GSVD: COMMON VULNERABILITY DATASET FOR SMART CONTRACTS ON BSC AND POLYGON

Ziniu Shen[1], Yunfang Chen[2] and Wei Zhang[3]

School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, China

ABSTRACT

The blockchain 2.0 age, marked by smart contract and Ethereum, has arrived couple years ago. Its technologies have expanded the application scenarios of blockchain technology and driven the boom of decentralized Finance. However, smart contract vulnerabilities and security issues are also emerging one after another. Hackers have exploited these vulnerabilities to cause huge economic losses. In recent years, a large amount of research on the analysis and detection of smart contract vulnerabilities has emerged, but there has been no common detection tool and corresponding test dataset. In this paper, we build GSVD dataset (Generalized Smart Contract Vulnerability Dataset) consisting four offline datasets using smart contracts on two chains, Polygon and BSC: two small Solidity datasets consisting of 153 labeled smart contract source codes, which can be used to test the performance of vulnerability mining tools; two large Solidity datasets consisting of 52,202 un labeled real smart contract source codes that can be used to verify the correctness of various theories and tools under a large number of real data conditions. At the same time, this paper integrates the scripting framework accompanying the GSVD dataset, which can execute a variety of popular automated vulnerability detection tools on top of these datasets and generate analysis results of contracts and potential vulnerabilities. We tested the Minor dataset under GSVD using three tools (Slither, Manticore, Mythril) that are kept up to date and found that the combined use of all tools detected 61.1% of labeled vulnerabilities, of which Mythril has the highest detection rate of 42.6%. It is not difficult to conclude that there`re still ample room for advancement for current smart contract vulnerability mining tools because of their underlying methods. Besides, our dataset can contribute to the ultimate target greatly by providing mining tools plenty real contracts information.

KEYWORDS

Smart Contract, Blockchain, Security, Vulnerability Detection, Dataset

1. INTRODUCTION

Since its birth, blockchain technology has gained widespread attention, developed numerous applications and occupied an increasing market share in the fields of computing and finance by virtue of its revolutionary decentralization and immutability. In 2013, Vitalik Buterin[1] released an Ethereum white paper, introducing smart contracts and Ethereum virtual machines (EVMs) for blockchain, marking the arrival of Blockchain 2.0. After several years of development, more and more platforms that support smart contract and EVM have emerged. Although Ethereum still dominates, other platforms have gradually grabbed more and more shares through iterative updates, which are represented by BSC, Polygon, Avalanche, etc. Ethereum`s share of total value locked by platforms has dropped all the way from 98% at the peak to 70.7% as shown in Figure 1

David C. Wyld et al. (Eds): ICAITA, CDKP, CBW, SOFT, CMC, Signal, MLT-2023
pp. 01-16, 2023. CS & IT - CSCP 2023 DOI: 10.5121/csit.2023.130601

and still keeps decreasing, while the other four platforms in the top five possess a combined share of more than 20%. In addition to differences such as Layer-2 optimizations, hard forks, and reconfigured consensus mechanisms for Ethereum flaws, these platforms all require the use of smart contracts, written through the Solidity language and executed in an EVM environment, as a result, they still have to face the inevitable security vulnerabilities of smart contracts themselves.

In today's explosive growth of decentralized finance (DeFi) and decentralized applications (DApp), the number of smart contracts continues to increase and contain more and more complex functions. However, in the context of development and iteration of the platform superior to anything else, the corresponding review and auditing mechanism are obviously not being paid enough attention. Meanwhile, the system and organization of anti-exploit are not well developed, all these factors have led to a proliferation of hacking attacks in recent years, caused huge losses to the market and users. In order to solve these problems, put aside the emergency countermeasures of the major platforms themselves, researchers in the field of security are also devoted to this area, invented a lot of vulnerability detection methods and tools. Tsankov *et al.*[2] proposed the tool Securify and found that 65.9% of the 24,594 real EVM contracts were containing vulnerabilities or errors. Luu *et al.*[3] proposed the symbolic execution tool Oyente and tested 19,366 real EVM contracts, of which 8,833 were flagged as containing vulnerabilities. Nikolic *et al.*[4] tested nearly 1 million contracts against three specific vulnerabilities, of which 34,200 were vulnerable. However, these methods and tools generally focus on only some of the vulnerabilities and detect specific types of vulnerabilities through specific methods, therefore these efforts have considerable limitations. In addition to open-source methods and tools from academia, most of the vulnerability detection methods and tools developed by blockchain platforms or private security companies are closed-source. Besides, we found that even if choose open-source tools, people will face the dilemma of lacking data in the process of validating existing findings or making comparison of the performance between chosen tool and new tool. Usually, people need to contact the author to get access to the test dataset of the tool but many authors stop updating and maintaining the tool after their graduation, which makes it difficult to get in touch and retrieve data.

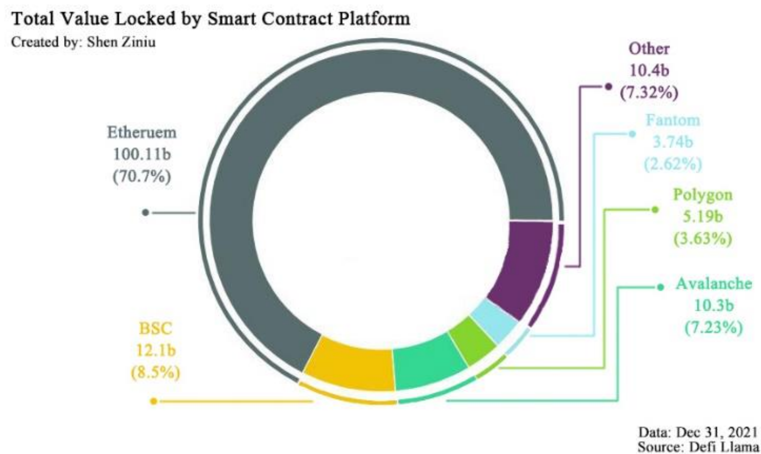

Figure 1. Total Value Locked by Smart Contract Platform by Dec,31,2021

In order to solve the above-mentioned real-life problems, we made targeted attempts and achieved some viable results, our purposes and work are as follows. First, considering the large scale and wide popularity of the Ethereum, both the quality of user and contract are influenced by the number of contracts, a significant number of beginners are using Ethereum to learn about blockchain-related content and a large number of contracts with no real meaning are deployed for only practice or even scams. To ensure fairness and accuracy when comparing the performance

