Analysis of Blockchain Smart Contracts: Techniques and Insights

Shinhae Kim
School of Computing
KAIST
Daejeon, Republic of Korea
shinhae1106@kaist.ac.kr

Sukyoung Ryu
School of Computing
KAIST
Daejeon, Republic of Korea
sryu.cs@kaist.ac.kr

***Abstract*—A blockchain records transactions among users on a public ledger. It has become front and center of the technology discussion in recent years. A piece of code deployed on a ledger and executed automatically by nodes on the network is a smart contract. While smart contracts have enabled a variety of applications on blockchain, they may contain security vulnerabilities, leading to massive research on smart contract analysis. This paper presents *the first comprehensive survey over smart contract analysis* by collecting 391 papers, extracting 67 analysis-related ones, and classifying them into three dominant topics: static analysis for vulnerability detection, static analysis for program correctness, and dynamic analysis. We further classify each topic and conclude with key insights in terms of unsolved challenges and directions in future research.**

***Index Terms*—Blockchain, smart contract, static analysis, dynamic analysis**

I. Introduction

A blockchain refers to a decentralized and public digital ledger on top of a node-connected network [1] and records transactions among users. It is a growing list of blocks, which contain transactional data and point to previous blocks. Once the network receives broadcasted transactions, mining nodes validate the transactions, create a block with them, and broadcast the block to other nodes. Each node has its own copy of the blockchain, and for a new block creation, it appends the block as a tail of its chain. This mechanism provides the immutability of transaction data. As a blockchain allows digital transactions among parties without any third-party support, various applications have appeared in diverse domains like finance [2], business [3], and IoT [4].

A smart contract [5] is a program written on the underlying ledger and has enabled such applications. Developers implement smart contracts in programming languages like Serpent [6], Solidity [7], and Vyper [8], and nodes on the network automatically execute their code. While the main usage is to transfer assets, smart contracts can express complex functionalities as well. Bartoletti and Pompianu [9] presented the popularity of smart contract applications through an empirical study on two most dominant platforms, Bitcoin [10] and Ethereum [11]. They collected 1,673,271 transactions from 834 contracts and classified them into five categories. Finance was the most popular domain on both platforms, with 1,094,437 transactions. Notary and games were the second most popular domains on Bitcoin and Ethereum, respectively. Despite its popularity, smart contracts may contain security vulnerabilities, which can lead to severe financial loss. For example, the reentrancy bug [12] of the DAO contract enabled an adversary to withdraw USD 50 million worth of cryptocurrency. Also, a vulnerability on the Parity Multisig Wallet contract [13] resulted in locking USD 146 million worth of coins. As a result, researchers have actively studied smart contract analysis.

This paper presents *the first comprehensive study over smart contract analysis* to understand the current research trends. We first collected 391 papers from various sources such as major conference proceedings and arXiv e-prints that contain the keyword "smart contract" in their titles or abstracts. Then, by examining each one closely, we extracted 67 papers relevant to smart contract analysis and classified them into three dominant topics: static analysis for vulnerability detection, static analysis for program correctness, and dynamic analysis. We explore each topic thoroughly with further classifications and provide key insights in terms of open challenges and future research directions.

II. Blockchain Smart Contracts

In 2008, Satoshi Nakamoto proposed a protocol of the first cryptocurrency called Bitcoin [14]. While Bitcoin served as a basis of smart contract development, it could not support general-purpose smart contracts, as it stored only transaction histories on blocks. Five years later, Vitalik Buterin [15] introduced the first platform for general-purpose smart contracts, Ethereum, and made blockchain applicable to various domains. Solidity is the most dominant programming language for Ethereum smart contracts, and EVM bytecode is the actual code deployed on the platform. This section explains the characteristics of Ethereum, Solidity, and EVM bytecode, as most research is based on them.

a) Ethereum: Ethereum is a public and distributed computing platform based on blockchain [16]. It has Ether as its underlying cryptocurrency and operates Ethereum Virtual Machine (EVM) [11]. The machine has its own instruction set, which is Turing-complete and much more expressive than the scripting language for Bitcoin [17]. It contains an operand stack where the instructions execute upon, a byte-addressed

volatile memory, and a permanent storage with 32-byte key-value pairs. Also, it executes under a resource mechanism called gas and consumes a certain amount of gas for executing each instruction. Nodes in Ethereum assign gas prices when submitting transactions, and miners get rewards depending on the gas prices and the total amount of gas required for execution. While Ethereum has a secure design [11], it has the scalability issue as every node holds the entire history of transactions.

b) Solidity: Solidity is contract-oriented, statically-typed, and motivated by C++, Python, and JavaScript. It supports various features such as inheritance, user-defined types, and libraries. "Contracts in Solidity are similar to classes in object-oriented languages. Each contract can contain declarations of State Variables, Functions, Function Modifiers, Events, Struct Types and Enum Type." [7] State variables are global variables, and contract storage stores their values permanently. Modifiers work as guarded conditions of executions. A library is a special contract, and once deployed at a specific address, the calling contracts pass their contexts to execute the library code through `DELEGATECALL`. Solidity provides not only elementary types like `address`, but also user-defined types. While the language is actively used, it contains various vulnerable features [18, 19, 20], and the compiler has released vulnerability patches constantly [21].

c) EVM bytecode: While Ethereum supports a variety of programming languages such as Solidity, their compilers translate the code to EVM bytecode [11]. The bytecode has 114 opcodes and expresses execution logic in stack-based representations. An EVM bytecode consists of creation code and runtime code. The former code corresponds to constructor instruction sequences, and EVM executes it only once when a client invokes the contract creation transaction. The latter code is the one actually deployed on blockchain with an address and its own storage. Every time a node invokes a function by submitting a transaction, Ethereum directs the program counter to the corresponding function logic by checking the initial four bytes in the input data. EVM instructions do not have annotated types and operate on 32-byte words.

III. Analysis of Blockchain Smart Contracts: Techniques

The extracted papers either statically or dynamically analyzed smart contracts. Especially, the portion of static analyzers was roughly twice as large as that of dynamic analyzers, so we classify them into two groups based on their purposes: vulnerability detection and program correctness. Figure 1 shows the overall classification where a ladybug indicates an analyzer that detects "exploitable vulnerabilities," of which we explain the definition later.

