Empirical Software Engineering (2022) 27:197
https://doi.org/10.1007/s10664-022-10218-2

# SmartFast: an accurate and robust formal analysis tool for Ethereum smart contracts

**Zhaoxuan Li**[1,2] · **Siqi Lu**[3,4] · **Rui Zhang**[1,2] · **Rui Xue**[1,2] · **Wenqiu Ma**[1,2] · **Rujin Liang**[3,4] · **Ziming Zhao**[5] · **Sheng Gao**[6]

Accepted: 29 July 2022 / Published online: 12 October 2022

**Abstract**
Recently, although state-of-the-art (SOTA) tools were designed and developed to analyze the vulnerabilities of smart contracts on Ethereum, security incidents caused by these vulnerabilities are still widespread. This can be attributed to the fact that each tool has various standards for judging the severity of vulnerabilities. More importantly, tools fail to identify all the vulnerabilities accurately and comprehensively as the evolution of vulnerabilities. To this end, we first propose a vulnerability assessment model to unify the vulnerability measurement standards. Next, we design a static analysis tool called SmartFast, which expresses the contract source code as a novel intermediate representation named SmartIR. Using preset rules and taint tracking technology, SmartFast matches SmartIR to locate the vulnerability code. Furthermore, SmartFast can recommend the optimization of the contract code automatically. Finally, we implement a prototype of SmartFast with 25K lines of code and compare it with 7 SOTA tools on three datasets (a total of 13,687 public contracts). The results indicate that SmartFast is efficient (only took a few seconds per contract) and robust (0.4% failure rate and resistance to the general code confusion methods). Besides, compared with other tools, SmartFast can detect more kinds of vulnerabilities (119) with a higher precision rate (98.43%) and a recall rate (85.12%), which confirms the conclusion of the theoretical analysis in the paper.

**Keywords** Blockchain · Smart contracts · Solidity · Security vulnerability · Formal static analysis

## 1 Introduction

As one of the core technologies of blockchain, smart contracts carry great economic value. Once smart contracts have security vulnerabilities, malicious users may launch attacks by leveraging the vulnerabilities to steal coins from users' accounts. For example, in April

Communicated by: Yuan Zhang

✉ Siqi Lu
080lusiqi@sina.com

Extended author information available on the last page of the article.

2018, the $900 million market value of Beautychain's token BEC fell to almost zero due to a vulnerability in the contract code (Bocek and Stiller 2018); In May 2019, Binance was hacked with more than 7,000 Bitcoins stolen. These frequent security incidents show that it is necessary to minimize the vulnerabilities in smart contracts. The most effective way is to formally analyze smart contracts to identify vulnerabilities and fix them before deployment. While this observation is ubiquitous, only a handful of smart contract projects (e.g., MakerDAO DappHub 2019) have been formally verified so far. Since the applications of smart contracts are ubiquitous in Ethereum (Wood et al. 2014), this paper focuses on achieving security analysis of Ethereum smart contracts.

**State-of-the-Art (SOTA) Analysis Tools** In Ethereum, most smart contracts are written in Solidity, a high-level programming language, and then executed in the form of Ethereum Virtual Machine (EVM) bytecode. Although some alternative programming languages (e.g., Vyper Team 2020, Bamboo Blockchain 2018, etc.) have been proposed, Solidity is still the most popular language in Ethereum. The methods used in analysis tools of Ethereum smart contracts can be roughly divided into dynamic analysis and static analysis.

Dynamic analysis refers to the use of techniques (e.g., theorem proving and fuzzy testing) to discover potential security threats during contract execution. The tools based on theorem proving (Grishchenko et al. 2018c; Hildenbrandt et al. 2018; Jiao et al. 2020) mainly use F*s framework and K framework to describe the properties of the contract, and then use theorem provers (e.g., Isabelle Nipkow et al. 2283) to determine whether its logic code is flawed. However, most of these tools require users to provide specifications or invariants manually, so they are difficult to achieve fully automatic analysis. The tools (Nguyen et al. 2020; Choi et al. 2021; He et al. 2019) leverage fuzzy testing to run contracts actually and discover vulnerabilities with pre-defined oracles. However, the oracles in these tools are challenging to define, and can only verify the limited vulnerabilities.

Compared with the above methods, static analysis is more comprehensive and efficient in implementing contract verification. According to different input types, static analysis tools can be divided into two categories that use EVM bytecode (e.g., Luu et al. 2016; Tsankov et al. 2018) and Solidity code (e.g., Tikhomirov et al. 2018; Kalra et al. 2018; Feist et al. 2019; Lu et al. 2019) as input. The tools (Luu et al. 2016; Krupp and Rossow 2018; Permenev et al. 2020; Frank et al. 2020; Software 2020) employ symbolic execution to build the Control Flow Graph (CFG) of the contract from the EVM bytecode, and then execute the predefined logic rules to identify vulnerabilities. Nevertheless, the cumbersome execution process of CFG and the unreadable bytecodes makes detection inefficient and rules difficult to formulate. Moreover, tools (Tsankov et al. 2018; Schneidewind et al. 2020; Chen et al. 2020) construct an intermediate representation (IR) by extracting the semantics of the EVM bytecode, and match the vulnerable instructions with the predefined security patterns. Since EVM bytecodes lack a lot of Solidity semantics, it is prone to false positives and false negatives when using these tools. For instance, most of the incorrect detection in Securify (Tsankov et al. 2018) are caused by semantic inconsistency between IR and Solidity code. Moreover, programmers will develop and revise contracts directly on the Solidity source code. Thus, analysis based on Solidity code is more efficient than EVM bytecode. As demonstrated by Durieux et al. (2020), Slither (Feist et al. 2019) and SmartCheck (Tikhomirov et al. 2018), as the two representatives of the tools based on Solidity analysis, comprehensively outperformed those based on EVM bytecode (e.g., Oyente Luu et al. 2016). They all abstract Solidity as IR, and utilize pattern matching to detect vulnerabilities. However, the IR used in Slither lacks some primitive semantics, such as var variables. Similarly, IR restrictions make tools such as SmartCheck (Tikhomirov et al. 2018) difficult to formulate vulnerability rules, resulting in a lot of false negatives and false positives. Thus, IRs and matching methods of these tools need to be improved.

**Key Challenges** To sum up, we design an advanced formal static analysis tool named SmartFast to address the following challenges.

(i) *Detect more kinds of vulnerabilities.* To the best of our knowledge, Slither (Feist et al. 2019) can detect 71 kinds of vulnerabilities. It can find the most vulnerabilities among SOTA tools. Nonetheless, by sorting out the vulnerabilities that have been reported so far, we found that there are at least 130 kinds of vulnerabilities in smart contracts.

(ii) *General severity assessment mechanism.* The severity of the same vulnerability varies from tools. For instance, a vulnerability called *unchecked-lowlevel* is classified as high-severity in SmartCheck (Tikhomirov et al. 2018), while it is classified as medium-severity in Slither (Feist et al. 2019). The inconsistency of the vulnerability severity evaluation mechanism in different tools will confuse the Ethereum security managers to deploy inappropriate security policies. In order to circumvent this problem, we propose a unified severity assessment mechanism in terms of risk degrees and utilization difficulties.

(iii) *More accurate and robust intermediate representation.* Most of the tools analyze the vulnerabilities based on their respective IRs. Since most IRs (e.g., SlithIR) are difficult to comprehensively and accurately express the original contract semantics, it will lead to false positives and false negatives. Moreover, most IRs based on the abstract syntax tree (AST), generated by contract compilation. However, AST will be lost when the contract cannot be compiled. As a result, IRs cannot be generated correctly. In order to address this issue, SmartFast parses the source code directly so that it can standardize the contract code well. More importantly, it can analyze smart contracts even if they are compiled improperly.

(iv) *More accurate and efficient pattern matching.* In the current security analysis tools (especially static analysis tools), the effect of tool detection is closely related to the accuracy of the matching rules. On the one hand, the existing tools have limited and inaccurate security rules, which prevent them from comprehensively discovering vulnerabilities. On the other hand, the complex matching algorithm will make the detection inefficient. For example, Securify (Tsankov et al. 2018) takes about 4 minutes to analyze a 121KB contract, which is 24 times that of Slither (Feist et al. 2019). In contrast, SmartFast adopts precise matching rules and efficient matching algorithms to analyze a contract in seconds.

(v) *Fully automated contract analysis.* In the tools developed by Grishchenko et al. (2018c) and Hildenbrandt et al. (2018), manual participation reduces the efficiency of contract analysis and causes inconvenience to users. Instead, SmartFast leverages the pre-defined patterns and automation engine to realize the fully automated contract analysis, which not only saves time and assets, but also alleviates human errors and malicious attacks.

**Contributions** The primary contribution of this paper is the design, implementation and evaluation of SmartFast. More specifically, we make the following contributions:

– *Universal vulnerability severity assessment.* We propose a unified vulnerability assessment model (cf. Section 3). This model divides the severity of vulnerabilities into five levels: *High*, *Medium*, *Low*, *Informational*, *Optimization*, in terms of hazard degree and utilization difficulty. In Section 3, we elaborate on its evaluation criteria, and leverage it to evaluate 136 kinds of vulnerabilities detected by 8 tools.

- *Strong vulnerability discovery capability.* We design and develop SmartFast (cf. Section 4), which is mainly composed of SmartIR and pattern verification. SmartIR provides three forms, namely XML, IR, and IR-SSA, with a more accurate and robust expression of contract semantics than SOTA tools. Based on SmartIR, we build a corresponding pattern matching framework to achieve efficient and accurate security analysis on smart contracts.
- *Superior detection performance.* We evaluate the accuracy, efficiency, and robustness of SmartFast on three datasets (13,687 contracts in total) (cf. Section 6). Compared with SOTA tools (e.g., Slither and SmartCheck), SmartFast can detect contract vulnerabilities more accurately (119 kinds, 98.43% precision rate, and 85.12% recall rate), efficiently (11.5 seconds per 121KB contract), and robustly (0.4% failed rate and resistance to the general code confusion methods). Moreover, it can discover the vulnerability codes accurately in major vulnerability incident contracts and investigate that 94% of the contracts in the Ethereum dataset can be optimized. In addition, we explore the deep insights of the correctness and effectiveness of SmartFast from the theoretical level (cf. Section 7).

## 2 Preliminaries and Motivating Examples

This section details the basic knowledge of SmartFast and 13 kinds of representative contract vulnerabilities.

### 2.1 Smart Contracts in Ethereum

Smart contracts are codes running on the blockchain, known as the “autonomous agents” of the blockchain. When a user creates a contract $A$ successfully, the blockchain will generate a corresponding identification address $\alpha_A$ (called contract address) for the contract. Moreover, the contract holds a certain amount of virtual currency Ether (called balance) and associates it with executable code. Contract code can operate on variables like traditional imperative programs. In comparison, the execution of the contract requires gas, which can be converted with Ether and sent to the miners as a reward. Ethereum virtual machine (EVM) bytecode, as the underlying code of Ethereum contract operation, is a stack-based language. Users use high-level programming languages (e.g., Solidity Foundation 2020) to develop smart contracts. Then, these contracts are compiled into EVM bytecode by compilers such as Solc. In order to invoke the contract $A$ at address $\alpha_B$, the user needs to send transaction $T = \langle \alpha_B, \alpha_A, E, G, D, \ldots \rangle$ to contract address $\alpha_A$, where $E$, $D$, $G$ represent the input amount, call parameters, and execution cost, respectively.

### 2.2 Intermediate Representation and Pattern Detection

Compared with contract source code, **Intermediate Representation (IR)** is a closer representation to executable code. It is generated by a converter based on high-level languages such as Solidity. Since the complexity of the contract Solidity code, it is infeasible to conduct static analysis on it directly. To this end, IRs make static analysis algorithms concise and efficient. For instance, based on Extensible Markup Language (XML), the corresponding source code can be expressed by defining tag types. It simplifies the analysis effort and makes the contract logic clearer. Similarly, languages such as Scilla (Sergey et al. 2018) and LLVM (Kalra et al. 2018) can be available as IRs.

**Static Single Assignment (SSA)** is a unique conversion format for IR that is frequently used in static analysis. Figure 1 shows an example of converting IR to SSA. In SSA, when a state variable $x$ is assigned, it is expressed as $x_1$ and $x_2$. In other words, SSA requires that each variable can only be assigned once. This makes the transitive relation between variables more evident. Thus, SSA can help us build data dependencies effectively. Moreover, there is an ambiguity when IR faces multiple paths. For example, in the last procedure of Fig. 1, it cannot be determined whether $y_3$ refers to $y_1$ or $y_2$. If $y_3$ refers to $y_1$, then $z_1 = x_2 + y_1$; otherwise, then $z_1 = x_2 + y_2$. To eliminate the uncertainty, SSA introduces a statement $\phi$ to select the corresponding variable version according to the running path. Then the obtained variable is defined as $y_3 = \phi(y_1, y_2)$. In other words, $\phi$ can indicate that a variable has multiple potential definitions. Also, at the beginning of the function, the state variable $x$ is set to its initial value that can be updated by executing any function. Therefore, a statement $\phi$ needs to be placed at the function entrance to select the version of each state variable read by the function.

**Pattern Detection** Based on IRs, detection algorithms (called patterns) are designed to identify vulnerable codes. It usually employs brute force search and taint analysis to analyze IRs of contracts. For example, XML Path Language (XPath) is the detection method of XML. It traverses the elements and attributes in XML to match the desired information. Furthermore, we need to combine the features of IR and vulnerabilities to develop corresponding detection rules (security patterns). Then vulnerabilities can be verified by matching the corresponding patterns in the IRs.

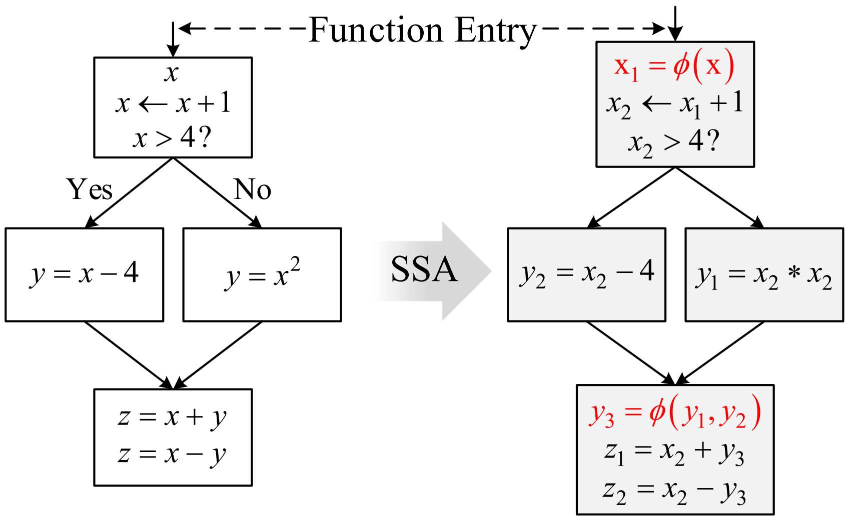


**Fig. 1** Example of IR to SSA conversion

### 2.3 Adversary Tools

In this paper, we will compare the various performance (e.g., accuracy, efficiency, etc.) between SmartFast and the following tools having open-source code. (i) *Oyente* (Luu et al. 2016) developed by Melonport combines with symbolic execution technology. It builds the CFG of the contract from EVM bytecodes and uses pre-defined logical rules to find potential problems in the contract. (ii) *Osiris* (Torres et al. 2018), improved based on Oyente, expends the detection of integer bugs. (iii) *Mythril* (Software 2020) developed by ConsenSys, incorporates concept analysis, taint analysis, and control flow inspection. (iv) *Securify* (Tsankov et al. 2018) developed by SRI System Laboratory (ETH Zurich), uses semantic facts and predefined patterns based on the bytecodes to identify contract vulnerabilities. (v) *Securify2.0* (SRI Lab 2020) developed by ETHZurich, takes the Solidity source code as input. It improves IR based on Securify so that it can support more vulnerabilities detection. (vi) *Slither* (Feist et al. 2019) is a static analysis framework developed by Trail of Bits. It uses SlithIR and pre-defined rules to match the problematic codes. (vii) *SmartCheck* (Tikhomirov et al. 2018) developed by SmartDec, determines contract vulnerabilities is similar to *Slither*.

