Chapter 7

Analyzing information flow in solidity smart contracts

Raju Halder[a], Md. Imran Alam[a,b], Akshay M. Fajge[a], Neeraj Kumar Singh[c], and Agostino Cortesi[b]
[a]*Indian Institute of Technology Patna, Patna, Bihar, India,* [b]*Università Ca' Foscari Venezia, Venezia, Italy,* [c]*INPT-ENSEEIHT/IRIT, University of Toulouse, Toulouse, France*

1. Introduction

About 12 years after its inception, Blockchain technology [1] is set to pave the way to connect companies, industries, and economies with transparency, security, and trust. As blockchain technology progressed from versions 1.0 to 4.0, it provided support for smart contracts. Smart contracts are executable codes that run on the blockchain to enable decentralized implementation and enforcement of agreements between untrustworthy parties [2]. With the addition of smart contracts, the technology enables the creation of decentralized apps, making it a good competitor for a vast number of applications in banking, finance, supply chain management, insurance, and real estate [3]. Therefore, paying attention to the quality of smart contracts is of the utmost importance, as once deployed, it is hard to change, update, or fix any bugs.

Solidity is by far considered as one of the most popular smart contract languages supported by Ethereum blockchain platform [4, 5]. Even though the Ethereum core community is continuously working hard to ease and improve development with Solidity, a number of security attacks [6] on deployed Ethereum have occurred in the recent past. One of the most challenging issues in this context is maintaining the confidentiality and integrity of sensitive data during smart contract development [7, 8]. *Confidentiality* refers to restricting sensitive information's access and disclosure to only authorized users. For example, when shopping online, sensitive information such as credit card details must be provided only to the payment gateway, not to the merchant or any other third party. The term *integrity* refers to the protection of data from unauthorized alterations in order to make sure its reliability and consistency. Standard security measures such as encryption and access control can effectively restrict information leakage at the source. However, they are incapable of ensuring the confidentiality and integrity of data during system execution.

To exemplify this, let us categorize the programs variables into two distinct sets, *private* (denoted by h) and *public* (denoted by l), with high and low sensitivity levels, respectively. An attacker can guess the sensitive values of private variables h by observing the values of the public variables l on the output channel, if the program contains an assignment statement $l := h$; or a conditional statement `if (h == 0) then l := 20; else l := -20`. The former is called a direct/explicit flow where the values of l are directly influenced by the values of h, whereas the latter is called indirect/implicit flow where the values of l are indirectly influenced by the values of h present in the guard. While the above is related to the confidentiality of sensitive data, a dual of this problem is the integrity where tainted input data flow along the program code and affect/corrupt critical computations [9].

Code Snippets 1 and 2 demonstrate an instance of an integrity breach. The random function in Code Snippet 1 generates a random integer using the block timestamp and later performs a critical task based on the generated random number. In line 2, timestamp of the block is assigned to a private variable salt, which is used to calculate the values of parameters x, y, and seed. Finally, it returns a random number whenever the function is externally called.

Code Snippet 1

Timestamp dependency [6]

```
1 ....
2 uint256 constant private salt=block.timestamp;
3 function random(uint Max) constant private returns (uint256 result){
4 uint256 x = salt * 100 /Max;
```

Distributed Computing to Blockchain. https://doi.org/10.1016/B978-0-323-96146-2.00024-3

```
5 uint256 y = salt * block.number / (salt % 5) ;
6 uint256 seed = block.number/3 + (salt % 300) + Last_Payout + y;
7 uint256 h = uint256(block.blockhash(seed));
8 return uint256((h/x)) % Max + 1 // random number between 1 and Max
9 }
```

Code Snippet 2 implements the condition where the random function is called in line 3 and its return value is assigned to the variable roll. Observe that the success of the critical computation at line 6 depends on the value of roll. A malicious miner can take advantages by modifying the local system's timestamp to trigger this call.

Code Snippet 2

Timestamp dependency [6]

```
1 ....
2 if( (deposit > 1 ether ) && (deposit > players[Payout_id].payout)){
3 uint roll = random(100); // create a random number
4 if( roll % 10 == 0 ) {
5 msg.sender.send(WinningPot);
6 WinningPot=0;
7 }}
```

Language-based information flow security analysis [7, 8, 10, 11] has developed as a viable technique for identifying undesired information flows within software systems, preventing unauthorized disclosure of sensitive information or alteration of crucial data. It captures security vulnerabilities at the application level. The fundamental advantage of language-based security analysis is the ability to naturally describe security policies and enforcement mechanisms employing well-developed formal techniques. These techniques enable both the formulation and verification of rigorous security policies.

The purpose of this chapter is to extend the traditional language-based information-flow analysis [12, 13] to the case of Solidity smart contracts in order to identify potential undesirable information flow within the contract code that could result in a violation of confidentiality or integrity. To summarize, our main contributions in this chapter are:

- We apply data-flow analysis which accumulates (direct/indirect) dependency information among the smart contract variables, indicating possible information flow along all possible paths in the smart contract.
- To this end, we define concrete semantics of Solidity language and we design its abstraction in the domain of positive propositional formula `Pos` which encodes variables' dependences in the form of logical formula.
- We detect insecure information flow based on the truth value assignments to these propositional variables according to the security levels of program's variables and the satisfiability checking of the logical formula accumulated at each program point of the program.
- We address both the flow and context sensitivity aiming to achieve more precision in the analysis results.
- We discuss the possibility of the integration of numerical abstract domains along with `Pos` as a way to improve precision of the analysis results by removing possible false dependences.

The remainder of this chapter is organized as follows: Section 2 summarizes the current state of the art in the literature. Section 3 discusses background information of Blockchain Technology, Language-based Information Flow, and Data-flow Analysis. The abstract syntax and concrete semantics of a subset of Solidity programming language are defined in Section 4. We define the dependency analysis of Solidity in the domain of `Pos` in Section 5 and the verification of undesired information flow is described in Section 6. We discuss possible improvement of the analysis by combining with other numerical abstract domains in Section 7. Finally, we conclude the chapter in Section 8.

2. Related work

Information flow security analysis involves statically determining how a program's outputs are related to its inputs, that is, how the former depend, directly or indirectly, on the latter. A series of works on information flow security analysis have been proposed for various programming paradigms [8, 10, 12, 14–20]. Denning in his seminal paper [10] first introduced lattice-based model to prevent sensitive information leakage. The lattice model defines a partial order among various security labels (e.g., $low \leqslant high$) and allows only upward flow of information to preserve the confidentiality of sensitive information.

The most popular approach to information flow analysis is security-type systems that attach security levels (e.g., *low* or *high*) to various specific language constructs and check the improper flow of sensitive information by using a set of typing rules [7, 8, 11, 21, 22]. A sound security-type system assures that no *high* inputs will flow to *low* outputs either directly, via data-flow (explicit-flow), or indirectly, via control-flow (implicit-flow), during program execution [14]. Authors in Ref. [21] present a type-based information flow analysis for ML. A generic type system for Java-like language is proposed in Ref. [22].

Beside the security-type systems, program dependence graph (PDG) is also another promising technique for information flow security analysis of programs [14, 23–25]. Unlike security-type systems, PDG-based approach is context, object, and flow sensitive, and can handle industrial-level programs.

Various formal method techniques such as model checking, abstract interpretation, Hoare logic, etc., have also been applied to detect improper information flow in the software systems [13, 20, 26–30]. Authors in Ref. [29] introduced a semantics characterization-based information flow analysis of programs. The semantics characterization is expressed in terms of equality between the programs and is capable enough to capture various covert flows as well as nontermination. Abstract interpretation-based information flow analysis of imperative programs in the domain of positive propositional formulae is presented in Ref. [13, 30]. In particular, variables dependencies are represented in the form of logic formula, which is then verified in terms of satisfiability after assigning truth values according their sensitivity levels. The analysis precision is then improved by combining with it the analysis results in a numerical abstract domain. Further, the authors in Refs. [12, 31, 32] extended the approach to the case of database languages. Hoare-like logic-based interprocedural information flow analysis of object-oriented program is discussed in Ref. [20]. The principle of noninterference property is expressed by defining assertion $a^{\bowtie}$, which states that two concurrent states s and s' agree on the values of a (i.e., there is no leak of private information via a). This assertion is then statically checked by using Hoare-like logic whose judgments take the form $\Vdash\{\phi\}\mathsf{S}\{\psi\}$, where ϕ are the assertions hold before execution of S, ψ are the assertions hold after execution of S. Amtoft and Banerjee [26] proposed Hoare-logic-based information flow analysis of imperative programs. In particular, the proposed approach first computes abstraction of prelude (a state transfer function) before and after the execution of a command and then formalizes Hoare-like logic for checking the noninterference property in the form of independences (a set of pair of the form $[x^{\bowtie}w]$). Authors in Ref. [27] define a set of proof rules for secure information flow based on axiomatic approach.

3. Background

3.1 Blockchain technology [1]

Most often, the terms Blockchain Technology and Distributed Ledger Technology (DLT) are synonymously used. However, it is critical to understand that a blockchain is a cryptographically connected, sequential chain of blocks. In contrast, DLT is a decentralized ledger shared among multiple nodes/participants who agree on the standard protocol for sharing, validating, and updating the ledger data. Unlike the traditional database system, DLT functions decentralized way and fully controls the ledger's data to users to promote transparency. All blockchains are DLTs.

A blockchain is a peer-to-peer distributed ledger, forged by consensus, combined with a system for smart contracts and other assistive technologies. Blockchain, which bundles transactions into blocks that are chained together, and then broadcasts them to the nodes in the network, is probably the best-known type of distributed ledger technology. It is difficult to disagree with immutability feature when deploying blockchain-based solutions for a variety of socioeconomic systems that are currently centralized. The blockchain's immutability feature makes it useful for various applications such as bookkeeping, access control, and asset management. Once a transaction is recorded on the blockchain, its modification is nearly impossible.

