Pervasive and Mobile Computing 67 (2020) 101227

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc

Review

Verification of smart contracts: A survey

Mouhamad Almakhour [a,b,*], Layth Sliman [c], Abed Ellatif Samhat [b], Abdelhamid Mellouk [a]

[a] Paris-Est Creteil University, LISSI-TincNET Research Team, Créteil and Vitry-sur-Seine, France
[b] Lebanese University, Faculty of Engineering-CRSI, University Campus, Hadath, Lebanon
[c] EFREI Engineering School-Paris, Villejuif, France

ARTICLE INFO

Article history:
Received 31 January 2020
Received in revised form 29 June 2020
Accepted 3 August 2020
Available online 8 August 2020

Keywords:
Smart contracts
Blockchain
Verification
Correctness
Security assurance

ABSTRACT

To achieve trust and continuity in the smart contracts-based business process execution, the verification of such smart contracts is mandatory. A blockchain-based smart contract should work as intended before using it. Due to the immutable nature of blockchain, any bugs or errors will become permanent once published and could lead to huge economic losses. To avoid such problems, verification is required to check the correctness and the security of the smart contract. In this survey, we consider the smart contracts and we investigate smart contacts formal verification methods. We also investigate the security assurance for smart contracts using vulnerabilities detection methods. In this context, we provide a detailed overview of the different approaches to verify the smart contracts and we present the used methods and tools. We show a description of each method as well as its advantages and limitations and we draw several conclusions.

Contents

* Corresponding author at: Paris-Est Creteil University, LISSI-TincNET Research Team, Créteil and Vitry-sur-Seine, France.
E-mail addresses: mouhamad.almakhour@univ-paris-est.fr (M. Almakhour), layth.sliman@efrei.fr (L. Sliman), samhat@ul.edu.lb (A.E. Samhat), mellouk@u-pec.fr (A. Mellouk).

https://doi.org/10.1016/j.pmcj.2020.101227
1574-1192/

1. Introduction

Boosted-up by the spread of ICT technologies, many new economic models are emerging in the global markets, such as demand-driven economy, virtual marketplaces and distributed supply chain. These economic and technological forces are producing more and more complex systems, where the interconnection between actors, the availability of trusted information, as well as cost and revenue sharing among the actors are the key factors to obtain sustainable and cost-effective businesses. These systems require the presence of decentralized, yet trusted, process and data management.

Distributed Legder Technology (DLT) can help addressing both, trust and decentralization problems in collaborative business processes. In this context, blockchain [1,2] as a DLT technology, initially proposed for cryptocurrency, has recently gained a lot of interest from a variety of sectors such as government [3], finance [4,5], industry [6], health [7], internet of thing IoT [8,9] and research [10]. It offers key functionalities including data persistence, anonymity, fault-tolerance, auditability, resilience, and decentralized execution. For instance, a company called Everledger [11] built a blockchain based system that allows tracking diamonds from their source, in order to help stakeholders, ensure their diamonds are conflict-free. In addition, several research works [12–15] have been conducted on this field to prove the feasibility of the blockchain-based collaborative business processes using a high-level notation, (such as the Business Process Model and Notation (BPMN))

More recently, the introduction of smart contracts has extended the functionalities of blockchains. A smart contract is a computer program intended to enforce the execution of a deal between two or more parties. In the context of blockchain, a smart contract is a immutable computer program stored in the blockchain and executed by some of its nodes. The smart contract is usually written in a high-level language such as Solidity or Vyper, and then it is compiled down to the bytecode that runs on the blockchain like Ethereum Virtual Machine (EVM) in Ethereum blockchain. Other blockchain platforms can create and run smart contracts written by different high-level languages. For example, the Hyperledger Fabric [16] is a permissioned blockchain infrastructure, contributed by IBM and Digital Asset. Hyperledger Fabric provides execution of Smart Contracts called "chain code", written in a Golang and javascript [17]. Also, Tezos is a blockchain-based cryptocurrency and a smart contracts platform for building decentralized applications (dApps) [18]. Tezos-smart contracts [19] are written by Liquidity high-level language with Michelson language (it is the domain-specific language used to write smart contracts on the Tezos blockchain). According to [20], Tezos-smart contracts were designed with security and formal verification in mind.

Thus the smart contracts [21] are paramount to design and implement a business process, which greatly contributes to dependence on the use of blockchain in business process management systems (BPMSs) [22]. The correctness and security of the smart contracts are required as smart contract failures may cause millions of dollars of lost funds. Thus, a blockchain applications based on smart contracts should be checked and verified to ensure the correctness, security, and safety of the smart contract implementations. We focus in this paper on the verification of smart contracts. Given that, the Ethereum platform is the most widely used in the world, especially in smart contracts, thus we only consider the verification of smart contracts on Ethereum blockchain. We investigate two aspects of smart contracts verification, the first is related to the correctness of smart contracts and the second one focuses on the security assurance of smart contracts.

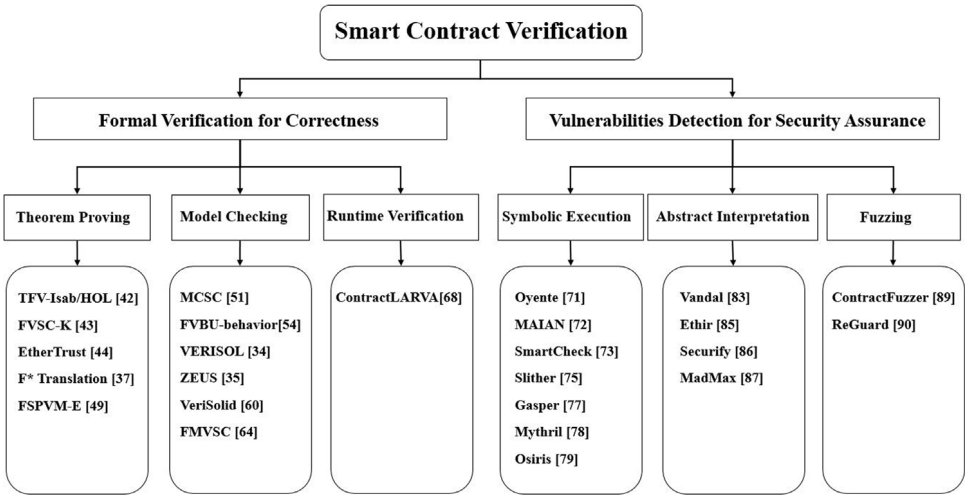

Fig. 1. Taxonomy of the smart contract verification's tools.

The correctness verification is about respecting the specifications that determine how users can interact with the smart contracts and how the smart contracts should behave when used correctly. There are two approaches used to verify the correctness: the formal verification and the programming correctness. The formal verification methods [23] are based on formal methods (mathematical methods [24]), while the programming correctness methods are based on ensuring the programming as code is correct, which means the program runs without entering the loop and gives correct outputs for correct inputs. We mainly focus on the formal verification approaches that is because formal verification is more rigorous and reliable.

On the other hand, the security assurance aspect is also important as the correctness aspect. In addition, the smart contracts are immutable nature, so any bugs or errors will become permanent once published and could lead to huge economic losses. To avoid this, we investigate the vulnerabilities detection methods that aim at improving the security of smart contracts by the study of vulnerabilities by verifying the smart contracts against a list of already defined and well-known vulnerabilities patterns. The vulnerabilities detection can avoid the same mistakes, which makes smart contracts more secure.

In this survey, the smart contract is the core of interest. This survey covers two aspects:

i Smart contract verification to achieve the correctness of collaboration process.
ii Vulnerabilities detection related to the security assurance of smart contracts to avoid bugs and errors.

For the formal verification, we investigate the methods based on Theorem Proving, Model Checking and Runtime Verification as well as the existing platforms based on these methods. And for the vulnerabilities detection methods, we present Symbolic Execution, Fuzzing and Abstract Interpretation as well as the existing platforms based on these methods. This work is helpful for researchers that would start working with the formal verification and the vulnerabilities detection of smart contracts.

In Fig. 1, we illustrate the taxonomy of smart contract verification and vulnerability detections methods adopted in this survey.

The rest of this paper is organized as follows; in Section 2, we give required background about smart contracts and the verification requirements. Then we consider the verification of correctness including the different models and platforms in Section 3. The vulnerabilities detection and their tools are presented in Section 4. Section 5 presents the discussion and future directions. Finally, Section 6 concludes the survey.

2. Theoretical background and definitions

In this section, we briefly provide the theoretical background related to blockchain and smart contracts in addition to the notions typically used.

2.1. Blockchain and smart contracts

Blockchain technology allows a distributed computing architecture where the transactions are publicly announced and the participants agree on a single history of these transactions (or ledger) [25]. The transactions are grouped into blocks, given timestamps, and then published. The hash of each block includes the hash of the previous block to form a chain,

making published blocks difficult to alter [26]. Blockchain technology is changing the way in which computer systems can regulate the interaction between real-world parties in a variety of ways. The requirements of the participation of trusted central authorities or resource managers were a limitation for the use of smart contracts [27]. Blockchain based smart contracts enforced implicitly by certain Blockchain architectures have opened wide opportunities. A smart contract is a computer program represented by its source code. It can implement automatically the content of a separate agreement expressed in natural language. Smart Contracts are often written using high-level languages such as Solidity (see Fig. 2), then are compiled into bytecode and embedded in self-contained that can be deployed in any node in the Blockchain. Due to the losses caused by uncorrected smart contracts, the verification of smart contracts is mandatory to avoid the failures and vulnerabilities. In the next section, we present several reasons to show why smart contracts should be verified before the deployment on Blockchain.

```
pragma solidity ^0.4.21;

contract User {
   struct user{

        int idUser;
        string nameUser;
        address addressUser;
        string posteUser;
        int yearsUser;
        int status;
    }
     mapping (uint => user)  User;
     user [] public UserS;
     user [] public RevocationUserListe;
     uint public counter=0;

    address private owner;
address private newOwner;


modifier onlyOwner() {
    require(msg.sender == owner);
    _;
}
/// @notice The Constructor assigns the message sender to be `owner`
function Owned() {
    owner = msg.sender;
}

function changeOwner(address _newOwner) onlyOwner {

    newOwner = _newOwner;}

function acceptOwnership() {
    if (msg.sender == newOwner) {
        owner = newOwner;
```

Fig. 2. Part of solidity's code.