### 2.4 Motivating Examples

In this section, we combine the contract example code to explain the vulnerabilities in terms of occurrence principle, impact effect, and repair countermeasures.

**Reentrancy with Ether (*reentrancy-eth*)** Reentrancy vulnerability is a classical problem, which leads to the asset loss of nearly $60 million in 2016 (Sergey and Hobor 2017). This vulnerability refers to reentry with the following features: reentrant call, Ether sending, and reading the variables before writing. An attack scenario is depicted in Listings 1 and 2. Bob first constructs a contract *Attack*, and then he performs the function “withdraw()” by invoking the attack(), which will trigger the fallback function. By this means, Bob implements multiple calls to withdraw(). Since the userBalance hadn’t changed before the call in withdraw(), Bob obtained more than the amount he deposited into the contract. It should be noted that the Ether sent cannot be zero. Otherwise, it will cause false positives. **Improvements to contracts:** put userBalance[msg.sender]=0 before the call function. That is, the contracts should employ the check-effects-interactions pattern to avoid this vulnerability.

**Right-to-Left-Override (*rtlo*)** This vulnerability can manipulate the logic of the contract by using a right-to-left-override character (U+202E). As shown in Listing 3, the contract *Token* uses the right-to-left-override character when invoking _withdraw() function. As a result, the fee is incorrectly sent to msg.sender, and the token is sent to the owner. **Improvements to contracts:** Remove the special control characters (i.e., U+202E) in the contract.

```
contract PullPayment {
  mapping (address => uint) userBalances;
  function withdraw(){
    //Reenter the function
    if(!(msg.sender.call.value(userBalance[msg.sender])())){ throw; }
    userBalance[msg.sender] = 0;
  }
}
```

**Listing 1** The sample of *reentrancy-eth*

**Listing 2** Attack contract

```
contract Attack {
  PullPayment object;
  function attack() payable {
    object.withdraw(1 ether);
  }
  function() public payable {
    object.withdraw(1 ether);
  }
}
```

**Lock Account Assets (*locked-ether*)** As shown in Listing 4, since the recharge() has a tag "payable" (i.e., supports receiving remittance), everyone can transfer amounts to the contract *Locked* through this function. However, *Locked* doesn't provide any functions with withdrawal power. Thus, every Ether sent to *Locked* will be lost. **Improvements to contracts:** Remove the payable attribute or add a function "withdraw" in the contract.

**Transaction Origin Address (*tx-origin*)** As the underlying property of the transaction, the origin address may be manipulated by the attacker, so that it is inappropriate to be used for authentication. An attack scenario is depicted in Listing 5. Bob is the owner of the contract *TxOrigin*. Bob calls Eve's contract. Eve's contract invokes *TxOrigin* and bypasses the tx.origin protection. Thus, the modifier "verify()" will lose its verification effect, making the contract abnormal. **Improvements to contracts:** Don't use tx.origin for authentication.

**Wrong Shift Parameters (*shift-parameter-mixup*)** The opcode shr(a,b) indicates that b is shifted right by a bits. However, the parameter errors for this opcode will get unexpected results. As shown in Listing 6, the shift statement right-shifts the constant 8 by b bits due to the opposite position of the parameters. Then, the function "f()" returns an incorrect value, which may cause unexpected issues. **Improvements to contracts:** Swap the order of parameters to shr(8,b).

**Shadowed Built-in Elements (*shadowing-builtin*)** Solidity enables the shadowing of most elements in the contract, such as variables and functions. The shadowed elements may not be invoked as the user wishes and cause the wrong results. This vulnerability indicates that the names of the elements (e.g., state variables and custom functions) conflict with the built-in symbols (e.g., "assert", "now", "sha3"). Moreover, "try", "case" and other reserved keywords are not recommended when defining the element name. Listing 7 shows an example of this vulnerability. Since the state variable and the time variable "now" have the same name, the current time will not be obtained when function "get_random()" is invoked. Moreover, the built-in symbol "require" is hidden by the function defined by the contract, which may cause the invalid authentication in get_random(). In short, these shadowed built-in elements lead to unexpected results. **Improvements to contracts:** Modify or delete declared elements such as "now" and "require(bool)".

**Listing 3** The sample of *rtlo*

```
contract Token {
  address payable o; // owner
  mapping(address => uint) tokens;
  function withdraw() public {
    uint amount = tokens[msg.sender];
    address payable d = msg.sender;
    tokens[msg.sender] = 0;
    _withdraw(/*owner*/o ,d /*destination*/
      /*value*/, amount);
  }
}
```

**Listing 4** The sample of *locked-ether*

```
contract Locked{
  mapping(address => uint) tokens;
  function recharge() payable public{
    tokens[msg.sender] += msg.value;
  }
}
```

**Listing 5** The sample of *tx-origin*

```
contract TxOrigin {
  address owner = msg.sender;
  modifier verify() {
    require(tx.origin == owner);
    _;
  }
}
```

**Listing 6** The sample of *shift-parameter-mixup*

```
contract C {
  function f() internal returns (uint b) {
    assembly {
      b := shr(b, 8)
    }
  }
}
```

**Listing 7** The sample of *shadowing-builtin*

```
contract ShadowedSC {
  uint now; //shadowing built-in symbols
  function get_random() private returns (uint) {
    require(owner == msg.sender); //invalidation
    return now + 259200; //unexpected result
  }
  function require(bool condition) public {
    ...;
  }
}
```

```
function transfer(bytes _signature,address _to,uint256 _value,uint256
     _gas,uint256 _nonce) public returns (bool){
  bytes32 txid = keccak256(abi.encodePacked(getTransferHash(_to, _value,
      _gas, _nonce), _signature));
  require(!signatureUsed[txid]);
  address from_address = recoverTransferPreSigned(_signature, _to, _value
      , _gasPrice, _nonce);
  balances[from_address] -= _value;
  balances[_to] += _value;
  signatureUsed[txid] = true;
}
```

**Listing 8** The sample of *signature-malleability*

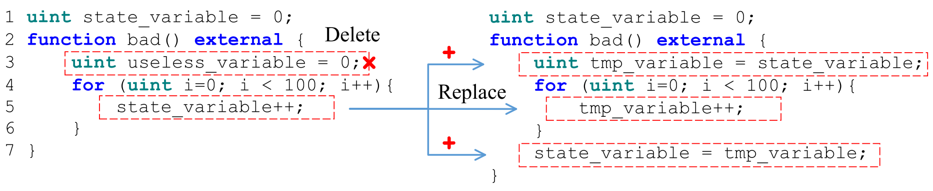
```
1   contract Timestamp{
2     uint time_now = 1577808000;
3     address private receiver;
4     function frangibility() public {
5       require(block.timestamp > time_now);
6       uint 1_ther = 10000000000000000000;
7       receiver = msg.sender;
8       require(receiver.call.value(1_ther).gas(7777)(""));
9     }
10  }
```

**Listing 9** The samples of *timestamp*, *low-level-calls*, *private-not-hidedata*, etc

**Non-compliant Signature (*signature-malleability*)** The use of signatures should follow the norms. Otherwise, it will cause unexpected impacts. Listing 8 shows an incorrect signature example. That is, keccak256 contains the existent signature "_signature". The attackers can modify the elements r\s\v in _signature to construct a new valid signature, which can get a valid address from the function "recoverTransferPreSigned". As _signature changes, the verification (line 3) will be passed, thereby allowing the attackers to get additional balance. **Improvements to contracts:** Remove _signature in keccak256.

**Timestamp Dependency (*timestamp*)** The object "block" contains many attributes (e.g., timestamp and block number), which can be manipulated by miners and nodes. As shown in Listing 9, attackers can control the block.timestamp to pass the verification (line 5) by conspiring with miners or nodes. Also, the contract owner confuses users with complex numbers (***too-many-digits***) and guided names, where the variable 1_ether is actually 10 ether. Furthermore, the function "frangibility()" uses this variable and the calls with poor safety (***low-level-calls***), which may cause security issues such as *reentrancy-eth*. In addition, although the visibility of the state variable "receiver" is declared private, miners can still be viewed in advance through the transaction (***private-not-hidedata***). **Improvements to contracts:** Use the scientific notation to modify the statement to 10_ether = 10**19. Moreover, the calls with low-level security and block attributes such as timestamp should be avoided. Also, it is recommended to use private data in the ciphertext.

**Useless Code (*code-no-effects*)** The execution of each operation in the contract will consume the specified gas. However, as shown in Fig. 2, useless_variable is clarified and without any use, which makes the gas of the function "bad()" wasted. Moreover, state variables consume more gas than local variables, so frequent manipulation of state_variable in the loop consumes additional gas (***costly-operations-loop***). **Improvements to contracts:** As shown in Fig. 2, the useless variable was removed. Also, the local variable "tmp_variable" was updated and then assigned to the state variable.



```
uint state_variable = 0;
function bad() external {
    uint useless_variable = 0;
    for (uint i=0; i < 100; i++){
        state_variable++;
    }
}
```

```
uint state_variable = 0;
function bad() external {
    uint tmp_variable = state_variable;
    for (uint i=0; i < 100; i++){
        tmp_variable++;
    }
    state_variable = tmp_variable;
}
```

**Fig. 2** The samples of *code-no-effects* and *costly-operations-loop*

## 3 Vulnerability Assessment Model

By explaining the above 13 vulnerabilities, it is known that these vulnerabilities have a variety of impacts and exploitability, thus giving the contract varying security. It is necessary to give priority to repairing vulnerabilities with greater severity. For example, in a limited time, repairing a vulnerability with more significant severity can quickly improve the security of the contract. However, the SOTA tools (e.g., Slither and SmartCheck) didn't give a well-founded assessment mechanism for the severity of the vulnerability. To this end, we propose a vulnerability assessment model in this section. Furthermore, this section summarizes and compares the kinds of vulnerabilities detected by different SOTA tools based on this assessment model.

### 3.1 Vulnerability Assessment Method

Combined with CVSS2.0 (Common Vulnerability Scoring System), the vulnerability severity of smart contract can be rated as *High*, *Medium*, and *Low* in terms of risk degrees and utilization difficulties. The detailed partitioning is shown in Fig. 3. The risk degree refers to the impact of vulnerability on the resources such as blockchain system and users. According to the three impact dimensions of confidentiality (C), integrity (I), and availability (A), this paper divides the risk degree into *High*, *Medium*, *Low*, *Informational* (*Info*) and *Optimization* (*Opt*). Specifically, *High* risk refers to the severe and almost irreversible harm, i.e., the vulnerability can seriously affect the CIA of smart contracts, and cause a lot of economic losses and data confusion to the contract business system. Including but not limited to: (i) the large assets being stolen or frozen; (ii) the core contract business cannot operate normally, such as denial of service; (iii) the core business logic of contracts is arbitrarily tampered with or bypassed, such as transfer, charging, and accounting; (iv) the fairness design of contracts is invalid, such as electronic voting, lottery, and auction.

*Medium* risk refers to a slight impact on the CIA of smart contracts, and may cause certain harm to the contract business system, such as little economic losses. Including but not limited to: (i) some assets being stolen or frozen; (ii) the non-core business logic of contracts is destroyed; (iii) the non-core business verification of contracts is bypassed; (iv) the contracts trigger error events or adopt non-standard interfaces, resulting in loss of the external system.

*Low* risk refers to a weak impact on the contract business system. Including but not limited to: (i) the stability of contract operation is affected, such as the abnormal increase in call failure rate and resource consumption; (ii) the contracts adopt substandard interfaces

| Risk / Utilization | High | Medium | Low | Informational | Optimization |
|---|---|---|---|---|---|
| Exactly | High | Medium | Low | Informational | Optimization |
| Probably | High | Medium | Low | Informational | Optimization |
| Possibly | Medium | Low | Low | Informational | Optimization |

**Fig. 3** Vulnerability assessment model

or their implementation, which affects the security and compatibility of interfaces; (iii) the contracts can trigger false events with few losses.

*Info* risk refers to hardly substantial harm to the contract business system and reminds contract developers that the contract code is prone to errors. Thus, the contract owners should develop the contracts following the specifications. Including but not limited to: (i) the sensitive function calls such as delegatecall; (ii) the precautions required by the security development specifications, such as variables and functions updated in version 0.5.0.

*Opt* risk refers to the improvement of the contracts, which can make the contract more efficient, readable, and less gas consumption. Including but not limited to: (i) remove useless operations and reduce operating overhead; (ii) optimize algorithm code to improve the running speed and security.

The utilization difficulty refers to the possibility of vulnerability occurrence. According to the three dimensions of attack cost (e.g., money, time, and technology), utilization condition (i.e., the difficulty of attack utilization), and trigger probability (e.g., vulnerabilities can only be triggered by a few users), this paper divides it into *exactly*, *probably*, and *possibly*. Generally, *exactly* utilization requires an inferior cost, and the vulnerabilities can be stably triggered without a special threshold. Including but not limited to: (i) easy to invoke; (ii) need few costs; (iii) hold few assets.

*Probably* utilization requires a certain cost and utilization conditions, and the vulnerabilities are not easy to trigger. Including but not limited to: (i) pay a certain cost but less than attack proceeds; (ii) require attackers to achieve certain normal conditions, such as collusion with miners or outbound nodes; (iii) cooperate with known attacks in smart contracts, such as attacking other on-chain Oracle contracts.

*Possibly* utilization requires expensive costs and strict utilization conditions, and the vulnerabilities are more difficult to trigger. Including but not limited to: (i) pay the cost that more than the attack proceeds; (ii) require the attackers to meet low-frequency conditions, such as belonging to a few critical accounts, and constructing a difficult specific signature.

### 3.2 Examples of Vulnerability Assessment

The model is used to assess the vulnerability examples in Section 2.4, so as to further illustrate the details of the model. Table 1 details the assessment results, indicating that these vulnerabilities have different risk degrees and utilization difficulties (i.e., there are 13 combinations). Thus, they are given various severity. Specifically, the *rtlo* vulnerability can destroy the core business logic and the fair design (*High* risk); it is well-characterized and can be performed deterministically (*exactly* utilization) after the contract deployment.

