Computers, Materials & Continua

Tech Science Press

DOI: 10.32604/cmc.2024.052887

REVIEW

A Systematic Review and Performance Evaluation of Open-Source Tools for Smart Contract Vulnerability Detection

Yaqiong He, Jinlin Fan[*] and Huaiguang Wu

School of Computer Science and Technology, Zhengzhou University of Light Industry, Zhengzhou, 450001, China

*Corresponding Author: Jinlin Fan. Email: fjl707235887@gmail.com

Received: 18 April 2024 Accepted: 30 May 2024 Published: 18 July 2024

ABSTRACT

With the rise of blockchain technology, the security issues of smart contracts have become increasingly critical. Despite the availability of numerous smart contract vulnerability detection tools, many face challenges such as slow updates, usability issues, and limited installation methods. These challenges hinder the adoption and practicality of these tools. This paper examines smart contract vulnerability detection tools from 2016 to 2023, sourced from the Web of Science (WOS) and Google Scholar. By systematically collecting, screening, and synthesizing relevant research, 38 open-source tools that provide installation methods were selected for further investigation. From a developer's perspective, this paper offers a comprehensive survey of these 38 open-source tools, discussing their operating principles, installation methods, environmental dependencies, update frequencies, and installation challenges. Based on this, we propose an Ethereum smart contract vulnerability detection framework. This framework enables developers to easily utilize various detection tools and accurately analyze contract security issues. To validate the framework's stability, over 1700 h of testing were conducted. Additionally, a comprehensive performance test was performed on the mainstream detection tools integrated within the framework, assessing their hardware requirements and vulnerability detection coverage. Experimental results indicate that the Slither tool demonstrates satisfactory performance in terms of system resource consumption and vulnerability detection coverage. This study represents the first performance evaluation of testing tools in this domain, providing significant reference value.

KEYWORDS

Blockchain security; ethereum; smart contracts detection; tools evaluation

1 Introduction

Blockchain is a distributed ledger that effectively addresses trust issues related to centralized authorities. In a blockchain network, multiple nodes collaborate to maintain a shared transaction record in a decentralized manner, eliminating the need for a trusted third party. In 2008, Nakamoto introduced Bitcoin [1], the first cryptocurrency to use blockchain as a foundational technology.

Smart contracts, a type of code written on the blockchain, are known for their ability to automatically execute contract terms based on predefined conditions. Computer scientist and cryptographer Szabo first proposed the concept of smart contracts in 1995 [2]. Implementing smart contracts requires a trusted environment, which blockchain technology provides by offering a transparent and consensus-driven operating environment without the need for third-party involvement. Initially, smart contracts were used for asset transfers through Bitcoin scripts [3,4]. In 2013, Buterin introduced Ethereum, a blockchain platform specifically designed for smart contracts [5]. This platform uses the Solidity language to write smart contracts, which are executed on the Ethereum Virtual Machine, marking the beginning of the Blockchain 2.0 era [6].

Smart contracts are software programs based on blockchain technology that execute in a decentralized manner. Like any other computer program, smart contracts are susceptible to code vulnerabilities. Additionally, smart contracts possess unique characteristics compared to traditional code, such as public visibility [7] and tamper resistance [8]. Therefore, smart contracts face significant security risks.

In recent years, the field of smart contracts has experienced numerous severe security incidents, resulting in significant economic losses and impacting the development of the blockchain industry. Research indicates that many smart contracts deployed on the Ethereum blockchain suffer from programming flaws [9,10], making them vulnerable to attacks. For instance, the 2016 case of the Decentralized Autonomous Organization (DAO) [11,12] smart contract demonstrated a reentrancy vulnerability that allowed malicious users to repeatedly call the same function, leading to repeated withdrawals of funds and causing millions of dollars in losses. Another example is the 2017 [13] incident involving the Parity multi-signature wallet project, where tens of millions of dollars worth of Ether was stolen, inflicting significant losses on users and the blockchain community. Additionally, the 2018 Beauty Chain (BEC) contract vulnerability incident [14] further illustrates the widespread existence of smart contract security issues. These incidents not only resulted in economic losses but also harmed user trust and the overall development of the blockchain industry. The security issues surrounding smart contracts have become a formidable challenge.

Despite numerous research efforts and improvements in smart contract security over the past few years, vulnerability detection remains a daunting challenge [15] due to the complexity and innovation of smart contracts.

Currently, several smart contract vulnerability analysis tools have been proposed in academia. These tools are significant for smart contract security. However, they face challenges such as slow updates and difficulty of use, which pose serious challenges to smart contract developers. Therefore, further research is necessary to expand and promote the practical application of these tools. By doing so, we can enhance the security of smart contracts and safeguard users' assets against vulnerability attacks more effectively.

For a comprehensive survey, this paper searches literature databases such as Google Scholar and Web of Science, retrieving a total of 780 articles from 2016 to 2023 on the detection of vulnerabilities in smart contracts. We then selected 141 articles related to tools and narrowed it down to 38, focusing on open-source and installable smart contract vulnerability detection tools.

The main contributions of this paper are as follows:

1. A summary of various techniques for analyzing smart contract code, including a detailed analysis of the underlying principles of 38 open-source smart contract vulnerability detection tools.

2. A classification of analysis tools based on their level of installation difficulty. Additionally, an extensible vulnerability detection framework is proposed, based on the most commonly used vulnerability detection tools, which is convenient for developers.

3. Comprehensive performance testing of the detection tools within the framework to determine the corresponding hardware requirements.

The structure of this paper is as follows: Section 2 provides an overview of Ethereum and smart contracts, including current mainstream vulnerability classifications and popular detection methods. Section 3 details open-source Ethereum smart contract analysis tools. Section 4 introduces a vulnerability detection framework based on a hybrid cloud platform, describing its architecture and implementation. Section 5 evaluates the performance of the tools within the framework, including running speed, multi-core optimization, and memory usage. Section 6 summarizes the paper and offers suggestions for future work.

2 Structure

2.1 Ethereum and Smart Contracts

Ethereum is an open-source platform based on blockchain technology. Unlike other cryptocurrencies such as Bitcoin, Ethereum not only facilitates digital currency transactions but also enables developers to build and deploy smart contracts. Smart contracts are programs that run on the blockchain, automatically executing predetermined conditions and operations. Ethereum's smart contracts utilize the Turing-complete programming language Solidity, allowing developers to write complex logic and algorithms. This makes Ethereum more than just a digital currency; it is also a powerful decentralized computing platform.

One of Ethereum's core components is the Ethereum Virtual Machine (EVM) [16], a stack-based virtual machine used to execute code for Ethereum smart contracts. The EVM uses an intermediate language called Ethereum bytecode to represent smart contract code. After developers write smart contract code in Solidity, it is compiled into Ethereum bytecode and deployed onto the Ethereum network.

The execution process of the EVM is stack-based. Each smart contract has its own storage and stack spaces. When executing a smart contract, the EVM sequentially reads the code line by line and places it into the stack. Instructions can include mathematical operations, conditional statements, jumps, storage, and loading operations. While executing instructions, the EVM manipulates the stack based on the opcode and operands of each instruction. For example, when executing an addition instruction, the EVM pops two operands from the stack, adds them together, and pushes the result back onto the stack.

The execution result of a smart contract is recorded on the Ethereum blockchain. Once a smart contract completes execution, it can change its own state and send messages to other smart contracts or call methods of other smart contracts. The execution result of a smart contract can also be recorded as part of a transaction on the Ethereum blockchain.

The emergence of smart contracts in the field of blockchain has garnered widespread attention and discussion. Traditional centralized institutions encounter several issues in areas such as financial transactions and contract execution, including high fees, lack of transparency, and security concerns. However, the decentralized nature of smart contracts built on blockchain offers a novel solution to these problems. Decentralization allows anyone to participate in the operation of smart contracts without relying on the trust of intermediaries. This has brought about more efficient, transparent, and secure solutions in various fields such as finance [17,18], supply chain management [19], Internet of Things [20–23],construction [24], and energy [25].

2.2 Security Vulnerabilities in Smart Contracts

Smart contracts have introduced numerous innovations and conveniences across various industries within the realm of blockchain technology. By enabling automated contract execution, they eliminate intermediaries and the issues of distrust commonly associated with traditional contracts. However, the security of smart contracts has always been a pressing concern. Security vulnerabilities primarily refer to logical flaws that arise during the design process of smart contracts. Once deployed on the blockchain, these logical flaws can easily lead to security issues, such as financial losses for users. Moreover, since smart contracts operate on a blockchain, once deployed, they cannot be altered, making security issues potentially consequential.

Currently, the most widely used programming language for smart contracts on the Ethereum platform, the largest smart contract operating platform, is Solidity. Solidity is based on a Turing-complete language [26], which offers flexibility and powerful features that allow developers to implement complex logic and business operations. However, this flexibility also provides opportunities for attackers to discover and exploit vulnerabilities.

Researchers have proposed classifications of vulnerabilities for Ethereum smart contracts. Li et al. [27] conducted a survey and analysis of six real attacks targeting blockchains. Nicola et al. [28] were the first to categorize smart contract vulnerabilities into three classes and studied twelve vulnerabilities related to smart contracts. Zhu et al. [29] identified forty different vulnerabilities. As shown in Table 1, we summarize the current common smart contract vulnerabilities.

Table 1: Introduction to common vulnerabilities in smart contract

Vulnerability	Description
Reentrancy	When a contract transfers Ether to an external contract, an attacker can construct a contract with malicious code in the external address's fallback function. The attacker then requests the contract to transfer Ether again to the external contract. Since the first transfer has not yet completed, the state variables of the original contract remain unchanged and are still marked as not withdrawn. As a result, the attacker can repeatedly withdraw funds until all the funds in the original contract have been taken out.
Integer overflow	During the execution of smart contracts, the EVM defines fixed-size integer data types. However, when an integer exceeds its designated size, integer overflow vulnerabilities can occur, including multiplication overflow, subtraction overflow, and addition overflow, among others. Since the parameters involved in these operations are often user inputs, if the computed result exceeds the storage range and lacks robust detection statements, attackers can exploit these vulnerabilities to bypass conditional checks or directly manipulate data.

(Continued)

Table 1 (continued)

Vulnerability	Description
Unchecked low level calls	In Solidity, there are two functions that can be used to enable transfer functionality: transfer() and send(). The difference between them lies in error handling and return values. When a transfer fails, the transfer() function throws an exception and rolls back the entire transaction, thereby aborting the execution of the contract. This helps avoid unnecessary code execution. On the other hand, the send() function, when a transfer fails, does not throw an exception but instead returns false, indicating that the contract can continue executing other code.
Tx.origin	In Ethereum smart contracts, Tx.origin is a global variable that represents the initiator of a transaction. It traverses the entire call stack and returns the address of the account that originally called the function. However, Tx.origin poses security risks as attackers can exploit the Tx.origin vulnerability to launch impersonation attacks, deceiving the contract into executing uncontrolled operations.
Time manipulation	In blockchain systems, generating entropy is not possible due to the deterministic and lack of external input or random events. As a result, Solidity, the programming language for smart contracts in blockchain systems, does not provide a rand() function or similar random number generation capability. In scenarios where random numbers are required, developers often need to implement randomness through alternative means, such as utilizing block hash values or participant addresses within the blockchain system to generate pseudo-random numbers. Although certain information within the blockchain system may appear random to many developers, it is actually controllable by miners. Therefore, these variables cannot be considered as a source of entropy.
Delegatecall	To enhance code reusability, Solidity provides the Call and DelegateCall mechanisms for invoking external code. When using Call, the invocation is executed within the context of the called contract. On the other hand, when using DelegateCall, the code of the called contract is executed within the calling contract. When contract A utilizes DelegateCall to invoke contract B, the context of contract A is preserved, while the code of contract B is executed. During this process, functions within contract B can access and modify the storage and state of contract A.
Access control	In general, the default visibility of a function is public, which means it can be accessed both internally and externally within a contract. However, setting a function that should only be called internally as public can lead to unexpected outcomes. In the case of the first hacking attack on the Parity MultiSig Wallet, the attacker exploited a vulnerability where the function visibility was not specified. They called the initWallet() function and set themselves as the owner of the wallet. Because of this security loophole, users suffered losses amounting to a staggering $31 million.

(Continued)

Table 1 (continued)

Vulnerability	Description
TOD	Transaction Order Dependence (TOD) refers to the uncertainty in the execution results of transactions on the Ethereum blockchain due to the dependency between transactions. Miners have the option to include transactions from the transaction pool in the current block, usually ordered by gasPrice. Attackers can monitor the transaction pool, gather data, and create transactions with higher gasPrices, allowing them to be included in blocks faster than the original transactions.
Denial of service	In smart contracts, the commonly used data structures are mappings and arrays, which are used to record and store various types of information. However, if an array is maliciously expanded, it can result in excessive time and resource consumption when attempting to delete the array. This can lead to the deletion requiring more gas than the block's limit allows, thus making it impossible to remove the array and potentially resulting in a denial-of-service attack.

2.3 Common Detection Methods for Smart Contracts

Currently, the mainstream techniques for detecting vulnerabilities in smart contracts can be categorized into four types: feature matching, fuzz testing, symbolic execution, and taint analysis.

Feature matching is a static analysis-based technique for detecting vulnerabilities in smart contracts. It abstracts the features of malicious code [30,31] and detects smart contract source code or bytecode using similarity comparing to features, which are representated by intermediate layer, such as an Abstract Syntax Tree (AST) [32] or Control Flow Graph (CFG) [33].

Fuzz testing [34] is a unique vulnerability detection technique. It provides smart contracts with a large amount of random or exceptional data to trigger potential abnormal states and observe the results of contract execution to discover possible vulnerabilities. The core of fuzz testing lies in the generation and evaluation of test cases. Based on the current execution results and coverage information, it selects optimal test cases as inputs for the next round, thereby improving detection efficiency and quality.