of smart contract vulnerability detection tools, we chose the smart contract platform BSC and Polygon, which are second only to Ethereum in terms of total value locked by platforms, to provide 2 kinds of data sets with a total of 4 copies. One of them consists of a small Solidity dataset containing the source code of several manually-labelled smart contracts and one of them consists of a large Solidity dataset containing the source code of several labelled real smart contracts. The small datasets for BSC and Polygon have 56 and 97 real contracts respectively, the large datasets have 10,600 and 41,602 real contracts respectively. In these datasets, contracts in small datasets usually have relative few lines of code and functions and contain one or more vulnerabilities. They are manually reviewed, auditors filtered and labelled eligible contracts to undertake performance testing of vulnerability mining tools, comparing it with existing tools and other duties. Contracts in large datasets are characterized by a relatively large number of lines of code and functions, uneven distribution, some containing vulnerabilities and some not, etc. They are selected from real contracts deployed on the chain over a period of time and can be used to take on the responsibility of big data analysis of information carried by real-world smart contracts. Second, in order to better unify existing open-source vulnerability detection tools and make them easy to use, we propose a GSVD companion execution script framework that integrates three SOTA vulnerability detection tools to analyse contracts and generate multiple contract content-related analysis reports by simply adjusting the parameters. Last, in order to validate the GSVD dataset and the utility of its scripting framework, we spent nearly 47 hours testing and validating $GSVD^{Minor}$ with Slither, Mythril and Manticore, obtained specific execution and performance analysis results for each tool, from which we also obtained many relevant conclusions for real-world smart contracts and platforms.

2. BACKGROUND AND RELATED WORK

2.1. Smart Contract based Platform

In the mid-1990s, cryptographer Nick Szabo introduced the concept of smart contracts[5], his definition of a smart contract is “A smart contract is a set of promises, specified in digital form, including protocols within which the parties perform on these promises”, but the technology was not widely supported and used due to the mismatch of technical means and the immaturity of the online trust environment at that time. It wasn't until the advent of blockchain technology and the maturation of the Internet at the turn of this century that it became useful. Blockchain supports programmable digital contracts and uses its features like decentralization, tamper proof, transparency and traceability to build a trusted community environment. Vitalik Buterin published the Ethereum white paper in 2013, officially introducing smart contracts to the blockchain, revealing the arrival of the blockchain 2.0 and since then the smart contract platform has become more and more important in computers, finance and many other fields.

Essentially speaking, smart contract is the digitalization of traditional contract, which is deployed on a blockchain network and contains specific triggers that can automatically execute once the real situation meets the trigger conditions. On the blockchain platform, it can carry out coordinated management of electronic assets and data, complete transactions between users. Smart contracts have the advantages of low cost, ease of use, globalization, and high trustworthiness when comparing with traditional contracts.

Smart contract platform usually refers to a blockchain platform that provides standards and interfaces for smart contract development and has an environment in which smart contracts can run. The earliest and by far the most successful smart contract platform is Ethereum, which provides a development interface and runtime environment that has become a model for later generations to emulate or directly reference. In order to better understand the different platforms

and the key points of the work in this paper, we are going to introduce some of the more mainstream smart contract platforms below.

Ethereum is the first blockchain platform to put smart contracts into practical use and the first to provide a complete smart contract development interface. It was originally released by Vitalik Buterin in 2013, unlike Bitcoin, Ethereum provides the Turing-complete programming language Solidity and the corresponding runtime environment EVM (Ethereum Virtual Machine). On this basis, users can write smart contracts and develop decentralized applications easily. In addition, Ethereum also developed many token standards, mainly represented by the token standard ERC20 and the digital asset standard ERC721. Token standards are a series of common program interface specifications in essence, smart contracts created following this standard can be traded with numerous exchanges and wallet accounts. These standardized supports make smart contract development easier and less risky. With the firstness and stability of the technology, Ethereum has gained a huge number of ordinary users and many enterprises. JP Morgan, Bank of New York, Microsoft, Intel and other giants have adopted Ether's smart contract system early. According to Messari, Ethereum`s market cap has now surpassed $340 billion, second only to Bitcoin and way ahead of other blockchain platforms.

The Binance Smart Chain (BSC) was born in April 2019 and it is not a Layer-2 solution, its motivation is to optimize the slow transaction speed of Ethereum, for which BSC uses Proof of Stacked Authority (PoA) consensus mechanism. The main schemes are blocks are mined by a limited number of validators, validators take turns to mine blocks in PoA, and the set of validators is selected and eliminated based on on-chain governance of equity pledges, etc. This reduces the block generation time to about 3 seconds and the transaction to less than 1 minute. In addition, the revenue on the BSC comes from fees, which are paid through BNB, a token that never inflate, so it would not generate mining revenue like Bitcoin or Ethereum. To ensure a user-friendly interactive environment and low learning cost migration, the BSC public chain is compatible with the Ethereum main net and Ethereum virtual machines, which means that a significant percentage of decentralized applications, components and tools can be migrated and run without modification or with only minor modifications from Ethereum to BSC.

Polygon, the platform of the Layer-2 solution for Ethereum, was born in 2021 and is known as the "Guardian of Ethereum", which was formerly known as the original MATIC blockchain on Ethereum to solve the problem of unsatisfactory transaction costs and time costs on Ethereum. There are two different types of chains in the Polygon ecosystem: the Stand-Alone Chain and the Secured Chain. The Stand-Alone Chain has its own unique consensus mechanism, which is more flexible but less secure than the Secured Chain, which uses the Ethereum blockchain as its underlying layer and is directly guaranteed by Ethernet. Polygon exists as a commit chain for Ethereum, with a tree-like of sub-chain created on top of the underlying layer, allowing transactions to be processed in batches before being sent to the underlying layer. This means that users on Polygon can create off-chain transactions to pay and interact with smart contracts, which significantly reduces transaction costs and increases transaction speed. Users can choose a Stand-Alone Chain or Secured Chain based on their specific needs and take advantage of Polygon's high scalability for smart contracts and decentralized application development.