A. Static Analysis for Vulnerability Detection

Smart contracts have suffered from a variety of vulnerabilities and exploits, resulting in financial losses [18]. As a result, researchers have adopted several techniques and developed static analyzers that detect vulnerabilities.

Symbolic execution, which simulates concrete executions with symbolic value inputs, is the most dominant technique for vulnerability detection in EVM bytecode. Oyente [22] targets transaction-ordering dependence, timestamp dependence, mishandled exceptions, and re-entrancy problems. Maian [23] analyzes multiple invocations of smart contracts to detect three types of trace vulnerabilities: greediness, prodigality, and suicidality. Chen et al. [24] identified seven gas-costly programming patterns and developed Gasper to detect them. teEther [25] automatically generates transactions that can exploit the given contracts. sCompile [26] reports program paths containing monetary transactions, called critical paths, since most vulnerabilities are related to money transfer [27]. Torres et al. [28] collected existing honeypots and classified them to disclose common techniques to trick users. Based on their findings, they implemented HoneyBadger and detected real-world honeypot contracts. Li and Long [29] focused on "standard violation errors" that token contracts deviate from the ERC-20 and ERC-721 standards. They developed Solar, which reports violation-triggering transactions.

Abstract interpretation [30] is a technique that soundly approximates program semantics. Several analyzers leveraged the technique for EVM bytecode as it can explore all possible executions. Securify [31] checks whether the input property from a user holds or not in a given smart contract. It works in two phases: extraction of semantic facts and checking against compliance and violation patterns. MadMax [32] identifies three gas-related vulnerabilities: unbounded mass operations, non-isolated external calls, and integer overflows. Vandal [33] leverages Soufflé [34] to produce results for custom vulnerability queries. Grishchenko et al. [35] developed a static analyzer called EtherTrust based on their previous work of EVM semantics formalization [36] and detected re-entrancy vulnerability. Zeus [37] detects seven previously-reported vulnerabilities in Solidity contracts by taking user policies.

The popularity of machine learning techniques has led to adopting them for smart contract domain as well. Tann et al. [38] adopted long-short term memory (LSTM) to detect suicidal, prodigal, and greedy contracts. They trained the LSTM model by labeling MAIAN-detected contracts as vulnerable, and the model showed a very high test accuracy. Huang [39] used a convolution neural network to detect compiler bugs such as DelegateCallReturnValue [7] automatically. While the previous two papers targeted EVM bytecode, Liu et al. [40] developed a tool called Ether* based on N-gram language model to audit Solidity contract.

Other than the ones mentioned above, there exist a variety of other techniques used for vulnerability detection. More than a half of such papers targeted Solidity contracts. Tikhomirov et al. [41] developed SmartCheck, which detects a wide range of vulnerabilities using XPath Query [42]. Bansal et al. [43] presented an automatic generation tool of commutativity conditions called Servois and detected concurrency-related vulnerabilities. SIF [44] is a framework where programmers can easily derive several analysis tools like an assertion checker to detect arithmetic vulnerabilities. Slither [45] is another

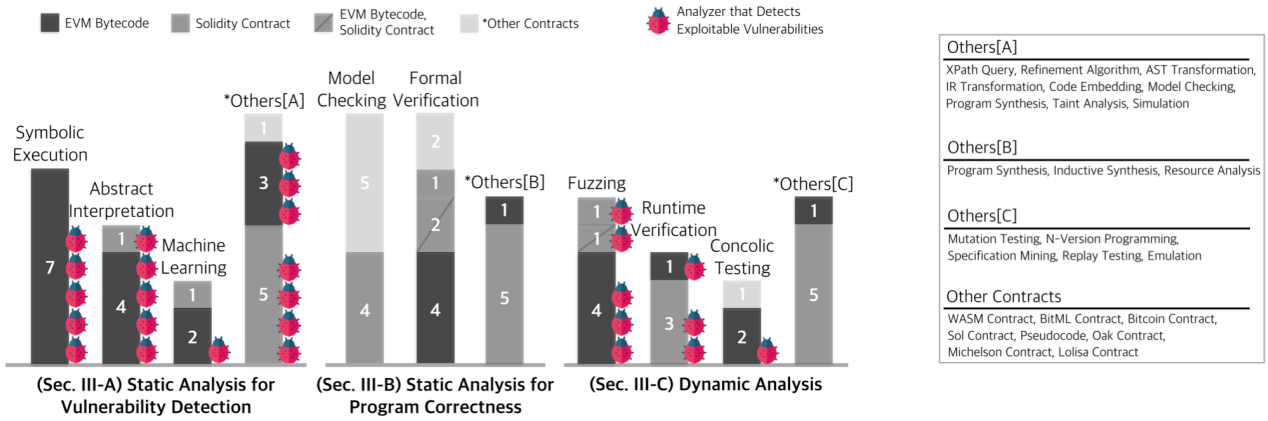

Fig. 1: Classification overview of smart contract analyzers

framework to support a variety of applications such as a vulnerability detector and a code visualizer. SMARTEMBED [46] embeds Solidity contracts into vectors and apply similarity checking to find code clones. It reports code fragments that are similar to known bugs as bugs.

The other techniques detected vulnerabilities in EVM bytecode or WebAssembly (WASM) [47]. Muller [48] developed Mythril, a model checking tool consisting of a few modules, each detecting one of the following vulnerabilities: unprotected `SUICIDE` instructions, unprotected Ether send, reentrancy, and `DELEGATECALL`. SMARTSCOPY [49] leverages program synthesis based on the victim contracts' ABI and automatically generates exploit contracts. EASYFLOW [50] is an arithmetic overflow detector based on taint analysis. By interpreting input transactions, it classifies contracts into potential overflow, overflow, or safe. EVulHunter [51] is the first detector for EOS WASM contracts [52] to which malicious users can transfer fake EOS tokens. It simulates WASM execution in a virtual environment on demand.

B. Static Analysis for Program Correctness

As smart contracts are immutable once deployed, guaranteeing their correctness is particularly more important than ordinary programs. Researchers have come up with diverse techniques to verify properties of smart contracts.

