Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Vulpedia: Detecting vulnerable ethereum smart contracts via abstracted vulnerability signatures[☆]

Jiaming Ye[a,*], Mingliang Ma[b], Yun Lin[c], Lei Ma[d], Yinxing Xue[e], Jianjun Zhao[a]

[a] Kyushu University, Japan
[b] Nanjing University, China
[c] National University of Singapore, Singapore
[d] University of Alberta, Canada
[e] University of Science and Technology of China, China

ARTICLE INFO

Article history:
Received 5 August 2021
Received in revised form 28 April 2022
Accepted 13 June 2022
Available online 18 June 2022

Keywords:
Software analysis
Software clone analysis
Smart contract
Blockchain security
Software testing

ABSTRACT

Recent years have seen smart contracts are getting increasingly popular in building trustworthy decentralized applications. Previous research has proposed static and dynamic techniques to detect vulnerabilities in smart contracts. These tools check vulnerable contracts against several predefined rules. However, the emerging new vulnerable types and programming skills to prevent possible vulnerabilities emerging lead to a large number of false positive and false negative reports of tools. To address this, we propose Vulpedia, which mines expressive vulnerability signatures from contracts. Vulpedia is based on the relaxed assumption that the owner of contract is not malicious. Specifically, we extract structural program features from vulnerable and benign contracts as *vulnerability signatures*, and construct a systematic detection method based on detection rules composed of vulnerability signatures. Compared with the rules defined by state-of-the-arts, our approach can extract more expressive rules to achieve better completeness (i.e., detection recall) and soundness (i.e., precision). We further evaluate Vulpedia with four baselines (i.e., Slither, Securify, SmartCheck and Oyente) on the testing dataset consisting of 17,770 contracts. The experiment results show that Vulpedia achieves best performance of precision on 4 types of vulnerabilities and leading recall on 3 types of vulnerabilities meanwhile exhibiting the great efficiency performance.

1. Introduction

Powered by the Blockchain technique (Beck et al., 2017), smart contracts (Szabo, 1996) have attracted much attention and been applied in various industries, e.g., financial service, supply chains, smart traffic, and IoTs. Solidity is the most popular language for smart contract for its mature tool support and simplicity. However, the public has witnessed several severe security incidents, including the notorious DAO attack (Zhao et al., 2017) and Parity wallet hack (Parity Technologies, 2017). According to previous reports (Atzei et al., 2016; Adrian Manning, 2018), up to 16 types of security vulnerabilities were found in Solidity programs. These security issues undermine the confidence of people who have executed transactions via smart contracts and eventually affect the trust towards the Blockchain ecosystem.

Witnessing the severity and urgency of this problem, researchers and security practitioners have made endeavors to develop automated security scanners (trailofbits, 2019c; melonproject, 2019; Kalra et al., 2018; Tsankov et al., 2018; Tikhomirov et al., 2018). Existing state-of-the-art scanners usually adopt the rule-based methods for vulnerability detection. Slither (trailofbits, 2019c) supports 39 hard-coded static rules; Securify (Tsankov et al., 2018) supports 15 rules for verifying the extracted path constraints from the contract with the SMT solvers (Unman, 1989); Oyente (melonproject, 2019) supports 8 rules for generating assertions for verifying the vulnerabilities. Each rule represents a pattern of vulnerable contract, which warns the programmers to avoid potential risks before deploying the contracts.

Although experiments have demonstrated their effectiveness, it is notable that rules behind these scanners are manually crafted by human experts. *The manually predefined rules can be obsolete*, because ❶ previously unseen vulnerable code may be introduced, which cannot be captured by the hard-coded rules, and ❷ new defense mechanisms (i.e., programming skills to prevent bugs) may have successfully mitigated the vulnerabilities, but the code may still match the predefined vulnerable pattern or rules. Therefore, most updated rules should be learned to distinguish vulnerable contracts from robust ones.

[☆] Editor: Alexander Chatzigeorgiou.
[*] Corresponding author.
E-mail address: yejjmg@gmail.com (J. Ye).

https://doi.org/10.1016/j.jss.2022.111410
0164-1212/© 2022 Elsevier Inc. All rights reserved.

In this work, we alleviate the incompleteness of detection rules by combining vulnerability signatures abstracted from both vulnerable and benign contracts (i.e., vulnerable signature and benign signature). The vulnerable signature is designed for matching commonalities of a particular vulnerability. Comparatively, the benign signature is abstracted from falsely reported contracts in order to reduce false alarms. For each vulnerability, we adopt vulnerable and benign signatures to synthesize detection rules of VULPEDIA. Note that VULPEDIA is built upon the relaxed assumption that the owner of contract is not malicious. Detecting malicious contract (e.g., contract with backdoors, exploit code) is different to the vulnerability detection (i.e., the target of VULPEDIA). Based on this assumption, the operations related to the contract owner are all deemed as vulnerability defense behaviors. Compared with previous work, the synthesized rules are more updated and expressive than the predefined rules in the state-of-the-art vulnerability scanners, capturing a lot of unseen patterns in practice.

In our implementation, we first collect truly and falsely reported contracts by applying three state-of-the-art vulnerability scanners (i.e., SLITHER trailofbits, 2019c, OYENTE Luu et al., 2016, and SECURIFY Tsankov et al., 2018) and manually evaluate their correctness. Based on the results, analyzing truly reported vulnerable contracts allows us to capture salient program signatures responsible for vulnerable contracts. In contrast, analyzing falsely reported vulnerable contracts allows us to capture signatures noticeable for avoiding false alarms. Next, we categorize the contracts by their vulnerability types (e.g., Reentrancy, Unchecked Low-level-call, etc.) and alarm types (i.e., true or false alarm). For each category, we cluster the contracts based on their tree edit distance (Pawlik and Augsten, 2015) and then extract program feature commonalities from the PDGs (program dependency graph) of each cluster to summarize vulnerability signatures. Finally, we abstract 4 vulnerable signatures and 6 benign signatures. They are integrated as 4 detection rules regarding 4 vulnerabilities (i.e., Reentrancy, SelfDestruct, Tx-origin, and Unexpected-Revert).

We conduct our signature abstraction on a set of 76,354 smart contracts and evaluations on a set of 17,770 contracts, respectively. The evaluation results show that, compared with the state-of-the-art vulnerability scanners (i.e., SLITHER, OYENTE, SMARTCHECK and SECURIFY), our approach achieves outstanding accuracy on 4 vulnerabilities and leading recall on 3 vulnerabilities. We make our tool, VULPEDIA, available at Anonymity (2019).

To summarize, we make the following contributions:

(1) We propose an approach to abstract vulnerability signatures and compose detection rules to report vulnerability. The learned rules are more expressive than rules of the state-of-the-art scanners, reporting vulnerabilities with better completeness and soundness
(2) On the 17,770 contracts crawled from Google, VULPEDIA yields the best precision on 4 vulnerabilities and leading recall on 3 ones, in comparison with the other state-of-the-art scanners.
(3) Experiments show that VULPEDIA is efficient in vulnerability detection. The detection speed of VULPEDIA on 17,770 contracts is far faster than OYENTE and SECURIFY.

This work is organized as: In Section 2, we first introduce the different types of vulnerabilities we address in our study, and explain why the state-of-the-art tools fail. In Section 3 we illustrate the basic steps of our proposed tool, namely VULPEDIA. In Section 4, we conduct an empirical study and introduce our method of signature abstraction. We also elaborate the effectiveness of signatures with examples. In Section 5, we compare VULPEDIA with the other state-of-arts using 17,770 real-world contracts deployed on Ethereum. Section 7 briefly introduces the related work and Section 8 concludes our study.

2. Background and motivation

In this section, we explain the 4 vulnerability types (i.e., `Reentrancy`, `The abuse of tx.origin`, `Unexpected Revert` and `Self-destruct Abusing`.) targeted by our study. The 4 vulnerabilities deeply threats the safety of transactions of smart contracts. For example, the `Reentrancy` caused the DAO attack in 2016 and resulted in hundreds of millions dollars losses; The `tx.origin` and `Unexpected Revert` vulnerability are listed in the Decentralized Application Security Project (DASP) (NCC Group, 2019); The `Self-destruct Abusing` vulnerability often appears with the use of `selfdestruct` instruction in Solidity, and is prone to being exploited if it is not well protected. We also show a real-world case, which is not well-handled by the state-of-the-art scanners, to motivate this work.

