State-based Invariant Property Generation of Solidity Smart Contracts using Abstract Interpretation

Raju Halder
Indian Institute of Technology Patna, India
Email: halder@iitp.ac.in

2024 IEEE International Conference on Blockchain (Blockchain) | 979-8-3503-5159-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/BLOCKCHAIN62396.2024.00038

***Abstract*—Solidity sets its place as one of the most popular and widely-used smart contract language for Ethereum in recent years. To automate the verification of Solidity codes, especially in case of critical systems, this is intrinsic to synthesize their invariant properties which hold in every valid executions. This paper defines a formal semantics of Solidity subset and introduces an Abstract Interpretation-based sound semantics approximation to infer state-based invariant properties of Solidity contracts. The intuition behind our proposal is to apply abstraction to all possible states reachable on the executions of the contract's functions in any order under all possible instantiation context, which reflects the decentralized mining strategies in the blockchain networks. The invariant computation is expressed as a fix-point solution of state-based functions in an abstract domain of interest, based on the generic static analysis framework.**

***Index Terms*—Blockchain, Smart Contract, Solidity, Semantics, Abstract Interpretation, Invariant Property**

I. Introduction

Thirteen years after its inception, Blockchain technology [1] is set to pave the way to connect companies, industries and economies with transparency, security and trust. The support of smart contract in blockchain is a major reason behind its success today [2]. However, amid this success, the primary concerns which researchers cannot avoid are the safety and security of the smart contracts, especially in case of critical systems. Therefore, before the deployment, it is crucial to ensure that the underlying smart contracts are correct w.r.t. formal specifications, as we can not change, update, or fix any bugs due to immutability of the blockchain.

Solidity is by far considered as one of the most popular smart contract languages supported by Ethereum blockchain platform [3]. Recently, Formal methods research community has been paying enough attention to guarantee the absence of security vulnerabilities and functional correctness of Solidity smart contracts [4], [5]. Observably, most of them are making use of the off-the-shelf verification tools and techniques, thereby limiting their scope [6]–[10]. Moreover, these approaches are unable to deal with crucial program constructs, including loops, while generating verification conditions under deductive reasoning [5].

A key problem that generally hinders the automation of software verification process is the synthesize of invariant which holds in every valid executions [11]. Even though the Solidity constructs, such as require, assert, etc., facilitate code annotations to ensure code correctness during run-time, they are unable to fulfil end-to-end verifiability due to the lack of invariant generation.

This paper defines a formal semantics of Solidity subset and introduces an automatic state-based invariant property generation of Solidity smart contracts. Gaining a motivated from [12], we apply the Abstract Interpretation theory to soundly approximate all possible run-time behaviours of the contracts in a systematic and constructive way to synthesize contract's invariant. The intuition behind the proposal is to apply abstraction to all possible states reachable on the executions of the contract's functions in any order under all possible instantiation context, which reflects the decentralized mining strategies in the blockchain networks. The invariant computation is expressed as a fix-point solution of state-based functions in an abstract domain of interest, based on the generic static analysis framework.

The structure of the rest of the paper is as follows: In Section II, we define the abstract syntax and concrete semantics of a subset of Solidity programming language. Semantics-based invariant synthesis as a fix-point solution of state-based functions is described in Section III. We extend the Abstract Interpretation theory to soundly approximate Solidity semantics as a way to infer contract's invariant properties in an abstract domain of interest in Section IV. Section V illustrates our proposal with a suitable example. A case study to detect reentrancy attack using invariant is described in Section VI. Finally, Section VII concludes our paper.

II. Solidity Language: Syntax and Semantics

This section defines the syntax and semantics of Solidity Language. For simplicity in the formalism, we consider a subset of language features relevant to our objective.

2834-9946/24/$31.00 ©2024 IEEE
DOI 10.1109/Blockchain62396.2024.00038

Expressions			
exp	$\in$	E	Arithmetic Expressions
exp	::=	v	INT or UINT value
	|	id	Identifier
	|	$\text{exp}_1 \oplus \text{exp}_2$	where $\oplus \in \{+, -, *, /\}$
bexp	$\in$	B	Boolean Expressions
bexp	::=	*true* | *false*	Truth/Falsity
	|	$\text{exp}_1 \oslash \text{exp}_2$	$\oslash \in \{\geqslant, \leqslant, <, >, ==\}$
	|	$\text{bexp}_1 \otimes \text{bexp}_2$	$\otimes \in \{\vee, \wedge\}$
	|	$\neg$bexp	Negation
Commands			
sc	$\in$	SC	Smart Contracts
type	::=	*int* | *uint* | *bool*	
	|	*address* | *address payable*	Value Types
store	::=	*storage* | *memory*	Data Location
ldecl	::=	type store id(:= exp)?;	Local Declaration
assgn	::=	id := exp;	Assignment
stmt	::=	ldecl | assgn	Statement
	|	*if* bexp *then* stmt_1 *else* stmt_2 *endif*	
	|	*while* bexp *do* stmt *done*	
	|	stmt_1 stmt_2	
par	::=	type store? id | par_1, par_2	Function Parameter
constr	::=	constructor(par){stmt}	Constructor
func	::=	*function* id(par) (*returns*(par)){stmt}	Function Definition
sc	::=	⟨SV, constr, Func⟩	Smart Contract

TABLE I
ABSTRACT SYNTAX OF A SUBSET OF SOLIDITY

A. Solidity Syntax

Solidity is the most popular smart contract language among Ethereum blockchain application developers. Its design is inspired by JavaScript language. Unlike JavaScript, it is a statically typed language instrumented with many blockchain-specific features. Table I depicts the abstract syntax of its subset. The arithmetic expressions exp can either be a numerical value v or a contract variable id or an expression obtained by applying a binary arithmetic operator $\oplus$ on two arithmetic expressions. Similarly, we define boolean expression bexp as well. We define a smart contract sc as a sequence of state variables' declarations sdecl followed by an optional constructor constr and a set of functions Func. While sdecl always refers to blockchain memory, the declaration ldecl local to a functions may refer to either *storage* or *memory* as denoted by store.

Definition 1 (Smart Contract sc*):* A Solidity smart contract sc is defined by a tuple ⟨SV, constr, Func⟩, where SV is the set of state variables, constr is the contract constructor, and Func is the set of function definitions.

B. Concrete Semantics

Let us now consider various semantics domains of Solidity variables and define the concrete semantics of the language in Table I. Our semantics formalism is primarily motivated from [13].

1) Semantic Domains: Solidity is a statically typed language where the type of each variable (state and local) needs to be specified. Let us describe the domains of various data types that we consider in our Solidity grammar.

1) *int* and *uint*: There are primarily two classes of integral types in solidity: Signed Integers and Unsigned Integers. Solidity provides multiple integer types for each category, such as *int***X** and *uint***X**, where **X** ranges from 8 to 256 in steps of 8 — for example, *uint8*, *int24*, etc.
2) *address* and *address payable*: The *address* data type has a capacity of 20 bytes and is used to hold an Ethereum contract account address or an address externally owned by users. *address payable* is same as address, but provides additional member methods such as *transfer* and *send*.
3) *bool*: The only valid values for the boolean type are *false* and *true*. It is important to remember that, unlike other programming languages, Solidity cannot convert boolean type to integer.