2.2. Smart Contract Vulnerability and Detection Tools

Blockchain technology`s feature like tamper proof makes it impossible to change a smart contract once it is on the chain, hackers can get the source code by decompiling the bytecode. Many projects adhering to the spirit of open source, directly disclose the source code of smart contracts, which also greatly reduce the difficulty and cost of hacking. In addition, the nature of smart contracts is code, but even experienced coders cannot guarantee whether the contracts have

vulnerabilities that can be exploited, plus the operating environment of smart contract platforms cannot guarantee that there are no native vulnerabilities or vulnerabilities that exist after interacting with different contracts, vulnerabilities and security issues have been with developers and users since the beginning of smart contracts. In 2016, The DAO project on Ethereum was hacked[6] and hackers directly diverted 3.6 million Ethereum tokens, accounting for 1/3 of the total project crowdfunding at the time. In 2018, a major security vulnerability similar to the BEC overflow occurred in SmartMesh, resulting in the theft of over $140 million by hackers. In April 2022, hackers attacked the Beanstalk Farms project through a malicious code logic vulnerability, which ultimately cost the project about $182 million. Smart contracts and the platforms themselves have been in the shadow of vulnerabilities and hackers.

With the advancement of blockchain and smart contract technology, decentralized applications and decentralized finance have also seen explosive growth, the booming market has brought about a continuous expansion in the number and scale of smart contracts. In order to meet the needs of more users, functions within the contract continue to become more complex and consequently, a wider variety of contract vulnerabilities and more serious problems begin to emerge. The NCC Group has previously summarized the issues related to vulnerabilities that appear more frequently and have a higher security threat[7] like *Reentrancy*, *Access Control*, *Arithmetic*, *Unchecked low-level calls*, *Denial of service*, *Time manipulation*, *Short address*, etc. Although the vulnerabilities in the early stages of smart contracts have been fixed and prevented to some extent by the platform and community, smart contract platforms or exchanges can only implement mandatory locking of transactions and accounts to rescue assets after an attack. Vulnerability detection for on-chain smart contracts is not technically or cost feasible, so contract vulnerability detection before deployment is almost the only option available to users.

For those who can afford, vulnerability detection before smart contracts` deployment on the chain is often manually reviewed by experienced security and coding experts, surely, the cost of which is high in terms of money and time. In this condition, low-cost open-source smart contract vulnerability mining tool is the right choice for the majority of ordinary users. A. López Vivar *et al[8]* have conducted a preliminary study on open-source smart contract vulnerability detection tools and proposed the ESAF framework. In order to find the right detection tool, we have further investigated these tools based on the selection of ESAF, and the list as well as the details are summarized in Table 1. Considering that the versions of Solidity language and Solc compiler have been iterating at a high speed, we finally chose Slither, Manticore and Mythril, three tools that have been kept up-to-date, to perform the detection work on the dataset.

Table 1. Potential detection tools for subsequent tests.

Tools	Ease of Installation	Up to Date	Tool URLs
ContractLarva [9]	Easy	No	https://github.com/gordonpace/contractLarva
Erays[10]	Medium	No	https://github.com/teamnsrg/erays
EtherTrust[11]	Hard	No	https://www.netidee.at/ethertrust
EthIR[12]	Hard	No	https://github.com/costa-group/EthIR
MadMax[13]	Medium	No	https://github.com/nevillegrech/MadMax
MAIAN[4]	Hard	No	https://github.com/MAIAN-tool/MAIAN
Manticore[14]	Medium	Yes	https://github.com/trailofbits/manticore/
Mythril[15]	Medium	Yes	https://github.com/ConsenSys/mythril-classic
Osiris[16]	Hard	No	https://github.com/christoftorres/Osiris
Oyente[3]	Hard	No	https://github.com/melonproject/oyente
Rattle	Easy	No	https://github.com/crytic/rattle
Securify[2]	Easy	No	https://github.com/eth-sri/securify
Slither[17]	Easy	Yes	https://github.com/crytic/slither
SmartCheck[18]	Easy	No	https://github.com/smartdec/smartcheck
Solgraph	Easy	No	https://github.com/raineorshine/solgraph
SolMet[19]	Easy	No	https://github.com/chicxurug/SolMet
Vandal[20]	Easy	No	https://github.com/usyd-blockchain/vandal

3. GSVD: GENERALIZED SMART CONTRACT VULNERABILITY DATASE

3.1. Steps to Generate the Dataset

GSVD is the abbreviation for Generalized Smart Contract Vulnerability Dataset. Its original intent was to provide the blockchain community with a reliable, sensible, and easy-to-use dataset, one that goes beyond just collecting data and then displaying it directly. To achieve this goal, we have designed several meticulous construction steps, which are shown in Figure 2. We have also written various automation scripts to match the proper operation of each step in the big data scenario.

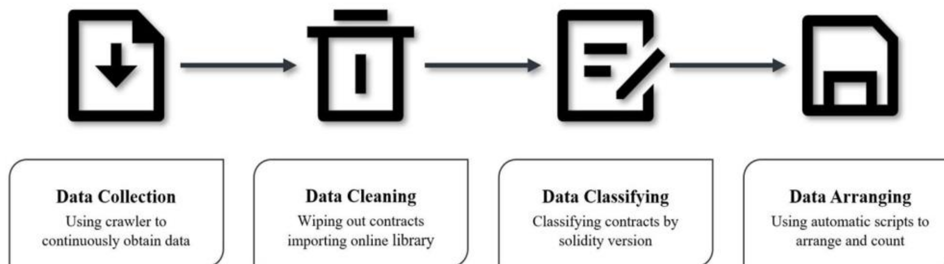

Figure 2. Steps to generate GSVD

The first thing to experience in organizing data is data collection. After investigation, we found that Google BigQuery [21] provides blockchain-related public datasets but is currently open source only for Bitcoin and Ethereum data. We then learned that PolygonScan and BscScan, two third-party websites, provide APIs related to queries and data downloads for smart contracts. After a short period of extensive data testing, we found that the APIs of these sites had restrictions such as an upper limit on the number of operations, so we wrote specific crawler scripts to obtain a variety of information such as contract source code from these two sites. Considering that the technical difficulty of deploying contracts is small and there will be

contracts that are not deployed for the purpose of use, we also set the condition that the current number of transactions is greater than 1 to perform the initial filter. After about six months of collection, we eventually acquired 77,178 contracts.