The most dominant technique is model checking, which takes a formal model of a finite set of states and automatically proves whether the input specification complies with the model. Solidity contracts have been mostly the target of modeling. Abdellatif and Brouscmiche [53] presented smart contract verification in their execution environments. Particularly, they reported probabilities of each possible scenario by hackers. Chatterjee et al. [54] analyzed payoffs expected to gain from interacting with other contracts, since a payoff out of an acceptable range implies a vulnerability. VeriSolid [55] takes two inputs from users: contracts as transition systems and properties in CTL. It transforms the systems multiple times and leverages the nuXmv model checker [56] to verify the properties. Wang et al. [57] presented VERISOL, which leverages the CORRAL model checking tool [58] to verify whether customer contracts in the Azure Blockchain [59] semantically conform to their policies.

Model checking has been mostly on but not limited to Solidity contracts. It has verified the contracts written in other languages as well. Atzei et al. [60] and Bartoletti and Zunino [61] verified the liquidity of BitML contracts. The property guarantees the existence of a path to decrease the contract balances. Andrychowicz et al. [62] designed a framework based on timed automata model to prove the security of Bitcoin contracts. Given a blockchain model, participants' models, and a specification under question, the UPPAAL model checker [63] automatically completes the verification. Shishkin [64] presented verification of Sol contracts, a subset of Solidity, along with a formal method to specify state and trace properties. van der Meyden [65] leveraged the MCK model checker [66] and verified whether the pseudocode of an escrow contract correctly exchanges the two parties' assets.

Formal verification is another dominant technique for program correctness. It mathematically proves the correctness of a given formal model against a formal specification. Amani et al. [67] presented an Isabelle/HOL-based framework to prove functional correctness of bytecode. Hirai [68] leveraged the same theorem prover to verify the Deed contract [69]. The verifier discussed in the papers by Park et al. [70] and Chen et al. [71] is a derivative from the K framework and verified high-profile smart contracts. While the aforementioned papers verified EVM bytecode, Bhargavan et al. [72] and SAFEVM [73] can verify both Solidity contracts and EVM bytecode. Bhargavan et al. designed a verification framework for both source-level and low-level properties using F* [74]. SAFEVM transforms input contracts into C programs with SV-COMP annotations [75]. Any verifiers compatible with the annotations can work as back-ends. SOLC-VERIFY [76] leverages modular analysis and SMT solving to automatically verify input properties of Solidity contracts.

Formal verification was not only for Solidity and EVM bytecode, but also for Oak and Michelson languages. Annenkov and Spitters [77] presented an embedding technique of Oak

contracts into Coq and verified a crowdfunding contract with respect to several properties. Mi-Cho-Coq [78] is a Coq-based framework for verification of Michelson smart contracts. A case study proved two properties of a multisig contract.

Other techniques used for program correctness are program synthesis, inductive synthesis and resource analysis [79]. FSolidM [80] provides a graphical user interface to specify smart contracts as finite state machines (FSMs) and automatically synthesizes their corresponding Solidity code. Also, it supports a set of security and functionality plugins. Mavridou and Laszka [81] demonstrated the tool by adopting a blind auction smart contract as a use case. Suvorov and Ulyantsev [82] presented another program synthesis technique of correct-by-construction Solidity contracts. Once users provide LTL specifications, the technique generates FSMs conformed to the specifications and then transforms them to corresponding Solidity contracts. TXSC [83] is a framework where users provide smart contracts with the supported primitives. It then translates them into concurrency bug-free Solidity contracts. VERISMART [84] takes a Solidity contract, inductively synthesizes transaction invariants, and verifies the safety of arithmetic operations. GASTAP [85] analyzes EVM bytecode and infers the gas bounds of all public contract functions. While the gas bounds are parametric to several factors such as contract states, the tool can prevent out-of-gas exceptions in advance.

C. Dynamic Analysis

Dynamic analysis of smart contracts has advantages over static analysis that it does not require any source code and has low false positive rates. Researchers have adopted a variety of dynamic techniques for both vulnerability detection and program correctness.

The most common technique is fuzzing [86], which executes target programs with random inputs. ContractFuzzer [87] detects seven kinds of vulnerabilities such as exception bypassing and reentrancy. It consists of an offline EVM instrumentation tool and an online fuzzing tool. ETHRACER [88] fuzzes input bytecode with function call sequences to detect "event-ordering" bugs. The sequences follow the "happens-before" relations, and the tool reports sequence pairs that result in different contract states. HARVEY [89] is a greybox fuzzer extended with two methods: input prediction and on-demand transaction sequence. BRAN [90] is another greybox fuzzer combined with an online static analysis to guide the fuzzer towards target locations. While the aforementioned fuzzers execute on EVM bytecode, ReGuard [91] works on both Solidity and bytecode. It reported seven reentrancy bugs out of five smart contracts from Etherscan. ContraMaster [92] is a Solidity-level grey-box fuzzer to detect seven kinds of vulnerabilities. It fuzzes with transaction sequences, validates the results based on its test oracle, and reports an exploiting script in case of a violation.

Runtime verification [93] has been another common technique for dynamic smart contract analysis. It monitors contract executions at runtime to detect or protect from malicious behaviors. Grossman et al. [94] defined two notions, dynamically effectively callback free (DECF) execution and statically effectively callback free (SECF) object. As non-DECF executions imply re-entrancy vulnerabilities, they implemented a runtime monitor called ECFChecker to detect such executions in Solidity contracts. DappGuard [95] is a system for live Solidity contract monitoring that detects transactions with known attacks and protects them prior to consequent exploits. It works in two phases, knowledge acquisition and active monitoring for detection, and handles various vulnerabilities including keeping secrets and reentrancy. Colombo et al. [96] suggested an extension of a Solidity runtime verification tool called CONTRACTLARVA [97] to support violation recovery as well. The conceptual idea is based on the notions of checkpointing and compensations. Sereum [98] is a runtime monitoring tool that prevents deployed EVM bytecode from reentrancy attacks.

The last common technique is concolic testing [99], which combines symbolic evaluation with concrete execution. Annotary [100] is a concolic execution framework to detect vulnerabilities in EVM bytecode. The framework supports inter-procedural as well as inter-transactional analysis while extracting on-chain values for concrete execution. Manticore [101] is another concolic execution framework that can analyze both native binaries and EVM bytecode. FEther [102] is a Coq-based interpreter, one of the components on the FSPVM-E [103] system under active development. By combining symbolic evaluation and theorem proving-based concrete execution, it executes and verifies Lolisa contracts.