2.1. Vulnerability types

(1) *Reentrancy (RE)* As the most famous Ethereum vulnerability, reentrancy recursively triggers the fall-back function (ethereum, 2017) to steal money from the victim's balance or deplete the gas of the victim. Reentrancy occurs when external callers manage to invoke the callee contract before the execution of the original call is finished, and it was mostly caused by the improper usages of the function `withdraw()` and `call.value(amount)()`. It was also reported in Atzei et al. (2016).
(2) *The Abuse of `tx.origin` (TX)* When the visibility is improperly set for some key functions (e.g., some sensitive functions with `public` modifier), the extra permission control then matters. However, issues can arise when contracts use the deprecated `tx.origin` (especially, `tx.origin==owner`) to validate callers for permission control. It is relevant to the *access control* vulnerability in NCC Group (2019). When a user *U* calls a malicious contract *A*, who intends to forward call to contract *B*. Contract *B* relies on vulnerable identity check (e.g., `require(tx.origin == owner)` to filter malicious access. Since `tx.orign` returns the address of *U* (i.e., the address of `owner`), malicious contract *A* successfully poses as *U*.
(3) *Unexpected Revert (UR)* In a smart contract, some operations may unfortunately fail. This can lead to two main impact: (1) the gas (i.e., the fee of executing an operation in Ethereum platform) of the transaction is wasted; (2) the transaction will be reverted, i.e., the denial of service (DoS). The denial of service attack is also termed "DoS with revert" in ConsenSys Diligence (2019). The attacker could deliberately make some operations fail for the purpose DoS. For example, some functions recursively send ethers to an array of users. If one of these calls fails, the whole transaction will be reverted. An attacker can deliberately fail this transaction to achieve a denial-of-service attack.
(4) *Self-destruct Abusing (SD)* This vulnerability allows the attackers to forcibly send Ether without triggering its fall-back function. Normally, the contracts place important logic in the fall-back function or making calculations based on a contract's balance. However, this could be bypassed via the `self-destruct` contract method that allows a user to specify a beneficiary to send any excess ether (ConsenSys Diligence, 2019). That is, a vulnerable contract is prone to being exploited to transfer all money to attacker' account meanwhile shut down the service.

```
1 function withdraw() {
2     require(msg.sender == owner);
3     uint256 amount = balances[msg.sender];
4     require(msg.sender.call.value(amount)());
5     balances[msg.sender] = 0;
6 }
```

Fig. 1. An example of a non-vulnerable code. This is misreported as vulnerability by Slither and Oyente.

2.2. Motivating examples

Fig. 1 is mistakenly alarmed by Slither and Oyente. The function `withdraw` intends to send ethers to the `msg.sender`. It first verifies the identity of caller at line 2. Then, the function reads the amount of current balance of the caller at line 3 and sends ethers to the caller by using a Solidity call `.call.value()()`. Finally, the function updates the balance of caller at line 5.

The reason of the false alarm of Slither is due to that Slither detects reentrancy with the following rule:

$$\mathit{DataDep}(\_, \mathit{var}_g) \succ \mathit{Call}(\_, \mathit{var}_g) \succ \mathit{DataDep}(\_, \mathit{var}_g) \Rightarrow \text{reentrancy} \tag{1}$$

In Rule 1, $\mathit{DataDep}(\_, \mathit{var\_g})$ denotes write and read operations to variables; var_g denotes a certain public global variable; $\succ$ denotes the execution order in the control flow; $\mathit{Call}(\_, \mathit{var}_g)$ denotes function call operations. This rule describes a common pattern for Reentrancy vulnerability. Fig. 1 shows a typical example. var_g is usually a balance account (e.g., `balances[msg.sender]`, line 3 in Fig. 1). An attacker just needs to create a fallback function that calls `withdraw()`. Once `msg.sender.call.value(amount)()` is executed and transfers the funds, the attacker's fallback function (ethereum, 2017) will be triggered and call `withdraw()` (line 1) again. By this means, the attacker can transfer more funds before `balances[msg.sender]` is reduced to 0. This continues until there is no *ether* remaining, or execution reaches the maximum stack size.

However, the pattern in Rule 1 is usually an over-estimation for real Reentrancy vulnerability. In fact, the example in Fig. 1 is a counter-example because the function `withdraw()` is protected by an identity check at line 2. This statement specifies a precondition for running the `withdraw()` function. Once the precondition is not satisfied, the execution will be aborted. In Fig. 1, the identity check indicates that the contract calling this `withdraw()` function is limited to its `owner` (i.e., the creator of the contract).

The reason of the false alarm of Oyente is due to that Oyente detects reentrancy with the following rule:

$$(\mathit{DataDep}(\_, \mathit{var}_g) \wedge (\mathit{gas}_{trans} > 2300) \wedge (\mathit{amt}_{bal} > \mathit{amt}_{trans})) \succ \mathit{Call}(\_, \mathit{var}_g) \Rightarrow \text{reentrancy} \tag{2}$$

In Rule 2, Oyente requires the gas expense less than a certain value. In Solidity programs, each transaction requires an amount of gas to complete in the runtime. $\mathit{gas}_{trans} > 2300$ means the gas used for transaction must be larger than 2300 (2300 is the least gas expense to conduct a transaction call). $\mathit{amt}_{bal} > \mathit{amt}_{trans}$ means the balance amount must be larger than transfer amount. Finally, the rule of Oyente requires call to external functions by *Call* meanwhile send money. Comparing with Rule 1 (defined by Slither), Oyente has more constraints for gas and balance value. Similar to the rule of Slither in Rule 1, Rule 2 also overestimates the condition where Reentrancy attack can happen. With the protection by the identity check (i.e., line 2 in Fig. 1), the execution of function calls conforms to the defined runtime conditions but is already free from the Reentrancy attack.

How Vulpedia can address this issue: In contrast, Vulpedia is equipped with detection rule composed of a vulnerable signature as shown in Rule 3 (i.e., the signature indicating potential vulnerability) and a benign signature as shown in Rule 4 (i.e., the signature indicating potential code behaviors defending or fixing vulnerability).

$$\mathit{DataDep}(\_, \mathit{var}_g) \succ \mathit{Call}(\_, \mathit{var}_g) \Rightarrow \mathit{reentrancy} \tag{3}$$

$$\mathit{ControlDep}(\mathit{msg.sender}, \_) \succ \mathit{DataDep}(\_, \mathit{var}_g) \succ \mathit{Call}(\_, \mathit{var}_g) \Rightarrow \mathit{reentrancy} \tag{4}$$

For the vulnerable signature, Vulpedia adopts valuable experience from Slither and Oyente and detects Reentrancy by matching data dependency of variables followed by call operations. As for the benign signature, Vulpedia eliminates false reports by filtering out functions which contain control dependency on msg.sender. For example, in Fig. 1 the code at lines 2 checks if the msg.sender equals the address of owner, and the function is not considered as vulnerability by Vulpedia.

3. Overview

Fig. 2 shows the workflow of abstracting vulnerability signatures for Vulpedia. The workflow can be roughly grouped into four steps: (1) The pre-detection of existing tools; (2) Vulnerability report inspection; (3) AST clustering and signature abstraction; (4) Rule composition. Note that manual efforts are involved in step 2 and step 4.

In the first two steps, we systematically evaluate (1) how accurately state-of-the-art tools can report the vulnerable smart contracts and (2) under what condition can those tools be ineffective. We collect the reports of the state-of-the-art tools on a training dataset of 76,354 contracts. Then, we employ three experienced smart contract developers to manually confirm the reports of the tools, and categorize them into two groups: truly alarmed vulnerable contracts and falsely alarmed vulnerable contracts.

In the last two steps, we first calculate the tree edit distance based on the ASTs of contracts in a particular vulnerability type and cluster the contracts of the type by defining the contract similarity. Next, we abstract common nodes from the PDGs (program dependency graph) of each cluster to summarize signatures (e.g., as shown in Fig. 4). From truly vulnerable contracts, we summarize vulnerable signatures. In contrast, from falsely alarmed vulnerable contracts, we summarize benign signatures. Finally, we manually integrate the vulnerable signatures and benign ones into vulnerability detection rules.

After we equip our Vulpedia detector with the composed rules, the detector takes unknown contracts as inputs and generate vulnerability reports based on the signatures. The figure of the architecture is shown in Fig. 3. Specifically, the detector first conduct preprocessing on the input smart contract code. The detector extracts normalized AST from the contract. Based on this normalized AST, the detector conducts a PDG extraction. Meanwhile, the detector extracts existing signatures from the

Table 1
The state-of-art tools for Solidity analysis.

Tool name	Method	Technique	Open source	Implementation	Adopted in experiment
Mythril (ConsenSys, 2019a)	Dynamic	Constraint Solving	●	Python	○
MythX (ConsenSys, 2019b)	Dynamic	Constraint Solving	○	N.A.	○
Slither (trailofbits, 2019c)	Static	CFG Analysis	●	Python	●
Echidna (trailofbits, 2019a)	Dynamic	Fuzzy Testing	●	Haskell	○
Manticore (trailofbits, 2019b)	Dynamic	Testing	●	Python	○
Oyente (Luu et al., 2016)	Dynamic	Constraint Solving	●	Python	●
SmartCheck (Tikhomirov et al., 2018)	Static	AST Analysis	●	Java	●
Octopus (2019)	Static	Reverse Analysis	●	Python	○
Zeus (Kalra et al., 2018)	Static	Formal Verification	○	N.A.	○
ContractFuzzer (Jiang et al., 2018)	Dynamic	Fuzzy Testing	●	Go	○

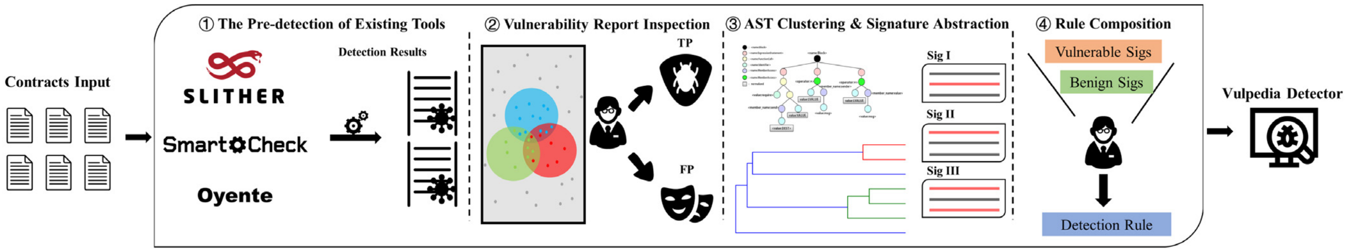


Fig. 2. The workflow of extracting vulnerability signatures of Vulpedia.