2.2. Verification needs for smart contracts

The concept of smart contracts in distributed ledger systems has been considered as a safe way of enforcing agreements between participating parties. However, unlike legal contracts, which talk about the behavior to be conducted and the consequences of behaving differently, the executable code embedded in smart contracts give explicit instructions on how to achieve compliance. Consequently, ensuring correctness of the executable codes is an important, challenging task because incidents may lead to huge financial losses due to bugs, breaches and flaws in smart contracts. For instance, as mentioned in [28], an attacker managed to drain more than 3.6 million Ether by exploiting a flaw in the Distributed Autonomous Organization (DAO) smart contract's source code. In September 2017, the multi-signature wallet Parity was attacked in Ethereum, which had resulted in more than 150,000 Ether (about $30 million) embezzlement. In April 2018, because of the BEC attack, about $900 million were stolen. Facing these losses, the verification of smart contracts before the deployment on blockchain has attracted extensive attention. We can identify at least three reasons to apply formal specifications and verifications to smart contracts: First, smart contract vulnerabilities, due to the immutable nature of smart contracts, any bugs or errors will become permanent once published and could lead to huge losses. For this, we need to ensure the safety and security of smart contracts. Vulnerabilities in smart contracts have resulted in several high-profile exploits in the blockchain technology. Second, smart contracts are often low-level implementations of a high-level workflow that comprises a state machine with different actions predicated by suitable access control to determine who has the permission to execute a given action [29]. The lack of knowledge at the programmers for the semantics of programming that reflects the high-level workflows may lead to many misconception issues. As a result, we obtain "unfair contracts", which are syntactically technically correct but do not implement the desired business logic. Therefore, there is a strong need for a high-level specification language to express the intent of the workflow in a smart contract. Third, with regard to correctness, many programming paradigms used to write smart contract are not designed to be used in the context of the blockchain environment [30], resulting in many problems during execution time.

2.3. Concepts and definitions

Since 2017, the number of papers related to smart contract verification and vulnerabilities detection has increased dramatically. To this end, many surveys and studies have been done. To the best of our knowledge, these works considered smart contract verification and vulnerabilities detection separately. For instance, [31] classified the vulnerabilities according to the location of vulnerability in Ethereum layers, and it presents some best practices to avoid these vulnerabilities. Also, [32] surveys 27 tools for analyzing smart contracts and describe their characteristics. [33] Identifies the security problems and vulnerabilities in Ethereum smart contracts which have caused severe attacks and classified it into three categories: static, dynamic and formal verification. In addition, [34] considers only the formal verification tools. Finally, [35] surveys the most tools of analyzing smart contracts and classified it under two concepts, tools to verify the correctness of smart contracts and tools to ensure the security of smart contracts. Different from the above surveys, this survey investigates both formal verification and vulnerabilities detection. We present the used methods, tools and approaches related to these two fields. This work skips many other surveys by providing a detailed description of each method and tool as well as its advantages and limitations. It also gives a detailed overview of 25 tools for verification of Ethereum-smart contracts, specially the formal verification tools and vulnerabilities detection tools. We then discuss the different methods and we draw several conclusions and future research directions.

This work is helpful for researchers that would start working with the formal verification and the vulnerabilities detection of smart contracts.

We briefly explain some notions mentioned in this survey.

Bytecode refers to the list of byte-size integers that serve as instructions for the Ethereum Virtual Machine (EVM).

Source code refers to a program in a high-level programming language, here Solidity.

Static analysis refers to a class of methods that examine the source code or bytecode of a contract without executing it.

Dynamic analysis means to observe a contract while executing (parts of) it in the original context.

Disassembling means to translate EVM bytecode into better readable assembly language, where machine operations and storage addresses are represented symbolically.

Decompilation is the process of transforming EVM bytecode to a more compact representation on a higher abstraction level (like intermediate or Solidity code) to enhance the readability of the code or to ease data flow analysis.

Basic block is a sequence of statements without branches.

Control flow graph (CFG) is a directed graph, where the basic blocks of a program serve as the nodes. An arc connects node A with node B if it is possible that block B gets executed immediately after block A. The arc may be labeled by the condition under which this path is chosen.

Abstract Syntax Tree (AST) represents the syntactic structure of Solidity code as a tree. It occurs as an intermediate product when compiling Solidity to bytecode. Often, it is better suited for analyzing Solidity code.

Execution trace is the sequence of instructions (possibly including additional information) executed during a particular run of the code.

Transformation from a stack to a register-oriented view is a particular de-compilation technique that replaces stack-oriented instructions of the Ethereum Virtual Machine by instructions operating on registers.

Finite state machines (FSMs) are abstract models of systems that can be in a finite number of states only. FSMs are characterized by listing all states, by designating the initial and final states, and by describing the actions that will cause the machine to transition to another state.

Horn logic is a restricted form of first-order logic where all formulas ('clauses') are if–then rules. Though restricted, Horn logic is still computationally universal, thus it can perform the same computations as any computer.

DataLog is a restricted form of Horn logic that is no longer computationally universal, but it allows for efficient processing, e.g. with tools like Soufflé.

3. Verification of correctness: Formal verification for smart contracts

Based on the formal methods, the formal verification of smart contracts provides the correctness of smart contracts in a rigorous and reliable mathematical model [23,24]. In order to perform formal verification, formal specifications could be used. By using mathematical methods, formal verification can attest that the final program behaves exactly as described in its specification. Formal verification is used in the fields where errors can be quite significant as it eliminates human error. The basic steps in formal verification as given in [36]: Formally model the system; Formalize the specification; and Prove that the model satisfies the specification. Most formal verification tools use human-readable (or close to human-readable) languages for specifications. Thus, the results of the audit look like comprehensive documentation more than a security report. We can use mathematical methods to model and rigorously verify the correctness of smart contracts and know if the program behaves as described in the specification. Since verification is automatic, it will be performed efficiently. Formal verification could detect unknown vulnerabilities because the formal verification audit cover the whole program, with no missed cases. As for formal verification weaknesses, it is clear that formal verification detects only inconsistencies between the formal specification and the implementation. This means if there are errors or deficiencies in the specification, many errors or breaches will be left undetected. Consequently, the specifications should be prepared carefully. In addition, formal verification demands a lot of preparation from the development team.

In the case of smart contracts verification, we can improve smart contracts security by ensuring the correctness of contracts using formal verification. There are several projects aimed at creating formally specified execution environments (virtual machines) for various networks such as (Implementations in F* [37], Formal Ethereum Virtual Machine semantics in the K Framework [38]). Formal verification uses many approaches, here we will limit our discussion to those which are widely used in smart contract context, mainly Theorem Proving, Model Checking and Runtime Verification.

3.1. Theorem proving

In theorem proving [39], the system is modeled mathematically, and the desired properties to be proven are specified. Then, the verification is performed. Theorem proving uses well-known axioms and simple inference rules. These are used to derive the new theorems, lemmas as needed for the proof [34]. Theorem proving is a very flexible verification method and it can be applied to different kind of systems as long as they can be expressed mathematically. Theorem proving can be interactive, automated or a hybrid [36]. Interactive theorem proving requires some human input. In contrast, the automated ones perform the theorem proving semi-automatically. Hybrid theorem proving works on verification of the complex parts with interactive theorem proving while the rest parts can be verified with automated theorem proving. For theorem proving, the most expressive logic is higher-order logic (HOL) [40]. Theorem proving suffers from many main limitations: the massive human investment to prove theorems, the need of a deep understanding of the method [34] and there are no fully automatic theorem provers [41]. The rest of this section provides the tools based theorem proving in smart contract context.

3.1.1. Towards Formal Verification in Isabelle/HOL (TFV-Isab/HOL)

In [42], the authors used the theorem prover Isabelle/HOL and an existing EVM-formal model to verify the bytecode of smart contracts. The goal was to create a sound program logic and to use the resulting program logic for verification. This is done by splitting the contracts into basic blocks of different types. To achieve the split, they decompiled the bytecode in order to extract Control Flow Graphs (CFG). In this case, basic blocks, which consist of code sequences that are not interrupted by jumps, are the vertices of the graph. They are connected by edges, which represent the jumps [42]. In this way, each basic block is a sequential piece of code, where the first instruction is executed first, and it continues uninterrupted until the last instruction of the basic block is finished. The basic blocks are further divided into different types, depending on the last instruction of the block. The types are Terminal, Jump, Jumpi, and Next [42]. When the basic blocks are obtained, the program logic is created using Hoare logic, which has Hoare triple designed for program correctness. As a result, a framework was created for expressing the EVM bytecode using logic, which is necessary in order to use a theorem prover for verification. The entire verification procedure was successfully applied to a case study. However, the framework does not support the full syntax of Solidity. For instance, loops and the message calls to other contracts are currently not supported.

3.1.2. Formal Verification of Smart Contracts with the K -Framework (FVSC-K)

In [43], a smart contract formal verification method based on K-Framework is proposed. K-Framework is one of the most robust and powerful language definition frameworks. It allows defining the programming language and providing a set of tools for that language, including both an executable model and a program verifier. This framework provides a user-friendly, modular, and mathematically rigorous meta-language for defining programming languages, type systems, and analysis tools. Additionally, the K-Framework enables verification of smart contracts. The K-Framework is composed of 8 components listed in Fig. 3. To prove that a program always does what it should, the contract's specification is required. However, creating a formal specification through a manual process requires considerable expertise. According to [43], runtime verification of bytecode is better than this of solidity as bytecode language can be used for any high-level contract language. They applied contract specification to refine (for bytecode) manually, and then came the role of the virtual semantic machine to know what each bytecode does and the bit-level data structure it works on. All of this is used as input to an automated theorem prover "K verification framework" with human help by adding the Hint component, and verified contract. Two messages from the prover are possible: "I created the proof that the contract is right" or "The contract is wrong".