After the collection of data, the initial construction of the dataset is completed. The next step is to filter and clean the collected data. Through our observation of Solidity contracts, we found that a significant number of contracts import third-party online security libraries such as OpenZeppelin, which will inevitably have an impact on detection efforts in offline environments. So, we performed text reading of all contracts and cleaned out all contracts that have imported third-party online libraries, which totalled 25,218 contracts, accounting for 33% of the total collected contracts.

Before classifying the dataset, it is important to note that contracts written in the Solidity will fail to execute when run in EVM because the Solc compiler version in the environment does not match the version declared in the contract. Therefore, we sampled the Solidity language and Solc version compatibility verification at first and we found that the final version of each generation is downward compatible with all versions of this generation. For instance, the final version of the generation 0.4, which is version 0.4.26, can be compatible with all contracts running between version 0.4.0 and version 0.4.26. We then proceeded to perform text reading of all contracts in the dataset and classified the contracts according to the Solidity version declared in it, dividing the dataset into six broad categories C4, C5, C6, C7, C8, and CU. Among which CU stands for Category Useless, containing a total of 246 contracts, which were also removed as obsolete cases.

3.2. Composition of GSVD

The GSVD consists entirely of real-world smart contracts written in Solidity and can be divided into two categories. The first category is $GSVD^{Minor}$, which contains several real-world smart contracts labeled with vulnerabilities and can be used to test the performance of vulnerability mining tools. The second category is $GSVD^{Major}$, which contains several real-world smart contracts with un labeled vulnerabilities and can be used to verify the correctness of various theories and tools under a large number of real data conditions. Considering the large-scale popularity of the Ethereum, the quality of users and contracts of this platform is compromised by the quantity. For example, a considerable number of beginners are using Ether to learn blockchain-related content and a large number of contracts with no realistic meaning for practice tests or even scams are deployed. So, in the first stage of dataset construction, we chose BSC and Polygon, the smart contract platforms whose total value locked by platform second to Ethereum, as the data source. The initial construction has been completed on the basis of these two platforms, whereby they can be further subdivided into ${GSVD^{Minor}}_{Polygon}$, ${GSVD^{Minor}}_{BSC}$, ${GSVD^{Major}}_{Polygon}$ and ${GSVD^{Major}}_{BSC}$. In the future, when we enter the subsequent phase of dataset construction, we can also continue to build extensions on different platforms. The specific structure of GSVD is shown in Figure 3.

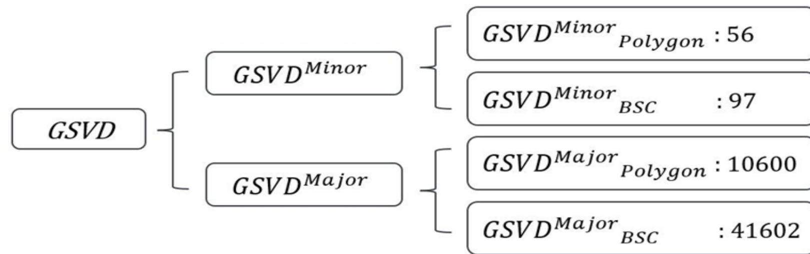

Figure 3. Structure and basic content of GSVD

3.2.1. GSVD^{Minor}: Small Dataset Focused on Vulnerabilities

The first thing to consider is the application scenario when building GSVD^{Minor}. Our vision is to provide blockchain security practitioners with a class of real-world smart contract datasets containing typical vulnerabilities. When researchers want to test their algorithms or tools on real-world contracts rather than typical vulnerability example contracts, they can experiment on the GSVD^{Minor} dataset and get a result that has been tested with real-world vulnerabilities. They can also perform control variates tests on open-source tools that do not open test dataset to compare the performance of their own algorithms or tools.

We combined a variety of practical situations, discussed and selected 9 typical vulnerabilities for the construction of the dataset and subsequent validation of the detection tool. When making the selection of specific contracts, we first used an automatic script to filter the C4 to C8 categories to get a batch of contracts with 100-300 lines of code in order to ensure the readability of the code, then obtained 83 Polygon contracts and 117 BSC contracts by random selection, while ensuring that at least one contract was selected in each category. Finally, these contracts are manually screened, those with more serious vulnerabilities are recorded and marked, those do not contain vulnerabilities or contain low-level vulnerabilities that do not affect their use are removed. After these steps of processing, we obtained 56 Polygon contracts containing 140 vulnerabilities and 97 BSC contracts containing 228 vulnerabilities, with the specific information shown in Table 2.

Table 2. Information and Details of Vulnerabilities in GSVD^{Minor}.

Categories	**Description**	**Level**	**Contracts/Vulns**	
			Polygon	**BSC**
Reentrancy	Contract fails to run calls of re-entrant function	Solidity	25/32	46/53
Access Control	Loss of contract authority caused by ambiguous function authority	Solidity	21/25	42/47
Time Manipulation	Manipulation of block timestamp	Blockchain	20/23	28/32
Illegal delegate call	Improper use of delegate call generates error	Solidity	6/7	3/3
Arithmetic	Integer over/underflows	Solidity	9/10	43/48
Denial of service	Execution of contract is overwhelmed with time-consuming computations	Blockchain	7/9	4/5
Trap of initialization	Contract is overridden in the cause of uninitialized variables	Solidity	9/9	9/12
Lock of balance	The balance is locked with no fallback function	Solidity	9/10	12/13
Token standard	Interaction failure caused by lack of token or grammar standardization	Solidity	15/15	14/15

In order to give readers a better understanding of GSVD^{Minor}, we have done a preliminary visualization of the two subsets and presented the number and distribution of different kinds of vulnerabilities in the dataset through Figure 4, which also reflects the distribution of smart contract vulnerabilities in the real environment to a certain extent. Reentrancy, Access Control, Arithmetic and Time Manipulation are the easier vulnerabilities to attack, while the gains from these vulnerabilities are relatively large.