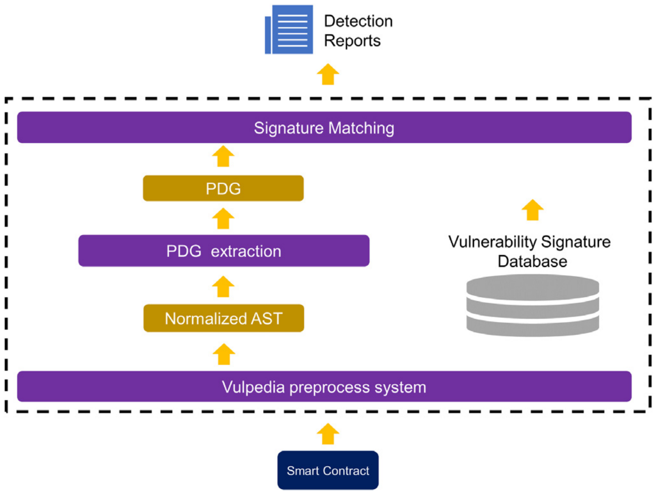


Fig. 3. The architecture diagram of Vulpedia.

vulnerability signature database. Lastly, the detector produces detection reports based on the comparison results. That is, if the PDG matches vulnerable signatures but not matched with benign signatures, the contract will be deemed as a vulnerable contract; otherwise, the detector will produce a non-vulnerable report.

4. Empirical study of signature abstraction

In this section, we first illustrate how we empirically collect contracts in this study. We report how we select the vulnerability scanners and how we construct a contract dataset. Next, we introduce our method of ❶ clustering similar contracts by comparing tree edit distance, ❷ abstracting commonalities from PDGs of clusters as signatures and ❸ detection rules composition based on the abstracted signatures. Finally, we elaborate the signatures with examples to evidence the representativeness of them.

4.1. Selected scanners and dataset

4.1.1. Choice of scanners and vulnerability types

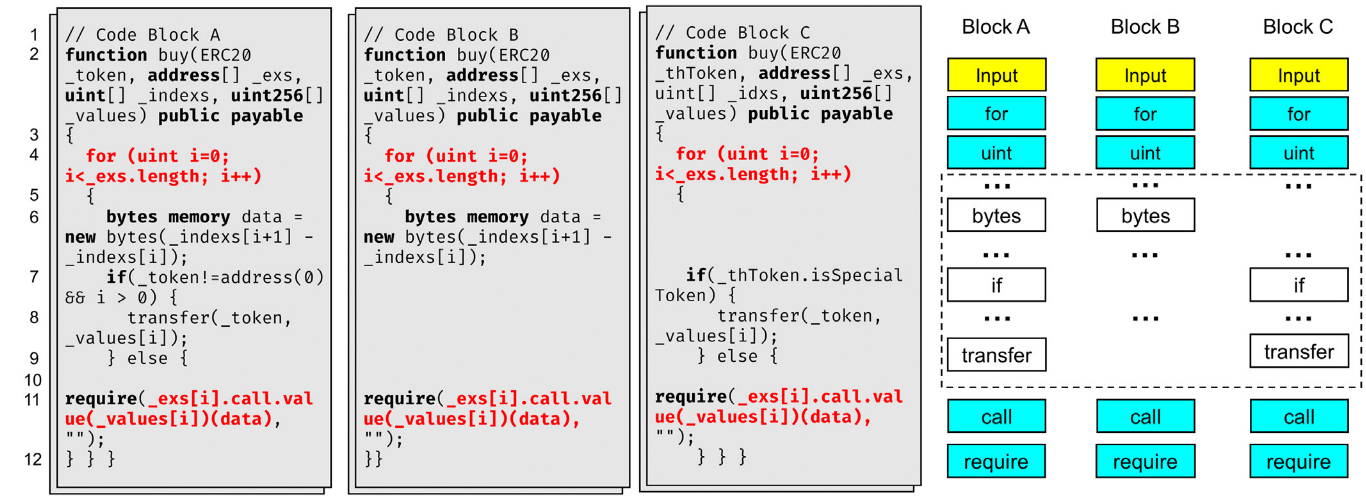
Overall, we select vulnerability scanners based on how practical they can be used in real-world scenarios. We investigate a list of static analyzers, including Slither (trailofbits, 2019c), Oyente (Luu et al., 2016), Zeus (Kalra et al., 2018) SmartCheck (Tikhomirov et al., 2018), and MythX (ConsenSys, 2019b). These tools utilize manually defined detection rules to detect vulnerabilities. The rules could match vulnerabilities in some cases but also generate much false reports. We also investigate dynamic detectors like Mythril (ConsenSys, 2019a), ContractFuzz (Jiang et al., 2018), Echidna (trailofbits, 2019a) and Manticore (trailofbits, 2019b). They exercise programs and check the runtime status of functions to find vulnerabilities. The dynamic analyzers often achieve high detection precision but suffer from limited scalability. Additionally, we investigate other analyzing tools (e.g., Solidity reverse engineering tool Octopus, 2019) to facilitate our exploiting contracts. A summary of the above tools can be found at Table 1. In our study, some tools are not selected because they are not open-sourced (Zeus Kalra et al., 2018, MythX ConsenSys, 2019b), not related to our task (Echidna trailofbits, 2019a, Octopus, 2019) and efficiency concerns (Mythril ConsenSys, 2019a, ContractFuzzer Jiang et al., 2018, Manticore trailofbits, 2019b).

Finally, we choose Slither v.0.4.0, Oyente v0.2.7 and SmartCheck v2.0 as our scanners.

4.1.2. Dataset for empirical study

We implement a web crawler to download Solidity files from accounts of Etherscan (Anon, 2019a), a famous third-party website on Ethereum block explorer. Etherscan provides APIs for downloading transaction information (e.g., transaction addresses, time).

We choose contracts deployed by Solidity 0.4.25 and 0.4.24. The reasons are two folds: (1) as listed in Table 2, Solidity 0.4 is the majority version among all versions and the 0.4.24 and 0.4.25 are the latest versions in Solidity 0.4; (2) The versions 0.4.24 and 0.4.25 are supported by most analyzers, so that they facilitate our study. Additionally, we find that the downloaded dataset has redundant contracts (contracts which share commonality with others). Regarding these redundant contracts, we remove contracts that are exactly same to others and contracts that are only



```
// Code Block A
function buy(ERC20
_token, address[] _exs,
uint[] _indexs, uint256[]
_values) public payable
{
  for (uint i=0;
i<_exs.length; i++)
  {
    bytes memory data =
new bytes(_indexs[i+1] -
_indexs[i]);
    if(_token!=address(0)
&& i > 0) {
      transfer(_token,
_values[i]);
    } else {

require(_exs[i].call.val
ue(_values[i])(data),
"");
} } }
```

```
// Code Block B
function buy(ERC20
_token, address[] _exs,
uint[] _indexs, uint256[]
_values) public payable
{
  for (uint i=0;
i<_exs.length; i++)
  {
    bytes memory data =
new bytes(_indexs[i+1] -
_indexs[i]);






require(_exs[i].call.val
ue(_values[i])(data),
"");
}}
```

```
// Code Block C
function buy(ERC20
_thToken, address[] _exs,
uint[] _idxs, uint256[]
_values) public payable
{
  for (uint i=0;
i<_exs.length; i++)
  {



    if(_thToken.isSpecial
Token) {
      transfer(_token,
_values[i]);
    } else {

require(_exs[i].call.val
ue(_values[i])(data),
"");
    } } }
```

Fig. 4. Three similar code blocks of Unexpected Revert that are found in real-world contracts. Based on their tree edit distance, we cluster them together and abstract a graph skeleton from their PDG. The yellow boxes denote function inputs, blue boxes denote common nodes on PDG and white boxes in dotted box represent different nodes. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 2
The percentages of adopted Solidity contracts versions. According to Tian et al. (2022).

Major version	# of smart contracts	Percentage
0.1	13	<0.1%
0.2	89	<0.1%
0.3	519	0.39%
0.4	71,350	**54.27%**
0.5	32,479	24.69%
0.6	22,171	16.85%
0.7	4200	3.19%
0.8	725	0.55%

Table 3
Number of collected contracts for each category.

Alarm type	RE	TX	UR	SD
True positive	46	38	421	3
False positive	720	179	2546	51

Table 4
The precision performance of three tools Slither, Oyente and SmartCheck on four vulnerabilities.

Vulnerability	Slither	Oyente	SmartCheck
RE	623 (3.53%)	143 (16.78%)	N.A.
TX	67 (28.35%)	N.A.	150 (12.66%)
UR	2678 (8.25%)	N.A.	289 (69.20%)
SD	54 (5.56%)	N.A.	N.A.

different in transfer address with others. Finally, we have 76,354 contracts for empirical study. Our crawler can be accessed at https://github.com/ToolmanInside/smart_contract_crawler.

Table 3 shows the number of contracts we collected in this study. Overall, among 76,354 contracts, the three tools report 508 true vulnerable contracts albeit 3496 false vulnerable ones. Table 4 shows the details on the number of reported contracts and precision performance of each tool. We observed that all the tools have a large number of false alarms. This is due to contract programmers have invented many heuristics to detect the potential vulnerabilities. In other words, most existing detection rules are obsolete. It motivates us to pursue (and generate) a more expressive and fine-grained rule to mitigate the false alarms.

4.2. Vulnerability rule abstraction

In this section, we introduce the definition of signature and show how we cluster and abstract the vulnerable/benign signatures from each cluster.