Much of the early interest in blockchain technology has been around its application in financial transactions [33]. However, blockchain today is applied in multiple areas such as supply chain, traffic management, health care, insurance, etc., in addition to financial transactions [34]. It is also used in tax collection, property deed transfers, social benefits distribution, and even voting procedures [35, 36]. Some recent works state that individuals can use this technology to hold and better control personal information, and then selectively share pieces of those records when needed [37, 38]. In addition, blockchain technology provides a better track of intellectual property rights and ownership for art, commodities, music, and film [39–41].

3.2 Solidity smart contracts [4, 5]

Smart contracts have achieved a lot of attention with the rise of blockchain technologies such as Ethereum. All smart contracts are written to implement business logic that handles cryptocurrencies or tokens denoting real-world assets. Smart contracts executing in a blockchain architecture must be deterministic, otherwise, consensus may never be reached.

To address this, many blockchain platforms require domain-specific languages such as Solidity to write smart contracts so that nondeterminism can be eliminated. Ethereum blockchain has an execution engine called the Ethereum virtual machine (EVM), which enables Ethereum-based blockchains to expand their capabilities by executing code on a distributed network in a decentralized way. As part of the Ethereum network, the EVM executes the code. Solidity is a programming language developed to write smart contracts exclusively for the Ethereum blockchain network. However, EVM is only intelligent enough to understand low-level instructions known as *byte code*, and the compiler is required to convert the Solidity's complex code to an EVM-compliant *byte code*. The Solidity programming language comes with a compiler called the Solidity compiler (SOLC) that does this translation.

Solidity is a statically typed programming language, which means that variable types are explicitly defined and thus decided during compilation. Solidity's syntax is remarkably similar to scripting languages such as JavaScript, and it is highly influenced by C++ and Python programming languages. To demonstrate the structure of Solidity Smart contracts, consider a simple *helloWorld* example depicted in Code Snippet 3.

Code Snippet 3

Hello world example

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity 0.8.0;
import './other_solidity_source_file.sol';
//Single line comment
/*
Multi-line comments:
Spans multiple lines
*/
contract HelloWorld {
    function getMessage() public pure returns (string memory) {
        return "Hello_From_Solidity";
    }
}
```

Most of the Solidity file is made up of the following components:

License specifier: This indicates the license under which the code is published.
Pragma directives: The pragma instruction defines the current version of the Solidity file compiler. One can specify the compiler version for the code using the pragma directive.
Import statement: The import keyword enables us to import Solidity files and access their code within the current Solidity file.
Comments: Like other programming languages, Solidity supports single- and multiline comments.
Contracts/libraries/interfaces: Along with pragma, imports, and comments, a single Solidity source file can define multiple contracts, libraries, and interfaces.

Typically, the definition of a contract is composed of various constructs such as state variables, enumeration definitions, structure definitions, event statements, error statements, modifier definitions, and function definitions. More information about Solidity smart contracts can be found in Ref. [4].

3.3 Language-based information flow security [10, 11]

Language-based information flow security analysis has a long history, going back to the seminal work of the Dennings in the 1970s [10]. Additional advances have been broadly outlined in survey by Sabelfeld and Myers [11].

The starting point in secure information flow analysis for confidentiality preservation is the classification of program variables into different security levels. The main significance is to classify variables as L, meaning *low* security, denoting public information; and other variables as H, meaning *high* security, representing private information. The security goal is to prevent information in H variables from being leaked improperly. Such leaks could take a variety of forms, of course, but certainly we need to prevent information in H variables from flowing to L variables. Since the purpose is to prevent flow of sensitive information from an H variable to a L one, the security lattice $low \sqsubseteq high$, ensure that sensitive information flows only upwards in the lattice.

On the other hand, in case of respecting integrity property, the security levels of the variables are classified as *tainted* and *untainted*. If we view some variables as containing possibly tainted information, then we may wish to prevent information from such variables flowing into untainted variables. This is modeled using a lattice with *untainted* $\sqsubseteq$ *tainted*.

The semantic of protecting secret information in the security literature is formalized as noninterference, and is described in terms of indistinguishable relation on program states. Two program states are indistinguishable for *low* if they agree on values of *low* variables. The noninterference property states that any two executions of a program with two initial states that are indistinguishable for *low* result in two final states that are indistinguishable for *low*. Goguen and Meseguer [42] defined noninterference property in terms of indistinguishable relation on program states as follows:

$$\sigma_1 \simeq_{low} \sigma_2 \Longrightarrow [[S]]\sigma_1 \simeq_{low} [[S]]\sigma_2 \tag{1}$$

where S is a program statement or program, σ_1 and σ_2 ($\sigma_1 \neq \sigma_2$) are two states of S before executing it, $[[S]]\sigma_1$ and $[[S]]\sigma_2$ are the states obtained after executing the program S on σ_1 and σ_2, respectively. $\sigma_1 \simeq_{low} \sigma_2$ implies that two program states are indistinguishable for *low* if they share same *low* input value. Eq. (1) clearly states that if two program states are indistinguishable for *low* then executing the program with different values of *high* variables is once again indistinguishable for *low*. This means that varying *high* input values during execution of a program gives same output. To exemplify this, let consider the code snippet $l := h$; `if` $(h == 10)$ `then` $l := 5$; `else` $l := 10$. According to Eq. (1), this code snippet is insecure because taking 10 and 15 as initial high values and 0 as initial low value, we get different final values for l (i.e., $[[l := h]](0, 10) = (10, 10) \not\simeq_{low} (15, 15) = [[l := h]](0, 15)$). However, the code snippet `if` $(l == 10)$ `then` $h := h + 5$; `else` $l := l + 10$; is a secure program as final values of l is only depend on the initial value of l.

The notion of noninterference can easily be extended to the case of program's integrity, where program's execution with different tainted data should not affect the values of the (untainted) variables involved in critical computational part of the program.

3.4 Data-flow analysis [43]

Data-flow analysis is a technique for investigating the inner workings of a computer program. The control-flow graph (CFG) of a program is used to determine how the information propagates within the program. Compilers typically make use of data-flow analysis information when optimizing a program. Data-flow analysis can provide a variety of useful details, including busy expressions, live variables, available expressions, and reaching definition.

Numerous categories of program analysis exist, including forward analysis, backward analysis, may analysis, and must analysis. A forwards analysis makes use of knowledge about each programming point's predecessors, whereas a backward analysis is based on the node's successors' information. A may analysis is used to explain information that could be true, whereas a must analysis is used to describe information that is certain to be true. Combinations of the above-mentioned program analysis techniques and their respective categories, represented in Table 1.

Example 1 Consider the program in Code Snippet 4 for reaching definition (RD) analysis. Its CFG is depicted in Fig. 1. Data-flow equations for RD analysis at a given program point p is computed using Eq. (2).

$$\mathrm{RD}(p) = \left\{ \bigcup_{p' \in predecessor(p)} \mathrm{RD}(p') \downarrow_{DEF(p)} \right\} \cup \{DEF(p)\} \tag{2}$$

where $DEF(p)$ represents variable definitions at program point p which are directly available following the program point p, whereas $\mathrm{RD}(p') \downarrow_{DEF(p)}$ represents the set of reaching definitions available at the program point p' excluding the definitions of the variable defined at program point p. Table 2 depicts the RD analysis of Code Snippet 4, where column 2 represents the data-flow equations and column 3 represents their fixed-point solution. Observe that no program point makes use of the

TABLE 1 Program analysis techniques.

	Forwards	Backwards
May	Reaching definition	Liveness
Must	Available expression	Very busy expression

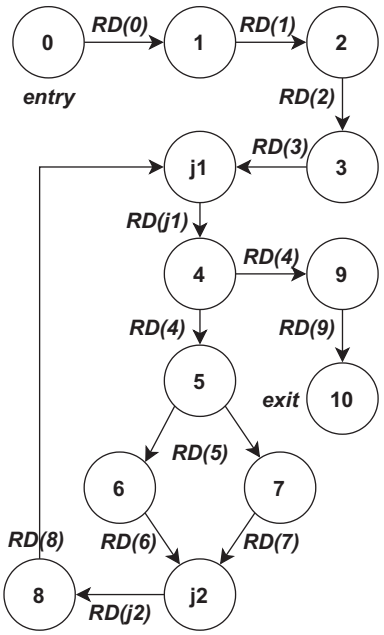

FIG. 1 CFG.

TABLE 2 Reaching definition analysis.

Prog. points	Data-flow equations	Least fixpoint solution
0	∅	∅
1	RD(0) ↓ x ⋃ {(x, 1)}	(x, 1)
2	RD(1)↓ y ⋃ {(y, 2)}	(x, 1), (y, 2)
3	RD(2)↓ x ⋃ {(x, 3)}	(y, 2), (x, 3)
j1	RD(3) ⋃ RD(8)	(y, 2), (x, 3), (z, 6), (z, 7), (x, 8)
4	RD(j1}	(y, 2), (x, 3), (z, 6), (z, 7), (x, 8)
5	RD(4)	(y, 2), (x, 3), (z, 6), (z, 7), (x, 8)
6	RD(5)↓ z ⋃ {(z, 6)}	(y, 2), (x, 3), (z, 6), (x, 8)
7	RD(5)↓ z ⋃ {(z, 7)}	(y, 2), (x, 3), (z, 7), (x, 8)
j2	RD(6) ⋃ RD(7)	(y, 2), (x, 3), (z, 6), (z, 7), (x, 8)
8	RD(j2)↓ x ⋃ {(x, 8)}	(y, 2), (z, 6), (z, 7), (x, 8)
9	RD(4)	(y, 2), (x, 3), (z, 6), (z, 7), (x, 8)

definition of variable *x* defined at program point 1, that is, (*x*, 1). As a result, the definition of variable *x* at program point 1 can be treated as a dead code and must be removed during optimization.

Code Snippet 4

Example

