Smart Contract Vulnerability Detection Technique: A Survey

Peng Qian[*†], Zhenguang Liu[*†], Qinming He[*], Butian Huang[†], Duanzheng Tian[†], Xun Wang[†]

[*]College of Computer Science and Technology, Zhejiang University, Hangzhou, China

[†]School of Computer and Information Engineering, Zhejiang Gongshang University, Hangzhou, China

Abstract—Smart contract, one of the most successful applications of blockchain, is taking the world by storm, playing an essential role in the blockchain ecosystem. However, frequent smart contract security incidents not only result in tremendous economic losses but also destroy the blockchain-based credit system. The security and reliability of smart contracts thus gain extensive attention from researchers worldwide. In this survey, we first summarize the common types and typical cases of smart contract vulnerabilities from three levels, *i.e., Solidity code layer, EVM execution layer, and Block dependency layer*. Further, we review the research progress of smart contract vulnerability detection and classify existing counterparts into five categories, *i.e.,* formal verification, symbolic execution, fuzzing detection, intermediate representation, and deep learning. Empirically, we take 300 real-world smart contracts deployed on Ethereum as the test samples and compare the representative methods in terms of accuracy, F1-Score, and average detection time. Finally, we discuss the challenges in the field of smart contract vulnerability detection and combine with the deep learning technology to look forward to future research directions.

Index Terms—Blockchain, Ethereum, smart contract, vulnerability detection, automated tool.

I. INTRODUCTION

BLOCKCHAIN has become one of the most prominent technologies in the past few years, attracting worldwide attention [1], [2]. Essentially, a blockchain is a distributed shared transaction ledger, which is maintained by all the participant nodes in the blockchain network and is restricted by the consensus mechanism [3]. With the characteristics of decentralization, tamper-proof, and irreversibility, blockchain is being endowed the ability to reform the inherent mode of traditional industries, making breakthroughs in many fields, such as health care [4]–[6], copyright protection [7]–[9], supply chain [10]–[12], energy grid [13]–[15], and Internet of Things [16]–[18].

Smart contract [19], [20], as one of the most successful applications of blockchain, has raised considerable enthusiasm in industry and academia. The concept of smart contracts can be traced back to 1994, which was first proposed by Nick Szabo [21]. However, there is no available execution environment provided for smart contracts at that time until the emergence of blockchain technology. A smart contract is a computer protocol running on the blockchain, which is written by the *Turing-complete* language, typically Solidity. So far, tens of thousands of smart contracts have been deployed on various blockchain platforms, *e.g.,* Ethereum [20], EOS [22], VNT Chain [23], and the number is still growing rapidly.

Unfortunately, with the increasing number of smart contracts, security issues are undesirably emerging. On one hand, the code security problems of smart contracts may inevitably be introduced during code development. On the other hand, a smart contract deployed on the public blockchain is usually exposed in an open environment, making it easy to be attacked by hackers. Furthermore, due to the immutable and irreversible of smart contracts, we can only watch the funds flow into the attacker's package and are unable to interrupt or prevent the contract execution when an attack occurs.

According to the statistics from Bcsec [24] and Slowmist [25], the economic losses caused by security issues in smart contracts have exceeded billions of dollars. For instance, in June 2016, hackers utilized the reentrancy vulnerability of the DAO (decentralized autonomous organization) contract [26] to steal around 60 million dollars worth of Ether (the digital currency of Ethereum). In July 2017, due to the delegatecall vulnerability of the Parity Multi-Sig Wallet contract [27], Ether worth nearly 300 million dollars was frozen. In April 2018, malicious attackers exploited the integer overflow vulnerability of the Beauty Chain contract [28] to issue an unlimited number of BEC tokens, leading to the evaporation of BEC value to zero. In May 2019, Binance Exchange [29] was compromised by hackers, resulting in the theft of more than 7,000 BTC. In recent months, smart contract games, *e.g.,* FarmEOS, Playgames, LuckBet, EOSPlaystation [30]–[33], have also suffered from attacks to a different extent, losing a total of nearly one million dollars. To summarize, the security problems of smart contracts not only lead to enormous financial losses but also destroy the fundamental trust based on blockchain applications. Therefore, effective security analysis and vulnerability detection methods for smart contracts are essential before their deployments.

The reasons why smart contracts are particularly susceptible to attacks can be summarized in the following four aspects. **(1)** Current programming languages and tools for smart contracts (*e.g.,* Solidity) are still nascent and primitive. Smart contracts written in such programming languages are relatively more difficult to check. Especially a smart contract is allowed to interact with external contract functions or interfaces, which may lead to repeated external invocations and unexpected security vulnerabilities. **(2)** Since smart contract developers cannot fully understand the basic execution logic of novel programming languages and audit tools, there may not be
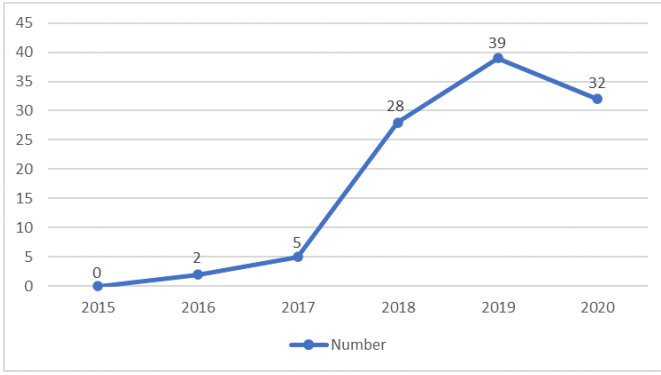
Fig. 1. The tendency change of literature in the area of smart contract security analysis and vulnerability identification

able to foresee all possible states and environments that a contract encounters in the future. This further makes the written contracts *error-prone*. **(3)** In traditional programs, developers are able to modify bugs when they discover security problems. However, different from conventional programs, smart contracts and their states are stored in an immutable blockchain network so that smart contracts are unalterable. Once deployed, there is no way to update or modify the corresponding code of a smart contract unless manipulates more than 51% computing power of the blockchain network (almost impossible). Therefore, this may make the contracts deployed on the blockchain network exist potential security issues but cannot be patched. **(4)** Since smart contracts host digital assets (such as Bitcoin [34] and Ether) worth millions of dollars, which makes them attractive to many malicious attackers. High profits drive attackers to exploit the vulnerabilities in smart contracts to steal money. In summary, compared to other software or application, smart contracts are susceptible to security vulnerabilities and malicious attacks, which further highlights the importance of smart contract security protection.

In this survey[1], we extensively investigate relevant works that propose a new method or theory in the area of smart contract security analysis and bug detection. These researches are summarized from authoritative databases, such as Web of Science, IEEE Xplore, and SpringerLink. Up to September 2020, there are more than one hundred relevant pieces of literature. It can be observed from Fig. 1 that the number of related works increased rapidly in the past three years.

The remainder of this survey is organized as follows. Section II classifies the smart contract vulnerabilities into three levels and restores five typical cases. Section III reviews the research progress of smart contract vulnerability detection from five aspects, *viz.* formal verification, symbolic execution, fuzzing detection, intermediate representation, and deep learning. Section IV compares and analyzes the automation level, open-source nature, and performance (*e.g.,* accuracy, F1-Score, average detection time) of various methods. Section V dis-

---

[1]This manuscript is the English translation version of our paper published in *Ruan Jian Xue Bao/Journal of Software, 22, 33(8).*

cusses the shortcomings of existing smart contract bug detection methods, and looks forward to future research challenges and directions in smart contract vulnerability identification.

II. SMART CONTRACT BUG CLASSES AND CASES

With the popularity of smart contracts and related decentralized applications (DAPP) [19], [35]–[37], the derived security vulnerabilities of smart contracts are being discovered and exploited by more and more malicious attackers [38]–[41]. Compared with traditional software programs, the security problems caused by smart contracts are more complicated, and the analysis and verification of smart contract vulnerabilities tend to be more difficult.

Ethereum [20] is the most popular and influential open source public blockchain platform, thanks to it holding the largest number of smart contracts and excellent DAPPs. Dune Analytics [42] reported that there are more than two million smart contracts deployed on the Ethereum platform in March 2020. Moreover, according to the proportion statistics of the DAPP market in the first two quarters of 2020, Ethereum DAPP accounted for 82% of the total created value, of which 80% belong to high-risk industries such as gambling and games. In particular, Ethereum smart contracts present an extraordinary variety of security vulnerabilities, which have caused tremendous economic losses of more than one billion dollars. More importantly, most of the typical vulnerability events (*e.g.,* The DAO) can be traced from Ethereum. Smart contracts of other platforms usually take Ethereum as the exemplar. In practice, developers and researchers regard Ethereum smart contracts as their primary research targets for security analysis and program verification. Therefore, we select Ethereum smart contracts as an example to analyze the specific security vulnerabilities and cases.

Technically, Ethereum smart contract vulnerabilities can be divided into three levels, *i.e.,* Solidity code layer, EVM execution layer, and Block dependency layer [43]. In this section, we introduce fifteen kinds of Ethereum smart contract vulnerabilities, while the specific vulnerability types and illustrations are listed in Table I.

A. Solidity Code Layer

(1) Reentrancy. Generally speaking, due to the atomicity and sequentiality of program execution, a new command will not be performed until the end of the current process. However, smart contracts do not comply with this rule. Malicious attackers are able to re-enter the called function during the current program execution [44]. Similar to most programming languages, smart contracts engage in cross-function or cross-contract invocations to process business logic. But the difference is that smart contracts usually involve sensitive operations, *e.g.,* money deposit or transfer.

Furthermore, due to the default settings of smart contracts, the transfer operation will inevitably trigger the *fallback* mechanism in the *recipient* contract. When a smart contract performs a cross-contract operation of transferring money, attackers may capture such external invocation and perform

TABLE I
SMART CONTRACT VULNERABILITY CLASSIFICATION AND EXPLANATION. '—' DENOTES NOT APPLICABLE

Vulnerability Level	Vulnerability Type	Vulnerability Definition	Related Attack	Security Issue
Solidity code layer	Reentrancy	The *fallback* function has recursive calls to external contracts	The DAO Attack	Unable to store and protect contract tokens
	Integer Overflow/Underflow	Value is out of the defined integer type range	Beauty Chain Integer Overflow Attack	Integer range error
	Access Control	Function or variable access is restricted to public type	—	Arbitrarily calls to function or variable
	Mishandled Exception	Return value and type are not checked after function call	The DAO Attack, KoET Attack	Exception handling failed
	Denial of Service	Unexpected revert; *Gas* exceeds upper limit; Unprotected `owner` account	KoET attack	Token frozen; unable to store and protect contract tokens
	Type Mismatch	Variable type definition error	—	Unable to store and protect contract tokens
	Unknown Function Call	The *fallback* function is triggered by calling unknown functions or transferring money	The DAO Attack	Unable to store and protect contract tokens
	Ether Frozen	Unauthorized use of contract self destruction	Parity Multi-Sig Wallet Attack	Inappropriate contract or function access
EVM execution layer	Short Address	Contract address fails to satisfy requirement (length is less than 20 digits)	—	Unable to store and protect contract tokens
	Ether Loss	Wrong or empty contract address	—	Unable to store and protect contract tokens
	Call-Stack Overflow	Exceed the upper limit of contract invocations	—	Buffer overflow
	Transaction Origin Use	Use `tx.origin` for smart contract authentication	—	Inappropriate contract or function access
Block dependency layer	Timestamp Dependency	Assign block timestamp to variables	—	Fail to use secure random numbers
	Block Dependency	Assign block-related parameters to variables	—	Fail to use secure random numbers
	Transaction Order Dependency	Inconsistent transaction sequence	—	Race conditions

some malicious operations. For example, an attacker designs malicious code in its *fallback* function, which can recursively re-enter a *victim* contract to call the *transfer* function to steal Ether. We consider that the reentrancy vulnerability is considered as an external transfer invocation that can call back to itself through a chain of calls. Reentrancy vulnerability has resulted in the notorious security incident in the history of smart contracts (*i.e.,* The DAO Attack [26]), which not only led to the losses of nearly 60 million dollars but also caused the *hard-fork* [43] of Ethereum.

