overprotected, where one of its function-level preconditions is redundant. Fig. 8 depicts a common implementation of *transferFrom*, facilitating token transfer on behalf of the token owner through two internal functions, *_transfer* and *_approve*. This design rationale primarily aims at direct code reuse for the other two public functions, *transfer* and *approve*. However, in the *transferFrom* function, both requirements (Line 2 and Line 10) check if the *sender* parameter is a zero address. Consequently, mutations on either Line 2 or Line 10 do not diminish the requirements that *transferFrom* should adhere to, resulting in a low mutation score for *transferFrom*. It is noteworthy that redundant requirements in smart contracts lead to higher gas consumption during transaction execution and should be minimized whenever possible.

Invariant crowdsourcing: Less popular smart contracts may have scarce transaction histories. For example, many ERC721 contract instances may not have enough transactions to infer high-quality invariants. Each contract instance can slightly deviate from the standard specifications. Therefore, we hypothesize that reverse engineering invariants from a single contract of limited transaction histories is inferior to that from multiple contracts. We validate this hypothesis on a set of 10ERC721 contracts, restricting the evaluation to at most 200 transactions per contract. The objective is to examine INVCON+'s effectiveness in recovering the ground truth invariants listed in Table III by combining invariant results from multiple contracts. Notably, to achieve meaningful combination, every invariant result will be normalized according to a universal ERC721 definition on the name of state variables and the name of function input variables.

Table VI presents the combined invariant results from all ERC721 contracts. It demonstrates that INVCON+successfully recovers the contract invariants, all postconditions, and nearly all preconditions except **a31** and **a33**, which contain disjunctions over predicates. Consequently, the combination of invariant results from multiple contracts significantly improves the overall recall rate (14/16). **Comparison with Cider [21].** We have contacted the authors of Cider and were unable to obtain a copy of the tool at the time of writing. Unable to compare with Cider experimentally, we highlight their qualitative differences. First, the type of invariants mined by Cider is limited to only contract-level invariants. In contrast, beyond contract-level invariants, INVCON+can also support function-level pre/post-conditions. Second, all the invariants mined by Cider are related to arithmetic operation, which account for only a small proportion of invariants detected by INVCON+. For instance, the non-arithmetic condition such as **a34** and **b30** in Table III cannot be generated by Cider.

Answer to RQ1: INVCON+is able to reverse engineer standard invariant specifications from contract transaction histories, achieving a recall of 0.80, and takes no more than five minutes per contract. Additionally, the uncommon invariants generated for ERC20 contracts capture important program semantics beyond the established standards. Moreover, the evaluation on ERC721 contracts demonstrates the advantage to mine common invariants from multiple contracts and their transaction histories.

D. RQ2: Impact of Transaction Histories

The length of the transaction histories used can influence the effectiveness of INVCON+. To investigate this impact, we selected the top 10 ERC20 contracts with the longest transaction histories, ensuring that all the chosen contracts have a history of at least 10,000 transactions. In evaluating the influence of transaction history length, we employed the earliest 4,000 transactions and divided them into 20 groups, each subsequent group having 200 more transactions than the previous one.

We utilized INVCON+*Primitive*as the baseline and compared with it on the number of verified invariants and the corresponding recall score. Additionally, to explore the effect of applying the detected partially supported invariant candidates, which hold for a subset of the transaction histories, we compared INVCON+with a variant, INVCON+*w/o Partial*, that does not use these partial candidates. In this experiment, we considered the ground truth invariants from the functions which are observed in the earliest 4,000 transactions, when computing the recall score, i.e., $\mathbf{Recall_{ERC20}}$.

Fig. 10 illustrates the number of verified invariants per contract corresponding to the use of different transaction history lengths. The impact of transaction history size on the number of verified invariants is evident, with INVCON+generating the most invariants, followed by INVCON+INVCON+*w/o Partial*. This demonstrates that the partially supported invariant candidates, although do not hold on their own, may be useful in constructing richer implication invariants. By incorporating partial invariant candidates, INVCON+captures subtle contract behaviors more effectively, resulting in more comprehensive invariant specifications—approximately two times and one time more than INVCON+*w/o Partial*and INVCON+*Primitive*, respectively.