The *reentrancy-eth* vulnerability can cause massive assets overspent or stolen (*High*); some conditions are required to trigger this vulnerability (*probably*). For instance, completing the attack requires auxiliary contracts.

The *locked-ether* vulnerability can make little assets be lost or frozen (*Medium*); it can be triggered stably after contract deployment (*exactly*).

The *tx-origin* vulnerability can cause the non-core business verification to be bypassed without direct economic losses (*Medium*); it requires a combination of ancillary contracts to obtain transactions from the verified party (*probably*).

Similar to *locked-ether*, the *shift-parameter-mixup* vulnerability can destroy the non-core business logic (*Medium*). Nonetheless, it requires that users are unclear about the parameters of shr(), thus giving it an inferior trigger probability (*possibly*).

**Table 1** The severity details of vulnerability examples supported by SmartFast

| Vulnerability | Description | Risk | Utilization | Severity |
|---|---|---|---|---|
| rtlo | Right-To-Left-Override control character is used. | High | exactly | High |
| reentrancy-eth | Ether stolen illegally by re-entering functions such as calls. | High | probably | High |
| locked-ether | The contract has a payment function but without withdrawal capacity. | Medium | exactly | Medium |
| tx-origin | Verification based on tx.origin may be bypassed. | Medium | probably | Medium |
| shift-parameter-mixup | The parameter errors of shift operation cause the opposite results. | Medium | possibly | Low |
| shadowing-builtin | The names of elements (e.g., state variables) conflict with the built-in symbols. | Low | exactly | Low |
| timestamp | Miners or nodes can manipulate block.timestamp to achieve their purpose. | Low | probably | Low |
| signature-malleability | The signature to be verified contains an existing signature. | Low | possibly | Low |
| low-level-calls | Low-level functions such as call and delegatecall are used in the contract. | Info | exactly | Info |
| too-many-digits | Complex number symbols will confuse contract users. | Info | probably | Info |
| private-not-hidedata | Private visibility doesn't guarantee data confidentiality. | Info | possibly | Info |
| code-no-effects | The useless code will increase gas consumption during running. | Opt | exactly | Opt |
| costly-operations-loop | Redundant operations in the loop will waste resources such as gas. | Opt | probably | Opt |

The *shadowing-builtin* vulnerability can introduce security risks such as unexpected results and verification failure, so as to affect the stability of the contract operation (*Low*); it is triggered by executing the vulnerability code after the contract deployment (*exactly*).

The *timestamp* vulnerability can introduce security risks such as verification failure and random number utilization (*Low*); it requires attackers to collude with miners or block nodes (*probably*).

The *signature-malleability* vulnerability can cause a signature replay attack that affects the stability of contract operation (*Low*). However, the eligible signature is difficult to be constructed and exploit (*possibly*).

The *low-level-calls* vulnerability is prone to cause contract errors, so it is necessary to focus on the use of these functions (*Info*); it can be invoked by executing the vulnerable code (*exactly*).

The *too-many-digits* vulnerability can make users challenging to read the contracts, and inaccurate variable names will mislead them (*Info*); it requires users to misunderstand the semantic information of the variable, thus giving it a weak trigger probability (*probably*).

The *private-not-hidedata* vulnerability can result in the disclosure of private data, thus reminding us that private content should be stored in ciphertext (*Info*); it requires attackers to meet the specific identity restrictions and combine with miners to get information (*possibly*).

The *code-no-effects* vulnerability will cause useless gas consumption, which can be further optimized for user convenience (*Opt*); it can be improved after the contract development (*exactly*).

Similar to *code-no-effects*, the *costly-operations-loop* vulnerability can be further optimized to reduce the consumed gas (*Opt*); it depends on the number of loops, thus giving it a certain trigger probability (*probably*).

The above 13 kinds of vulnerabilities cover all combinations of risk degree and utilization difficulties. Thus, the assessment model describes the severity of vulnerabilities in a detailed and clear manner, which can help users understand the security of the contract. In addition, we have collected 136 kinds of vulnerabilities so far, and the complete assessment results are shown in Table 2.[1] Among them, the number of the vulnerability severity is distributed as *High* (28), *Medium* (41), *Low* (26), *Info* (27), and *Opt* (14). Although half of the vulnerabilities with weak severity (i.e., *Low*, *Info*, and *Opt*), it is still necessary to detect them considering the comprehensive requirements, which can help users develop secure and specification-compliant contracts. Moreover, since vulnerabilities such as *redundant-fallback* refer to multiple scenarios, they have multiple levels of severity. For example, the *visibility* vulnerability indicates that errors are induced by function visibility. That is, the default function visibility (i.e., public) tends to cause critical functions to be exploited by attackers (*Info*), so these functions need to be concerned. However, the version 0.5.0 of Solidity has explicit requirements for function visibility, and functions without marked visibility will result in a compilation failure (*High*). In conclusion, the assessment results are detailed and comprehensive.

It is noted that some vulnerabilities are related in Table 2. For instance, *suicidal* and *arbitrary-send* vulnerabilities have similar principles (i.e., permission issues). Also, *reentrancy-no-eth* is a variant of *reentrancy-eth*. Toward exploring the association between the vulnerabilities, we classified them into 14 categories, such as Access Control and Arithmetic, according to the their characteristics (e.g., call mode, variable type, risk degree). Table 3 details the vulnerability categories and their vulnerability composition. For example, the reentrancy category consists of 7 vulnerabilities such as *reentrancy-eth*. It has holds the following five features: reentrant function type (i.e., send\transfer function with a limit of 2300 gas, reentrant call function such as low_level_call without gas restriction), Ether sending, reading data before invoking the function, changing data after invoking the function, and triggering event after invoking the function. Moreover, each category includes vulnerabilities with multiple levels of severity. For example, the severity of the access control category is distributed as *High* (5), *Medium* (2), *Low* (3), and *Info* (1). In the real world, vulnerabilities with clear severity can allow people to understand contract security better, while convenient for repairing the vulnerabilities. To this end, this paper conducts detection and comparison around the 136 kinds of vulnerabilities listed in Table 2.

## 3.3 Performance of the SOTA Tools in Detecting Vulnerabilities

In order to evaluate the ability of SOTA tools to detect vulnerabilities, we compare the situation of vulnerabilities detected by SmartFast and tools such as Slither (described in Section 2.3) based on the above vulnerability severity. The vulnerabilities detected by each tool are summarized in Table 4. From the table, the same vulnerability may be detected

[1] The vulnerabilities are detailed in https://github.com/SmartContractTools/SmartFast/blob/main/VulnerabilityDescription.xlsx.

**Table 2** The list of vulnerabilities (security patterns). Among them, 62-63 are combined into integer overflow/underflow in tools such as SmartFast

| ID | Vulnerability&Pattearn | Severity | ID | Vulnerability&Pattearn | Severity | ID | Vulnerability&Pattearn | Severity |
|---|---|---|---|---|---|---|---|---|
| 1 | abiencoderv2-array | High | 47 | constant-function-state | Medium | 93 | multiple-calls-in-transaction | Low |
| 2 | array-by-reference | High | 48 | divide-before-multiply | Medium | 94 | repeat-call | Low |
| 3 | multiple-constructors | High | 49 | erc20-approve | Medium | 95 | missing-input-validation | Info |
| 4 | names-reused | High | 50 | function-problem | Medium | 96 | assembly | Info |
| 5 | public-mappings-nested | High | 51 | mul-var-len-arguments | Medium | 97 | assert-state-change | Info |
| 6 | rtlo | High | 52 | reentrancy-no-eth | Medium | 98 | delete-dynamic-arrays | Info |
| 7 | shadowing-state | High | 53 | reused-constructor | Medium | 99 | deprecated-standards | Info |
| 8 | suicidal | High | 54 | tx-origin | Medium | 100 | erc20-indexed | Info |
| 9 | uninitialized-state | High | 55 | typographical-error | Medium | 101 | erc20-throw | Info |
| 10 | uninitialized-storage | High | 56 | unchecked-lowlevel | Medium | 102 | length-manipulation | Info |
| 11 | unprotected-upgrade | High | 57 | unchecked-send | Medium | 103 | low-level-calls | Info |
| 12 | redundant-fallback | High/Opt | 58 | uninitialized-local | Medium | 104 | msgvalue-equals-zero | Info |
| 13 | arbitrary-send | High | 59 | unused-return | Medium | 105 | naming-convention | Info |
| 14 | continue-in-loop | High | 60 | writeto-arbitrarystorage | Medium | 106 | pragma | Info |
| 15 | controlled-array-length | High | 61 | callstack | Medium | 107 | solc-version | Info |
| 16 | controlled-delegatecall | High | 62 | integer-underflow | Medium | 108 | unimplemented-functions | Info |
| 17 | incorrect-constructor | High | 63 | integer-overflow | Medium | 109 | upgrade-050 | Info |
| 18 | parity-multisig-bug | High | 64 | truncation | Medium | 110 | function-init-state | Info |
| 19 | reentrancy-eth | High | 65 | signedness | Medium | 111 | complex-function | Info |
| 20 | storage-array | High | 66 | division | Medium | 112 | hardcoded | Info |
| 21 | weak-prng | High | 67 | modulo | Medium | 113 | overpowered-role | Info |
| 22 | solidity-dos-with-throw | High | 68 | Exception-State | Medium | 114 | reentrancy-limited-events | Info |
| 23 | TOD-receiver | High | 69 | costly-loop | Low | 115 | reentrancy-limited-gas | Info |
| 24 | TOD-ether | High | 70 | shift-parameter-mixup | Low | 116 | reentrancy-limited-gas-no-eth | Info |
| 25 | taint-pass-project | High | 71 | shadowing-builtin | Low | 117 | similar-names | Info |

**Table 2** (continued)

| ID | Vulnerability&Pattearn | Severity | ID | Vulnerability&Pattearn | Severity | ID | Vulnerability&Pattearn | Severity |
|---|---|---|---|---|---|---|---|---|
| 26 | money-concurrency | High | 72 | shadowing-function | Low | 118 | too-many-digits | Info |
| 27 | jump-arbitrary-instruction | High | 73 | shadowing-local | Low | 119 | private-not-hidedata | Info |
| 28 | assert-violation | Medium | 74 | transfer-to-zeroaddress | Low | 120 | safemath | Info |
| 29 | constructor-return | Medium | 75 | uninitialized-fptr-cst | Low | 121 | visibility | Info/High |
| 30 | default-return-value | Medium | 76 | variable-scope | Low | 122 | state-default-visibility | Info |
| 31 | enum-conversion | Medium | 77 | void-cst | Low | 123 | array-instead-bytes | Opt |
| 32 | erc1155-interface | Medium | 78 | incorrect-modifier | Low | 124 | boolean-equal | Opt |
| 33 | erc1410-interface | Medium | 79 | assemblycall-rewrite | Low | 125 | code-no-effects | Opt |
| 34 | erc20-interface | Medium | 80 | block-other-parameters | Low | 126 | constable-states | Opt |
| 35 | erc223-interface | Medium | 81 | calls-loop | Low | 127 | event-before-revert | Opt |
| 36 | erc621-interface | Medium | 82 | events-access | Low | 128 | external-function | Opt |
| 37 | erc721-interface | Medium | 83 | events-maths | Low | 129 | extra-gas-inloops | Opt |
| 38 | erc777-interface | Medium | 84 | extcodesize-invoke | Low | 130 | missing-inheritance | Opt |
| 39 | erc875-interface | Medium | 85 | fallback-outofgas | Low | 131 | redundant-statements | Opt |
| 40 | incorrect-equality | Medium | 86 | incorrect-blockhash | Low | 132 | return-struct | Opt |
| 41 | incorrect-signature | Medium | 87 | incorrect-inheritance-order | Low | 133 | revert-require | Opt |
| 42 | locked-ether | Medium | 88 | missing-zero-check | Low | 134 | send-transfer | Opt |
| 43 | mapping-deletion | Medium | 89 | reentrancy-benign | Low | 135 | unused-state | Opt |
| 44 | shadowing-abstract | Medium | 90 | reentrancy-events | Low | 136 | costly-operations-loop | Opt |
| 45 | tautology | Medium | 91 | timestamp | Low | | | |
| 46 | boolean-cst | Medium | 92 | signature-malleability | Low | | | |

**Table 3** The details of vulnerability (security pattern) categories. Among them, the vulnerability number refers to Table 2

| Category | Description | Vulnerability number |
|---|---|---|
| Access Control | Since objects (such as contracts and functions) hold unrestricted permissions, attackers can illegal invoke them. | 8,11,13,18,27,54, 60,82,84,95,113 |
| Arithmetic | Inconsistent behavior due to arithmetical errors such as the integer overflow and underflow. | 48,55,62–67,83 |
| Block parameter dependence | Using block-related variables that miners can manipulate effortlessly. | 21,80,91 |
| Denial of service | The contract is overwhelmed with time-consuming computations. | 22,81,93 |
| Front running | Two dependent transactions that invoke the same contract are included in one block. | 23–24,26,49 |
| Reentrancy | The calls of reentrant functions make contracts operate in unexpected ways. | 19,52,89–90,114–116 |
| Unchecked calls/ parameters | Ignore the check for the calls, including the return value and their parameters. | 15–16,25,56–57,74, 88,92,103–104,112 |
| Infrastructure errors | Inconsistent behavior due to errors of infrastructure (e.g., compiler, library function). | 1,5,20,31,51,70,86 |
| Dangerous variables/ attributes | Inconsistent behavior due to dangerous use of variables or their attributes (e.g., type and visibility). | 2,43,79,96–98,102, 110,122 |
| Deliberate violation | Inconsistent behavior due to the intentional violation of development specification. | 6,9–10,30,32–42, 45–46,58–59,75,94, 100–101,105,118–120 |
| Compile and runtime errors | Inconsistent behavior due to compilation and operation errors. | 3,14,17,28–29,47,50,53, 61,68–69, 76–78,85,99, 106–109,111,121 |
| Duplicate names | Inconsistent behavior due to duplicate names (e.g., variable, function, and contract). | 4,7,44,71–73,87,117 |
| Optimized operations | Operations can be optimized to improve running efficiency and reduce running gas, etc. | 12,123–136 |
| Unknown Unknows | Vulnerabilities not identified above. | – |

by multiple tools. For instance, tools such as Securify2.0 and Oyente can detect *timestamp* vulnerabilities (No. 91). Also, this table demonstrates the number of vulnerabilities detected by the tools at each severity. The number of vulnerabilities detected by pattern matching-based tools such as Slither and SmartCheck is more than that of symbolic execution-based tools such as Oyente ($71 > 44 \gg 8$). It can be attributed to the flexible usage of IRs and the convenient formulation of security patterns in pattern matching. Besides, tools such as Slither can identify vulnerabilities with *Info* and *Opt* severity, which can help contract developers to further improve contracts. In total, SmartFast can detect not only 22 *High-risk*

**Table 4** Comparison of vulnerabilities detected by tools

| Severity | SmartFast | SmartCheck | Slither | Securify | Securify2.0 | Oyente | Osiris | Mythril |
|---|---|---|---|---|---|---|---|---|
| High | 1–21,121 | 12,14,22 | 1–11,13,15–16, 19–21 | 19*2,23,26 | 6–9,16,19*2, 23–26 | 18–19,26 | 19,26 | 8,16,19,27 |
| Number | 22 | 3 | 17 | 4 | 11 | 3 | 2 | 4 |
| Medium | 28–60,63 | 28,30*2,40–42, (45, 47)*2, 48–49,54,56 | 31,34,37,40,42–48 (47*2),52–59 | 42,56–57,60 | 34,37,40,42,48, 52,54,56–60 | 28,61–63 | 28,61–67 | 54,56–57, 60,63,68 |
| Number | 34 | 14 | 20 | 4 | 12 | 4 | 8 | 6 |
| Low | 69–92 | 69,79,81,86,91 | 70–71,73,75–78, 81–83,88–91 | 94 | 71,73,77,81, 89,91,94 | 91 | 91 | 80,89,93 |
| Number | 24 | 5 | 14 | 1 | 7 | 1 | 1 | 3 |
| Info | 96–120 | 96,98–104(99*2), 107, 109,112–113, 119–121 | 96–97,100,103, 105–108,110,115, 117–118 | 95 | 95–96,100,103,105, 107,118,122 | – | – | – |
| Number | 25 | 16 | 13 | 1 | 8 | 0 | 0 | 0 |
| Opt | 123–136 | 123,128–129, 132–134 | 124,126,128,130–131, 135–136 | – | 126,128,135 | – | – | – |
| Number | 14 | 6 | 7 | 0 | 3 | 0 | 0 | 0 |
| Total Number | 119 | 44 | 71 | 10 | 41 | 8 | 11 | 13 |

vulnerabilities, 34 *Medium-risk* vulnerabilities, and 24 *Low-risk* vulnerabilities, but also 24 *Info* and 14 *Opt* problems, which are the most in each severity level.