Symbolic execution [35] is a program analysis technique that employs abstract symbols to represent program variables and simulates the execution of program instructions while collecting path constraints for each branch encountered. When encountering program branches, symbolic execution explores each branch path. Once a path is executed, the path constraints generated during symbolic execution are handed over to a satisfiability modulo theories (SMT) solver for resolution. If the path constraints can be solved by the SMT solver, concrete values that can execute the corresponding path are obtained. If the solution fails, it indicates that the path is unreachable. However, due to the need for symbolic execution to explore two program paths at each conditional branch, the number of program paths to be explored grows exponentially as the program size increases. This leads to the problem of path explosion, which is an inherent limitation of symbolic execution.

Taint analysis [36] is used to detect whether sensitive data or operations in smart contracts are influenced by external input or untrusted data, leading to security vulnerabilities. This technique tracks the generation, propagation, and elimination of tainted information in the contract to determine if it interferes with critical operations.

In recent years, deep learning-based techniques for smart contract vulnerability detection have received widespread attention and research. These techniques leverage deep neural networks to learn the semantic and structural features of smart contracts, enabling automatic identification and classification of vulnerability types. Compared to traditional approaches, this technology does not rely on manually defined vulnerability rules or patterns. It can handle large-scale smart contract data, adapt to different smart contract languages and platforms, and detect multiple types of vulnerabilities. Currently, deep learning-based detection techniques can be categorized into three main types: text processing, static analysis, and image recognition. Text processing-based deep learning methods employ natural language processing (NLP) techniques to extract semantic features from smart contracts, such as word embedding, sentence embedding, and attention mechanisms. Static analysis-based deep learning approaches utilize static analysis techniques to extract structural information from smart contracts, such as control flow graphs and abstract syntax trees. Image recognition-based deep learning methods convert smart contracts into images and employ convolutional neural networks (CNN) or other image processing models for feature extraction and classification to determine the presence and type of vulnerabilities. However, it is regrettable that most current deep learning-based methods suffer from usability challenges and lack corresponding installation and usage instructions. Therefore, they are beyond the scope of this paper's discussion.

3 Smart Contract Vulnerability Detection Tools

3.1 Systematic Literature Review

We searched for relevant works in Google Scholar and Web of Science using the keywords 'smart contract tools' and 'smart contract detection'. In the first stage, 781 articles were found.

In the second stage, we selected papers related to our work by defining a set of inclusion and exclusion rules, which are shown in Table 2. A total of 112 articles that meet the criteria were selected through reviewing abstracts and main findings from 781 literature pieces.

Table 2: Filter rules

Inclusion criteria	Exclusion criteria
Research field of smart contracts	White papers
Vulnerability detection tool	Review papers

In the third stage, we conducted detailed research and selected 38 open-source articles that provided installation methods out of the 112 articles about tools to be included in our research. The above work was independently completed by three individuals.

3.2 Brief Introductions

This section examines working principle and the methods used to identify targets in the 38 articles.

Vulpedia [37] is a static analysis tool developed using Python in 2022. Vulpedia identifies vulnerabilities using vulnerability signatures. It clusters contracts with the same vulnerabilities using abstract syntax trees and summarizes signatures through program dependency graphs. To build a feature library, it extracts vulnerability signatures from vulnerable contracts of the same type and benign signatures from benign contracts. If a vulnerable signature is matched and no benign signature is found, the contract is considered vulnerable.

SpCon [38] is a dynamic analysis tool that was released on 22 January, 2022. The tool utilises the role-based access control (RBAC) model to divide contract users into different roles, each with different permissions. Lower roles are unable to read or write data owned by higher roles, which necessitates user mining and identity determination. Existing role mining techniques often assume that user permission assignments are fully observed, but this is not the case in smart contracts. Therefore, this tool considers users with the same behavior to have the same role and uses the average frequency vector (AFV) for quantitative capture. The AFV measures how often users with a particular role execute different permissions and serves as the signature of the role. The similarity in AFV between different roles is low. Any discrepancy found between what is allowed and what is actually found indicates a potential permission flaw.

Published in 2022, xFuzz [39] is a vulnerability analysis tool that utilises machine learning and fuzz testing methods. The tool focuses on vulnerabilities that are most likely to cause cross-contract problems. As multiple contracts interacting increases the search space, machine learning methods are used to guide the fuzz testing of smart contracts. The contract vulnerability is transformed into a vector using Word2Vec and combined with the contract's static characteristics. This combined information is then used as input for the machine learning model to train it. The trained model is then used to determine the potential location of the vulnerability in the contract. This approach reduces the search space and improves efficiency.

ESBMC [40] is a context-bounded model checker based on SMT that was published in 2022. The tool initially receives the source code of the smart contract and analyses it through lexical and syntactic analysis. It then converts the smart contract sol into an abstract syntax tree (AST) in JSON format. The resulting AST is then transformed into the intermediate representation (IR) of ESBMC, which is used to generate the symbol table. The table is then converted into a GOTO program and delivered to the symbolic execution engine (SymEx) for processing to generate its static single assignment (SSA) form. This form is then used to generate verification conditions. Finally, create logistic equations to represent constraints (C) and properties (P) from the SSA form. When the SMT solver finds a solution that satisfies the equation, a vulnerability is considered to exist.

In 2022, eTainter [41] was published. It takes the bytecode of smart contracts as input and builds a control flow graph (CFG). By identifying the bytecode, it extracts the EVM instructions for reading user data and can identify gas-related vulnerable EVM instructions. CFG paths are then extracted and analyzed for contamination.

Sailfish [42] is a tool that identifies inconsistent state errors in smart contracts. It was released in 2022 and consists of two parts: the EXPLORE phase and the REFINE phase. During the EXPLORE phase, the contract is converted into a storage dependency graph (SDG), which is then queried to match vulnerable subgraphs. During the REFINE phase, the tool will utilise symbolic execution to analyse contracts that match vulnerable subgraphs and use pre-stored variable constraints as a prerequisite for symbolic execution.

RA [43] was released in 2021 and is used to detect reentrant vulnerabilities. It combines symbolic execution and SMT solvers to simulate and verify reentrant vulnerabilities.

Eth2Vec [44] is a tool that automatically extracts features for each contract using machine learning methods and neural networks for natural language processing. The tool will be released in 2022. Its purpose is to detect vulnerabilities in contracts by comparing the similarity of the extracted features to previously learned code.

EtherSolve [45] released in 2021 proposes a symbolic execution algorithm based on the Ethereum operand stack. The bytecode is converted to CFG by parsing the jump operation in the contract, and reentrancy vulnerabilities are detected through CFG.

Gasgauge [46] was published in 2021. The tool divides its detection process into three stages: detection, identification, and correction. In the detection phase, static analysis is used to identify all loops in the contract. Then, fuzz testing methods are used to generate inputs that exhaust contract gas. Finally, vulnerabilities are reported to users through the results of the running verification and static analysis phases.

Echidna [47] is a smart contract vulnerability detection fuzz testing tool published in 2020. It provides a complete Ethereum smart contract fuzz testing framework, which can analyze and simulate the execution of smart contract source code and generate compliance with contract call specifications. The tool uses random transaction data to fuzz test the contract and introduces coverage information to detect the execution efficiency of fuzz testing.

WANA [48] was published in 2020. It uses the Wasm symbolic execution engine to parse, load, and initialize the Wasm bytecode. It checks unsatisfied paths by calling the Z3 constraint solver and collects execution information for vulnerability analysis. WANA performs vulnerability analysis based on the collected execution information during symbolic execution.

SmartEmbed [49], published in 2020, is an effective tool for checking bugs in smart contracts. It allows for easy development of error checking rules and addition of new errors. The similarity checker is the core component of SmartEmbed, which takes in the error embedding matrix, code embedding matrix, and embedding vector as input, and outputs error reports.

VERISMART [50] is a fully automatic tool based on static and dynamic program analysis technology, implemented in OCaml and published in 2020. This tool analyzes the basic path of a smart contract. Verify the generation conditions and detect vulnerabilities by collecting unknown paths.

sFuzz [51] was published in 2020 and aims to improve test case coverage by adopting a feedback adaptive fuzzing strategy, while learning from the traditional fuzz testing tool AFL. The tool is composed of three main modules: runner, libfuzzer, and liboracles. The runner module creates a private test network for deploying test contracts and executing transactions. The libfuzzer component generates test cases, while the liboracles component detects the execution of test cases and checks for vulnerabilities.

EthBMC [52] is a dynamic symbolic execution-based bounded model checker published in 2020. It can automatically generate specific inputs to simplify further analysis of EVM bytecode. EthBMC addresses identified issues by more precisely reasoning about the internals of the EVM. The contract is encoded with constraints and then solved using an SMT solver. EthBMC is effective in solving the parity error vulnerability.

The EVMFuzzer [53] is a fuzz testing tool designed to detect abnormalities in the Ethereum Virtual Machine (EVM). The tool was published in 2019. It uses smart contracts as input, generates random mutation data, and sends it to the target EVM and baseline EVM. The output results and status changes of each EVM are then compared. Any differences or abnormalities found are reported as suspicious cases.

SolidityCheck [54] was published in 2019. This is a static open-source tool written in C++ that uses regular expressions and program instrumentation to address vulnerabilities in smart contracts [55]. The tool's detection process is divided into four main stages: 1. Formatting the smart contract code; 2. Filtering keywords to extract problematic statements; 3. Prevention and detection; 4. Generating detection reports and prevention contracts. SolidityCheck primarily identifies reentrancy vulnerabilities and integer overflows with exceptional accuracy.

Published in 2019, Vultron [56] is a dynamic smart contract fuzz testing framework developed using JavaScript. It can detect irregular transactions caused by various types of adversarial exploits.

VeriSolid [57] is an open-source web-based verification framework developed in 2019. It is built on webGME and FSolidM and allows for collaborative development of Ethereum contracts with built-in version control.

SIF [58] is a static vulnerability analysis and detection tool that is open source. It was developed in C++ language and was invented in 2019. This tool is a comprehensive tool for analyzing, querying, detecting, and generating code for Solidity contracts. Additionally, the tool incorporates a module called Assertion Analyzer, which identifies Integer Overflow vulnerabilities based on the analysis.

FEther [59] is an open-source static smart contract analysis and detection tool developed using Coq. Published in 2019, this tool is designed to provide objective evaluations of contract code. It takes Solidity as input and analyzes and detects contract code through symbolic execution and logical constraint solving.

Published in 2018, Vandal [60] is a static and open-source tool for detecting smart contract vulnerabilities. The tool decompiles the bytecode of the Ethereum Virtual Machine (EVM) into an intermediate representation (IR) and builds the control flow graph (CFG) of the program. The Vandal tool converts the IR and CFG into logical relationships, which are then stored in a knowledge base. The tool uses the Soufflé language to write security analysis specifications, which are inputted into the Soufflé engine along with the knowledge base for query and reasoning. Finally, the tool generates security analysis specifications based on the output results of the Soufflé engine. The report is then produced.

Securify [61] is a static and open-source smart contract vulnerability detection tool that was published in 2018. It uses symbolic execution to analyze smart contracts and extract data flow, control flow, function calls, and other information from the code. The tool then checks for violations of the preset pattern.

EthIR [62] is a static and open-source smart contract vulnerability detection tool published in 2018. Its main contribution is converting Ethereum bytecode into a rule-based representation (RBR), which allows for easy application of advanced analysis methods to infer the nature of EVM code. To begin, generate a control flow graph (CFG) from Ethereum bytecode using OYENTE. Next, extract the basic blocks and jump targets from the CFG using a decompiler. Then, extract the rules and operand stacks from the basic blocks using an interpreter. Finally, convert the rules and operand stacks to RBR using a reconstructor.

MAIAN [10] was published in 2018. It uses symbolic execution and Hybrid Depth First Search (Hybrid Depth First Search) technologies to analyze possible state transitions and abnormal behaviors that may occur during multiple calls of smart contracts. The process mainly consists of four steps: preprocessing, symbolic execution, recording paths and conditions that may trigger the vulnerability using the Z3 theorem prover, and hybrid depth-first search. Use a heuristic algorithm to sort the paths obtained by symbolic execution, and select the path most likely to trigger the vulnerability for further analysis; Verification, use specific input and transaction sequences to verify the results obtained by symbolic execution and hybrid depth-first search vulnerability path and report the discovered vulnerabilities.

Published in 2017, Porosity [63] is a static detection tool that converts EVM bytecode into an intermediate representation (IR). It then analyses basic blocks and boundaries based on jump instructions and function calls. Porosity connects basic blocks into CFGs and identifies loops, conditional branches, and function entries. Finally, it converts the CFG into a Solidity syntax contract and outputs it to standard output or a file.

FSolidM [64] is a tool for detecting static smart contract vulnerabilities that was published in 2017. It is based on the Finite State Machine (FSM) model, which abstracts smart contract behavior into a set of states and transitions. The tool includes a graphical editor that enables developers to create and modify FSMs using drag-and-drop. Additionally, FSolidM provides a code generator that can convert FSMs into Solidity language smart contracts and deploy them to the Ethereum network.

KEVM [65] was created in 2016 as a tool for detecting static smart contract vulnerabilities. It uses the K framework to define the syntax and semantics of the EVM's bytecode stack language, allowing it to execute and simulate the behavior of the EVM. KEVM can simulate all instructions and abnormal behaviors of the EVM, and its correctness and performance can be verified through the official Ethereum test suite.