In Fig. 11, the recall score of invariant results is presented for varying transaction history lengths. Clearly, all recall scores increase with longer transaction histories, as more function

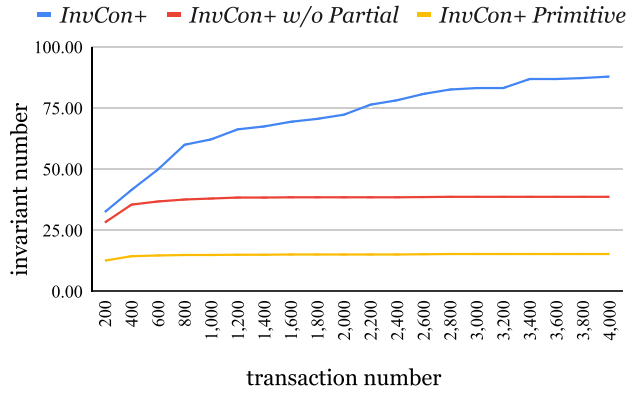

Fig. 10. The averaged number of invariants generated with different number of transactions.

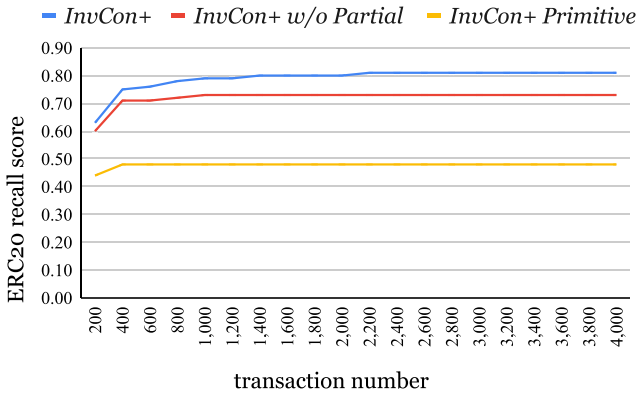

Fig. 11. The averaged ERC20 recall score of the invariant results generated with different number of transactions.

invocations are observed. Notably, InvCon+achieves a higher recall score compared to the baselines. The figure also indicates a more significant gain in recall score from 200 to 400 transactions, with negligible gains after 400, 1,000, and 2,200 transactions for InvCon+*Primitive*, InvCon+*w/o Partial*, and InvCon+, respectively. This observed difference suggests that InvCon+has a higher chance of capturing more comprehensive invariant specifications with increased transaction histories. The number of transactions would affect the overall execution time of InvCon+where longer transactions necessitate more time. However, the experiment results indicate that InvCon+is efficient and takes only 0.12 seconds on average to yield invariants per transaction. Additionally, to effectively apply InvCon+, it is recommended to use around 2,000 transactions for invariant detection.

Answer to RQ2: The scale of transaction histories affect the invariant results of InvCon+, while longer histories empower InvCon+to generate more comprehensive invariant specifications. For practicality, we recommend using 2,000 transactions for invariant detection.

TABLE VII
COMMON ERC20 VULNERABILITIES

ID	Vulnerability Types	Total	Detected
v1	batchTransfer-overflow	13	Yes
v2	totalsupply-overflow	521	Yes
v3	verify-invalid-by-overflow	2	Yes
v4	owner-control-sell-price-for-overflow	1	Yes
v5	owner-overweight-token-by-overflow	9	Yes
v6	owner-decrease-balance-by-mint-by-overflow	487	Yes
v7	excess-allocation-by-overflow	1	Yes
v8	excess-mint-token-by-overflow	9	Yes
v9	excess-buy-token-by-overflow	4	Yes
v10	verify-reverse-in-transferFrom	79	Yes
v11	pauseTransfer-anyone	1	No
v12	transferProxy-keccak256	10	Yes
v13	approveProxy-keccak256	10	Yes
v14	constructor-case-insensitive	4	N/A
v15	custom-fallback-bypass-ds-auth	1	N/A
v16	custom-call-abuse	144	N/A
v17	setowner-anyone	3	Yes
v18	allowAnyone	4	Yes
v19	approve-with-balance-verify	18	Yes
v20	check-effect-inconsistency	1	Yes
v21	constructor-mistyping	4	N/A
v22	fake-burn	2	Yes
v23	getToken-anyone	3	N/A
v24	constructor-naming-error	1	N/A

E. RQ3: Application in Securing Smart Contracts

The invariants generated by InvCon+capture the key semantics of smart contracts under normal executions, which may serve as a basis for formal contract specifications. High-quality contract specifications have been shown to be effective in securing smart contracts through runtime validation [16] and static verification [29]. To answer RQ3, we evaluated InvCon+on a set of benchmark contracts from SECBIT [43], which contains 24 types of vulnerabilities in real-world ERC20 contracts exposed to security attacks that have resulted in significant financial losses.