(2) Integer Overflow/Underflow. Integer overflow is a common vulnerability in many programs, usually divided into overflow and underflow. There are three types of integer overflow vulnerabilities in smart contracts, *i.e.,* multiplication overflow, addition overflow, and subtraction underflow. In the source code of smart contracts, integers are treated as fixed-size and unsigned integer types, and the value of integer variables is limited to the range of the defined type. Obviously, if an integer variable exceeds a certain range, an integer overflow error will occur.

Ethereum smart contracts are written in high-level languages, such as *Solidity*, which supports the integer range from *uint8* to *uint256*. For example, if a number v is of type *uint8*, its value is stored as a 8-bits unsigned number ranging from 0 to $2^8 - 1$. If the value out of this range is assigned to a

variable of *uint8* type, Ethereum virtual machine (EVM) will automatically truncate the high digits. Different from other programs, the losses caused by integer overflow vulnerability in smart contracts are enormous and irreparable. For example, the integer overflow vulnerability was exploited by attackers who copy BEC tokens indefinitely, leading to the value of BEC evaporating to zero [28]. Currently, to prevent the integer overflow of smart contracts, developers are not only required to check the code manually but also employ the *SafeMath* [45] library to verify the arithmetic logic.

(3) Access Control. The fundamental reason for the access control vulnerability is that the access permissions of functions in the contract are not clearly or carefully checked, thereby allowing malicious attackers to utilize functions or variables that should not be accessed by them. Access control vulnerability is usually reflected in two levels: 1) Code Level. There are four types of access restrictions in smart contract functions and variables, namely *public, private, external,* and *internal*; 2) Logic Level. A modifier is usually used to constrain the access rights of functions in smart contracts, such as *onlyOwner* and *onlyAdmin*. Functions without modifier restrictions show that anyone has the right to access and manipulate them, which may cause the key functions to be manipulated by malicious attackers, thus endangering the security of smart contracts.

(4) Mishandled Exception. Exception handling is one of the

most important mechanisms to improve program reliability and robustness. In smart contracts, there are three kinds of exceptions. 1) *Gas Exhausted*. Here, *Gas* means the cost of deploying or executing a smart contract. When the *Gas* was used up, an exception will be thrown. 2) *Call-Stack Overflow*. When the number of invocations exceeds the maximum settings of EVM, a call-stack overflow exception will be triggered. 3) *Exception Statement*. There is an exception handling instruction, *e.g., throw* in the execution statement.

In general, a smart contract handles the abnormal behavior through rolling back, namely terminates the current contract execution, restores to the previous state, and returns an error identifier, *i.e., false*. However, since there is no unified method for handling exceptions, the *caller* contract may not be able to obtain the exception information from the *callee* contract. For example, when an exception occurs in a sub-call of smart contracts, it will be propagated to its superior automatically. Nevertheless, some underlying function invocations (e.g., `send`, `delegatecall`) only return *false* without throwing an exception. Therefore, it is not safe to judge whether the contract is successfully executed based on if the exception is thrown. We consider that the return value should be checked strictly when calling the underlying functions, and the exception should be handled consistently in the meantime.

(5) Denial of Service. Denial of service (DoS) [46] is a common vulnerability of Ethereum smart contracts. Attackers usually exploit such vulnerability to destroy the original logic and consume additional resources, e.g., *Gas* and *Ether*, making the contract unable to provide normal services for a while or forever. There are usually three types of DoS attacks against smart contracts.

- DoS attack launched through (unexpected) revert. When the state update of the smart contract depends on the execution results of an external function, the smart contract will be susceptible to the DoS attack once the failure of external function execution is not handled in time.
- DoS attack launched through the upper limit of gas. Each block in Ethereum has the upper limit of gas. A transaction initiated by the contract will be blocked as long as the cost of gas exceeds this limit. Therefore, even if there is no malicious attack, a smart contract may also have problems due to being out of the gas limit. More seriously, if an attacker maliciously manipulates the cost of gas so that the gas reaches the limit unintentionally, the transaction process of the contract will end in failure.
- DoS attack launched through the `owner` account of contracts. Most smart contracts have an `owner` account, which has the ability to control the contract. If the `owner` account is not protected, it is probably manipulated by attackers, which may further make the contract in danger (*e.g.,* Ether is frozen permanently).

(6) Type Mismatch. Solidity is a strongly typed programming language, which can automatically check whether there is a type mismatch in the program. For example, if the value of a *string* type is assigned to an *integer* variable, a type
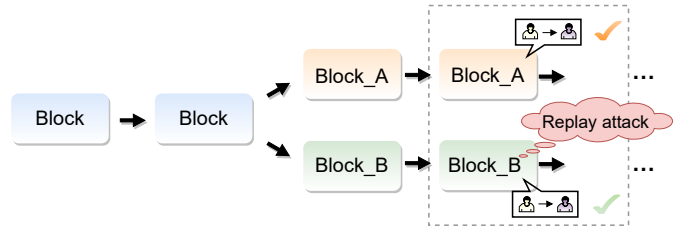

Fig. 2. A simplified illustration of the replay attack

mismatch will occur. In smart contracts, however, even if the type mismatch in some cases, the contract cannot raise an exception during program execution. Generally, developers default that the contract has already checked whether the type is matched in the program, thereby ignoring the manual audit and leading to unexpected vulnerability.

(7) Unknown Function Call. Similar to most programming languages, a smart contract ensures the uniqueness of functions by matching the function name and parameter number. When a contract calls a function in an external contract, if the function name and parameter number fail to match any function in the callee contract, the fallback function in this contract will be triggered automatically [46]. At this time, a security problem may occur unexpectedly, if the malicious operation designed by the attacker is hidden in the fallback function.

(8) Ether Frozen. Transfer operation is one of the important and unique abilities of smart contracts which means the smart contract can receive or transfer Ether from/to other contracts. It is worth mentioning that some contracts do not need to implement the *transfer* function themselves. Instead, they can utilize `delegatecall` to call a transfer function in an external contract to achieve the transfer operation. However, if the called external transfer function has self-destruction operations such as *Self-destruct* and *Suicide*, Ether on the current contract (which calls this *transfer* function through `delegatecall`) is likely to be frozen due to such a self-destruction operation executed.

(9) Replay Attack. Replay attack [47] was discovered after the Ethereum *hard-fork*. Since the address, private key, algorithm, and transaction format are the same before the *hard-fork*, transactions initiated on another chain are also valid. As shown in Fig. 2, a user who received a certain number of tokens from someone else through one ledger could switch to another ledger, which causes the transaction replicated, and an identical amount of tokens will be fraudulently transferred to the user's account again.

B. EVM Execution Layer

(1) Short Address. The length of a contract address is 20 digits, which is defined by the Ethereum ABI specification [48]. When the length of a smart contract address is less than a regular one, EVM will fill zero automatically so that the length of the address is equal to 20. This may leave a large room for malicious attackers. For example, the attacker deliberately inputs a short address, which triggers the EVM to take the

```
1  contract Victim{
2      address public owner;
3      constructor(address _owner){
4          owner = _owner;
5      }
6      function withdraw(address _recipient)
7  public {
8          require(tx.origin == owner);
9          _recipient.transfer(this.balance);
10     }
11 }
12
```
```
1  contract Attacker {
2      Victim victim;
3      address attacker;
4      constructor(Victim _victim,address
5  _attacker){
6          victim = _victim;
7          attacker = _attacker;
8      }
9      function ()  payable{
10         victim.withdraw(attacker);
11     }
12 }
```

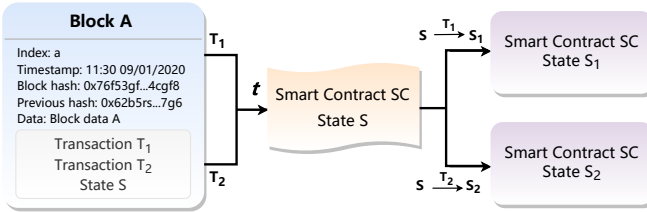Fig. 3. A simplified example of the `tx.origin` vulnerability

Fig. 4. An example of the transaction ordering dependency

missing coding digits from the next parameter (*i.e., Ether amount*) to complete the address, and then fill the end of the whole string of binary codes with 0. This indicates that the parameter of *Ether amount* has been shifted to the left by one byte. If the transfer operation is performed at this time, the contract may be transferred out to the attacker more than the actual Ether that should be forwarded.

(2) Ether Loss. When a smart contract transfers Ether, the recipient contract address needs to be specified and standardized. Assuming this address is an independent empty address and is not associated with any other users or contracts, it probably causes the Ether to lose forever once Ether is transferred to such a contract address.

(3) Call-Stack Overflow. When a smart contract calls an external function or executes a self-call, this will increase the call-stack depth of the contract. In the Ethereum virtual machine, the upper limit of the call stack is 1,024. Therefore, if an attacker designs a series of nested calls, it probably causes the call-stack overflow, which further leads to unforeseen security problems.

(4) Transaction Origin Use. Ethereum smart contract has a global variable, i.e., `tx.origin`, which can backtrack the entire call stack and return the original address of the called contract. If a contract uses this global variable for authorization and authentication, attackers can exploit the characteristics of `tx.origin` to design the corresponding attack to steal Ether. We describe a specific case of `tx.origin` vulnerability shown in Fig. 3. An attacker calls the *withdraw* function of the *Victim* contract in its *fallback* function to induce the *Victim* contract to transfer Ether to the *Attacker* contract. However, due to the statement of "$tx.origin == owner$" (line 8 in the left of Fig. 3), it is difficult to detect anomalies so that all the Ether in the *Victim* contract are transferred to the *Attacker*.

C. Block Dependency Layer

(1) Timestamp Dependency. Smart contracts usually utilize the block timestamp confirmed by miners (*i.e.,* nodes in the blockchain network) to achieve time constraints. All transactions in the block share the same timestamp, which ensures the consistency of the contract state. However, miners who confirm the block may deliberately choose a timestamp that is beneficial for themselves to grab benefits.

(2) Block Dependency. Ethereum smart contracts cannot directly call the built-in functions in smart contracts to generate a random number. Therefore, developers tend to use the block parameters, such as block number (*block.number*), block timestamp (*block.timestamp*), block hash (*block.blockhash*), or other related block parameters as the basic seeds to implement a random number generation function. However, similar to the timestamp dependency, the block parameters can be manipulated in advance by attackers, resulting in the generated random number being predictable, which may be exploited by malicious attackers to produce random numbers beneficial to themselves.

(3) Transaction Order Dependency. We have already mentioned in the previous paragraph that miners can determine the order of transaction execution in the blockchain network. However, the state updating of smart contracts strictly depends on the order of transaction execution, and the wrong order may incur a negative impact. Here, we describe a specific case of transaction order dependency shown in Fig. 4. Users A and B respectively submit transactions T_1 and T_2 at time t. Due to the execution order of T_1 and T_2 determined by miners, if T_1 is executed first, the contract state will be updated from S to S_1 and vice versa. Therefore, the final contract state depends on the transaction execution order confirmed by miners. At this point, if a malicious miner monitors the contract transaction in the block, the updating of the current contract state is likely to be controlled by blocking corresponding transactions, thereby deploying an attack in advance.

D. Smart Contract Security Incidents

In the real world, there exist many well-known smart contract security incidents. Here, we elaborate on several typical cases that have caused tremendous economic losses and seriously hindered the development of smart contracts.

(1) The DAO Attack. DAO is actually a decentralized autonomous organization that implements the DAO contract used for crowdfunding, which has already raised approximately 245 million US dollars of Ether before being attacked. In 2016, attackers exploited the reentrancy vulnerability of the DAO contract to steal the Ether worth 60 million US dollars, directly causing the *hard-fork* [26] of Ethereum. In order to have a deeper understanding of the DAO attack, we present a simplified version as shown in Fig. 5. Specifically, the *Bank* contract has a *userBalance* variable and two functions: *deposit* and *withdraw*. Function *deposit* allows users to deposit money, while function *withdraw* enables users to withdraw money via invoking *call.value* (*Bank*, line 9). To attack this contract, the *Attacker* contract, shown on the right of Fig. 5, implements
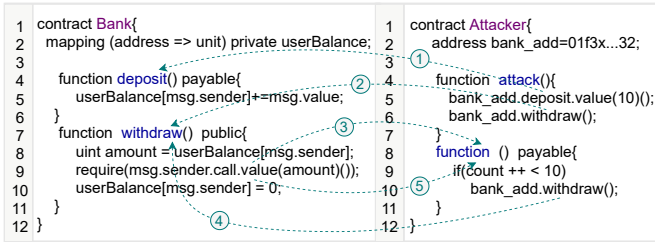
Fig. 5. A simplified version of The DAO attack

Fig. 6. An example of King of the Ether Throne Attack

Fig. 7. An example of Parity Multi-Sig Wallet Attack

two functions, *i.e., attack* and anonymous *fallback* functions (*Attacker*, lines 8–11). We would like to highlight that the *fallback* function of the smart contract, namely the function without name and argument, will be automatically invoked when the contract receives any Ether.