3.1.3. EtherTrust

Grishchenko et al. [44] provides the first sound static analyzer for EVM bytecode. This tool supports reachability properties, which contain the most important security properties for smart contracts (e.g., single-entrancy and transaction environment dependency). The soundness is proven against a complete and mechanized semantics of EVM bytecode. EtherTrust translates bytecode to Horn clauses and it checks properties using the SMT solver Z3. This approach does not detect vulnerabilities but provides guarantees that the code is free of certain ones.

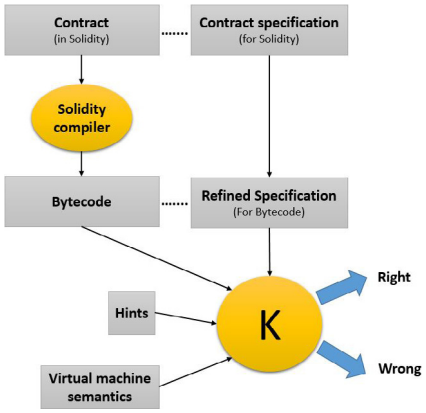

Fig. 3. Workflow of K-framework.

3.1.4. FSPVM-E

Using Hoare logic in Coq [45], which is one of the best higher-order logic theorem proving assistants, based on the Calculus of inductive construction (Cic) that supports Curry Howard Isomorphism (CHI), the work presented in [46] provided a novel formal symbolic process virtual machine (FSPVM), called FSPVM-E [46]. FSPVM-E is used to verify the reliability, security and functional correctness properties of smart contracts. First, the authors present EVI execution verification isomorphism, an extension of CHI that combines the advantages of model checking and theorem proving technology. EVI is the basic theory for combining Higher-Order Logic Systems (HOLS) and Symbolic Execution Technology to solve the problems in higher-order logic theorem proving. Second, FSPVM takes EVI as the fundamental theoretical framework to develop an FSPVM for Ethereum (FSPVM-E). The FSPVM-E has three main parts [46]: GERM a formal memory framework that supports different formal verification specifications at the code level; Lolisa an extensible formal intermediate programming language, and FEther a formally verified interpreter for Lolisa. These parts work conjointly: GERM simulates the physical memory hardware structure, Lolisa is a large subset of the Solidity programming language mechanized in Coq, and finally, FEther the virtual execution engine to symbolically execute and verify the formal version of smart contracts in FSPVM-E.

3.1.5. F* translation

In cooperation between Microsoft Research and Harvard University, a framework is done to analyze and formally verify Ethereum smart contracts using F* functional programming language [37,47]. Such contracts are generally written in Solidity [48] and compiled down to the Ethereum Virtual Machine (EVM) byte-code. In [37] they develop a language-based approach for verifying smart contracts. Two prototype tools based on F* are presented and a smart contract verification architecture is proposed and illustrated in Fig. 4.

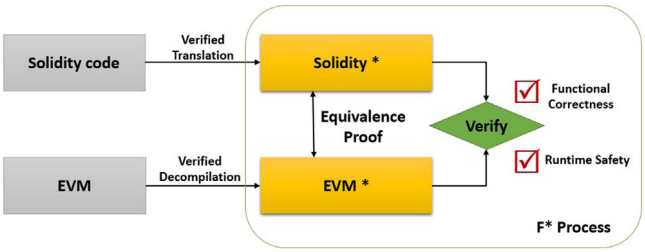

Fig. 4. F* architecture.

Translation Solidity to Solidity*

```
Contarct A {

mapping (address ➔ uint) balances;

Function transfer (unit K){
Balances = Balances - K;
}
```

```
Module A
Open solidity

type state = { balances: mapping address uint;}
val store : state = {balances = ref empty_map}

Let transfer (): Eth unit=
Update_map store.balances  msg.sender
(sub(lookup store.balance msg.sender) K)
```

Fig. 5. A simple solidity code translated by F*.

The contracts are translated into F* programs that call an F* runtime library for all Ethereum operations. The two tools are Solidity* and EVM*. Solidity* is a tool that compiles Solidity contracts into F* [37] in order to verify, at the source level, functional correctness specifications and safety with respect to runtime errors. The Fig. 5 show a simple solidity code translated by F* to Solidity*. EVM* is a tool that decompiles EVM bytecode into more succinct F* code hiding the details of the stack machine [37]. It allows to analyze low-level properties. Given a Solidity program and equivalent EVM bytecode, both are translated into F*. Then their equivalence is verified using relational reasoning [49].

3.2. Model checking

Given a finite-state model of a system and a formal property, model checking is an automated technique that systematically checks whether this property holds for that model [50]. The verification is done with model checking software like NuSMV, SPIN, etc. The model checker checks automatically if each state of the model satisfies the specifications given by the user. In case that there is a property not satisfied, the model checker provides a counterexample that can help us to identify mistakes and to correct bugs. On the other hand, if each state of the model satisfies the specification, the model is formally verified for that specific property. The Fig. 6 shows the procedure of the model checker. We can summarize the procedure of the model checking by the following steps [50]:
i. modeling (system–>model)
ii. Specification (natural language specification–>property in formal logic)
iii. Verification (algorithm to check whether a model satisfies a property). The rest of this section provides the formal verification tools based on model checking.

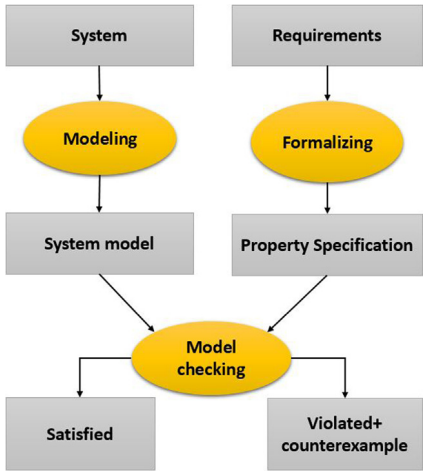

Fig. 6. Procedure of model checker.

3.2.1. Model checking for smart contracts (MCSC)

This work aims at establishing a generic modeling method of Ethereum application to apply a model-checking approach on smart contracts and its execution environment [51]. The proposed model is written in NuSMV input language [52] and the properties to check are formalized into temporal logic CTL (Computation Tree Logics) [53]. It has three components: the kernel layer that captures the (Ethereum) blockchain behavior, the application layer that models the smart contracts themselves (Translation rules from Solidity to NuSMV language have been provided to build the application layer), and the environment layer that determines an execution framework for the application [51]. To check whether the contracts behave as they are supposed to do, expected properties of the application have to be formalized into temporal logic. Properties include safety, fairness, reachability, real-time properties, etc. If a property does not hold, the model-checking provides a counter-example. Thus it is possible to determine the nature and the source of the defect. The method is quite generic and can be applied to various Ethereum applications. According to the case study given in [51], the model checking approach was able to verify four of the five properties. For the unsatisfied property, a counter-example was provided by the model checker.

3.2.2. Formal verification of smart contracts based on users and blockchain behaviors models (FVBU-behaviors)

In [54], the authors present a new approach to model smart contract and blockchain protocol execution along with users' behaviors based on a formal model checking language. This new approach was applied to a user registry smart contract, which was then formally verified by using model checking on the contract itself by and considering the user-behavior (if it is a hacker or not). In [54], they analyzed the safety of the registry smart contract execution by introducing a hacker behavior model. The hackers purpose is to steal the identity of a user by registering with his account. Three scenarios are evaluated in order to determine the probability of the hacker to successfully register the name. In Scenario

1, the hacker finds the name after the registration of the user from the mined blocks. In Scenario 2, the hacker retrieves the name from the pending transactions data which has not been mined yet. In Scenario 3, that is the easiest, the hacker gets the name directly from the user call to the contract. Using the model checker in Behavior Interaction Priorities (BIP) [55] and the Statistical Model Checking (SMC) to verify the probabilities, the following results were obtained: In scenario 1 the hacker has 0% success probability. In scenario 2 and 3, the hacker has an average of 12% and 25% respectively to hack the register.

3.2.3. VERISOL (Verifier for Solidity)

VERISOL (Verifier for Solidity) [29] it aims at prototyping a formal verification and analysis system for smart contracts developed using the Solidity programming language. It is based on translating programs written in Solidity language into programs in Boogie intermediate verification language, and then leveraging and extending the verification tool chain for Boogie programs. According to [29], the verifier "VeriSol" is built on the Boogie tool chain, because it can be used for both verification and counter-example generation. VERISOL takes as input a Solidity contract annotated with assertions, and then it yields one of the following three outcomes: fully verified, that means that the assertions in the contract are guaranteed not to fail under any usage scenario. Refuted that indicates at least one input and invocation sequence of the contract functions under which one of the assertions is guaranteed to fail. Partially verified, that is the case when VERISOL can neither verify nor refute contract correctness. It performs bounded verification to establish that the contract is safe up to k transactions. As shown in Fig. 7, VERISOL has three modules, namely (a) Boogie Translation from a Solidity program, (b) Invariant Generation to infer a contract invariant as well as loop invariants and procedure summaries, and (c) Bounded Model Checking. If VERISOL fails to verify contract correctness using monomial predicate abstraction, it employs an assertion directed bounded verifier, namely CORRAL, to look for a transaction sequence leading to an assertion violation [29]. The results of applied smart contracts in VERISOL verifier are relatively good for simple contracts. But according to the table of results in [29], the number of lines of Solidity code increases after the instrumentation (in average 159 versus 79 before the instrumentation). In addition, VERISOL finds bugs in 4 of 11 well-tested contracts and precisely pinpoints the trace leading to the violation, but the researchers did not mention the running time of VERISOL when it applied on original contracts. This leads us to conclude that VERISOL is not suitable for complex contracts.