```
1 x = 25
2 y = 4
3 x = 10
4 while (x > 32){
```

```
5    if(y > x)
6        z = y;
7    else z = y * y;
8    x = z;
9 }
10 print(x, y, z)
```

4. 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 for our objective.

4.1 Solidity syntax

Solidity is a well-known smart contract language among Ethereum developers. Its design is inspired by JavaScript language. Unlike JavaScript, it is a statically typed language instrumented with many blockchain-specific features. Table 3 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 number 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`.

4.2 Concrete semantics

Let us now consider various semantics domains of Solidity variables and define the concrete semantics of the language in Table 3. Our semantics formalism is primarily motivated from Ref. [44].

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.

Environments and states

Ethereum, taken as a whole, can be viewed as a transaction-based state machine [5]. 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 here:

- The set of local environments is defined as LEnv : Var→MLoc.
- The set of local stores is defined as LStore : MLoc→Val.

TABLE 3 Abstract syntax of a subset of solidity.

Expressions				
	exp	∈	*E*	Arithmetic expressions
	exp	::=	*v*	INT or UINT value
		|	id	Identifier
		|	$\texttt{exp}_1 \oplus \texttt{exp}_2$	where $\oplus = \{+, -, *, /\}$
	bexp	∈	*B*	Boolean expressions
	bexp	::=	*true* | *false*	Truth/falsity
		|	$\texttt{exp}_1 \oslash \texttt{exp}_2$	where $\oslash = \{\geq, \leqslant, <, >, ==\}$
		|	$\texttt{bexp}_1 \otimes \texttt{bexp}_2$	where $\otimes = \{\vee, \wedge\}$
		|	¬bexp	Negation
Labeled commands				
	ℓ	∈	L	Labels
	sc	∈	SC	Smart contracts
	type	::=	*int* | *uint* | *bool*	
		|	*address* | *address payable*	Value types
	store	::=	*storage* | *memory*	Data location
	ldecl	::=	$^{\ell}$type store id(:=exp)?;	Local variable declaration
	assgn	::=	$^{\ell}$id:=exp;	Assignment
	stmt	::=	ldecl | assgn	Statement
		|	*if* $^{\ell}$bexp *then* $\texttt{stmt}_1$ *else* $\texttt{stmt}_2$ *endif*	
		|	*while* $^{\ell}$bexp *do* stmt *done*	
		|	$\texttt{stmt}_1\texttt{stmt}_2$	
	sdecl	::=	$^{\ell}$type id; | $\texttt{sdecl}_1\texttt{sdecl}_2$	State variable declaration
	par	::=	$^{\ell}$type store? id | $\texttt{par}_1$, $\texttt{par}_2$	Function parameter
	constr	::=	$^{\ell}$constructor(par){stmt}	Constructor
	func	::=	$^{\ell}$*function* id(par) (*returns*(par)){stmt}	Function definition
	sc	::=	*contract* id{sdecl constr? func*}	Smart contract

- The set of blockchain environments is defined as BEnv : Var→SLoc.
- The set of blockchain stores is defined as BStore : SLoc→Val.
- The set of states is defined as Σ : LEnv×LStore × BEnv × 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.

Semantics

We are now in a position to define the concrete semantics of various Solidity constructs below. We use the functions currLabel() and nextLabel() which, given a smart contract statement c, return the label of c and the label of the statement following c, respectively.

(1) Expressions

$$\mathcal{L}[[v]] = \{(\rho, v) \mid \rho \in \Sigma\}$$

$$\mathcal{L}[[\texttt{id}]] = \begin{cases} \{(\rho,v) \mid \rho \in \Sigma,\ ls(le(\text{id})) = v\} & \text{if } le(\text{id}) \in \text{LEnv} \\ \{(\rho,v) \mid \rho \in \Sigma,\ bs(be(\text{id})) = v\} & \text{if } be(\text{id}) \in \text{BEnv} \end{cases}$$

$$\mathcal{L}[[\texttt{exp}_1 \oplus \texttt{exp}_2]] = \{(\rho, v_1 \oplus v_2) \mid (\rho, v_1) \in \mathcal{L}[[\texttt{exp}_1]], (\rho, v_2) \in \mathcal{L}[[\texttt{exp}_2]]\}$$
$$\mathcal{L}[[true]] = \{(\rho, true) \mid \rho \in \Sigma\}$$
$$\mathcal{L}[[false]] = \{(\rho, false) \mid \rho \in \Sigma\}$$
$$\mathcal{L}[[\texttt{exp}_1 \oslash \texttt{exp}_2]] = \{(\rho, v_1 \oslash v_2) \mid (\rho, v_1) \in \mathcal{L}[[\texttt{exp}_1]], (\rho, v_2) \in \mathcal{L}[[\texttt{exp}_2]]\}$$
$$\mathcal{L}[[\texttt{bexp}_1 \otimes \texttt{bexp}_2]] = \{(\rho, b_1 \otimes b_2) \mid (\rho, b_1) \in \mathcal{L}[[\texttt{bexp}_1]], (\rho, b_2) \in \mathcal{L}[[\texttt{bexp}_2]]\}$$
$$\mathcal{L}[[\neg \texttt{bexp}]] = \{(\rho, \neg b) \mid (\rho, b) \in \mathcal{L}[[\texttt{bexp}]]\}$$

(2) Local variable declaration

$$\mathcal{L}[[^{\ell}\texttt{type store id}]] \triangleq \begin{cases} \{(\ell,\rho) \to (\ell',\rho') \mid \rho \in \Sigma\} & \texttt{if store is } \textit{memory} \\ \{(\ell,\rho) \to (\ell',\rho'') \mid \rho \in \Sigma\} & \texttt{if store is } \textit{storage} \end{cases}$$

where $\ell' \in \texttt{nextLabel}(^{\ell}\texttt{type store id})$, $\rho = (le, ls, be, bs)$, $\rho' = (le[\texttt{id} \to l'], ls[l' \to v'_d], be, bs)$ with l' as a fresh local memory location and v'_d as the default memory value, and $\rho'' = (le, ls, be[\texttt{id} \to l''], bs[l'' \to v''_d])$ with l'' as a fresh storage location and v''_d as the default storage value.

$$\mathcal{L}[[^{\ell}\texttt{type store id} := \texttt{exp}]] \triangleq \begin{cases} \{(\ell,\rho) \to (\ell',\rho') \mid \rho \in \Sigma\} & \texttt{if store is } \textit{memory} \\ \{(\ell,\rho) \to (\ell',\rho'') \mid \rho \in \Sigma\} & \texttt{if store is } \textit{storage} \end{cases}$$

where $\ell' \in \texttt{nextLabel}(^{\ell}\texttt{type store id:=exp})$, $\rho = (le, ls, be, bs)$, $(\rho, v) \in \mathcal{L}[[\texttt{exp}]]$, $\rho' = (le[\texttt{id} \to l'], ls[l' \to v], be, bs)$ with l' as a fresh local memory location, and $\rho'' = (le, ls, be[\texttt{id} \to l''], bs[l'' \to v])$ with l'' as a fresh storage location.

(3) Assignment statement

$$\mathcal{L}[[^{\ell}\texttt{id} := \texttt{exp}]] \triangleq \begin{cases} \{(\ell,\rho) \to (\ell',\rho') \mid \rho \in \Sigma\} & \texttt{if le(id)} \in \texttt{LEnv} \\ \{(\ell,\rho) \to (\ell',\rho'') \mid \rho \in \Sigma\} & \texttt{if be(id)} \in \texttt{BEnv} \end{cases}$$

where $\ell' \in \texttt{nextLabel}(^{\ell}\texttt{id:=exp})$, $\rho = (le, ls, be, bs)$, $(\rho, v) \in \mathcal{L}[[\texttt{exp}]]$, $\rho' = (le, ls[le(\texttt{id}) \to v], be, bs)$, $\rho'' = (le, ls, be, bs[be(\texttt{id}) \to v])$.

(4) Sequence

$$\mathcal{L}[[\texttt{stmt}_1\ \texttt{stmt}_2]] \triangleq \{(\ell,\rho) \to (\ell',\rho') \mid \rho \in \Sigma, (\ell,\rho) \to (\ell'',\rho'') \in \mathcal{L}[[\texttt{stmt}_1]], (\ell'',\rho'') \to (\ell',\rho') \in \mathcal{L}[[\texttt{stmt}_2]]\}$$

where $\ell \in \texttt{currLabel}(\texttt{stmt}_1)$ and $\ell' \in \texttt{nextLabel}(\texttt{stmt}_2)$.

(5) Conditional

$$\begin{aligned} &\mathcal{L}[[\textit{if } {}^{\ell}\texttt{bexp } \textit{then } \texttt{stmt}_1 \textit{ else } \texttt{stmt}_2 \textit{ endif}]] \\ &\triangleq \{(\ell,\rho) \to (\ell',\rho') \mid \rho \in \Sigma,\ (\rho, true) \in \mathcal{L}[[^{\ell}\texttt{bexp}]],\ ((\texttt{currLabel}(\texttt{stmt}_1), \rho) \to (\ell',\rho')) \in \mathcal{L}[[\texttt{stmt}_1]]\} \\ &\cup \{(\ell,\rho) \to (\ell',\rho'') \mid \rho \in \Sigma,\ (\rho, false) \in \mathcal{L}[[^{\ell}\texttt{bexp}]],\ ((\texttt{currLabel}(\texttt{stmt}_2), \rho) \to (\ell',\rho'')) \in \mathcal{L}[[\texttt{stmt}_2]]\} \end{aligned}$$