4.2.1. Definition

We define a vulnerability rule for a Solidity contract as following BNF:

$$\langle rule \rangle ::= \langle comp\_sig \rangle$$

$$\langle comp\_sig \rangle ::= \neg \langle comp\_sig \rangle \mid (\langle comp\_sig \rangle \vee \langle comp\_sig \rangle) \mid (\langle comp\_sig \rangle \wedge \langle comp\_sig \rangle) \mid (\langle comp\_sig \rangle \succ \langle comp\_sig \rangle) \mid \langle sig \rangle$$

$$\langle sig \rangle ::= \text{DataDep}(X,Y) \mid \text{ControlDep}(X,Y) \mid \text{ForLoop} \mid \text{IsInstance}(X,Y) \mid \text{Call}(L,X) \mid \text{SelfDestruct}(X) \mid \text{msg.sender} \mid \text{tx.origin} \mid$$

Here, the detection rule is composite of signatures. A composite signature is a negation (¬) of itself, or conjunction (∧), union (∨), succeed (≻) with another composite signature. A composite signature can also be a single vulnerability signature. Specifically, vulnerability signature indicates basic program relationships and built-in keywords of Solidity language. For example, the data dependency (*DataDep*(X, Y)) relationship denotes that variable X has data dependency to Y (i.e., variable assignment operations). The control dependency (*ControlDep*(X,Y)) indicates assertation operations (e.g., require, assert) between variables X and Y. The for loop *ForLoop* denotes the function body exists a for loop statement. The *IsInstance*(X,Y) denotes the variable X is a type of variable Y. Call operation *CALL*(L,X) includes low level calls (e.g., `call.value()` and `send()` in Solidity) and high level calls (i.e., user-defined function calls). Here, variable L represents the result of call operations and variable X represents the parameters required by the call. *SelfDesutrct*(X) is a built-in function call in Solidity. Once it is called, the service of current contract is stopped and the rest balance is transferred to an arbitrary receiver X. The msg.sender and tx.origin are built-in variables. Specifically, msg.sender denotes the address of current contract and tx.origin denotes the origin of call chains (ethereum, 2017).

Algorithm 1: Contract Clustering and Signature Abstraction Algorithm

```
    input  : $SourceCode$, source code of smart contracts
    output: $SignatureCands$, abstracted signature candidates
1   // Contract Clustering Process
2   ASTs = $getAST(SourceCode)$
3   nASTs = $ASTNomalization(ASTs)$
4   distanceMatrix = $List[N, N]$
5   // N is the number of trees
6   foreach idx $i \in range(nASTs)$ do
7       foreach idx $j \in range(nASTs)andi \neq j$ do
8           treeEdtDist = ARTED(nASTs[i], nASTs[j])
9           // Calculate the distance between two trees
10          distanceMatrix[i, j] = treeEdtDist
11  Clusters = $hierarchicalClustering(distanceMatrix)$
12  // Signature Abstraction Process
13  $SignatureCands \leftarrow \emptyset$
14  foreach cluster $c \in Clusters$ do
15      $PDGs \leftarrow \emptyset$
16      foreach tree $t \in c$ do
17          PDG $p \leftarrow getPDG(t)$
18          $pn \leftarrow PDGNormalization(p)$
19          $PDGs \leftarrow PDGs \cup pn$
20      $commonSeq \leftarrow LCS(PDGs)$
21      $SignatureCands \leftarrow SignatureCands \cup commonSeq$
22  return $SignatureCands$
```

4.2.2. Contract clustering

In this section, we first define contract similarity on normalized ASTs, and then we cluster similar trees by using hierarchical clustering algorithm. The clustering procedure can be found at line 1 to line 10 in Algorithm 1.

Contract Similarity. We define the contract similarity by considering both semantic and structural information of the code. To this end, we use AST (Abstract Syntax Tree) to represent the code of the functions of each contract. For each AST of a Solidity function, we normalize the concrete nodes in the AST for retaining core information and abstracting away unimportant details such as variable names or constant values, as shown in line 2 of Algorithm 1. For each AST corresponding to a function, we just retain the information such as node type, name, parameter and return value (if contained). For the variable names (e.g., `_indexs` in code block A and code block B of Fig. 4), they will be normalized with the token asterisk "$*$". Similarly, we repeat the same normalization for constant values of the types `string`, `int`, `bytes` or `uint`.

Given two trees, we use the tree edit distance between two normalized ASTs as their distance. The AST normalization process is shown in line 3 of Algorithm 1. In this work, we apply a robust algorithm for the tree edit distance (ARTED) (Pawlik and Augsten, 2015), which computes the optimal path strategy by performing an exhaustive search in the space of all possible path strategies. Here, *path strategy* refers to a mapping between two paths of the two input trees (or subtrees), as the distance between two (sub)trees is the minimum distance of four smaller problems, i.e., (1) the edit distance between two empty trees, (2) the edit distance of transferring a tree F to an empty tree, (3) the edit distance of transferring an empty tree to a tree F and (4) the edit distance of transferring a tree F to another tree G. Note that though ARTED runs in *quadratic* time and space complexity, it is guaranteed to perform as good or better than its competitors (Pawlik and Augsten, 2015).

Contract Clustering. We cluster the ASTs via hierarchical clustering algorithm with complete linkage (Defays, 1977), as shown in line 4 to line 10 in Algorithm 1. Then, we group the codes in Fig. 4 with considerable modification. We deem that the ASTs in each cluster share commonalities as a feature (or signature) for a vulnerability category.

4.2.3. Signature abstraction

After clustering contract functions with AST, we abstract signature by referring to their PDG (Program Dependency Graph) information. The reason lies in that PDG allows us to capture the code semantic features like control and data dependencies.

PDG Representation. For each AST, we transfer its code into a PDG including all its depended code elements such as global variables and called functions, as shown in line 13 to line 17 in Algorithm 1. In a PDG, each of its nodes is an instruction and the edge between nodes indicates data dependency, control dependency, and call relation between the nodes. Thus, given a cluster containing N Solidity functions, we reduce it into a problem of finding the common subgraph of N PDGs. The normalization of PDGs is shown in line 18 in Algorithm 1.

PDG Matching. The graph matching problem is a NP-complete problem. We simplify the problem with the following steps. Before matching, we also abstract away variable names and constant values in the PDGs as we do that for AST. Next, we simplify the calculation by flattening the graph into a node sequence (via depth first order search) and align the sequences by LCS algorithm (Maier, 1978), as shown in line 20 in Algorithm 1 The aligned graph nodes are considered as commonalities shared by the code in the same cluster.

As a result, the signature abstracted from a cluster is essentially a graph skeleton, as shown in Fig. 4. Then, we manually inspect those skeletons and refine them into usable signatures. The refining process requires manual efforts because some signatures are semantically similar to others but different in syntax. These signatures require to be filtered out by human expert. After we repeat the above procedures on both vulnerable and benign contracts, we construct a set of vulnerable and benign signatures.

4.2.4. Rule composition

In this work, we follow the following heuristics to integrate the signatures into a rule. Generally, a *detection rule is a composite boolean expression of vulnerability signatures*. Given a vulnerability category, a detection rule first requires the input contract match with the vulnerable signatures. The vulnerable signatures are essential ingredients of forming a vulnerability. Therefore, if the input contract is not matched with vulnerable signatures, the contract should be considered as invulnerable. Next, the input contract is required not to match with benign signatures. The benign signatures are the best practices to defend vulnerabilities. If the input contract matches with them, it suggests that the contract is capable for defending vulnerabilities and should not be reported as vulnerability.

4.3. Case study: Abstracted signatures

We applied the three chosen scanners to 76,354 contracts. Overall, Slither reports the most vulnerabilities, in total 3422 (623 + 67 + 2678 + 54) candidates covering four types. In contrast, SmartCheck reports 439 (150 + 289) candidates and Oyente reports only 143 candidates. After they are processed by our methods, we abstract 4 vulnerable signatures and 6 benign signatures, as shown in Table 5. Based on these signatures, we further integrate them into 4 detection rules, as shown in Table 6. In this section, we elaborate the signatures with examples to evidence their representativeness.

Signature of Reentrancy. We extract 4 signatures from TPs and FPs of reported reentrancy vulnerabilities, including 1 vulnerable signature (**SIG1**) and 3 benign signatures (**SIG2**, **SIG3**, **SIG4**).

Table 5
Extracted signatures from different vulnerability categories.

ID	Vulnerability	V/B	Signature
1	Reentrancy	V	*DataDep*(_,*X*) ≻ *Call*(_,*X*)
2		B	*ControlDep*(msg.sender,*X*) ≻ *DataDep*(_,*X*) ≻ *Call*(_,*X*)
3		B	*DataDep*(_,*X*) ≻ *IsInstance*(*X*,addr) ≻ *Call*(_,*X*)
4		B	*ControlDep*(*Y*,_) ≻ *DataDep*(_,*X*) ≻ *Call*(_,*X*) ≻ *DataDep*(*Y*,_)
5	Unexpected revert	V	*ForLoop* ≻ *Call*(*L*,*X*) ≻ *ControlDep*(*L*,_)
6		B	*ForLoop* ≻ (*IsInstance*(*X*,addr) ∧ *Call*(*L*,*X*)) ≻ *ControlDep*(*L*,_)
7	Abuse of Tx.origin	V	*DataDep*(tx.origin,*X*) ≻ *ControlDep*(*X*,_)
8		B	*DataDep*(msg.sender,*Y*) ≻ *DataDep*(tx.origin,*X*) ≻ *ControlDep*(*X*,*Y*)
9	SelfDestruct	V	*DataDep*(_,*X*) ≻ *SelfDestruct*(X)
10		B	*ControlDep*(msg.sender,*X*) ≻ *DataDep*(_,*X*) ≻ *SelfDestruct*(X)

Table 6
Detection rules for each vulnerability.