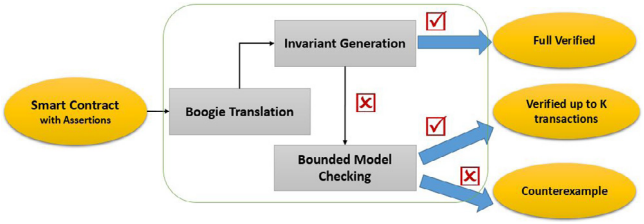

Fig. 7. Schematic workflow of VERISOL.

3.2.4. ZEUS framework

ZEUS [30] is a framework for automatic formal verification of smart contracts built by IBM India Research to verify the correctness and validate the fairness of smart contracts. ZEUS uses abstract interpretation and symbolic model checking. It is a tool-chain for smart contract verification that consists of (a) policy builder, (b) source code translator, and (c) verifier [30]. ZEUS takes as input the smart contracts written in high-level languages and leverages user assistance to help generate the correctness and/or fairness criteria in an eXtensible Access Control Markup Language (XACML) styled template [56]. It translates these contracts and the policy specification into a low-level intermediate representation (IR) by Low-Level Virtual Machine (LLVM) bitcode [57] encoding the execution semantics to correctly reason about the contract behavior. It then performs static analysis atop the IR to determine the points at which the verification predicates (as specified in the policy) must be asserted. Finally, ZEUS feeds the modified IR to a verification engine that leverages Constrained Horn Clauses (CHCs) [58,59]. CHCs provide a suitable mechanism to represent verification conditions, which can be discharged efficiently by SMT solvers (Satisfiability Modulo Theories) and quickly ascertain the safety of the smart contract.

3.2.5. VeriSolid

VeriSolid [60] is an open-source and web-based framework that is built on top of WebGME [61] and FSolidM [62]. The VeriSolid framework introduces formal verification capabilities, by providing an approach for correct-by-design development of smart contracts. First, the tool generates an augmented transition system from an initial transition system, based on the operational semantics of supported Solidity statements. Second, a Behavior Interaction-Priority (BIP) model of the contract is automatically generated from the augmented transition system. Then, the specified properties are automatically translated to Computational Tree Logic (CTL) [63]. The verification of safety and liveness properties of the model can be verified using the BIP-to-NuSMV tool [55], to translate the BIP to NuSMV, the input language of the nuXmv symbolic model checker [52]. A new analysis is made if the required properties are not satisfied by the model. As a final step, if all properties are satisfied and the developers are satisfied with the design, Verisolid generates automatically an equivalent Solidity code of the contract. Fig. 8 illustrates the Workflow of VeriSolid.

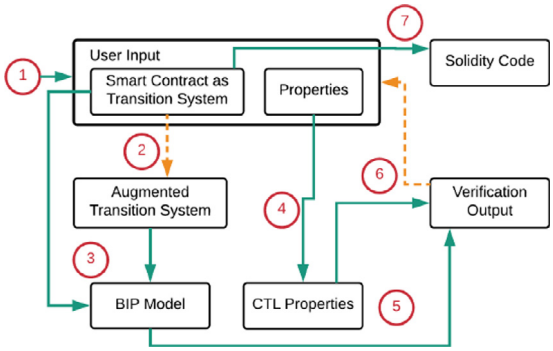

Fig. 8. Workflow of VeriSolid [60].

3.2.6. Formal Modeling and Verification of Smart Contracts (FMVSC)

The method in [64] aims to combine the formal methods and smart contracts to reduce potential errors and costs during the contract development process. A general smart contract template is provided from applying formal methods to smart contracts (it can be represented by tuple and finite state machine). They used the PROMELA [65] (Process Meta Language) to model a shopping smart contract (SSC) and then the model is verified by the model checking SPIN [65] to verify the correctness and important properties of this model.

3.3. Runtime verification

In the formal methods areas, runtime Verification is a computing system analysis based on observing executions of a system that allow it to extract information from a running system and using it to detect the system behaviors satisfying or violating certain correctness properties [66]. Generation of a monitor from specifications and analyze system traces against the generated monitor present the principal aspect of runtime verification. The input of RV system are a system and a set of properties expressed in a formal specification language. Runtime verification process composed from 3 stages [67]. First, a monitor generated from proprieties. The monitor takes events produced by a running system and checks it against the satisfaction of property. Second, the studying system is instrumented, then events are generated to be used by the monitor. This steps often called the instrumentation of the system. Finally, the system's execution is analyzed by the monitor. Most of the applications in runtime verification have been focused on the dynamic analysis of software. Now we present you the runtime verification tool for smart contracts called CONTRACTLARVA [68].

3.3.1. CONTRACTLARVA

In [68] a new concept in formal verification of smart contracts is applied by using the runtime verification tool "LARVA" [69]. The authors propose "ContractLARVA" to ensure that all execution paths followed at runtime satisfy the required specification [68]. In this approach, different ways to react to violations are supported. An approach based on a stake-placing strategy is also supported. In such approach, any party that can potentially violate the contract pays in a stake before running the contract, which will be given to aggrieved parties in case of a violation, but returned to the original owner if the contract terminates without violations [68]. These are automatically transformed into a safe contract which behaves just like the original one but, in addition, can identify when the specification is violated and trigger remedial behavior. All is done in a decentralized manner. This work used the subset of the DATE (Dynamic Automata with Timers and Events) without timers. The work does not show any practical application of "ContractLARVA" and all depends on the consent of the parties on the stake-placing strategy.

4. The vulnerabilities analysis in SC: vulnerabilities detection

Vulnerabilities detection aims to improve the security of smart contracts. This can be done by studying the possible vulnerabilities and verifying the smart contracts against a list of already defined and well-known vulnerabilities patterns. The vulnerabilities detection can also avoid mistakes, which makes smart contracts more secure. On other hand, the Vulnerabilities detection is inefficient to analyze the complex smart contracts. Moreover, it is easy to ignore some vulnerabilities, this is because of the non-exhaustive analysis. In the rest section, we discuss the widely used smart contracts vulnerabilities detection tools according to their verification methods. The methods we are going to explain are Symbolic Execution, Fuzzing, Abstract Interpretation. Before beginning this section, Table 1 briefly explains some vulnerabilities mentioned in this survey.

Table 1
Types of smart contracts vulnerabilities.

Vulnerabilities	Vulnerability mechanism
Re-entrancy	Recursively calling a function from a fallback function
Integer overflow/underflow	When performing addition, subtraction, or storing user input with integer variables that contain value limitations, overflow/underflow can occur
Transaction order dependency	Inconsistent transactions orders with respect to the time of invocations
Call-stack depth limit	Exceeding the limit of number of calls to a contract method
Block timestamp dependency	This vulnerability occurs when a contract uses the block timestamp as part of the conditions to perform a critical operation (sending ether) and exploited from a malicious miner
Authentication through tx.origin	This vulnerability occurs when a contract uses tx.origin for authorization, which can be compromised by a phishing attack
DoS with unbounded operations	This vulnerability is caused by improper programming with unbounded operations in a contract
Short address	EVM does not check the validity of addresses
Self-destruct/suicidal contract	Contract is susceptible to be destroyed by unauthorized users
Unchecked and failed send	Send ethers without checking the conditions
Unsecured balance	The ether balance in a contract is exposed because of the modifier public to theft by an anonymous caller
Greedy contract	Locking the contract fund or ether balance indefinitely
Prodigal contract	Leaking fund or ether balance to arbitrary users
Mishandled exceptions	When an exception is thrown in a smart contract through a call from another contract and it is not managed properly by the caller
Gas overspent	Contract code execution consumes more gas unnecessarily

4.1. Symbolic execution

Symbolic execution is a popular program analysis technique that replaces the values of program variables as symbolic expressions to discover all feasible execution paths by building the control flow graph (CFG) [70]. There are path conditions for each symbolic path and the path is feasible if its path condition is satisfactory. The resulting representation (CFG) is checked by SMT-solvers to identify and detect vulnerabilities. Most researchers in the smart contacts verification field take the symbolic execution way to find vulnerabilities. In this section, we show different tools that use symbolic execution.

4.1.1. Oyente

OYENTE is a static analysis tool that detects security vulnerabilities. Designed by Luu et al. [71], it is one of the earliest security tools for Solidity smart contracts. OYENTE tool uses symbolic execution to check for the following vulnerabilities: transaction ordering dependency, reentrancy, timestamp dependence, and unhandled exceptions. The tool takes as input the bytecode of a smart contract and a state of the Ethereum blockchain. The Ethereum global state provides the initialized (or current) values of contract variables, thus enabling more precise analysis. Oyente consists of four main components [71], namely CFGBuilder, Explorer, CoreAnalysis and Validator. An illustration of the Oyente Workflow can be seen in Fig. 9. CFGBuilder builds a control flow graph from the smart contract bytecode, where nodes are basic execution

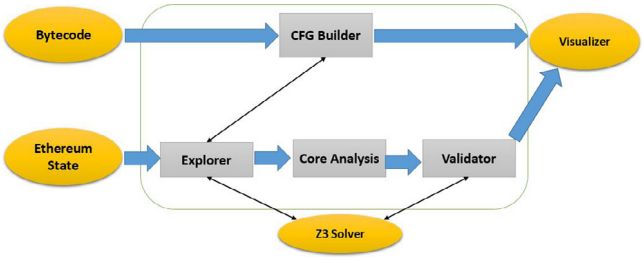

Fig. 9. Oyente workflow [71].

blocks, and edges represent execution jumps between the blocks. Then, the Explorer executes the smart contract code symbolically. The output from the Explorer is fed as the input to the CoreAnalysis component. The identified vulnerabilities are targeted to implement the logic in the CoreAnalysis module. Finally, the last component is Validator which attempts to remove false positives, and then the results are visualized to the users.