where $\ell' \in \texttt{nextLabel}(\textit{if } {}^{\ell}\texttt{bexp } \textit{then } \texttt{stmt}_1 \textit{else } \texttt{stmt}_2 \textit{endif})$.

(6) Iteration

$$\mathcal{L}[[\textit{while}\ ^{\ell}\texttt{bexp}\ \textit{do}\ \texttt{stmt}\ \textit{done}]] = \texttt{lfp}^{\subseteq}_{\emptyset}(F)$$

where

$$\begin{aligned} F(\psi) = \ & \{(\ell,\rho) \to (\ell,\rho') \mid \rho \in \Sigma,\ (\rho,\textit{true}) \in \mathcal{L}[[^{\ell}\texttt{bexp}]],\ ((\texttt{currLabel}(\texttt{stmt}),\rho) \to (\ell,\rho')) \in \mathcal{L}[[\texttt{stmt}]]\} \\ & \cup \{(\ell,\rho) \to (\ell',\rho) \mid \rho \in \Sigma,\ (\rho,\textit{false}) \in \mathcal{L}[[^{\ell}\texttt{bexp}]]\} \\ & \cup \{(\ell,\rho) \to (\ell,\rho'') \mid \rho \in \Sigma,\ (\rho,\textit{true}) \in \mathcal{L}[[^{\ell}\texttt{bexp}]], ((\ell,\rho') \to (\ell,\rho'')) \in \psi, \\ & \qquad ((\texttt{currLabel}(\texttt{stmt}),\rho) \to (\ell,\rho')) \in \mathcal{L}[[\texttt{stmt}]]\} \end{aligned}$$

where $\ell' \in \texttt{nextLabel}(\textit{while}\ ^{\ell}\texttt{bexp}\ \textit{do}\ \texttt{stmt}\ \textit{done})$ and $\texttt{nextLabel}(\texttt{stmt}) = \texttt{currLabel}\ (\textit{while}^{\ell}\texttt{bexp}\ \textit{do}\ \texttt{stmt}\ \textit{done}) = \ell$.

(7) State variable declaration

$$\mathcal{L}[[^{\ell}\texttt{type id}]] \triangleq \{(\ell,\rho) \to (\ell',\rho') \mid \rho \in \Sigma\}$$

where $\ell' \in \texttt{nextLabel}(^{\ell}\texttt{type id})$, $\rho = (le, ls, be, bs)$, $\rho' = (le, ls, be[\texttt{id} \to l], bs[l \to v_d])$ with l as a fresh storage location and v_d as the default storage value.

(8) Constructor

$$\mathcal{L}[[^{\ell}\texttt{constructor(par)\{stmt\}}]] \triangleq \{((\ell,\rho), v_{in}) \to (\ell',\rho') \mid \rho \in \Sigma\}$$

where $\ell' \in \texttt{nextLabel}(^{\ell}\texttt{constructor(par)\{stmt\}})$, $\rho = (le,\ ls,\ be,\ bs)$, $\rho'' = (le[\texttt{Par} \to l],\ ls[l \to v_{in}],\ be,\ bs)$, $((\texttt{currLabel}(\texttt{stmt}),\rho'') \to (\ell',\rho')) \in \mathcal{L}[[\texttt{stmt}]]$.

(9) Function

Let $\texttt{Par}^{\text{memory}}_{\text{in}}$ and $\texttt{Par}^{\text{storage}}_{\text{in}}$ 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 $\texttt{Par}^{\text{memory}}_{\text{in}}$ and $\texttt{Par}^{\text{memory}}_{\text{in}}$, respectively, where $v_{in} = v^{memory}_{in} \cup v^{storage}_{in}$. The semantics of the function is defined as follows:

$$\mathcal{L}[[^{\ell}\textit{function}\ \texttt{id}(\texttt{par}_{\texttt{in}})\ (\textit{returns}(\texttt{par}_{\texttt{out}}))\{\texttt{stmt}\}]] \triangleq \{((\ell,\rho), v_{in}) \to ((\ell',\rho'), v_{out}) \mid \rho \in \Sigma\}$$

where

- $\rho = (le, ls, be, bs)$,
- $\rho'' = (le[\texttt{Par}^{\text{memory}}_{\text{in}} \to l], ls[l \to v^{memory}_{in}], be[\texttt{Par}^{\texttt{storage}}_{\text{in}} \to l'], bs[l' \to v^{storage}_{in}])$,
- $((\texttt{currLabel}(\texttt{stmt}),\rho'') \to (\ell',\rho')) \in \mathcal{L}[[\texttt{stmt}]]$, where $\ell' \in \texttt{nextLabel}(^{\ell}\textit{function}\ \texttt{id(par}_{\texttt{in}}\texttt{)}\ (\textit{returns}\ \texttt{(par}_{\texttt{out}}\texttt{)})\texttt{\{stmt\}})$,
- $(\rho', v_{out}) \in \mathcal{L}[[(\texttt{par}_{\texttt{out}})]]$.

(10) Smart contract

$$\mathcal{L}[[\textit{contract}\ \texttt{id\{sdecl constr? func}^{*}\texttt{\}}]] \triangleq \{((\ell,\rho) \to (\ell',\rho')) \mid \rho \in \Sigma\}$$

where $\ell \in \texttt{currLabel}(\texttt{sdecl})$, $\ell' \in \texttt{nextLabel}(\texttt{func}\star)$, $((\ell,\rho) \to (\ell'',\rho'')) \in \mathcal{L}[[\texttt{sdecl}]]$ and $((\ell'',\rho'') \to (\ell''',\rho''')) \in \mathcal{L}[[\texttt{constr?}]]$ and $((\ell''',\rho''') \to (\ell',\rho')) \in \mathcal{L}[[\texttt{func}^{*}]]$.

5. Formal dependency analysis of solidity smart contracts

In order to analyze Solidity smart contracts for identifying insecure information flow, in this section, we define an abstract semantics of Solidity in the domain of positive propositional formulae Pos. The primary goal of this domain is to encode variables' dependences in the form of logical formula whose satisfiability based on the truth values as per variables' security levels determines possible security property violation.

5.1 Abstract domain: Pos [45–47]

Dependencies among program's variables and statements are the key players in the analysis of information flow along program's control paths. As cited in Ref. [45], many static analyses use Boolean functions to express such dependencies. Following the similar idea of dependency-based analysis, in this section, we consider Boolean function of the form $x_p \rightarrow y_q$ to represent the dependency such as "whenever x has property p, y has property q."

To this aim, we use the abstract domain Pos which receives much attention in case of abstract interpretation of logic programs [45–47]. This domain is most commonly applied to the analysis of groundness dependencies for logic programs.

Let $\mathtt{PropVar} = \{\overline{x_i} \mid i \in \mathbb{N}\}$ be a countably infinite set of propositional variables and $\mathtt{PropCon} = \{\wedge, \vee, \rightarrow, \neg\}$ be the set of logical connectives. We denote by PF the set of all propositional formula defined over PropVar and PropCon.

Let $\overline{B} \triangleq \mathtt{PropVar} \rightarrow \{true, false\}$ be the set of *truth-assignment* functions that assign truth value (either *true* or *false*) to each propositional variable in PropVar. Given a formula $f \in \mathtt{PF}$ and a truth-assignment $r \in \overline{B}$, we mean by $r \models f$ that f is satisfied under the truth-assignment r.

Given two propositional formulae f_1 and f_2, we use the shorthand notation $f_1 \models f_2$ to denote "$r \models f_1$ implies $r \models f_2$" for any $r \in \overline{B}$ [13]. The domain PF is ordered by $f_1 \preccurlyeq f_2$ iff $f_1 \models f_2$. Two formulae f_1 and f_2 are logically equivalent, denoted $f_1 \equiv f_2$, iff $f_1 \preccurlyeq f_2$ and $f_2 \preccurlyeq f_1$.

The unit assignment u is defined as follows: $\mathtt{u}(\overline{x_i}) = true$ for all $\overline{x_i} \in \mathtt{PropVar}$. The set of positive formulae is defined by $\mathtt{Pos} = \{f \in \mathtt{PF} \mid \mathtt{u} \models f\}$.

5.2 Abstract semantics

As already mentioned earlier, dependencies among program's variables determine the flow of information along it. To capture this, we define the abstract semantics of Solidity language in the domain Pos of logic formulae representing dependences between variables. In particular, the logic formulae are of the form [13, 48]: $\wedge_{0 \le i \le n\ 0 \le j \le m}\ \overline{x_i} \rightarrow \overline{y_j}$, where $\overline{x_i}, \overline{y_j} \in$ PropVar. This indicates that the values of the program variable y_j corresponding to the propositional variable $\overline{y_j}$ depend on the values of the program variable x_i corresponding to the propositional variable $\overline{x_i}$.

Example 2 Consider the following code snippet:

```
1 y := 7;
2 z := 5;
3 x := z;
4 if(y >= 5)
5     z := x-1;
```

The logic formula $\overline{z} \rightarrow \overline{x}$ at program point 3 represents an explicit flow from the program variable z to the program variable x due to the assignment statement, whereas the logic formula $(\overline{y} \rightarrow \overline{z}) \wedge (\overline{x} \rightarrow \overline{z})$ at program point 5 indicates an implicit flow due to the conditional statement.

The security analysis (confidentiality or integrity violation) is then carried out by evaluating the formulae based on truth value assignment according to the variables' security levels.

Formally, we define the set of abstract states as follows: $\overline{\Sigma} \triangleq L \times \mathtt{Pos}$. Therefore, an abstract state $\overline{\rho} \in \overline{\Sigma}$ is a pair $\langle \ell, \psi \rangle$, where $\psi \in \mathtt{Pos}$ represents the set of logic formulae indicating variables dependences up to the program point $\ell \in L$.

Let us now define the abstract semantics of various Solidity constructs below. In the formalism, we use the simplification operator $\ominus$ defined in Ref. [48]: $\psi_1 \ominus \psi_2 = \wedge(\mathtt{SF}(\psi_1) \setminus \mathtt{SF}(\psi_2))$, where $\mathtt{SF}(\psi)$ denotes the set of subformulas in ψ. The functions Var() and DefVar() return the *used* and *defined* variables, respectively.

(1) Assignment statement

$$\bar{\mathcal{L}}[[^{\ell}\mathrm{id} := \mathrm{exp}]] \triangleq \{(\ell, \psi) \rightarrow (\ell', \psi_o) \mid \psi \in \overline{\Sigma}\}$$

where $\ell' \in \mathtt{nextLabel}(^{\ell}\mathtt{id:=exp})$, $\psi_o = \wedge\{\overline{\mathtt{id}_i} \rightarrow \overline{\mathtt{id}} \mid \mathtt{id}_i \in \mathtt{Var(exp)} \wedge \overline{\mathtt{id}_i} \neq \overline{\mathtt{id}}\} \wedge \{\overline{\mathtt{id}_j} \rightarrow \overline{\mathtt{id}_k} \mid \overline{\mathtt{id}_j} \rightarrow \overline{\mathtt{id}_r}, \overline{\mathtt{id}_r} \rightarrow \overline{\mathtt{id}_k} \in \psi\} \wedge (\psi \ominus \wedge\{\overline{\mathtt{id}_l} \rightarrow \overline{\mathtt{id}} \mid \overline{\mathtt{id}_l} \in \mathtt{PropVar} \wedge \mathtt{id}_l \notin \mathtt{Var(exp)}\})$

(2) Sequence

$$\overline{\mathcal{L}}[[\texttt{stmt}_1\ \texttt{stmt}_2]] \triangleq \overline{\mathcal{L}}[[\texttt{stmt}_1]] \cup \overline{\mathcal{L}}[[\texttt{stmt}_2]]$$

(3) Conditional

$$\begin{aligned}&\overline{\mathcal{L}}[[\textit{if}\ {}^{\ell}\texttt{bexp}\ \textit{then}\ \texttt{stmt}_1\ \textit{else}\ \texttt{stmt}_2\ \textit{endif}]]\\ &\quad\triangleq \{(\ell,\psi)\rightarrow(\texttt{currLabel}(\texttt{stmt}_1),\psi)\}\ \cup\ \{(\ell,\psi)\rightarrow(\texttt{currLabel}(\texttt{stmt}_2),\psi)\}\\ &\qquad\cup\ \overline{\mathcal{L}}[[\texttt{stmt}_1]]\cup\overline{\mathcal{L}}[[\texttt{stmt}_2]]\\ &\qquad\cup\ \{(\ell,\psi)\rightarrow(\ell',\psi_1)\}\ \cup\ \{(\ell,\psi)\rightarrow(\ell',\psi_2)\}\end{aligned}$$

where

- $\ell' \in \texttt{nextLabel}(\textit{if}\ {}^{\ell}\texttt{bexp}\ \textit{then}\ \texttt{stmt}_1 \textit{else}\ \texttt{stmt}_2 \textit{endif})$
- $\psi_1 = \psi \wedge \wedge\{\overline{\texttt{id}_i} \rightarrow \overline{\texttt{id}_j} | \texttt{id}_i \in \texttt{Var}(\texttt{bexp}) \wedge \texttt{id}_j \in \texttt{DefVar}(\texttt{stmt}_1) \wedge \overline{\texttt{id}_i} \neq \overline{\texttt{id}_j}\}$
- $\psi_2 = \psi \wedge \wedge\{\overline{\texttt{id}_i} \rightarrow \overline{\texttt{id}_j} | \texttt{id}_i \in \texttt{Var}(\texttt{bexp}) \wedge \texttt{id}_j \in \texttt{DefVar}(\texttt{stmt}_2) \wedge \overline{\texttt{id}_i} \neq \overline{\texttt{id}_j}\}$

(4) Iteration

$$\begin{aligned}&\overline{\mathcal{L}}[[\textit{while}\ {}^{\ell}\texttt{bexp}\ \textit{do}\ \texttt{stmt}\ \textit{done}]]\\ &\quad\triangleq\{(\ell,\psi)\rightarrow(\texttt{currLabel}(\texttt{stmt}),\psi)\}\ \cup\ \overline{\mathcal{L}}[[\texttt{stmt}]]\\ &\qquad\cup\ \{(\ell,\psi)\rightarrow(\ell',\psi_1)\}\end{aligned}$$

where

- $\ell' \in \texttt{nextLabel}(\textit{while}\ {}^{\ell}\texttt{bexp}\ \textit{do}\ \texttt{stmt}\ \textit{done})$
- $\psi_1 = \psi \wedge \wedge\{\overline{\texttt{id}_i} \rightarrow \overline{\texttt{id}_j} | \texttt{id}_i \in \texttt{Var}(\texttt{bexp}) \wedge \texttt{id}_j \in \texttt{DefVar}(\texttt{stmt}) \wedge \overline{\texttt{id}_i} \neq \overline{\texttt{id}_j}\}$

(5) Constructor

$$\begin{aligned}&\overline{\mathcal{L}}[[{}^{\ell}\texttt{constructor(par)\{stmt\}}]]\\ &\quad\triangleq\{(\ell,\psi)\rightarrow(\ell',\psi_o) \mid ((\texttt{in(stmt)},\psi)\rightarrow(\ell',\psi_o)) \in \overline{\mathcal{L}}[[\texttt{stmt}]]\}\end{aligned}$$

(6) Function

$$\begin{aligned}&\overline{\mathcal{L}}[[{}^{\ell}\textit{function}\ \texttt{id}(\texttt{par}_{\texttt{in}})\ (\textit{returns}(\texttt{par}_{\texttt{out}}))\{\texttt{stmt}\}]]\\ &\quad\triangleq\{(\ell,\psi)\rightarrow(\ell',\psi_o) \mid ((\texttt{in(stmt)},\psi)\rightarrow(\ell',\psi_o)) \in \overline{\mathcal{L}}[[\texttt{stmt}]]\}\end{aligned}$$

(7) Smart contract

$$\begin{aligned}&\overline{\mathcal{L}}[[\textit{contract}\ \texttt{id\{sdecl constr? func}^*\texttt{\}}]]\\ &\quad\triangleq\{((\ell\ \texttt{in(sdecl)},\psi)\rightarrow(\ell',\psi)) \in \overline{\mathcal{L}}[[\texttt{sdecl}]]\}\quad \cup\{((\ell',\psi)\rightarrow(\ell'',\psi_1)) \in \overline{\mathcal{L}}[[\texttt{constr?}]]\}\\ &\qquad\cup\{((\ell'',\psi_1)\rightarrow(\ell',\psi_o)) \in \overline{\mathcal{L}}[[\texttt{func}^*]]\}\end{aligned}$$

6. Confidentiality and integrity properties verification

Given a smart contract sc, let F_c : Var → {*Low*, *High*, *Unknown*} be a function that assigns to each program variable in sc a security class, either public (*Low*) or private (*High*) or unknown (*Unknown*). Once the abstract semantics of the smart contract sc is computed, the security class of a variable id with *unknown* level at program point ℓ is upgraded to either *Low* or *High* according to the following function [32]:

$$\text{Upgrade}(\texttt{id}) = Z \text{ if } \exists\, (\overline{\texttt{id}_i} \rightarrow \overline{\texttt{id}}) \in \psi.\ F_c(\texttt{id}_i) = Z \wedge F_c(\texttt{id}_i) \neq \mathit{Unknown} \wedge F_c(\texttt{id}) = \mathit{Unknown} \tag{3}$$

where ψ is the set the logical formulae up to the program point ℓ.

To verify the confidentiality property, we use the following truth-assignment function $\overline{T}$:

$$\overline{T}(\overline{\text{id}}) = \begin{cases} \mathit{True} & \text{if } F_c(\text{id}) = \mathit{High} \\ \mathit{False} & \text{if } F_c(\text{id}) = \mathit{Low} \end{cases}$$

where $\overline{\texttt{id}} \in \texttt{PropVar}$ corresponding to smart contract's variables $\texttt{id} \in \texttt{Var}$. A smart contract `sc` respects the confidentiality property if there is no information flow from *High* to *Low* variables. Given an abstract state (ℓ, ψ), if $\overline{T}$ does not satisfy a propositional formula in ψ (that is, $\mathit{True} \rightarrow \mathit{False}$), this indicates that there is a flow from *High* to *Low* variables and the analysis reports a possible violation of confidentiality property.

In case of Integrity property, we consider the function $F_i : \texttt{Var} \rightarrow \{\mathit{Taint}, \mathit{Untaint}, \mathit{Unknown}\}$ which assigns to each variable in the smart contract `sc` a security class, either Tainted (*Taint*) or Untainted (*Untaint*) or Unknown (*Unknown*). We say that the smart contract `sc` respects the integrity property, if and only if there is no information flow from tainted to untainted variables. To this aim, we define the corresponding truth-assignment function $\overline{T}$ as follows:

$$\overline{T}(\overline{\text{id}}) = \begin{cases} \mathit{True} & \text{if } F_i(\text{id}) = \mathit{Taint} \\ \mathit{False} & \text{if } F_i(\text{id}) = \mathit{Untaint} \end{cases}$$

Example 3 Let us illustrate this using Code Snippet 5 to understand better how the proposed information flow analysis works.

Code Snippet 5

Lottery example