ID	Vulnerability	Rule
1	Reentrancy	SIG1 ∧ ¬ (SIG2 ∨ SIG3 ∨ SIG4)
2	Revert	SIG5 ∧ ¬ SIG6
3	Tx.origin	SIG7 ∧ ¬ SIG8
4	Self-destruct	SIG9 ∧ ¬ SIG10

```
1 contract BancorLender {
2   ERC constant public bancorToken =
3     ERC(0
       x1f573d6fb3f13d689ff844b4ce37794d79a7ff1c);
4   function closePosition(uint _idx) public {
5     ...
6     bancorToken.transfer(agets[_idx].lender,
      amount);
7     return;
8     } }
```

Fig. 5. A real case of using SIG3 (a hard-coded address at line 3), a FP of *reentrancy* for Slither.

SIG1 is abstracted from general patterns of reentrancy vulnerabilities. This signature consists of two parts: (1) the read or write operation of variable X (i.e., *DataDep*(_, X)) and (2) the call operation with the parameter variable X (i.e., *Call*(_, X)).

SIG2 adds various forms of checks (i.e., in `require` or `assert`) for msg.sender compared with SIG1. For example, SIG2 checks whether the identity of msg.sender is checked under certain conditions (e.g., equal to the owner, or with a good reputation, or having the dealing history) before calling the external payment functions. With the identity check, the function is only accessible to related users, blocking the malicious attack from attackers. Example of this signature can be found at Fig. 1.

SIG3 describes a falsely reported case of transferring balance to a fixed address. In Fig. 5, the function `closePosition` sends balance to a token `bancorToken` which is assigned with a fixed address at line 2. According to the detection rule of Slither (See Rule 1), this code is a vulnerability because — (1) it reads the `public` variable `agets[_idx]`; (2) then calls external function `bancorToken.transfer()`; (3) last, writes to the `public` variable `agets[_idx]`. However, in practice, this contract can never be easily exploited to steal ethers due to the hard-coded address constant (i.e., `0x1F...FF1C`). Note that the constant address can be a malicious address, under such circumstance this address cannot protect contract. However, this case is very rare. Therefore, we choose to trust the creator of the contract as well as the designated addresses are benign.

SIG4 is to prevent from the recursive entrance of the function — eliminating the issue from root. For instance, in Fig. 6, the internal instance variable `reEntered` will be checked at line 5 before processing the business logic between line 8 and 10. To prevent the reentering due to calling `ZTHTKN.buyAndSetDivPercentage.value()`, `reEntered` will be switched to `true`; after the transaction is done, it will be reverted to `false` to allow other transactions.

```
1 contract ZethrBankroll is ERC223Receiving {
2 ZTHInterface public ZTHTKN;
3 bool internal reEntered;
4 function receiveDividends() public payable {
5   if (!reEntered) {
6     ...
7     if (ActualBalance > 0.01 ether) {
8       reEntered = true;
9       ZTHTKN.buyAndSetDivPercentage.value(
      ActualBalance)(address(0x0), 33, "");
10    }
11 } } }
```

Fig. 6. A real case of using SIG4 (an execution lock of `reEntered`), an FP of *reentrancy* for Slither.

```
1 function withdraw() private {
2   for(uint i = 0; i < player_[uid].planCount; i
      ++) {
3     ...
4     address sender = msg.sender;
5     sender.transfer(amount);
6   } }
7
```

Fig. 7. A real FP of Unexpected Revert reported by SmartCheck, where only one account is involved (SIG6).

Signature of Unexpected Revert. We extract 2 signatures from reported Unexpected Revert vulnerabilities, including 1 vulnerable signature **SIG5** and 1 benign signature **SIG6**.

SIG5 represents general patterns of Unexpected Revert vulnerabilities. This signature consists of three parts: (1) the for loop program structure (i.e., *ForLoop*); (2) the call operation of the variable X (i.e., *Call*(_, X)); (3) the result of call operation is further checked by assertions.

According to the recent technical article (Anon, 2019b), the rules of *Call/Transaction in Loop* are neither sound nor complete to cover most of the unexpected revert cases. At least, modifier `require` is often ignored, which makes Slither and SmartCheck incapable to check possible revert operations on multiple account addresses. Here, multiple accounts must be involved for exploiting this attack — the failure on one account blocks other accounts via reverting the operations for the whole loop. Hence, in the example of Fig. 7, the operations in the loop are all on the same account (i.e., `sender` at line 5) and potential revert will not affect other accounts. Therefore, the transfer operation of which target is a single address is considered as **SIG6**.

Signatures of `Tx.Origin` Abusing. We extract 2 signatures from the truly vulnerable contracts and falsely reported contracts, including 1 vulnerable signature (**SIG7**) and 3 benign signatures (**SIG8**).

For **SIG7**, this signature is extracted from general patterns of tx.origin vulnerabilities. This vulnerability first reads the value of

```
1 function destroyDeed() public {
2   require(msg.sender == owner);
3   if (owner.send(address(this).balance)) {
4     selfdestruct(burn);}
5 }
```

Fig. 8. A real FP of *self-destruct abusing* by Slither, as `selfdestruct()` is used under two checks at line 2,3 (SIG10).

tx.origin, followed by an assignment to variable X (i.e., *DataDep*(tx.origin, X)). After this, the function has an assertion to this variable (i.e., *ControlDep*(X,_)). While we extract signatures from the TPs of vulnerabilities, we find that our **SIG7** is slightly looser than the detection rule in Slither. Slither skips the function if there exists a read operation to a particular variable msg.sender, ignoring that some of these variables are irrelevant to tx.origin. In order not to overlook potential vulnerabilities, our **SIG7** only requires a read of tx.origin, followed by an assertion on this variable.

For **SIG8**, we observe that SmartCheck reports much more cases (210) than Slither (34), but has lower precision performance than Slither. After our investigation, we find that the incorrect reports of SmartCheck are due to the unsound rules (as shown in Rule 5). That is, SmartCheck simply reports vulnerability once tx.origin appears in assertion statements. However, under some circumstance (e.g., comparing msg.sender with tx.origin), the use of tx.origin should not be reported. We summarize the **SIG8** based on the FPs of SmartCheck.

$$\boxed{\mathit{DataDep}(\mathit{tx.Origin}, X) \succ \mathit{ControlDep}(X, \_) \Rightarrow \text{Tx.Origin abusing} \quad (5)}$$

Signature of Self-destruct Abusing. We extract 2 signatures from the self-destruct vulnerabilities, including 1 vulnerable signature **SIG9** and 1 benign signature **SIG10**.

SIG9 is extracted from general patterns of self-destruct vulnerabilities. This signature consists of two parts: (1) the read or write operation of variable X (i.e., *DataDep*(_, X)) and (2) the call operation of the self-destruct with the parameter X (i.e., *SelfDestruct*(X)).

For **SIG10**, we extract this signature from FPs of tools. In the existing scanners, only Slither detects the misuse of self-destruct, which is called suicidal detection. In total, Slither reports 54 cases of suicidal via its built-in rule — as long as function *SelfDestruct* is used, no matter what the context is, Slither will report it. Obviously, the Slither's rule is too simple and too general. It mainly works for directly calling *SelfDestruct* without permission control or conditions of business logic — under such circumstance (3 out of 54), the Slither rule can help to detect the abusing. In practice, in most cases (51 out of 54) *SelfDestruct* is called with the `admin` or `owner` permission control or under some strict conditions in business logic. For example, *SelfDestruct* is indeed required in the business logic at line 2 of Fig. 8. As the owner wants to stop the service of the contract via calling *SelfDestruct*, after the transactions are all done, the contract becomes inactive. Note that parameter `burn` is just padded to call *SelfDestruct* in a correct way. Hence, we summarize the **SIG10**, adding a strict condition control or a self-defined modifier for identity check when using *SelfDestruct*.

In brief, for a vulnerability type, we use the vulnerable signatures to match potential vulnerabilities, which yield a better recall. Then, we leverage corresponding benign signatures to filter out false reports.

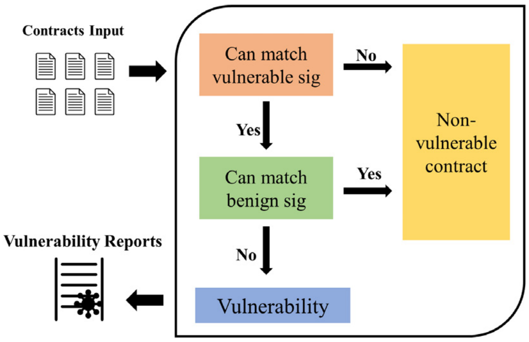


Fig. 9. The detection workflow of Vulpedia.

4.4. Vulnerability detection

The implementation of the vulnerability detection of Vulpedia is based on the previously abstracted signatures and integrated detection rules, but slightly different from them. The workflow of detection is shown in Fig. 9. Specifically, in this workflow, Vulpedia reports vulnerability only when the vulnerable signatures are matched meanwhile the benign signatures are not matched. That is, the vulnerable and benign signatures are separated things. However, in previous subsection, the signatures are combined to form detection rules. The reason is that our benign signatures are designed to filter out false positive reports. The detection rules shown in Table 6 all follow the pattern that the vulnerable signatures should be matched but the benign ones should not. Therefore, though the implementation of the detection process seems differently, the logic of the workflow is the same with previous designs.