4.1.2. MAIAN

The authors of [72] built a dynamic tool to analyze contracts to find vulnerabilities across long sequences of invocations of a contract. This tool extends the Oyente approach [71]. It can find the vulnerabilities directly from the bytecode of Ethereum smart contracts, without requiring source code access. In this context, the smart contracts are divided in three categories: greedy contracts, prodigal contracts, and suicidal contracts. The greedy contracts can lock funds by being unable to send Ether, while the prodigal ones can leak Ether to a user they have never interacted with. The suicidal contracts can kill the contract or force it to execute the suicide instruction. MAIAN uses systematic techniques to find the violations of the defined specific properties of traces in smart contract executions. Each run of the contract called an invocation, may exercise an execution path in the contract code under a given input context. It has two components, symbolic analysis and concrete validation [72] as showing in Fig. 10. The symbolic analysis component takes as input

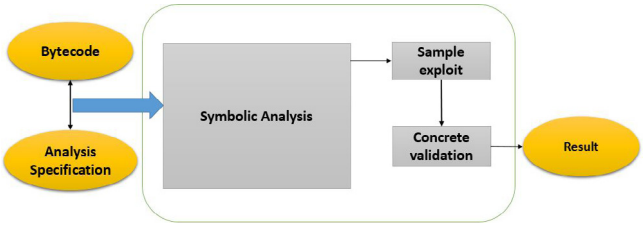

Fig. 10. Architecture of MAIAN.

the contract bytecode and analysis specifications. The specifications include the vulnerability category to search for and depth of the search space. A custom Ethereum virtual machine was implemented to facilitate the symbolic execution of SC bytecode. Ethereum virtual machine runs possible execution traces symbolically for each contract candidate until it finds a trace which satisfies a set of predetermined properties. Every execution trace takes a set of symbolic variables as its input. Once a contract is flagged, the symbolic analysis component will return concrete values for these variables. In the final step, the symbolic analysis component generated inputs, the concrete validation component takes these inputs and checks the exploit of the contract on a private fork of Ethereum blockchain. It confirms the correctness of bugs found in the candidate smart contract without affecting the state of the contract on the Ethereum blockchain [72].

4.1.3. SmartCheck

SmartCheck [73] analyzes the high-level solidity source code of the smart contract to find vulnerabilities using static analysis. It is also able to find a wide range of vulnerabilities, such as the Re-entrancy. SmartCheck uses the Extensible Markup Language (XML) syntax tree [74] to convert the solidity contract in an intermediate representation (IR). The vulnerabilities are specified as XQuery path expressions that are used to search the vulnerability patterns in the Extensible Markup Language tree by using XPath queries on the intermediate representation. The complex rules cannot be precisely described with XPath, which leads to false positives. SmartCheck is written in Java with a command-line version that is available on Github.

4.1.4. Slither

Slither [75] is a static analysis framework designed to provide granular information about smart contract code. There are four use-cases provided by Slither framework (according to [75]): (1) Automated vulnerability detection that detects smart contract bugs without user intervention. (2) Automated optimization detection that detects code optimizations. (3) Code understanding, to improve the user's knowledge of the contracts and (4) Assisted code review, which allows users to interact with Slither. Slither analyzes in a multistage procedure using static analysis. Solidity compiler generates the Solidity Abstract Syntax Tree (AST) from the contract source code; then Slither takes it as initial inputs. Slither recovers essential information in the first stage, such as the contract's inheritance graph, the control flow graph (CFG), and the list of expressions. In the second stage, the entire code of the contract is transformed into an internal representation language called SlithIR, which uses a single static assessment (SSA) [76] to facilitate the computation of a variety of code analyses. In the final stage, the information to the other modules is provided by computing a set of pre-defined analyses using code analysis. Slither allows for the application of commonly used program analysis techniques like dataflow and taint tracking.

4.1.5. Gasper

A static analysis tool named GASPER was developed by Chen et al. [77] that focus on gas costly patterns from the existing smart contracts instead of the vulnerabilities. Gasper looks for patterns such as dead code or expensive operations in loops to help contract developers reduce gas costs. The authors of [77] identified 7 gas-costly patterns. Gasper takes the bytecode as input to identify gas costly patterns. Gasper builds a control flow graph from the EVM bytecode. Using the symbolic execution, Gasper finds all the reachable code blocks in a candidate smart contract. First, Gasper disassembles its bytecode to construct the control flow graph (CFG) of the smart contract. Then, symbolic execution starts from the root node of the CFG and traverses the CFG. If a conditional jump is found, GASPER checks its feasibility by querying the Z3 SMT-solver.

4.1.6. Mythril

Mythril [78] is a static security analysis tool for Ethereum smart contracts. It analyzes the smart contracts interactively using the symbolic execution of backend LASER-Ethereum. Then, the tool visualizes the CFG with the nodes containing disassembled code and the edges being labeled by path formulas. A SMT-solver Z3 is used to exploiting the potential vulnerabilities. Mythril can detect 8 types of vulnerabilities, such as the manipulated balance vulnerability, the authentication through tx.origin vulnerability, Unchecked call, Unchecked math, and others.

4.1.7. Osiris

Osiris [79] is a static analysis tool that combines symbolic execution and taints analysis to detect integer bugs in Solidity smart contracts. It has evaluated according to a large experimental dataset containing more than 1.2 million smart contracts. The tool covers three different types of integer bugs: arithmetic bugs, truncation bugs, and signedness bugs. Taint analysis is a technique that consists of tracking the propagation of data across the control flow of a program and used by numerous integer error detection tools to reduce the number of false positives [80]. It works at bytecode level and consists of three components: symbolic analysis, taint analysis, and integer error detection (see Fig. 11). The symbolic analysis constructs a Control Flow Graph (CFG) and executes the different paths of the contract. Every executed instruction passed to the taint analysis and then to the integer error detection. The tool uses taint analysis (tracking the propagation of data across the control flow of code) to distinguish between benign and malicious overflows. The integer error detection component checks whether an integer bug is possible.

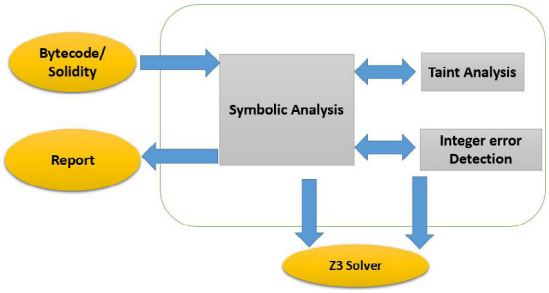

Fig. 11. Architecture of Osiris.

4.2. Abstract interpretation

Abstract interpretation formalizes the idea of abstraction of mathematical structures that aims to achieve the soundness in program analysis by giving over-approximate the semantics of a program [81]. The concrete semantics is a mathematical characterization that formalizes the set of all possible executions in all possible execution environments. In smart contracts context, abstract interpretation ignores certain instructions while executing the bytecode by translating instructions to another formalism (like DataLog [82]) and then exploring all possible executions.

4.2.1. Vandal

Vandal [83] is a security analysis framework for Ethereum smart contracts. The EVM bytecode is converted to the logic relations using an analysis pipeline as the first part. The logic relations expose data and control-flow dependencies of the bytecode. The second part is a set of logic specifications for security analysis problems. To express the logic specifications for security analysis, Vandal uses the Souffle language [84]. The analysis pipeline in Vandal consists of 5 components [83]: The scraper extracts bytecode of smart contracts on a bulk basis; the disassembler that translates bytecode into opcodes(disassemble patterns); the decompiler translates the low-level bytecode to a register transfer language and the extractor introduce the logic semantic relations by the translation of the register transfer language. Finally, the security analysis reports any possible vulnerabilities of the examined smart contracts. Vandal can detect some vulnerabilities such as unchecked send, re-entrancy, unsecured balance, destroyable contract, and use of origin problem [83].

4.2.2. Ethir

Based on the rule-based representations of the control flow graphs (CFG) produced by the OYENTE tool [71], Ehtir [85] analyze the smart contracts statically. Unlike oyente, Ethir includes all possible jump addresses. It disassembles the given bytecode and constructs a CFG. Then, The basic blocks are transformed from a stack to a register-oriented view. The control flow is represented as guarded rules to examine the conditional and unconditional jump instructions by the high-level static analyzer SACO (Static Analyzer for Concurrent Objects).

4.2.3. Securify

Securify is a static security analyzer for Ethereum smart contracts [86]. It takes Ethereum virtual machine bytecode and security properties as inputs; then it checks the smart contract behaviors with respect to a given property. A Security property consists of compliance and violation patterns written in a domain-specific language that is derived from the known attacks and the best practices. Securify's analysis [86] consists of two steps: first, it symbolically analyzes the contract's dependency graph to extract precise semantic information from the code. Then, it checks compliance and violation patterns that capture sufficient conditions to prove whether a property exists or not. In other words, Securify decompiles the Ethereum virtual machine bytecode into a stackless static-single assignment form and represents the code as DataLog [82] facts (see Fig. 12). It then infers facts from the contract, such as data and control-flow dependencies. Finally, the security patterns are coded as DataLog rules and check it against its facts.

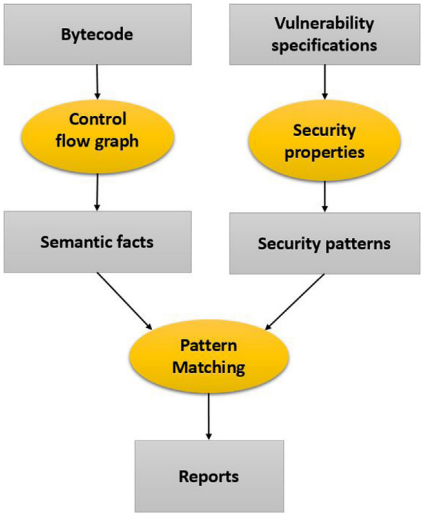

Fig. 12. Securify workflow.