Table VII provides an overview of the verification results for the evaluated ERC20 contracts, categorized by vulnerability types. It contains information about the overall count of vulnerabilities and the effectiveness of our generated invariants in detecting them. The benchmark contracts used in our evaluation encompass 9 instances of integer overflow vulnerabilities and 15 other vulnerability types. However, certain vulnerabilities are beyond the scope of formal specifications, such as **v14**, **v21**, and **v24** which are related to constructor naming, **v15** and **v16** which are associated with different Solidity versions, and **v23** which pertains to function visibility. We focused on the remaining 18 types of vulnerabilities. Note that some of the vulnerabilities identified are beyond the specifications outlined in the ERC20 standard (see Table II) and they can only be detected using richer customized specifications.

```
1 function batchTransfer(address[] _receivers,
  ↪ uint256 _value) public whenNotPaused returns
  ↪ (bool) {
2 uint cnt = _receivers.length;
3 uint256 amount = uint256(cnt) * _value;
4 require(cnt > 0 && cnt <= 20);
5 require(_value > 0 && balances[msg.sender] >=
  ↪ amount);
6
7 [msg.sender] = balances[msg.sender].sub(amount);
8 for (uint i = 0; i < cnt; i++) {
9 balances[_receivers[i]] =
  ↪ balances[_receivers[i]].add(_value);
10 Transfer(msg.sender, _receivers[i], _value);
11 }
12 return true;
13 }
```

Fig. 12. batchTransfer function in BEC contract.

For each of vulnerability types, we evaluated the verification results of the invariants generated by InvCon+on the corresponding benchmark contracts. We selected the top three contracts with the highest occurrence of each vulnerability type and assessed whether the invariants detected by InvCon+could prevent the corresponding attacks on these contracts. The results are shown in Table VII. We found that InvCon+was able to detect all overflow vulnerabilities in the benchmark contracts. For instance, Fig. 12 demonstrated that InvCon+detected the integer overflow vulnerability (CVE-2018-10299) in the *batchTransfer* function of the `BEC` contract. This vulnerability is caused by the unchecked multiplication of `cnt` and `_value` in Line 3. If an attacker calls `batchTransfer` with a large `cnt` value, the unsigned integer `amount` will overflow, potentially allowing the attacker to receive more tokens than intended. However, such a transaction would violate invariant **c1** in Table II, as the `totalSupply` would no longer equal to the sum of all balances. Thus, such an attack can be effectively prevented, if the generated invariants are enforced for each function execution.

InvCon+is unable to detect some remarkable mistakes that totally deviate from programmer expectations. For example, **v11** is a vulnerability that allows any party to halt the token transfer process. This issue arises from the modification of the `onlyFromWallet` modifier, wherein "==" was mistakenly replaced with "!=". Consequently, anyone other than `walletAddress` can arbitrarily invoke the two permissioned functions: `enableTokenTransfer` and `disableTokenTransfer`. InvCon+failed to detect this vulnerability for two primary reasons. First, the `onlyFromWallet` function is not specified in the ERC20 standard, preventing the application of the existing invariant templates. Second, the contract histories contain many irregular behaviors exploiting these functions, hindering InvCon+from inferring correct invariants related to `onlyFromWallet`.

Answer to RQ3: InvCon+is able to detect invariants that are useful for preventing 17 types of real-world smart contract vulnerabilities. Enforcing invariants in contract executions may ensure the security and reliability of smart contracts.

F. Discussion

Recently, large language models have been applied to generate invariant properties for smart contracts [44], [45]. SmartInv [44] leverages fine-tuned large language models (LLMs) to automatically infer invariants for smart contracts. Although its source code is publicly available, the fine-tuned model itself is not released, limiting reproducibility and deployment. PropertyGPT [45], like SmartInv, relies on LLMs to generate invariants through prompt-based inference. All three tools, SmartInv, PropertyGPT, and InvCon+, perform formal verification of the inferred invariants, either by integrating existing verifiers such as VeriSol [29] (used by SmartInv and InvCon+) or using a custom prover (e.g., SolSEE [46] for PropertyGPT).

To distinguish InvCon+from these LLM-based approaches, we focus on the advantages of its template-based inference framework compared to the LLM-based generation used in SmartInv and PropertyGPT.