Attack. As shown in Fig. 5, contract *Attacker* can steal Ether from contract *Bank* through the following steps. First, *Attacker* deposits 10 Ether in contract *Bank* by calling the *deposit* function. Then, *Attacker* withdraws the 10 Ether by invoking the *withdraw* function (step 2). When the contract *Bank* sends Ether to *Attacker* using *call.value* (*Bank*, line 9), the *fallback* function of *Attacker* will be automatically invoked (step 3). In its *fallback* function, *Attacker* calls *withdraw* again (step 4). Since the *userBalance* of *Attacker* has not yet been updated (*Bank*, line 10), *Bank* believes that *Attacker* still has Ether balance in the contract, thus transferring 10 Ether to *Attacker* again (Step 5). The withdraw loop lasts for 9 times ($count + + < 10$, *Attacker* line 11). Finally, *Attacker* obtains much more Ether (100 Ether) than expected (10 Ether).

Underlying issue. The underlying security problem is that the balance of *Attacker* is updated after money transfer. Therefore, *Attacker* can call *withdraw* again utilizing the *fallback* mechanism, making contract *Bank* wrongly believe that it still has enough balance and transfer money to *Attacker* again.

(2) King of the Ether Throne. King of the Ether Throne (KoET) is an Ethereum gambling game [49], in which players can compete for the *Ethereum throne* to win all the bonuses in the contract. Whenever a player sends a *competition fee* (i.e., Ether) to compete for the *throne*, the *fallback* function in the KoET contract will be triggered. Then, the *fallback* function will check whether the *competition fee* is sufficient. If not, the contract throws an exception and rolls back this transaction. Otherwise, the player will become the new *Ethereum throne*. In the meantime, the KoET contract needs to send part of the *competition fee* as a reward to the former *throne*, and the rest remains in the contract as the final bonus. After 24 hours, the player who competes successfully will receive all the bonuses in the KoET contract.

However, this gambling game is not as reasonable as it seems, which is susceptible to denial of service (DoS) attack. As shown in Fig. 6, the attacker will constantly occupy the *throne* until he wins all the bonuses in the contract through the following three steps.

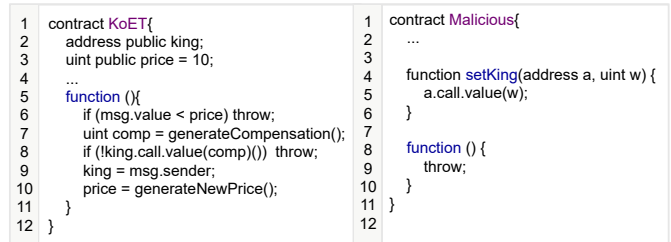- First, the attacker executes the *setKing* function in the
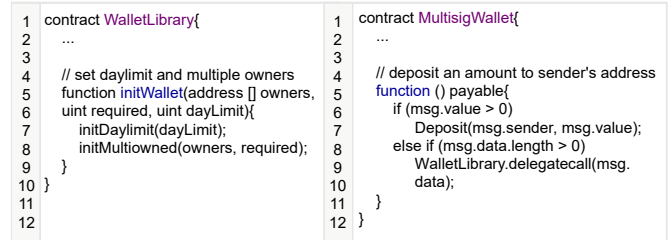
Malicious contract to send enough *competition fee* to compete for becoming a new *throne*.
- Then, when a new player participates in the competition and pays a sufficient *competition fee*, KoET contract will send rewards to the current *throne* (i.e., *Malicious*), further triggering the *fallback* function in the *Malicious*, where only exists a statement of exception throw.
- Finally, due to the exception handling, KoET tries to send rewards to *Malicious* again. In the end, sending rewards multiple times will exhaust *Gas*, and nobody can compete for the *throne* again so the attacker occupies the *throne* all the time and wins all the final bonuses.

(3) Parity Multi-Sig Wallet. The parity multi-signature wallet is a public library contract implemented to manage the digital asset of users, which includes some common functions and logic code. Users can call functions in this public library contract in their wallet contract to execute the corresponding business logic. However, centralized management of functions in the public library contract can easily become the target of attackers. Since most of the user wallets in Ethereum rely on the parity multi-signature wallet, attackers can disturb all the wallet contracts by attacking the public library contract.

Fig. 7 describes the example of Parity Multi-Sig Wallet, including the contract fragment of *WalletLibrary* and *Multisig-Wallet*. In the *WalletLibrary*, the *initWallet* function is used to initialize the usage date and owner of the wallet. Other wallet contracts can use *delegatecall* to call the *initWallet* function. As shown in line 9 of the *MultisigWallet* contract, the wallet uses *delegatecall* to call the public library *WalletLibrary*. Since all the public functions in *WalletLibrary*, such as *initDayLimit* and *initMulitowned*, can be called by anyone without authorization, the attacker is also able to control the multi-signature wallet by calling the *initWallet* in *WalletLibrary* to claim the ownership of the multi-signature wallet. Therefore, if the

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

Fig. 8. An example of Beauty Chain Integer Overflow Attack

```
1  contract Rubixi {
2     uint private balance=0;
3     uint private collectedFees=0;
4     uint private feePercent=10;
5     uint private Order=0;
6     uint private pyramidMultiplier=300;
7     address private creator;
8     struct Participant { uint payout; }
9     Participant [] private participants;
10
11    function Rubixi(){ creator=msg.sender; }
12    function (){ addPayout(); }
13
14    function collectAllFees() onlyowner {
15       if (collectedFees==0) throw;
16       creator.send(collectedFees);
17       collectedFees=0;
18    }
19
20    function addPayout() private {
21       uint fee = feePercent;
22       participants.push(Participant(msg.sender,
23                  (msg.value*pyramidMultiplier)/100));
24       if (participants.length==10)
25          pyramidMultiplier=200;
26       else if (participants.length==25)
27          pyramidMultiplier=150;
28       balance += (msg.value*(100-fee)) / 100;
29       collectedFees += (msg.value * fee) / 100;
30       while (balance>participants[Order].payout) {
31          uint payoutToSend = participants[Order].payout;
32          participants[Order].etherAddress.send(payoutToSend);
33          balance -= participants[Order].payout;
34          Order += 1;
35       }
36    }
37 }
```

Fig. 9. An example of Ponzi Scheme Attack – Rubixi

attacker destroys the wallet through the *self-destruct* operation, this will freeze all the user's wallets that depend on this public multi-signature wallet.

(4) Beauty Chain Integer Overflow. The improper assignment operations of integer variables in smart contracts are prone to integer overflow. In the real world, there are many cases of integer overflow/underflow in smart contracts. For example, in the Proof-of-Week-Hands (POWH) [50], around 2,000 Ether was stolen due to its integer overflow vulnerability. In recent years, the most famous smart contract integer overflow incident is the Beauty Chain vulnerability, in which attackers exploit the batch transfer method *batchTransfer* in the contract to generate an unlimited number of BEC token, leading to the value of the BEC token evaporated to zero.

Fig. 8 presents the code snippets of *batchTransfer* function in the BEC contract. *batchTransfer* is a batch transfer function. Unfortunately, this function has an integer overflow vulnerability, which lies in line 6, *i.e.,* "$uint256\ amount = uint256(cnt) * \_value$", where *uint256* represents a 256-bit unsigned integer with a data range in [0, 2^{256}-1]. The attacker just exploits this nature by assigning a large value into *_value* that makes the *amount* variable exceed the data range of *uint256*, incurring an integer overflow vulnerability. Then, attackers copy an unlimited number of BEC tokens, thereby causing the value of the BEC token to evaporate to zero.

(5) Ponzi Scheme Rubixi. A Ponzi scheme is a classic investment fraud method, which is typically characterized by paying existing investors a portion of the participation fee for new investors in return. The originator of a Ponzi scheme usually promises that joining an investment will generate high returns. The rewards are low and the risk is low to attract new investors. However, these types of scams ultimately only harm the interests of the vast majority of participants.

With the widespread use of smart contracts, Ponzi schemes have gradually taken on new characteristics. Due to the anonymity of the blockchain, all participants cannot know the real identity of the contract initiator, resulting in many Ponzi schemes that can be disguised in smart contracts. under disguise. We call this type of Ponzi scheme a smart contract Ponzi scheme. Since smart contracts are self-executing and cannot be tampered with, it is one of the most beneficial means for Ponzi schemes to attract victims. In order to better

understand the smart contract Ponzi scheme, this subsection presents a typical *Rubixi* case in Fig. 9.

The *Rubixi* contract contains a constructor *Rubixi*, a fallback function, a function *collectAllFees*, and a function *addPayout*, respectively.

- The *Rubixi* constructor is executed when the contract is created, and only once.
- The fallback function (i.e. *function () { addPayout(); }*) will be automatically executed when an ether transfer is received, and the fallback function will be triggered automatically when the participant puts Ether into the *Rubixi* contract.
- The function *collectAllFees* is used by the contract creator to withdraw funds deposited in the contract.
- The *addPayout* function is the most critical function, which implements the main logic of the Ponzi scheme: (1) Record the participant's address and participation fee; (2) Calculate the participant's investment fee; (3) When the balance in the contract is sufficient, then the remuneration fee is paid to the existing participants.

Obviously, it can be seen from the code snippet in Fig. 9, *pyramidMultiplier* is a key variable that controls how much profit a participant can earn. To attract early investment participants, the contract owner pre-sets its value to 300, when

the number of participants reaches 10 and 25, the value of *pyramidMultiplier* is reduced to 200 and 150, respectively. From the perspective of the entire contract logic, the main purpose of the contract initiator is to steal investment participants' investment fees, such as *collectedFees+=(msg.value*fee)/100* (Line 29) is used to charge a 10% participation fee for each investment, and the fee is extracted by calling the *collectAllFees* function *creator.send(collectedFees)* (Line 16).

III. VULNERABILITY DETECTION TECHNIQUES FOR SMART CONTRACTS

The security issues of smart contracts are becoming a significant concern for both researchers and developers. To prevent malicious attackers from exploiting smart contract vulnerabilities, researchers have tried various methods to comprehensively analyze the source code or bytecode of Ethereum smart contracts.

Traditional program vulnerability detection employs feature matching techniques [51], which extract the malicious code and analyzes the source code through the matching module. Unfortunately, this method restricts the application scope and incurs high false negatives. In recent years, there are five majority methods for smart contract vulnerability detection, including formal verification [52]–[54], symbolic execution [55]–[57], fuzzing detection [58]–[60], intermediate representation [61] and deep learning [62]–[66].

A. Formal Verification

Formal verification is one of the significant technologies in security verification, which can transform the concepts, judgments, and ratiocination into a smart contract model through formal language, thereby eliminating the ambiguity and non-universality in the contract program. Moreover, this method verifies the correctness and safety of functions with rigorous logic and proof. Common formal verification methods include model checking and deductive verification. Model-checking lists all possible states and checks them individually to confirm whether the contract has the corresponding characteristic through state-space search. Deductive verification is based on the ideology of theorem-proof that uses logical formulas to describe the properties and proves the characteristics of the system through the evidence rules. Generally, formal verification takes advantage of mathematical logic to verify the code implementation whether satisfies some pivotal characteristics.

Currently, formal verification technology has been successfully applied to many fields with high-security requirements, such as nuclear power [67] and aerospace [68]. By using formal methods for security analysis of smart contracts, we can ensure that contract generation is more standardized and contract execution is more reliable. To summarize, we conclude five formal verification methods as follows.