4.2.4. MadMax

GRECH et al. [87] provide MadMax, a static program analysis technique to detect gas-related vulnerabilities the integer overflows vulnerabilities automatically. It is the first tool to detect so-called unbounded mass operations where a loop is bounded by a dynamic property such as the number of users, causing the contract to always run out of gas passed a certain number of users. MadMax uses the analysis pipeline and takes the EVM bytecode as input to convert it into an intermediate representation. The main MadMax analysis operates on the output of the Vandal decompiler using logic-based specifications then it encodes properties of the smart contract into Datalog [82]. According to [87], this tool can analyze all the contracts of the Ethereum blockchain in only 10 h.

4.3. Fuzzing

Fuzz testing (fuzzing) is a software testing technique that provides random data called FUZZ and used it into a computer program. Fuzz testing is used to check the vulnerability of software [88]. It is a very cost-effective testing technique. Recently many projects use fuzzing in smart contracts context to detect vulnerabilities such as ContractFuzzer [89] and ReGuard [90].

4.3.1. ContractFuzzer

A dynamic tool called ContractFuzzer [89] used Fuzz testing to detect vulnerabilities in smart contracts. This tool contains an offline EVM instrumentation tool (the responsible for instrumenting the EVM) and an online fuzzing tool (which can monitor the execution of smart contracts to extract information for vulnerability analysis). The ContractFuzzer tool analyzes the ABI interface and the bytecode of the smart contract. Then, it extracts the data types of each argument of ABI functions as well as the signatures of the functions used in each ABI function. In addition to this, ContractFuzzer performs the ABI signature analysis and indexes all smart contracts crawled from Ethereum. Then, the ContractFuzzer generates inputs for the contracts based on the application binary interface (ABI). Before the last step, the tool instruments the EVM to log smart contracts runtime behaviors and records the executed instructions during fuzzing. Finally, a logs analysis is performed to find vulnerabilities in the contract. ContractFuzzer can detect five types of smart contract vulnerabilities, such as the reentrancy vulnerability and the unchecked call return value vulnerability.

4.3.2. ReGuard

ReGuard [90], is a tool dynamic for detecting re-entrancy vulnerabilities through fuzzing. It automatically generates flagged reentrant errors. ReGuard takes solidity code or bytecode as inputs and then parses it into an intermediate representation (IR). The intermediate representation (IR) of source code is an abstract syntax tree (AST) and the bytecode is a control-flow graph. ReGuard has three main components: Contract Transformer, Fuzzing Engine, and Core Detector. ReGuard transforms the intermediate representation (IR) to C++ through the Contract Transformer. The fuzzing engine generates the random bytes iteratively using runtime coverage information as feedback. ReGuard interprets the bytes as transaction requests sent to the contract. Then, the tool executes the contract and dumps runtime trace of analysis-relevant operations. The trace is a combination of the C++ smart contract, transactions and a runtime library. Finally, The trace is passed to the Core Detector for reentrancy analysis, and a report is made if any bugs are found.

5. Discussion and future directions

After investigating a set of tools related to the verification of the smart contracts, we summarize the different characteristics in Tables 2 and 3 and then, we draw some conclusions. In Table 2 there are six model-checking based tools, five theorem proving based tools, and one runtime verification tool. Where theorem proving is unable to perform without human intervention (the high skill of users are required), model checking requires no human oversight. In contrast, theorem proving can hand complex system but model checking is not feasible for complex data path. In case of negative results, model checking generates counter-examples where theorem proving provides useful insight. Further, model checking is fully automatic while theorem proving ranges from semi-automatic to interactive. Some papers explain how runtime verification can avoid the traditional complexity problems in model checking and theorem proving. But due to the cost and gas execution of smart contracts every time we use runtime verification as well as its dynamic behavior, it is actually not an efficient practical method for smart contracts.

Table 2
Formal verification tools.

Tools	Verification methods	Type	Level	Behavior based	SC properties based	Code transformation
[29]	Model checking	Static analysis	Solidity & bytecode	✗	✓	Boogie translation
[64]	Model checking	Static analysis	Solidity code	✓	✓	PROMELA
[51]	Model checking	Dynamic analysis	bytecode	✗	✓	Not applicable
[54]	Model checking	Dynamic analysis	Solidity code	✓	✗	Not applicable
[60]	Model checking	Dynamic analysis	Smart contract as transition system	✓	✓	BIP
[30]	Abstract interpretation & Model checking	Static analysis	Solidity code	✗	✓	AST analysis
[43]	Theorem proving	Static analysis	Bytecode	✓	✓	K language
[37]	Theorem proving	Static analysis	Solidity code & bytecode	✗	✓	F* translation
[46]	Theorem proving	Dynamic analysis	Solidity code	✗	✓	Not applicable
[42]	Theorem proving	Static analysis	Bytecode	✗	✓	Control flow graph
[44]	Abstract interpretation & Theorem proving	Static analysis	Bytecode	✗	✗	Horn clauses
[68]	Runtime verification	Dynamic analysis	Solidity code	✓	✓	Not applicable

In Table 2, for F* [37], the preliminary results and the suggested approach show that it cannot handle all the Solidity syntaxes. The design of Zeus [30] is similar to VERISOL [29] in that it translates Solidity to an intermediate language and uses SMT based solvers to discharge the verification problem. Seven out twelve tools use the static analysis and the remainder tools use dynamic analysis. F* and VERISOL can take the code or the bytecode of smart contracts as inputs. Four Out twelve tools, [51,54,68] and [46] do not use the translation of smart contracts to another intermediate language. Most of the tools in Table 2 have used the model checker. The formal verification aimed to verify the correctness of smart contracts but some tools can also detect security issues like Zeus [30], VERISOL [29], F* [37] and K-EVM [43]. Most tools succeeded in some use-cases and their source codes have been published on Github (except [30,51,64] and [54]). Five out twelve tools in Table 2 based on user behavior verification.

According to Table 3, we show different tools based on the three verification methods used in vulnerabilities detection context: symbolic execution, abstract interpretation and fuzzing. Those are the most used methods to detect smart contracts vulnerabilities.

Despite the fact that symbolic execution is widely used by the vulnerabilities detection methods in smart contracts as showing in Table 3, it has some limitations. The symbolic execution cannot identify the infeasible paths and cannot proceed if the number of iterations in the loop is known. Another issue is related to the invocation of any out-of-line code or module calls.

As for abstract interpretation, it is more exhaustive than symbolic execution. Abstract interpretation-based static analysis is automatic, sound and scalable.

In fuzzing (or fuzz testing) case, the great advantage is that the test design is extremely simple, and free of preconceptions about system behavior. In contrast, Fuzzers usually find the simple bugs but not the complex one. In addition, it suffers from some limitations when doing black-box-testing. As shown in Table 3, seven out of thirteen tools

Table 3
Summary of all vulnerabilities detection tools based on: Type, level and verification methods.

Tools	Verification methods	Type	Level
Oyente	Symbolic execution	Static analysis	Bytecode
OSIRIS	Symbolic execution	Static analysis	Bytecode
MAIAN	Symbolic execution	Dynamic analysis	Bytecode
SmartCheck	Symbolic execution	Static analysis	Solidity code
Slither	Symbolic execution	Static analysis	Bytecode
Mythril	Symbolic execution	Static analysis	Bytecode
Gasper	Symbolic execution	Static analysis	Bytecode
MadMax	Abstract interpretation	Static analysis	Bytecode
Vandal	Abstract interpretation	Static analysis	Bytecode
Ethir	Abstract interpretation	Static analysis	Bytecode
Securify	Abstract interpretation	Dynamic analysis	Bytecode
ContractFuzzer	Fuzzing	Dynamic analysis	Bytecode
ReGuard	Fuzzing	Dynamic analysis	Solidity or Bytecode

Table 4
Summary of vulnerabilities detection tools based on: vulnerabilities and related attacks.

Tools	Detecting vulnerabilities	Attacks
Oyente	Re-entrancy, Exception handling, Transaction ordering, Block timestamp dependency, Call stack depth limitation	Integer overflow/under flow The DAO attack
Vandal	Re-entrancy, Unchecked and failed send, Destroyable/suicidal contract, Unsecured balance, tx.origin	The DAO attack, Parity multisig wallet attack
Ethir	Re-entrancy, Exception handling, Transaction ordering, Block timestamp dependency	The DAO attack
Securify	Exception handling, Transaction ordering, Call stack depth limitation, Unchecked and failed send, No restricted write, No restricted transfer, Non-validated arguments	Parity multisig wallet attack
Osiris	Integer bugs	Integer overflow/under flow attack
MAIAN	Call stack depth limitation, Destroyable/suicidal contract, Unsecured balance, Greedy contracts, Prodigal contracts	Parity multisig wallet attack
Smartcheck	Re-entrancy, Transaction ordering, Block timestamp dependency, Integer over/under flow, unchecked, failed send, Destroyable/suicidal contract, Unsecured balance	The DAO attack, Integer over/under flow attack
Slither	Re-entrancy, Exception handling, Transaction ordering, Block timestamp dependency	The DAO attack
Gasper	Gas costly code patterns exist	–
Madmax	Unbounded mass operations, integer overflows	The DAO attack, Integer over/under flow attack
ContractFuzzer	Re-entrancy vulnerabilities, Unchecked call	The DAO attack
ReGuard	Re-entrancy vulnerabilities	The DAO attack
Mythril	Re-entrancy vulnerabilities, Unchecked call, tx.origin, Unchecked math, manipulated balance vulnerability	The DAO attack

use the symbolic execution method. Four methods use the dynamic analysis, and the remainder tools use static analysis. Most of the tools take the smart contracts as bytecode level excepted Smartcheck [73] and ReGuard [73](ReGuard supports both solidity and bytecode level).