The remaining papers of dynamic analysis have adopted various techniques. Wu et al. [104] and ContractMut [105] proposed that mutation testing can aid developers to prevent mistakes in developing smart contracts. Wu et al. defined 15 mutation operators and showed how effective they are in identifying vulnerabilities in Solidity contracts. ContractMut mutates Solidity contracts with similar operators while tracking gas consumption to filter out out-of-gas execution. Breidenbach et al. [106] leveraged N-version programming [107] and presented the Hydra Framework, which induces hackers to disclose Solidity contract bugs with an incentive mechanism. Guth et al. [108] presented the first specification mining [109] for Solidity contracts. It expresses smart contracts as finite automata and provides an intuitional understanding of their behaviors. ContractVis [110] extracts transaction histories of a verified Solidity contract and re-executes them on Truffle [111] to analyze its transparency. Chen et al. [112] pointed out that the current gas mechanism is not defensive to DoS attacks. They implemented an emulation-based framework for EVM bytecode and demonstrated that current gas costs are not proportional to consumption.

While the static or dynamic analyzers that detect vulnerabilities target different vulnerability types, we extract the ones already exploited or known to be exploitable and call them "exploitable vulnerabilities." Figure 2 shows the list of exploitable vulnerabilities along with the analyzers that detect them. The referenced sources for checking exploitability are [18, 19, 113].

Vulnerability	Description	Symbolic Execution					Abstract Interpretation						Machine Learning		Others[A]							Fuzzing							Runtime Verification		Concolic Testing
		[22]	[23]	[24]	[25]	[26]	[31]	[32]	[33]	[35]	[37]	[38]	[41]	[43]	[44]	[45]	[48]	[49]	[50]	[87]	[88]	[89]	[91]	[92]	[94]	[95]	[98]	[100]
Arbitrary Writes to Storage	Any users can overwrite values in storage slots [31].	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	●
Block Variable Dependency	Contracts use block hash or timestamp to generate random values [18, 19].	●	○	○	○	○	○	○	○	○	●	○	●	○	○	○	○	●	○	●	○	○	○	●	○	○	○	○
Careless Handling of `tx.origin`	`tx.origin` returns the address of a transaction-initiated account [27].	○	○	○	○	○	○	○	●	○	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
`DELEGATECALL`	`DELEGATECALL` passes the caller context to its callee [19].	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○	●	○	○	○	○	○	○	○	○
Inability to Transfer Ether	Contracts can receive Ether but do not provide a way for users to withdraw [23].	○	●	○	○	●	●	○	○	○	○	●	●	○	○	●	○	○	○	●	○	○	○	●	○	○	○	○
DoS: External Call Failures	A callee may throw an exception. For example, `send` assigns 2300 units of gas only [18].	○	○	○	○	○	○	●	○	○	●	○	●	○	○	○	○	○	○	●	○	○	○	●	○	●	○	○
Unexpected Fallback Function	Fallback executions take place in case of `send` or no matching function signature [18].	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○
Integer Division	Dividing integers rounds down the quotient, always resulting in an integer [41].	○	○	○	○	○	○	○	○	○	○	○	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○
Integer Overflow/Underflow	Arithmetic operations can result in overflow/underflow [113].	○	○	○	○	○	○	●	○	○	●	○	●	○	●	○	○	●	●	○	○	○	○	●	○	●	○	○
Private Information Leakage	Private fields do not guarantee confidentiality as transaction records are public [18].	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○
Reentrancy	An external malicious callee can re-enter the body of its caller [18, 19, 20].	●	○	○	○	○	●	○	●	●	●	○	●	○	○	●	●	●	○	●	○	○	●	●	●	●	●	○
Usage of `call.value`	`call.value` forwards all remaining gas [41].	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Return Value Bypassing	In case of exceptions, the built-in functions `send` and `call` return `false` [18, 19].	●	○	○	○	○	●	○	●	○	●	○	●	○	○	○	○	●	○	●	○	○	○	●	○	●	○	○
Transaction Order Dependency	Miners can control the order of deploying transactions in a block [18, 19, 20].	●	○	○	○	○	●	○	○	○	●	○	○	●	○	○	○	○	○	○	●	○	○	○	○	●	○	○
Transfer to an Orphan Address	Accounts lose Ether if they send it to a non-existing address [26].	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Unauthorized Ether Withdrawal	Invoking Ether transfers is possible by any users [31].	○	●	○	●	○	●	○	●	○	○	●	○	○	○	●	●	●	○	○	○	○	○	○	○	○	○	○
Unauthorized `SELFDESTRUCT`	Any users can destroy a contract if no authorization exists for `SELFDESTRUCT` [31].	○	●	○	●	●	○	○	●	○	○	●	○	○	○	●	●	●	○	○	○	○	○	○	○	○	○	○
Function without Visibility	Functions without visibility declarations are public by default [19].	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●
Unbounded Mass Operations	Iterating a loop can cause an out-of-gas exception if its iteration number is too high [32].	○	○	●	○	○	○	●	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○

Fig. 2: Exploitable vulnerabilities of smart contracts and analyzers that detect them

IV. Analysis of Blockchain Smart Contracts: Insights

This section discusses unsolved challenges and possible directions in future research of smart contract analysis.

A. Open Challenges

1) Ambiguity in Contract Behaviors: As smart contracts execute on a decentralized system and interact with other parties, several ambiguities in their behaviors may arise. One possible case is unexpected states caused by miner-dependent transaction ordering. As disclosed in previous papers [18, 19, 20], miners can freely choose the order of deploying transactions, and contract states become non-trivial. Also, many contracts include logic to interact with off-chain sources to obtain necessary data. However, the sources can provide unexpected data [114], and even if they do not, the decentralized environment may result in nodes receiving different data. As smart contracts quite often behave differently depending on received data, their behaviors can be non-deterministic. The last factor of behavior ambiguity is code opacity that most contracts do not have source code available. If interacting contracts are opaque, contract behaviors become ambiguous.

Existing papers adopted a timed automata model and the BIP framework [115] to model smart contract behaviors in their execution environments. Andrychowicz et al. [62] proposed a framework that models each party into a timed automaton and verified two Bitcoin contracts. Also, Abdellatif and Brouscmiche [53] leveraged the BIP framework and expressed each party as a combination of finite-state automata. While the two papers provided novel modeling approaches, they still do not enable code-level analysis in the presence of various parties such as users and miners.