SmartCheck [30], a static open-source smart contract vulnerability detection tool developed in Java, was published in 2018. SmartCheck conducts lexical and semantic analysis on Solidity source code and utilizes ANTLR [66] and a custom Solidity language to generate an XML parse tree as an intermediate representation. Users can detect vulnerability patterns by employing XPath queries on the intermediate representation. SmartCheck offers significant improvements compared to existing alternatives, as it has been extensively tested on real-world contracts and successfully identified code issues in the majority of them.

Slither [31], published in 2019, is an open-source static analysis framework for smart contract vulnerabilities. This tool serves four main purposes: (1) automatic detection of vulnerabilities, (2) automatic detection of code optimization opportunities, (3) enhancing user understanding of contracts, and (4) assisting with code reviews. Its working principle involves transforming Solidity smart contracts into an intermediate representation called SlithIR. SlithIR utilizes Static Single Assignment (SSA) form and a simplified instruction set to streamline the implementation of analysis while retaining semantic information. Slither allows the application of common program analysis techniques such as data flow and taint tracking to detect vulnerabilities. The tool is capable of detecting Reentrancy, Integer Overflow, Unchecked Low Level Calls, Tx.origin, Time Manipulation, Delegatecall, and Access Control.

Mythril [67], published in 2017, is a static and open-source smart contract vulnerability detection tool. It utilizes symbolic execution techniques to simulate the execution process of a program. Instead of using concrete input values, it uses symbolic variables to explore all possible execution paths of the program. In addition, Mythril incorporates techniques such as taint analysis and control flow graphs to enhance the efficiency and accuracy of the analysis.

Oyente [68], a tool that emerged in 2016, utilizes symbolic execution to analyze smart contract bytecode. It consists of four key components: CFG Builder, Explorer, Core Analysis, and Validator. Oyente operates at the bytecode level and dynamically explores the program's control flow graph during symbolic execution. It detects contract vulnerabilities by considering path constraints, variable origins, and other relevant information.

Solmet [69], published in 2018, is built upon the existing ANTLR4 grammar to parse Solidity language. Although it does not directly detect vulnerabilities, Solmet analyzes Solidity smart contracts and calculates relevant metrics.

Solhint [70], introduced in 2017, is a static analysis tool for smart contract vulnerability detection. It begins by converting the source code of a smart contract into an abstract syntax tree (AST) using a parser. Then, it traverses the AST using a visitor while loading user-specified configurations, which include enabled or disabled rules and plugins. Solhint checks each AST node based on the rules in the configuration file and collects detected issues. Finally, it formats the problems in different output formats, such as console or file.

ContractFuzzer [71], published in 2018, employs fuzz testing techniques to identify security vulnerabilities in Ethereum smart contracts. It operates in four steps: firstly, it generates fuzzy inputs based on the smart contract's ABI specification; then, it uses predefined test cases to detect common vulnerability types; next, it utilizes an EVM instrumenter to record the runtime behavior of the smart contract; finally, it employs a log analyzer to analyze the recorded logs and report any discovered vulnerabilities.

Osiris [72], which was published in 2018, is a static and open-source framework for smart contracts vulnerability detection and analysis. It primarily utilizes a combination of symbolic execution and taint analysis to detect vulnerabilities. The tool converts the bytecode of smart contracts into an intermediate representation (IR) and performs static analysis to identify instructions and variables that may lead to integer-related vulnerabilities. Then, Osiris utilizes taint analysis to mark inputs, outputs, and state variables related to integer-related vulnerabilities, creating a taint graph that represents their data flow dependencies. Subsequently, the tool employs symbolic execution to traverse all paths in the taint graph and generate corresponding path constraints and objective functions. Finally, Osiris uses a constraint solver, such as Z3, to solve the path constraints and objective functions, resulting in a set of input values satisfying the conditions, i.e., exploits. If no satisfying input values are found, it indicates that the path is secure or unreachable.

HoneyBadger [73], published in 2019, consists of three components: symbolic analysis, cash flow analysis, and honeypot analysis. The symbolic analysis component constructs a control flow graph (CFG) and executes its different paths symbolically. The results of symbolic analysis are then propagated to the cash flow analysis and honeypot analysis components. The cash flow analysis section utilizes the results from symbolic analysis to detect whether a contract can receive and transfer funds. Lastly, the honeypot analysis component aims to detect different honeypot techniques studied in this article by combining heuristic and symbolic analysis results. All components employ the z3 SMT solver to check for constraint satisfaction.

3.3 Comparison of Open Source Tools

This section presents a summary of current open source tools. Table 3 presents specific information, such as the detection method, project address, update time, installation method, and ease of use. Ease of use is divided into five levels: 5★ indicates that it is extremely easy to install and does not require the installation of any environment; 4★ indicates that it is easy to install and only requires the installation of one environment dependency; 3★ indicates that it is easy to install but requires the installation of two environment dependency; 2★ indicates that requires the installation of three environment dependency; 1★ indicates that more than three environment dependencies.

Table 3: Solidity detection tool project address and environmental dependencies

Tool name	Detection method	Project address	Update time	Installation method	Installation difficulty
Vulpedia	Feature matching	https://github.com/ToolmanInside/vulpedia_demo Accessed: Mar. 19, 2024.	2022.4.26	Docker	4★
SpCon	Feature matching, taint analysis	https://github.com/Franklinliu/SpCon-Artifact Accessed: Mar. 19, 2024.	2024.3.13	Docker, Python3.8	4★
xFuzz	Fuzz testing	https://github.com/zhang-alt/xFuzz Accessed: Mar. 19, 2024.	2022.4.30	Docker, solc-select+ python+ sFuzz+ slither/surya	4★
ESBMC	Symbolic execution	https://ssvlab.github.io/esbmc/documentation.html Accessed: Mar. 19, 2024.	2022.7.26	solc, esbmc	3★
eTainter	Feature matching, taint analysis	https://github.com/DependableSystemsLab/eTainter Accessed: Mar. 19, 2024.	2024.3.19	Python+ Z3-solver+ pysha3	3★
Sailfish	Symbolic execution	https://github.com/ucsb-seclab/sailfish Accessed: Mar. 19, 2024.	2022.6.5	Docker, python	4★
RA	Symbolic execution	https://github.com/wanidon/RA	2021.9.22	Python3.7+Z3-solver+pysha3	3★

(Continued)

Table 3 (continued)

Tool name	Detection method	Project address	Update time	Installation method	Installation difficulty
		Accessed: Mar. 19, 2024.			
Eth2Vec	Feature matching	https://github.com/fseclab-osaka/eth2vec Accessed: Mar. 19, 2024.	2021.3.26	Kam1n0 server [74]+py-solc-x	1★
EtherSolve	Symbolic execution	https://github.com/SeUniVr/EtherSolve Accessed: Mar. 19, 2024.	2023.8.17	Java 11.0.8, Gradle [75]	3★
Gas gauge	Fuzz testing	https://github.com/gasgauge/gasgauge.github.io Accessed: Mar. 19, 2024.	2021.5.7	Slither+ Truffle+solc+ python+NodeJS	2★
Echidna	Fuzz testing	https://github.com/crytic/echidna Accessed: Mar. 19, 2024.	2024.3.19	Docker, Homebrew [76], Nix [77], Stack [78]	4★
WANA	Symbolic execution	https://github.com/gongbell/WANA Accessed: Mar. 19, 2024.	2021.3.28	solc+six+ func_timeout+ Z3-solver	3★
SmartEmbed	Feature matching	https://github.com/beyondacm/SmartEmbed Accessed: Mar. 19, 2024.	2023.3.13	Flask+ WTForms+ genism+SciPy+ python	3★
VERISMART	Feature matching, symbolic execution	https://github.com/kupl/VeriSmart-public Accessed: Mar. 19, 2024.	2023.1.17	OCaml+Z3 +solc	2★

(Continued)

Table 3 (continued)

Tool name	Detection method	Project address	Update time	Installation method	Installation difficulty
sFuzz	Fuzz testing	https://github.com/duytai/sFuzz Accessed: Mar. 19, 2024.	2022.3.24	Web	5★
EthBMC	Symbolic execution	https://github.com/RUB-SysSec/EthBMC Accessed: Mar. 19, 2024.	2022.12.31	Go-ethereum [50]+Rust [51]+Yices2 [52]	1★
EVMFuzzer	Fuzz testing	https://github.com/EVMFuzzer/EVMFuzzer Accessed: Mar. 19, 2024.	2019.3.19	Python+ ethereum+solc	4★
SolidityCheck	Feature matching	https://github.com/xf97/SolidityCheck Accessed: Mar. 19, 2024.	2021.6.28	Docker	4★
Vultron	Fuzz testing	https://github.com/ntu-SRSLab/vultron Accessed: Mar. 19, 2024.	2019.11.5	Node.js: 12.12.0 +Truffle: 5.0.42+Go-ethereum: 1.8	2★
VeriSolid	Symbolic execution	https://github.com/anmavrid/smart-contracts Accessed: Mar. 19, 2024.	2023.1.4	NodeJS (v4.x.x)+ MongoDB+ webgme+bower	1★
SIF	Feature matching	https://github.com/chao-peng/SIF Accessed: Mar. 19, 2024.	2020.6.27	solc	4★

(Continued)

Table 3 (continued)

Tool name	Detection method	Project address	Update time	Installation method	Installation difficulty
FEther	Feature matching	https://github.com/openethereum/fEther Accessed: Mar. 19, 2024.	2020.2.11	Web	5★
Vandal	Feature matching	https://github.com/usyd-blockchain/vandal Accessed: Mar. 19, 2024.	2020.7.29	Souffle+python	3★
Securify	Symbolic execution	https://github.com/eth-sri/securify Accessed: Mar. 19, 2024.	2020.1.24	Docker, solc+Java 8+ Soufflé	3★
EthIR	Feature matching	https://github.com/costa-group/ethIR Accessed: Mar. 19, 2024.	2024.3.18	solc+ethereum (last version tested 1.8.18)+Z3 (last version tested 4.5.0)+pip	3★
MAIAN	Symbolic execution	https://github.com/ivicanikolicsg/MAIAN Accessed: Mar. 19, 2024.	2021.10.4	python	4★
Porosity	Feature matching	https://github.com/msuiche/porosity Accessed: Mar. 19, 2024.	2019.1.10	solc	4★

(Continued)

Table 3 (continued)

Tool name	Detection method	Project address	Update time	Installation method	Installation difficulty
FSolidM	Symbolic execution	https://github.com/contractshark/fsolidm Accessed: Mar. 19, 2024.	2020.10.30	NodeJS (v4.x.x recommended)+ MongoDB+ webgme	1★
KEVM	Symbolic execution	https://github.com/runtimeverification/evm-semantics Accessed: Mar. 19, 2024.	2024.3.18	kup	3★
SmartCheck	Feature matching	https://github.com/smartdec/smartcheck Accessed: Mar. 19, 2024.	2023.5.25	NodeJS	4★
Slither	Feature matching	https://github.com/crytic/slither Accessed: Mar. 19, 2024.	2024.3.19	Docker, python	4★
Mythril	Symbolic execution, taint analysis	https://github.com/ConsenSys/mythril Accessed: Mar. 19, 2024.	2024.3.16	Docker, solc+software-properties-common+ python3	4★
Oyente	Symbolic execution	https://github.com/enzymefinance/oyente Accessed: Mar. 19, 2024.	2020.11.7	Docker, solc+Go-ethereum+Z3+ pip	4★
Solmet	Feature matching	https://github.com/chicxurug/SolMet-Solidity-parser Accessed: Mar. 19, 2024.	2023.8.8	Java	4★

(Continued)

Table 3 (continued)

Tool name	Detection method	Project address	Update time	Installation method	Installation difficulty
Solhint	Feature matching	https://github.com/protofire/solhint Accessed: Mar. 19, 2024.	2024.3.16	NodeJS	4★
ContractFuzzer	Fuzz testing	https://github.com/gongbell/ContractFuzzer Accessed: Mar. 19, 2024.	2020.3.16	Docker	4★
Osiris	Feature matching, taint analysis, symbolic execution	https://github.com/christoftorres/Osiris Accessed: Mar. 19, 2024.	2023.3.7	Docker, solc+Go-ethereum+Z3+python	4★
HoneyBadger	Symbolic execution	https://github.com/christoftorres/HoneyBadger Accessed: Mar. 19, 2024.	2023.3.7	Docker, Go-ethereum+Z3+python	4★

3.4 Evaluation Vulnerability Detection Coverage

Table 4 displays the vulnerabilities that can be detected by the tools mentioned in this paper. It is important to note that some tools only convert the code into an easy-to-analyze form without directly instrumenting the smart contract. Although this method of converting code may aid in further analysis and identification of potential vulnerabilities, it still necessitates manual review and verification. Therefore, the security of smart contracts still requires a combination of expertise and experience when using these tools.