(1) F* Framework. F* framework is one of the formal frameworks developed by [69], which formalizes the semantics of the EVM bytecode and compiles the bytecode into *Ocaml* form. Then, the smart contract source code and bytecode
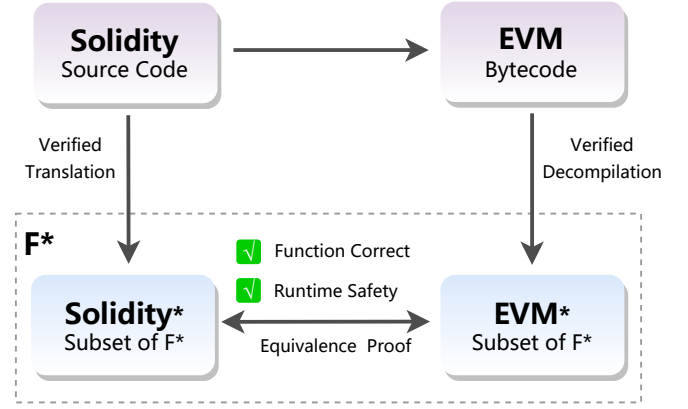

Fig. 10. The overall architecture of *F** Framework

are converted into the functional programming language *F** for analyzing the security and verifying the correctness of functions during the contract runtime. Fig. 10 depicts the overall architecture of the *F** framework, which implements the two modules, i.e., *Solidity** and *EVM**, to verify the functional equivalence between source code and bytecode, ensuring the correctness of the outputs.

(2) KEVM Framework. KEVM [70] is a formal analysis framework, which utilizes the $\mathbb{K}$ framework to construct an executable formal specification based on the EVM bytecode stack. Further, KEVM serves as a platform for building a wide range of analysis tools and other semantic extensions for EVM.

(3) Isabelle/HOL. Isabelle/HOL [71] is a proof assistant designed to infer and validate the correctness of EVM bytecode based on separation logic. This tool constructs the bytecode sequence into linear code blocks and splits the contract into basic blocks. Then, it further builds a logic program on this basis for reasoning verification.

(4) ZEUS. ZEUS [72] is a static analysis tool, which can verify the correctness of smart contracts and validate their fairness. This method employs abstract interpretation, symbolic model checking, and constraint statements to quickly verify the security of smart contracts. In total, it can detect six security vulnerabilities in smart contracts including reentrancy, integer overflow/underflow, transaction order dependence, etc.

(5) VaaS. VaaS [73] is a "one-click" smart contract security detection platform based on the formal verification method. While this tool is able to automatically detect 10 major items and 32 small items of conventional security vulnerabilities in smart contracts, it can also accurately and efficiently locate the risk code location and provide modification suggestions.

B. Symbolic Execution

The primary idea of symbolic execution is to symbolize variables in the program code, which maintains a set of constraints for all execution paths by symbolizing program input. With symbolic execution, the constraint solver is used to solve the constraints and determine the reason for execution input. Finally, developers can use the constraint solver to get
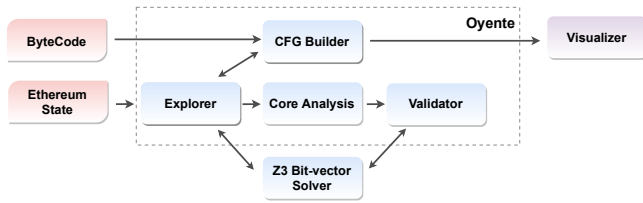
Fig. 11. The overall architecture and execution process of Oyente

Fig. 12. The overall architecture and execution process of Maian

Fig. 13. The overall architecture and execution process of ILF

a new test input to detect whether the symbol value has a potential vulnerability.

The execution process of symbolic execution applied to smart contract vulnerability detection can be classified into the following four steps: 1) symbolize the variable values in the contract; 2) explain the instructions in the execution program one by one; 3) update the execution status and collect path constraints to explore all executable paths in the program; 4) discover the corresponding security issues. In this subsection, we present eight symbolic execution methods.

(1) Oyente. Oyente [74] is one of the pioneer vulnerability detection tools for smart contracts, which utilizes symbolic execution to detect smart contract vulnerabilities based on the control flow graph (CFG), which takes the bytecode and the state of smart contracts as input to simulate EVM and traverse different execution paths of a certain contract. There are four modules in Oyente, including *CFGBuilder*, *Explorer*, *CoreAnalysis*, and *Validator*, and the overall architecture is illustrated in Fig. 11.

(2) Mythril. Mythril [75] is a smart contract static analysis tool that combines concept analysis, taint analysis, and control flow verification to detect common vulnerabilities in Ethereum smart contracts, including reentrancy vulnerabilities, integer overflow, exception handling, etc.

(3) Osiris. Osiris [76] is a static analysis framework for smart contracts, which consists of three components, i.e., symbolic analysis, taint analysis, and integer error detection. This tool can detect three different types of integer errors, namely arithmetic errors, truncation errors, and signature errors.

(4) Gasper. To monitor the gas consumption of smart contracts, Chen et al. [77] present a static analysis tool named Gasper, which focuses on gas costly patterns from the existing smart contracts. Gasper takes the bytecode as the input to identify gas costly patterns, which runs symbolic execution on bytecode to find all the reachable code blocks in a smart contract. Specifically, this tool employs the *Z3* solver [78] to confirm whether the condition is true or false.

(5) Maian. Maian [79] detects smart contract vulnerabilities by using dynamic analysis, which discovers security vulnerabilities through a long sequence of invocations during the contract runtime. Fig. 12 describes the overall architecture of Maian, which consists of two major components, namely symbolic analysis and concrete validation. Different from other detection tools, Maian focuses on the problematic smart contracts that can be labeled into three categories: *Greedy*, *Prodigal*, and *Suicidal*. Specifically, asset frozen indefinitely is
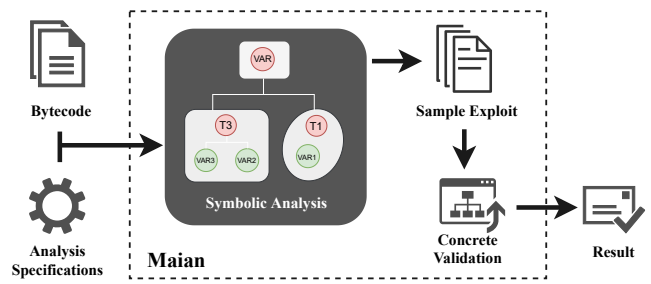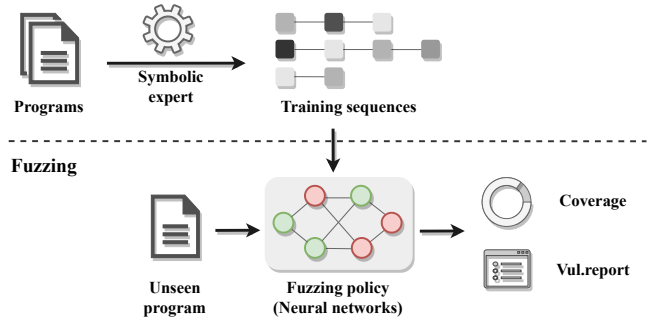
regarded as *Greedy*, asset prone-leaked to unfamiliar accounts is looked as *Prodigal*, and a contract destroyed arbitrarily by everyone is treated as *Suicidal*.

(6) Securify. Securify [80] is a static security analyzer for Ethereum smart contracts, which has the characteristics of scalability, fully automated, and high accuracy. Securify identifies the smart contacts vulnerabilities by analyzing the dependency graph and extracting precise semantic information from the bytecode, and checks compliance and violation patterns that capture sufficient conditions for proving if a property holds or not.

(7) TeEther. Different from traditional vulnerability detection tools, TeEther [81] considers the automatic identification of smart contract vulnerabilities and designs the method of contract generation. Based on the symbolic execution, this tool transverses the critical execution paths by analyzing the bytecode in order to solve the security issues in the contract.

(8) Sereum. Sereum [82] is a novel detection solution that focuses on the reentrancy vulnerability of smart contracts. The tool employs dynamic taint tracking to monitor the data flow during the contract execution, thereby automatically avoiding inconsistent states and effectively preventing reentrant attacks.

C. Fuzzing Test

Fuzzing test is one of the most popular software analysis and vulnerability detection techniques. Conceptually, the core idea of fuzzing test is to provide a large number of test samples for the program and monitor the abnormal behavior during the program execution. Compared with other testing technologies, fuzzing test is easy to be deployed and has good scalability
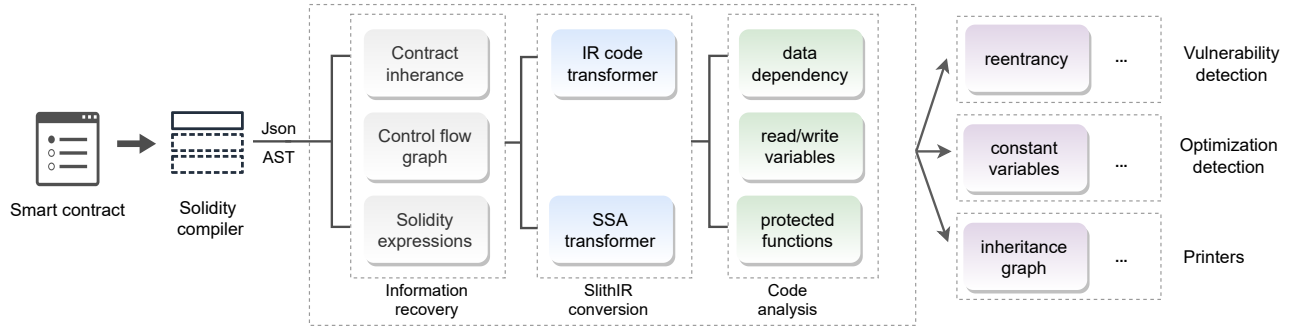
Fig. 14. The overall architecture and execution process of Slither

and applicability. In this subsection, we introduce three fuzzing detection methods for smart contract vulnerability detection.

(1) ContractFuzzer. ContractFuzzer [83] is the first fuzzing-based dynamic analysis method for detecting Ethereum smart contract security vulnerability, which generates fuzzing input based on the smart contract ABI specification and designs a test plan to detect vulnerabilities. First, ContractFuzzer configures the EVM and records the runtime behavior of the smart contract. Then, it detects vulnerabilities by analyzing these recorded logs.

(2) Regurad. Regurad [84] is also a fuzzing analyzer, which focuses on the reentrancy vulnerabilities of smart contracts. Regurad performs the fuzzing test by iteratively generating random and diversifying test cases, thereby tracking the contract execution and further identifying the reentrancy vulnerability dynamically.

(3) ILF. ILF [85] is a neural network-based smart contract *Fuzzer*, which is dedicated to generating better test cases and transaction sequences in the fuzzing of smart contracts. The solution of ILF is to first employ the symbolic execution engine to produce a large number of excellent call sequences, and then use the neural networks to learn the characteristics of these invocation sequences to guide the fuzzing engine to yield excellent scheduling strategies. Fig. 13 illustrates the high-level idea of ILF, which trains a suitable architecture of neural networks that captures a probabilistic fuzzing policy for generating contract inputs. Then, ILF utilizes the learned policy to produce input sequences for fuzzing unseen contracts.

D. Intermediate Representation

To analyze smart contracts more accurately, researchers explore converting smart contract source code or bytecode into an intermediate representation (IR) with highly semantic information. They discover security issues by analyzing the intermediate representation of the contract. Here, we summarize the following six types of smart contract analysis tools using intermediate representation.

(1) Slither. Slither [86] is a static analysis framework for Ethereum smart contract analysis to provide rich information, which combines data flow analysis and taint analysis. This tool converts the smart contract source code into an intermediate representation named *SlithIR*, which employs a static single al-

location (SSA) form and reduced instruction set to simplify the analysis process while retaining the lost semantic information when the source code is converted to EVM bytecode. Fig. 14 describes the core detection process of Slither, which is not only used to detect common vulnerabilities in smart contracts but also can give suggestions on contract code optimization.

(2) Vandal. Vandal [87] is a smart contract static analysis tool at the EVM bytecode level, consisting of an analysis pipeline and a decompiler. The decompiler performs abstract interpretation to convert the contract bytecode into a high-level intermediate representation (IR) in the form of logical relations, and then uses novel logic-driven methods to analyze security vulnerabilities.

(3) Madmax. Madmax [88] is a gas-oriented vulnerability analysis tool for Ethereum smart contracts, which performs control flow analysis and implements a decompiler program structure to detect smart contract security vulnerabilities based on Vandal. The tool similarly decompiles the EVM bytecode into the high-semantic intermediate representation but focuses on detecting gas-oriented vulnerabilities, *e.g.,* Ether frozen.