2) Unstable Language Semantics: A smart contract is yet another program, implemented in a programming language. However, existing languages either lack language specifications or undergo continuous semantic changes due to security pitfalls. Solidity has an official documentation [7], but has frequent version updates [21] to address such pitfalls. One specific example is an uninitialized storage pointer addressed in the version 0.5.0 release. An uninitialized variable of array or user-defined types was used to point to the first storage slot by default and might have corrupted the storage. From version 0.5.0 onwards, the issue does not arise, but many contracts already compiled by previous versions still remain deployed on the blockchain, which may be vulnerable.

3) Lack of Clear Property Definition: Although many attempts have focused on detecting vulnerable contracts, they lack clear property definitions for vulnerabilities. The patterns that previous security tools defined do not necessarily imply vulnerabilities. For example, Oyente just checks the existence of two traces with different Ether flows and reports it as the transaction order vulnerability. Defining safety/liveness properties is a mandatory task for precise static analysis

of smart contracts. The decentralized and multi-transactional environments make the task non-trivial, and Sergey and Hobor [116] suggested that the analogy with concurrent objects in shared memory can give an intuition.

MAIAN [23] is one early approach to define trace properties of target vulnerabilities. They defined greediness, prodigality, and suicidality of contracts in terms of safety/liveness properties and enabled analysis of multiple contract invocations. Another approach is a set of security properties that Grishchenko et al. [36] defined. They defined call integrity, atomicity, and independence of miners in terms of hyper/safety properties and presented how known attacks violate the properties. However, both work cover only a limited range of possible security threats and cannot detect other kinds of vulnerabilities.

B. Directions in Future Research

From the observations above, we suggest directions in future research of smart contract analysis.

1) Language Design: Researchers have been designing new languages amenable for analysis. Following the trend, it seems promising to design a new language with fully specified semantics. An example language is Scilla [117], a smart contract intermediate-level language designed both as an independent framework and a compilation target. It models contracts as communicating automata and provides its embedding into the Coq [118] proof assistant. The separation between in-contract computation and explicit communication enabled a principled semantic specification and formal reasoning over contract behaviors. As translating Solidity to Scilla is possible, analyzing Solidity programs is possible by analyzing the translated Scilla programs. Although the current syntax supports a small subset of Solidity, it showed how designing a new language design can aid smart contract analysis.

2) Type-Based Approaches: Enriching a language with an expressive type system is another way to analyze smart contracts. It would be especially beneficial in smart contract context, because currently popular languages do not support strong type checking, although most of them are statically-typed. For example, Solidity programs compiled before the version 0.5.0 allow explicit conversions of contracts with the address type and any other contract types, because the previous version compilers do not check whether they adhere to their target types. While adopting a strong type system for smart contracts is still in its early stage, Pettersson and Ebstroom [119] first showed how dependent and polymorphic types can enable smart contract analysis. Specifically, through implementing the concepts in Idris [120], they statically detected the unexpected state vulnerability and prevented the privacy violation vulnerability. Another type-based approach was Lolisa [121], a large subset of Solidity. It has a stronger static type system than Solidity for enhanced type safety.

3) Machine Learning: Following the popularity of machine learning, researchers have also started to adopt learning models. Tann et al. [38] adopted a LSTM model to detect trace vulnerabilities. Huang [39] trained a CNN model [122] by translating a large set of Ethereum bytecode into RGB color code, and the model automatically analyzes input bytecode to report potential compiler bugs. Liu et al. [40] parsed contracts into a list of tokens and leveraged an N-gram language model to audit smart contracts automatically. Machine learning techniques may improve the scalability and accuracy of smart contract analysis research.

4) Rising Platforms and Languages: One observation from the survey is that most research focused on Ethereum for platform and Solidity and/or EVM bytecode for language. However, the blockchain community keeps evolving very fast, and the most popular platform and language at one time can soon turn out-dated. For example, Hyperledger Fabric [123] set up by the Linux Foundation is gaining its popularity as a viable Ethereum alternative. It provides a set of JavaScript-based tools for developers to implement smart contracts more easily and efficiently. Another rising platform is EOS [52], which employs the proof-of-stack consensus algorithm with EOS tokens. Serpent [6] is a popular language for Ethereum developers because the syntax is very similar to Python [124]. Many other platforms and languages keep appearing as well. Since blockchain movement is fast and has no standardization, research on such platforms and languages seems promising.

V. CONCLUSION

Blockchain is gaining its popularity with many applications, along with the support of smart contracts. However, smart contracts contain several security drawbacks, which led to massive analysis research. This paper presented the first comprehensive survey over smart contract analysis by collecting 391 papers, extracting 67 relevant ones, and classifying them into three dominant topics with further classifications: static analysis for vulnerability detection, static analysis for program correctness, and dynamic analysis.

Based on the comprehensive study, we discussed unsolved challenges and directions in future research. The biggest challenge is ambiguity in program behaviors arisen from transaction ordering dependency, reliance on off-chain sources, and code opacity. Unstable semantics of programming languages and lack of clear property definitions for vulnerabilities are two other unsolved challenges. We propose that designing new languages or type systems is a promising research direction, as linguistic supports can ease smart contract analysis. Also, adopting machine learning techniques in this domain is at an early stage with large potentials. Lastly, research on rising platforms and scripting languages seems promising, considering that blockchain is still evolving very fast. We believe that our survey can serve as not only a checkpoint of massive research on smart contract analysis but also a promising guidance of future research.

ACKNOWLEDGMENT

This work was supported by National Research Foundation of Korea (NRF) (Grants NRF-2017R1A2B3012020 and 2017M3C4A7068177) and the Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government (MSIT) (2018-0-00251).

REFERENCES

[1] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An overview of blockchain technology: Architecture, consensus, and future trends," in *Proceedings of the IEEE International Congress on Big Data*, Dec. 2017.