Table 4: Detecting tool vulnerability coverage

Tool name	Reentrancy	Integer overflow	Unchecked level calls	low Tx.origin	Time manipulation	Delegatecall	Access control	TOD	Denial of service
Vulpedia	√	×	√	√	×	×	×	×	√
SpCon	×	×	×	√	×	√	√	×	×
xFuzz	√	×	×	√	×	√	×	×	×
ESBMC	×	√	×	√	×	×	×	×	×
eTainter	×	×	√	×	×	×	×	×	√
Sailfish	√	×	×	×	×	×	×	√	×

(Continued)

Table 4 (continued)

Tool name	Reentrancy	Integer overflow	Unchecked low level calls	Tx.origin	Time manipulation	Delegatecall	Access control	TOD	Denial of service
RA	√	×	×	×	×	×	×	×	×
Eth2Vec	√	√	×	×	√	×	√	×	√
EtherSolve	√	×	×	×	×	×	×	×	×
Gas gauge	×	×	×	×	×	×	×	×	√
Echidna	×	×	×	×	×	×	×	×	√
WANA	×	×	×	×	√	√	√	×	×
SmartEmbed	√	√	√	√	√	×	√	×	×
VERISMART	×	√	×	×	×	×	×	×	×
sFuzz	×	√	×	×	√	√	√	√	√
EthBMC	×	×	×	×	×	√	×	×	×
EVMFuzzer	×	×	√	×	×	×	×	×	×
SolidityCheck	√	×	×	×	×	×	×	×	×
Vultron	√	√	√	×	×	×	×	×	√
VeriSolid	√	×	√	×	×	×	×	×	√
SIF	×	√	×	×	×	×	×	×	×
FEther	×	×	×	×	×	×	×	×	√
Vandal	√	×	×	√	×	×	×	×	×
Securify	√	√	×	√	√	√	√	√	√
EthIR	×	×	×	×	×	×	×	×	√
MAIAN	×	×	×	×	×	×	√	×	×
Porosity	√	×	√	×	√	×	×	×	×
FSolidM	√	×	×	×	×	×	√	√	×
KEVM	×	×	×	×	×	×	×	×	√
SmartCheck	√	√	√	√	√	×	√	×	√
Slither	√	√	√	√	√	√	√	×	×
mythril	√	√	√	×	√	√	√	×	×
oyente	√	×	√	×	√	×	×	√	×
Solmet	×	×	×	×	×	×	×	×	×
Solhint	√	×	√	√	√	×	√	×	×
ContractFuzzer	√	×	√	×	√	√	√	×	×
Osiris	√	√	×	×	√	×	×	×	×
HoneyBadger	×	×	×	×	×	×	×	×	×

4 Vulnerability Detection Framework Based on Hybrid Cloud Platform

By researching the majority of existing vulnerability detection tools, we have discovered that they are relatively challenging to use. López Vivar et al. [79] proposed the ESAF framework for a unified analysis of smart contract vulnerabilities. This tool integrates several existing tools, allowing developers to leverage the strengths of different tools for smart contract vulnerability analysis. However, the tool itself relies on the user's local setup of Python, Docker, and MongoDB, which makes practical usage somewhat challenging. Ferreira et al. [80] introduced SmartBugs as a tool to assist developers in comparing their work with existing tools. This tool is Docker-configured, which enables users to utilize it through the command-line or web-UI. After execution, it generates reports based on the corresponding tool's returned results. However, practical users are required to possess certain knowledge and skills, and some of the tools within SmartBugs have longer execution times, making them unsuitable for running on personal computers.

Through literature research and practical experience, we have identified several challenges that arise when using Solidity-based code vulnerability detection tools in real-world applications:

Significant time spent on environment configuration and debugging. Different tools require different runtime environments, which result in unnecessary cost of time, thus prolongs project completion cycles;

Non-uniform input leads to strong independence. Current tools often exist as standalone entities, meaning that developers need to input contracts into multiple detection tools for testing manually. This adds complexity to the workflow and hinders the automation of testing processes;

Comprehension difficulty due to different output formats. The output formats and interpretation methods of existing tools vary greatly, necessitating manual analysis by individuals with specialized knowledge to obtain the final detection results.

These above issues increase the difficulty of smart contract development and maintenance. Therefore, to facilitate the usage for developers, we have designed a vulnerability detection framework. This framework provides a unified contract detection interface, simplifies the vulnerability analysis process and offers more intuitive results.

The framework design encompasses the following four points:

- It provides developers with a more convenient experience using unified contract detection interface. By reducing the learning curve and improving work efficiency. Developers no longer need to learn various cumbersome tool usage methods.
- It possesses high scalability for tools. This scalability ensures that our framework can continuously update alongside the development of vulnerability detection techniques.
- It offers a historical record query feature. Developers can conveniently review previous vulnerability analysis results and relevant information for comparison and analysis. Developers can better track the progress of vulnerability fixes and promptly engage in vulnerability prevention work.
- It exhibits compatibility with multiple platforms, facilitating cross-platform usage for users. Whether operating on Windows, Mac, or Linux systems, developers can effortlessly employ our framework for vulnerability analysis, enhancing work flexibility and convenience.

4.1 Environment and Related Technologies

In this section, we will introduce the operating environment of the framework. For the overall framework operation, we utilize Python as the primary building tool to implement the user interface and interaction logic. Python is a versatile programming language with powerful cross-platform capabilities, which allowing it to run on different operating systems.

In the vulnerability detection section, we employ VMware to run the detection tools. Specifically, we choose VMware as the platform to execute the detection tools. VMware is a virtualization software that enables the creation of multiple virtual machines on a single host, each running different operating systems. This allows for the simulation of various environments to accommodate the use of different detection tools.

Regarding the storage of user information and running results, we utilize MySQL as the database system. MySQL is an open-source relational database management system that is suitable for small to medium-sized applications. It is user-friendly, providing sufficient performance and reliability to meet most typical data storage and querying needs.

4.2 Module Function

The design philosophy of our framework is based on converting functional requirements analysis into a modular code structure. This design allows us to achieve better modularity and maintainability. Each functionality is encapsulated as an independent module and interacts with other modules through well-defined interfaces. This design enables us to conveniently expand and modify functionalities while reducing code coupling.

Our framework consists of four parts: the display module, the preprocessing module, the database module, and the virtualization validation module. The web interaction module serves as the user interface of the system, providing users with convenient features such as uploading contract files, viewing detection results, and accessing historical information. The preprocessing module performs initial processing on the uploaded contracts to improve the efficiency and accuracy of subsequent detection. The database module is an essential component of the smart contract vulnerability detection system. It primarily handles the storage and management of contract detection-related data, including preprocessed contracts, detection results, and the order of contract detection. The preprocessed contract results are stored in the database in the form of file addresses, facilitating subsequent testing and analysis. The virtualization testing module is the core part of the smart contract vulnerability detection system. It is responsible for comprehensive testing of smart contracts and feeding the results back to the database module for further analysis and processing.

Through the combination of these four modules, our framework is capable of achieving comprehensive detection and analysis of smart contracts. Moreover, this modular design allows for convenient expansion and modification of functionalities, enhancing the system's maintainability. Additionally, the interaction between modules through well-defined interfaces reduces code coupling. This design makes our framework more flexible and user-friendly.

4.3 Module Implementation

In this section, we will briefly describe the implementation details of the module based on the schematic diagram.

The operational logic of these interconnected modules is illustrated in Fig. 1. The system comprises four key modules: the Display Module, the Preprocessing Module, the Database Module, and the Validation Module. The Display Module functions as the user interface, allowing users to input contract codes. The Preprocessing Module processes these input codes, preparing them for use by subsequent modules. The Database Module acts as an intermediary for storage and transmission, storing the preprocessed codes and transferring the data to the Validation Module in the order received. The Validation Module verifies the received data and sends the validation results back to the Database Module. Finally, the Database Module presents these validation outcomes to the user, enabling them to review and understand the system's evaluations.

By employing such a modular operational logic, users can conveniently input contract codes and obtain corresponding validation results. This modular design enables each functional module of the system to work independently, facilitating ease of maintenance and expansion.

4.3.1 Display Module

The display module serves as the user interface of the system, providing a platform for users to interact with it. It encompasses functionalities such as uploading contract files, viewing detection results, and accessing historical information. Fig. 2 shows the interface and key features of the display module.

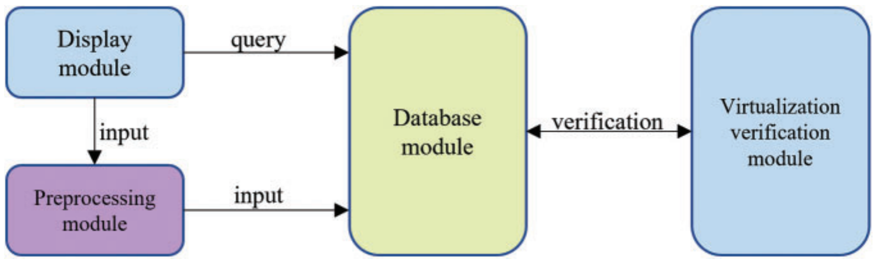


Figure 1: Overall architecture of the framework

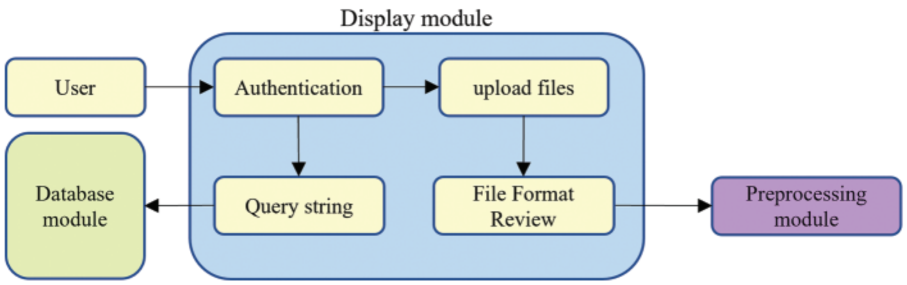


Figure 2: Schematic diagram of the display module

In terms of user authentication, this framework provides functions such as registration, modification, and login to facilitate user identity management.

Throughout the entire interaction process, users can conveniently upload their contract files through a concise and intuitive interface. The process of uploading contract files is extremely simple, with just a click on the upload button and selecting the local file.

Simultaneously, the display module will perform a format check on the contract file uploaded by the user to ensure that it is indeed a valid contract file. For example, Solidity contract files typically have a .sol extension. This module will examine the file to ensure that it meets the required format. Once the contract file is uploaded, the web interaction module will pass it on to the system's preprocessing module for initial processing.

4.3.2 Preprocessing Module

As shown in Fig. 3, the preprocessing module performs initial processing on the contracts uploaded by users to improve the efficiency and accuracy of subsequent detection. This module accomplishes two main tasks: removing comments from the source code to reduce code volume, and performing symbol substitution by replacing symbols in the code (such as variable names, function names, etc.) with fixed identifiers for further compilation and detection.

In the preprocessing module, the first step is to remove comments from the contracts. Although comments contribute to code readability, they are not necessary for compilation and detection. Therefore, removing comments can reduce code volume, save time and resources for subsequent processing.

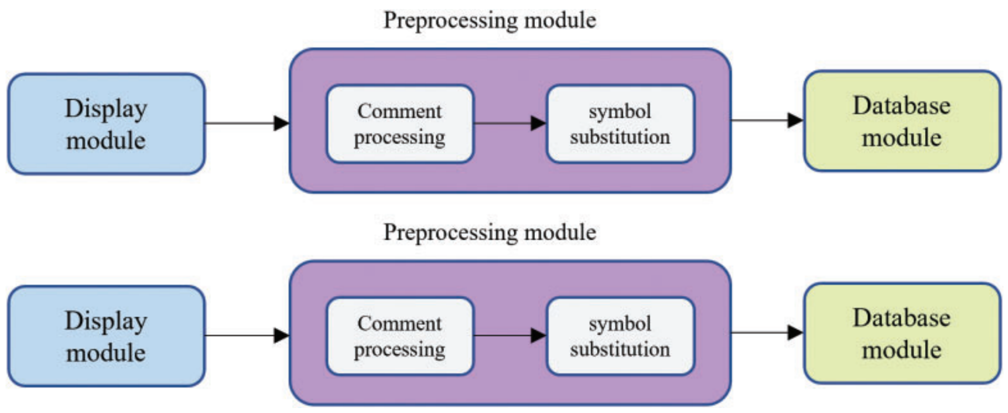


Figure 3: Schematic diagram of the preprocessing module

To facilitate further compilation and detection, the preprocessing module also performs additional operations such as deleting unnecessary characters like spaces and newline symbols, ensuring code compliance and ease of subsequent detection.

4.3.3 Database Module

As shown in Fig. 4, the entire database module is a vital component of the Smart Contract Vulnerability Detection Framework. It is primarily responsible for storing and managing data related to contract detection, including preprocessed contracts, detection results, and contract detection sequences. The results of the preprocessed module are stored in the database as file addresses, facilitating subsequent testing and analysis.

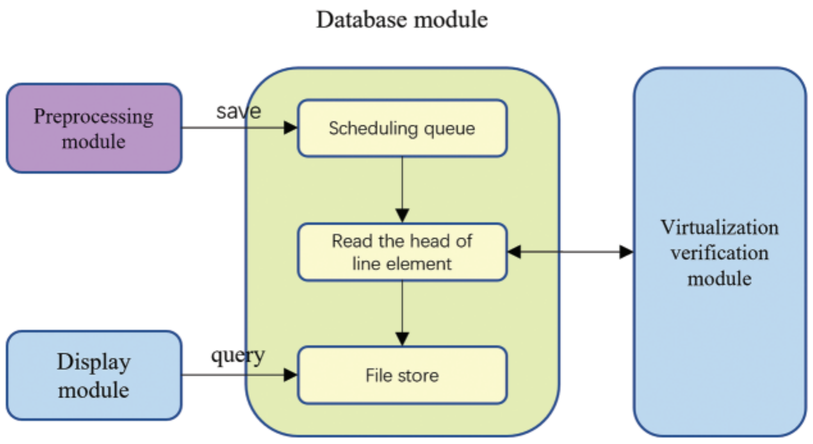


Figure 4: Schematic diagram of the database module

The relevant tables in the database contain attributes such as contract ID, user ID, detection results, and upload time. These attributes serve to uniquely identify contracts, differentiate between users, record the storage location of contracts and reports, and track the historical records and status of contracts. To ensure user privacy and confidentiality, appropriate permission controls have been implemented on the database, ensuring that each user can only view the results and reports of their own detected contracts.

The database module also includes a scheduling queue, which stores the IDs corresponding to contracts in the database. This queue is used to receive contracts processed by the preprocessing module. It stores the contract IDs passed from the preprocessing module and sends the contracts to the virtualized testing module. Once the virtualized testing module returns the results, they are stored in the database.