(4) Ethir. Albert et al. [89] present a static analysis tool *Ethir*, to analyze Ethereum smart contracts at the bytecode level on the basis of the control flow graph (CFG) generated by Oyente. Then, the tool converts the CFG into a rule-based intermediate representation (RBP) to analyze and infer the security properties of the EVM bytecode.

(5) Smartcheck. SmartCheck [90] is an extensible static analysis tool for smart contracts, which converts the source code into an XML-based intermediate representation. Then, SmartCheck detects the smart contract vulnerabilities in the middle representation of the contract based on the analysis of the XPath patterns. Moreover, this tool can be improved in multiple directions, such as improving the grammar, making patterns more precise, and adding new patterns.

(6) ContractGuard. ContractGuard [91] is the first intrusion detection tool for Ethereum smart contracts against attacks. Based on the intrusion detection system (IDS), ContractGuard detects the abnormal control flow caused by potential attacks, which realizes intrusion detection by embedding the corresponding decentralized nature in the contracts to profile context-tagged acyclic path.

(a) Graph generation phase   **(b) Graph normalization phase**   **(c) Message flow phase and reentrancy detection**
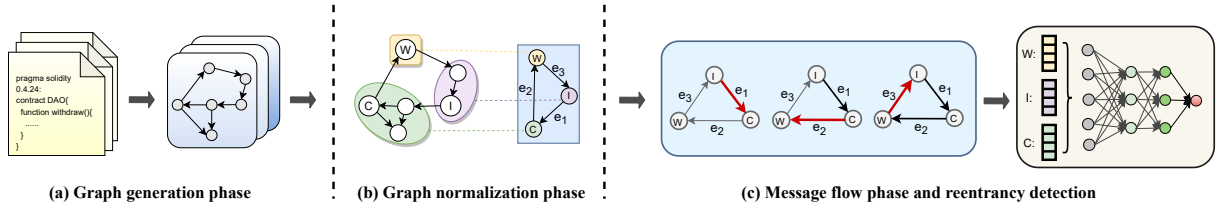
Fig. 15. The overall architecture of TMP. (a) The contract graph generation phase, which constructs the graph from source code by extracting the control- and data- flow semantics in the code; (b) the graph normalization phase, which passes the features of normal nodes to their neighboring core nodes; (c) the novel temporal message flow network for vulnerability modeling and detection.

E. Deep Learning

In recent years, there has been an increasing practice of deep learning in the field of program security analysis [62]–[66], which has achieved encouraging results. The advancement of deep learning technology promotes the birth of various security detection methods using deep learning models. Besides, deep learning-based methods have good expansibility and adaptability to new vulnerabilities. In this subsection, we review five recent kinds of research using deep learning models for smart contract vulnerability detection.

(1) SaferSC. SaferSC [92] is the first vulnerability detection model based on deep learning for smart contracts. This model focuses on three smart contract vulnerabilities defined by Maian and achieves a higher detection accuracy than Maian [79]. Besides, SaferSC analyzes the operation code (opcode) of the smart contract and employs the LSTM network to build a sequence model at the opcode level to achieve precise vulnerability detection.

(2) RecChecker. RecChecker [100] is a deep learning-based method, which focuses on detecting smart contract reentrancy vulnerability. This method captures the basic semantic information and control flow dependency information in a smart contract by converting a contract source code into the form of a *contract snippet*. Then, RecChecker constructs the Bidirectional Long Short-Term Memory (BLSTM) with attention mechanism (Attention) [93] to achieve the automatic detection of reentrancy vulnerability.

(3) DR-GCN. DR-GCN [94] is the first to explore the *contract graph* for smart contract vulnerability detection, which uses the graph convolutional neural network (GCN) [95] to construct a vulnerability detection model by converting a smart contract into a structure of *contract graph* with a high degree of semantic representation. DR-GCN analyzes smart contracts on two platforms, *i.e.,* Ethereum and VNT Chain, and detects three types of vulnerabilities, *i.e.,* reentrancy, timestamp dependency, and infinite loop.

(4) TMP. Temporal message propagation (TMP) is an advanced version of using *contract graph* for smart contract vulnerability detection [94]. By constructing a *contract graph* of a smart contract, the key functions and variables are converted into core nodes with high semantic information, while the execution modes are reflected into the directed edge depending on the control- and data- dependency. Furthermore, TMP considers the temporal information on the edges of the contract graph and builds the temporal message propagation graph neural network [93], [96]. As shown in Fig. 15, TMP consists of three phases: (1) a graph generation phase, which extracts the control flow and data flow semantics from the source code and explicitly models the fallback mechanism; (2) a graph normalization phase inspired by the k-partite graph, and (3) a novel temporal message flow network for vulnerability modeling and detection.

(5) ContractWard. ContractWard [97] utilizes machine learning techniques to detect vulnerabilities in smart contracts, which learns the patterns of vulnerable contracts in training samples. ContractWard extracts bigram features from smart contract opcodes and employs a variety of machine learning algorithms and sampling algorithms, which can effectively and efficiently detect six types of vulnerabilities based on extracted static characteristics.

(6) CGE. CGE [98] is the first to investigate the idea of fusing conventional expert patterns and graph-neural-network extracted features for smart contract vulnerability detection. CGE proposes to characterize the contract function source code as contract graphs. Then, CGE explicitly normalizes the graph to highlight key variables and invocations. Finally, CGE utilizes a novel temporal message propagation network to automatically capture semantic graph features and combines the graph feature with designed expert patterns to yield a final detection.

(7) AME. AME [99] is a new system beyond pure neural networks that can automatically detect vulnerabilities and incorporate expert patterns into networks in an explainable fashion. Firstly, AME extracts expert patterns of a specific vulnerability from the function code. Second, AME utilizes a graph construction and normalization module which transforms the function code into the code semantic graph vector. Finally, AME combines local expert patterns and the global graph feature for vulnerability detection and outputs explainable weights. AME is an important tool for explainable and accurate contract vulnerability detection.

IV. EVALUATION

In this section, we first overview 29 different smart contract detection methods. Then, we analyze these methods from different aspects and compare how well they support detecting various types of vulnerabilities. Finally, we evaluate the performance of these tools in terms of accuracy, F1-Score, and average detection time.

Analysis Method	Detection Tool	Automatic Degree	Language Support	Public Available	Ref.
Formal verification	F* framework	Semi-automatic	EVM, Ocaml	Not Available	[69]
	KEVM	Semi-automatic	Solidity, EVM	https://github.com/kframework/evm-semantics	[70]
	Isabelle/HOL	Semi-automatic	EVM, Ocaml	https://github.com/pirapira/eth-isabelle	[71]
	ZEUS	Fully automatic	Solidity, EVM	Not Available	[72]
	VaaS	Fully automatic	Solidity, EVM	Not Available	[73]
Symbolic execution	Oyente	Fully automatic	Solidity, EVM	https://github.com/melonproject/oyente	[74]
	Mythril	Fully automatic	Solidity, EVM	https://github.com/ConsenSys/mythril	[75]
	Osiris	Fully automatic	Solidity, EVM	https://github.com/christoftorres/Osiris	[76]
	Gasper	Fully automatic	Solidity, EVM	Not Available	[77]
	Maian	Fully automatic	Solidity, EVM	https://github.com/MAIAN-tool/MAIAN	[79]
	Securify	Fully automatic	Solidity, EVM	https://github.com/eth-sri/securify2	[80]
	TeEther	Fully automatic	Solidity, EVM	https://github.com/nescio007/teether	[81]
	Sereum	Fully automatic	Solidity, EVM	https://github.com/uni-due-syssec/eth-reentrancy-attack-patterns	[82]
Fuzzing detection	ContractFuzzer	Fully automatic	Solidity, EVM	https://github.com/gongbell/ContractFuzzer	[83]
	Regurad	Fully automatic	Solidity, EVM	Not Available	[84]
	ILF	Fully automatic	Solidity, EVM	https://github.com/eth-sri/ilf	[85]
Intermediate representation	Slither	Fully automatic	Solidity, EVM	https://github.com/crytic/slither	[86]
	Vandal	Fully automatic	Solidity, EVM	https://github.com/usyd-blockchain/vandal	[87]
	Madmax	Fully automatic	Solidity, EVM	https://github.com/nevillegrech/MadMax	[88]
	Ethir	Fully automatic	Solidity, EVM	https://github.com/costa-group/ethIR	[89]
	Smartcheck	Fully automatic	Solidity, XML	https://https://github.com/smartdec/smartcheck	[90]
	ContractGuard	Fully automatic	Solidity	https://https://github.com/contractguard/experiments	[91]
Deep learning	SaferSC	Fully automatic	Solidity, EVM	https://github.com/wesleyjtann/Safe-SmartContracts	[92]
	RecChecker	Fully automatic	Solidity	https://github.com/Messi-Q/ReChecker	[100]
	DR-GCN	Fully automatic	Solidity	https://github.com/Messi-Q/GraphDeeSmartContract	[94]
	TMP	Fully automatic	Solidity	https://github.com/Messi-Q/GNNSCVulDetector	[94]
	ContractWard	Fully automatic	Solidity	Not Available	[97]
	CGE	Fully automatic	Solidity	https://github.com/Messi-Q/GPSCVulDetector	[98]
	AME	Fully automatic	Solidity	https://github.com/Messi-Q/AMEVulDetector	[99]

A. Overview of Existing Methods

Table II displays different smart contract detection methods, in which the first column represents five effective smart contract analysis methods, while the second column lists the corresponding detection tools. In the third column, we describe the automatic degree of each detection tool. Here, *fully automatic* refers to the end-to-end solution, which means that a tool takes a contract as input and outputs the specific vulnerability detection results, and *semi-automatic* represents manually defining relevant contract attributes during the detection process. For example, formal verification methods employ theorems to prove the security of smart contracts. Since these proofs are semi-automated, formal verification methods require a lot of manual operations to conduct smart contract verification and analysis. The fourth column summarizes the smart contract languages and forms supported by the corresponding detection tools, and the fifth column presents the degree of open source with the open-source addresses. In the last column, we list the references for each method.

According to the statistical results in Table II, we present the overall analysis as follows.

- Compared with other methods, formal verification has a low degree of automated and open-sourced nature.
- There are many kinds of detection tools based on symbolic execution and intermediate representation, and all of them can perform fully automatic vulnerability detection. Most of them have open-source code.
- There are few detection tools based on the fuzzing

test. The reason may be that the implementation and operation of dynamic fuzzing detection methods are more complicated and cumbersome. Furthermore, due to the randomness of test cases, the methods of fuzzing detection cannot cover all the test paths.

- Most of the smart contract vulnerability detection methods based on deep learning still focus on the level of the contract source code. They can not only perform fully automatic detection but also with a high degree of open-sourced nature.

B. Result Analysis

Table III summarizes 29 smart contract vulnerability detection tools and the detectable vulnerabilities supported by them, which are concluded into three categories. We elaborate on the specific analysis of the detectable vulnerabilities and performance of existing methods in detail.

1) Comparison of existing methods from different levels: From the analysis of Solidity code layer, most of the detection tools support verifying the reentrancy vulnerability. Since the most famous *The DAO incident* is caused by the reentrancy vulnerability, most researchers and developers first focus on the analysis of such vulnerability. In addition, many tools pay close attention to the integer overflow, exception handling, unknown function call, and Ether frozen, while these security vulnerabilities have incurred major security incidents, such as Beauty Chain integer overflow and Parity multi-signature wallet frozen. Further, we need to point out that there are few tools for analyzing access control, denial of service, type

Detection Tool	Solidity code layer									EVM execution layer				Block dependency layer		
	Reentrancy	Integer Overflow	Access Control	Exception Handling	Denial of Service	Type Mismatch	Unknown Function Call	Ether Frozen	Replay Attack	Short Address	Ether Loss	Call-Stack Overflow	Tx.origin	Timestamp Dependency	Block Parameter Dependency	Transaction Ordering Dependency
F* framework	✓			✓			✓							✓	✓	✓
ZEUS	✓	✓							✓					✓	✓	✓
VaaS	✓	✓		✓	✓		✓						✓	✓	✓	
Oyente	✓	✓		✓					✓			✓		✓		✓
Mythril	✓	✓		✓	✓				✓		✓		✓	✓	✓	
Osiris	✓	✓		✓					✓			✓		✓		✓
Maian	✓							✓			✓	✓				
Securify	✓			✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
TeEther	✓						✓	✓								
Sereum	✓															
ContractFuzzer	✓			✓			✓	✓						✓	✓	
Reguard	✓															
ILF	✓			✓				✓			✓			✓	✓	
Slither	✓					✓	✓	✓				✓	✓	✓		
Vandal	✓			✓				✓			✓	✓				✓
Madmax		✓						✓								
Ethir	✓			✓										✓		✓
Smartcheck	✓	✓	✓		✓		✓	✓					✓	✓		
ContractGuard	✓	✓		✓	✓	✓	✓			✓			✓	✓	✓	✓
SaferSC	✓							✓			✓	✓				
RecChecker	✓															
DR-GCN	✓													✓		
TMP	✓													✓		
ContractWard	✓	✓										✓		✓		✓
CGE	✓													✓		
AME	✓													✓		
KEVM framework	KEVM framework and Isabelle/HOL provide verification conditions for contract program analysis and formal verification methods. The functional correctness and program logical rationality are verified during contract execution, but it is not used to detect specific contract vulnerabilities.															
Isabelle/HOL																
Gasper	Gasper is a tool for automatically locating gas costly patterns by analyzing smart contracts' bytecode.															

mismatch, and replay attack, while these four vulnerabilities occur less frequently and are easy to prevent.