```
1 address public winner;
2 function Lottery() public {
3     uint salt = block.timestamp;
4     uint hash = uint(blockhash(salt%1500));
5     uint random = uint(hash) % 2510;
6     if( winner == address(0x0) && random == 32 ) {
7         winner = msg.sender;
8     }
9 }
```

Initially, the variable *block.timestamp* and *msg.sender* are considered as *Taint* as they can be influenced by miners or other users, whereas the variable *winner* is *Untaint* as it is involved in the critical decision-making process. We assume that the security levels of the rest of the variables are *Unknown*.

Table 4 represents the abstract states at each program point, which consists of a set of propositional formulae from `Pos`, uncovering the relationships between variables. After applying the `Upgrade` function, the security levels of all the variables with *Unknown* level are upgraded to *Taint* according to the dependency observed at each program point. Finally, the truth-assignment function $\overline{T}$ maps the propositional variables as follows: $\{\overline{\mathit{winner}} \mapsto \mathit{False}\} \cup \{\overline{\texttt{id}} \mapsto \mathit{True} \mid \texttt{id} \in \texttt{Var}(\text{Lottery}) \setminus \{\overline{\mathit{block.timestamp}}\}$. Observe that the smart contract above code violates the integrity property as there is a logic formula $\overline{\mathit{random}} \rightarrow \overline{\mathit{winner}}$ which yields False after the following truth assignment: $\overline{T}(\overline{\mathit{random}}) = \mathit{True}$ and $\overline{T}(\overline{\mathit{winner}}) = \mathit{False}$.

TABLE 4 Results of the analysis by Pos domain.

Label	Propositional formula
3	$\overline{\mathit{block.timestamp}} \rightarrow \overline{\mathit{salt}}$
4	$\overline{\mathit{salt}} \rightarrow \overline{\mathit{hash}} \wedge \overline{\mathit{block.timestamp}} \rightarrow \overline{\mathit{hash}}$
5	$\overline{\mathit{hash}} \rightarrow \overline{\mathit{random}} \wedge \overline{\mathit{salt}} \rightarrow \overline{\mathit{random}} \wedge \overline{\mathit{block.timestamp}} \rightarrow \overline{\mathit{random}}$
7	$\overline{\mathit{msg.sender}} \rightarrow \overline{\mathit{winner}} \wedge \overline{\mathit{random}} \rightarrow \overline{\mathit{winner}} \wedge \overline{\mathit{hash}} \rightarrow \overline{\mathit{random}} \wedge \overline{\mathit{salt}} \rightarrow \overline{\mathit{random}} \wedge \overline{\mathit{block.timestamp}} \rightarrow \overline{\mathit{random}}$

7. Refining analysis by combining numerical abstract domains

In this section, we consider a reduced product which combines the information flow analysis results obtained in the domain of Pos with a numerical abstract domain, aiming to filter out false dependences which may arise when we consider the semantics of Solidity smart contracts. Interestingly, this allows to tune the level of abstractions as a way to trade-off the precision and efficiency of the analysis. Let us recall a brief description of various numerical abstract domains from Ref. [16] and the definition of the reduced product operation for dependency refinement from Refs. [13, 49].

7.1 Relational and nonrelational abstract domains

As a number of numerical abstract domains, nonrelational and relational, exist in the literature [50–55], let us briefly illustrate few of the widely used domains here.

We call an abstract domain nonrelational if the domain does not preserve any relation among the program variables [50–52]. In contrast, relational abstract domains maintains relations among the program variables [53, 55], yielding a more precise analysis results compared to that of nonrelational abstract domains. Few popular nonrelational abstract domains include "sign" domain for sign property analysis, "parity" domain for parity property analysis, "interval" domain for division-by-zero or overflows [50], whereas the domains of polyhedra, octagons, difference-bound matrices (DBM), etc. [53–56], are few examples of widely used relational abstract domains. In Fig. 2, a set of points (in • form on xy-plane) are abstracted by both nonrelational ("sign" and "interval") and relational abstract domains ("octagon" and "polyhedra").

Intervals [50, 52]

Let $X \subseteq \mathbb{Z}$ be a set of numerical values. This is approximated by $[l, h]$ where $l = \texttt{minimum}(X)$, $h = \texttt{maximum}(X)$, and $l \leqslant h$. In case the upper and lower bound are not known, this is approximated by $[-\infty, +\infty]$. Let $\mathrm{L} = \langle \wp(\mathbb{Z}), \subseteq, \varnothing, \mathbb{Z}, \cap, \cup \rangle$ be a concrete lattice of $\wp(\mathbb{Z})$. Let $\mathbb{I} = \{[l, h] | l \in \mathbb{Z} \cup \{-\infty\}, h \in \mathbb{Z} \cup \{+\infty\}, l \leq h\} \cup \bot$ be the abstract domain of intervals which form an abstract lattice $\mathrm{A} = \langle \mathbb{I}, \sqsubseteq, \bot, [-\infty, +\infty], \sqcap, \sqcap \rangle$, where

- $[l_1, h_1] \sqsubseteq [l_2, h_2] \Leftrightarrow l_2 \leqslant l_1 \wedge h_2 \geq h_1$
- $[l_1, h_1] \sqcap [l_2, h_2] = [\texttt{maximum}(l_1 l_2), \texttt{minimum}(h_1 h_2)]$
- $[l_1, h_1] \sqcap [l_2, h_2] = [\texttt{minimum}(l_1, l_2), \texttt{maximum}(h_1 h_2)]$

The Galois connection $\langle \mathrm{L}, \alpha_{\mathbb{I}}, \gamma_{\mathbb{I}}, \mathrm{A} \rangle$ between L and A is formalized as follows:

$$\alpha_{\mathbb{I}}(X) = \begin{cases} \bot & \text{if } X = \varnothing \\ [l,\ h] & \text{if } \texttt{minimum}(X) = l \wedge \texttt{maximum}(X) = h \\ [-\infty,\ h] & \text{if } \nexists \texttt{minimum}(X) \wedge \texttt{maximum}(X) = h \\ [l,\ +\infty] & \text{if } \texttt{minimum}(X) = l \wedge \nexists \texttt{maximum}(X) \\ [+\infty,\ -\infty] & \text{if } \nexists \texttt{minimum}(X) \wedge \nexists \texttt{maximum}(X) \end{cases}$$

$$\gamma_{\mathbb{I}}(a) = \begin{cases} \varnothing & \textit{if } a = \bot \\ \{k \in \mathbb{Z} |\ l \leq k \leq h\} & \text{if } a = [l,\ h] \\ \{k \in \mathbb{Z} |\ k \leq h\} & \text{if } a = [-\infty,\ h] \\ \{k \in \mathbb{Z} |\ l \leq k\} & \text{if } a = [l,\ +\infty] \\ \mathbb{Z} & \textit{if } a = [+\infty,\ -\infty] \end{cases}$$

where $X \in \wp(\mathbb{Z})$ and $a \in \mathbb{I}$. This is depicted pictorially in Fig. 3.

Octagons [55]

Given a set of points in n-dimensional space, it can be approximated by octagonal abstract domain which encodes binary constraints between program variables in the form of $k_i x_i + k_j x_j \leqslant k$, where $k_i, k_j \in [-1, 0, 1]$ are coefficients, x_i, x_j are program variables, and $k \in \mathbb{R}$. The set of points satisfying the conjunction of such constraints forms an octagon. Difference bound matrix (DBM) [54] is a most suitable representation of these octagonal constraints. Given a set of octagonal constraints over n program variables, we define DBM m of size $n \times n$ as follows:

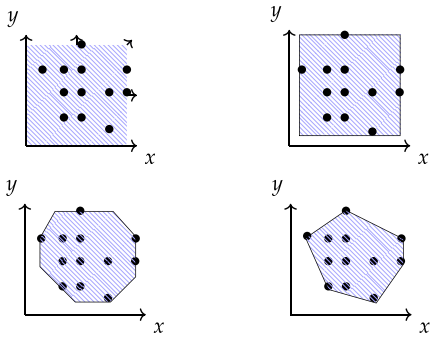

FIG. 2 Abstractions of a set of points by the domains of "sign" (*upper-left*), "interval" (*upper-right*), "octagons" (*lower-left*), and "polyhedra" (*lower-right*).

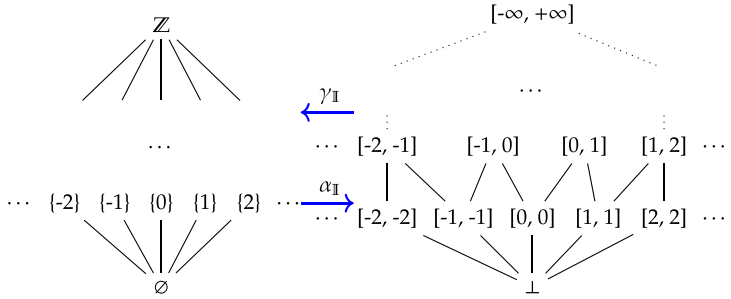

FIG. 3 Galois connection between L and A.

$$\mathtt{m}_{ij} \triangleq \begin{cases} k \text{ if } (x_j - x_i \leqslant k) \text{ where } x_i, x_j \in \text{Var and } k \in \mathbb{R} \\ \infty \qquad \text{otherwise} \end{cases}$$

In order to allow a more general form $\pm x_i \pm x_j \leqslant k$ of octagonal constraints, a DBM $\mathtt{m}$ of size $n \times n$ defined over $\mathtt{Var}$ is extended to another DBM $\mathtt{m}'$ of size $2n \times 2n$ over the set of enhanced variables $\mathtt{Var}' = \{x'_1, \ldots, x'_{2n}\}$ where each variable $x_i \in \text{Var}$ comes in two forms: a positive form x'_{2i-1}, denoted x_i^+ and a negative form x'_{2i}, denoted x_i^-.

Let $\mathtt{L} = \langle \wp(\mathbb{R}^n), \subseteq, \varnothing, \mathbb{R}^n, \cap, \cup \rangle$ be the concrete lattice. We define the abstract lattice $\mathtt{A} = \langle \mathbb{M}_\perp, \sqsubseteq, \mathtt{m}_\perp, \mathtt{m}_\top, \sqcap, \sqcap \rangle$, where $\mathbb{M}$ is the set of all DBMs representing the domain of octagons, $\mathtt{m}_\perp$ is the bottom element that contains an unsatisfiable set of constraints, $\mathbb{M}_\perp = \mathbb{M} \cup \{\mathtt{m}_\perp\}$, and $\mathtt{m}_\top$ is the top element for which the bound for all constraints is ∞.

The Galois connection between L and A is formalized as $\langle \text{L}, \alpha_\mathbb{M}, \gamma_\mathbb{M}, \text{A} \rangle$, where $\alpha_\mathbb{M}$ and $\gamma_\mathbb{M}$ on $S \in \wp(\mathbb{R}^n)$ and $\mathtt{m} \in \mathbb{M}_\perp$ are defined here:

- if $S = \varnothing$: $\alpha_\mathbb{M}(S) \triangleq \mathtt{m}_\perp$
- if $S \neq \varnothing$: $\alpha_\mathbb{M}(S) = \mathtt{m}$ where $\mathtt{m}_{ij} \triangleq$