4.3.4 Virtualization Verification Module

As shown in Fig. 5, we have the Virtualization Testing Module, which serves as the core component of the Vulnerability Detection Framework Based on Hybrid Cloud Platform. Its main responsibility is to comprehensively test smart contracts and provide feedback to the Database Module for subsequent queries. These tools may require different environmental conditions during runtime or even face conflicting situations. To address this issue, we have employed virtualization technology, which provides each tool with an independent operating environment that meets its specific runtime requirements, thereby avoiding conflicts between the tools.

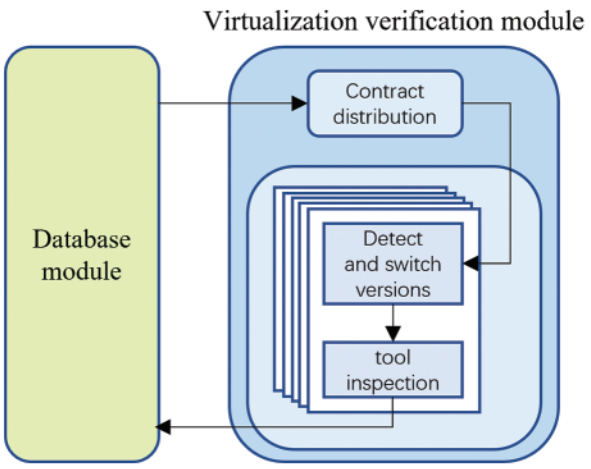

Virtualization verification module
Database module
Contract distribution
Detect and switch versions
tool inspection

Figure 5: Schematic diagram of virtualization verification module

Furthermore, the Virtualization Testing Module also takes into account that different versions of compilers may generate varying bytecode or even fail to compile, which could impact the execution results of the contracts. Therefore, we have incorporated an automatic compiler version switching module. Based on the contract's version information, the system will automatically switch to the corresponding compiler version, ensuring the fluency and reliability of the testing process. Once the virtualization testing is completed, the testing results will be transmitted back to the Database Module for further queries.

4.4 Network Design Based on Hybrid Cloud Platform

To ensure the integrity of the study and ensure the scalability of the detection framework, we further propose the corresponding network structure and hardware planning. The purpose of these supplementary works is to address the limitations of our method in specific tasks and provide more efficient and accurate solutions.

As shown in Fig. 6, the web server provides services through a unique domain name. The user's access request first reaches the Display and Preprocessing Module Server, where the router evenly distributes the requests to the servers and performs authentication services. At the same time, this server can preprocess the contracts after user submission. This architecture ensures that the system maintains good performance and availability even under high concurrent requests.

The Database Server is responsible for receiving smart contracts and storing data locally to ensure the security of user information. Additionally, this server can transmit the contracts to the virtualized detection module in the public cloud via the VPN Server through a dedicated link. The public cloud offers almost unlimited scalability. When demand increases, it is easy to add more virtual machines. Similarly, when demand decreases, it is convenient to reduce the number of virtual machines. This flexibility allows for dynamic adjustment of resources based on demand without the need for expensive hardware investments. Furthermore, deploying virtual machines in the public cloud usually takes only a few minutes, while deploying new physical servers locally may take several days or even weeks. This enables the framework to quickly scale with new tools.

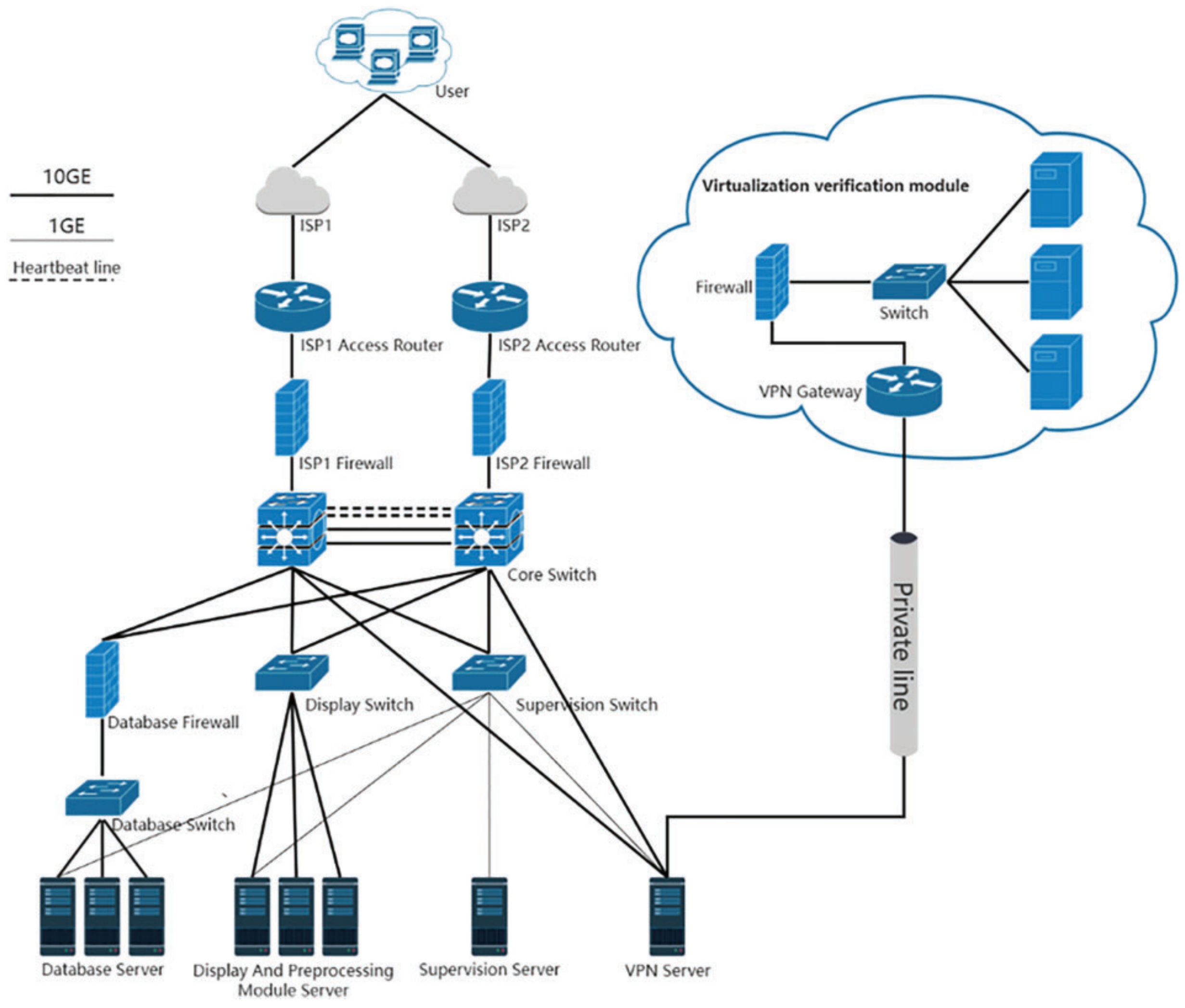


Figure 6: Network design schematic of the vulnerability detection framework based on a hybrid cloud platform

Additionally, the Supervision Server allows for obtaining information from each server, including performance logs and detection logs, for analysis.

4.5 *Framework Functional Tests*

a. As shown in Figs. 7 and 8, the framework offers user registration and login functionality, enabling users to create accounts and access the framework by logging in.

b. As shown in Figs. 9 and 10, the framework possesses the capability of unified detection and tool-specific detection to meet user requirements. Users can perform unified detection operations through the framework or choose to use different tools for detection. This flexibility allows users to select the most suitable detection method based on their own needs.

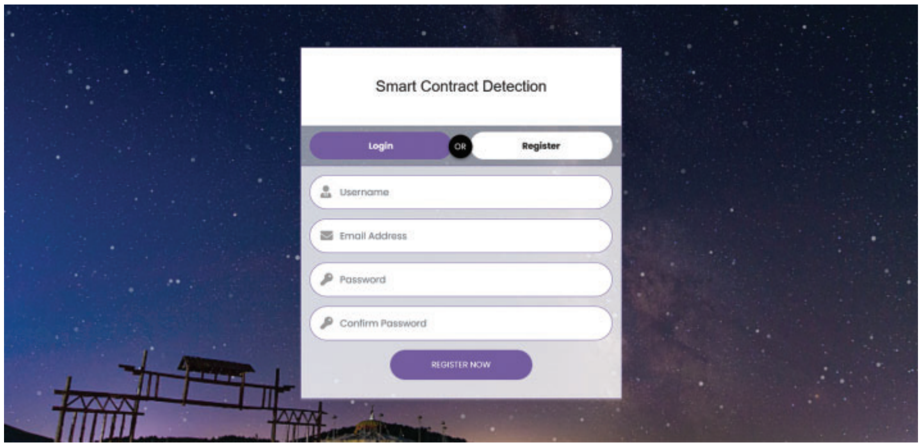


Figure 7: Framework registration function

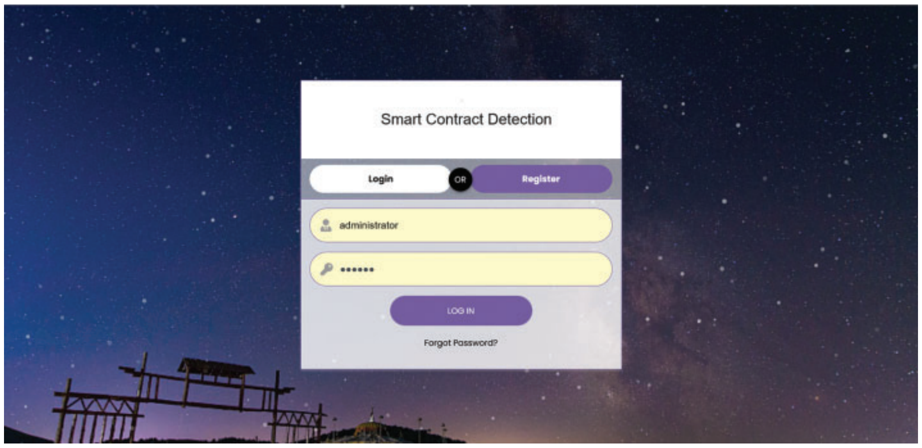
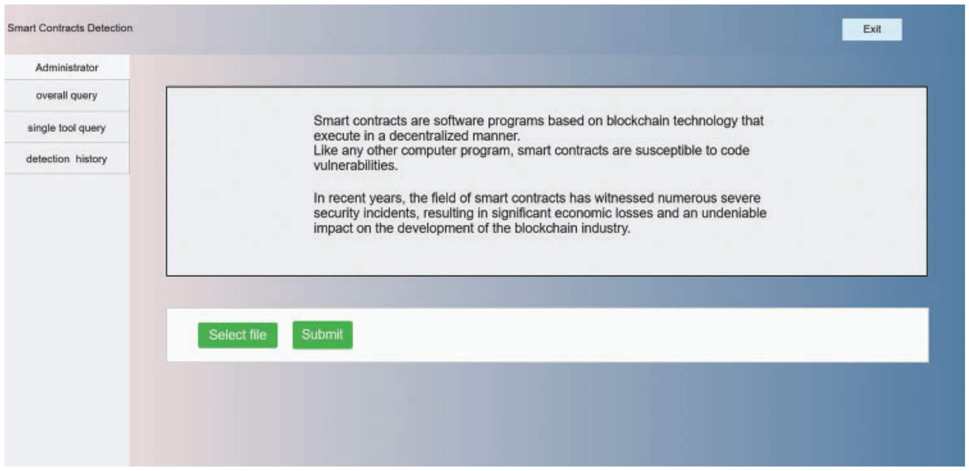


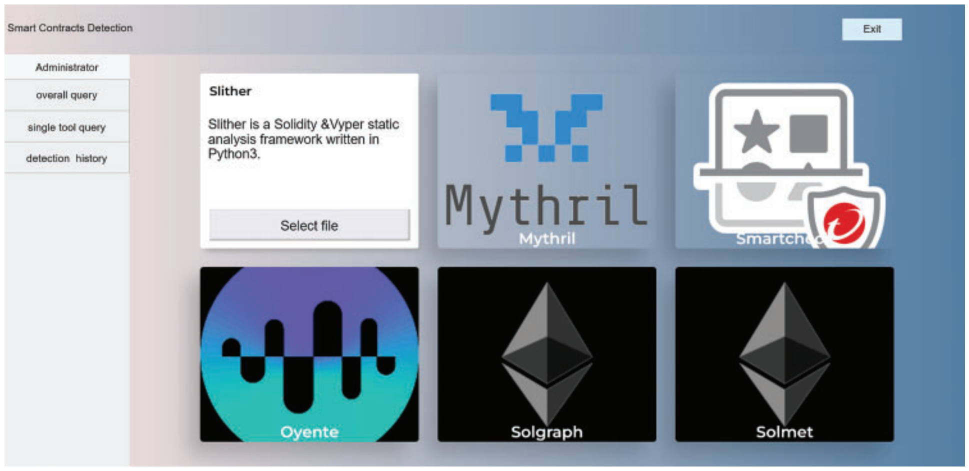
Figure 8: Framework login function



Smart contracts are software programs based on blockchain technology that execute in a decentralized manner.
Like any other computer program, smart contracts are susceptible to code vulnerabilities.

In recent years, the field of smart contracts has witnessed numerous severe security incidents, resulting in significant economic losses and an undeniable impact on the development of the blockchain industry.

Figure 9: Overall query



Slither is a Solidity &Vyper static analysis framework written in Python3.