The Table 4 summarizes the vulnerabilities and related attacks that can be detected by the mentioned tools. More than half of the tools in Table 4 (eight tools) can detect the re-entrancy vulnerability.

To this day, and based on the existing tools the verification is successfully done only for simple smart contract. Thus a future research direction could include the development of approaches to address the complex smart contracts such as those that contain external calls or loops. We can also conclude that both correctness and security assurance verification are required to achieve the most of the smart contracts use. Thus, we suggest the combination of model checking with abstract interpretation in order to increase the performance of the verification tool. This can be efficient in the case of complex contracts and facilitate understanding the contracts which help us to avoid security issues before deploying the smart contracts on the blockchain. This hybrid method could be also part of our future research direction.

6. Conclusion

In this survey, we show a detailed overview of the smart contracts verification methods. Due to the immutable nature of distributed ledger technology on the blockchain, a smart contract should work as intended before using it. Any bugs

or errors will become permanent once published and could lead to huge economic losses. Thus, ensuring the security of smart contracts is important to achieve trust and continuity in the Blockchain-based business process execution. To avoid such problems, verification is required to check the smart contract. This verification relies on two major aspects: Security assurance and Correctness of smart contracts. We focus on the verification frameworks related to the correctness and security assurance of smart contracts, especially to the correctness based on Formal Verification and vulnerabilities detection. Based on the investigated methods, one can conclude that they verify only the simple smart contracts and not the complex ones. This motivates the need to elaborate more work on the verification of smart contracts. In the future, we will work to design and implement a new tool for verification of smart contracts that takes the benefit of both formal verification methods and vulnerabilities detection methods.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, 2008, https://bitcoin.org/bitcoin.pdf (Accessed: 01-07-2015).
[2] Y. Lu, The blockchain: State-of-the-art and research challenges, J. Ind. Inf. Integr. 15 (2019) 80–90, http://dx.doi.org/10.1016/j.jii.2019.04.002.
[3] S. Ølnes, J. Ubacht, M. Janssen, Blockchain in Government: Benefits and Implications of Distributed Ledger Technology for Information Sharing, Elsevier, 2017.
[4] H. Hassani, X. Huang, E. Silva, Banking with blockchain-ed big data, J. Manag. Anal. 5 (4) (2018) 256–275, http://dx.doi.org/10.1080/23270012.2018.1528900.
[5] S. Demirkan, I. Demirkan, A. McKee, Blockchain technology in the future of business cyber security and accounting, J. Manag. Anal. 7 (2) (2020) 189–208, http://dx.doi.org/10.1080/23270012.2020.1731721.
[6] S. Perera, S. Nanayakkara, M. Rodrigo, S. Senaratne, R. Weinand, Blockchain technology: Is it hype or real in the construction industry?, J. Ind. Inf. Integr. (2020) 100125.
[7] K.N. Griggs, O. Ossipova, C.P. Kohlios, A.N. Baccarini, E.A. Howson, T. Hayajneh, Healthcare blockchain system using smart contracts for secure automated remote patient monitoring, J. Med. Syst. 42 (7) (2018) 130.
[8] C. Zhang, Y. Chen, A review of research relevant to the emerging industry trends: Industry 4.0, iot, block chain, and business analytics, J. Ind. Integr. Manag. (2020).
[9] W. Viriyasitavat, L. Da Xu, Z. Bi, A. Sapsomboon, New blockchain-based architecture for service interoperations in internet of things, IEEE Trans. Comput. Soc. Syst. 6 (4) (2019) 739–748.
[10] Y. Lu, Blockchain and the related issues: a review of current research topics, J. Manag. Anal. 5 (4) (2018) 231–255, http://dx.doi.org/10.1080/23270012.2018.1516523.
[11] Everledger company, https://www.everledger.io.
[12] M. Dumas, R. Hull, J. Mendling, I. Weber, Blockchain technology for collaborative information systems (dagstuhl seminar 18332), Dagstuhl Rep. 8 (8) (2018) 67–129, http://dx.doi.org/10.4230/DagRep.8.8.67.
[13] M. von Rosing, S. White, F. Cummins, H. de Man, Business process model and notation - BPMN, in: The Complete Business Process Handbook: Body of Knowledge from Process Modeling to BPM, Vol. I, 2015, pp. 429–453, http://dx.doi.org/10.1016/B978-0-12-799959-3.00021-5.
[14] C.D. Ciccio, A. Cecconi, M. Dumas, L. García-Bañuelos, O. López-Pintado, Q. Lu, J. Mendling, A. Ponomarev, A.B. Tran, I. Weber, Blockchain support for collaborative business processes, Inform. Spektrum 42 (3) (2019) 182–190, http://dx.doi.org/10.1007/s00287-019-01178-x.
[15] W. Viriyasitavat, D. Hoonsopon, Blockchain characteristics and consensus in modern business processes, J. Ind. Inf. Integr. 13 (2019) 32–39.
[16] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A.D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S.W. Cocco, J. Yellick, Hyperledger fabric: a distributed operating system for permissioned blockchains, in: Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018, 2018, pp. 30:1–30:15, http://dx.doi.org/10.1145/3190508.3190538.
[17] IBMHyperledger Projects, https://www.hyperledger.org/projects/composer.
[18] L. Goodman, Tezos—a self-amending crypto-ledger white paper, 2014, https://www.tezos.com/static/papers/white_paper.pdf.
[19] Tezos documentation, https://learn.tqtezos.com.
[20] Tezos Technology, https://tezos.com.
[21] D.D. Wood, Ethereum: a secure decentralised generalised transaction ledger, 2014.
[22] O. López-Pintado, L. García-Bañuelos, M. Dumas, I. Weber, Caterpillar: A blockchain-based business process management system, in: Proceedings of the BPM Demo Track and BPM Dissertation Award Co-Located with 15th International Conference on Business Process Modeling (BPM 2017), Barcelona, Spain, September 13, 2017, 2017, http://ceur-ws.org/Vol-1920/BPM_2017_paper_199.pdf.
[23] R. Drechsler, Formal System Verification, Springer, 2018.
[24] D.A. Peled, Formal methods, in: Handbook of Software Engineering, Springer, 2019, pp. 193–222.
[25] I. Bashir, Mastering Blockchain: Distributed Ledger Technology, Decentralization, and Smart Contracts Explained, Packt Publishing Ltd, 2018.
[26] V. Gatteschi, F. Lamberti, C. Demartini, C. Pranteda, V. Santamaria, To blockchain or not to blockchain: That is the question, IT Prof. 20 (2) (2018) 62–74.
[27] M. Almakhour, L. Sliman, A.E. Samhat, W. Gaaloul, Trustless blockchain-based access control in dynamic collaboration, in: Proceedings of the 1st International Conference on Big Data and Cyber-Security Intelligence, BDCSIntell 2018, Hadath, Lebanon, December 13-15, 2018, 2018, pp. 27–33, http://ceur-ws.org/Vol-2343/paper8.pdf.
[28] M. Gelvez, Explaining the DAO exploit for beginners in solidity, 2016.
[29] S.K. Lahiri, S. Chen, Y. Wang, I. Dillig, Formal specification and verification of smart contracts for azure blockchain, 2018, CoRR abs/1812.08829 arXiv:1812.08829 http://arxiv.org/abs/1812.08829.
[30] S. Kalra, S. Goel, M. Dhawan, S. Sharma, ZEUS: analyzing safety of smart contracts, in: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018, 2018, http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-1_Kalra_paper.pdf.
[31] H. Chen, M. Pendleton, L. Njilla, S. Xu, A survey on ethereum systems security: Vulnerabilities, attacks and defenses, 2019, CoRR abs/1908.04507 arXiv:1908.04507 http://arxiv.org/abs/1908.04507.