First, InvCon+offers significantly lower deployment overhead. Both PropertyGPT and SmartInv depend on resource-intensive foundation models such as GPT, which incur high computational and financial costs. Moreover, these approaches require manually curated training datasets, which are labor-intensive to construct and often limited in scope. For example, PropertyGPT is trained on only 23 annotated smart contract projects. In contrast, InvCon+is lightweight, self-contained, and easy to deploy. It does not require any labeled training data or LLM infrastructure, making it well-suited for use in continuous monitoring scenarios on blockchains without imposing additional operational burdens.

Second, the invariants produced by InvCon+are more structured and predictable. SmartInv generates invariants at arbitrary program points, including intermediate states during function execution. However, such states can be semantically ambiguous, especially in the presence of transaction reversion, making their correctness difficult to confirm. PropertyGPT, while using in-context learning to guide invariant inference, has been observed to produce invariants that do not align with any ground truth, requiring human review to assess their validity. In contrast, InvCon+enforces that all generated invariants conform to well-defined invariant templates (see Fig. 2), ensuring that all outputs fall within a semantically meaningful and verifiable class of specifications.

In summary, InvCon+distinguishes itself from LLM-driven approaches by offering a more deployable, predictable, and verifiable framework for invariant generation, making it especially suitable for real-world smart contract monitoring [47].

VII. Related Work

The related works can be broadly categorized into smart contract security analysis and invariant inference.

A. Smart Contract Security Analysis

The security analysis primarily focuses on detecting smart contract vulnerabilities. Common vulnerabilities in smart contracts include integer overflow/underflow [4], reentrancy [48],

and dangerous delegatecall operations [49]. For instance, in 2017, the Parity wallet contract was hacked due to missing protection for the delegatecall operation, a feature that allows one contract to securely delegate part of its functionality to another contract. As a result, the attacker gained control of the wallet and stole 150,000 ETH, valued at approximately $30 million USD at the time.

These common vulnerabilities have been extensively studied in [15], [50], [51], [52], [53], [54], [55], [56], [57], [58], [59], [60], [61]. Most static analysis tools, such as Slither [30], Securify [53], Zeus [58], and Ethainter [51], utilize control-flow, data-flow, or taint-flow analysis for vulnerability detection, usually achieving a high recall but low precision rate. In contrast, the others [56], [57], [62] use symbolic execution for program path exploration to identify contract vulnerabilities, along with a higher precision but lower recall rate. There are also formal verification tools for ensuring the correctness of functional properties [18], [19], [63], and workflow policy [29] in smart contracts. The dynamic analyses [15], [59], [61], [65], [66] perform random or model-based testing on smart contracts and then check execution results against predefined oracles for finding a wide range of vulnerabilities. Although these tools have been proven effective in detecting common vulnerabilities, unfortunately, Zhang et al. [67] found that only 20.5% of real-world smart contract bugs can be successfully detected by state-of-the-art tools. This is because the existing tools use simple, generic, and hard-coded security patterns or oracles, which are ineffective to recognize subtle logic bugs on specific contracts. To mitigate this gap, Zhang [68] proposes an abstract type system that can model a large part of business logic bugs into accounting errors, which are detectable with type checking.

Because there is no one-size-fits-all patterns or oracles for identifying contract logic bugs, most valued Web3 projects hire third-party security auditing companies to manually review their contracts. Despite undergoing costly code auditing, numerous projects still fall victim to security breaches [69]. In our opinion, one root cause is that contract developers and the corresponding auditors may have divergent expectations on smart contracts, which are not easy to pinpoint without sufficient contract specifications. Therefore, apart from enhancing existing security tools, the invariants generated by InvCon+can reinforce contract specifications to mitigate the incompleteness and inaccuracy issues of automated verification and contract auditing.