Figure 10: Single tool query

c. As shown in Fig. 11, the framework offers a functionality to query the history records, allowing users to conveniently retrieve their previous operation records. Users can trace their historical queries on the framework through this feature, facilitating reference, review, or analysis purposes.

d. As shown in Fig. 12, the framework utilizes backend virtualization technology to establish a detection server, which provides a detection interface to the frontend. Users can interact with the backend virtualized detection server through the frontend page, submitting data for detection and retrieving the corresponding detection results.

5 Performance Experimental Analysis of Vulnerability Detection Tools

To assess the stability of the framework, we conducted over 1700 h of testing. Simultaneously, we aimed to investigate the hardware requirements of these tools by addressing the following three research questions:

RQ1: How does the execution speed of each tool compare?

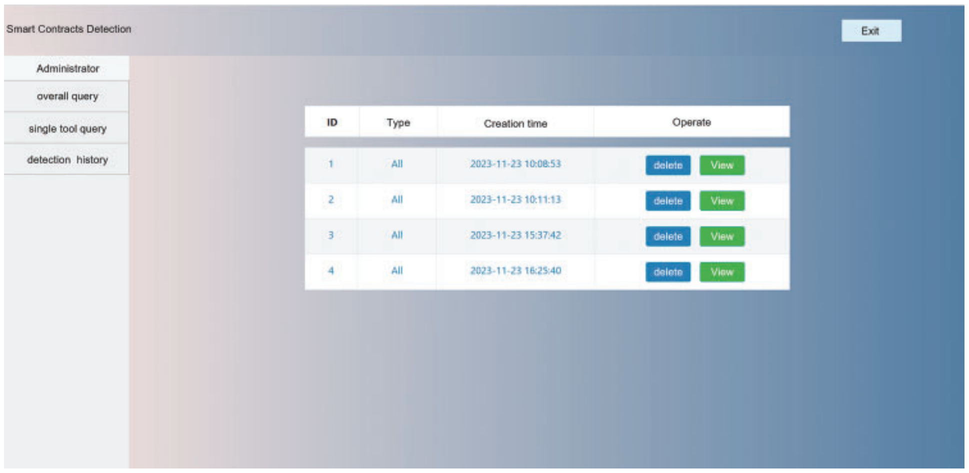
To answer this question, we tested and compared different types of contracts, and calculated the running time of each tool.

RQ2: Do different tools optimize for multicore processing, and does using more cores result in faster execution speed?

To answer this question, we tested the running speed of various tools under different core counts.

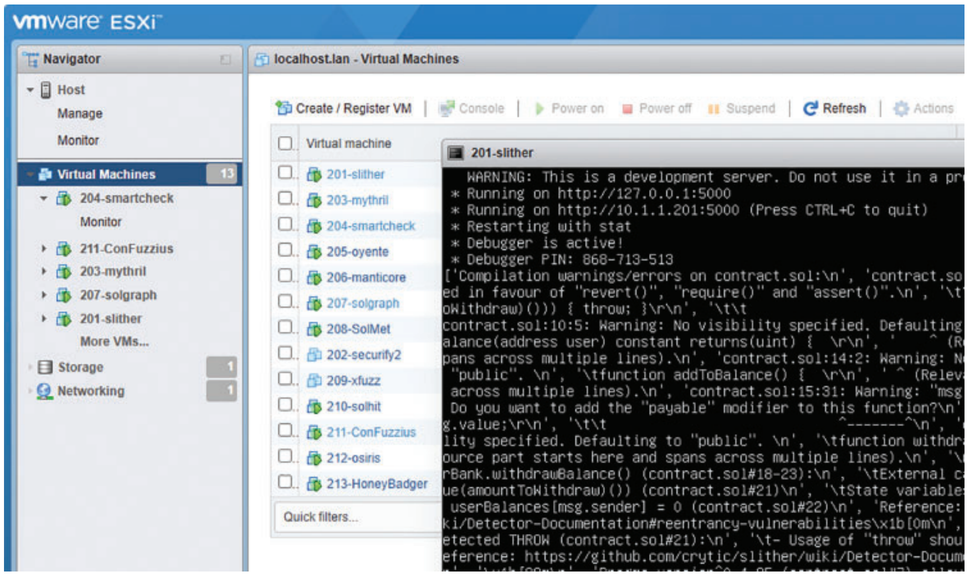
RQ3: How does each tool consume memory during the execution process?

To answer this question, we tested the memory usage of different tools during the testing of hundreds of contracts, and compared their memory requirements under different core counts.



ID	Type	Creation time	Operate
1	All	2023-11-23 10:08:53	delete View
2	All	2023-11-23 10:11:13	delete View
3	All	2023-11-23 15:37:42	delete View
4	All	2023-11-23 16:25:40	delete View

Figure 11: Detection history query



```
   WARNING: This is a development server. Do not use it in a pr
 * Running on http://127.0.0.1:5000
 * Running on http://10.1.1.201:5000 (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 868-713-513
['Compilation warnings/errors on contract.sol:\n', 'contract.so
ed in favour of "revert()", "require()" and "assert()".\n', '\t
oWithdraw)())) { throw; }\r\n', '\t\t
contract.sol:10:5: Warning: No visibility specified. Defaulting
alance(address user) constant returns(uint) {  \r\n', '     ^ (R
pans across multiple lines).\n', 'contract.sol:14:2: Warning: N
 "public". \n', '\tfunction addToBalance() {  \r\n', ' ^ (Relev
 across multiple lines).\n', 'contract.sol:15:31: Warning: "msg
 Do you want to add the "payable" modifier to this function?\n'
g.value;\r\n', '\t\t                               ^-------^\n', '
lity specified. Defaulting to "public". \n', '\tfunction withdr
ource part starts here and spans across multiple lines).\n', '\
rBank.withdrawBalance() (contract.sol#18-23):\n', '\tExternal c
ue(amountToWithdraw)()) (contract.sol#21)\n', '\tState variable
 userBalances[msg.sender] = 0 (contract.sol#22)\n', 'Reference:
ki/Detector-Documentation#reentrancy-vulnerabilities\x1b[0m\n',
etected THROW (contract.sol#21):\n', '\t- Usage of "throw" shou
eference: https://github.com/crytic/slither/wiki/Detector-Docum
```

Figure 12: Virtualized server interface

Section 5.1 introduces our datasets and the hardware environment used. In Section 5.2, we conducted tests for different core counts and compared the differences in execution speed. In Section 5.3, we performed memory tests to showcase the memory requirements of the tools during different detection processes.

5.1 Dataset

In recent years, the Ethereum community has developed numerous tools for analyzing vulnerable smart contracts. However, there is a lack of standardized datasets available. To gather sufficient experimental data, we utilized the dataset called SB Curated, manually constructed by Durieux et al. [81]. This dataset consists of 112 smart contracts, categorized into the five most common vulnerabilities: access control, denial of service, reentrancy, time manipulation, and unchecked low-level calls.

5.2 Implement Environment and Related Settings

We conducted the tests on a computer equipped with an Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50 GHz (48 CPUs) and 128 GB of memory. We chose ESXI 7.0 as the host system and used Ubuntu 20.04 as the virtual machine system.

For the batch testing script files, we used Python 3.8.10 as the programming language. Additionally, unless otherwise specified by the tools, the environment consisted of Docker version 20.10.16, npm version 6.14.4, and Java version 11.0.20.1.

We selected nine tools with different features and functionalities to cover various contract testing requirements. To ensure the accuracy of the test results and eliminate any interference from unexpected situations, we performed five tests on each contract and excluded the highest and lowest values to eliminate possible outliers. In total, we tested over 25,000 contracts. The entire testing process was time-consuming, totaling 1796.47 h. Among them, the main runtime was occupied by the Mythril tool for approximately 1700 h. To meet the running requirements of Mythril, we employed multiple servers and virtual machines for parallel execution.

In the experimental section, we conducted subgroup analysis to divide the study sample into different subgroups and compared the differences among them to draw conclusions.

5.3 Tool Runtime Testing

To evaluate the operational efficiency of the tool, we conducted tests on the tool's runtime. We standardized the configuration to 8 threads (4 cores, 2 threads) to simulate common setups. During the testing process, we recorded the runtime of each tool and performed a statistical analysis of the results. By comparing the runtime of different tools, we were able to assess their efficiency in handling contracts.

As shown in Fig. 13, differences in runtime between various smart contract detection tools are evident. Notably, Solmet is the fastest tool, completing the detection in just 28.53393111 s. This tool primarily focuses on extracting relevant metrics from contracts, contributing to its faster processing speed.

Next in line are Slither and Solhint, with runtimes of 60.2818644 and 77.7760623 s, respectively. Both of these tools employ feature matching techniques, which involve analyzing the relationships between feature codes, thereby requiring less time. In contrast, ContractFuzzer and Honeybadger have relatively longer runtimes, clocking in at 1134.240254 and 3048.585183 s, respectively. The slowest tool is Mythril, which takes 254086.36 s. We can observe that tools utilizing symbolic execution and fuzz testing for detection operate at a slower pace compared to those employing feature matching. This can be attributed to Slither and Solhint directly analyzing Solidity code.

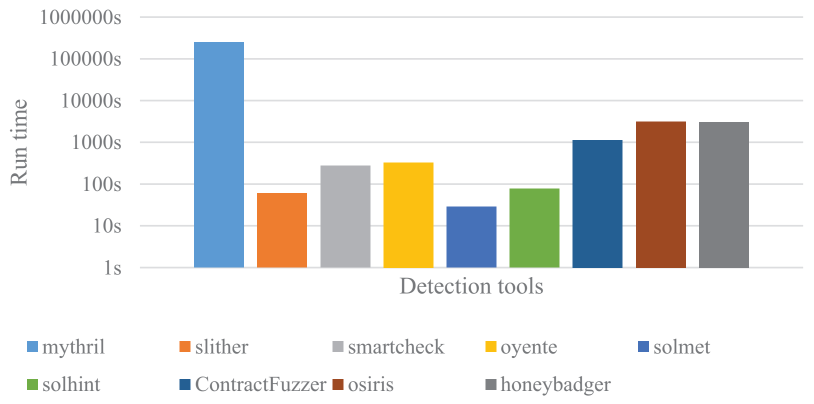


Figure 13: Speed comparison for detection tools

5.4 Multi-Core Optimization Testing

To further evaluate the operational efficiency of the smart contract detection tool and enhance its performance in real-world applications, we conducted multi-core optimization testing. These tests measured the recommended number of cores utilized by different tools and provided crucial indicators for optimizing resource utilization in subsequent frameworks.

During the testing process, we modified the configuration settings to various core counts and ran the tools for performance testing, while recording the execution time. We adjusted the test configuration to 1 thread, 2 threads, 4 threads, 8 threads, and 16 threads, respectively, and noted the execution time for each tool under each configuration. Subsequently, we compared the execution times to assess the impact of multi-core optimization on tool performance.

Fig. 14 shows that, in most cases, the number of threads does not affect the running speed of the tools. This is because the execution process of these tools primarily depends on other factors rather than CPU processing power, such as the speed of input data and the bottleneck of memory reading. Therefore, the performance of some tools remains relatively stable, regardless of whether they are executed in a single-threaded or multi-threaded environment.

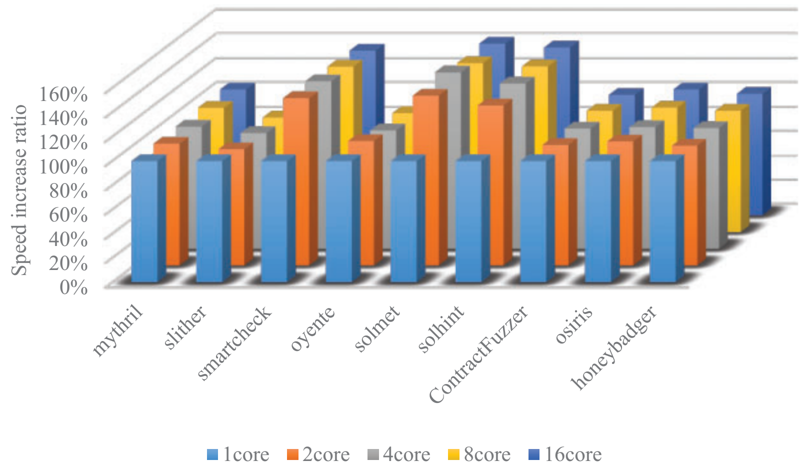


Figure 14: Test results for multi-core running speed of the detection tool

When studying the tools SmartCheck, Solmet, and Solhint, which utilize feature matching, we discovered their sensitivity to the number of threads through Fig. 15. Under configurations with a high number of threads, these tools exhibited only a slight reduction in running time, indicating marginal effects. However, as the number of threads decreased, the performance of these tools noticeably declined. Particularly, the transition from 2 threads to a single thread resulted in a significant increase in running time, suggesting a high level of support for multi-threaded parallel computing by these tools.

Furthermore, during our multi-threading optimization testing, we also observed some anomalies. Sometimes, increasing the number of threads did not necessarily result in a significant reduction in run time. As in Fig. 16, the slither tool could even lead to a slight increase. This may be attributed to competition among threads in a multi-core system, causing an uneven distribution of resources. In particular, the slither tool demonstrated noticeable advantages in a single-threaded environment. This probably because it can more efficiently utilize the computational resources of a single core. However, in a multi-threaded environment, its performance decreases due to the presence of resource competition.