[32] M.D. Angelo, G. Salzer, A survey of tools for analyzing ethereum smart contracts, in: IEEE International Conference on Decentralized Applications and Infrastructures, DAPPCON 2019, Newark, CA, USA, April 4-9, 2019, 2019, pp. 69–78, http://dx.doi.org/10.1109/DAPPCON.2019.00018.
[33] 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.
[34] Y. Murray, D.A. Anisi, Survey of formal verification methods for smart contracts on blockchain, in: 10th IFIP International Conference on New Technologies, Mobility and Security, NTMS 2019, Canary Islands, Spain, June 24-26, 2019, 2019, pp. 1–6, http://dx.doi.org/10.1109/NTMS.2019.8763832.
[35] J. Liu, Z. Liu, A survey on security verification of blockchain smart contracts, IEEE Access 7 (2019) 77894–77904, http://dx.doi.org/10.1109/ACCESS.2019.2921624.
[36] J. Rushby, Theorem proving for verification, in: Summer School on Modeling and Verification of Parallel Processes, Springer, 2000, pp. 39–57.
[37] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, et al., Formal verification of smart contracts: Short paper, in: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, 2016, pp. 91–96.
[38] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, et al., Kevm: A complete formal semantics of the ethereum virtual machine, in: 2018 IEEE 31st Computer Security Foundations Symposium (CSF), IEEE, 2018, pp. 204–217.
[39] J. Harrison, Theorem proving for verification (invited tutorial), in: International Conference on Computer Aided Verification, Springer, 2008, pp. 11–18.
[40] M. Nesi, A brief introduction to higher order logic and the HOL proof assistant, 2011.
[41] Y. Hu, Exploring Formal Verification Methodology for FPGA-Based Digital Systems, Sandia National Laboratories, New Mexico, California, 2012.
[42] S. Amani, M. Bégel, M. Bortin, M. Staples, Towards verifying ethereum smart contract bytecode in Isabelle/HOL, in: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, 2018, pp. 66–77.
[43] M. Sotnichek, Formal verification of smart contracts with the k framework, 2018.
[44] I. Grishchenko, M. Maffei, C. Schneidewind, EtherTrust: Sound Static Analysis of Ethereum Bytecode, Tech. Rep., Technische Universität Wien, 2018.
[45] T.C. development team, The coq proof assistant, 1999–2018, http://coq.inria.fr.
[46] Z. Yang, H. Lei, W. Qian, A hybrid formal verification system in coq for ensuring the reliability and security of ethereum-based service smart contracts, IEEE Access 8 (2020) 21411–21436, http://dx.doi.org/10.1109/ACCESS.2020.2969437.
[47] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J.K. Zinzindohoue, S.Z. Béguelin, Dependent types and multi-monadic effects in f, in: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, 2016, pp. 256–270, http://dx.doi.org/10.1145/2837614.2837655.
[48] Solidity Documentation, http://solidity.readthedocs.io.
[49] G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy, S.Z. Béguelin, Probabilistic relational verification for cryptographic implementations, in: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014, 2014, pp. 193–206, http://dx.doi.org/10.1145/2535838.2535847.
[50] C. Baier, J. Katoen, Principles of Model Checking, MIT Press, 2008.
[51] Z. Nehai, P. Piriou, F.F. Daumas, Model-checking of smart contracts, in: IEEE International Conference on Internet of Things (IThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), IThings/GreenCom/CPSCom/SmartData 2018, Halifax, NS, Canada, July 30 - August 3, 2018, 2018, pp. 980–987, http://dx.doi.org/10.1109/Cybermatics_2018.2018.00185.
[52] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, The nuxmv symbolic model checker, in: Computer Aided Verification - 26th International Conference, CAV 2014, Held As Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, 2014, pp. 334–342, http://dx.doi.org/10.1007/978-3-319-08867-9_22.
[53] M.C. Browne, E.M. Clarke, O. Grumberg, Characterizing finite kripke structures in propositional temporal logic, Theoret. Comput. Sci. 59 (1988) 115–131, http://dx.doi.org/10.1016/0304-3975(88)90098-9.
[54] T. Abdellatif, K. Brousmiche, Formal verification of smart contracts based on users and blockchain behaviors models, in: 9th IFIP International Conference on New Technologies, Mobility and Security, NTMS 2018, Paris, France, February 26-28, 2018, 2018, pp. 1–5, http://dx.doi.org/10.1109/NTMS.2018.8328737.
[55] A. Basu, M. Bozga, J. Sifakis, Modeling heterogeneous real-time components in BIP, in: Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06), Ieee, 2006, pp. 3–12.
[56] R. Sinnema, E. Wilde, Extensible access control markup language (XACML) XML media type, Internet Eng. Task Force (IETF) (2013) 1–8.
[57] C. Lattner, V. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, in: International Symposium on Code Generation and Optimization, 2004. CGO 2004., IEEE, 2004, pp. 75–86.
[58] N. Bjørner, K.L. McMillan, A. Rybalchenko, Program verification as satisfiability modulo theories., SMT@ IJCAR 20 (2012) 3–11.
[59] A. Gurfinkel, T. Kahsai, A. Komuravelli, J.A. Navas, The seahorn verification framework, in: International Conference on Computer Aided Verification, Springer, 2015, pp. 343–361.
[60] A. Mavridou, A. Laszka, E. Stachtiari, A. Dubey, Verisolid: Correct-by-design smart contracts for ethereum, in: Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers, 2019, pp. 446–465, http://dx.doi.org/10.1007/978-3-030-32101-7_27.
[61] M. Maróti, T. Kecskés, R. Kereskényi, B. Broll, P. Völgyesi, L. Jurácz, T. Levendovszky, Á. Lédeczi, Next generation (meta)modeling: Web- and cloud-based collaborative tool infrastructure, in: Proceedings of the 8th Workshop on Multi-Paradigm Modeling Co-Located with the 17th International Conference on Model Driven Engineering Languages and Systems, MPM@MODELS 2014, Valencia, Spain, September 30, 2014, 2014, pp. 41–60, http://ceur-ws.org/Vol-1237/paper5.pdf.
[62] A. Mavridou, A. Laszka, Tool demonstration: Fsolidm for designing secure ethereum smart contracts, in: Principles of Security and Trust - 7th International Conference, POST 2018, Held As Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, 2018, pp. 270–277, http://dx.doi.org/10.1007/978-3-319-89722-6_11.
[63] Y. Li, Y. Li, Z. Ma, Computation tree logic model checking based on possibility measures, Fuzzy Sets and Systems 262 (2015) 44–59.
[64] X. Bai, Z. Cheng, Z. Duan, K. Hu, Formal modeling and verification of smart contracts, in: Proceedings of the 7th International Conference on Software and Computer Applications, ICSCA 2018, Kuantan, Malaysia, February 08-10, 2018, 2018, pp. 322–326, http://dx.doi.org/10.1145/3185089.3185138.
[65] E. Mikk, Y. Lakhnech, M. Siegel, G.J. Holzmann, Implementing statecharts in PROMELA/SPIN, in: 2nd Workshop on Industrial-Strength Formal Specification Techniques (WIFT '98), October 20-23, 1998, Boca Raton, FL, USA, 1998, pp. 90–101, http://dx.doi.org/10.1109/WIFT.1998.766303.
[66] Y. Falcone, K. Havelund, G. Reger, A tutorial on runtime verification, Eng, Dependable Softw. Syst. (2013) 141–175, http://dx.doi.org/10.3233/978-1-61499-207-3-141.

[67] C. Sánchez, G. Schneider, W. Ahrendt, E. Bartocci, D. Bianculli, C. Colombo, Y. Falcone, A. Francalanza, S. Krstić, J.M. Lourenço, et al., A survey of challenges for runtime verification from advanced application domains (beyond software), Form. Methods Syst. Des. 54 (3) (2019) 279–335.
[68] J. Ellul, G.J. Pace, Runtime verification of ethereum smart contracts, in: 2018 14th European Dependable Computing Conference (EDCC), IEEE, 2018, pp. 158–163.
[69] C. Colombo, G.J. Pace, Runtime verification using LARVA, in: RV-CuBES 2017. an International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA, 2017, pp. 55–63, http://www.easychair.org/publications/paper/Jwmr.
[70] J.C. King, Symbolic execution and program testing, Commun. ACM 19 (7) (1976) 385–394.
[71] L. Luu, D. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016, 2016, pp. 254–269, http://dx.doi.org/10.1145/2976749.2978309.
[72] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, A. Hobor, Finding the greedy, prodigal, and suicidal contracts at scale, in: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018, 2018, pp. 653–663, http://dx.doi.org/10.1145/3274694.3274743.
[73] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, Y. Alexandrov, Smartcheck: Static analysis of ethereum smart contracts, in: 1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018, 2018, pp. 9–16, http://ieeexplore.ieee.org/document/8445052.
[74] A.V. Aho, R. Sethi, J.D. Ullman, Compilers, principles, techniques, Addison wesley 7 (8) (1986) 9.
[75] J. Feist, G. Grieco, 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, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019, 2019, pp. 8–15, http://dx.doi.org/10.1109/WETSEB.2019.00008.
[76] B.K. Rosen, M.N. Wegman, F.K. Zadeck, Global value numbers and redundant computations, in: Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988, 1988, pp. 12–27, http://dx.doi.org/10.1145/73560.73562.
[77] T. Chen, X. Li, X. Luo, X. Zhang, Under-optimized smart contracts devour your money, in: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017, 2017, pp. 442–446, http://dx.doi.org/10.1109/SANER.2017.7884650.
[78] B. Mueller, Smashing ethereum smart contracts for fun and real profit, HITB SECCONF Amsterdam.
[79] C.F. Torres, J. Schütte, R. State, Osiris: Hunting for integer bugs in ethereum smart contracts, in: Proceedings of the 34th Annual Computer Security Applications Conference, 2018, pp. 664–676.
[80] M. Pomonis, T. Petsios, K. Jee, M. Polychronakis, A.D. Keromytis, IntFlow: improving the accuracy of arithmetic error detection using information flow tracking, in: Proceedings of the 30th Annual Computer Security Applications Conference, 2014, pp. 416–425.
[81] P. Cousot, Formal verification by abstract interpretation, in: A. Goodloe, S. Person (Eds.), NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings, in: Lecture Notes in Computer Science, vol. 7226, Springer, 2012, pp. 3–7, http://dx.doi.org/10.1007/978-3-642-28891-3_3.
[82] J. Ullman, Principles of Database and Knowledge-Base Systems, volume Volume I-Fundamental Concepts, Computer Science Press, New York, 1988.
[83] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, B. Scholz, Vandal: A scalable security analysis framework for smart contracts, 2018, CoRR abs/1809.03981 arXiv:1809.03981 http://arxiv.org/abs/1809.03981.
[84] H. Jordan, B. Scholz, P. Subotić, Soufflé: On synthesis of program analyzers, in: International Conference on Computer Aided Verification, Springer, 2016, pp. 422–430.
[85] E. Albert, P. Gordillo, B. Livshits, A. Rubio, I. Sergey, Ethir: A framework for high-level analysis of ethereum bytecode, in: Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings, 2018, pp. 513–520, http://dx.doi.org/10.1007/978-3-030-01090-4_30.
[86] P. Tsankov, A.M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, M.T. Vechev, Securify: Practical security analysis of smart contracts, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, on, Canada, October 15-19, 2018, 2018, pp. 67–82, http://dx.doi.org/10.1145/3243734.3243780.
[87] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, Y. Smaragdakis, Madmax: Surviving out-of-gas conditions in ethereum smart contracts, in: Proceedings of the ACM on Programming Languages, Vol. 2 (OOPSLA), 2018, pp. 1–27.
[88] G. Klees, A. Ruef, B. Cooper, S. Wei, M. Hicks, Evaluating fuzz testing, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 2123–2138.
[89] B. Jiang, Y. Liu, W. Chan, Contractfuzzer: Fuzzing smart contracts for vulnerability detection, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 259–269.
[90] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, B. Roscoe, Reguard: finding reentrancy bugs in smart contracts, in: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), IEEE, 2018, pp. 65–68.