[2] N. Lindner. Applications of blockchain to financial services: Three banking use cases. [Online]. Available: https://finsia.com/insights/news/news-article/2018/05/10/applications-of-blockchain-to-financial-services-three-banking-use-cases

[3] J. Rampton. Five applications for blockchain in your business. [Online]. Available: https://execed.economist.com/blog/industry-trends/5-applications-blockchain-your-business

[4] ConsenSys. 5 incredible blockchain IoT applications. [Online]. Available: https://blockgeeks.com/5-incredible-blockchain-iot-applications/

[5] S. Nosikov. What are smart contracts? [Online]. Available: https://www.cryptoninjas.net/what-are-smart-contracts/

[6] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi. A programmer's guide to Ethereum and Serpent. [Online]. Available: https://mc2-umd.github.io/ethereumlab/docs/serpent_tutorial.pdf

[7] Solidity official documentation. [Online]. Available: https://solidity.readthedocs.io/en/v0.5.1/

[8] Vyper official documentation. [Online]. Available: https://vyper.readthedocs.io/en/latest/

[9] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: Platforms, applications, and design patterns," in *Proceedings of the International Conference on Financial Cryptography and Data Security*, Apr. 2017.

[10] Bitcoin news and technology source. [Online]. Available: https://www.bitcoin.com

[11] Ethereum project. [Online]. Available: https://www.ethereum.org

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

[13] S. Palladino. The parity wallet hack explained. [Online]. Available: https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7

[14] S. Baghla. Origin of Bitcoin: A brief history from 2008 crisis to present times. [Online]. Available: https://www.analyticsindiamag.com/origin-bitcoin-brief-history/

[15] A. Barkley. Vitalik Buterin and Ethereum: Background and history. [Online]. Available: https://cryptodaily.co.uk/2018/12/vitalik-buterin-and-ethereum-background-and-history

[16] Cryptotvplus. Blockchain business. [Online]. Available: https://cryptotvplus.com/ethereum/

[17] Bitcoin wiki: Script. [Online]. Available: https://en.bitcoin.it/wiki/Script

[18] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts," in *Proceedings of the International Conference on Principles and Security and Trust*, Apr. 2017.

[19] A. Manning. Solidity security: Comprehensive list of known attack vectors and common anti-patterns. [Online]. Available: https://blog.sigmaprime.io/solidity-security.html

[20] Ethereum smart contract best practices: known attacks. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/known_attacks/

[21] GitHub: Solidity version releases. [Online]. Available: https://github.com/ethereum/solidity/releases

[22] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the ACM Conference on Computer and Communications Security*, Oct. 2016.

[23] I. Nikolić, A. Kolluri, I. Sergey, P. Savena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the Annual Computer Security Applications Conference*, Dec. 2018.

[24] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*, Feb. 2017.

[25] J. Krupp and C. Rossow, "teEther: Gnawing at Ethereum to automatically exploit smart contracts," in *Proceedings of the USENIX Security Symposium*, Aug. 2018.

[26] J. Chang, B. Gao, H. Xiao, J. Sun, and Z. Yang, "sCompile: Critical path identification and analysis for smart contracts," Aug. 2018. [Online]. Available: https://arxiv.org/abs/1808.00624

[27] J. Valaska. Summary of the common smart contracts vulnerabilities. [Online]. Available: https://nethemba.com/summary-of-the-common-smart-contracts-vulnerabilities/

[28] C. F. Torres, M. Steichen, and R. State, "The art of The scam: Demystifying honeypots in Ethereum smart contracts," Feb. 2019. [Online]. Available: https://arxiv.org/abs/1902.06976

[29] A. Li and F. Long, "Detecting standard violation errors in smart contracts," Dec. 2018. [Online]. Available: https://arxiv.org/abs/1812.07702

[30] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 1977.

[31] P. Tsankov, A. Dan, D. Draschsler-Cohen, A. Gervais, F. Bunzil, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2018.

[32] N. Grech, M. Kong, A. Jurisevic, L. Brent, and B. Scholz, "MadMax: Surviving out-of-gas conditions in Ethereum smart contracts," in *Proceedings of the Object-Oriented Programming, Systems, Languages & Applications*, Nov. 2018.

[33] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," Sep. 2018. [Online]. Available: https://arxiv.org/abs/1809.03981

[34] GitHub: The Souffle project. [Online]. Available: https://github.com/souffle-lang/souffle

[35] I. Grishchenko, M. Maffei, and C. Schneidewind, "Foundations and tools for the static analysis of Ethereum smart contracts," in *Proceedings of the International Conference on Computer Aided Verification*, Jul. 2018.

[36] ——, "A semantic framework for the security analysis of Ethereum smart contracts," in *Proceedings of the International Conference on Principles of Security and Trust*, Apr. 2018.

[37] S. Kalra, S. Goel, and M. Dhawan, "Zeus: Analyzing safety of smart contracts," in *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2018.

[38] W. J.-W. Tann, X. J. Han, S. S. Gupta, and Y.-S. Ong, "Towards safer smart contracts: A sequence learning approach to detecting security threats," Nov. 2018. [Online]. Available: https://arxiv.org/abs/1811.06632

[39] T. H.-D. Huang, "Hunting the Ethereum smart contract: Color-inspired inspection of potential attacks," Jul. 2018. [Online]. Available: https://arxiv.org/abs/1807.01868

[40] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun, "S-gram: Towards semantic-aware security auditing for Ethereum smart contracts," in *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering*, Sep. 2018.

[41] S. Tikhomirov, E. Voskresenskaya, and I. Ivanitskiy, "SmartCheck: Static analysis of Ethereum smart contracts," in *Proceedings of the International Workshop on Emerging Trends in Software Engineering on Blockchain*, May 2018.

[42] XPath tutorial. [Online]. Available: https://www.w3schools.com/xml/xpath_intro.asp

[43] K. Bansal, E. Koskinen, and O. Tripp, "Automatic generation of precise and useful commutativity conditions (extended version)," in *Proceedings of the Tools and Algorithms for Construction and Analysis of Systems*, Apr. 2018.

[44] C. Peng, S. Akca, and A. Rajan, "SIF: A framework for solidity code instrumentation and analysis," May 2019. [Online]. Available: https://arxiv.org/abs/1905.01659

[45] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," Aug. 2019. [Online]. Available: https://arxiv.org/abs/1908.09878

[46] Z. Gao, V. Jayasundara, L. Jiang, X. Xia, D. Lo, and J. Grundy, "SmartEmbed: A tool for clone and bug detection in smart contracts through structural code embedding," Aug. 2019. [Online]. Available: https://arxiv.org/abs/1908.08615

[47] WebAssembly. [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly

[48] B. Muller, "Smashing Ethereum smart contracts for fun and real profit," in *Proceedings of the Hack In The Box Security Conference*, Apr. 2018.