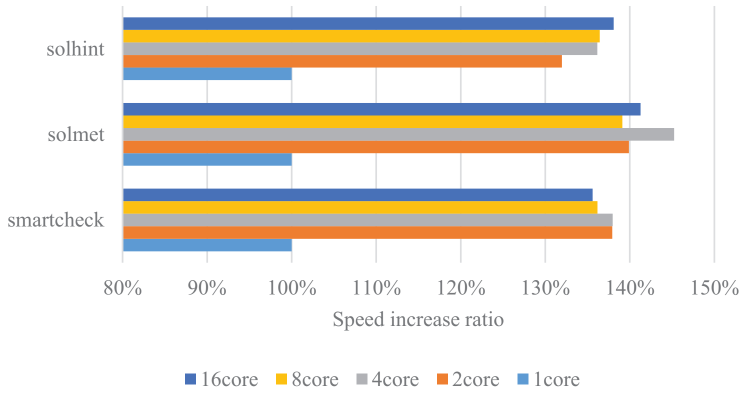


Figure 15: Test results for speed improvement of detection tools with multiple cores

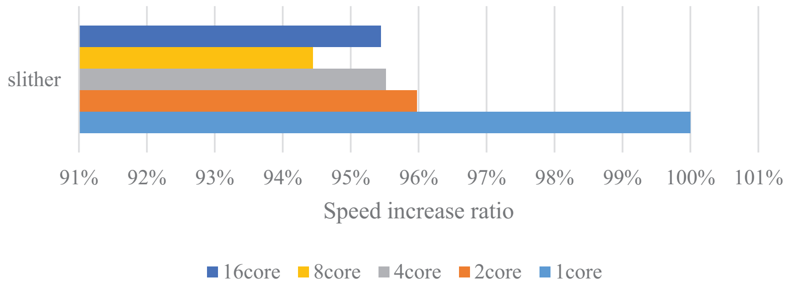


Figure 16: Test results for multi-core detection speed of the Slither

5.5 Memory Usage Test

To assess the memory usage of the system, we conducted a series of memory occupancy tests. These tests aimed to measure the amount of memory the system uses under different load conditions and provide us with crucial indicators to evaluate the system's resource utilization.

During the memory occupancy tests, we first ensured that the system was in a stable state and recorded the initial memory usage. Then, we employed various tools to test different types of contracts, such as withdrawal contracts, game contracts, gambling contracts, and more, to simulate diverse usage scenarios. Throughout the execution process of each test scenario, we continuously monitored the system's memory usage and recorded the memory occupancy at each time point.

To accurately measure memory occupancy, we utilized the psutil [82] module, a professional Python library for monitoring system resource usage. Throughout the testing process, we consistently used 16 GB of memory for the tests and employed the psutil module to record the memory occupancy of each contract at different time intervals.

After analyzing the experimental results in Fig. 17, we observed that Mythril and Honey Badger consume a significant amount of memory during operation, reaching 13.76 and 3.952 GB, respectively. In contrast, Solhint, Solmet, and SmartCheck exhibit noticeably lower memory usage rates, with only 82, 98, and 393 MB, respectively. Therefore, it can be concluded that symbol execution tools with deep path coverage require a higher amount of memory. Conversely, tools based on feature matching require lower memory resources.

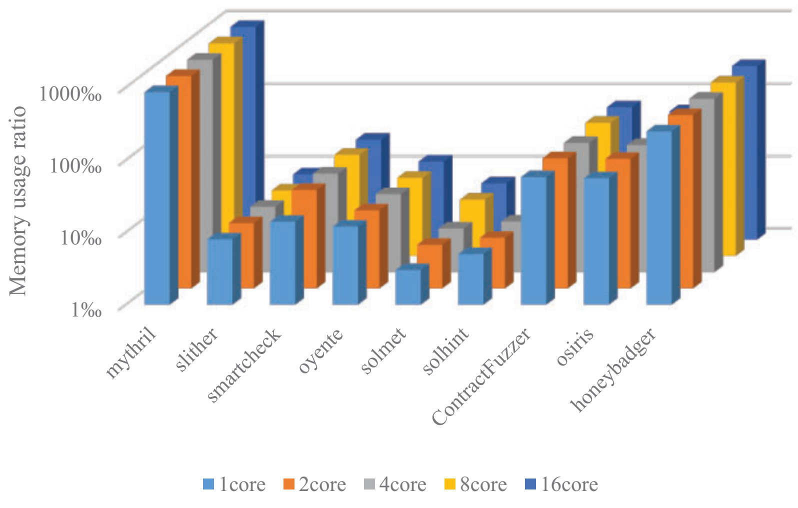


Figure 17: Test on memory usage of detection tools

It should be noted that certain tools exhibit a high sensitivity of memory usage to the number of cores. As shown in Fig. 18, Solmet and SmartCheck showed a memory usage increase of up to double when running in multi-threaded mode compared to single-threaded mode. This indicates that increasing the number of cores significantly increases the memory requirements of these tools. This phenomenon could be attributed to the data interaction and synchronization operations involved in parallel processing. It also explains why these tools show a significant improvement in detection speed with higher thread counts.

5.6 Vulnerability Detection Coverage

In this section, we conducted a practical evaluation of the vulnerability detection capability of the aforementioned tool and provided corresponding vulnerability detection results. However, honeybadger, which can only detect honeypot contracts, and Solmet, which does not test the contracts themselves, were excluded from the evaluation.

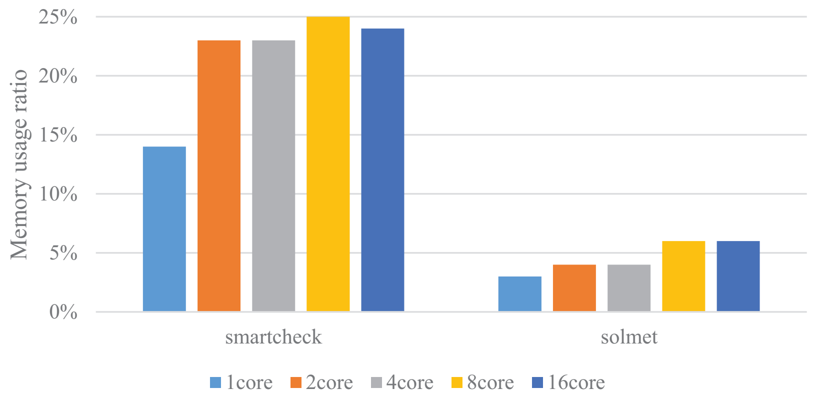


Figure 18: Visualizing the results of a comparative test on memory usage for selected detection tools with different core counts

Fig. 19 illustrates the specific vulnerability detection capabilities of each tool on SB Curated. Among them, Slither, SmartCheck, and Solhint exhibit higher vulnerability coverage compared to other tools. Slither and Solhint can cover all Time Manipulation vulnerabilities, while SmartCheck can cover all Access Control and Unchecked Low-level Calls vulnerabilities. For Reentrancy, Slither tool demonstrates the best coverage. On the contrary, Mythril and SmartCheck have lower coverage for Time Manipulation, and Oyente exhibits poor coverage for Unchecked Low-level Calls.

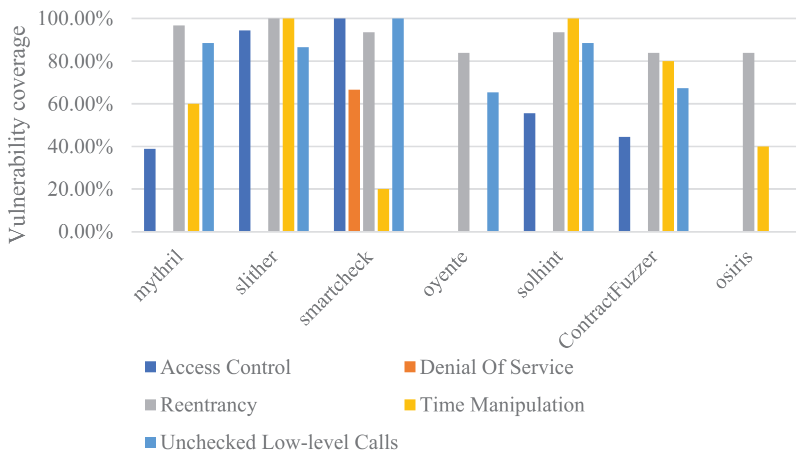


Figure 19: Vulnerability detection capabilities of each tools

6 Conclusion

This paper presents a systematic review of currently available open-source smart contract vulnerability detection tools. We conducted a thorough analysis of the installation complexity of different open-source tools and their dependencies, providing a point of reference for future work.

Based on our current work, we propose a vulnerability detection framework to assist smart contract developers in analyzing vulnerabilities and ensuring the security of their contracts. Additionally, we introduce a network topology that corresponds to the cloud-based vulnerability detection framework. To evaluate the performance of the tools used in the framework, extensive testing was performed. The computational resource consumption and multi-core optimization of these tools were evaluated by monitoring their CPU usage and memory consumption during large-scale smart contract instrumentation. This allowed us to determine the processing power and memory capacity required for actual deployment during contract analysis. This work also provides a reference for future research. However, this paper did not evaluate the methods used by non-open-source tools, resulting in the omission of some relevant studies and thereby affecting the comprehensiveness of the review.

In future work, we aim to enhance and expand our vulnerability detection framework by integrating more smart contract vulnerability detection tools. As new vulnerability types and attack techniques are constantly emerging, we will monitor these developments closely and collaborate with the research community and security experts to update our research on vulnerability detection tools in a timely manner.

Acknowledgement: I would like to express my heartfelt thanks to my colleagues, fellow researchers, and mentors for their constant support, insightful discussions, and constructive feedback throughout the course of this project. Their guidance and expertise have been invaluable in shaping the direc-tion and improving the outcomes of this research. I am also grateful to the technical support team for their invaluable assistance in ensuring the availability and functionality of the necessary equipment and software required for conducting experiments and analyzing data. Their expertise and prompt response to technical issues have significantly contributed to the quality and reliability of the results obtained.

Funding Statement: The work is supported by the Major Public Welfare Special Fund of Henan Province (No. 201300210200), and the Major Science and Technology Research Special Fund of Henan Province (No. 221100210400).

Author Contributions: The authors confirm contribution to the paper as follows: study conception and design: Yaqiong He, Jinlin Fan; data collection: Jinlin Fan; analysis and interpretation of results: Jinlin Fan, Yaqiong He, HuaiguangWu; draft manuscript preparation: Jinlin Fan, Yaqiong He. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: The dataset used in this study is the SmartBugs Curated dataset, which is a publicly available collection of Ethereum smart contracts with known vulnerabilities. The dataset is hosted on GitHub at the following link: https://github.com/smartbugs/smartbugs-curated (accessed on 15/03/2024).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” Accessed: Mar. 15, 2024. [Online]. Available: https://bitcoin.org/bitcoin.pdf
[2] N. Szabo, “Formalizing and securing relationships on public networks,” *First Monday*, vol. 2, no. 9, 1997.
[3] S. Lande and R. Zunino, “SoK: Unraveling bitcoin smart contracts,” *Princ. Secur. Trust Lncs*, vol. 10804, p. 217, 2018.
[4] M. Andrychowicz, S. Dziembowski, D. Malinowski, and A. Mazurek, “Secure multiparty computations on bitcoin,” *Commun. ACM*, vol. 59, no. 4, pp. 76–84, 2016. doi: 10.1145/2896386.

[5] V. Buterin, “A next-generation smart contract and decentralized application platform,” *White Paper*, vol. 3, no. 37, pp. 1–2, 2014.
[6] M. Xu, X. Chen, and G. Kou, “A systematic review of blockchain,” *Financ. Innov.*, vol. 5, no. 1, pp. 1–14, 2019. doi: 10.1186/s40854-019-0147-z.
[7] X. Wang, J. He, Z. Xie, G. Zhao, and S. Cheung, “ContractGuard: Defend Ethereum smart contracts with embedded intrusion detection,” *IEEE Trans. Serv. Comput.*, vol. 13, no. 2, pp. 314–328, 2019. doi: 10.1109/TSC.2019.2949561.
[8] M. Wohrer and U. Zdun, “Smart contracts: Security patterns in the ethereum ecosystem and solidity,” in *2018 Int. Workshop Blockchain Orient. Softw. Eng. (IWBOSE)*, Campobasso, Italy, IEEE, 2018, pp. 2–8.
[9] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, “ZEUS: analyzing safety of smart contracts,” in *Network Distrib. Syst. Secur. (NDSS)*, San Diego, CA, USA, 2018, pp. 1–12.
[10] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, “Finding the greedy, prodigal, and suicidal contracts at scale,” in *Proc. 34th Annu. Comput. Secur. App. Conf.*, San Juan, PR, USA, 2018, pp. 653–663.
[11] P. Daian, “Analysis of the DAO exploit,” Accessed: Dec. 20, 2023. [Online]. Available: https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/
[12] V. Dhillon, D. Metcalf, M. Hooper, V. Dhillon, D. Metcalf and M. Hooper, “The DAO hacked,” in *Blockchain Enabled App.: Understand Blockchain Ecosyst. How Make Work You*, 2017, pp. 67–78.
[13] L. Breidenbach, P. Daian, A. Juels, and E. G. Sirer, “An in-depth look at the parity multisig bug,” 2023. Accessed: Mar. 15, 2023. [Online]. Available: https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/
[14] Etherscan, “Beautychain (BEC) token tracker | etherscan,” 2003. Accessed: Mar. 15, 2023. [Online]. Available: https://etherscan.io/token/0xc5d105e63711398af9bbff092d4b6769c82f793d
[15] A. Singh, R. M. Parizi, Q. Zhang, K. R. Choo, and A. Dehghantanha, “Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities,” *Comput. Secur.*, vol. 88, pp. 101654, 2020. doi: 10.1016/j.cose.2019.101654.
[16] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum Project Yellow Paper*, vol. 151, no. 2014, pp. 1–32, 2014.
[17] J. L. Zhao, S. Fan, and J. Yan, “Overview of business innovations and research opportunities in blockchain and introduction to the special issue,” *Financ. Innov.*, vol. 2, no. 1, pp. 1–7, 2016. doi: 10.1186/s40854-016-0049-2.
[18] L. W. Cong and Z. He, “Blockchain disruption and smart contracts,” *Rev. Financ. Stud.*, vol. 32, no. 5, pp. 1754–1797, 2019. doi: 10.1093/rfs/hhz007.
[19] Y. Wang, J. H. Han, and P. Beynon-Davies, “Understanding blockchain technology for future supply chains: A systematic literature review and research agenda,” *Supply Chain Manag.: Int. J.*, vol. 24, no. 1, pp. 62–84, 2019. doi: 10.1108/SCM-03-2018-0148.
[20] A. Reyna, C. Martín, J. Chen, E. Soler, and M. Díaz, “On blockchain and its integration with IoT. Challenges and opportunities,” *Future Gen. Comput. Syst.*, vol. 88, no. 3, pp. 173–190, 2018. doi: 10.1016/j.future.2018.05.046.
[21] A. H. Lone and R. Naaz, “Applicability of blockchain smart contracts in securing internet and IoT: A systematic literature review,” *Comput. Sci. Rev.*, vol. 39, no. 1, pp. 100360, 2021. doi: 10.1016/j.cosrev.2020.100360.
[22] K. Peng, M. Li, H. Huang, C. Wang, S. Wan and K. R. Choo, “Security challenges and opportunities for smart contracts in internet of things: A survey,” *IEEE Internet Things J.*, vol. 8, no. 15, pp. 12004–12020, 2021. doi: 10.1109/JIOT.2021.3074544.
[23] S. Dustdar, P. Fernández, J. M. García, and A. Ruiz-Cortés, “Elastic smart contracts in blockchains,” *IEEE/Caa J. Autom. Sin.*, vol. 8, no. 12, pp. 1901–1912, 2021. doi: 10.1109/JAS.2021.1004222.
[24] J. Li and M. Kassem, “Applications of distributed ledger technology (DLT) and blockchain-enabled smart contracts in construction,” *Autom. Constr.*, vol. 132, no. 1, pp. 103955, 2021. doi: 10.1016/j.autcon.2021.103955.