$$\begin{cases} \max\{\rho(x_l) - \rho(x_k) | \rho \in S\} & \text{when } i = 2k-1, j = 2l-1 \\ & \text{or } i = 2l, j = 2k \\ \max\{\rho(x_l) + \rho(x_k) | \rho \in S\} & \text{when } i = 2k, j = 2l-1 \\ \max\{-\rho(x_l) - \rho(x_k) | \rho \in S\} & \text{when } i = 2k-1, j = 2l \end{cases}$$

$$\gamma_\mathbb{M}(\mathtt{m}) = \begin{cases} \varnothing & \text{if } \mathtt{m} = \mathtt{m}_\perp \\ \mathbb{R}^n & \text{if } \mathtt{m} = \mathtt{m}_\top \\ \{(k_1, \ldots, k_n) \in \mathbb{R}^n | (k_1, -k_1 \ldots k_n, -k_n) & \\ \in \text{dom}(\mathtt{m}) \text{ and } \forall i, j : x_j - x_i \leqslant \mathtt{m}_{ij} & \text{otherwise} \end{cases}$$

TABLE 5 A summary on various abstract domains [16].

Domain	Invariants	Time cost	Memory cost	Precision
Interval	$x \in \{[l,\ h] \mid l,\ h \in \mathbb{R}, l \leq h, x \in \mathbb{V}\}$	$O(n)$	$O(n)$	*Low*
Octagon	$\pm x_i \pm x_j \leqslant k,\ x_i, x_j \in \text{Var} \wedge k \in \mathbb{R}$	$O(n^3)$	$O(n^2)$	*Medium*
Polyhedra	$\Sigma_{i=1}^{n}\ a_i x_i \geq k,\ x_i \in \text{Var}\ \wedge\ a_i, k \in \mathbb{R}^n$	$O(2^n)$	$O(2^n)$	*High*

Polyhedra [53, 56]

Intuitively, the support of more number of relations among program's variables in a relational abstract domain improves the analysis precision of a program. Cousot and Halbwachs in their seminal work [53] first introduced the domain of polyhedra for static determination of linear equality and inequality relations among program variables, and interestingly it is observed that the analysis in this domain, although computationally costly, improves precision significantly compared to the octagon abstract domain.

Polyhedra is defined as a region in an n-dimensional space $\mathbb{R}^n$ which is bounded by a finite set of hyperplanes. Let n be the number of variables in a program and $\vec{v} = \langle v_1, v_2, \ldots, v_n \rangle$ be the possible values of these variables. In fact, $\vec{v}$ represents a point in $\mathbb{R}^n$. Let us denote a linear inequality over $\mathbb{R}^n$ which defines an affine half-space of $\mathbb{R}^n$ as $\alpha \triangleq \vec{v}.\ \vec{x} \geq m$, where $\vec{v} \neq \vec{o}$, $\vec{x} = \langle x_1, x_2, \ldots, x_n \rangle$, $m \in \mathbb{R}$. A convex polyhedron P is expressed as the intersection of a finite set of such affine half-spaces. Formally, $P \triangleq (X, n)$, where $X = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$ on $\mathbb{R}^n$ is the set of linear inequalities [57]. On the other hand, a set of solutions or points in $\mathbb{R}^n$ satisfying the set of linear inequalities X defines a polyhedron P.

Let $\mathrm{L} = \langle \wp(\mathbb{R}^n),\ \subseteq, \varnothing, \mathbb{R}^n,\ \cap, \cup \rangle$ be the concrete lattice defined over the concrete domain. The set of polyhedra $\mathbb{P}$ with partial order $\sqsubseteq$ forms an abstract lattice $\mathrm{A} = \langle \mathbb{P},\ \sqsubseteq, \mathbb{P}_\perp, \mathbb{P}_\top,\ \sqcap, \sqcap \rangle$.

The concretization function $\gamma_{\mathbb{P}}$ from abstract domain to concrete domain is defined here:

$$\gamma_{\mathbb{P}}(P) = \begin{cases} \varnothing & \text{if } P = P_\perp \\ \mathbb{R}^n & \text{if } P = P_\top \\ \left\{ \rho \in \Sigma \mid \forall (\vec{v}\ .\ \vec{x} \geq k) : \vec{v}\ .\rho(\vec{x}) \geq k \right\} & \text{otherwise} \end{cases}$$

where $P \in \mathbb{P}$ and the environment $\rho \in \Sigma \triangleq \text{Var} \mapsto \mathbb{R}$ maps each variable to its value in $\mathbb{R}$.

A summary on the strength and weakness of different relational and nonrelational abstract domains is reported in Table 5.

7.2 The reduced product

The abstract semantics defined earlier accumulates variables dependences at different program points. These dependences are determined based on the syntactic presence of variables in program statements. Such syntax-based computation yields false positives [58]. For instance, consider the statement "$y = 4w \bmod 2$," even though the variable y is syntactically dependent on w, we can observe that semantically there is no dependency as the expression value "$4w \bmod 2$" is always equal to 0. Few proposals in the literature attempted to remove such false dependences from syntax-based analysis by considering program's semantics in suitable abstract domains [16, 59, 60]. In similar direction as suggested in Refs. [13, 48], in this section, we describe how to improve precision of the analysis result obtained in the domain `Pos` by combining it with the analysis results in other widely used numerical abstract domains.

The static analyses of programs in the above-mentioned numerical abstract domains (domain of *intervals*, *polyhedra*, *octagons*, etc.) determine a sound approximation of programs' runtime behaviors accumulating possible values of variables (in abstract form) at each program point [61, 62]. As suggested in Refs. [13, 48], here we recall how a reduced product between both the *propositional formulae domain* and the *numerical abstract domain* allows to exclude false dependences among program's variables.

Consider an abstract domain $\mathbb{D}$ where the numerical values are abstracted by, for instance, an interval or an octagon or a polyhedra. Suppose Σ^* and $\overline{\Sigma^*}$ represent the set of concrete and abstract traces, respectively. Given the lattices

$L_c = \langle \wp(\Sigma^*), \subseteq, \varnothing, \Sigma^*, \cap, \cup \rangle$, $L_p = \langle \wp(\overline{\Sigma^*}), \sqsubseteq_p, \bot_p, \top_p, \sqcap_p, \sqcap_p \rangle$ and $L_n = \langle \mathbb{D}, \sqsubseteq_n, \bot_n, \top_n, \sqcap_n, \sqcap_n \rangle$, let ($L_c$, α_0, γ_0, L_p) and (L_c, α_1, γ_1, L_n) be two Galois Connections. The reduced product $\mathtt{RP} : \wp(\overline{\Sigma^*}) \times \mathbb{D} \rightarrow \wp(\overline{\Sigma^*})$ be a reduced product operator defined as $\mathtt{RP}(S, d) = S'$, where $S \in \wp(\overline{\Sigma^*})$ is a set of partial traces, $d \in \mathbb{D}$, and

$$S' = \left\{ \langle \ell_i, \psi_k \rangle \mid \langle \ell_i, \psi_j \rangle \in S \wedge \psi_k = (\psi_j \ominus \{x \rightarrow y \mid y \in d\}) \right\}$$

8. Conclusion

This chapter addressed two crucial security issues—confidentiality and integrity—of smart contract-enabled Ethereum decentralized applications. The primary reason behind this vulnerability is information flow in an uncontrolled way when smart contracts are in execution. Being part of smart contracts statements, variables with different sensitivity levels influence each other when the statements execute. We considered a formal language-based security analysis which performs a lattice-theoretic security policy enforcement while verifying the smart contracts for possible security compromise. The use of Abstract Interpretation theory served as a generic framework to adopt semantics-based analysis and verification on top of its syntactic counterpart, as a way to improve analysis precision with a wise choice of abstraction level. Though we addressed a subset of Solidity features in our proposal, Solidity possesses many other crucial blockchain-specific features, for example, account-based financial transactions, which require appropriate treatment in this perspective. Our future work aims a prototype development based on our proposal to demonstrate the efficacy of the system, with an extended support to more Solidity features.

Acknowledgment

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

References

[1] S. Nakamoto, Bitcoin: a peer-to-peer electronic cash system, 2008. https://bitcoin.org/bitcoin.pdf.
[2] V. Buterin, Ethereum white paper: a next-generation smart contract and decentralized application platform, 2014. https://translatewhitepaper.com/wp-content/uploads/2021/04/EthereumOrijinal-ETH-English.pdf.
[3] M. Pilkington, Blockchain technology: principles and applications, in: Handbook of Research on Digital Transformations, Edward Elgar, 2016.
[4] Ethereum, Solidity Documentation—Release 0.8.4., http://solidity.readthedocs.io/.
[5] G. Wood, Ethereum: a secure decentralised generalised transaction ledger, 2014. https://ethereum.github.io/yellowpaper/paper.pdf.
[6] P. Praitheeshan, L. Pan, J. Yu, J. Liu, R. Doss, Security analysis methods on Ethereum smart contract vulnerabilities: a survey, 2019. arXiv preprint arXiv:1908.08605.
[7] D. Volpano, C. Irvine, G. Smith, A sound type system for secure flow analysis, J. Comput. Secur. 4 (2–3) (1996) 167–187.
[8] G. Smith, Principles of secure information flow analysis, in: M. Christodorescu, S. Jha, D. Maughan, D. Song, C. Wang (Eds.), Malware Detection, Advances in Information Security, vol. 27, Springer US, Novy Smokovec, Slovakia, 2007, pp. 291–307.
[9] D. Hedin, A. Sabelfeld, A perspective on information-flow control, in: Proceedings of the 2011 Marktoberdorf Summer School, IOS Press, 2011.