*Remark 1* (Completeness of patterns) Compared with contract detection tools such as Slither and SmartCheck, SmartFast can detect more kinds of vulnerabilities and provide a more complete optimization direction. In other words, SmartFast can analyze the contract security comprehensively and assist users in developing secure smart contracts.

## 4 Design of SmartFast

In this section, we describe the main components of SmartFast in details.

### 4.1 Overview

Figure 4 shows the workflow of SmartFast, which can be divided into four stages.

- **Information extraction stage.** In this stage, semantic information such as the code execution sequence of the contract is extracted to construct feature maps such as program control flow graphs.
- **Language conversion stage.** This stage combines SmartIR's conversion syntax to formally describe the obtained feature maps, and finally obtains three forms of SmartIR in XML\IR\IR-SSA forms.
- **Module analysis stage.** Based on SmartIR, this stage summarizes information such as data dependencies and numerical operations, which can be invoked directly in the security patterns. For instance, if modifier "isowner()" wants to determine whether the owner is related to msg.sender in a security pattern, it can invoke data_denpendence(owner,msg.sender) directly. Thus, this reduces a lot of repetitive operations during security patterns development and pattern matching, making the detection more efficient.
- **Pattern detection stage.** In the light of SmartIR and vulnerability features, the security patterns are programmed in advance and employed as input for detection. In this stage, these security patterns are matched by XPath and command execution according to their forms (i.e., XML or IR/IR-SSA). Ultimately, these verification results will be aggregated and exploited to generate a detailed detection report.

In addition, the third-party tools can leverage the internal components of SmartFast to perform more advanced analysis. For example, running symbol execution method based on

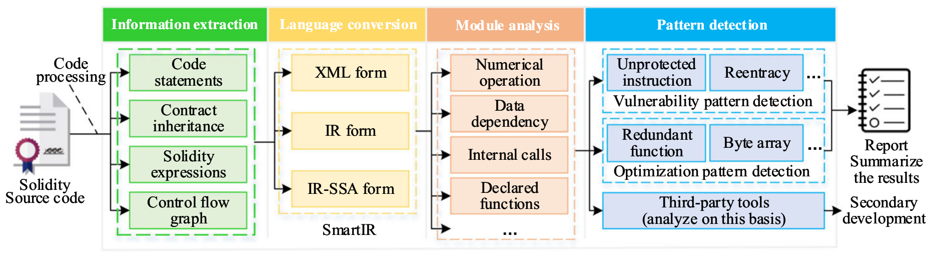


**Fig. 4** Workflow of SmartFast

SmartIR, or transforming SmartIR to IRs such as LLVM (IR of ZEUS) for the secondary development.

## 4.2 SmartIR

SmartIR is used to represent the Solidity source codes. It provides three forms of IR, namely XML, IR and IR-SSA.

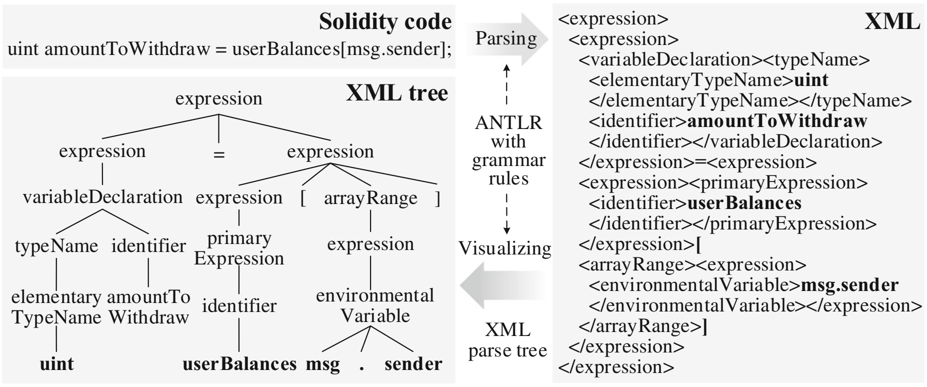
### 4.2.1 XML Form

In scenarios with Ether transfer, the balance corresponding to the address is usually inspected. Figure 5 shows the Solidity code and the corresponding XML form for checking the account balance. According to the grammar rules of Solidity, the grammar parser ANTLR parses the Solidity source code into an IR in XML format. In the example shown in Fig. 5, the expression is parsed into multiple parts based on the attributes of each field. For example, *uint* and *msg.sender* are parsed as *elementaryTypeName* and *environmentalVariable*, respectively. In order to define the matching rules conveniently, we leverage ANTLR to visualize the XML form and obtain the XML parse tree. In this tree, leaf nodes represent expression fields, and other nodes indicate represent the properties of fields.

### 4.2.2 IR Form

As the second form of SmartIR, the IR form includes more than 40 instructions, and can be obtained by converting the CFG constructed in the previous stage. The descriptions of some critical instructions are listed below.

1) Variables: According to the representation, contract variables are divided into existing variables (state variables, local variables, constants, Solidity variables, and tuple variables) and auxiliary variables (temporary variables and reference variables). Existing variables refer to variables directly used in the contract source code. Auxiliary variables represent intermediate operations or temporary conversion variables. Reference variables (REFVAL) are auxiliary variables used for mapping/index access. In addition, we use LVAL and RVAL to represent the left value and the right value in the expression, respectively. They can distinguish the allocated variables and the read variables.



```
uint amountToWithdraw = userBalances[msg.sender];
```

```
<expression>
  <expression>
    <variableDeclaration><typeName>
      <elementaryTypeName>uint
      </elementaryTypeName></typeName>
      <identifier>amountToWithdraw
      </identifier></variableDeclaration>
    </expression>=<expression>
    <expression><primaryExpression>
      <identifier>userBalances
      </identifier></primaryExpression>
    </expression>[
    <arrayRange><expression>
      <environmentalVariable>msg.sender
      </environmentalVariable></expression>
    </arrayRange>]
  </expression>
</expression>
```

**Fig. 5** The process of converting Solidity to XML form

2) Common operations: The operations can be divided into four types: assignment, arithmetic, index, and member. The assignment operation can be expressed as LVAL:=RVAL, where RVAL is generally a variable such as Tuple or Function (dynamic function assignment). Arithmetic operations are divided into unary operations and binary operations according to the variable number involved in the operation. The unary operation can be expressed as LVAL=(!,˜,...) RVAL, where ! is logical negation and ˜ is bitwise negation. Binary operations include 19 common operations such as addition and subtraction, which can be expressed as LVAL=RVAL (+, −) RVAL. Index operation is mainly employed for array and mapping to access the corresponding values, which can be expressed as REFVAL → LVAL [RVAL] (REFVAL points to the memory location). Member operations are used to access the internal data of structures such as struct, which can be expressed as REFVAL → (LVAL, CONTRACT, ENUM).RVAL.
3) Other operations: The remaining operations include create, push, etc. Create operation can be expressed as LVAL = NEW_ARR, where NEW_ARR is employed to create a new array. Push operation refers to adding variables to arrays or dynamic functions, which can be expressed as PUSH LVAL RVAL and PUSH LVAL Func. Delete operation represents deleting the corresponding element, expressed as Del LVAL. The type conversion operations are often used in contracts, such as string→uint256. To this end, we have introduced type conversion operations, which are expressed as CONVERT LVAL RVAL TYPE (uint, etc.). The Unpack operation is utilized to extract the corresponding elements in the tuple, expressed as LVAL=UNPACK TUPLEVARIABLE INDEX(:int). Condition operation refers to the conditions in loops and if operations. The array initialization operation (Arr Init) is introduced to judge the array initialization. Finally, the function can return values, generally empty (None), variables (RVAL), and tuples (TUPLE).
4) Call function operations: In addition, the program codes should be able to correctly invoke the functions in the contract. According to the security level and degree of influence, the operations of invoking function can be divided into two types HIGH_LEVEL_CALL and LOW_LEVEL_CALL. They can be expressed as LVAL=LEVEL_CALL DST Func (Params), where Params mainly includes GAS and VALUE. LOW_LEVEL_CALL refers to calls of functions with a greater impact, such as suicide and unprotected call functions for Ether transfers. On the contrary, HIGH_LEVEL_CALL refers to calls for high-security functions such as transfer. According to the owner of the function, we can divide the invoked function into IN_CALL and SOLIDITY_CALL, they can be expressed as CALL Func [Params]. IN_CALL mainly refers to calls of internal functions declared by the contract developers. When the function type is dynamic, it can be expressed as IN_DYN_CALL. SOLIDITY_CALL represents invoking the Solidity functions such as keccak256(). In addition, according to the properties of the function, the calls of functions can be divided into LIB_CALL, EVENT_CALL, SEND, and TRANSFER. LIB_CALL refers to calls of functions in the imported library, such as safemath (using safemath). EVENT_CALL is mainly used to record the execution status of the function, which can be expressed as EVENT_CALL Event (Params). Both SEND and TRANSFER are transfer operations (gas limit is 2,300), which can be expressed as (SEND, TRANSFER) DEST VALUE. However, TRANSFER automatically rolls back the state in case of an accident, so it is safer than SEND. It should be noted that the same function may have multiple categories, for example, suicide belongs to both LOW_LEVEL_CALL and SOLIDITY_CALL.

### 4.2.3 IR-SSA Form

As the third form of SmartIR, the IR-SSA form is the static single allocation (SSA) representation of IR. Figure 6 shows an example of a contract converted from Solidity source code to IR and IR-SSA. Solidity snippet calls the function *withdrawBalance()* to send the balance *userBalances[msg.sender]* to the address *msg.sender*. When the transfer is invoked successfully, the balance will be set to 0. Since the state variables are changed after the event occurs, and the call function without gas limit, the code satisfies the features of the reentry attack (refer to Section 2.4). Thus, this code has a reentry vulnerability. In order to detect the vulnerability, we first extract semantics such as call function for Ether transfer and balance resetting from contract source code, and then summarize information such as CFG. Based

**Solidity code**

```
mapping (address => uint) private userBalances;
function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    if (!(msg.sender.call.value(amountToWithdraw)())) { throw; }
    userBalances[msg.sender] = 0;
}
```

**IR**

```
Function withdrawBalance():
    REFVAL_3(uint256) -> userBalances[msg.sender]
    amountToWithdraw(uint256) := REFVAL_3(uint256)
    TEMP_2(bool) = LOW_LEVEL_CALL, dest:msg.sender, function:call,
parameters:[] value:amountToWithdraw
    TEMP_3 = ! TEMP_2
    CONDITION TEMP_3
    REFVAL_6(uint256) -> userBalances[msg.sender]
    REFVAL_6 (->userBalances) := 0(uint256)
```

**IR-SSA**

```
Function withdrawBalance():
    userBalances_4(mapping(address => uint256)) :=  ϕ (['userBalances_1',
'userBalances_5', 'userBalances_3', 'userBalances_0'])
    REFVAL_3(uint256) -> userBalances_4[msg.sender]
    amountToWithdraw_1(uint256) := REFVAL_3(uint256)
    TEMP_2(bool) = LOW_LEVEL_CALL, dest:msg.sender, function:call,
parameters:[] value:amountToWithdraw_1
    TEMP_3 = ! TEMP_2
    CONDITION TEMP_3
    REFVAL_6(uint256) -> userBalances_4[msg.sender]
    userBalances_5(mapping(address => uint256)) :=  ϕ (['userBalances_4'])
    REFVAL_6 (->userBalances_5) := 0(uint256)
```

**Fig. 6** The process of converting Solidity to IR and IR-SSA forms

on this information and aforementioned conversion rules, the IR described in the figure is further obtained, which disassembles and represents the execution process of the function *withdrawBalance()*. For example, REFVAL_3 is used to point out the storage location of userBalances[msg.sender], and the variable type is uint256; LOW_LEVEL_CALL realizes the transfer operation with the value of amountToWithdraw, and expresses the returned call status as the memory variable TEMP_2(bool). As can be seen from the IR form, the judgment leads to the difference execution, resulting in the variable *userBalances[msg.sender]* having different values. However, in the IR form, there is no distinction during the subsequent use of the variables. To solve this problem, IR-SSA form leverages the $\phi$ statement to judge the different execution paths automatically, and the return value is expressed as *userBalances_5*. Similarly, the values of the state variables such as *userBalances* are judged by the $\phi$ statement due to they may be changed before invoking the function *withdrawBalance()*. It is consistent with the description in Section 2.2. Furthermore, in IR-SSA form, each assignment of variables (e.g., *userBalances*) employs varied representations, ensuring that each variable is assigned only once.