From the analysis of the EVM execution layer, there are few tools for detecting the short address vulnerability, which can be attributed to the low probability of occurrence and ease of verification. Besides, both methods based on formal verification and deep learning fail to detect smart contract vulnerabilities at the EVM execution level, such as ZEUS, F* framework, RecChecker, DR-GCN, and TMP.

From the analysis of Block dependency layer, timestamp dependency is a common smart contract vulnerability and easy to detect. Most detection tools support detecting timestamp dependency, and it is worth noting that F* framework, ZEUS, ContractGuard, and Securify are able to detect all the vulnerabilities in the block dependency layer.

In summary, although most vulnerability types can be detected by corresponding tools, some easy-to-verify vulnerabilities are only supported by a few detection tools. For example, only two tools can detect access control and short address vulnerabilities, while the probability of these two vulnerabilities is relatively low, but the losses caused by such vulnerabilities are also inestimable. Therefore, comprehensive coverage for various vulnerabilities is still one of the urgent problems to be solved by these automated detection tools. Currently, with the rapid growth of the number of smart contracts, the numbers and types of smart contract vulnerabilities are also increasing. Using automated detection tools to conduct more comprehensive and scalable detection for smart contracts is a key issue that deserves to be further studied.

2) Comparison of detectable vulnerabilities of existing methods: According to the statistics in Table III, formal verification methods detect few contract vulnerabilities. Among them, KEVM and Isabelle/HOL can support contract program analysis and verify functional correctness and program logical rationality of smart contracts during execution, while they are unable to detect specific contract vulnerabilities. Moreover, F* framework, ZEUS, and VaaS do not support detecting vulnerabilities in the EVM execution layer. It is worth to point that most formal verification methods employ mathematical theorem proofs and complex mechanisms for verification,

which are not easy to use.

Most of the detection tools based on symbolic execution can detect more kinds of contract vulnerabilities. For example, Oyente, Mythril, and Securify can respectively detect 7, 9, and 13 vulnerabilities, of which Securify can support detecting the most types among the 27 detection tools. Osiris, Gasper, Maian, TeEther, and Sereum detect fewer vulnerabilities, while Maian is designed to solve three unique vulnerabilities (Greedy, Prodigal, Suicidal), and Sereum only focuses on the reentrancy vulnerability.

Compared with other methods, ContractFuzzer, Reguard, and ILF based on fuzzing detection can only detect a few vulnerabilities. ContractFuzzer and ILF both support detecting 6 vulnerabilities, while Reguard only focuses on the reentrancy vulnerability.

Moreover, the detection tools based on the intermediate representation have achieved good results. Among them, Vandal, Slither, Smartcheck, and ContractGuard support detecting 6, 7, 8, and 10 vulnerabilities respectively, while Madmax and Ethir can only detect 2 and 4 vulnerabilities.

It is worth mentioning that deep learning-based methods are also for few vulnerabilities detection. SaferSC and Contract-Ward can detect 4 and 5 vulnerabilities respectively, while RecChecker only focuses on the reentrancy vulnerabilities. DR-GCN and TMP are both constructed for detecting reentrancy and timestamp dependency vulnerabilities.

To summarize, vulnerabilities covered by various detection tools are still incomplete. Most of them can only detect low-level security violations and vulnerabilities, and lack inferences during contract execution, making it difficult to detect external security issues caused by contract invocations. Therefore, in the context of the current growing number of smart contracts, it is still challenging to use a single detection tool to fully verify smart contract vulnerabilities.

3) Performance comparison of existing methods: We compare and analyze the performance of different vulnerability detection tools in detail shown in Table IV. First, we select the representative detection tools from the five smart contract security analysis methods described in section III, namely VaaS, Oyente, Smartcheck, Contract Fuzzer, and TMP. Then, we randomly collect 300 Ethereum smart contracts as test samples from the official website Etherscan [23]. The performance of detection tools is evaluated from three aspects, i.e., accuracy, F1-Score, and average detection time, and we focus on three smart contract vulnerabilities, reentrancy, integer overflow, and timestamp dependency.

(1) Accuracy. To evaluate the pros and cons of detection tools, we first focus on the most common evaluation indicator, namely accuracy. Generally, we determine a classifier whether is effective in a sense through accuracy, which can objectively reflect the most direct effect of various detection tools. Vulnerability detection is actually a binary classification problem, that is, the detection tool predicts whether there is a certain vulnerability in a contract. For the binary classification problem, the matching result is usually used as an important evaluation indicator, including the following four situations:

- True positive (TP). For a contract, the detection result is true and the real value is also true, which implies the detection result is correct.
- False positive (FP). For a contract, the detection result is true and the real value is also false, which implies the detection result has a false positive.
- False negative (FN). For a contract, the detection result is false and the real value is also true, which implies the detection result has a false negative.
- True negative (TN). For a contract, the detection result is false and the real value is also false, which implies the detection result is correct.

In this experiment, the result of TP+FP+FN+TN is equal to the amount of smart contract test samples, and the calculation of accuracy is denoted in formula 1.

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \quad (1)$$

According to the evaluation results in Table IV, TMP achieves the highest accuracy (84.48%) in the reentrancy vulnerability detection, while VaaS and ContractFuzzer have the accuracy of 82.54% and 67.89% respectively. In contrast, the accuracy of Oyente and Smartcheck is slightly insufficient, only 61.62% and 52.97% respectively. For integer overflow vulnerability, the accuracy of VaaS is as high as 86.80%, while Oyente and Smartcheck only have 66.85% and 58.48%, ContractFuzzer and TMP do not support detecting such vulnerability. For timestamp dependency, VaaS and TMP achieve higher detection accuracy of 89.20% and 83.45% respectively, while the detection accuracy of Oyente, Smartcheck, and ContractFuzzer are very low, only 59.45%, 51.32%, and 68.08% respectively.

(2) F1-Score. F1-Score is an important measurement indicator in the binary classification problem. It is the harmonic average of precision and recall, which is usually treated as the final evaluation standard for some classification missions. The calculation of F1-Score is shown in formula (2-4).

$$Precision = \frac{TP}{TP + FP} \quad (2)$$

$$Recall = \frac{TP}{TP + FN} \quad (3)$$

$$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall} \quad (4)$$

According to the evaluation results of Table IV, TMP achieves the highest F1-Score (74.15%) in the detection of reentrancy vulnerability, followed by VaaS (73.95%), and the remaining detection tools are relatively low. For integer overflow vulnerability, VaaS achieves an F1-Score of 80.10%, while Oyente and Smartcheck get the F1-Score of 59.64% and 54.96% respectively. For timestamp dependency, VaaS and TMP both obtain good F1-Score of 82.46% and 79.19%.

(3) Average detection time. Average detection time is also one of the significant indicators for evaluating automated detection tools. Currently, the long audit time of most detection

Vulnerability Type	Detection Tool	Accuracy (%)	F1-Score (%)	Average Detection Time (s)
Reentrancy	VaaS	82.54	73.95	159.4
	Oyente	61.62	44.96	29.6
	Smartcheck	52.97	30.10	14.5
	ContractFuzzer	67.89	52.67	352.2
	TMP	84.48	74.15	2.5
Integer overflow	VaaS	86.80	80.10	159.4
	Oyente	66.85	59.3	29.6
	Smartcheck	58.48	54.96	14.5
	ContractFuzzer	—	—	—
	TMP	—	—	—
Timestamp dependency	VaaS	89.20	82.46	159.4
	Oyente	59.45	41.53	29.6
	Smartcheck	51.32	40.18	14.5
	ContractFuzzer	68.08	52.49	352.2
	TMP	83.45	79.19	2.1

tools leads to the low efficiency of vulnerability analysis. According to the evaluation results in Table IV, the average detection time of VaaS and ContractFuzzer is 159.4 seconds and 352.2 seconds, respectively. In comparison, the average detection time consumed by Oyente and Smartcheck is 29.6 seconds and 14.5 seconds respectively. It is worth noting that TMP has a floating detection time for various vulnerabilities due to its different models for different vulnerabilities. For example, TMP takes 2.5 seconds for detecting reentrancy vulnerability, while only needs 2.1 seconds for timestamp dependency. Based on the above analysis, we summarize these five detection tools as follows.

- **VaaS** achieves high accuracy and F1-Score in the detection of the three vulnerabilities, but its average detection time is relatively long. VaaS is a "one-click" formal verification platform, which employs a variety of formal verification methods and has the characteristics of high accuracy, verification efficiency, and automation.
- **Oyente** is a contract analysis tool based on symbolic execution, which has an insufficient detection degree on the three vulnerabilities. Oyente has a high probability of leading to high false-negative and false-positive by simplifying the loop processing in the contract and the judgment based on rule matching.
- **Smartcheck** is a static tool that uses the XML-based intermediate representation to express and analyze smart contract security issues. However, it relies on inherent and simple logic rules, which cause high false-positive for smart contract vulnerability detection, resulting in low accuracy and F1- Score. It is worth mentioning that it takes relatively little time to detect vulnerabilities due to depending on rigid rules.
- **ContractFuzzer** is a smart contract security vulnerability fuzzing tool based on the Ethereum platform. Experimentally, it only supports detecting reentrancy and timestamp dependency vulnerabilities. Since fuzzing use cases have

a limit on covering all the execute paths, it cannot achieve the ideal path coverage. In addition, ContractFuzzer runs on the Ethereum platform so it also needs to get a response from Ethereum when conducting the detection, which takes a lot of time to perform one detection.
- **TMP** is a novel smart contract vulnerability detection model based on a graph neural network, which has the characteristics of high scalability, accuracy, and batch detection. Technically, TMP can detect reentrancy and timestamp dependency, achieving encouraging results. Due to using the pre-trained detection model, the average detection time of TMP is very low. Moreover, TMP is the first to explore combing deep learning technology and employing graph neural networks in smart contract vulnerability detection, which greatly improves efficiency and accuracy.

C. Evaluation and Improvement

1) Limitation Analysis: Although current smart contract vulnerability detection methods can detect smart contract vulnerabilities, they still have inherent limitations. This subsection specifically analyzes and discusses the aforementioned five types of vulnerability detection methods.

- The formal verification method uses some mathematical means to deduce and prove the smart contract in its life cycle, which requires interactive verification and judgment, so the degree of automation is low. Moreover, it relies on manual secondary verification, which makes it incompatible with EVM Execution layer vulnerability. At the same time, because the formal verification method relies on rigorous mathematical derivation and verification, it cannot perform dynamic analysis. Thus, it lacks the detection and judgment of the executable path in the contract, resulting in a high false positive rate and leakage. For example, F* framework and KEVM convert smart contract bytecode into formal models and verify various

properties in the contract code to detect vulnerabilities, they are still semi-automatic. ZEUS and VaaS have well-implemented fully automatic formal verification, but the vulnerabilities detected by them do not have a reachable execution path, producing a high false positive rate.