[10] D.E. Denning, A lattice model of secure information flow, Commun. ACM 19 (1976) 236–243.
[11] A. Sabelfeld, A.C. Myers, Language-based information-flow security, IEEE J. Sel. Areas Commun. 21 (2003) 5–19.
[12] A. Cortesi, R. Halder, Information-flow analysis of hibernate query language, in: Proceedings of the 1st International Conference on Future Data and Security Engineering, Springer LNCS 8860, Vietnam, 2014, pp. 262–274.
[13] A. Cortesi, P. Ferrara, R. Halder, M. Zanioli, Combining Symbolic and Numerical Domains for Information Leakage Analysis, Springer, 2018, pp. 98–135.
[14] C. Hammer, G. Snelting, Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs, Int. J. Inf. Secur. 8 (2009) 399–422.
[15] S. Just, A. Cleary, B. Shirley, C. Hammer, Information flow analysis for Javascript, in: Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients, ACM Press, Portland, Oregon, USA, 2011, pp. 9–18.
[16] A. Jana, R. Halder, K.V. Abhishekh, S.D. Ganni, A. Cortesi, Extending abstract interpretation to dependency analysis of database applications, IEEE Trans. Softw. Eng. 46 (5) (2018) 463–494.
[17] M.I. Alam, R. Halder, Refining dependencies for information flow analysis of database applications, Int. J. Trust Manage. Comput. Commun. 3 (3) (2016) 193–223.
[18] R. Akella, H. Tang, B.M. McMillin, Analysis of information flow security in cyber-physical systems, Int. J. Crit. Infrastruct. Prot. 3 (3–4) (2010) 157–173.

[19] D. Hedin, A. Birgisson, L. Bello, A. Sabelfeld, JSFlow: tracking information flow in JavaScript and its APIs, in: Proceedings of the 29th Annual ACM Symposium on Applied Computing, 2014, pp. 1663–1671.
[20] T. Amtoft, S. Bandhakavi, A. Banerjee, A logic for information flow in object-oriented programs, ACM SIGPLAN Not. 41 (1) (2006) 91–102.
[21] F. Pottier, V. Simonet, Information flow inference for ML, in: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2002, pp. 319–330.
[22] U. Schöpp, C. Xu, A generic type system for featherweight Java, in: Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-Like Programs, 2021, pp. 9–15.
[23] C. Hammer, J. Krinke, G. Snelting, Information flow control for Java based on path conditions in dependence graphs, in: Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE 2006), IEEE, Arlington, VA, 2006, pp. 87–96.
[24] S. Cavadini, Secure slices of insecure programs, in: Proceedings of the ACM Symposium on Information, Computer and Communications Security, ACM Press, Tokyo, Japan, 2008, pp. 112–122.
[25] D. Wasserrab, D. Lohner, G. Snelting, On PDG-based noninterference and its modular proof, in: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, ACM, 2009, pp. 31–44.
[26] T. Amtoft, A. Banerjee, A logic for information flow analysis with an application to forward slicing of simple imperative programs, Sci. Comput. Program. 64 (2007) 3–28.
[27] G.R. Andrews, R.P. Reitman, An axiomatic approach to information flow in programs, ACM Trans. Program. Lang. Syst. 2 (1) (1980) 56–76.
[28] R. Dimitrova, B. Finkbeiner, M. Kovács, M.N. Rabe, H. Seidl, Model checking information flow in reactive systems, in: Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation, Springer-Verlag LNCS, Philadelphia, PA, 2012, pp. 169–185.
[29] R. Joshi, K.R.M. Leino, A semantic approach to secure information flow, Sci. Comput. Program. 37 (1–3) (2000) 113–138.
[30] M. Zanioli, P. Ferrara, A. Cortesi, SAILS: static analysis of information leakage with sample, in: Proceedings of the 2012 ACM Symposium on Applied Computing, ACM Press, Riva del Garda (Trento), 2012, pp. 1308–1313.
[31] R. Halder, M. Zanioli, A. Cortesi, Information leakage analysis of database query languages, in: Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC '14, ACM, Gyeongju, Korea, 2014, pp. 813–820.
[32] R. Halder, A. Jana, A. Cortesi, Data leakage analysis of the hibernate query language on a propositional formulae domain, in: Trans. Large-Scale Data- and Knowledge-Centered Systems, vol. 23, Springer, 2016, pp. 23–44.
[33] D. Vujičić, D. Jagodić, S. Ranđić, Blockchain technology, bitcoin, and Ethereum: a brief overview, in: 2018 17th International Symposium Infoteh-Jahorina (INFOTEH), IEEE, 2018, pp. 1–6.
[34] A.A. Monrat, O. Schelén, K. Andersson, A survey of blockchain from the perspectives of applications, challenges, and opportunities, IEEE Access 7 (2019) 117134–117151.
[35] S. Krishnapriya, G. Sarath, Securing land registration using blockchain, Procedia Comput. Sci. 171 (2020) 1708–1715, https://doi.org/10.1016/j.procs.2020.04.183.
[36] F.Þ. Hjálmarsson, G.K. Hreiðarsson, M. Hamdaqa, G. Hjálmty'sson, Blockchain-based e-voting system, in: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), IEEE, 2018, pp. 983–986.
[37] B. Putz, M. Dietz, P. Empl, G. Pernul, Ethertwin: blockchain-based secure digital twin information management, Inf. Process. Manag. 58 (1) (2021) 102425.
[38] M. Shuaib, S. Alam, M.S. Alam, M.S. Nasir, Self-sovereign identity for healthcare using blockchain, Mater. Today Proc. (2021).
[39] A. Whitaker, A. Bracegirdle, S de Menil, M.A. Gitlitz, L. Saltos, Art, antiquities, and blockchain: new approaches to the restitution of cultural heritage, Int. J. Cult. Policy 27 (3) (2021) 312–329.
[40] N. Wang, H. Xu, F. Xu, L. Cheng, The algorithmic composition for music copyright protection under deep learning and blockchain, Appl. Soft Comput. 112 (2021) 107763.
[41] J. Shen, Blockchain technology and its applications in digital content copyright protection, in: Proceedings of the 4th International Conference on Economic Management and Green Development, Springer, 2021, pp. 18–25.
[42] J.A. Goguen, J. Meseguer, Unwinding and inference control, in: IEEE Symposium on Security and Privacy, IEEE, 1984, p. 75.
[43] F.E. Allen, J. Cocke, A program data flow analysis procedure, Commun. ACM 19 (3) (1976) 137.
[44] C. Marco, C. Gabriele, 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.
[45] T. Armstrong, K. Marriott, P. Schachte, H. Søndergaard, Two classes of Boolean functions for dependency analysis, Sci. Comput. Program. 31 (1) (1998) 3–45.
[46] A. Cortesi, G. Filé, W.H. Winsborough, Prop revisited: propositional formula as abstract domain for groundness analysis, in: LICS, 1991, pp. 322–327.
[47] A. Cortesi, G. Filé, W.H. Winsborough, Optimal Groundness Analysis Using Propositional Logic, J. Log. Program. 27 (2) (1996) 137–167.
[48] M. Zanioli, A. Cortesi, Information leakage analysis by abstract interpretation, in: SOFSEM 2011: Theory and Practice of Computer Science—37th Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 22–28, 2011. Proceedings, 2011, pp. 545–557.
[49] A. Cortesi, G. Costantini, P. Ferrara, A survey on product operators in abstract interpretation, in: EPTCS, Semantics, Abstract Interpretation, and Reasoning about Programs, vol. 129, 2013, pp. 325–336, https://doi.org/10.4204/EPTCS.129.19.

[50] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of the POPL'77, 1977, pp. 238–252.
[51] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM Press, San Antonio, TX, 1979, pp. 269–282.
[52] P. Cousot, Constructive design of a hierarchy of semantics of a transition system by abstract interpretation, Theor. Comput. Sci. 277 (1–2) (2002) 47–103.
[53] P. Cousot, N. Halbwachs, Automatic discovery of linear restraints among variables of a program, in: Proceedings of the POPL '78, 1978, pp. 84–96.
[54] A. Miné, A new numerical abstract domain based on difference-bound matrices, in: Programs as Data Objects, Second Symposium, PADO, 2001, pp. 155–172.
[55] A. Minè, The octagon abstract domain, Higher Order Symbol. Comput. 19 (1) (2006) 31–100.
[56] L. Chen, A. Minè, P. Cousot, A sound floating-point polyhedra abstract domain, in: Proc. of the 6th Asian Symposium on PLS, 2008, pp. 3–18.
[57] N.V. Chernikoba, Algorithm for discovering the set of all the solutions of a linear programming problem, USSR Comput. Math. Math. Phys. 8 (6) (1968) 282–293.
[58] I. Mastroeni, D. Zanardini, Data dependencies and program slicing: from syntax to abstract semantics, in: PEPM '08: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press, San Francisco, CA, 2008, pp. 125–134.
[59] R. Halder, A. Cortesi, Abstract program slicing on dependence condition graphs, Sci. Comput. Program. 78 (9) (2013) 1240–1263, https://doi.org/10.1016/j.scico.2012.05.007.
[60] R. Halder, A. Cortesi, Abstract program slicing of database query languages, in: Proc. of the 28th Symposium On Applied Computing, ACM Press, Coimbra, Portugal, 2013, pp. 838–845.
[61] L. Chen, A. Miné, P. Cousot, A sound floating-point polyhedra abstract domain, in: Proc. of the 6th Asian Symposium on Programming Languages and Systems, ACM Press, Bangalore, India, 2008, pp. 3–18.
[62] A. Miné, The octagon abstract domain, Higher Order Symbol. Comput. 19 (2006) 31–100.