2) Environments and States: Ethereum, taken as a whole, can be viewed as a transaction-based state machine [2]. In general, Ethereum maintains a set of users accounts, each of which is associated with a unique address. Each account, irrespective of their types, comprises the following four fields: *nonce*, *balance*, *storageRoot*, and *codeHash*. *nonce* is a scalar value that indicates the number of transactions sent from this account. In case of smart contract account, this number represents the number of contracts created by this account. *balance* indicates the number of Wei owned by this address. *storageRoot* is a 256-bit hash of the root node of a Merkle Patricia tree that encodes the storage contents of the account. The hash of the EVM code of the account is represented by *codeHash*.

In order to define the semantics of Solidity smart contracts, let us define environments, stores, and smart contracts states. Let `Var`, `Val`, `MLoc`, and `SLoc` be the set of variables, the domain of values, the set of local memory locations, and the set of blockchain storage locations respectively. The set of environments, stores and states are defined below:

- The set of local environments is defined as `LEnv` : `Var` $\longrightarrow$ `MLoc`
- The set of local stores is defined as `LStore` : `MLoc` $\longrightarrow$ `Val`
- The set of blockchain environments is defined as `BEnv` : `Var` $\longrightarrow$ `SLoc`
- The set of blockchain stores is defined as `BStore` : `SLoc` $\longrightarrow$ `Val`
- The set of states is defined as Σ : `LEnv` $\times$ `LStore` $\times$ `BEnv` $\times$ `BStore`. So, a state $\rho \in \Sigma$ is denoted by a tuple $\langle le, ls, be, bs \rangle$ where $le \in$ `LEnv`, $ls \in$ `LStore`, $be \in$ `BEnv` and $bs \in$ `BStore`.

3) Semantics: We are now in a position to define the concrete semantics of various Solidity constructs below.

a) Expressions:

$\mathscr{S}[\![v]\!] = \{(\rho, v) \mid \rho \in \Sigma\}$

$\mathscr{S}[\![\mathsf{id}]\!] = \begin{cases} \{(\rho, v) \mid \rho \in \Sigma,\ ls(le(\mathsf{id})) = v\} & \text{if } le(\mathsf{id}) \in \mathtt{LEnv} \\ \{(\rho, v) \mid \rho \in \Sigma,\ bs(be(\mathsf{id})) = v\} & \text{if } be(\mathsf{id}) \in \mathtt{BEnv} \end{cases}$

$\mathscr{S}[\![\mathsf{exp}_1 \oplus \mathsf{exp}_2]\!] = \{(\rho, v_1 \oplus v_2) \mid (\rho, v_1) \in \mathscr{S}[\![\mathsf{exp}_1]\!], (\rho, v_2) \in \mathscr{S}[\![\mathsf{exp}_2]\!]\}$

$\mathscr{S}[\![true]\!] = \{(\rho, true) \mid \rho \in \Sigma\}$

$\mathscr{S}[\![false]\!] = \{(\rho, false) \mid \rho \in \Sigma\}$

$\mathscr{S}[\![\mathsf{exp}_1 \oslash \mathsf{exp}_2]\!] = \{(\rho, v_1 \oslash v_2) \mid (\rho, v_1) \in \mathscr{S}[\![\mathsf{exp}_1]\!], (\rho, v_2) \in \mathscr{S}[\![\mathsf{exp}_2]\!]\}$

$\mathscr{S}[\![\mathsf{bexp}_1 \otimes \mathsf{bexp}_2]\!] = \{(\rho, b_1 \otimes b_2) \mid (\rho, b_1) \in \mathscr{S}[\![\mathsf{bexp}_1]\!], (\rho, b_2) \in \mathscr{S}[\![\mathsf{bexp}_2]\!]\}$

$\mathscr{S}[\![\neg \mathsf{bexp}]\!] = \{(\rho, \neg b) \mid (\rho, b) \in \mathscr{S}[\![\mathsf{bexp}]\!]\}$

b) Local Variable Declaration:

$$\mathscr{S}[\![\mathsf{type\ store\ id}]\!] \triangleq \begin{cases} \{\rho \to \rho' \mid \rho \in \Sigma\} & \text{if store is } \textit{memory} \\ \{\rho \to \rho'' \mid \rho \in \Sigma\} & \text{if store is } \textit{storage} \end{cases}$$

where $\rho = (le, ls, be, bs)$, $\rho' = (le[\mathsf{id} \to a^l_{in}], ls[a^l_{in} \to v'_d], be, bs)$ with $a^l_{in} \in \mathsf{MLoc}$ as a fresh local memory location and v'_d as the default memory value, and $\rho'' = (le, ls, be[\mathsf{id} \to a^s_{in}], bs[a^s_{in} \to v''_d])$ with $a^s_{in} \in \mathsf{SLoc}$ as a fresh storage location and v''_d as the default storage value.

$$\mathscr{S}[\![\mathsf{type\ store\ id := exp}]\!] \triangleq \begin{cases} \{\rho \to \rho' \mid \rho \in \Sigma\} \\ \quad \text{if store is } \textit{memory} \\ \{\rho \to \rho'' \mid \rho \in \Sigma\} \\ \quad \text{if store is } \textit{storage} \end{cases}$$

where $\rho = (le, ls, be, bs)$, $(\rho, v) \in \mathscr{S}[\![\mathsf{exp}]\!]$, $\rho' = (le[\mathsf{id} \to a^l_{in}], ls[a^l_{in} \to v], be, bs)$ with $a^l_{in} \in \mathsf{MLoc}$ as a fresh local memory location, and $\rho'' = (le, ls, be[\mathsf{id} \to a^s_{in}], bs[a^s_{in} \to v])$ with $a^s_{in} \in \mathsf{SLoc}$ as a fresh storage location.

c) Assignment Statement:

$$\mathscr{S}[\![\mathsf{id := exp}]\!] \triangleq \begin{cases} \{\rho \to \rho' \mid \rho \in \Sigma\} & \text{if } le(\mathsf{id}) \in \mathtt{LEnv} \\ \{\rho \to \rho'' \mid \rho \in \Sigma\} & \text{if } be(\mathsf{id}) \in \mathtt{BEnv} \end{cases}$$

where $\rho = (le, ls, be, bs)$, $(\rho, v) \in \mathscr{S}[\![\mathsf{exp}]\!]$, $\rho' = (le, ls[le(\mathsf{id}) \to v], be, bs)$, $\rho'' = (le, ls, be, bs[be(\mathsf{id}) \to v])$.