- The symbolic execution method uses symbols to replace specific execution program instructions, collect path constraints, and traverse all executable paths in the contract program. Although this method effectively improves the detection effect of symbolic execution, it also significantly increases the computational resources and time overhead in the vulnerability analysis process. In addition, they cannot completely solve the problems of state space explosion and exponential growth of execution paths. For example, Oyente and Maian limit the number of loop conditions to improve efficiency in order to prevent the problem of path explosion, but it also leads to high leakage. It is worth pointing out that many symbolic execution methods cannot be fully automated, and also require human assistance and feedback.

- Fuzzing methods largely rely on well-designed test cases, which monitor the abnormal behavior of contracts during dynamic execution. However, fuzzing has limited insight into the specific semantic code that leads to vulnerabilities, which makes it difficult to It is difficult to track down the exact code location where the vulnerability exists. For example, ContractFuzzer effectively reduces the false positive rate, but cannot achieve the ideal path coverage due to the randomness of its test case generation, making it difficult to find all potential threats.

- Intermediate representations use control flow, data flow, and taint analysis to review contracts by converting the original smart contract into a corresponding intermediate representation, but they often rely on predefined semantic rules or analysis lists, making it impossible to detect smart contracts that have complex business logic. In addition, they cannot traverse the execution paths that may exist in the contract. For example, Slither's intermediate representation SlithIR relies on fixed semantic rules and lacks formal semantics, which is limited to performing more detailed security analysis, so it cannot accurately detect the corresponding vulnerabilities. Smartcheck relies on rigid and simple predefined rules, so it cannot detect some contract vulnerabilities verified by taint analysis or dynamic execution.

- The deep learning method usually preprocesses smart contracts to construct a data set that is conducive to model learning. For example, the literature [100] uses the LSTM model to process the source sequence fragments of the smart contracts, and the literature [94] processes the smart contracts through the GNN model. However, on one hand, these methods cannot highlight the key variables in the source code of smart contracts, resulting in insufficient semantic modeling and unsatisfactory detection results. On the other hand, due to the black-box nature of neural networks, their interpretability is poor in most cases,

that is, they cannot give the exact location or code line where there may be a vulnerability like traditional detection tools. For example, TMP is an end-to-end vulnerability detection model, with contract testing Set as input, output the corresponding vulnerability detection results, its intermediate processing flow is a black box, so its interpretability is poor, and the detection results are unconvincing.

2) Research challenges and ideas for improvement: In view of the problems of the existing smart contract vulnerability detection methods, this section discusses and analyzes the research challenges and improvement ideas they face, mainly focusing on the following five aspects.

(1) Most of the existing formal verification technology research work is not highly automated, and the detected vulnerabilities may not have an accessible program path. Current formal verification methods use mathematical derivation to analyze contracts that may have complex vulnerabilities, which can be difficult. In addition, the detection of broader smart contract vulnerabilities using formal verification techniques still faces serious challenges. Future research should customize the corresponding verification for different vulnerability detection target specification descriptions, break through the technical limitations such as its inability to adapt to large-scale contracts and multiple vulnerability types, and expand the application scope of formal verification. Realize the transition from verifying general functional attributes and security attributes, and detecting common vulnerabilities to gradually solving the analysis and verification of smart contract vulnerabilities in complex business logic in business scenarios.

(2) The main challenge of symbolic execution currently is the problem of state space explosion and the exponential growth of execution paths. A feasible method in the future is to combine the audit experience of existing symbolic execution tools and vulnerability analysis to find high-risk smart contracts that are prone to vulnerabilities. Instructions, such as `SUICIDE`, `CALL`, `DELEGATECALL`, `ORIGIN`, `ASSERT`, define the paths involving these opcodes as key paths. In order to improve the efficiency of vulnerability detection, it is not necessary to check all possible execution paths in a specific practice, only symbolically execute the focused paths and perform vulnerability verification, thereby reducing the path space.

(3) Compared with traditional applications, smart contracts have many unique variables and functions, which brings new challenges to fuzzing smart contracts.

First, due to the characteristics of the state updating of smart contracts, it is extremely difficult to generate effective test cases. The traditional program fuzzing scheme only considers a single test case when generating test cases, so it is not suitable for smart contracts. Second, smart contracts run on virtual machines, and the reasons for their vulnerabilities are quite different from traditional programs. They neither cause program crashes nor do they have many common features for vulnerability detection. The origin of these vulnerabilities may come from different levels such as blockchain, virtual

machines, and high-level language, and there are many differences between them, which also brings great challenges to the vulnerability detection of smart contracts.

Specifically, fuzzing relies on the robustness of its test cases, so it is necessary to further improve the existing test case generation algorithms. For example, using multi-objective optimization algorithms. In addition, fuzzing can also consider combining other detection methods to improve detection efficiencies, such as adopting a strategy that combines static analysis or symbolic execution. For example, static analysis is used to extract critical paths, and test cases are generated through symbolic execution, thereby improving the efficiency of fuzzing.

(4) Intermediate representation method usually converts smart contract source code or bytecode into a unique intermediate representation and then detects certain types of vulnerabilities based on the IR. At the same time, they also rely on vulnerability rules defined by experts. But these rules are often rigid and simple, which are easy to be exploited by attackers. Therefore, in order to improve the expansibility and adaptability of this kind of vulnerability detection method, researchers should focus on making the intermediate representation of smart contracts more universal, so that the unified representation of different smart contracts can be considered while detecting various types of vulnerabilities. In addition, the combination of static analysis and dynamic execution is an effective method to improve the accuracy of vulnerability detection. Currently, most of the detection methods based on intermediate representations are static analysis, which lacks the use of dynamic execution for verification. Therefore, this is not only the key challenge currently faced by intermediate representations but also the main direction of future research.

(5) Most of the existing deep learning-based smart contract vulnerability detection methods are black-box detection processes, which give the final vulnerability detection results by training vulnerability detection models. Due to the inherent black-box nature of the deep learning model, its internal specific working status and processing process are opaque, so there is a lack of reasonable explanations for the vulnerability detection results (such as labeling the exact code location or code line where there may be vulnerabilities), which results in the detection result being unconvincing. Therefore, the deep learning model should consider giving a reasonable explanation of its interpretability while outputting the vulnerability detection results. It is worth mentioning that the expert rules defined in traditional detection tools are also powerful tools for analyzing contract vulnerabilities, and future deep learning models should consider integrating the expert rules related to vulnerabilities in traditional detection methods to better improve the accuracy of vulnerability detection.

V. Discussion and conclusion

Smart contracts, as one of the most successful applications of the blockchain, have greatly expanded the application scenarios and practical significance of blockchain, playing a vital role in the blockchain ecosystem. With the maturity of

blockchain technology and the prevalence of smart contracts, the security and reliability of smart contracts have become an increasingly important hot research topic. This survey summarizes the current common security vulnerabilities in Ethereum smart contracts and restores typical cases in the history of smart contract security. To prevent the occurrence of contract vulnerability, researchers have proposed a series of smart contract vulnerability detection methods, which are concluded into five categories: formal verification, symbolic execution, fuzzing detection, intermediate representation, and deep learning. We introduce and analyze the principles and characteristics of various methods in detail. Furthermore, we compare and evaluate the detectable vulnerability types and performance of representative smart contract automated detection tools.

Technically, the automated vulnerability detection methods are able to deal with the endless smart contract vulnerabilities accurately and efficiently, reducing the false-positive rate and false-negative rate that may be caused by manual verification and analysis. Therefore, it is of great significance to employ a precise and effective smart contract detection method to solve the problem of contract vulnerability mining. This survey analyzes various research methods and points out that although existing efforts have made breakthroughs and encouraging results in the field of smart contract vulnerability detection, current detection methods are not perfect and face the following key problems.

- **Low accuracy of vulnerability detection.** Currently, most smart contract detection tools still catch a high false-positive rate and false-negative rate. Take the evaluations of smart contract vulnerability detection tools in section IV as an example, VaaS and TMP both achieve high accuracy, which is more than 80%, while the accuracy of the other three detection tools is only around 60%, which is far from satisfying the current application scenarios of a large number of contracts and various smart contract vulnerabilities. Therefore, the accuracy of smart contract vulnerability detection tools is a key issue that needs to be improved.

- **Low coverage of vulnerability type.** Due to the various and complex smart contract vulnerabilities, most detection tools fail to cover all the varieties of vulnerabilities. For example, see the evaluations of section IV, Securify can detect most types of vulnerabilities. However, some other detection tools only support a single vulnerability or verify low-level security issues that lack monitoring and inference during the contract execution so it is difficult to discover and locate cross-contract vulnerabilities. Therefore, making detection tools cover more comprehensive smart contract vulnerabilities is also a crucial challenge.

- **Time-consuming of vulnerability audit.** Efficient and fast audit of smart contract vulnerabilities is also a key element to ensure contract security. Current detection tools have low efficiency in vulnerability mining, which

hinders the development and expansion of smart contracts. For example, the average detection time of Mythril is 225.6 seconds, and VaaS is about 159.4 seconds while ContractFuzzer takes about 352.2 seconds. Thus, under the background of the ever-increasing number of smart contracts, ensuring the efficiency of vulnerability auditing is also a difficulty that needs to resolve urgently.

- **Automation level of vulnerability detection.** The characteristic of full automation is also an essential part of ensuring the efficiency of vulnerability detection. It needs to point out that existing detection tools are not fully automated, such as F^* and KEVM framework. Besides, some automatic detection tools (e.g., Securify and Smartcheck) are unable to clarify whether there are vulnerabilities in the output contract, which require the manual classification for the detected suspected vulnerabilities, increasing the workload in the smart contract detection process and reducing the detection efficiency to a large extent. Therefore, how to achieve a more comprehensive automated detection method need to resolve in future research.

- **Language diversity of smart contracts.** Specifically, there are many kinds of programming languages in the real world. At present, dozens of languages can be used to implement smart contracts, such as Solidity, Go, C, Java, and so on. Different languages, however, have different syntax and semantic rules, which results in different contract structures. How to make smart contract vulnerability detection tools adapt to most programming languages is also challenging and difficult.

Although there still exist many difficulties and challenges in the current development of smart contract vulnerability detection, it is also an opportunity to explore novel technology for opening up a new direction. In recent years, researchers have combined deep learning models to detect smart contract vulnerabilities, making encouraging progress. In the following discussions, we look forward to future research directions and propose suggestions by integrating deep learning and smart contract vulnerability detection technology.

- **Constructing a unified and standardized smart contract vulnerability dataset.** First of all, if we want to make a breakthrough in the detection of smart contract vulnerabilities based on deep learning, we rely on the comprehensive dataset of smart contract vulnerabilities. Currently, due to the lack of a standard dataset, existing deep learning-based methods (such as RecChecker, TMP) can only support detecting a few contract vulnerabilities. Therefore, we need to construct a unified and standardized dataset that covers all the vulnerabilities as much as possible, making the deep learning model play a better role, and promoting the research in this field.

- **Building a comprehensive model for both static and dynamic analysis.** As we know, smart contract security vulnerability detection based on deep learning is still in its infancy (such as SaferSC, RecChecker, TMP). Most

methods can only analyze the contract at the level of static source code or bytecode. However, such static analysis tends to miss possible existing execution paths. In the meantime, due to the lack of dynamic interaction with external contracts, it usually leads to a high false-positive rate or false-negative rate. Therefore, in order to satisfy the requirements of significant application scenarios, we need to consider combining dynamic and static analysis to build a comprehensive deep learning model.

- **Training a reusable and scalable vulnerability detection model.** With the explosive growth of the number of smart contracts, the corresponding security vulnerabilities are becoming more and more complex and unpredictable. At present, existing vulnerability detection methods based on deep learning are focusing on training the models for the vulnerabilities that have been discovered. Therefore, whether they can quickly adapt to the new kinds of vulnerabilities still needs to be further studied. We believe that the rich security vulnerabilities in the open-source smart contract ecosystem should be fully utilized to build a reusable and scalable vulnerability detection model to cope with the new and endless smart contract vulnerabilities.

To summarize, the rapid development of blockchain technology provides a reliable and feasible execution environment for smart contracts. As smart contracts are popularized in various decentralized applications, the security issues of smart contracts are becoming more and more important. This survey sorts out the common smart contract vulnerabilities and compares the *accuracy*, *F1-Score*, and *average detection time* of existing vulnerability detection methods in detail. Furthermore, we give suggestions for the problems that exist in the current research work and discuss the challenges in future research, as well as the possibility of combing the recent achievements in deep learning technology, in order to inspire future research work.