5. Evaluation

Experimental Environment. Throughout the evaluation, all the steps are conducted on a machine running on Ubuntu 18.04, with 8 core 2.10 GHz Intel Xeon E5-2620V4 processor, 32 GB RAM, and 4 TB HDD. For the scanners used in evaluation, no multithreading options are available and only the by-default setting is used for them.

Tool Implementation. Vulpedia is implemented based on the Slither analyzer. We adopt the AST analysis from Slither, and we build PDG analysis based on the CFG (control flow graph) and call graph of Slither. The vulnerability signatures are implemented as detectors in nearly 1000 lines of Python code. The demo of our tool can be found at https://github.com/ToolmanInside/vulpedia_demo.

Dataset for Tool Evaluation. To take a different dataset from contracts we used in empirical study, we get another address list of contracts from Google BigQuery Open Dataset. After removing contracts that already used in our empirical study, we get the other 17,770 real-world contracts deployed on Ethereum, on which we fairly compare our resulted tool Vulpedia with the version of the scanners: Slither v0.6.4. Oyente v0.2.7,1u SmartCheck v2.0 and Securify v1.0 that is open-sourced at Dec 2018. The evaluation dataset is opened along with empirical study dataset at https://drive.google.com/file/d/1kizsz0_8B8nP4UNVr0gYjaj25VVZMO8C.

The evaluations are conducted based on a relaxed assumption that the owners of contracts are not malicious. That is, the operations of the owners are all deemed as defense behaviors to vulnerabilities. The evaluations aim to answer these RQs:

RQ1. How is the precision of Vulpedia, compared with the existing scanners in vulnerability detection?

Table 7
The detection performance for our tool and other existing ones on the 17,770 contracts, where #N refers to the number of detections, P% and R% refer to the precision rate and the recall rate among the number of detections, respectively. Note that P% = (#TP of the tool)/#N, and R% = (#TP of the tool)/(#TP in union of all tools).

Vulnerability	Slither			Oyente			SmartCheck			Securify			Vulpedia		
	#N	P%	R%	#N	P%	R%	#N	P%	R%	#N	P%	R%	#N	P%	R%
Reentrancy	162	9.8%	32.6%	28	7.1%	4.1%	N.A.	N.A.	N.A.	797	1.1%	18.3%	119	**28.5%**	**69.3%**
Abuse of tx.origin	23	43.4%	33.3%	N.A.	N.A.	N.A.	44	33.3%	56.6%	N.A.	N.A.	N.A.	98	**88.7%**	**96.6%**
Unexpected revert	356	5.8%	67.7%	N.A.	N.A.	N.A.	51	47.1%	**77.4%**	N.A.	N.A.	N.A.	43	**48.8%**	67.7%
Self destruct	18	16.6%	42.8%	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	20	**35.0%**	**100%**

RQ2. How is the recall of Vulpedia? Can our signature-based method report more vulnerabilities?
RQ3. How is the efficiency of Vulpedia, in tool comparison on the datasets?

5.1. RQ1: Evaluating the precision of tools

As mentioned in Section 4.3, we have learned 10 signatures in total for the four types of vulnerabilities. To evaluate the effectiveness of the resulted vulnerable signatures and detection rules, we apply them on the 17,770 newly collected contracts and compare with the other state-of-the-arts detection tools. Details on the performance of each tool are shown in Table 7. Note that all TPs are manually verified by our authors.

In Table 7, we list 280 detection results of Vulpedia, with an averaged precision of 50.2%, regardless of vulnerability types. In comparison, Slither has an averaged precision of 18.9%; Oyente's averaged precision is 7.1%; SmartCheck's averaged precision is 40.2%; and Securify's precision is surprisingly only 1.1%. In the rest of this section, we analyze the false positives of these tools from the perspective of supporting vulnerability signatures.

FPs of Reentrancy. Among the four supported tools except SmartCheck, Vulpedia yields the lowest FP rate (71.5%) owing to the adoptions of benign signatures for reentrancy. FP rates of other tools are even higher. For example, the FP rate of Securify is 98.9%, as its detection pattern is too general but has not considered possible defense to vulnerabilities in code. Slither adopts Rule 1 to detect, but it supports no benign signatures — its recall is acceptable, but FP rate is high. Oyente adopts Rule 2 and has no benign signatures — its recall is low due to the strict rule, and its FP rate is also high.

FPs of Unexpected Revert. As summarized in Section 4.3, Slither reports Unexpected Revert vulnerability when a call in loop is detected, ignoring the potential false alarms (i.e., low level call in a loop). This coarse detection rule leads to 335 FPs. SmartCheck handles SIG5 but not SIG6 and leads to 27 FPs. In comparison, Vulpedia combines **SIG5** and **SIG6** for integrating detection rule, yielding the lowest FP rate 51.2%.

FPs of `Tx.Origin` Abusing. Slither has a strict rule for detecting this type, only checking the existence of `tx.Origin == msg.sender`. We find that this tool also skips the function if there exists a read operation to a particular variable msg.sender, ignoring that some of these variables are irrelevant to tx.origin. For the case that tx.origin is compared with an unrelated address variable, Slither reports it as vulnerability, causing FPs. Comparatively, SmartCheck and Vulpedia manage to include all the identity check cases, but meanwhile also lead to FPs due to the fact — accurate symbolic analysis is not adopted in SmartCheck or Vulpedia to suggest whether tx.Origin can be used to rightly replace msg.sender. Hence, the FP rate due to ignoring **SIG8** is higher than that of Vulpedia.

FPs of Self-destruct Vulnerability. Vulpedia has 13 FPs. After inspecting, we find 10 FPs are due to the unsatisfactory handling of **SIG10**. That is, the identity check hides in self-defined modifiers. Function modifiers are overlooked by Vulpedia, causing FPs.

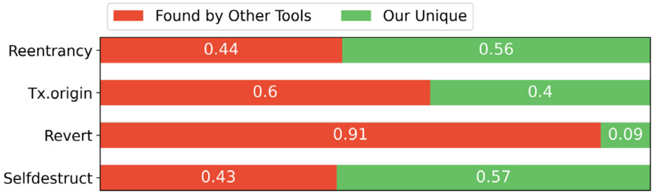


Fig. 10. Comparing the vulnerabilities only reported by Vulpedia with vulnerabilities reported by other tools. "Our Unique" means those only found by Vulpedia. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Comparatively, Slither only reports 3 true positives. The reason is that Slither simply reports vulnerability when a *SelfDestruct* call is detected. Due to the inconsideration of the potential access controls, Slither performs less precision than Vulpedia.

Answer to RQ1: Vulpedia performs best in evaluations of precision among tools. In detecting `tx.origin` vulnerability, Vulpedia outperforms the second best tool by 45.3% (88.7% - 44.3%). The reason of the high precision performance is Vulpedia adopts effective benign signatures to remove false reports.

5.2. RQ2: Evaluating the recall of tools

In Table 7, in most cases, Vulpedia yields the best recall except on unexpected revert, where R% for SmartCheck is 77.4% and R% for Vulpedia is 67.7%. Based on the vulnerable signature abstracted in empirical study, we expect Vulpedia can find more similar vulnerable candidates. A comparison between vulnerabilities *only* found by Vulpedia (denoted by green bars) and vulnerabilities found by other tools (represented by red bars) is shown in Fig. 10.

Recall of Reentrancy. In this vulnerability, Vulpedia performs best by report 69.3% vulnerabilities. Among all TPs, Vulpedia finds 56% unique TPs that are missed by other evaluated tools. We find that the other three tools commonly fail to consider the user-defined function `transfer()`, not the built-in payment function `transfer()`. For the example in Fig. 11, Slither and Securify miss it as they mainly check the external call for low-level functions (e.g., `send()`, `value()`) and built-in `transfer()`, ignoring user defined calls. Oyente does not report this example, as it fails in the balance check according to Rule 2. Comparatively, Vulpedia detects this vulnerability, as we have an vulnerable signature that has a high code similarity with this example. Notably, though Vulpedia has the best recall of 69.3%, it misses 30.7% TPs. This is due to the fact that reentrancy has many forms, and our vulnerable signature is not sufficient to cover those TPs.

Recall of Unexpected Revert. In this vulnerability, the performance of Vulpedia is slightly worse than SmartCheck (77.4%). Specifically, Vulpedia only reports 9% unique TPs while 91% TPs are found by other tools. The reason of the TPs missed by Vulpedia (reported by SmartCheck) are due to the incompleteness

```
contract Alice {
  ...
  function aliceClaimsPayment(bytes32 _dId, uint
      _amount, address _addr) external {
    require(deals[_dId].state==DS.Initialized);
    ...
    deals[_dId].state = DS.PaymentSentToAlice;
    if (_addr == 0x0) {msg.sender.transfer(
    _amount);}
    else {
      ERC20 token = ERC20(_addr);
      assert(token.transfer(msg.sender, _amount)
    ); }
  }
}
```

Fig. 11. A real case of reentrancy. This is a TP for Vulpedia but a FN for Slither, Oyente and Securify.

Table 8
The time (min.) of vulnerability detection for each scanner on 76,354 and 17,770 contracts. "S.C." denotes SmartCheck.

Dataset	Slither	Oyente	S.C.	Securify	Vulpedia
76,354	156	6434	641	N.A.	883
17,770	52	1352	141	8859	295

of our vulnerable signature **SIG6**. That is, the signature requires a *ControlDep* after *Call*. However, the *ControlDep* is unnecessary when the *Call* is a high level call (e.g., user defined function call) because assertion operations are already integrated in high level calls. Therefore, the signature causes FNs.