d) Sequence:

$$\mathscr{S}[\![\mathsf{stmt}_1\ \mathsf{stmt}_2]\!] \triangleq \{\rho \to \rho' \mid \rho \in \Sigma, \rho \to \rho'' \in \mathscr{S}[\![\mathsf{stmt}_1]\!], \rho'' \to \rho' \in \mathscr{S}[\![\mathsf{stmt}_2]\!]\}$$

e) Conditional:

$$\mathscr{S}[\![\textit{if } \mathsf{bexp} \textit{ then } \mathsf{stmt}_1 \textit{ else } \mathsf{stmt}_2 \textit{ endif}]\!] \triangleq \{\rho \to \rho' \mid \rho \in \Sigma,\ (\rho, true) \in \mathscr{S}[\![\mathsf{bexp}]\!],\ (\rho \to \rho') \in \mathscr{S}[\![\mathsf{stmt}_1]\!]\} \cup \{\rho \to \rho'' \mid \rho \in \Sigma,\ (\rho, false) \in \mathscr{S}[\![\mathsf{bexp}]\!],\ (\rho \to \rho'') \in \mathscr{S}[\![\mathsf{stmt}_2]\!]\}$$

f) Iteration:

$$\mathscr{S}[\![\textit{while } \mathsf{bexp} \textit{ do } \mathsf{stmt} \textit{ done}]\!] = \mathsf{lfp}^{\subseteq}_{\emptyset}(F)$$

where

$$\begin{aligned} F(\psi) = &\{\rho \to \rho' \mid \rho \in \Sigma,\ (\rho, true) \in \mathscr{S}[\![\mathsf{bexp}]\!],\ (\rho \to \rho') \in \mathscr{S}[\![\mathsf{stmt}]\!]\} \\ &\cup \{\rho \to \rho \mid \rho \in \Sigma,\ (\rho, false) \in \mathscr{S}[\![\mathsf{bexp}]\!]\} \\ &\cup \{\rho \to \rho'' \mid \rho \in \Sigma,\ (\rho, true) \in \mathscr{S}[\![\mathsf{bexp}]\!], (\rho' \to \rho'') \in \psi, \\ &\quad (\rho \to \rho') \in \mathscr{S}[\![\mathsf{stmt}]\!]\} \end{aligned}$$

g) Constructor: Let $\mathsf{sc} = \langle \mathsf{SV}, \mathsf{constr}, \mathsf{Func} \rangle$ be a smart contract where $\mathsf{constr} ::= \mathsf{constructor(par)\{stmt\}}$. Let $v_{in} \in \mathsf{Val}$ be the input values, $\rho_0 = \langle le_0, ls_0, be_0, bs_0 \rangle$ be the initial state at the time of contract deployment, and $a^l_{in} \in \mathsf{MLoc}$ and $a^b_{in} \in \mathsf{SLoc}$ be fresh memory addresses in the local memory and the blockchain storage respectively. The semantics of the contract constructor is defined as:

$$\mathscr{S}[\![\mathsf{constr(par)\{stmt\}}]\!] \triangleq \{(\rho_0, v_{in}) \to \rho' \mid \rho_0 \in \Sigma\}$$

where $\rho'' = (le_0[\mathsf{Par} \to a^l_{in}], ls_0[a^l_{in} \to v_{in}], be_0[\mathsf{SV} \to a^b_{in}], bs_0)$, $(\rho'' \to \rho') \in \mathscr{S}[\![\mathsf{stmt}]\!]$.

In the case when no constructor is defined in the contract explicitly, the default constructor will be executed. The semantics of the default constructor (denoted as constr) is defined below:

$$\mathscr{S}[\![\mathsf{constr}]\!] \triangleq \{(\rho_0, v_{in}) \to \rho' \mid \rho_0 \in \Sigma\}$$

where v_{in} represents the default values for the storage variables SV, and $\rho' = (le_0, ls_0, be_0[\mathsf{SV} \to a^s_{in}], bs_0[a^s_{in} \to v_{in}])$ with $a^s_{in} \in \mathsf{SLoc}$ as a fresh storage location.

h) Function: Let $\mathsf{Par}^{\mathsf{memory}}_{\mathsf{in}}$ and $\mathsf{Par}^{\mathsf{storage}}_{\mathsf{in}}$ be the distinct set of function parameters which are located in memory and blockchain storage respectively. Let v^{memory}_{in} and $v^{storage}_{in}$ be the input values corresponding to $\mathsf{Par}^{\mathsf{memory}}_{\mathsf{in}}$ and $\mathsf{Par}^{\mathsf{storage}}_{\mathsf{in}}$ respectively, where $v_{in} = (v^{memory}_{in} \cup v^{storage}_{in}) \in \mathsf{Val}$. The semantics of the function is defined as follows:

$$\mathscr{S}[\![\textit{function } \mathsf{id}(\mathsf{par}_{\mathsf{in}})\ (\textit{returns}(\mathsf{par}_{\mathsf{out}}))\{\mathsf{stmt}\}]\!] \triangleq \{(\rho, v_{in}) \to (\rho', v_{out}) \mid \rho \in \Sigma\}$$

where

- $\rho = (le, ls, be, bs)$,
- $\rho'' = (le[\mathsf{Par}^{\mathsf{memory}}_{\mathsf{in}} \to a^l_{in}], ls[a^l_{in} \to v^{memory}_{in}], be[\mathsf{Par}^{\mathsf{storage}}_{\mathsf{in}} \to a^s_{in}], bs[a^s_{in} \to v^{storage}_{in}])$, where $a^l_{in} \in \mathsf{MLoc}$ and $a^b_{in} \in \mathsf{SLoc}$ be fresh memory addresses in the local memory and the blockchain storage respectively.
- $(\rho'' \to \rho') \in \mathscr{S}[\![\mathsf{stmt}]\!]$ and $(\rho', v_{out}) \in \mathscr{S}[\![(\mathsf{par}_{\mathsf{out}})]\!]$

i) Smart Contract Fixpoint Semantics: We define the semantics of a contract in terms of a set of traces which represent possible evolution history of the contract after its deployment in the blockchain. The set of initial states are states obtained just after the execution of its constructor on deployment.

Definition 2 (Smart Contract Fixpoint Semantics): Let $\mathsf{sc} = \langle \mathsf{SV}, \mathsf{constr}, \mathsf{Func} \rangle$. Given an initial state ρ_0 at the time of contract deployment, the contract semantics is defined as:

$$\mathscr{S}[\![\mathsf{sc}]\!] = \mathrm{lfp}^{\subseteq}_{\emptyset} \lambda T.\ \left\{\mathscr{S}[\![\mathsf{Constr}]\!](\rho_0, v_{in}) \mid v_{in} \in \mathsf{Val}\right\} \cup \left\{\rho_0 \xrightarrow{\ell_0} \dots \xrightarrow{\ell_{n-1}} \rho_n \xrightarrow{(\mathsf{func}, v_{in})} \rho_{n+1} \mid \rho_0 \to \dots \to \rho_n \in T, (\rho_{n+1}, v_{out}) \in \mathscr{S}[\![\mathsf{func}]\!](\rho_n, v_{in}), \mathsf{func} \in \mathsf{Func}\right\}$$

where $\ell_i \in (\mathsf{Func} \times \mathsf{Val})$.

III. Semantics-based Inference of Contract Invariant

With aim of abstracting the semantics of the smart contract, let us now define the states reachable during the computation, forgeting the trace histories.

Definition 3 (Function Collecting Semantics): Let $\mathsf{func} \in \mathsf{Func}$ be a smart contract function and $S \in \wp(\Sigma)$ be a set of states. The collecting semantics of func is defined as:

$$\mathscr{S}[\![\mathsf{func}]\!](S) = \{\rho' \mid \rho \in S, v_{in} \in \mathsf{Val}, (\rho', v_{out}) \in \mathscr{S}[\![\mathsf{func}]\!](\rho, v_{in})\}$$

Definition 4 (Reachable States Semantics): The reachable states semantics of the smart contract sc in the fixpoint form is defined below:

$$\mathscr{S}[\![\mathsf{sc}]\!] = \mathrm{lfp}^{\subseteq}_{\emptyset}\ \lambda\ S.\ \mathbb{I}_0\ \cup \bigcup_{\mathsf{func} \in \mathsf{Func}} \mathscr{S}[\![\mathsf{func}]\!](S)$$

where $\mathbb{I}_0 = \left\{\mathscr{S}[\![\mathsf{Constr}]\!](\rho_0, v_{in}) \mid v_{in} \in \mathsf{Val}\right\}$.