B. Invariant Inference

The static and dynamic invariant inference has been well-studied for traditional programs. ESC/Java [70] is a well-known static checking tool for Java programs. It leverages invariant annotations to define properties in the code, improving the precision of static checking. ESC/Java's emphasis on invariants helps developers express expectations precisely, allowing potential issues to be detected early in development. Daikon [13] is a well-known dynamic invariant detection tool to automatically infer likely invariants from program executions. Daikon takes program execution traces as input, which are typically obtained through testing. These execution traces consist of sequences of program states and variable values observed during the program's runtime. InvCon [20] was the first tool that generates *likely* invariants for smart contracts. With Daikon as the back-end invariant detection engine, InvCon implemented an intermediary input transformer that converts historic contract transactions to the compatible data trace files accepted by Daikon. In addition, some invariant templates are customized to support unique Solidity features, e.g., MappingItem. There also exist other works related to invariant generation for smart contracts. SolType [17] is a type checking tool for Solidity smart contracts. It enables developers to add refinement type annotations to smart contracts, incorporating static analysis to prove that arithmetic operations are safe from integer overflows or underflows. SolType can infer useful type annotations, but they are limited to only contract-level invariants related to arithmetic operation. Using SolType as a verifier to learn a policy, Cider [21] applies deep reinforcement learning to automatically learn contract invariants. The learned contract invariants are mainly used to guard arithmetic operations in smart contracts to avoid integer overflows and underflows. However, the correctness of the learned contract invariants is still not formally verified.

Distinguished from the aforementioned works, InvCon+is the first to implement a unified invariant generation framework for Solidity contracts encompassing techniques from both dynamic detection and static inference, where the generated invariants are verified against the contract code.

VIII. Conclusion

We have presented InvCon+, a novel invariant generation framework for Solidity smart contracts where the invariants result from the integration between dynamic invariant detection and static inference. Because implication invariants are important to capture more fine-grained program semantics of smart contracts, InvCon+devises an iterative process to repeat the generation and verification of implications to overcome its combination explosion problem. We have evaluated InvCon+on real-world ERC20 and ERC721 contracts and demonstrated that InvCon+is able to achieve good recall to recover common specifications. In addition, the experiments on mutation testing and vulnerable benchmark contracts have shown that the invariant specifications generated are effective to exclude program mistakes and make contracts secure from vulnerabilities.

Acknowledgments

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 MOE and NTU-CCTF.

References

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, 2014.
[2] "Binance smart chain," *Introduction Binance Smart Chain*, 2020. [Online]. Available: https://docs.binance.org/smart-chain/guides/bsc-intro.html,
[3] T. Chen et al., "Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1503–1520.

[4] Blockchain-Projects, "Overflow attack in ethereum smart contracts," 2020. [Online]. Available: https://blockchain-projects.readthedocs.io/overflow.html

[5] A. Shevchenko, "Bancor's bug exposes dangerously common practice in ethereum DeFi," 2020. Accessed: Jun., 6, 2020. [Online]. Available: https://cointelegraph.com/news/bancors-bug-exposes-dangerously-common-practice-in-ethereum-defi

[6] H.-A. Moon and S. Park, "Conformance evaluation of the top-100 ethereum token smart contracts with ethereum request for Comment-20 functional specifications," *IET Softw.*, vol. 16, no. 2, pp. 233–249, 2022.

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

[8] Y. Guo, "An incompatibility in ethereum smart contract threatening dApp ecosystem," 2018. [Online]. Available: https://medium.loopring.io/an-incompatibility-in-smart-contract-threatening-dapp-ecosystem-72b8ca5db4da

[9] A. Hui, "Ethereum tokens worth 1b vulnerable to 'Fake Deposit Attack'," 2020. [Online]. Available: https://www.coindesk.com/tech/2020/08/25/ethereum-tokens-worth-1b-vulnerable-to-fake-deposit-attack/

[10] "OpenZeppelin," *openZeppelin Contracts*, 2022. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts,

[11] "Inconsistency between the code and the doc of VestingWallet.release," 2022. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3368

[12] C. Zhu, Y. Liu, X. Wu, and Y. Li, "Identifying solidity smart contract API documentation errors," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Oct. 2022, pp. 1–13.

[13] "Daikon," *Daikon Invariant Detect.*, 2021. [Online]. Available: http://plse.cs.washington.edu/daikon/

[14] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for esc/java," in *Proc. Int. Symp. Formal Methods Europe*, 2001, pp. 500–517.

[15] H. Wang et al., "Oracle-supported dynamic exploit generation for smart contracts," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 3, pp. 1795–1809, May/Jun. 2022.

[16] A. Li, J. A. Choi, and F. Long, "Securing smart contract with runtime validation," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2020, pp. 438–453.

[17] B. Tan, B. Mariano, S. K. Lahiri, I. Dillig, and Y. Feng, "Soltype: Refinement types for arithmetic overflow in solidity," in *Proc. ACM Program. Lang.*, vol. 6, no. POPL, 2022, pp. 1–29.