Recall of `Tx.Origin` Abusing. For this type, 96.6% TPs are found by Vulpedia— almost all TPs are found by Vulpedia. Additionally, Vulpedia reports 40% unique TPs which are missed by other tools. The reason is that we matches identity check of `Tx.Origin` in self-defined modifiers, which is commonly overlooked by other tools.

Recall of Self-destruct Abusing. For this type, all vulnerabilities (100%) are found by Vulpedia. Comparatively, Slither only reports 42.8% vulnerabilities. 57% of TPs are only found by Vulpedia. The rationale of TPs missed by Slither is that Slither skips the function if the function is only accessible to internal calls (i.e., set visibility to `internal`). These functions are however prone to being exploited by internal calls. Therefore, they should not be overlooked. Vulpedia leverages **SIG9** to match vulnerability candidates, so we have better recall performance.

> **Answer to RQ2:** Vulpedia has best performance on detection recall. Except Unexpected Revert, Vulpedia outperforms other tools on three vulnerabilities. The reason of this leading performance is our abstracted vulnerable signatures can represent essence of most vulnerabilities.

5.3. RQ3: Evaluating the efficiency

On Dataset for Empirical Study. In Table 8, Slither takes the least time (only 156 *min*) in detection. SmartCheck and Vulpedia have the comparable detection time (500∼1000 *min*). They are essentially of the same type of technique — pattern based static analysis. In practice, they may differ in performance due to implementation differences, but still, they are significantly faster than Oyente that applies symbolic execution. Compared with other dynamic analysis or verification tools (i.e., Mythrill and Securify that cannot finish in three days for the 76,354 contracts), Oyente is quite efficient. Notably, the signature abstraction time of Vulpedia is not included in the detection time, as it could be done off-line separately. Since signature abstraction is analogical to rules formulation, it is not counted in the detection time.

On Dataset for Tool Evaluation. On the smaller dataset, we observe the similar pattern of time execution — Slither is the most efficient, Oyente is least efficient (except Securify), and SmartCheck and Vulpedia have the comparable efficiency. Notably, Securify can finish the detection on 17,770 contracts, but it takes significantly more time than other tools. The performance issue of Securify rises due to the conversion of EVM IRs into datalog representation and then the application of verification technique. Oyente is also less efficient, as it relies on symbolic execution for analysis. Vulpedia should be comparable to SmartCheck and Slither, as these three all adopt rule based matching analysis. The extra overheads of Vulpedia, compared with Slither and SmartCheck, are signature-based code matching.

> **Answer to RQ3:** Vulpedia outperforms Securify and Oyente regarding the detection efficiency on both empirical evaluation and tool comparison. In general, Vulpedia is efficient as a signature-based vulnerability detection tool.

5.4. Threats to validity

In our experiments, we adopt recall rate as a metric, which is a potential threat. Generally, the recall rate indicates the number of TPs divided by the number of all vulnerabilities. However, it requires an overwhelming effort to find out all vulnerabilities (i.e., the ground truth). In our study, we evaluate recall performance based on the union of vulnerabilities reported by all tools. Additionally, in the abstraction of signatures, we manually confirm signatures, which may introduce bias. To alleviate this, we repeat our experiments for 3 times. Also, we note that the randomness is an inevitable factor in the evaluations of efficiency. We repeat the experiments for 5 times and record the average values. Besides, the abstracted signatures are prone to introducing incompleteness. To alleviate this, we implement our methods on the top of Slither, which facilitates our signature abstraction from PDGs.

6. Discussions

6.1. The relaxed security assumption

The experiments and comparisons are all conducted based on the relaxed security assumption. That is, we assume the operations of contract owner are not malicious behaviors. We follow this assumption because the security-design is more strict than ordinary contract when the contract is designed for industry needs. In fact, existing successful contracts (e.g., e-voting, NFT) have been audited by experts to be protected from rogue owners. To avoid our tool been blindly used by users and developers, this assumption should be pointed out.

6.2. The Weakness of Vulpedia

In this section, we discuss the improvement of the weakness of Vulpedia found in our experiment practice. In our view, involving manual efforts brings biases, and the biases may affect the effectiveness of the tool. However, Vulpedia relies on manual efforts, mainly in the two steps: (1) manually confirm the reports of existing tools in our empirical study. We add manpowers in this step because the existing static tools have severe limitations and produce a large number of false reports. Due to Ren et al. (2021), the Slither tool has a false positive rate over 70%. If the false reports are not removed from all reports, the dataset cannot be correctly labeled, and our signature abstraction is infeasible. (2) We manually integrate the vulnerable signatures and benign ones into vulnerability detection rules. In this step, we use manual efforts to filter out ineffective signatures. This is due to the lack of smart contract vulnerability benchmark. If we have a benchmark, we can replace the man-powers in this step and filter out ineffective signatures by running testing on the benchmark.

7. Related work

Vulnerability Detection in Smart Contracts. There is already a list of security scanners on smart contracts. From the perspective of software analysis, these scanners could be categorized into static- or dynamic-based. In the former category, Slither (trailofbits, 2019c) aims to be the analysis framework that runs a suite of vulnerability detectors. Oyente (Luu et al., 2016) analyzes the bytecode of the contracts and applies Z3-solver (de Moura and Bjørner, 2008) to conduct symbolic executions. Recently, SmartCheck (Tikhomirov et al., 2018) translates Solidity source code into an XML-based IR and defines the XPath-based patterns to find code issues. Securify (Tsankov et al., 2018) is proposed to detect the vulnerability via compliance (or violation) patterns to guarantee that certain behaviors are safe (or unsafe, respectively). These static tools usually adopt symbolic execution or verification techniques, being relevant to Vulpedia. However, none of them applies code-similarity based matching technique or takes into account the possible DMs in code to prevent from attacks.

There are some other tools that enable the static analysis for smart contracts. VeriSmart (So et al., 2020) proposes a domain-specific algorithm for verifying smart contracts. VerX (Permenev et al., 2020) combines symbolic execution and contract status abstraction to verify transactions. Zeus (Kalra et al., 2018) adopts XACML as a language to write the safety and fairness properties, converts them into LLVM IR (Anon, 2019c) and then feeds them to a verification engine such as SeaHorn (Gurfinkel et al., 2015). Besides, there is another EVM bytecode decompiling and analysis frame, namely (Octopus, 2019), which needs the users to define the patterns for vulnerability detections. To prevent the DAO, Grossman et al. propose the notion of effectively Callback Free (ECF) objects in order to allow callbacks without preventing modular reasoning (Grossman et al., 2018). Maian is presented to detect greedy, prodigal, and suicidal contracts (Nikolic et al., 2018), and hence the vulnerabilities to address differ from the types we address in this paper. The above tools are relevant, but due to various reasons (e.g., issues in tool availability), we cannot have a direct comparison with them.

The less relevant category includes dynamic testing or fuzzing tools: Manticore (trailofbits, 2019b), Mythril (ConsenSys, 2019a), MythX (ConsenSys, 2019b), Echidna (trailofbits, 2019a) and Ethracer (Kolluri et al., 2019). sFuzz (Nguyen et al., 2020) and Harvey (Wüstholz and Christakis, 2020) use the advanced techniques (e.g., concolic testing, fuzzing and tainting) for detection. Dynamic tools often target certain vulnerability types and produce results with few FPs. However, they are unsuitable for a large-scale detection due to the efficiency issue.

Code-similarity based Vulnerability Detection. In general, similar-code matching technique is widely adopted for vulnerability detection. In 2016, VulPecker (Li et al., 2016) is proposed to apply different code-similarity algorithms in various purposes for different vulnerability types. It leverages vulnerability signatures from National Vulnerability Database (NVD) (NIST, 2019) and applies them to detect 40 vulnerabilities that are not published in NVD, among which 18 are zero-days. As VulPecker works on the source code of C, Bingo (Chandramohan et al., 2016) can execute on binary code and compare the assembly code via tracelet (partial trace of CFG) extraction (David and Yahav, 2014) and similarity measuring. Vuddy (Kim et al., 2017) targets at exact clones and parameterized clones, not gapped clones, as it utilizes hashing for matching for the purpose of high efficiency. To sum up, these studies usually resort to the vulnerability database of C language for discovering similar zero-days. In contrast, plenty of our efforts are exhausted in gathering vulnerabilities from other tools for smart contracts and auditing them manually. Vulpedia adopts a more robust algorithm (e.g., LCS), which can tolerate big or small code gaps across the similar candidates of a vulnerability.

8. Conclusion

In this study, we propose Vulpedia, a static analyzer based on abstracted signatures. We focus on addressing one key challenges: the manually predefined detection rules can be obsolete. To this end, we first conduct an empirical study for signature abstraction. We leverage state-of-the-arts scanners to detect vulnerabilities on our training dataset. Based on their results, we propose a method to cluster similar contracts and abstract vulnerable signatures and benign signatures, respectively. After we collect all signatures, we conduct comparative evaluations with state-of-the-art tools. The results show that Vulpedia exhibits best performance of precision on 4 types of vulnerabilities and leading recall on 3 types of vulnerabilities with great efficiency performance.

CRediT authorship contribution statement

Jiaming Ye: Writing – original draft, Validation, Writing – review & editing. **Mingliang Ma:** Software, Investigation. **Yun Lin:** Methodology. **Lei Ma:** Visualization. **Yinxing Xue:** Conceptualization, Supervision. **Jianjun Zhao:** Methodology.

Declaration of competing interest

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

References

Adrian Manning, 2018. Solidity security: Comprehensive list of known attack vectors and common anti-patterns. https://blog.sigmaprime.io/solidity-security.html, Online; accessed 29 January 2019.