A. Contract Invariant

Let $\mathsf{sc} = \langle \mathsf{SV}, \mathsf{constr}, \mathsf{Func} \rangle$ where $|\mathsf{Func}| = n$ be the number of functions in in sc. Recalling the definition 4, let us consider the following recursive equations:

$$S = S_0\ \cup \bigcup_{1 \leqslant i \leqslant n} S_i \tag{1}$$

where

- $S_0 = \mathbb{I}_0$
- $S_i = \mathscr{S}[\![\mathsf{func}_i]\!](S),\ 1 \leqslant i \leqslant n$

A solution to the above equation is of the form $\langle S, S_0, \dots, S_n \rangle$ where

- S denotes contract invariant which is the set of all states reachable after constructor execution and at the entry and exit points of all functions of the smart contract.
- S_0 is the constructor post-condition which is set of all states reachable after constructor execution.
- S_i is the post-condition of i-th function $\mathsf{func_i}$ which is the set of all states reachable after the execution of $\mathsf{func_i}$.

Lemma 1 (Contract Property): The tuple $\langle \mathscr{S}[\![\mathsf{sc}]\!], \mathbb{I}_0, \mathscr{S}[\![\mathsf{func}_1]\!](\mathscr{S}[\![\mathsf{sc}]\!]), \dots, \mathscr{S}[\![\mathsf{func}_n]\!](\mathscr{S}[\![\mathsf{sc}]\!]) \rangle$ is the least solution of the equation 1.

Due to incompatibility of the Contract property, in the subsequent section we use Abstract Interpretation to safely approximate it.

IV. Abstract Interpretation

Abstract Interpretation theory [14] provides a constructive and systematic way to soundly approximate program's semantics, enabling one to infer run-time behavioral properties of interest. This approximation is achieved by substituting the concrete domain of values and their basic semantic operations with their counterparts in an abstract domain of interest. In particular, the abstract domain represents a property of interest about the concrete values and the abstract operations simulate the behaviour of the concrete operations w.r.t. the property.

Let $\mathbb{C}$ and $\mathbb{A}$ be the domains of concrete semantics and abstract semantics respectively. Let $\langle \mathbb{C}, \sqsubseteq \rangle$ and $\langle \mathbb{A}, \widetilde{\sqsubseteq} \rangle$ form complete lattices where the elements in $\mathbb{C}$ and $\mathbb{A}$ are partially ordered by $\sqsubseteq$ and $\widetilde{\sqsubseteq}$ respectively. The order $x \sqsubseteq y$ (or $\tilde{x} \widetilde{\sqsubseteq} \tilde{y}$) indicates that x is more precise than y (or $\tilde{x}$ is more precise than $\tilde{y}$). To establish a correspondence between there two semantics domains, we form the Galois connection, defined as $(\langle \mathbb{C}, \sqsubseteq \rangle, \alpha, \gamma, \langle \mathbb{A}, \widetilde{\sqsubseteq} \rangle)$ where $\alpha : \mathbb{C} \longrightarrow \mathbb{A}$ and $\gamma : \mathbb{A} \longrightarrow \mathbb{C}$ are the abstraction and concretization functions respectively. Interestingly, this forms an adjunction as follows: For a given program P, let $\mathscr{S}_c[\![P]\!] \in \mathbb{C}$ and $\mathscr{S}_a[\![P]\!] \in \mathbb{A}$ be the concrete and abstract semantics of P respectively. Then, $\alpha(\mathscr{S}_c[\![P]\!]) \widetilde{\sqsubseteq} \mathscr{S}_a[\![P]\!] \Leftrightarrow \mathscr{S}_c[\![P]\!] \sqsubseteq \gamma(\mathscr{S}_a[\![P]\!])$, guaranteeing that $\mathscr{S}_a[\![P]\!]$ is a a safe approximation of $\mathscr{S}_c[\![P]\!]$.

Definition 5 (Galois Connections [14]): Given two partial order sets $L_1 = \langle S_1, \leqslant_1 \rangle$ and $L_2 = \langle S_2, \leqslant_2 \rangle$. L_1 and L_2 forms the Galois Connection, denoted by $(L_1, \alpha, \gamma, L_2)$ or $L_1 \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} L_2$ where α: $S_1 \to S_2$ and γ: $S_2 \to S_1$, if the followings hold:

- $\forall x \in S_1 :\ x \sqsubseteq \gamma \circ \alpha(x)$.
- $\forall y \in S_2 :\ \alpha \circ \gamma(y) \sqsubseteq y$.
- α and γ are monotonic.

In other words, iff $\forall x \in S_1, y \in S_2 :\ \alpha(x) \sqsubseteq y \Leftrightarrow x \leqslant \gamma(y)$.

There have been a number of relational (e.g., the domains of Parity, Sign, Intervals) and non-relational abstract domains (e.g., the domains of Difference-Bound Matrices, Octagons, Ployhedra) introduced over the last decades [15]–[18]. While the relational abstract domains preserves relation among the program variables, the non-relational abstract domain, on the other hand, care abount the properties of individual variable values. Figure 1 depicts a set of points (in • form on xy-plane) which are abstracted by both non-relational ('sign' and 'interval') and relational abstract domains ('octagon' and 'polyhedra').

Both relational and non-relational abstract domains pose certain strength and limitations in terms of computation cost and precision of the analysis. For example, as the abstractions in any non-relational abstract domains do not capture any relation among the program variables, which yields a highly approximated analysis-results with reduced computational cost. In contrast,

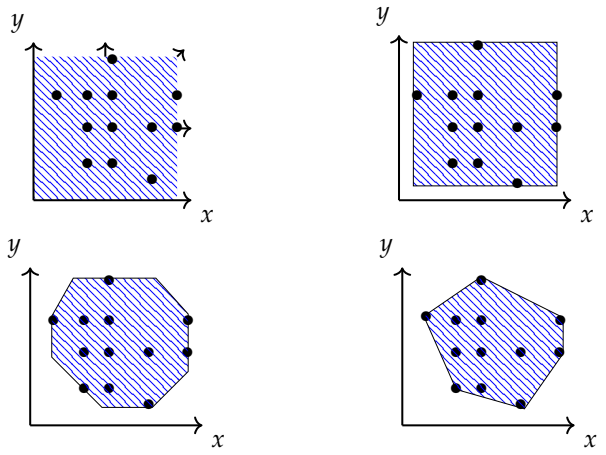

Fig. 1. Abstractions of a set of points by the domains of 'sign' (upper-left), 'interval' (upper-right), 'octagons' (lower-left) and 'polyhedra' (lower-right) [19]

while the analyses exhibit a high computation cost in relational abstract domains, the analysis results in those domains are comparatively more precise. Intuitively, preciseness of the analysis in relational abstract domain improves significantly when more number of relations among variables is encountered in the program itself. Powerset operator, on the other hand, can generate very expressive interpretations, gaining a capability of expressing the logical disjunction of the properties represented by the original domain. A summary on the strength and weakness of different relational and non-relational abstract domains is reported in Table II.