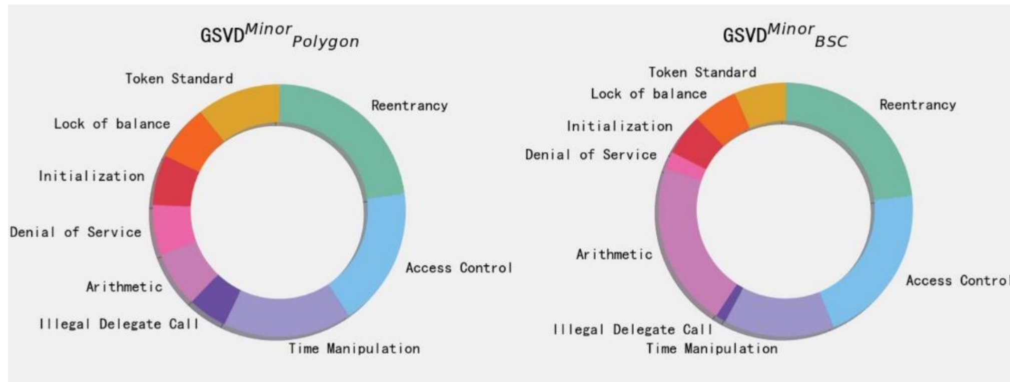

Figure 4. Number and distribution of vulnerabilities in $GSVD^{Minor}$

3.2.2. $GSVD^{Major}$: Large Dataset Focused on Real-world Environments

In addition to $GSVD^{Minor}$, which is used for performance testing of vulnerability detection tools, there is GSVDMajor, which is used for big data analysis and experiment on contracts and platforms in real-world environments. A dataset that consistently integrates a large number of deployed smart contracts in real-world chains is of considerable relevance. When blockchain security workers come up with a set of smart contract-related theories or develop a series of related algorithms, they can experiment directly on GSVDMajor to validate their theories and algorithms. In other words, researchers can use the Minor dataset to validate the performance of their vulnerability detection algorithms or tools, but when it comes to big data observation type of research other than pure vulnerability detection techniques, they need more data to support their theories and algorithms based on big data in a broad sense, the Major dataset is undoubtedly the best option.

Like the Minor dataset, GSVDMajor also has two subsets. In Section 3.1 we introduced the filtering and classification of the raw data, we ended up with 10,600 Polygon contracts and 41,602 BSC contracts after processing. We visualize the classification criterion as a reference and Figure 5 shows the details of this dataset, with Solidity versions corresponding to C4, C5, C6, C7 and C8 in the classification of Section 3.1, where CU has been cleaned up at the time of classification.

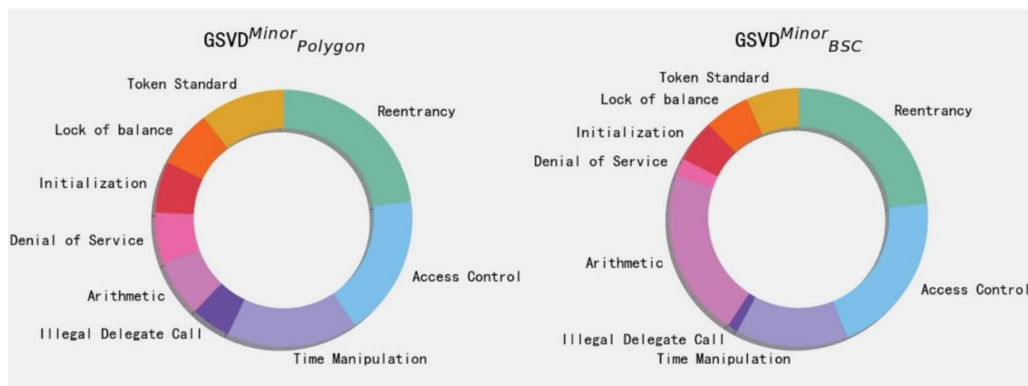

Figure 5. Number and solidity version distribution of $GSVD^{Major}$

4. EXPERIMENT AND RESULT ANALYSIS

4.1. Automatic Scripts and Detection Tools

Before starting the testing experiment, a brief introduction of the three detection tools we have chosen is given.

Slither: A smart contract vulnerability detection tool using static analysis. It achieves the goal of mining smart contract vulnerabilities by transforming Solidity contracts into an intermediate expression called SlithIR, and then performing programmatic analysis means such as data flow analysis or taint tracking. Developed by Trail of Bits.

Mythril: A security analysis tool using symbolic execution. It compiles and runs the contract while performing operations such as taint analysis and control flow analysis on the EVM bytecode to detect possible vulnerabilities. Developed by ConsenSys.

Manticore: A smart contract vulnerability detection tool using symbolic execution. It compiles and runs contracts, identifies constraints on program inputs and outputs, searches and analyses program execution path, detects contract vulnerabilities. Also developed by Trail of Bits.

Since all three tools selected support command line execution (CLI), we wrote automated execution scripts to perform batch processing of contracts, logging of terminal execution results and batch storage management. In addition, several statistical analysis scripts have been written to complement the analysis of the test results. These scripts are integrated and can be executed by simple terminal commands after modifying the input and output addresses, integrating a variety of contract detection, report generation, data statistics and other functions. Because each tool has different requirements for the system environment, in order to achieve the purpose of controlling the variables, we created three Conda virtual environments in the same device to construct the running environments of different tools and conducted test experiments in turn. After completing this work, the performance of different tools could be compared and analysed under the same hardware conditions.

4.2. Results and Data Analysis

4.2.1. Detection Result and Analysis of Minor Dataset

Table 3 shows the preliminary results of $GSVD^{Minor}_{Polygon}$ and $GSVD^{Minor}_{BSC}$ detection, we found that these three tools cannot detect all nine kinds of vulnerabilities. For example, Slither cannot detect Arithmetic, Manticore and Mythril cannot detect Denial of service, Lock of balance and Token standard these three kinds of vulnerabilities, which shows that the mainstream open-source vulnerability detection tools can only focus on some kinds of vulnerabilities because of their practical application scenarios and underlying technology, which leads to their inability to detect other kinds of vulnerabilities.

Table 3. Labeled vulnerabilities detected in $GSVD^{Minor}$.