[18] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "VERISMART: A highly precise safety verifier for ethereum smart contracts," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 1678–1694.

[19] Á. Hajdu and D. Jovanović, "Solc-verify: A modular verifier for solidity smart contracts," in *Proc. Verified Softw. Theories, Tools, Experiments, 11th Int. Conf.*, New York City, NY, USA, Jul. 2019, pp. 161–179.

[20] Y. Liu and Y. Li, "Invcon: A dynamic invariant detector for ethereum smart contracts," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2022, pp. 1–4.

[21] J. Liu, Y. Chen, B. Tan, I. Dillig, and Y. Feng, "Learning contract invariants using reinforcement learning," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2022, pp. 1–11.

[22] "Openzeppelin erc20 contract specifications," 2023. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master

[23] "Openzeppelin erc721 contract specifications," 2023. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master

[24] G. Rou, "ERC20-K: Formal executable specification of ERC20," Mar. 2023. [Online]. Available: https://github.com/runtimeverification/erc20-semantics

[25] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in c," *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 388–402, Jun., 2004.

[26] K. R. M. Leino, G. Nelson, and J. B. Saxe, "Esc/Java user's manual," *ESC*, vol. 2000, 2000, Art. no. 002.

[27] K. R. M. Leino, "This is Boogie 2," *Manuscript KRML*, vol. 178, no. 131, pp. 1–52, 2008.

[28] J. W. Nimmer and M. D. Ernst, "Invariant inference for static checking: An empirical evaluation," *ACM SIGSOFT Softw. Eng. Notes*, vol. 27, no. 6, pp. 11–20, 2002.

[29] Y. Wang et al., "Formal specification and verification of smart contracts for azure blockchain," 2018, *arXiv:1812.08829*.

[30] Slither, "Solidity source analyzer," 2021. [Online]. Available: https://github.com/crytic/slither,

[31] H. Wang, Y. Li, S.-W. Lin, L. Ma, and Y. Liu, "VULTRON: Catching vulnerable smart contracts once and for all," in *Proc. 41st Int. Conf. Softw. Eng., New Ideas Emerg. Results*, 2019, pp. 1–4.

[32] Y. Liu, Y. Liu, Y. Li, and C. Artho, "Specification mining for smart contracts with trace slicing and predicate abstraction," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reengineering*, 2025, pp. 147–158.

[33] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Proc. Formal Methods Compon. Objects, 4th Int. Symp.*,2005, Amsterdam, The Netherlands, Nov. 2006, pp. 364–387.

[34] M. Research, "Z3," 2022. Accessed: Dec., 15, 2023. [Online]. Available: https://github.com/Z3Prover/z3,

[35] E. Hildenbrandt et al., "KEVM: A complete formal semantics of the ethereum virtual machine," in *Proc. IEEE 31st Comput. Secur. Foundations Symp.*, 2018, pp. 204–217.

[36] X. Li, C. Su, Y. Xiong, W. Huang, and W. Wang, "Formal verification of BNB smart contract," in *Proc. 5th Int. Conf. Big Data Comput. Commun.*, 2019, pp. 74–78.

[37] E. ETL, "Ethereum in BigQuery: A public dataset for smart contract analytics," 2017. [Online]. Available: https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics

[38] Certora, "Securing web3 with decentralized intelligence," 2023. [Online]. Available: https://www.certora.com/

[39] "Openzeppelin pausable contract specifications," 2023. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master

[40] OpenZeppelin, "ERC 20 - OpenZeppelin docs," 2023. [Online]. Available: https://docs.openzeppelin.com/contracts/3.x/api/token/ERC20

[41] "ERC 721 - OpenZeppelin docs," 2023. [Online]. Available: https://docs.openzeppelin.com/contracts/2.x/api/token/ERC721

[42] Certora, "Gambit: Mutant generation for solidity," 2022. Accessed: Dec. 9, 2023. [Online]. Available: https://github.com/Certora/gambit

[43] S. Labs, "sec-bit/awesome-buggy-erc20-tokens: A collection of vulnerabilities in ERC20 smart contracts with tokens affected," Aug. 2018. [Online]. Available: https://github.com/sec-bit/awesome-buggy-erc20-tokens

[44] S. J. Wang, K. Pei, and J. Yang, "Smartinv: Multimodal learning for smart contract invariant inference," in *Proc. IEEE Symp. Secur. Privacy*, 2024, pp. 2217–2235.