Domain	Invariants	Time cost	Memory cost	Precision
Interval	$x \in \{[l,\ h] \mid l,\ h \in \mathbb{N}, l \leq h, x \in \mathbb{V}\}$	$O(n)$	$O(n)$	*low*
Octagon	$\pm x_i \pm x_j \leqslant k, x_i, x_j \in \mathtt{Var} \wedge k \in \mathbb{N}$	$O(n^3)$	$O(n^2)$	*medium*
Polyhedra	$\Sigma_{i=1}^{n}\ a_i x_i \geqslant k,\ x_i \in \mathtt{Var}\ \wedge\ a_i, k \in \mathbb{N}^n$	$O(2^n)$	$O(2^n)$	*high*

TABLE II
A SUMMARY ON VARIOUS ABSTRACT DOMAINS [19]

A. Inference of Contract Invariant in Abstract Domain

Given the abstract domain $\widetilde{D}$, let $\mathbb{I}_0$ be the abstraction of the initial states which is obtained after the constructor is executed during smart contract deployment, such that $\mathbb{I}_0 \subseteq \gamma(\widetilde{\mathbb{I}_0})$.

The abstraction of collecting function semantics is defined as $\widetilde{\mathscr{S}}[\![\mathsf{func}]\!] \in [\widetilde{D} \to \widetilde{D}]$, such that

$$\forall S \in \wp(\Sigma):\ \mathscr{S}[\![\mathsf{func}]\!](S) \subseteq \gamma(\widetilde{\mathscr{S}}[\![\mathsf{func}]\!](\alpha(S))) \qquad (2)$$

Definition 6 (Soundness of class invariant): Let $\mathbb{I}_0 \in \widetilde{D}$ be a sound approximation of the initial states after constructor execution. Let $\mathscr{S}[\![\mathsf{func}]\!] \in [\widetilde{D} \to \widetilde{D}]$ be an abstract semantic fucntion that satisfies the Equation 2. Then the tuple $\langle \widetilde{A}, \widetilde{A_0}, \widetilde{A_1}, \ldots, \widetilde{A_n} \rangle$ is the solution of the following equation:

$$\widetilde{S} = \widetilde{S_0}\ \sqcup \bigsqcup_{1 \leqslant i \leqslant n} \widetilde{S_i} \qquad (3)$$

where

- $\widetilde{S_0} = \widetilde{\mathbb{I}_0}$
- $\widetilde{S_i} = \widetilde{\mathscr{S}}[\![\mathsf{func}_i]\!](\widetilde{S})$, $1 \leqslant i \leqslant n$

and $\mathscr{S}[\![\mathsf{sc}]\!] \subseteq \gamma(\widetilde{A})$, $\mathbb{I}_0 \subseteq \gamma(\widetilde{A_0})$, and $\forall 1 \leqslant i \leqslant n:\ \mathscr{S}[\![\mathsf{func}_1]\!](\mathscr{S}[\![\mathsf{sc}]\!]) \subseteq \gamma(\widetilde{A_i})$.

V. Illustration with an Example

Let us first provide a quick tour of static program analysis in the domain of a relational abstract domain, namely difference-bound-matrix, which we would then apply to infer contract invariant property.

A. Relational Abstract Domain of DBM

Let $\mathtt{Var} = \{\mathsf{id}_1, \ldots, \mathsf{id}_n\}$ be a finite set of program variables whose semantics value domain is $\mathtt{Val}$. Let $\mathtt{Val}_\infty = \mathtt{Val} \cup \infty$. The abstract domain of difference bound matrix (DBM) represents invariant as a conjunction of a set C of constraints, each of the form either $\mathsf{id}_j - \mathsf{id}_i \leqslant v$ or $\pm\mathsf{id}_i \leqslant v$ where $\mathsf{id}_i, \mathsf{id}_j \in \mathtt{Var}$ and $v \in \mathtt{Val}$. The encoding of these constraints in the form of DBM is defined below:

$$\mathsf{m}_{ij} \triangleq \begin{cases} v & \text{if } (\mathsf{id}_j - \mathsf{id}_i \leqslant\ v) \in C, \\ \infty & \text{otherwise.} \end{cases}$$

Observe that a DBM m can be seen as the adjacency matrix of a directed weighted graph. In case of the constraints $\mathsf{id}_i \leqslant v$ or $-\mathsf{id}_i \leqslant v$, a special variable id_0 with its semantics 0 is added to rewrite them as $\mathsf{id}_i - \mathsf{id}_0 \leqslant v$ or $\mathsf{id}_0 - \mathsf{id}_i \leqslant v$ respectively. As there may exist more than one DBM to represent the same value domain for $\mathtt{Var}$, the abstract computation considers a normal form of m which is obtained by applying a closure operation based on Bellman-Ford algorithm.

Definition 7 (Galois Connections for DBM): Let $\mathtt{L}_c = \langle \wp(\mathtt{Val}^n), \subseteq, \emptyset, \mathtt{Val}^n, \cap, \cup \rangle$ be the concrete lattice. Let $\mathbb{M}$ be the set of all closed DBMs representing the domain of difference bound matrices. Let $\mathbb{M}_\perp = \mathbb{M} \cup \{\mathsf{m}_\perp\}$ where $\mathsf{m}_\perp$ represents the bottom element that contains an unsatisfiable set of constraints. We define the abstract lattice $\mathtt{L}_a = \langle \mathbb{M}_\perp, \sqsubseteq, \mathsf{m}_\perp, \mathsf{m}_\top, \sqcap, \sqcup \rangle$ where $\mathsf{m}_\top$ represents the top element for which the bound for all constraints is ∞. The partial order, meet and join operations in $\mathtt{L}_a$ are defined as follows:

- $\forall \mathsf{m}, \mathsf{n} \in \mathbb{M}_\perp$: $\mathsf{m} \sqsubseteq \mathsf{n} \iff \forall i, j$: $\mathsf{m}_{ij} \leqslant \mathsf{n}_{ij}$.
- $\forall \mathsf{m}, \mathsf{n} \in \mathbb{M}_\perp$: $(\mathsf{m}\ \sqcap\ \mathsf{n}) = \mathsf{m}'$ where $\forall i, j$: $\mathsf{m}'_{ij} \triangleq \min(\mathsf{m}_{ij}, \mathsf{n}_{ij})$.
- $\forall \mathsf{m}, \mathsf{n} \in \mathbb{M}_\perp$: $(\mathsf{m}\ \sqcup\ \mathsf{n}) = \mathsf{m}'$ where $\forall i, j$: $\mathsf{m}'_{ij} \triangleq \max(\mathsf{m}_{ij}, \mathsf{n}_{ij})$.