#### 4.2.4 Advantages of SmartIR over Other IR

At present, SlithIR (Feist et al. 2019), Scilla (Sergey et al. 2018) and other IRs (Kalra et al. 2018; Tsankov et al. 2018; SRI Lab 2020) are used for contract security analysis. SlithIR is the IR used by Slither, and it has the following problems: (i) SlithIR constructs CFG by extracting relevant information from AST. Since AST forfeits part of the original semantics, SlithIR cannot represent the original contract accurately, making Slither unable to detect some vulnerabilities. For example, the var variable (automatic adaptation type) is deprecated after solc0.5.0 (Solidity 2020), so it is necessary to detect it. However, in AST generated by solc compilation, var variables are directly converted to type such as uint8, so that it cannot be identified by SlithIR. (ii) The compilation errors of the contract cause the AST information to be lost, preventing SlithIR from being built. (iii) Function identifiers *constant* and *assembly* are retained in AST, but SlithIR ignores them. It makes Slither unable to identify problems associated with these identifiers. These issues can be resolved with SmartIR. It can be attributed to the XML form in SmartIR obtained by parsing the grammatical rules, so that the complete contract semantics can be obtained for normal analysis even if the contract failed to compile. Nonetheless, there are many restrictions to analysis based only on XML form (e.g., SmartCheck), such as the inability to consider the relationship between components (e.g., contracts and variables). It leads to a limited detection range and decreased matching accuracy. In contrast, SmartIR provides IR and IR-SSA forms to address the aforementioned drawbacks.

Scilla (Sergey et al. 2018) is the IR used by the Zilliqa blockchain. It is unclear whether Scilla can represent the entire Solidity language or only a subset. Similar IRs include Michelson (Tezos 2020), IELE (Kasampalis et al. 2018), Tezla (Reis et al. 2020), etc. However, these IRs store limited contract semantics so that they are unsuitable for static analysis. To sum up, SmartIR integrates XML, IR, and IR-SSA forms to express the whole original semantics of the contract accurately and robustly.

*Remark 2* (Completeness of IR) Compared with IRs such as SlithIR, SmartIR can express more contract semantics, support more complete and precise contract matching operations, and deliver a more robust conversion process.

### 4.3 Pattern Verification/Matching

#### 4.3.1 Xpath Matching for XML Form

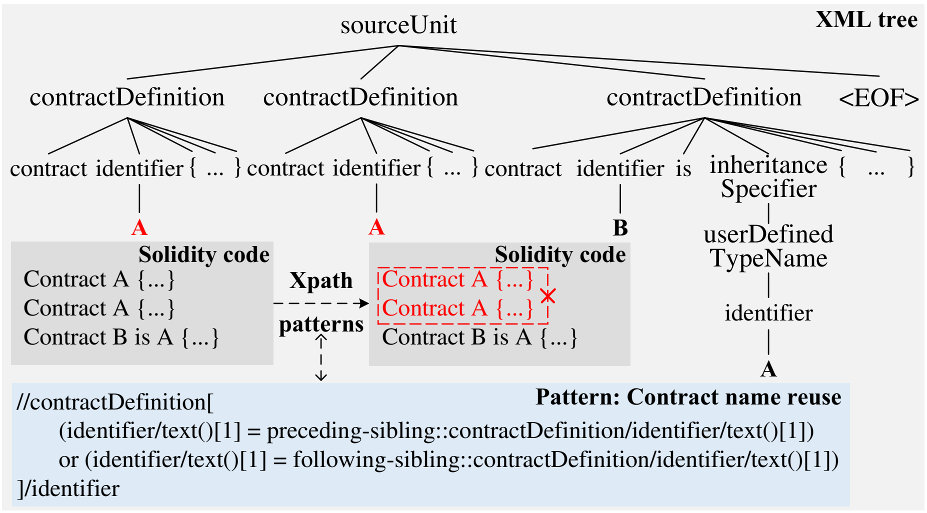
Figure 7 shows the process of SmartFast using Xpath to discover vulnerabilities. It can be seen from the figure that the code has a vulnerability named *names-reused*, that is, there are two contracts whose names are both "A". When the code is compiled, the first contract can pass successfully, while the second contract will be discarded. If contract "B" inherits the second "A" contract, it will cause the contract not to be executed as the author wishes, resulting in unexpected errors. However, since this vulnerability cannot be detected after contract compilation, it blinds the tools (e.g., Slither and Oyente), while it can be detected by SmartFast. Specifically, according to the principle of vulnerability and the XML parse tree, the security pattern in the figure is formulated by XPath language. The security pattern discovers the vulnerable code in XML form by checking whether the current contract name is the same as the previous and subsequent contracts.

#### 4.3.2 Command Execution for IR/IR-SSA Form

However, it is difficult to consider the relationship between components (e.g., contract inheritance) for patterns using XPath language, which makes these patterns unsuitable for detecting some complex vulnerabilities. For example, the code with a vulnerability named balance equality is as follows:

```
uint a = this.balance; if (a==100 ether) {...}
```

Although the variable *this.balance* will not be used directly when judging equality in the code, it will indirectly participate in the judgment and cause the condition of *if* statement to be interfered by the miner. Thus, we introduce a taint tracking mechanism based on IR and IR-SSA forms to detect contract codes that indirectly cause problems. Algorithm 1 details the detection process using the code with *reentrancy-eth* vulnerabilities in Fig. 6.



```
//contractDefinition[
    (identifier/text()[1] = preceding-sibling::contractDefinition/identifier/text()[1])
    or (identifier/text()[1] = following-sibling::contractDefinition/identifier/text()[1])
]/identifier
```

**Fig. 7** The process of XPath (*names-reused*)

```
Input: The contract in IR/IR-SSA
Output: Vulnerable node/IR and other information
result = $\emptyset$;
The first stage: basic information integration;
for node ← functions_and_modifiers_declared ← contracts do
    smartir_operations $\xleftarrow{\text{traverse}}$ node.internal_calls;
    node.context.written = node.write_variables;
    Traverse underlying elements and count information;
    for ir in node.irs + smartir_operations do
        if can_callback(ir) then
            update calls and reads_prior_calls in node.context
        if can_send_eth(ir) and not data_denpendence(ir,0) then
            node.context.send_eth $\cup =$ ir.node
        if isinstance(ir, EventCall) then
            node.context.events $\cup =$ ir.node
The second stage: matching nodes with reentrant features;
for node ← functions_and_modifiers_declared ← contracts do
    context = node.context;
    if context.calls and context.send_eth and find_value( context.written, context.calls,
    context.reads_prior_calls) then
        result $\cup =$ (node,node.function);
return result
```

**Algorithm 1** Security pattern *reentrancy-eth.*

In line with the features of reentrancy vulnerabilities, we develop a security pattern called *reentrancy-eth*, which analyzes the known information using a python script.

In this algorithm, the variable *contracts* represents the collection of all contracts, including inheritance contracts. The composition relationship between contract components is *contracts* $\rightarrow$ *functions* $\rightarrow$ *nodes* $\rightarrow$ *IRs/IRs-SSA*. *functions_and_modifiers_declared* involves the functions and modifiers declared in the *contract*. In this security pattern, the first stage is the integration of basic information. All of the IRs and call operations are traversed to summarize the information such as write variables of each node. Since the transfer with 0 Ether does not result in the loss of Ether, *send_eth* can be combined with *data_denpendence(ir,0)* to filter this situation, where *data_denpendence* uses taint tracking technology to judge the dependency between ir and 0. This stage can simplify the work of the subsequent pattern matching stage, and reduce the duplicated operations in other security patterns. The second stage is the feature matching. The nodes in each contract are traversed and the *find_value* function is used to find variables that are read before the call and written after the call. If these features are met, it indicates that the contract with a *reentrancy-eth* vulnerability. Finally, when all nodes have been traversed, $\emptyset$ will be returned if no vulnerabilities are found. Otherwise, it returns the matched node and function. The *reentrancy-eth* pattern is one of the security patterns with the Reentrancy category detailed in Table 3, and there are others, such as *reentrancy-no-eth*. We have formulated corresponding security patterns for the 119 kinds of vulnerabilities.[2] These security patterns verify the logical relationship of contracts in the form of the above script. The main difference between

[2]The code of patterns is detailed in https://github.com/SmartContractTools/SmartFast/tree/main/smartfast/smartfast/detectors.

them lies in the core idea of detecting vulnerabilities, which is related to the vulnerability features. With the discovery of emerging vulnerabilities, we only need to find their features and define the security patterns to detect them. Thus, SmartFast delivers superior scalability.

## 5 Implementation

SmartFast is mainly implemented in Python with an estimated 25K lines of code.[3] Its entire execution process is shown in Algorithm 2. Specifically, SmartFast incorporates the following 5 components, and each part is responsible for different sub-processes.

**Importer** The input of SmartFast consists of the contract source code $C$ and the defined security pattern set $B$. On the one hand, when the contract is analysed, Importer leverages ANTRL to parse the code into different code statements. On the other hand, it employs compilers such as Truffle, Solc, and Remix to extract the AST of source code, and then constructs the contract inheritance relationship, control flow graph, and Solidity statement expression.

**Converter** Converter constructs SmartIR based on the analysis results of Importer. The construction process is detailed. However, we found many repeated operations during the matching process. For example, traversing the functions' IRs to obtain the dependencies between the variables. Thus, to improve matching efficiency, Converter analyzes SmartIR and aggregates the results of repeated operations into module information to Matcher.

```
Input: Contract source code to be tested $C$, security pattern set $B = B_1 \cup B_2$ ($B_1$ for
       XML form, $B_2$ for IR/IR-SSA form)
Output: Problem report rep
Detection results result = $\emptyset$;
The Solidity contract source code is converted to SmartIR;
$\widehat{C}_1 \xleftarrow{\text{Parsing}} C, \widehat{C}_2 \xleftarrow{\text{Construct}} \text{AST} \xleftarrow{\text{Compile}} C, \widehat{C}_3 \xleftarrow{\text{SSA}} \widehat{C}_2$;
Matching/verification of security pattern;
foreach $b \in B_1$ do
    if $(\{(b, XML)\} = XPath\ (\widehat{C}_1, b)) \neq \emptyset$ then
        result $\cup = (\{(b, C_1)\} \xleftarrow[C]{\text{Convert}} \{(b, \text{XML})\})$;
foreach $b \in B_2$ do
    if $(\{(b, IR/IR\text{-}SSA)\} = Verify\ (\widehat{C}_2/\widehat{C}_3, b)) \neq \emptyset$ then
        result $\cup = (\{(b, C_2)\} \xleftarrow[C]{\text{Convert}} \{(b, \text{IR/IR-SSA})\})$;
rep $\xleftarrow{\text{Generate}}$ result;
return rep;
```

**Algorithm 2** Overall analysis of SmartFast.

[3] SmartFast is available at https://github.com/SmartContractTools/SmartFast/tree/main/smartfast.

**Matcher** Since the difference of detection methods for XML and IR/IR-SSA forms, the security patterns need to be defined as $B_1$ and $B_2$, respectively, and the pattern set $B = B_1 \vee B_2$. Among them, the security patterns for detecting a vulnerability may involve a single or both methods. For example, security pattern $b_1 \in B_1$ for vulnerability named *names-reused*, security pattern $b_2 \in B_2$ for vulnerability named *shadowing-state*, and security pattern $b_3 \in B$ for vulnerability named *deprecated-standards*. In a word, the most suitable detection method should be established according to the vulnerability and detection method features. Furthermore, Matcher assigns the corresponding matching algorithm based on IR forms to validate patterns during the pattern matching process. That is, the XML form selects XPath, and the IR/IR-SSA form chooses command execution. If the detection result is not equal to $\emptyset$, it means that a vulnerability has been detected. The specific verification process is detailed in Section 4.3.

**Integrator** As mentioned above, vulnerabilities such as *deprecated-standards* involve both the matching process of XPath and command execution. Thus, on the one hand, Integrator needs to integrate the detection results of the same vulnerability in the two matching results together. On the other hand, the detection results of different vulnerabilities need to be collected and sorted by Integrator. Since the the pattern matching results are expressed as the problematic SmartIR based on XML and IR/IR-SSA forms, they need to be further converted into "problem source code" $C_1/C_2$ combing with the source code. As described in Algorithm 2, the result $(b, C_1/C_2)$ will be integrated into the detection results $result$, which marks the end of Integrator's work.

**Exporter** Based on $result$, Exporter analyzes the causes and execution process of the vulnerability, and proposes corresponding repair measures. These contents will eventually comprise a complete analysis report $rep$.[4] More importantly, the whole process is automatic and without manual intervention.

## 6 Evaluation Result

In this section, we comprehensively evaluate SmartFast in terms of accuracy, robustness and performance. Also, the experimental result demonstrate the correctness of the theorems in Section 7 from an experimental perspective. Section 6.1 describes the experimental objectives and related settings. Sections 6.2∼6.6 answer the following questions:

**RQ1.** [Effectiveness] What is the effectiveness of SmartFast in detecting vulnerabilities of Solidity smart contracts? In this question, we are interested in comparing the accuracy of SmartFast with the SOTA tools in detecting vulnerabilities of known faulty smart contracts.

**RQ2.** [Robustness] Can SmartFast be suitable for the robust detection of contract vulnerabilities?

**RQ3.** [Production] How many vulnerabilities are present in the Ethereum blockchain?

**RQ4.** [Performance] How much overhead (such as time and memory) does SmartFast require to analyze the smart contracts? Furthermore, we compare its performance with the SOTA tools. The aim is to identify which tool is the most efficient.

[4]Examples of analysis reports is available on https://github.com/SmartContractTools/SmartFast/tree/main/Report.

**RQ5.** [Authenticity] Can SmartFast discover contracts with substantial and serious vulnerabilities in public blockchains such as Ethereum? The goal is to verify whether SmartFast is effective in the real world.

## 6.1 Experimental Setup

**Objectives** We compare the detection results of SmartFast and 7 SOTA tools (described in Section 2.3) with the results of manual audits. These tools can be divided into two classes according to analysis methods, the pattern matching-based tools (Slither Feist et al. 2019, SmartCheck Tikhomirov et al. 2018, Securify Tsankov et al. 2018, Securify2.0 SRI Lab 2020) and symbolic execution-based tools (Oyente Luu et al. 2016, Osiris Torres et al. 2018, Mythril Software 2020). All experiments were conducted on a machine with Intel Core i7-10875H and 8GB of RAM.

**Evaluation Measures** We define the discovery of vulnerability as a problem. By comparing the detection result of tools with the previous vulnerability label, we can measure whether the problem occurs. In this way, all problems found by tools are marked as true positive (TP) or false positive (FP). Moreover, for each tool, a false negative (FN) is a true finding that was not detected by the tool, and the false discovery rate (FDR) is the number of FPs divided by the number of all issues reported by the tool:

$$FDR = \frac{\sum_{i=1}^{n} FP_i}{\sum_{i=1}^{n} (TP_i + FP_i)}, \tag{1}$$

where $n$ is the number of detected contracts. Also, Recall is the number of TPs divided by the number of real vulnerabilities in the contract:

$$Recall = \frac{\sum_{i=1}^{n} TP_i}{\sum_{i=1}^{n} (TP_i + FN_i)} = 1 - FNR. \tag{2}$$

where the false-negative rate (FNR) refers to the ratio of undetected vulnerabilities to all of the vulnerabilities.