Anon, 2019a. A block explorer and analytics platform for ethereum. https://etherscan.io/, Online; accessed 29 January 2019.

Anon, 2019b. Secure smart contract security with transaction-ordering dependence. https://www.nvestlabs.com/2019/03/18/secure-smart-contract-security-with-transaction-ordering-dependence/, Online; accessed 29 January 2019.

Anon, 2019c. Llvm language reference manual. https://blog.sigmaprime.io/solidity-security.html, Online; accessed 29 January 2019.

Anonymity, 2019. Vulpedia: Detecting vulnerable smart contracts via abstracted vulnerability signatures. https://vulpedia.readthedocs.io/en/latest/, Online; accessed 29 July 2019.

Atzei, N., Bartoletti, M., Cimoli, T., 2016. A survey of attacks on ethereum smart contracts. IACR Cryptol. EPrint Arch. 2016, 1007.

Beck, R., Avital, M., Rossi, M., Thatcher, J.B., 2017. Blockchain technology in business and information systems research. Bus. Inform. Syst. Eng. 59 (6), 381–384.

Chandramohan, M., Xue, Y., Xu, Z., Liu, Y., Cho, C.Y., Tan, H.B.K., 2016. Bingo: cross-architecture cross-OS binary search. In: FSE 2016. pp. 678–689.

ConsenSys, 2019a. Mythril. https://github.com/ConsenSys/mythril-classic, Online; accessed 29 January 2019.

ConsenSys, 2019b. Mythx. https://mythx.io/, Online; accessed 29 January 2019.

ConsenSys Diligence, 2019. Ethereum smart contract best practices:Known attacks. https://consensys.github.io/smart-contract-best-practices/known_attacks/, Online; accessed 29 January 2019.

David, Y., Yahav, E., 2014. Tracelet-based code search in executables. In: PLDI '14. pp. 349–360.

Defays, D., 1977. An efficient algorithm for a complete link method. Comput. J. 20 (4), 364–366. http://dx.doi.org/10.1093/comjnl/20.4.364.

ethereum. 2017. Solidity Document, Website, https://solidity.readthedocs.io/en/v0.4.24/contracts.html?highlight=fallback.

Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M., Zohar, Y., 2018. Online detection of effectively callback free objects with applications to smart contracts. PACMPL 2 (POPL), 48:1–48:28. http://dx.doi.org/10.1145/3158136.

Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A., 2015. The SeaHorn verification framework. In: CAV 2015. pp. 343–361.

Jiang, B., Liu, Y., Chan, W.K., 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In: ASE. ACM, pp. 259–269.

Kalra, S., Goel, S., Dhawan, M., Sharma, S., 2018. ZEUS: analyzing safety of smart contracts. In: NDSS 2018.

Kim, S., Woo, S., Lee, H., Oh, H., 2017. VUDDY: A scalable approach for vulnerable code clone discovery. In: IEEE Symposium on S & P. IEEE Computer Society, pp. 595–614.

Kolluri, A., Nikolic, I., Sergey, I., Hobor, A., Saxena, P., 2019. Exploiting the laws of order in smart contracts. In: Proceedings of the ISSTA 2019, Beijing, China, July 15-19, 2019.. pp. 363–373. http://dx.doi.org/10.1145/3293882.3330560.

Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., Hu, J., 2016. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In: ACSAC. ACM, pp. 201–213.

Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A., 2016. Making smart contracts smarter. In: CCS 2016. pp. 254–269.

Maier, D., 1978. The complexity of some problems on subsequences and supersequences. J. ACM 25 (2), 322–336. http://dx.doi.org/10.1145/322063.322075.

melonproject, 2019. Oyente. https://github.com/melonproject/oyente, Online; accessed 29 January 2019.

de Moura, L.M., Bjørner, N., 2008. Z3: an efficient SMT solver. In: TACAS 2008. pp. 337–340.

NCC Group, 2019. Decentralized application security project (or DASP) top 10 of 2018. https://dasp.co/, Online; accessed 29 January 2019.

Nguyen, T.D., Pham, L.H., Sun, J., Lin, Y., Minh, Q.T., 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 778–788.

Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A., 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018. pp. 653–663. http://dx.doi.org/10.1145/3274694.3274743.

NIST, 2019. National vulnerability database (NVD). https://www.nist.gov/programs-projects/national-vulnerability-database-nvd, Online; accessed 29 January 2019.

Octopus, 2019. Octopus. https://github.com/quoscient/octopus, Online; accessed 29 January 2019.

Parity Technologies, 2017. The multi-sig hack: A postmortem. https://www.parity.io/the-multi-sig-hack-a-postmortem/, Online; accessed 29 January 2019.

Pawlik, M., Augsten, N., 2015. Efficient computation of the tree edit distance. ACM Trans. Database Syst. 40 (1), 3:1–3:40.

Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M., 2020. Verx: Safety verification of smart contracts. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 1661–1677.

Ren, M., Yin, Z., Ma, F., Xu, Z., Jiang, Y., Sun, C., Li, H., Cai, Y., 2021. Empirical evaluation of smart contract testing: what is the best choice? In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 566–579.

So, S., Lee, M., Park, J., Lee, H., Oh, H., 2020. VeriSmart: A highly precise safety verifier for ethereum smart contracts. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 1678–1694.

Szabo, N., 1996. Smart contracts: Building blocks for digital markets. http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html. Online; accessed 29 January 2019.

Tian, Z., Tian, J., Wang, Z., Chen, Y., Xia, H., Chen, L., 2022. Landscape estimation of solidity version usage on ethereum via version identification. Int. J. Intell. Syst. 37 (1), 450–477.

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y., 2018. SmartCheck: Static analysis of ethereum smart contracts. In: WETSEB@ICSE 2018. pp. 9–16.

trailofbits, 2019a. Echidna. https://github.com/trailofbits/echidna, Online; accessed 29 January 2019.

trailofbits, 2019b. Manticore. https://github.com/trailofbits/manticore, Online; accessed 29 January 2019.

trailofbits, 2019c. Slither. https://github.com/trailofbits/slither.

Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T., 2018. Securify: Practical security analysis of smart contracts. In: CCS 2018. pp. 67–82.

Unman, J.D., 1989. Principles of database and knowledge-base systems. Computer Science Press.

Wüstholz, V., Christakis, M., 2020. Harvey: A greybox fuzzer for smart contracts. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1398–1409.

Zhao, X., Chen, Z., Chen, X., Wang, Y., Tang, C., 2017. The DAO attack paradoxes in propositional logic. In: ICSAI 2017. pp. 1743–1746. http://dx.doi.org/10.1109/ICSAI.2017.8248566.

Mr. Jiaming Ye received the BE degree from NorthEastern University and M.Sc. degree from University of Science and Technology of China (USTC). Since Oct. 2020, he has been a Ph.D. student in Kyushu University, Japan. He is now advised by Prof. Jianjun Zhao and co-advised by Prof. Lei Ma. His research interest lies in software analysis, software testing and smart contract security.

Mr. Mingliang Ma received the BE degree from Dalian Polytechnic University and M.Sc. degree from University of Science and Technology of China (USTC). Since April. 2021, he has been a Ph.D. student in Nanjing University. He is now advised by Prof. Lin Chen. His research interest lies in software analysis and smart contract testing.

Dr. LIN Yun, Senior Research Fellow in School of Computing, National University of Singapore. He received his Ph.D degree in Computer Science from Fudan University and Bachelor degree in Software Engineering from East China Normal University. His research interests span software debugging, testing, code recommendation, and cybersecurity.

Prof. Lei Ma is currently an Associate Professor with shared appointments between (1) University of Alberta, Canada and (2) Kyushu University, Japan. He is also honorably selected as a Canada CIFAR AI Chair and Fellow at Alberta Machine Intelligence Institute (Amii). Previously, he received the B.E. degree from Shanghai Jiao Tong University, Shanghai, China, and the M.E. and Ph.D. degrees from The University of Tokyo, Tokyo, Japan. His recent research centers around the interdisciplinary fields of software engineering (SE) and trustworthy artificial intelligence (AI) with a special focus on the quality, reliability, safety and security assurance of machine learning and AI Systems. Many of his works were published in top-tier software engineering, AI, and security venues (e.g., TSE, TOSEM, EMSE, ICSE, FSE, ASE, ISSTA, CAV, TNNLS, ICML, NeurIPS, ACM MM, AAAI, IJCAI, TDSC). He is a recipient of more than 10 prestigious academic awards, including three ACM SIGSOFT Distinguished Paper Awards.

Prof. Yinxing Xue received the BE and ME degrees from Wuhan University, China, and the Ph.D. degree in computer science from National University of Singapore (NUS), in 2013. He is currently a pretenure research professor with the University of Science and Technology of China (USTC). Since Apr. 2018, he has been a research professor with the School of Computer Science and Technology, USTC. His research interest includes software program analysis, software testing, cyber security issues (e.g., malware detection, intrusion detection and vulnerability detection).

Prof. Jianjun Zhao received a B.S. degree from Tsinghua University, China, in 1987 and a Ph.D. degree from Kyushu University, Japan, in 1997, both in computer science. He joined the Department of Computer Science and Engineering at Fukuoka Institute of Technology, Japan, as an Assistant Professor and was then promoted to Associate Professor in 2000. From 2005 to 2016, he worked as a Professor in the School of Software and the Department of Computer Science and Engineering at Shanghai Jiao Tong University. Since 2016, he has joined the School of Information Science and Electrical Engineering at Kyushu University as a Professor. His research interests include program analysis and verification, quality assurance for AI systems, automatic programming, software testing, and programming language design.