The Galois connection between $\mathtt{L}_c$ and $\mathtt{L}_a$ is formalized as $\langle \mathtt{L}_c, \alpha, \gamma, \mathtt{L}_a \rangle$ where α and γ on $S \in \wp(\mathtt{Val}^n)$ and $\mathsf{m} \in \mathbb{M}_\perp$ are defined below:

$$\alpha(S) \triangleq$$

$$\begin{cases} \mathsf{m}_\perp & \text{if } S = \emptyset \\ \mathsf{m} \text{ where } \mathsf{m}_{ij} \triangleq max\{\rho(\mathsf{id}_j) - \rho(\mathsf{id}_i) \mid \rho \in S\} & \text{otherwise} \end{cases}$$

$$\gamma(\mathsf{m}) =$$

$$\begin{cases} \emptyset & \text{if } \mathsf{m} = \mathsf{m}_\perp \\ \mathtt{Val}^n & \text{if } \mathsf{m} = \mathsf{m}_\top \\ \left\{(v_1, \ldots, v_n) \in \mathtt{Val}^n \mid (v_1, \ldots v_n) \in \mathtt{dom}(\mathsf{m})\right\} & \text{otherwise} \end{cases}$$

Operators and Transfer Functions*:* Given the set B of boolean expressions, the semantics of boolean expression is defined in terms of state-filtering based on the boolean satisfiability as follows: $\mathscr{S}_f : \left(B \mapsto \wp(\Sigma)\right) \mapsto \wp(\Sigma)$. The corresponding sound abstract function $\widetilde{\mathscr{S}_f}$ in the domain of DBM is defined as $\widetilde{\mathscr{S}_f} : (B \mapsto \mathbb{M}_\perp) \mapsto \mathbb{M}_\perp$. Similarly, given the set ST of Solidity statements, the semantic function for the effects of commands on concrete states is defined as $\mathscr{S}_c : \left(ST \mapsto \wp(\Sigma)\right) \mapsto \wp(\Sigma)$ and its corresponding sound abstract function $\widetilde{\mathscr{S}_c}$ in DBM is defined as $\widetilde{\mathscr{S}_c} : (ST \mapsto \mathbb{M}_\perp) \mapsto \mathbb{M}_\perp$.

a) Guard:: Given a boolean expression $b \in B$ which defines a constraint over `Var`. The boolean transfer function over an abstract state in the domain of DBM is defined below, aiming at achieving an overapproximation $\gamma(\mathsf{m}_{+b}) \supseteq \{\rho \in \gamma(\mathsf{m}) \mid \rho \text{ satisfies } b\}$:

- If $b = (\mathsf{id}_{j_0} - \mathsf{id}_{i_0} \leqslant v)$ with $\mathsf{id}_{j_0} \neq \mathsf{id}_{i_0}$, then $\widetilde{\mathscr{S}_f}[\![\mathsf{id}_{j_0} - \mathsf{id}_{i_0} \leqslant v]\!]\mathsf{m} = \mathsf{m}'$where
$$\mathsf{m}'_{ij} \triangleq \begin{cases} \min(\mathsf{m}_{ij}, v) & \text{if } i = i_0 \text{ and } j = j_0, \\ \mathsf{m}_{ij} & \text{otherwise} \end{cases}$$
- If $b = (\mathsf{id}_{j_0} - \mathsf{id}_{i_0} = v)$ with $\mathsf{id}_{j_0} \neq \mathsf{id}_{i_0}$, then $\mathsf{m}' = \mathsf{m}_{+(\{\mathsf{id}_{j_0} - \mathsf{id}_{i_0} \leqslant v,\ \mathsf{id}_{i_0} - \mathsf{id}_{j_0} \leqslant -v\})}$

b) Assignment:: Given an assignment statement $\mathsf{id}_i := \mathsf{exp}$ and an abstract state m representing the DBM constraints, the abstract semantics of the assignment statement on m yields m' which is an upper approximation such that

$$\gamma(\mathsf{m}') \supseteq \{\rho[v_i \leftarrow v] \mid \ \rho \in \gamma(\mathsf{m}) \wedge v = \mathscr{T}_e[\![\mathsf{exp}]\!]\rho\} \text{ where}$$

$\mathscr{T}_e$ is the semantic function of arithmetic expression and $\rho[v_i \leftarrow v]$ denote ρ with its i^{th} component changed into v. The following two possible assignment cases may arise:

1) $\mathsf{id}_{i_0} = \mathsf{id}_{i_0} + v$: In this case, we subtract v from inequalities having negative coefficient for id_{i_0} and we add v to inequalities having positive coefficient for id_{i_0}. Therefore,

$$\widetilde{\mathscr{T}_c}[\![\mathsf{id}_{i_0} = \mathsf{id}_{i_0} + v]\!]\mathsf{m} = \mathsf{m}' \text{ where}$$
$$\mathsf{m}'_{ij} \triangleq \begin{cases} \mathsf{m}_{ij}\text{-}v & \text{if } i = i_0,\ j \neq j_0 \\ \mathsf{m}_{ij}\text{+}v & \text{if } i \neq i_0,\ j = j_0, \\ \mathsf{m}_{ij} & \text{otherwise} \end{cases}$$

2) $\mathsf{id}_{i_0} = \mathsf{id}_{j_0} + v$ and $i_0 \neq j_0$: In this case, the inequalities $\mathsf{id}_h - \mathsf{id}_l \leqslant v$ and $\mathsf{id}_l - \mathsf{id}_h \leqslant -v$ are added into the octagon. Therefore,

$$\widetilde{\mathscr{T}_c}[\![\mathsf{id}_{i_0} = \mathsf{id}_{i_0} + v]\!]\mathsf{m} = \mathsf{m}' \text{ where } \mathsf{m}' = (\mathsf{m} \backslash \mathsf{id}_{i_0})_{+(\{\mathsf{id}_{i_0} - \mathsf{id}_{j_0} \leqslant v, \mathsf{id}_{j_0} - \mathsf{id}_{i_0} \leqslant -v\})}$$

B. Inferring Invariant in the domain of DBM

Let us illustrate this using code snippet 1 to understand better how the proposed analysis works.

```
contract EtherStore {
    mapping(address => uint) public balances;

    function deposit(address x) public payable
    {
        balances[x] += 100;
    }

    function withdraw(address from, uint amt) public
    {
        if(balances[from]>=50){
             from.transfer(50);
             balances[from] - = 50;
    }

   function getBalance(address a) public view returns (uint)
    {
        return balances[a];
    }
}
```

Code Snippet 1. Lottery Example in Solidity Language

Consider the domain of difference bound matrices $\mathsf{L}_a = \langle \mathbb{M}_\perp, \sqsubseteq, \mathsf{m}_\perp, \mathsf{m}_\top, \sqcap, \sqcup \rangle$. Let $\mathsf{m}_\perp$ be the initial abstract state on which the constructor will execute on contract deployment. Accordingly, the initial solution of Equation 3 in the domain of DBM is represented by the tuple $R_0 = \langle \mathsf{m}_\perp, \mathsf{m}_\perp, \mathsf{m}_\perp, \mathsf{m}_\perp \rangle$, where the elements in order represent contract invariant, initial state after constructor execution, possible states after deposit(), and possible states after withdraw() respectively. As the function getBalance() does not change any state, we ignore this function in invariant computation.

After the constructor is executed on deploying the contract, the initial abstract state would be represented by the difference bound matrix $\mathsf{m}_0 = \{-balanace[i] \leqslant 0\}$. Therefore, the contract invariant becomes $\langle \perp \rangle \sqcup \langle \mathsf{m}_0 \rangle = \langle \mathsf{m}_0 \rangle$. This leads to the solution $R_1 = \langle \mathsf{m}_0, \mathsf{m}_0, \perp, \perp \rangle$.

On executing the functions deposit() and withdraw(), we obtain the following solutions in the subsequent iteration: $m_1 = \mathscr{S}[\![deposit()]\!]m_0 = \{-balanace[i] \leqslant [-100, 0]\}$ and $m_2 == \mathscr{S}[\![withdraw()]\!]m_0 = \{-balanace[i] \leqslant [-50, 0]\}$.