**DataSets** Table 5 shows the details of three datasets in the experiment. Dataset_1 is a set of Solidity smart contracts with known vulnerabilities, which is composed of labeled datasets from each SOTA tool. This dataset contains 149 contracts and 6,331 lines of code. In order to ensure the correctness of the labels, we have revised and supplemented these labels based on the verification results of experts. This makes Dataset_1 suitable for analyzing the security of Solidity smart contracts (exploring RQ1 and RQ2). According to the vulnerability severity mentioned in Section 3, we can divide the label of Dataset_1 into: *High* (218), *INFO* (1,152), *OPT* (760), etc. In order to have a representative picture of the practice and (potential) vulnerabilities that are present in the production environment, we have downloaded

**Table 5** Details of datasets

| Dateset | Number of issues | | | | | Contract details | | |
|---|---|---|---|---|---|---|---|---|
| | High | Medium | Low | Info | Opt | Rows | Size | Nums |
| Dataset_1 | 218 | 561 | 253 | 1,152 | 760 | 6,331 | 158.9KB | 149 |
| Dataset_2 | – | – | – | – | – | 8,069,699 | 288.3MB | 13,509 |
| Dataset_3 | – | – | – | – | – | 6,491 | 233.0KB | 29 |

13,509 real-world Solidity contracts by invoking the Etherscan API (Etherscan 2017). These contracts make up Dataset_2 (the size is 288.3MB). Due to the large number of contracts in this dataset and the absence of original labels, the dataset is used only to discuss the number of vulnerabilities in the Ethereum blockchain (exploring RQ2 $\sim$ RQ5). Moreover, Dataset_3 consists of the widely collected contract source code for 29 well-known vulnerability events, which was leveraged to further evaluate the authenticity of SmartFast.

## 6.2 Precision of SmartFast (RQ1)

To answer the first research question, we compare the ability of the 8 tools in detecting the vulnerabilities present in the Dataset_1. Specifically, the methodology is as follows. (i) We executed the 8 tools on the 149 contracts.[5] (ii) We extracted all the vulnerabilities detected by the tools into a JSON file. (iii) As shown in Table 4, we manually annotated each vulnerability detected by the tools as one of the 136 categories (unified labels).[6] For example, SmartCheck detects a vulnerability called *SOLIDITY_TX_ORIGIN* that we link to the category *tx-origin* (No.54 in Table 2). The results of the first study are presented in Table 6, which illustrates the FDR and FNR for each tool. It contains three parts: (i) FDR and FNR of some vulnerabilities (e.g., *reentrancy Ether*); (ii) the overall FDR and FNR of each vulnerability severity; (iii) the overall FDR and FNR of each tool.

As from the part (i), for most vulnerabilities, SmartFast has a lower FDR and a higher Recall than other tools. For instance, SmartFast identifies the *locked-ether* vulnerabilities (15 in total) with an FDR of 0%, while the FDR of 50% for Securify. Moreover, the recall rate detected by SmartFast is 73.33%, which is much higher than SmartCheck (46.67%). Against the vulnerabilities such as *timestamp*, some tools have lower FDRs than SmartFast, due to their inferior recall rate and undetectable for these vulnerabilities. For example, Mythril cannot detect *timestamp* vulnerability so that its Recall = 0% and FDR = 0%. However, a fact that cannot be ignored is that SmartFast loses its advantage in detecting a few vulnerabilities. It can be attributed to the inherent shortcomings of pattern matching. That is, the inability to execute contracts, makes the pattern matching-based tools unsuitable for detecting vulnerabilities in contract execution. For the detection of these vulnerabilities, SmartFast may be weaker than tools using techniques such as symbolic execution (e.g., Osiris). Nevertheless, the detection effect of SmartFast is significant compared to other pattern matching-based tools. For example, for the *reentrancy-eth* vulnerability, the FDR of SmartFast is inferior to that of Slither and Securify2.0 ($18.18\% < 30.77\% < 42.86\%$). Also, its recall rate is superior to Securify2.0 ($37.5\% > 16.67\%$).

By observing the severity from *Opt* to *High* (i.e., part (ii)), the detection effect of SmartFast is always the best compared to the other 7 tools. For example, for *High* severity detection, the FDR of SmartFast is 2.56% ($<14.29\% \ll 70\%$), and Recall = 69.7% ($>60.09\% \gg 0.46\%$). However, some tools enjoy the superior FDR and poor Recall. For example, the FDR detected by Slither for a vulnerability with *Low* severity was 1.77%. It is because Slither is unclear about these vulnerabilities (with a poor recall rate).

From the part (iii), it can be concluded that the detection effect of SmartFast (FDR = 1.57% and Recall = 85.12%) is significantly better than other tools for both FDR and Recall.

[5]The source code and execution result of contracts are available on https://github.com/SmartContractTools/SmartFast/tree/main/Dataset1.

[6]For details on vulnerability detection of each tool, please refer to https://github.com/SmartContractTools/SmartFast/tree/main/VulnerabilityMapping.

**Table 6** Comparative results of false discovery rate (FDR), recall rate (Recall), and failure rate detected by each tool on Dataset_1

| Severity | Project | Metrics | SmartFast | SmartCheck | Slither | Securify | Securify2.0 | Oyente | Osiris | Mythril |
|---|---|---|---|---|---|---|---|---|---|---|
| High | reentrancy-eth | TP/FP/FN | 9/2/15 | 0/0/24 | 9/4/15 | 12/7/12 | 4/3/20 | 7/0/17 | 19/3/5 | 1/0/23 |
| | | FDR (%) | 18.18 | 0.00 | 30.77 | 36.84 | 42.86 | 0.00 | 13.64 | 0.00 |
| | | Recall (%) | 37.50 | 0.00 | 37.50 | 50.00 | 16.67 | 29.17 | 79.17 | 4.17 |
| | Total | TP/FP/FN | 152/4/66 | 1/0/217 | 131/59/87 | 53/92/165 | 33/77/185 | 18/5/200 | 30/8/188 | 6/1/212 |
| | | FDR (%) | 2.56 | 0.00 | 31.05 | 63.45 | 70.00 | 21.74 | 21.05 | 14.29 |
| | | Recall (%) | 69.7 | 0.46 | 60.09 | 24.31 | 15.14 | 8.26 | 13.76 | 2.75 |
| Medium | locked-ether | TP/FP/FN | 11/0/4 | 7/1/8 | 11/0/4 | 11/11/4 | 2/0/13 | 0/0/15 | 0/0/15 | 0/0/15 |
| | | FDR (%) | 0.00 | 12.50 | 0.00 | 50.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | | Recall (%) | 73.33 | 46.67 | 73.33 | 73.33 | 13.33 | 0.00 | 0.00 | 0.00 |
| | Total | TP/FP/FN | 460/15/101 | 105/24/456 | 187/7/374 | 95/221/466 | 51/142/510 | 40/20/521 | 38/2/523 | 53/9/508 |
| | | FDR (%) | 3.16 | 18.60 | 3.61 | 69.94 | 73.58 | 33.33 | 5.00 | 14.52 |
| | | Recall (%) | 82.71 | 18.72 | 33.33 | 16.93 | 9.09 | 7.13 | 6.77 | 9.45 |
| Low | timestamp | TP/FP/FN | 10/0/2 | 3/0/9 | 10/0/2 | 0/0/12 | 4/0/8 | 2/0/10 | 2/0/10 | 7/0/5 |
| | | FDR (%) | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | | Recall (%) | 83.33 | 25.00 | 83.33 | 0.00 | 33.33 | 16.67 | 16.67 | 58.33 |
| | Total | TP/FP/FN | 246/3/7 | 40/1/213 | 111/2/142 | 0/11/253 | 7/2/246 | 2/0/251 | 2/0/251 | 23/6/230 |
| | | FDR (%) | 1.20 | 2.44 | 1.77 | 1.00 | 22.22 | 0.00 | 0.00 | 20.69 |
| | | Recall (%) | 73.95 | 15.81 | 43.87 | 0.00 | 2.77 | 0.79 | 0.79 | 9.09 |
| Info | Total | TP/FP/FN | 965/18/187 | 415/243/737 | 559/9/593 | 64/25/1,088 | 288/42/864 | 0/0/1,152 | 0/0/1,152 | 0/0/1,152 |
| | | FDR (%) | 1.83 | 36.93 | 1.58 | 28.09 | 12.73 | 0.00 | 0.00 | 0.00 |
| | | Recall (%) | 83.77 | 36.02 | 48.52 | 5.56 | 25.00 | 0.00 | 0.00 | 0.00 |
| Opt | unused-state | TP/FP/FN | 47/0/26 | 0/0/73 | 47/4/26 | 0/0/73 | 45/22/28 | 0/0/73 | 0/0/73 | 0/0/73 |
| | | FDR (%) | 0.00 | 0.00 | 7.84 | 0.00 | 32.84 | 0.00 | 0.00 | 0.00 |
| | | Recall (%) | 64.38 | 0.00 | 64.38 | 0.00 | 61.64 | 0.00 | 0.00 | 0.00 |

**Table 6** (continued)

| Severity | Project | Metrics | SmartFast | SmartCheck | Slither | Securify | Securify2.0 | Oyente | Osiris | Mythril |
|---|---|---|---|---|---|---|---|---|---|---|
| | Total | TP/FP/FN | 683/0/77 | 383/3/377 | 600/3/160 | 0/0/760 | 225/29/535 | 0/0/760 | 0/0/760 | 0/0/760 |
| | | FDR (%) | 0.00 | 0.78 | 0.50 | 0.00 | 11.42 | 0.00 | 0.00 | 0.00 |
| | | Recall (%) | 89.87 | 50.39 | 78.95 | 0.00 | 29.61 | 0.00 | 0.00 | 0.00 |
| Overall | Total | TP/FP/FN | 2,506/40/438 | 944/271/2,000 | 1588/80/1356 | 212/349/2,732 | 604/290/2,340 | 60/25/2,884 | 70/10/2,874 | 82/16/2,862 |
| | | FDR (%) | 1.57 | 22.30 | 4.80 | 62.21 | 32.59 | 29.41 | 12.50 | 16.33 |
| | | Recall (%) | 85.12 | 32.07 | 53.94 | 7.20 | 20.52 | 2.04 | 2.37 | 2.79 |
| | | Failed | 0 (0%) | 0 (0%) | 7 (10.5%) | 9 (6.0%) | 85 (57.0%) | 42 (28.2%) | 45 (30.2%) | 21 (14.1%) |

**Listing 10** Contract with arbitrary-send

```
function direct() public{
  msg.sender.send(address(this).balance);
}
```

**Answer to RQ1. What is the accuracy of SmartFast in detecting vulnerabilities on Solidity smart contracts?** Among the other 7 SOTA tools, Slither is the most accurate detection, which has a precision rate of 95.2%, while SmartFast has an precision rate of 98.43%. Moreover, SmartFast achieved a satisfactory recall rate of 85.12%, which proves its remarkable effectiveness compared with other tools. It still works for vulnerabilities above *Low* severity (i.e., FDR=2.5% and Recall=83.14%).

### 6.3 Robustness of SmartFast (RQ2)

Considering the two aspects of analysis failure rate and identification accuracy, the robustness can be divided into detection robustness and analysis robustness. The detection robustness is correlated with the ratio of contracts that can be analyzed (i.e., 1-failure rate). The analysis robustness reflects the performance of tool resistance to the contract code obfuscation (misleading reviewers). For the detection robustness, as shown in Table 6, SmartFast can detect all contracts in Dataset_1 (Failed = 0%). This is impossible for tools such as Slither and Securify2.0. It is because Slither cannot obtain the AST syntax tree of the contract when the contract compilation fails, which causes the analysis to fail. For tools such as Securify2.0, the analysis failure is mainly attributed to errors in internal analysis components (e.g., IR).

For the analysis robustness, the code of some contracts in Dataset_1 was confused by inserting irrelevant statements, disrupting data dependencies, etc. (as shown in Listing 10∼13).[7] Table 7 illustrates the analysis results of the tools before and after the obfuscation, where each severity incorporates 3 kinds of vulnerabilities. For instance, for the *arbitrary-send* vulnerability, anyone can invoke the function "direct()" shown in Listing 10 to extract all the contract balances due to the lack of permission restrictions. Towards making the vulnerability code as invisible as possible, the variables "msg.sender" and "address(this).balance" were replaced by "destination" and "value" in Listing 11, and the assignment statement for irrelevant variables (line 4) was employed to disrupt the connection of the context. As envisioned, SmartFast is able to resist these confusion methods, and the vulnerability can be detected for both contracts. It can be attributed to the data dependencies and taint analysis mechanisms adopted within SmartFast.

Obfuscated contract code may lead to false positives by the tools, which need to be prevented for SmartFast. For the contract shown in Listing 12, it addresses *integer-overflow* vulnerabilities by using the function "assert()" to check the operation variables. Similarly, as shown in Listing 13, the variable "in_c" was checked in the assert() instead of the variable "c", as well as the statement in line 4 was added. As a result, both contracts were correctly marked as secure by SmartFast. Thus, SmartFast can robustly identify contract vulnerabilities for these superficial obfuscation methods that disrupt contextual relationships. However, tools such as SmartCheck are vulnerable to being confused by these methods for their failure to consider relationships between variables. For instance, SmartCheck has missed a timestamp vulnerability in the confused contract *timestamp*. In the future, we will investigate

[7]The code is available on https://github.com/SmartContractTools/SmartFast/tree/main/Obfuscation.

**Table 7** The detection results of the contract code before and after the obfuscation. Among them, Before_Loc and After_Loc indicate the line number of the vulnerability code before and after obfuscation, respectively. Moreover, “×” indicates that the contract caused the tool analysis to fail, “-” means that the tool cannot detect the corresponding vulnerability, and “A(B)” represents that the detection results before and after obfuscation are B and A respectively

| Severity | Vulnerabilities | Contracts | Befor_Loc | After_Loc | SmartFast | SmartCheck | Slither | Securify | Securify2.0 | Oyente | Osiris | Mythril |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| High | arbitrary-send | arbitrary_send | 14 | 17 | TP(TP) | – | TP(TP) | TP(TP) | – | – | – | – |
| | reentrancy-eth | reentrancy_eth | 15 | 18 | TP(TP) | – | TP(TP) | TP(TP) | FN(FN) | FN(FN) | TP(TP) | FN(FN) |
| | rtlo | rtlo | 7 | 9 | TP(TP) | – | TP(TP) | – | FN(FN) | – | – | – |
| Medium | no integer-overflow | integer_overflow | 10 | 10 | TN(TN) | – | – | – | – | ×(×) | ×(×) | ×(×) |
| | tautology | tautology | 3 | 5 | TP(TP) | – | TP(TP) | – | – | – | – | – |
| | tx-origin | tx_origin | 10 | 12 | TP(TP) | FN(TP) | FN(FN) | – | FN(FN) | – | – | FN(FN) |
| Low | block-other-parameters | block_parameters | 24 | 26 | TP(TP) | – | – | – | – | – | – | TP(TP) |
| | timestamp | timestamp | 5 | 7 | TP(TP) | FN(TP) | FN(FN) | – | FN(FN) | FN(FN) | FN(FN) | FN(FN) |
| | transfer-to-zeroaddress | zero_address | 5 | 7 | TP(TP) | – | – | – | – | – | – | – |
| Info | low-level-calls | low_level_calls | 13 | 16 | TP(TP) | TP(TP) | TP(TP) | – | FN(FN) | – | – | – |
| | msgvalue-equals-zero | msgvalue_zero | 8 | 11 | TP(TP) | TP(TP) | – | – | – | – | – | – |
| | similar-names | similar_names | 3 | 5 | TP(TP) | – | FN(FN) | – | – | – | – | – |
| Opt | boolean-equal | boolean_equal | 15 | 17 | TP(TP) | – | TP(TP) | – | – | – | – | – |
| | event-before-revert | event_revert | 6 | 8 | TP(TP) | – | – | – | – | – | – | – |
| | extra-gas-inloops | extra_gas_inloops | 4 | 7 | TP(TP) | TP(TP) | – | – | – | – | – | – |

**Listing 11** Contract after obfuscation

```
function direct() public{
  address destination = msg.sender;
  uint value = address(this).balance;
  uint unuseless_variable = 0;
  destination.send(value);
}
```

the performance of SmartFast in the face of more complex code obfuscation techniques to further validate its analysis robustness.