[25] D. Kirli *et al.*, "Smart contracts in energy systems: A systematic review of fundamental approaches and implementations," *Renew. Sustain. Energ. Rev.*, vol. 158, no. 1, pp. 112013, 2022. doi: 10.1016/j.rser.2021.112013.

[26] M. Alharby, A. Aldweesh, and A. van Moorsel, "Blockchain-based smart contracts: a systematic mapping study of academic research, 2018," in *2018 Int. Conf. Cloud Comput., Big Data Blockchain (ICCBB)*, Fuzhou, China, IEEE, 2018, pp. 1–6.

[27] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems," *Future Gen. Comput. Syst.*, vol. 107, pp. 841–853, 2020. doi: 10.1016/j.future.2017.08.020.

[28] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (SOK)," in *Princ. Secur. Trust: 6th Int. Conf.*, Uppsala, Sweden, Springer, 2017, pp. 164–186.

[29] L. Zhu, B. Zheng, M. Shen, F. Gao, H. Li and K. Shi, "Data security and privacy in bitcoin system: A survey," *J. Comput. Sci. Technol.*, vol. 35, no. 4, pp. 843–862, 2020. doi: 10.1007/s11390-020-9638-7.

[30] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko and Y. Alexandrov, "SmartCheck: Static analysis of ethereum smart contracts," in *Proc. 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, Gothenburg, Sweden, 2018, pp. 9–16.

[31] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, Montreal, QC, Canada, IEEE, 2019, pp. 8–15.

[32] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *Proc. 2005 Int. Workshop Min. Softw. Repositor.*, St. Louis, Missouri, 2005, pp. 1–5.

[33] F. E. Allen, "Control flow analysis," *ACM Sigplan Not.*, vol. 5, no. 7, pp. 1–19, 1970. doi: 10.1145/390013.808479.

[34] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *NDSS*, San Diego, CA, USA, 2008, pp. 151–166.

[35] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976. doi: 10.1145/360248.360252.

[36] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software," in *NDSS*, Citeseer, San Diego, CA, USA, 2005, pp. 3–4.

[37] J. Ye, M. Ma, Y. Lin, L. Ma, Y. Xue and J. Zhao, "VULPEDIA: Detecting vulnerable ethereum smart contracts via abstracted vulnerability signatures," *J. Syst. Softw.*, vol. 192, no. 6, pp. 111410, 2022. doi: 10.1016/j.jss.2022.111410.

[38] Y. Liu, Y. Li, S. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Republic of Korea, 2022, pp. 716–727.

[39] Y. Xue *et al.*, "XFuzz: Machine learning guided cross-contract fuzzing," *IEEE Trans. Dependable Secur. Comput.*, vol. 21, no. 2, pp. 515–529, 2022.

[40] K. Song, N. Matulevicius, E. B. de Lima Filho, and L. C. Cordeiro, "ESBMC-solidity: An SMT-based model checker for solidity smart contracts," in *Proc. ACM/IEEE 44th Int. Conf. Softw. Eng.: Companion Proc.*, Pittsburgh, PA, USA, 2022, pp. 65–69.

[41] A. Ghaleb, J. Rubin, and K. Pattabiraman, "eTainter: Detecting gas-related vulnerabilities in smart contracts," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Republic of Korea, 2022, pp. 728–739.

[42] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel and G. Vigna, "SAILFISH: Vetting smart contract state-inconsistency bugs in seconds," in *2022 IEEE Symp. Secur. Priv. (SP)*, San Francisco, CA, USA, IEEE, 2022, pp. 161–178.

[43] Y. Chinen, N. Yanai, J. P. Cruz, and S. Okamura, "RA: A static analysis tool for analyzing re-entrancy attacks in ethereum smart contracts," *J. Inf. Process.*, vol. 29, pp. 537–547, 2021. doi: 10.2197/ipsjjip.29.537.

[44] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura, "Eth2Vec: Learning contract-wide code representations for vulnerability detection on ethereum smart contracts," in *Proc. 3rd ACM Int. Symp. Blockchain Secur. Crit. Infrastruct.*, Hong Kong, China, 2021, pp. 47–59.

[45] F. Contro, M. Crosara, M. Ceccato, and M. Dalla Preda, "EtherSolve: Computing an accurate control-flow graph from ethereum bytecode," in *2021 IEEE/ACM 29th Int. Conf. Program Compr. (ICPC)*, Madrid, Spain, IEEE, 2021, pp. 127–137.

[46] B. Nassirzadeh, H. Sun, S. Banescu, and V. Ganesh, "Gas Gauge: A security analysis tool for smart contract out-of-gas vulnerabilities," in *Math. Res. Blockchain Econ.: 3rd Int. Conf. MARBLE 2022, Vilamoura, Portugal*, Vilamoura, Portugal, Springer, 2023, pp. 143–167.

[47] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: Effective, usable, and fast fuzzing for smart contracts," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, USA, 2020, pp. 557–560.

[48] D. Wang, B. Jiang, and W. K. Chan, "WANA: Symbolic execution of wasm bytecode for cross-platform smart contract vulnerability detection," arxiv Preprint arxiv:2007.15510, 2020.

[49] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *IEEE Trans. Softw. Eng.*, vol. 47, no. 12, pp. 2874–2891, 2020. doi: 10.1109/TSE.2020.2971482.

[50] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "VERISMART: A highly precise safety verifier for ethereum smart contracts," in *2020 IEEE Symp. Secur. Priv. (SP)*, San Francisco, CA, USA, IEEE, 2020, pp. 1678–1694.

[51] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sFuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, Seoul, Republic of Korea, 2020, pp. 778–788.

[52] J. Frank, C. Aschermann, and T. Holz, "ETHBMC: A bounded model checker for smart contracts," in *Proc. 29th USENIX Conf. Secur. Symp.*, Berkeley, CA, USA, 2020, pp. 2757–2774.

[53] Y. Fu *et al.*, "Evmfuzzer: detect EVM vulnerabilities via fuzz testing," in *Proc. 2019 27th ACM Joint Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Tallinn, Estonia, 2019, pp. 1110–1114.

[54] P. Zhang, F. Xiao, and X. Luo, "Soliditycheck: quickly detecting smart contract problems through regular expressions," arxiv Preprint arxiv:1911.09425, 2019.

[55] P. Zhang, F. Xiao, and X. Luo, "Soliditycheck/soliditycheck.pdf at master · xf97/soliditycheck," 2023. Accessed: Mar. 20, 2023. [Online]. Available: https://github.com/xf97/SolidityCheck

[56] H. Wang, Y. Li, S. Lin, L. Ma, and Y. Liu, "Vultron: Catching vulnerable smart contracts once and for all," in *2019 IEEE/ACM 41st Int. Conf. Softw. Eng.: New Ideas Emerg. Results (ICSE-NIER)*, Montreal, Quebec, Canada, IEEE, 2019, pp. 1–4.

[57] A. Mavridou *et al.*, "VeriSolid: Correct-by-design smart contracts for ethereum," in *Financ. Cryptogr. Data Secur.: 23rd Int. Conf.*, Frigate Bay, St. Kitts, Nevis, Springer, 2019, pp. 446–465.

[58] C. Peng, S. Akca, and A. Rajan, "SIF: A framework for solidity contract instrumentation and analysis," in *2019 26th Asia-Pacific Softw. Eng. Conf. (APSEC)*, Putrajaya, Malaysia, IEEE, 2019, pp. 466–473.

[59] Z. Yang and H. Lei, "FEther: An extensible definitional interpreter for smart-contract verifications in coq," *IEEE Access*, vol. 7, pp. 37770–37791, 2019. doi: 10.1109/ACCESS.2019.2905428.

[60] L. Brent *et al.*, "Vandal: A scalable security analysis framework for smart contracts," arxiv Preprint arxiv:1809.03981, 2018.

[61] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. 2018 ACM SIGSAC Conf. Comput. Commun. Secur.*, Toronto, ON, Canada, 2018, pp. 67–82.

[62] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "ETHIR: A framework for high-level analysis of ethereum bytecode," in *Autom. Tech. Verif. Anal.: 16th Int. Symp.*, Los Angeles, CA, USA, Springer, 2018, 513–520.

[63] M. Suiche, "Porosity: A decompiler for blockchain-based smart contracts bytecode," *Def. Con.*, vol. 25, no. 11, pp. 11, 2017.

[64] A. Mavridou and A. Laszka, "Tool demonstration: Fsolidm for designing secure ethereum smart contracts," in *Princ. Secur. Trust: 7th Int. Conf., POST 2018, Held Part Eur. Joint Conf. Theory Pract. Softw.*, Thessaloniki, Greece, Springer, 2018, pp. 270–277.

[65] E. Hildenbrandt *et al.*, "KEVM: A complete semantics of the ethereum virtual machine," 2017. Accessed: Mar. 10, 2024. [Online]. Available: https://www.ideals.illinois.edu/items/102260

[66] Ant, “Antlr,” Accessed: Mar. 10, 2024. [Online]. Available: http://www.antlr.org/
[67] Consensys, “Consensys/mythril,” Accessed: Mar. 15, 2024. [Online]. Available: https://github.com/ConsenSys/mythril
[68] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” in *Proc. 2016 ACM SIGSAC Conf. Comput. Commun. Secur.*, Vienna, Austria, 2016, pp. 254–269.
[69] P. Hegedűs, “Towards analyzing the complexity landscape of solidity based ethereum smart contracts,” in *Proc. 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, Gothenburg, Sweden, 2018, pp. 35–39.
[70] Protofire, “Solhint,” 2023. Accessed: Mar. 22, 2023. [Online]. Available: https://protofire.github.io/solhint/
[71] B. Jiang, Y. Liu, and W. K. Chan, “Contractfuzzer: Fuzzing smart contracts for vulnerability detection,” in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, Montpellier, France, 2018, pp. 259–269.
[72] C. F. Torres, J. Schütte, and R. State, “Osiris: Hunting for integer bugs in ethereum smart contracts,” in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, San Juan, PR, USA, 2018, pp. 664–676.
[73] C. F. Torres and M. Steichen, “The art of the scam: Demystifying honeypots in ethereum smart contracts,” in *28th USENIX Secur. Symp. (USENIX Secur. 19)*, Santa Clara, CA, USA, 2019, pp. 1591–1607.
[74] McGill-DMaS, “Github-mcgill-dmas/kam1n0-community: the kam1n0 assembly analysis platform,” 2023. Accessed: Mar. 03, 2023. [Online]. Available: https://github.com/McGill-DMaS/Kam1n0-Community
[75] Gradle, “Gradle build tool,” 2023. Accessed: Mar. 03, 2023. [Online]. Available: https://gradle.org/
[76] Homebrew, “Homebrew,” 2023. Accessed: Mar. 03, 2023. [Online]. Available: https://brew.sh/
[77] NixOS, “Download nix/nixos,” 2023. Accessed: March 03, 2023
[78] Commercialhaskell, “The Haskell tool stack,” 2023. Accessed: Mar. 03, 2023. [Online]. Available: https://docs.haskellstack.org/en/stable/
[79] A. López Vivar, A. L. Sandoval Orozco, and L. J. García Villalba, “A security framework for ethereum smart contracts,” *Comput. Commun.*, vol. 172, no. 5, pp. 119–129, 2021. doi: 10.1016/j.comcom.2021.03.008.
[80] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, “SmartBugs: A framework to analyze solidity smart contracts,” in *Proc. 35th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Australia, 2020, pp. 1349–1352.
[81] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, “Empirical review of automated analysis tools on 47,587 ethereum smart contracts,” in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, Seoul, Republic of Korea, 2020, pp. 530–541.
[82] Giampaolo, “Psutil,” Accessed: Mar. 15, 2024. [Online]. Available: https://github.com/giampaolo/psutil