Categories	Labelled vulnerabilities detected in $GSVD^{Minor}{}_{Polygon}$				Labelled vulnerabilities detected in $GSVD^{Minor}{}_{BSC}$			
	slither	manticore	mythril	Total	slither	manticore	mythril	Total
Reentrancy	25/32 78%	10/32 31%	22/32 69%	30/32 94%	43/53 81%	19/53 36%	38/53 72%	47/53 89%
Access control	7/25 28%	8/25 32%	10/25 40%	12/25 48%	15/47 32%	6/47 13%	18/47 38%	20/47 43%
Time manipulate	10/23 43%	2/23 9%	3/23 13%	11/23 48%	12/32 38%	2/32 6%	11/32 34%	14/32 44%
Illegal delegation	3/7 43%	2/7 29%	3/7 43%	4/7 57%	2/3 67%	1/3 33%	1/3 33%	2/3 67%
Arithmetic	0/10 0	3/10 30%	8/10 80%	8/10 80%	0/48 0	10/48 21%	40/48 83%	40/54 83%
Denial of service	1/9 11%	0/9 0	0/9 0	1/9 11%	2/5 40%	0/5 0	0/5 0	2/5 40%
Trap of initialize	3/9 33%	1/9 11%	2/9 22%	3/9 33%	5/12 42%	1/12 8%	1/12 8%	5/12 42%
Lock of balance	5/10 50%	0/10 0	0/10 0	5/10 50%	9/13 69%	0/13 0	0/13 0	9/13 69%
Token standard	4/15 27%	0/15 0	0/15 0	4/15 27%	7/15 47%	0/15 0	0/15 0	7/15 47%
Total	61/140 45%	26/140 19%	48/140 34%	79/140 57%	95/228 42%	39/228 17%	109/228 48%	146/228 64%

In terms of a single statistic, the Slither has the highest accuracy on the Polygon dataset, which has a much smaller number of contracts, at 45%, while Mythril has the highest accuracy on the BSC dataset, which has a much larger amount of data, at 48%, but Manticore has poor results on both datasets, with a detection rate of less than 20% in both cases. After reviewing the contracts and detection results, we totalled each tool's detections for each type of vulnerability and the specific data is filled in the far-right column of Table 3. From these data, we can see that after combining the three tools for contract detection, the combined detection has a detection rate of more than 50% for all four vulnerabilities: Reentrancy, Illegal delegate calls, Arithmetic and Lock of balance, some rate even reached more than 80% among them. Besides, the combined results for the nine vulnerabilities showed that the combined use of the three tools achieved an accuracy rate of 57% on $GSVD^{Minor}{}_{Polygon}$ and 64% on $GSVD^{Minor}{}_{BSC}$.

We found in Section 3.2 when building the Minor dataset that Reentrancy is the most numerous of the vulnerabilities contained in the real-world contract and in the process of using tools to detect the Minor dataset, we found that all three tools have the ability to detect Reentrancy with a much higher accuracy rate than the other vulnerabilities, with Slither and Mythril having an accuracy rate of 70%-80%. It is easy to see that in the real world, Reentrancy is extremely easy to occur and the irregular syntax habits of contract authors and conflicts within common contract functions can be the cause of this phenomenon. Because this type of vulnerability is basically only found at the Solidity level, hackers don`t need a high level of technical skills to find this vulnerability in the code to launch a re-entrancy attack, which is mainly related to functions such as money transfers and once successful, hackers can gain huge profits. From another perspective, vulnerability detection tools can also easily detect possible Reentrancy in contracts. Although the combined detection rates of all three tools are relatively high, it should be noted that several vulnerabilities are supported by the combined detection rates of a single tool. For example, combined detection rate of Arithmetic is more than 80% on both datasets but Slither has a detection rate of 0, Manticore has a detection rate of less than 30% and only Mythril has a

detection rate of more than 80%. More similar situations arise in the detection of vulnerabilities such as Denial of service, Lock of balance and Token standard.

Table 4. All vulnerabilities detected in $GSVD^{Minor}$

Categories	All vulnerabilities detected in $GSVD^{Minor}_{Polygon}$				All vulnerabilities detected in $GSVD^{Minor}_{BSC}$			
	slither	manticore	mythril	Total	slither	manticore	mythril	Total
Reentrancy	41	20	31	92	58	32	47	137
Access control	16	8	16	40	21	8	26	55
Time manipulate	18	4	4	26	13	5	11	29
Illegal delegation	12	2	4	18	2	1	2	5
Arithmetic	0	5	10	18	0	23	56	79
Denial of service	1	0	0	1	5	0	0	5
Trap of initialize	3	1	2	6	8	1	1	10
Lock of balance	5	0	0	5	9	0	0	9
Token standard	7	0	0	7	9	0	0	9
Others	133	47	86	266	207	73	135	415
Total	235	87	153	477	332	143	278	753

In addition to the vulnerabilities we labelled in advance, these detection tools also detected some unlabelled vulnerabilities in the Minor dataset and we made a count of the number of vulnerabilities detected, which is presented in Table 4. Out of the 9 different vulnerabilities detected, 3 tools showed some degree of false detection. Besides, there were a large number of other vulnerabilities detected by these three tools, which we labelled as Others and recorded in Table 4. Because of the presence of these two phenomena, there are some cells in Table 4 where the data recorded also exceeds the number of vulnerabilities labelled.

Because the performance of the device running the inspection tools is not good, the execution time is much higher than the time that each tool should theoretically achieve. However, we can still use the control variates to find out how the execution comparisons between the three tools. Since we used automatic batch-processing scripts for the experiment, we only need to record the starting and ending time of the script. Table 5 shows information about the specific processing time of each tool. From Table 5 we can see that Slither is the tool with the shortest execution time, taking only 22 minutes to complete the detection of 153 smart contracts in two subsets of $GSVD^{Minor}$, while Manticore and Mythril take significantly longer, with Manticore, which uses dynamic symbolic execution technology, taking the longest time, nearly 42 hours, to complete. However, combined with Table 3 above, we can see that Manticore, which takes the longest time, has the lowest detection rate for vulnerabilities.

The reason for this big difference in processing time is that Slither is a static analysis tool that only needs to check and analyse the code to reach a conclusion, while the other two tools need to compile and simulate the execution of the contract, as well as testing the contract, so it is clear that compiling the contract is a rather time-consuming process.