**Answer to RQ2. Can SmartFast be suitable for the robust detection of contract vulnerabilities?** Compared with 7 SOTA tools, SmartFast can analyze more contracts successfully (all 149 contracts in Dataset_1 were successfully analyzed) and provide better resistance to the confusion of contract code, which proves the superior robustness of SmartFast.

## 6.4 Vulnerabilities in Production Smart Contracts (RQ3)

To answer the third research question, we analyzed the performance of the 8 tools to detect contract vulnerabilities in Dataset_2. Table 8 presents the results of executing the 8 tools on the 13,509 Ethereum contracts. It shows the detection situation of various severity for different tools. Among them, SmartFast detected a large number of contract vulnerabilities, focusing on the severity of *Low* and *Info*. Although as described in Durieux et al. (2020), we found that the contract detection results include many vulnerabilities with inferior utilization difficulties (i.e., the proportions of probably and possibly are High (70%), Medium (89.6%), Low (85.6%), respectively), marking these problems can draw the attention of contract developers to possible security threats, thereby reducing the risk of the contracts. It should be noted that the number of false positives made by tools should be within acceptable limits. Otherwise, it will pose workload challenges for the users. Combining the results of Sections 6.2 and 6.3, SmartFast can evaluate the security of the contract with lower false positives and false negatives than others. Moreover, the vulnerabilities identified by SmartCheck are mainly focused on the severity of *Medium* and *Info*. Also, it can only identify three kinds of vulnerabilities with *High* severity, reflecting the necessity to enhance its ability to exploit critical vulnerabilities. In contrast, Securify2.0 detected numerous vulnerabilities with *High* severity. Combined with the analysis results of Table 6 (i.e., it misreported 70% of vulnerabilities with *High* severity), there may be a large number of false positives. In addition, tools such as Securify2.0 identified less total number of vulnerabilities than others. This can be attributed to Securify2.0 has 7,327 unanalyzable contracts (Failed = 54.2%). The main reason for this phenomenon is that Securify2.0 can only analyze contracts with compiled versions above 0.5.0. In conclusion, compared with other tools, SmartFast can identify vulnerabilities more accurately due to the precise IR and comprehensive security patterns. Also, only a few contracts cannot be analyzed by SmartFast (Failed = 0.4%), which further demonstrates its superior robustness.

```
function mul(uint256 a, uint256 b) ... returns (uint256) {
  uint256 c = a * b;
  assert(a == 0 || c / a == b);
  return c;
}
```

**Listing 12** Contract without integer-overflow

```
function mul(uint256 a, uint256 b) ... returns (uint256) {
  uint256 c = a * b;
  uint256 in_c = c;
  address unuseless_value = 0x0;
  assert(a == 0 || in_c / a == b);
  return c;
}
```

**Listing 13** Contract after obfuscation

More importantly, through this experiment, we found that all tools except Mythril have discovered many vulnerabilities and there are few contracts without vulnerabilities. It is noteworthy that the harm caused by the contract is related to both vulnerabilities and frequency of use. The contracts with vulnerabilities alone may not cause damage to users. Thus, in order to evaluate the security of the Ethereum contracts more reasonably, we have introduced the number of contract transactions as the activity frequency of the contract. Figure 8 presents the correlation between the number of vulnerabilities with various severity and the number of contract transactions. It shows the following phenomenon. (i) The number of vulnerabilities with different severity and the number of contract transactions hold a similar distribution, indicating that contracts tend to have vulnerabilities with varying severity. (ii) The number of contract transactions is widely distributed, and most transactions involve contracts with vulnerabilities. (iii) There is a polarization between the number of transactions and vulnerabilities. It is dangerous to have many vulnerabilities in contracts with greater than 10K transactions, and instead, the harm of contracts with less than 20 transactions and many vulnerabilities may not be serious. (iv) The number of vulnerabilities with severity levels from *High* to *Info* has gradually increased, suggesting that the contracts on Ethereum need to be further improved.

**Answer to RQ3. How many vulnerabilities are present in the Ethereum blockchain?** Most of the contracts in Dataset_2 were detected with vulnerabilities ranging from *Opt* to *High* severity. Among them, the vulnerable contracts with *High*, *Medium*, and *Low* severity accounted for 31%, 94%, and 80%, respectively. Also, the proportion of the contracts where vulnerabilities with inferior utilization difficulties (i.e., probably and possibly) are High (70%), Medium (89.6%), and Low (85.6%), respectively. Although these vulnerabilities are not prone to cause harm, identifying them can allow contract owners to focus on risky code and then develop normative contracts. Moreover, many contracts (94%) in Ethereum can be optimized to improve the operation status of contracts.

### 6.5 Execution Overhead of SmartFast (RQ4)

In this section, we present the execution overhead required by the tools for analyzing contracts on the public blockchain (e.g., Ethereum). First, we selected about 100 contracts with a size of about 121KB, such as the contract with address 0xce5b23f11c486be7f8be4fac3b4ee6372d7ee91e (3,049 lines). Then oscillo was employed to record the time and memory overhead of detecting these contracts.[8] It can be seen from

[8]See https://pypi.org/project/oscillo/ for details.

**Table 8** Comparison of vulnerabilities number detected on 13,509 contracts

| Severity | Project | SmartFast | SmartCheck | Slither | Securify | Securify2.0 | Oyente | Osiris | Mythril |
|---|---|---|---|---|---|---|---|---|---|
| High | reentrancy-eth | 3,620 | – | 3,699 | 608 | 939 | 16 | 223 | 13 |
| | shadowing-state | 867 | – | 3,743 | – | 2,394 | – | – | – |
| | Total | 11,930 | 1,250 | 11,962 | 5,969 | 25,508 | 432 | 466 | 57 |
| Medium | locked-ether | 3,030 | 3,168 | 2,063 | 6,811 | 702 | – | – | – |
| | tautology | 2,144 | – | 430 | – | – | – | – | – |
| | Total | 229,980 | 30,226 | 77,307 | 35,104 | 83,126 | 19,787 | 2,543 | 333 |
| Low | block-other-parameters | 4,925 | – | – | – | – | – | – | 175 |
| | timestamp | 14,501 | 36 | 13,541 | – | 1,737 | 111 | 58 | – |
| | Total | 323,244 | 10,779 | 217,648 | 5,247 | 14,994 | 111 | 58 | 305 |
| Info | deprecated-standards | 11,634 | 11,474 | 1,844 | – | – | – | – | – |
| | solc-version | 51,952 | 36,397 | 38,516 | – | 4,369 | – | – | – |
| | Total | 555,909 | 219,751 | 369,658 | 16,561 | 95,678 | – | – | – |
| Opt | unused-state | 7,962 | – | 14,722 | – | 11,846 | – | – | – |
| | constable-states | 10,601 | – | 10,614 | – | 5,447 | – | – | – |
| | Total | 206,875 | 160,146 | 199,834 | – | 73,491 | – | – | – |
| Overall | Total | 1,327,938 | 422,152 | 876,409 | 62,881 | 292,797 | 20,330 | 3,067 | 695 |
| | Failed | 55 (0.4%) | 0 (0%) | 226 (1.7%) | 272 (2.0%) | 7,327 (54.2%) | 8,815 (65.3%) | 8,782 (65.0%) | 266 (2.0%) |
| | Security contract | 10 | 58 | 27 | 4,220 | 17 | 1,265 | 2,617 | 12,678 |

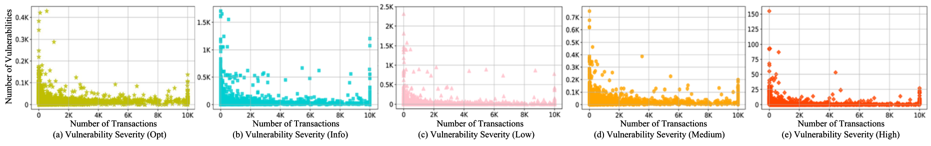


**Fig. 8** Correlation between the number of vulnerabilities detected by SmartFast and the number of contract transactions. In the figure, the upper limit of the number of transactions is set to 10,000 (that is, 10,000 for more than 10,000)

Fig. 9(a) that the pattern matching-based tools (SmartFast, SmartCheck, Slither) generally require less time overhead than the symbolic execution-based tools (Oyente, Mythril, Osiris). But there are exceptions. For instance, Securify2.0 takes an average of 450 seconds to analyze a 121KB contract, which is more than tools such as Oyente. It can be attributed to the cumbersome internal detection mechanism in securify2.0. On the contrary, SmartFast adopts a streamlined design principle to complete the detection in an average of 11.5 seconds. However, since it makes up for the shortcomings of SmartCheck and Slither, more operations are required, which brings an acceptable additional time overhead (about 3∼4 seconds).

As shown in Fig. 9(b), due to thousands of search paths need to be traversed and executed in symbolic execution-based tools such as Osiris, they generally require more memory overhead than pattern matching-based tools. It reflects the main advantage of pattern matching-based tools that contracts can be accurately detected in a finite time. In addition, SmartFast has less memory overhead than other tools (e.g., Slither), which benefits from the optimized patterns. The phenomenon brings great development potential for SmartFast. For instance, the forms of XML and IR in SmartIR can be analyzed in parallel to make full use of memory space. This can further improve the detection efficiency of SmartFast.

**Answer to RQ4. How much overhead does SmartFast require to analyze the smart contracts?** SmartFast takes an average of 11.5 seconds to analyze a 121KB Ethereum contract, which is only one-tenth (or even less) of symbolic execution-based such as Oyente. Compared with pattern matching-based tools such as SmartCheck, the time is within an acceptable range. Moreover, SmartFast has a superior memory overhead (60MB), which provides its expansive development potential.

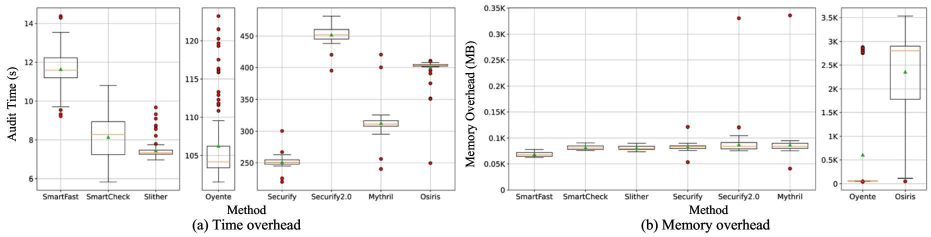


**Fig. 9** Comparison of program overhead in terms of time and memory. Each value in the figure refers to the average of 50 execution results

**Table 9** Real examples of smart contracts on Ethereum. 1st column: detected contract vulnerabilities. 2nd column: contract name. 3rd column: the Ethereum address of the contract. 4th column: number of contract transactions. 5th column: contract balance in Wei (1 ether = 1.00E+18 Wei). 6th column: the line number of the vulnerable code. 7th column: detection results of SmartFast, where "TP" means vulnerability can be detected correctly. 8th column: detection results of other tools, where FNs and FPs represent false negatives and false positives, respectively

| Vulnerabilities | Name | Contract address | Txnums | Balance | Loc | Ours | Others |
|---|---|---|---|---|---|---|---|
| integer-overflow | WinStar | 0x80d0f44bf75ec3e591ac137ac4b88989bb43a006 | 25 | 1.60E+15 | 43 | TP | FNs such as Slither |
| | ETHMaximalist | 0xc36fe594db560bfde1bdf5d6b40a7a775d702a1c | 78 | 5.10E+17 | 171 | TP | FNs such as Slither |
| | AceTokens | 0xf87046beda71bf66c54b74b25ba4b116d93bdb85 | 143 | 2.06E+16 | 255 | TP | FNs such as Osiris |
| | Eightherbank | 0xc6e5e9c6f4f3d1667df6086e91637cc7c64a13eb | 9,455 | 1.52E+19 | 585 | TP | FNs such as SmartCheck |
| | AceDapp | 0xe65f525ec48c7e95654b9824ecc358454ea9185e | 14,202 | 9.60E+19 | 242 | TP | FNs such as Slither |
| No integer-overflow | WCT | 0x2c53d82384c6fc9bd6e4e16ecaa5d85861ac5ce6 | 209 | 9.00E+16 | - | TP | FPs such as Oyente |
| | CoinSuter | 0xba8c0244fbdeb10f19f6738750daeedf7a5081eb | 10,000 | 2.49E+17 | - | TP | FPs such as Oyente |
| reentrancy-eth | RedExchange | 0x5409Fcd56836e0e0459C12Ab45e7Ef23c6094bEd | 266 | 2.00E+16 | 243 | TP | FNs such as oyente |
| | CaptureTheFlag | 0x89B6dDa81ed2eBB5DAD7D4cC3F870a7C00B4e641 | 14 | 0.00E+00 | 127 | TP | FNs such as Mythril |
| shadowing-state | SimpleAssetManagement | 0xf0700f407ca510b8798953386cb75bde0b9337c2 | 61 | 1.06E+18 | 1,052 | TP | FNs such as Securify2.0 |
| | GameFair | 0x86Ab844a067094eC2c5a4badE21fcf2aEEE32122 | 49 | 1.31E+19 | 5 | TP | FNs such as Securify2.0 |
| No shadowing-state | ProxyEvent | 0x8F814C61F7D22f138B743f3FaEd9B4C0FD00f2f9 | 82 | 1.05E+16 | - | TP | FPs such as Securify2.0 |
| | BDSF | 0xe5795bd10c245a0a7b8193f93f568deb568cf18a | 3 | 0.00E+00 | - | TP | FPs such as Slither |
| erc20-approve (TOD) | Auction | 0xa1123ed5a45ffc312aa6964552308f64a1c6cda1 | 18 | 4.94E+17 | 62 | TP | FNs such as Osiris |
| | GooglierToken | 0x270b2fa01da432c34e96375c29c5c1b90a444682 | 6 | 0.00E+00 | 74 | TP | FNs such as Securify2.0 |
| locked-ether | TokenFactoryProxy | 0x000000000092c287eb63e8c2c30b4a74787054f8 | 3,023 | 4.50E+18 | 17 | TP | FNs such as Slither |
| uninitialized-state | Token | 0xf6E435b7e8a9fbC1C2B73c9a9FC08aaf65070671 | 9 | 1.10E+16 | 111 | TP | FNs such as Slither |
| unused-state | Oracle | 0x634ab8f3f791a905b6e6ea72c33483401ed56e6b | 5,462 | 5.16E+17 | 2,979 | TP | FNs such as Slither |
| No unused-state | Oracle | 0x634ab8f3f791a905b6e6ea72c33483401ed56e6b | 5,462 | 5.16E+17 | 676 | TP | FPs such as Slither |

```
1 mapping (address => uint256) invested;
2 mapping (address => uint256) atBlock;
3 function () external payable {
4   uint256 amount = invested[msg.sender] * 4 / 100 * (block.number - atBlock[
        msg.sender]) / 5900;
5   msg.sender.transfer(amount);
6   ...
7 }
```

**Listing 14** Contract with *integer-overflow*

## 6.6 Authenticity of SmartFast (RQ5)

Towards exploring the superior performance of SmartFast, we demonstrated the detection results of real-world contracts (including those deployed on Ethereum in Dataset_2 and those derived from announced vulnerability incidents in Dataset_3). Table 9 shows some examples of Ethereum contracts in Dataset_2. It contains the detection results of tools, as well as the address, balance and other information of the contracts.