[49] Y. Feng, E. Torlak, and R. Bodik, "Precise attack synthesis for smart contracts," Feb. 2019. [Online]. Available: https://arxiv.org/abs/1902.06067

[50] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan, and Z. Chen, "EASYFLOW: Keep Ethereum away from overflow," Nov. 2018. [Online]. Available: https://arxiv.org/abs/1811.03814

[51] L. Quan, L. Wu, and H. Wang, "EVulHunter: Detecting fake transfer vulnerabilities for EOSIO's smart contracts at Webassembly-level," Jun. 2019. [Online]. Available: https://arxiv.org/abs/1906.10362

[52] Katalyse.io. EOS platform - what you should know. [Online]. Available: https://cryptodigestnews.com/eos-platform-what-you-should-know-58da830d2aa8

[53] T. Abdellatif and K.-L. Brouscmiche, "Formal verification of smart contracts based on users and blockchain behaviors models," in *Proceedings of the International Workshop on Blockchains and Smart Contracts*, Feb. 2018.

[54] K. Chatterjee, A. K. Goharshady, and Y. Velner, "Quantitative analysis of smart contracts," in *Proceedings of the European Symposium on Programming*, Apr. 2018.

[55] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "VeriSolid: Correct-by-design smart contracts for Ethereum," Jan. 2019. [Online]. Available: https://arxiv.org/abs/1901.01292

[56] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv symbolic model checker," in *Proceedings of the International Conference on Computer Aided Verification*, Jul. 2014.

[57] Y. Wang, S. K. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, and I. Naseer, "Formal specification and verification of smart contracts for Azure blockchain," Dec. 2018. [Online]. Available: https://arxiv.org/abs/1812.08829

[58] A. Lal, S. Qadeer, and S. K. Lahiri, "A solver for reachability modulo theories," in *Proceedings of the International Conference on Computer Aided Verification*, Jul. 2012.

[59] Microsoft Azure blockchain. [Online]. Available: https://azure.microsoft.com/en-us/solutions/blockchain/

[60] N. Atzei, M. Bartoletti, S. Lande, N. Yoshida, and R. Zunino, "Developing secure Bitcoin contracts with BitML," May 2019. [Online]. Available: https://arxiv.org/abs/1905.07639

[61] M. Bartoletti and R. Zunino, "Verifying liquidity of Bitcoin contracts," May 2019. [Online]. Available: https://arxiv.org/abs/1905.07639

[62] M. Andrychowicz, S. Dziembvowski, D. Malinowski, and L. Mazurek, "Modeling Bitcoin contracts by timed automata," in *Proceedings of the International Conference on Formal Modeling and Analysis of Timed Systems*, Sep. 2014.

[63] Uppaal model checker. [Online]. Available: http://www.uppaal.org

[64] E. Shishkin, "Debugging smart contract's business logic using symbolic model-checking," Dec. 2018. [Online]. Available: https://arxiv.org/abs/1812.00619

[65] R. van der Meyden, "On the specification and verification of atomic swap smart contracts," Nov. 2018. [Online]. Available: https://arxiv.org/abs/1811.06099

[66] P. Gammie and R. van der Meyden, "MCK: Model checking the logic of knowledge," in *Proceedings of the International Conference on Computer Aided Verification*, Jul. 2004.

[67] S. Amani, M. Begel, M. Bortin, and M. Staples, "Towards verifying Ethereum smart contract bytecode in Isabelle/HOL," in *Proceedings of the ACM SIGPLAN International Conference on Certified Programs and Proofs*, Jan. 2018.

[68] Y. Hirai. Formal verification of deed contract in Ethereum name service. [Online]. Available: https://yoichihirai.com/deed.pdf

[69] Maurelian. Explaining the Ethereum namespace auction. [Online]. Available: https://medium.com/the-ethereum-name-service/explaining-the-ethereum-namespace-auction-241bec6ef751

[70] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Rosu, "A formal verification tool for Ethereum VM bytecode," in *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Nov. 2018.

[71] X. Chen, D. Park, and G. Rosu, "Language-independent approach to smart contracts verification," in *Proceedings of the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, Nov. 2018.

[72] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, "Short paper: Formal verification of smart contracts," in *Proceedings of the ACM Workshop on Programming Languages and Analysis for Security*, Oct. 2016.

[73] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "SAFEVM: A safety verifier for Ethereum smart contracts," Jun. 2019. [Online]. Available: https://arxiv.org/abs/1906.04984

[74] A higher-order effectful language designed for program verification. [Online]. Available: https://www.fstar-lang.org/#talks

[75] International competition on software verification. [Online]. Available: https://gitlab.com/sosy-lab/sv-comp

[76] A. Hajdu and D. Jovanović, "solc-verify: A modular verifier for Solidity smart contracts," Jul. 2019. [Online]. Available: https://arxiv.org/abs/1907.04262

[77] D. Annenkov and B. Spitters, "Towards a smart contract verification framework in Coq," Jul. 2019. [Online]. Available: https://arxiv.org/abs/1907.10674

[78] Bruno Bernardo and Raphaël Cauderlier and Zhenlei Hu and Basile Pesin and Julien Tesson, "Mi-Cho-Coq, a framework for certifying Tezos smart contracts," Sep. 2019. [Online]. Available: https://arxiv.org/abs/1909.08671

[79] A. Flores-Montoya and R. Hähnle, "Resource analysis of complex programs with cost equations," in *Proceedings of the Asian Symposium on Programming Languages and Systems*, Nov. 2014.

[80] A. Mavridou and A. Laszka, "Designing secure Ethereum smart contracts: A finite state machine based approach," in *Proceedings of the International Conference on Financial Cryptography and Data Security*, Mar. 2018.