Table 5. Execution Time of Each Tool.

Number	Tools	Avg Exec Time	Total Exec Time
1	Slither	0 m 9 s	0 h 22 m 31 s
2	Manticore	16 m 24 s	41 h 54 m 2 s
3	Mythril	1 m 42 s	4h 15 m 6 s

4.2.2. Data Analysis of Major Dataset

Since the performance of the devices used for the experiments is not top-notch, we only performed some statistical and analytical work on the Major dataset. In other words, we hope to conduct not only contract vulnerability-related experiments on the Major dataset, but also a variety of diversified big data analysis and real phenomenon feedback. Figure 6 shows the distribution of contract versions of Polygon and BSC subset from the Major dataset. We can see that just over a year after its announcement, Solidity v0.8.0 is already widely used in real-world contracts on real-world platforms. From the composition of GSVD, v0.8.0 occupies an almost 80% share of the absolute dominance on Polygon, while on BSC, v0.8.0 also has a 51% share. In addition, v0.6.0 has more than 3 times the share of v0.7.0. This phenomenon also shows that the new features in v0.7.0 are not easy to use and not popular.

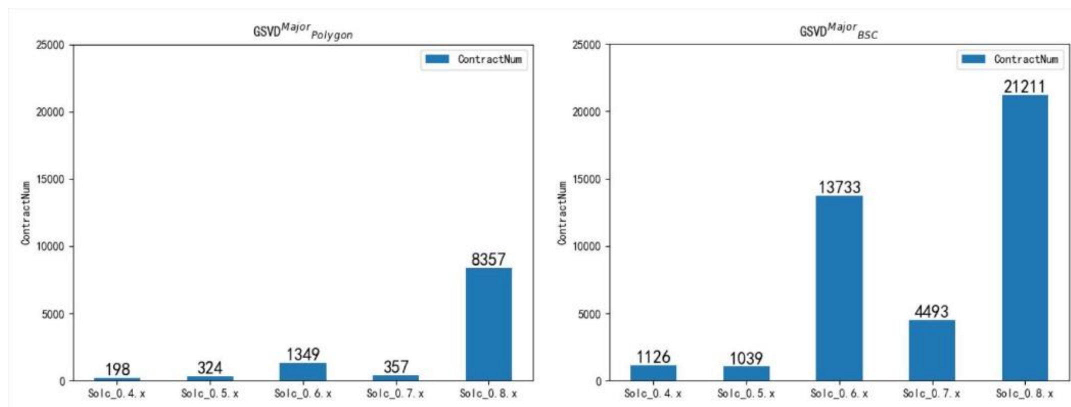

Figure 6. Distribution of solidity versions in GSVD^{Major}

In addition, we also did preliminary statistics and analysis on the speed of smart contract updates, the number of codes, and other information during the collection process. Figure 7 shows the average daily number of smart contract deployments per month on BSC and Polygon starting in November 2021. The line graph shows that the average daily number of contracts deployed on both platforms is growing at an extremely fast rate over time, reflecting the fact that as decentralized finance (DeFi) and decentralized applications (DApp) are booming, more users are using smart contracts and more professional workers are joining in the development and O&M of smart contract.

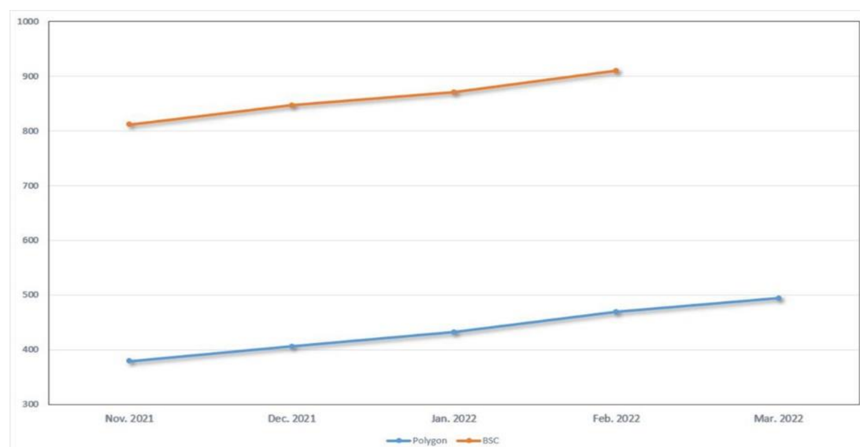

Figure 7. Average Daily Number of Smart Contract Deployed on BSC and Polygon Per Month

5. CONCLUSIONS

In this paper, we collected and built a new smart contract dataset GSVD, containing 2 categories, 2 smart contract platforms and 4 subsets, which currently has over 52,000 different real-world smart contracts after the first phase of work. Based on these contracts, we filtered and classified them into Minor and Major dataset, focusing on vulnerability detection and real smart contract environment of the moment respectively. After completing the build, we performed vulnerability detection on the Minor dataset and statistical analysis on the Major dataset, which yielded a lot of useful vulnerability data and discovered many smart contract-related phenomena worthy of deeper investigation. Because of the huge number of contracts and lines of code in the dataset, we also wrote multiple automated execution scripts, integrated them and worked with the entire dataset. The dataset, supporting scripting tools and experimental results presented in this paper are of considerable value to the development of work related to smart contract vulnerabilities and platforms. We will continue to work on the basis of the existing results, update and expand the dataset, push the experiment to a larger space and provide high-value content for related workers.

6. FUTURE WORKS

Although the number of GSVD data is large enough and the coverage is relatively wide, it is worth noting that the variety of vulnerabilities in the Minor dataset is not enough and the distribution is not uniform. Besides, although the Major dataset is large enough, the update speed of smart contracts in the real world is extremely fast and therefore, the contracts in the Major dataset are not able to maintain timeliness. In the construction and experimentation of GSVD, we ensured the replicability of GSVD by streamlining the workflow and writing automatic execution scripts. In the first phase we collected contract content for the BSC and Polygon over a period of time. With everything going well, we plan to continuously update the contracts for both platforms in Phase 2, expanding the Minor and Major datasets after screening and cleaning. In addition to this, we have plans to expand other blockchain platforms that support smart contracts and use EVM in phase 2.