[45] Y. Liu et al., "Propertygpt: LLM-driven formal verification of smart contracts through retrieval-augmented property generation," in *Proc. 32nd ed. Netw. Distrib. System Secur. Symp.*, 2024, pp. 1–17.

[46] S.-W. Lin, P. Tolmach, Y. Liu, and Y. Li, "Solsee: A source-level symbolic execution engine for solidity," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Foundations Softw. Eng.*, 2022, pp. 1687–1691.

[47] Z. Chen, Y. Liu, S. M. Beillahi, Y. Li, and F. Long, "Demystifying invariant effectiveness for securing smart contracts," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 1772–1795, 2024.

[48] D. Siegel, "Understanding the DAO attack," 2016. [Online]. Available: https://www.coindesk.com/understanding-dao-hack-journalists

[49] P. Santiago, "The parity wallet hack explained," 2017. [Online]. Available: https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/

[50] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. IEEE/ACM 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2019, pp. 8–15.

[51] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: A smart contract security analyzer for composite vulnerabilities," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2020, pp. 454–469.

[52] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proc. 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2018, pp. 9–16.

[53] Securify, "Software reliability lab," 2019. [Online]. Available: https://securify.ch/

[54] Y. Feng, E. Torlak, and R. Bodik, "Precise attack synthesis for smart contracts," 2019, *arXiv:1902.06067*.

[55] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 254–269.

[56] Manticore, "Symbolic execution tool for smart contracts," 2019. [Online]. Available: https://github.com/trailofbits/manticore,

[57] Mythril, *A Secur. Anal. Tool for EVM Bytecode*, 2019. [Online]. Available: https://github.com/ConsenSys/mythril
[58] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–12.
[59] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 259–269.
[60] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Foundations Softw. Eng.*, 2020, pp. 1398–1409.
[61] Echidna, "Trail of bits," 2019. [Online]. Available: https://github.com/trailofbits/echidna
[62] Oyente, "An analysis tool for smart contracts," 2019. [Online]. Available: https://github.com/melonproject/oyente,
[63] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 1661–1677.
[64] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "Sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, 2020, pp. 778–788.
[65] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Testing Anal.*New York, NY, USA, Jul., 2022, pp. 716–727.
[66] Z. Zhang, B. Zhang, W. Xu, and Z. Lin, "Demystifying exploitable bugs in smart contracts," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng.*, 2023, pp. 615–627.
[67] B. Zhang, "Towards finding accounting errors in smart contracts," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, 2024, pp. 1–13.
[68] Sayfer, "3 hacks an audit could not find," 2023. Accessed: Dec. 18, 2023. [Online]. Available: https://sayfer.io/blog/3-hacks-an-audit-could-not-find/
[69] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2002, pp. 234–245.

Ye Liu received the BE degree from Northeastern University, China, in 2016, the MSc degree from Beihang University, in 2019, and the PhD degree from the College of Computing and Data Science, Nanyang Technological University (NTU), in 2023. He was a research assistant for half a year with Cybersecurity Lab, NTU. He is currently a research scientist with Singapore Management University. His main research interests includes smart contract security and program analysis. He has won two Distinguished Paper Awards from top-tier conferences (NDSS'25 and ISSTA'22).

Chengxuan Zhang received the BE degree in computer science from the Nanyang Technological University. He started his PhD study in 2022, under the supervision of Assoc. Prof. Yi Li. He is currently working toward the PhD degree with the College of Computing and Data Science, Nanyang Technological University. His research interests mainly focus on smart contract reliability and security.

Yi Li (Member, IEEE) is currently an associate professor with the College of Computing and Data Science, Nanyang Technological University (NTU). Dr. Li has been leading the Software Reliability and Security Lab (SRSLab@NTU) since 2018. Together with his research team, he develops solutions enabling the construction of high-quality software systems that are both reliable and sustainable. His research interests include program analysis and automated reasoning techniques with applications in software engineering and software security, security and fairness of decentralized applications and blockchain systems, and robustness and dependability of AI systems. His work in these areas won five Distinguished Paper Awards and two Best Artifact Awards at top-tier conferences, including NDSS'25, ASE'23, ISSTA'22, FSE'21, ICSME'20, and ASE'15. He regularly serves on the program committees of many flagship conferences in software engineering, including ICSE, FSE, ASE, and ISSTA.