As $\langle \mathsf{m}_0 \rangle \sqcup \langle \mathsf{m}_1 \rangle \sqcup \langle \mathsf{m}_2 \rangle = \mathsf{m}_1$, the solution in the next iteration is $R_2 = \langle \mathsf{m}_1, \mathsf{m}_0, \mathsf{m}_1, \mathsf{m}_2 \rangle$. As further iteration yields the same result R_2, we reach a fixpoint solution where the contract invariant guarantees that the balance would always be positive.

If we consider a variant of the above code where deposit amount x and withdraw amount y are passed as function parameters, enabling the user to deposit and withdraw any amount of her choice, we may consider the abstract domain of polyhedra to acheive a sound approximation of the invariant property of the contract.

VI. A Case Study: detecting reentrancy attack using invariant property

The DAO attack is one of the most infamous Solidity hacks which took place in the year 2016, where the attacker was able to steal 3.6 million of Ether. The hack used an exploit commonly referred to as a reentrancy attack [20]. As shown in the code snippet 2, the attacker embeds the withdraw function (defined in DAO.sol) in the fallback function of the smart contract DAOAttack.sol. This enables a recursive call to the withdraw function (hence withdrawing Ethers repeatedly before the balance is set to 0) via the the fallback function whenever the attacker receives any funds.

```
1  //DAO.sol: The DAO contract
2  contract DAO {
3      mapping(address => uint) public balances;
4
5      function deposit() public payable {
6          balances[msg.sender] += msg.value;
7      }
8
9      function withdraw() public {
10         uint bal = balances[msg.sender];
11         if (bal > 0) {
12           msg.sender.call.value(bal)();
13           balances[msg.sender] = 0;}
14     }
15
16     ......
17 }
18
19 //DAOAttack.sol: Attacker Contract
20 contract DAOAttack {
21     DAO public contract;
22
23     constructor(address DAOaddress) {
24         contract = DAO(DAOaddress);
25     }
26
27     // Fallback function.
28     fallback() external payable {
29         contract.withdraw();
30     }
31
32     function attack() external payable {
33         require(msg.value >= 1 ether);
34         contract.deposit{value: 1 ether}();
35         contract.withdraw();
36     }
37 }
```

Code Snippet 2. The DAO attack

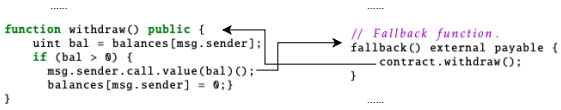

Fig. 2. Snapshot of the external call

The snapshot of the recursive call between the withdraw function and fallback function is depicted in Figure 2. Let us now perform semantics-based static analysis of the above code in the abstract domain of intervals, aiming at inferring invariant which is suitable for detecting the presence of reentrancy.

Let $\mathsf{L}_c = \langle \wp(\mathbb{N}), \subseteq, \emptyset, \mathbb{N}, \cap, \cup \rangle$ be a concrete lattice of the powerset of numerical values $\mathbb{N}$. Let $\mathbb{I} = \{[l,\ h] \mid l \in \mathbb{N} \cup \{-\infty\},\ h \in \mathbb{N} \cup \{+\infty\},\ l \leq h\} \cup \bot$ be the abstract domain of intervals forming an abstract lattice $\mathsf{L}_a = \langle \mathbb{I}, \sqsubseteq, \bot, [-\infty, +\infty], \sqcap, \sqcup \rangle$, such that:

- $[l_1,\ h_1] \sqsubseteq [l_2,\ h_2] \iff l_2 \leqslant l_1 \wedge h_2 \geqslant h_1$
- $[l_1,\ h_1] \sqcap [l_2,\ h_2] = [max(l_1\ l_2),\ min(h_1\ h_2)]$
- $[l_1,\ h_1] \sqcup [l_2,\ h_2] = [min(l_1,\ l_2),\ max(h_1\ h_2)]$

The Galois connection between L_c and L_a is formalized as $\langle \mathsf{L}_c,\ \alpha,\ \gamma,\ \mathsf{L}_a \rangle$ where $\forall S \in \wp(\mathbb{N})$ and $\forall \tilde{v} \in \tilde{\mathbb{I}}$:

$$\alpha(S) = \begin{cases} \bot & \text{if } S = \emptyset \\ [l,\ h] & \text{if } min(S) = l \ \wedge \ max(S) = h \\ [-\infty,\ h] & \text{if } \nexists min(S) \ \wedge \ max(S) = h \\ [l,\ +\infty] & \text{if } min(S) = l \ \wedge \ \nexists max(S) \\ [+\infty,\ -\infty] & \text{if } \nexists min(S) \ \wedge \ \nexists max(S); \end{cases}$$

$$\gamma_{\mathbb{I}}(\tilde{v}) = \begin{cases} \emptyset & \text{if } \tilde{v} = \bot \\ \{k \in \mathbb{R} \mid \ l \leq k \leq h\} & \text{if } \tilde{v} = [l,\ h] \\ \{k \in \mathbb{R} \mid \ k \leq h\} & \text{if } \tilde{v} = [-\infty,\ h] \\ \{k \in \mathbb{R} \mid \ l \leq k\} & \text{if } \tilde{v} = [l,\ +\infty] \\ \mathbb{R} & \text{if } \tilde{v} = [+\infty,\ -\infty]. \end{cases}$$

The pictorial representation of the Galois connection $\langle \mathsf{L}_c, \alpha, \gamma, \mathsf{L}_a \rangle$ is shown in Figure 3.

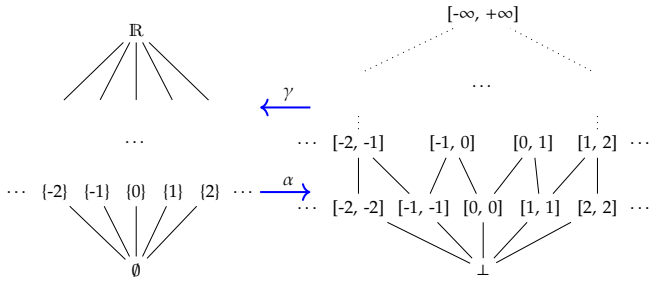

Fig. 3. Galois Connection between L_c and L_a

Approximation of fix-point semantics using Widening:

Let $\mathcal{D}$ be the semantic domain. Let $\text{lfp}_{\bot}\mathcal{F}$ where $\mathcal{F} \in \mathcal{D} \mapsto \mathcal{D}$ be the semantics of a given program P. Assume the Galois Connection $\langle (\mathcal{D}, \mathcal{F}, \bot), \alpha, \gamma, (\widetilde{\mathcal{D}}, \widetilde{\mathcal{F}}, \widetilde{\bot}) \rangle$ where α and γ are abstraction and concretization functions respectively, such that $\text{lfp}_{\widetilde{\bot}}(\widetilde{\mathcal{F}})$ is not computable iteratively in finitely many steps. Suppose $\widetilde{\mathcal{A}}$ is an upper approximation of $\text{lfp}_{\widetilde{\bot}}(\widetilde{\mathcal{F}})$ which is effectively computed using widening [21]. Since $\text{lfp}_{\bot}\mathcal{F} \sqsubseteq \text{lfp}_{\widetilde{\bot}}(\widetilde{\mathcal{F}})$ and $\text{lfp}_{\widetilde{\bot}}(\widetilde{\mathcal{F}}) \sqsubseteq \widetilde{\mathcal{A}}$, so by monotonicity and transitivity, we get $\text{lfp}_{\bot}\mathcal{F} \sqsubseteq \widetilde{\mathcal{A}}$.