REFERENCES

[1] M. Swan, *Blockchain: Blueprint for a new economy.* " O'Reilly Media, Inc.", 2015.

[2] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey," *International Journal of Web and Grid Services*, vol. 14, no. 4, pp. 352–375, 2018.

[3] L. S. Sankar, M. Sindhu, and M. Sethumadhavan, "Survey of consensus protocols on blockchain applications," in *2017 4th international conference on advanced computing and communication systems (ICACCS)*. IEEE, 2017, pp. 1–5.

[4] Q. Xia, E. B. Sifah, A. Smahi, S. Amofa, and X. Zhang, "Bbds: Blockchain-based data sharing for electronic medical records in cloud environments," *Information*, vol. 8, no. 2, p. 44, 2017.

[5] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman, "Medrec: Using blockchain for medical data access and permission management," in *2016 2nd international conference on open and big data (OBD)*. IEEE, 2016, pp. 25–30.

[6] P. Zhang, D. C. Schmidt, J. White, and G. Lenz, "Blockchain technology use cases in healthcare," in *Advances in computers*. Elsevier, 2018, vol. 111, pp. 1–41.

[7] Z. Meng, T. Morizumi, S. Miyata, and H. Kinoshita, "Design scheme of copyright management system based on digital watermarking and blockchain," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2. IEEE, 2018, pp. 359–364.

[8] M. Holland, C. Nigischer, and J. Stjepandić, "Copyright protection in additive manufacturing with blockchain approach," in *Transdisciplinary Engineering: A Paradigm Shift.* IOS Press, 2017, pp. 914–921.

[9] P. Qian, Z. Liu, X. Wang, J. Chen, B. Wang, and R. Zimmermann, "Digital resource rights confirmation and infringement tracking based on smart contracts," in *2019 IEEE 6th International Conference on Cloud Computing and Intelligence Systems (CCIS).* IEEE, 2019, pp. 62–67.

[10] S. Saberi, M. Kouhizadeh, J. Sarkis, and L. Shen, "Blockchain technology and its relationships to sustainable supply chain management," *International Journal of Production Research*, vol. 57, no. 7, pp. 2117–2135, 2019.

[11] S. A. Abeyratne and R. P. Monfared, "Blockchain ready manufacturing supply chain using distributed ledger," *International journal of research in engineering and technology*, vol. 5, no. 9, pp. 1–10, 2016.

[12] S. Chen, R. Shi, Z. Ren, J. Yan, Y. Shi, and J. Zhang, "A blockchain-based supply chain quality management framework," in *2017 IEEE 14th International Conference on e-Business Engineering (ICEBE).* IEEE, 2017, pp. 172–176.

[13] E. Mengelkamp, B. Notheisen, C. Beer, D. Dauer, and C. Weinhardt, "A blockchain-based smart grid: towards sustainable local energy markets," *Computer Science-Research and Development*, vol. 33, no. 1, pp. 207–214, 2018.

[14] C. Pop, T. Cioara, M. Antal, I. Anghel, I. Salomie, and M. Bertoncini, "Blockchain based decentralized management of demand response programs in smart energy grids," *Sensors*, vol. 18, no. 1, p. 162, 2018.

[15] F. Knirsch, A. Unterweger, G. Eibl, and D. Engel, "Privacy-preserving smart grid tariff decisions with blockchain-based smart contracts," in *Sustainable Cloud and Energy Services.* Springer, 2018, pp. 85–116.

[16] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *Ieee Access*, vol. 4, pp. 2292–2303, 2016.

[17] A. Bahga and V. K. Madisetti, "Blockchain platform for industrial internet of things," *Journal of Software Engineering and Applications*, vol. 9, no. 10, pp. 533–546, 2016.

[18] N. Kshetri, "Can blockchain strengthen the internet of things?" *IT professional*, vol. 19, no. 4, pp. 68–72, 2017.

[19] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An overview of blockchain technology: Architecture, consensus, and future trends," in *2017 IEEE international congress on big data (BigData congress).* Ieee, 2017, pp. 557–564.

[20] R. Wang, Z. Lin, and H. Luo, "Blockchain, bank credit and sme financing," *Quality & Quantity*, vol. 53, no. 3, pp. 1127–1140, 2019.

[21] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, no. 37, pp. 2–1, 2014.

[22] S. Jani, "Smart contracts: Building blocks for digital transformation," *Indira Gandhi National Open University*, 2020.

[23] "Etherscan," 2014, https://etherscan.io/.

[24] "Bcsec," 2018, https://bcsec.org/.

[25] "Slowmist," 2018, https://hacked.slowmist.io/.

[26] "The dao," 2016, https://en.wikipedia.org/wiki/TheDAO(organization)/.

[27] "Parity multisig bug," 2017, https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug//.

[28] "Beautychain integer overflow," 2018, https://etherscan.io/token/0xc5d105e63711398af9bbff092d4b6769c82f793d.

[29] "Eos official portal," 2019, https://eos.io/.

[30] "Farmeos," 2018, https://bcsec.org/index/detail/id/456.

[31] "Playgames," 2019, https://bcsec.org/index/detail/id/459.

[32] "Luckbet," 2019, https://bcsec.org/index/detail/id/461.

[33] "Eoslots," 2019, https://bcsec.org/index/detail/id/477.

[34] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.

[35] B. K. Mohanta, S. S. Panda, and D. Jena, "An overview of smart contract and use cases in blockchain technology," in *2018 9th international conference on computing, communication and networking technologies (ICCCNT).* IEEE, 2018, pp. 1–4.

[36] D. Macrinici, C. Cartofeanu, and S. Gao, "Smart contract applications within blockchain technology: A systematic mapping study," *Telematics and Informatics*, vol. 35, no. 8, pp. 2337–2354, 2018.

[37] R. J. X. Bao, "Mimic blockchain—solution to the security of blockchain," vol. 30, no. 6, p. 1681, 2019.

[38] M. FU, L. WU, Z. HONG, and W. FENG, "Research on vulnerability mining technique for smart contracts," *Journal of Computer Applications*, vol. 39, no. 7, p. 1959, 2019.

[39] H. Wang, F. Zhang, L. I. Tian, M. Gao, and D. U. Xinyu, "Security and privacy-protection technologies in smart contract."

[40] L. Z.-G. QIAN Peng, "Smart contract vulnerability detection technique: A survey," *Journal of Software*, vol. 33, no. 8, p. 3059, 2022.

[41] D. He, Z. Deng, Y. Zhang, S. Chan, and N. Guizani, "Smart contract vulnerability analysis and security audit," *IEEE Network*, vol. PP, no. 99, pp. 1–7, 2020.

[42] "Dune analytics," 2020, https://www.duneanalytics.com/.

[43] "Hard-fork," 2020, https://www.investopedia.com/terms/h/hard-fork.asp/.

[44] 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," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–28, 2017.

[45] "Safemath," 2020, https://docs.statechannels.org/contract-api/natspec/SafeMath/.

[46] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security analysis methods on ethereum smart contract vulnerabilities: a survey," *arXiv preprint arXiv:1908.08605*, 2019.

[47] "Replay attack," 2017, https://github.com/sheharbano/byzcuit/tree/replay-attacks.

[48] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International conference on principles of security and trust.* Springer, 2017, pp. 164–186.

[49] "Koet," 2017, https://www.kingoftheether.com/thrones/kingoftheether/index.html/.

[50] "Etherscan," https://etherscan.io/token/0x317eb3b357e5cb2c94dca5586f018d594d3d8091, 2018.

[51] H. Songming, B. LIANG, J. HUANG, and S. Wenchang, "Dc-hunter: Detecting dangerous smart contracts via bytecode matching," *Journal of Cyber Security*, vol. 5, no. 3, pp. 100–112, 2020.

[52] 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.

[53] X. Bai, Z. Cheng, Z. Duan, and K. Hu, "Formal modeling and verification of smart contracts," in *Proceedings of the 2018 7th international conference on software and computer applications*, 2018, pp. 322–326.

[54] T. Abdellatif and K.-L. Brousmiche, "Formal verification of smart contracts based on users and blockchain behaviors models," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS).* IEEE, 2018, pp. 1–5.

[55] 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," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 2019, pp. 1186–1189.

[56] E. Shishkin, "Debugging smart contract's business logic using symbolic model checking," *Programming and Computer Software*, vol. 45, no. 8, pp. 590–599, 2019.

[57] W. ZHAO, W. ZHANG, J. WANG, H. WANG, and C. WU, "Smart contract vulnerability detection scheme based on symbol execution," *Journal of Computer Applications*, vol. 40, no. 4, p. 947, 2020.

[58] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for software security testing and quality assurance.* Artech House, 2018.

[59] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[60] Y. Wang, P. Jia, L. Liu, C. Huang, and Z. Liu, "A systematic review of fuzzing based on machine learning techniques," *PloS one*, vol. 15, no. 8, p. e0237749, 2020.

[61] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formalizing the llvm intermediate representation for verified program transformations," in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2012, pp. 427–440.

[62] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE).* IEEE, 2016, pp. 297–308.

[63] Y. Shi, Y. E. Sagduyu, K. Davaslioglu, and R. Levy, "Vulnerability detection and analysis in adversarial deep learning," in *Guide to Vulnerability Analysis for Computer Networks and Systems.* Springer, 2018, pp. 211–234.

[64] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *2017 3rd IEEE international conference on computer and communications (ICCC)*. IEEE, 2017, pp. 1298–1302.

[65] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[66] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[67] J. Deutch, E. Moniz, S. Ansolabehere, M. Driscoll, P. Gray, J. Holdren, P. Joskow, R. Lester, and N. Todreas, "The future of nuclear power," *an MIT Interdisciplinary Study, http://web. mit. edu/nuclearpower*, 2003.

[68] A. P. Mouritz, *Introduction to aerospace materials*. Elsevier, 2012.

[69] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *International Conference on Principles of Security and Trust*. Springer, 2018, pp. 243–269.

[70] 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.

[71] S. Amani, M. Bégel, M. Bortin, and 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.

[72] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: analyzing safety of smart contracts." in *Ndss*, 2018, pp. 1–12.

[73] I. Garfatta, K. Klai, W. Gaaloul, and M. Graiet, "A survey on formal verification for solidity smart contracts," in *2021 Australasian Computer Science Week Multiconference*, 2021, pp. 1–10.

[74] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[75] B. Mueller, "A framework for bug hunting on the ethereum blockchain," 2017.

[76] C. F. Torres, J. Schütte, and 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.

[77] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017, pp. 442–446.

[78] L. d. Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[79] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th annual computer security applications conference*, 2018, pp. 653–663.

[80] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.

[81] J. Krupp and C. Rossow, "{teEther}: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1317–1333.

[82] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," *arXiv preprint arXiv:1812.05934*, 2018.

[83] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 259–269.

[84] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and 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.

[85] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 531–548.

[86] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WET-SEB)*. IEEE, 2019, pp. 8–15.

[87] 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," *arXiv preprint arXiv:1809.03981*, 2018.

[88] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[89] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Ethir: A framework for high-level analysis of ethereum bytecode," in *International symposium on automated technology for verification and analysis*. Springer, 2018, pp. 513–520.

[90] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[91] X. Wang, J. He, Z. Xie, G. Zhao, and S.-C. Cheung, "Contractguard: Defend ethereum smart contracts with embedded intrusion detection," *IEEE Transactions on Services Computing*, vol. 13, no. 2, pp. 314–328, 2019.

[92] 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," *arXiv preprint arXiv:1811.06632*, 2018.

[93] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International conference on machine learning*. PMLR, 2017, pp. 1263–1272.

[94] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network." in *IJCAI*, 2020, pp. 3283–3290.

[95] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.

[96] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[97] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "Contractward: Automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 1133–1144, 2020.

[98] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang, "Combining graph neural networks with expert knowledge for smart contract vulnerability detection," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2021. [Online]. Available: https://doi.org/10.1109%2Ftkde.2021.3095196

[99] Z. Liu, P. Qian, X. Wang, L. Zhu, Q. He, and S. Ji, "Smart contract vulnerability detection: from pure neural network to interpretable graph feature and expert pattern fusion," *arXiv preprint arXiv:2106.09282*, 2021.

[100] P. Qian, Z. Liu, Q. He, R. Zimmermann, and X. Wang, "Towards automated reentrancy detection for smart contracts based on sequential models," *IEEE Access*, vol. 8, pp. 19 685–19 695, 2020.