[81] ——, "Tool demonstration: FSolidM for designing secure Ethereum smart contracts," Feb. 2018. [Online]. Available: https://arxiv.org/abs/1802.09949

[82] D. Suvorov and V. Ulyantsev, "Smart contract design meets state machine synthesis: Case studies," Jun. 2019. [Online]. Available: https://arxiv.org/abs/1906.02906

[83] V. Zakhary, D. Agrawal, and A. E. Abbadi, "Transactional smart contracts in blockchain systems," Sep. 2019. [Online]. Available: https://arxiv.org/abs/1909.06494

[84] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "VeriSmart: A highly precise safety verifier for Ethereum smart contracts," Aug. 2019. [Online]. Available: https://arxiv.org/abs/1908.11227

[85] E. Albert, P. Gordillo, A. Rubio, and I. Sergey, "Running on fumes–preventing out-of-gas vulnerabilities in Ethereum smart contracts using static resource analysis," Nov. 2018. [Online]. Available: https://arxiv.org/abs/1811.10403

[86] J. Li, B. Zhao, and C. Zhang, "Fuzzing: A survey," *Cybersecurity*, vol. 2, no. 2, Dec. 2018.

[87] B. Jiang, Y. Liu, and W. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2018.

[88] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, "Exploiting the laws of order in smart contracts," Oct. 2018. [Online]. Available: https://arxiv.org/abs/1810.11605

[89] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," May 2019. [Online]. Available: https://arxiv.org/abs/1905.06944

[90] ——, "Targeted greybox fuzzing with static lookahead analysis," May 2019. [Online]. Available: https://arxiv.org/abs/1905.07147

[91] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "ReGuard: Finding reentrancy bugs in smart contracts," in *Proceedings of the IEEE/ACM International Conference on Software Engineering: Companion*, Jun. 2018.

[92] H. Wang, Y. Li, S.-W. Lin, C. Artho, L. Ma, and Y. Liu, "Oracle-supported dynamic exploit generation for smart contracts," Sep. 2019. [Online]. Available: https://arxiv.org/abs/1909.06605

[93] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, 2009.

[94] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2018.

[95] T. Cook, A. Latham, and J. H. Lee, "DappGuard: Active monitoring and defense for Solidity smart contracts," 2017. [Online]. Available: https://courses.csail.mit.edu/6.857/2017/project/23.pdf

[96] C. Colombo, J. Ellul, and G. J. Pace, "Contracts over smart contracts: Recovering from violations dynamically," in *Proceedings of the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Nov. 2018.

[97] GitHub: ContractLarva: Runtime verification of Solidity smart contracts. [Online]. Available: https://github.com/gordonpace/contractLarva

[98] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," Dec. 2018. [Online]. Available: https://arxiv.org/abs/1812.05934

[99] K. Sen, "Concolic testing," in *Proceedings of the International Conference on Automated Software Engineering*, 2007.

[100] K. Weiss and J. Schütte, "Annotary: A concolic execution system for developing secure smart contracts," Jul. 2019. [Online]. Available: https://arxiv.org/abs/1907.03868

[101] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," Jul. 2019. [Online]. Available: https://arxiv.org/abs/1907.03890

[102] Z. Yang and H. Lei, "FEther: An extensible definitional interpreter for smart-contract verifications in Coq," Oct. 2018. [Online]. Available: https://arxiv.org/abs/1810.04828

[103] Z. Yang, H. Lei, and W. Qian, "A hybrid formal verification system in Coq for ensuring the reliability and security of Ethereum-based service smart contracts," Feb. 2019. [Online]. Available: https://arxiv.org/abs/1902.08726

[104] H. Wu, X. Wang, J. Xu, W. Zou, L. Zhanga, and Z. Chen, "Mutation testing for Ethereum smart contract," Aug. 2019. [Online]. Available: https://arxiv.org/abs/1908.03707

[105] P. Hartel and R. Schumi, "Gas limit aware mutation testing of smart contracts at scale," Sep. 2019. [Online]. Available: https://arxiv.org/abs/1909.12563

[106] L. Breidenbach, P. Daian, F. Tramer, and A. Juels, "Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts," in *Proceedings of the USENIX Security Symposium*, Aug. 2018.

[107] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Proceedings of the International Symposium on Fault-Tolerant Computing*, Jun. 1995.

[108] F. Guth, V. Wüstholz, M. Christakis, and P. Müller, "Specification mining for smart contracts with automatic abstraction tuning," Jul. 2018. [Online]. Available: https://arxiv.org/abs/1807.07822

[109] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2002.

[110] P. Hartel and M. van Staalduinen, "Truffle tests for free – replaying Ethereum smart contracts for transparency," Jul. 2019. [Online]. Available: https://arxiv.org/abs/1907.09208

[111] Truffle Suite: Sweet tools for smart contracts. [Online]. Available: https://www.lazenca.net/display/TEC/04.Concolic+execution

[112] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang, "An adaptive gas cost mechanism for Ethereum to defend against under-priced DoS attacks," in *Proceedings of the International Conference on Information Security Practice and Experience*, Dec. 2017.

[113] DASP - top 10. [Online]. Available: https://dasp.co

[114] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town Crier: An authenticated data feed for smart contracts," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2016.

[115] S. Yovine. BIP: Language and tools for component-based construction. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=173E4B326BBB9227C884DEAACEF72323?doi=10.1.1.543.7558&rep=rep1&type=pdf

[116] I. Sergey and A. Hobor, "A concurrent perspective on smart contracts," in *Proceedings of the International Conference on Financial Cryptography and Data Security*, Apr. 2017.

[117] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao, "Safer smart contract programming with Scilla," in *Proceedings of the Object-Oriented Programming, Systems, Languages & Applications*, Oct. 2019.

[118] The Coq proof assistant. [Online]. Available: https://coq.inria.fr

[119] J. Pettersson and R. Ebstroom, "Safer smart contracts through type-driven development," Master's thesis, Chalmers University of Technology, 2016.

[120] A language with dependent types. [Online]. Available: https://www.idris-lang.org

[121] Z. Yang and H. Lei, "Lolisa: Formal syntax and semantics for a subset of the Solidity programming language," Mar. 2018. [Online]. Available: https://arxiv.org/abs/1803.09885

[122] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *Proceedings of the International Conference on Engineering and Technology*, 2017.

[123] A blockchain framework by the Linux foundation. [Online]. Available: https://www.hyperledger.org/projects/fabric

[124] C. Seberino. Serpent: Introduction to the best Ethereum classic smart contract language. [Online]. Available: https://ethereumclassic.github.io/blog/2017-02-10-serpent/