In addition to plans to expand the size of the dataset, we also have plans to optimize its use, such as providing more granular filtering, publishing to the website and providing conditional query services, so that users do not need to download the full dataset to get the part of the contract they want. However, when users use the query and download service, they can only query by objective conditions such as balance, number of transactions, Solidity version number, etc. When it comes to the contract content, there is nothing they can do. Therefore, in addition to the expansion and optimization in the second phase, we are going to do further research on user notice through other existing tools or develop our own tools, so that we can display the contract content and make the dataset easier to use and provide the users with a better experience.

Besides, we noticed that some innovative modern methods are emerging and beat traditional detection methods in some aspects. For example, N Ashizawa *et al.*[22] tried to introduce machine learning to detect smart contract vulnerability, L Zhang *et al.* [23]proposed hybrid methods combing with word embedding and different deep learning methods. We also plan to expand our detecting module with newly-released tools which applying deep learning methods.

In addition to theoretical and software work, the subsequent phase of work cannot be carried out without better equipment. We are already planning to use a multi-core processor, large memory cloud platform as the execution platform for experiments and datasets, which will greatly improve the speed of dataset construction and experiment execution.

ACKNOWLEDGEMENTS

The authors would like to thank everyone, just everyone! Plus, this work is supported by National Key RD Program of China (No.2019YFB2101701).

REFERENCES

[1] Buterin, V. (2014). A next-generation smart contract and decentralized application platform. white paper, 3(37), 2-1.

[2] Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., & Vechev, M. (2018, October). Securify: Practical security analysis of smart contracts. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (pp. 67-82).

[3] Luu, L., Chu, D. H., Olickel, H., Saxena, P., & Hobor, A. (2016, October). Making smart contracts smarter. In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security (pp. 254-269).

[4] Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., & Hobor, A. (2018, December). Finding the greedy, prodigal, and suicidal contracts at scale. In Proceedings of the 34th annual computer security applications conference (pp. 653-663).

[5] Szabo, N. (1996). Smart contracts: building blocks for digital markets. EXTROPY: The Journal of Transhumanist Thought,(16), 18(2), 28.

[6] Wikipedia contributors. (2023b, January 9). The DAO (organization). Wikipedia. https://en.wikipedia.org/w/index.php?title=The_DAO_(organization).

[7] NCC Group. (2018, July 28). Decentralized Application Security Project Top 10 of 2018. NCC Group. https://dasp.co/index.html.

[8] Vivar, A. L., Orozco, A. L. S., & Villalba, L. J. G. (2021). A security framework for Ethereum smart contracts. Computer Communications, 172, 119-129.

[9] Azzopardi, S., Ellul, J., & Pace, G. J. (2018, November). Monitoring smart contracts: Contractlarva and open challenges beyond. In International Conference on Runtime Verification (pp. 113-137). Springer, Cham.

[10] Zhou, Y., Kumar, D., Bakshi, S., Mason, J., Miller, A., & Bailey, M. (2018). Erays: reverse engineering ethereum's opaque smart contracts. In 27th USENIX Security Symposium (USENIX Security 18) (pp. 1371-1385).

[11] Grishchenko, I., Maffei, M., & Schneidewind, C. (2018, April). A semantic framework for the security analysis of ethereum smart contracts. In International Conference on Principles of Security and Trust (pp. 243-269). Springer, Cham.

[12] Albert, E., Gordillo, P., Livshits, B., Rubio, A., & Sergey, I. (2018, October). Ethir: A framework for high-level analysis of ethereum bytecode. In International symposium on automated technology for verification and analysis (pp. 513-520). Springer, Cham.

[13] Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., & Smaragdakis, Y. (2018). Madmax: Surviving out-of-gas conditions in ethereum smart contracts. Proceedings of the ACM on Programming Languages, 2(OOPSLA), 1-27.

[14] Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., ... & Dinaburg, A. (2019, November). Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 1186-1189). IEEE.

[15] Mueller, B. (2018). Smashing ethereum smart contracts for fun and real profit. HITB SECCONF Amsterdam, 9, 54.

[16] Torres, C. F., Schütte, J., & State, R. (2018, December). Osiris: Hunting for integer bugs in ethereum smart contracts. In Proceedings of the 34th Annual Computer Security Applications Conference (pp. 664-676).

[17] Feist, J., Grieco, G., & Groce, A. (2019, May). Slither: a static analysis framework for smart contracts. In 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB) (pp. 8-15). IEEE.

[18] Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., & Alexandrov, Y. (2018, May). Smartcheck: Static analysis of ethereum smart contracts. In Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (pp. 9-16).

[19] Hegedűs, P. (2018, May). Towards analyzing the complexity landscape of solidity based ethereum smart contracts. In Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (pp. 35-39).
[20] Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., ... & Scholz, B. (2018). Vandal: A scalable security analysis framework for smart contracts. arXiv preprint arXiv:1809.03981.
[21] Google Cloud. (2011). BigQuery: Cloud Data Warehouse. Google Cloud. https://cloud.google.com/bigquery.
[22] Ashizawa, N., Yanai, N., Cruz, J. P., & Okamura, S. (2021, May). Eth2Vec: learning contract-wide code representations for vulnerability detection on ethereum smart contracts. In Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure (pp. 47-59).
[23] Zhang, L., Chen, W., Wang, W., Jin, Z., Zhao, C., Cai, Z., & Chen, H. (2022). Cbgru: A detection method of smart contract vulnerability based on a hybrid model. Sensors, 22(9), 3577.

AUTHORS

Ziniu Shen (1998-). Male. Born in Taizhou, China. College of Computer Science, Nanjing University of Posts and Telecommunications. Master. Focus on Blockchain, Smart Contract Vulnerabilities, Deep Learning.

Yunfang Chen (1976-). Male. Born in Zhenjiang, China. College of Computer Science, Nanjing University of Posts and Telecommunications. Professor, Ph.D. Focus on Computer Network, Blockchain, Artificial Intelligence.

Wei Zhang (1973-). Male. Born in Taizhou, China. College of Computer Science, Nanjing University of Posts and Telecommunications. Professor, Ph.D. Focus on Cyber Security, Blockchain, Artificial Intelligence.

© 2023 By AIRCC Publishing Corporation. This article is published under the Creative Commons Attribution (CC BY) license.