Let us recall from [21] the definition of widening operator in the domain of Intervals:

Definition 8 (Widening): Let $\langle \mathcal{D}, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ be a lattice. The partial operator $\nabla : \mathcal{D} \times \mathcal{D} \mapsto \mathcal{D}$ is a widening if

- for each $x, y \in \mathcal{D}$: $x \sqsubseteq x \nabla y$ and $y \sqsubseteq x \nabla y$.
- for each increasing chain $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$, the increasing chain defined by $y_0 = x_0$, $y_{n+1} = y_n \nabla x_{n+1}$ for $n \in \mathbb{N}$, is not strictly increasing.

Theorem 1 (Convergence): Given a sequence $\{x_0, x_1, x_2, \ldots\ldots\}$, the increasing chain $\langle y_k, k \in \mathbb{N}\rangle$ defined by $y_0 = x_0$ and

$$y_{n+1} := \begin{cases} y_n & \text{if } \exists l \leq n : \ x_{n+1} \sqsubseteq y_l \\ y_n \ \nabla \ x_{n+1}, & \text{otherwise} \end{cases}$$

is strictly increasing up to a least $l \in \mathbb{N}$ such that $x_l \sqsubseteq y_l$ and the sequence is stationary at l onwards.
The following example defines a widening operator for the domain of intervals.

Example 1 (Widening for intervals): Consider a lattice of intervals $\mathcal{D} = \{\bot\} \cup \{[l, u] \mid l \in \mathbb{Z} \cup \{-\infty\}, \ u \in \mathbb{Z} \cup \{+\infty\}\}$ ordered by $\forall x \in \mathcal{D}$, $\bot \leq x$ and $[l_0, u_0] \leq [l_1, u_1]$ if $l_1 \leq l_0$ and $u_0 \leq u_1$. Let k be a fixed positive integer constant, and I be any set of indices. The following defines a threshold widening operator defined on $\mathcal{D}$ by

$$\nabla^k(\{\bot\}) = \bot \qquad \nabla^k(\{\bot\} \cup S) = \nabla^k(S)$$

$$\nabla^k(\{[l_i, u_i] : \ i \in I\}) = [h_1, h_2]$$

where

$$h_1 = min\{l_i : \ i \in I\} \ \text{ if } \ min\{l_i : \ i \in I\} > -k \text{ else } -\infty$$

$$h_2 = max\{u_i : \ i \in I\} \ \text{ if } \ max\{u_i : \ i \in I\} < k \text{ else } +\infty$$

Performing the inter-procedural data flow analysis [22] of the above code snippet in the domain of Intervals, we observe that the attack function in line 32 initiates balances[msg.sender] with [1,1]. Following this, the interval analysis (using the widening operator) of the recursive call (shown in Figure 2) via the fallback function yields the abstract property of the account balance of the sender (represented by msg.sender.balance) as [1,∞]. Moreover, resetting of the balances in line 13 makes balances[msg.sender] as [0,1]. This clearly reveals the violation of the expected outcome.

VII. Conclusion and Future Plan

In this paper, we applied the Abstract Interpretation theory to compute abstract semantics of Solidity contracts in a systematic and constructive way. This enables one to infer state-based invariant properties of Solidity smart contracts w.r.t. the properties of interest. This generic framework allows one to apply appropriate abstract domains representing the properly of interest, achieving a trade-off between computation cost and precision. While we presented a novel semantics-based perspective to address Solidity vulnerabilities by adopting Abstract Interpretation framework, our future plan involves its practical evaluation and comparison w.r.t. the state-of-the-art solutions.

Acknowledgement

This research is partially supported by the Core Research Grant (CRG/2022/005794) from the Science and Engineering Research Board (SERB), Department of Science and Technology, Government of India.

References

[1] S. Nakamoto. (2008) Bitcoin: A peer-to-peer electronic cash system. [Online]. Available: https://bitcoin.org/bitcoin.pdf
[2] G. Wood. (2014) Ethereum: A secure decentralised generalised transaction ledger. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf
[3] Ethereum. Solidity documentation – release 0.8.4. [Online]. Available: http://solidity.readthedocs.io/
[4] T. Durieux *et al.*, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: ACM, 2020, p. 530–541.
[5] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, "A survey of smart contract formal specification and verification," *ACM Comput. Surv.*, vol. 54, no. 7, jul 2021.
[6] L. Luu *et al.*, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Vienna, Austria: ACM, 2016, p. 254–269.
[7] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Proc. of the Network and Distributed Systems Security Symposium (NDSS'18)*, 01 2018.
[8] J. Feist, G. Greico, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, ser. WETSEB '19. IEEE Press, 2019, p. 8–15.
[9] P. Tsankov *et al.*, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 67–82.
[10] S. Tikhomirov *et al.*, "Smartcheck: Static analysis of ethereum smart contracts," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2018, pp. 9–16.
[11] P. Cousot, R. Giacobazzi, and F. Ranzato, "Program analysis is harder than verification: A computability perspective," in *Computer Aided Verification*. Springer LNCS, 2018, pp. 75–95.
[12] F. Logozzo, "Class invariants as abstract interpretation of trace semantics," *Computer Languages, Systems and Structures*, vol. 35, no. 2, pp. 100–142, 2009.
[13] C. Marco, C. Gabriele, and A. Vincenzo, "Towards an operational semantics for solidity," in *The Eleventh International Conference on Advances in System Testing and Validation Lifecycle (VALID 2019)*, 2019, pp. 1–6.
[14] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. of the POPL'77*, 1977, pp. 238–252.
[15] L. Chen, A. Minè, and P. Cousot, "A Sound Floating-Point Polyhedra Abstract Domain," in *Proc. of the 6th Asian Symposium on PLS*, 2008, pp. 3–18.
[16] P. Cousot and N. Halbwachs, "Automatic Discovery of Linear Restraints Among Variables of a Program," in *Proceedings of the POPL '78*, 1978, pp. 84–96.
[17] A. Miné, "A New Numerical Abstract Domain Based on Difference-Bound Matrices," in *Programs as Data Objects, Second Symposium, PADO*, 2001, pp. 155–172.
[18] A. Minè, "The Octagon Abstract Domain," *Higher Order Symbol. Comput.*, vol. 19, no. 1, pp. 31–100, 2006. http://www.astree.ens.fr/.
[19] A. Jana *et al.*, "Extending abstract interpretation to dependency analysis of database applications," *IEEE Transactions on Software Engineering*, vol. 46, no. 5, pp. 463–494, 2018.
[20] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security analysis methods on ethereum smart contract vulnerabilities: A survey," 2020.
[21] A. Cortesi and M. Zanioli, "Widening and narrowing operators for abstract interpretation," *Computer Languages, Systems & Structures*, vol. 37, pp. 24–42, 2011.
[22] N. D. Jones and S. S. Muchnick, "A flexible approach to interprocedural data flow analysis and programs with recursive data structures," in *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on POPL*. Albuquerque, New Mexico: ACM, 1982, p. 66–74.