**Detection of *integer-overflow* Vulnerabilities** The contract *WinStar* currently has a balance of 1.60E+15Wei and involves 25 transactions. There is an *integer-overflow* vulnerability in line 4 of Listing 14 (corresponding to line 43 in the contract), which occurs when atBlock[msg.sender]=1 and invested[msg.sender]=115792089237316195423570985008687907853269984665640564039457584007913129639935. The vulnerability will make users extract the incorrect amount. It can be detected by SmartFast, but yet not detected by tools like Slither and Osiris. Similar contracts include *ETHMaximalist* (line 171), *AceTokensden* (line 255), *Eighterbank* (line 585), and *AceDapp* (line 242). For these contracts, SmartFast can accurately find the line number of vulnerable code, while tools such as Slither and SmartCheck are helpless. As we all know, Oyente supports the detection of *integer-overflow* vulnerabilities. However, against the contract *WCT* (balance 9.00E+16Wei), it reported that there is an integer overflow vulnerability in line 43 with the code c=a+b (as shown in Listing 15). Since the arithmetic variables were validated by assert(c$\geq$a). Thus, this is a false positive for Oyente. As expected, SmartFast detects that this code is secure. Similar contracts include *CoinSuter* and so on.

**Detection of *reentrancy-eth* Vulnerabilities** SmartFast detected a *reentrancy-eth* vulnerability in line 243 (corresponding to line 14 of Listing 16) of the contract *RedExchange* (2.00E+16Wei). The function "payFund()" is declared as public. Although this function is guarded with the modifier "onlyAdministrator", anyone can become a member of administrators by invoking the function "RedExchange". Moreover, the gas specified by the call function is too large, while the secure gas is usually 2300 ($\ll$40,000). Thus, for this contract, attackers can construct an attacking contract and leverage the function "setBondFundAddress()" to set the withdrawal address "bondFundAddress" as the attacking contract address, thereby realizing a reentrancy attack. However, tools such as Oyente didn't identify this

```
library SafeMath {
  function add(uint256 a, uint256 b) ... returns (uint256 c) {
    c = a + b;
    assert(c >= a);
    return c;
  }
}
```

**Listing 15** Contract without *integer-overflow*

```
function RedExchange() public {
  administrators[msg.sender] = true;
}
modifier onlyAdministrator() {
  require(administrators[msg.sender]);
  _;
}
function setBondFundAddress(address _newBondFundAddress)
     onlyAdministrator() public {
  bondFundAddress = _newBondFundAddress;
}
function payFund() payable public onlyAdministrator() {
  ...;
  totalEthFundRecieved = SafeMath.add(totalEthFundRecieved,
       ethToPay);
  if(!bondFundAddress.call.value(_bondEthToPay).gas(400000)()) {
    totalEthFundRecieved = SafeMath.sub(totalEthFundRecieved,
         _bondEthToPay);
    ...;
  }
}
```

**Listing 16** Contract with *reentrancy-eth*

vulnerability. Similarly, there is a *reentrancy-eth* vulnerability in line 127 of the contract *CaptureTheFlag*, while tools like Oyente and Mythril cannot detect it.

**Detection of *shadowing-state* Vulnerabilities** For contract *SimpleAssetManagement* (1.06E+18Wei), SmartFast discovered the state variables "_name" and "_symbol" (as shown in Lines 2 and 8 of Listing 17) are redeclared when the contract is inherited. It causes the functions "name()'' and "symbol()'' inherited by *Dpass* to always return the initial values regardless of how both variables change. Similarly, there are the variable "ethWei" and the function "getLevel()" in the contract *GameFair* (1.31E+19Wei). However, tools such as Securify2.0 cannot detect these vulnerabilities. To be worse, Securify2.0 inspects a vulnerability about the variable "version" in the contract *ProxyEvent*. However, this variable has not been used in the parent contract, so this is a false-positive for Securify2.0. Similarly, the contract *BDSF* was misreported. But it is secure for SmartFast to detect two false-positive codes.

**Detection of *erc20-approve* Vulnerabilities** The ERC20 approve attack is a type of transaction sequence dependency vulnerability (TOD). This vulnerability can be attributed to the approve function, which is employed to authorize others to use tokens. When the authorizer changes the original authorization, the user creates a consumption transaction that spends the original authorization token and sets more Gas than the changed authorization transaction. In this way, the miners will give priority to the consumption transaction, so that the user can spend the old authorized amount as well as the new authorized amount. SmartFast detected this vulnerability in Contract *Auction* (4.94E+17Wei) and *GooglierToken*, while tools like Osiris and Securify2.0 missed these two vulnerabilities.

```
1  contract ERC721Metadata is ... {
2    string private _name, _symbol;
3    function name() external view returns (string memory) {return _name;}
4    function symbol() external view returns (string memory) {return _symbol;}
5  }
6  contract ERC721Full is ..., ERC721Metadata { ... }
7  contract Dpass is ERC721Full, ... {
8    string private _name = "Diamond Passport", _symbol = "Dpass";
9  }
```

**Listing 17** Contract with *shadowing-state*

```
mapping(address => uint256) balances;
uint256 totalSupply_;
function totalSupply() public ... {return totalSupply_;}
function transfer(address _to, uint256 _value) public returns (bool) {
  ...
  balances[msg.sender] = balances[msg.sender].sub(_value);
  ...
}
```

**Listing 18** Contract with *uninitialized-state*

**Detection of *locked-ether* Vulnerabilities** SmartFast detected a *locked-ether* vulnerability in the contract *TokenFactoryProxy* (4.5E+18Wei). In this contract, the function "upgradeToAndCall()" (line 17) can receive Ether value. However, the contract has no withdrawal function, so the user cannot retrieve Ether exploited to call the contract. For this vulnerability, tools such as Slither will underreport.

**Detection of *uninitialized-state* Vulnerabilities** As shown in Listing 18, in the function "totalSupply()" of the contract *Token* (1.10E+16Wei), the variable "totalSupply_" is called directly without initializing. This violates the development specification. Moreover, the contract does not provide a function to initialize the mapping variable "balances", so that the function "transfer()" makes an error when it is invoked. Thus, we should develop contracts in compliance with the development specifications to improve the security of contracts. For these vulnerabilities, SmartFast can provide warnings, which is impossible with tools such as Slither.

**Detection of *unused-state* Optimizations** In addition to discovering vulnerabilities, SmartFast can also implement code optimization. It detects that the state variables such as "week" (line 2,979 of the contract *Oracle*) are declared but not used, which makes extra gas wasted. However, tools such as SmartCheck cannot support this detection. Although Slither can detect this problem, it located the state variables that have been used such as "ABI_INTERFACE_ID" (a false positive).

In addition, we leverage the contracts of well-known vulnerability incidents in Dataset_3 to further clarify the performance of SmartFast.[9] Table 10 describes the information about the contracts, including security incidents, vulnerability names, contract addresses, economic losses, and detection results. **The DOS vulnerability incident for the KotET contract.** *KotET* designed a game "throne race", in which the player with the largest amount of competition will win the throne. However, in February 2016, the players could not win the throne, no matter how much ETH they sent to the contract. Cause of the vulnerability: the vulnerable contract is shown in Listing 19. The attacker first constructs an attacking contract containing a fallback function with function "revert()", and invokes the function "bid()" through the contract to become the king. Thus, the function "send()" (line 12) triggers the fallback function and always returns the false, so as to intercept the execution of bid(). As a result, SmartFast benefits from its perfect vulnerability patterns to detect this vulnerability, but it is missed by tools such as securify2.0.

**The Dao Vulnerability Incident** In June 2016, the *reentrancy-eth* vulnerability in the *DAO* contract caused a loss of $60 million ETH. Cause of the vulnerability: as shown

[9]The contract code is detailed in https://github.com/SmartContractTools/SmartFast/tree/main/VulnerabilityIncidents.

**Table 10** Examples of smart contract security events. Among them, "-" indicates that the information is unknown

| Date | Vulnerabilities | Incidents | Contract address | Money | Loc | SmartFast | Others |
|---|---|---|---|---|---|---|---|
| February 2016 | DOS | KotET | 0xb336a86e2Feb1E87a328FCb7DD4D04dE3DF254D0 | – | 105 | TP | FNs such as Securify2.0 |
| June 2016 | reentrancy-eth | The Dao | 0x304a554a310C7e546dfe434669C62820b7D83490 | $60 million | 201 | TP | FNs such as Securify2.0 |
| July 2017 | parity-multisig-bug | parity bug | 0xBEc591De75b8699A3Ba52F073428822d0Bfc0D7e | $30 million | 223 | TP | FNs such as Slither |
| November 2017 | access permissions | parity bug 2 | 0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4 | $1.52 billion | 111 | TP | FNs such as Slither |
| February 2018 | integer-overflow | EMVC | 0xd3F5056D9a112cA81B0e6f9f47F3285AA44c6AAA | – | 16 | TP | FNs such as Oyente |
| April 2018 | integer-overflow | BEC | 0xC5d105E63711398aF9bbff092d4B6769C82F793D | – | 261 | FN | FNs such as Slither |
| April 2018 | integer-overflow | SMT | 0x43eE79e379e7b78D871100ed696e803E7893b644 | – | 134 | TP | FNs such as SmartCheck |
| July 2018 | access permissions | AMORCOIN | 0x14Fb4c93Fe461EC3F9F22B61aB7030F258867969 | – | 32 | TP | FNs such as Slither |
| July 2018 | integer-overflow | AMR | 0x96c833e43488c986676e9f6b3b8781812629bbb5 | – | 205 | TP | FNs such as Osiris |
| August 2018 | block parameters | FOMO3D | 0xA62142888ABa8370742bE823c1782D17A0389Da1 | $3 million | 1365 | TP | FNs such as SmartCheck |
| – | integer-overflow | SIGMA | 0x03AF37073258B08FfFF303e9E07E8a0B7bfc4fd9 | – | 1873 | TP | FNs such as Mythril |
| – | integer-overflow | UCN | 0x6EF5B9ae723Fe059Cac71aD620495575d19dAc42 | – | 20 | TP | FNs such as Oyente |
| – | integer-overflow | ETHX | 0x1c98eea5fe5e15d77feeabc0dfcfad32314fd481 | – | 205 | TP | FNs such as Oyente |
| – | integer-overflow | H3H3 | 0x04129cff7d79a652256c73b1407f34828e79a4de | – | 205 | TP | FNs such as Oyente |
| – | integer-overflow | NUMB | 0x310b15ad0a8b282ee8011d91c79feba80cb179c8 | – | 205 | TP | FNs such as Oyente |
| – | integer-overflow | POSH | 0x98307e036c5d024447ea59296f6e41cb998b38d0 | – | 183 | TP | FNs such as Oyente |
| – | integer-overflow | POWC | 0x0f7524d64ceb92dae1d1a56b20b6d520134795d4 | – | 205 | TP | FNs such as Oyente |
| – | integer-overflow | POWH | 0xa7ca36f7273d4d38fc2aec5a454c497f86728a7a | – | 205 | TP | FNs such as Oyente |
| – | integer-overflow | POWH | 0xe923dD860176d3ef69D7852257cC773900807f7C | – | 205 | TP | FNs such as Oyente |
| – | integer-overflow | POWH3 | 0xc825aa83f12e4d225ea1f21511a68e7aa78a002f | – | 181 | TP | FNs such as Oyente |
| – | integer-overflow | PWHS | 0x9f4fd6c336388f2ab7dc7bbe4740ae7b88b880d7 | – | 207 | TP | FNs such as Oyente |

**Listing 19** KotET

```
address public currentLeader;
uint256 public highestBid;
function bid() public payable {
  require(msg.value > highestBid);
  require(currentLeader.send(highestBid));//Dos code
  currentLeader =msg.sender;
  highestBid = msg.value;
}
```

in Listing 20, the behaviors of lines 4-5 describe the implementation of the function "withdrawRewardFor". Since the operation balances[msg.sender] = 0 is completed after executing the call function without a gas limit. It allows attackers to invoke splitDAO() repeatedly without performing the code in line 6. Thanks to the accurate IR, SmartFast can identify this vulnerability, while tools like securify2.0 and Mythril cannot.

**The *parity-multisig-bug* Vulnerability Incident for the Parity Wallet** In July 2017, a permission vulnerability in the Party contract library was exploited by attackers, resulting in the theft of over $30 million ETH. Cause of the vulnerability: as shown in Listing 21, since the visibility of the function "initWallet" is public, attackers can change the owner of the wallet by invoking the initWallet. Similarly, SmartFast has identified the vulnerable code due to its comprehensive security patterns. However, tools such as Slither and SmartCheck without the ability to detect this vulnerability.

**The Second Parity Wallet Vulnerability Incident** Towards repairing the above vulnerability, Parity supplemented the modifier "only" (line 12 in Listing 22) to restrict initMultiowned() to be invoked only once. However, in November 2017, the revised contract caused about $152 million ETH to be frozen due to a permission vulnerability. Cause of the vulnerability: as shown in Listing 22, an attacker can invoke the initMultiowned() through the attacking contract. Since the variables such as m_numOwners are operated from the attacking contract (i.e., m_numOwners = 0), the modifier "only" will be passed and the owner of the library contract will be updated to the attacker. Fortunately, SmartFast was able to identify this vulnerability, demonstrating its superior identification ability for vulnerabilities.

**The *integer-overflow* Vulnerability Incident for the SMT Contract** In April 2018, the transactions of the SmartMesh (SMT) contract were suspended by various platforms such as Ethereum. Cause of the vulnerability: as shown in Listing 23, attackers can manipulate the input parameter of the function "transferProxy" to make _fee+_value = 0 (*integer-overflow*), so that the verification in line 3 will be passed. Thus, attackers can obtain plenty of money. Also, contracts such as *EMVC* caused economic losses due to the ineffective arithmetic examination performed by the non-conforming SafeMath library. However, although the arithmetic variables are checked in *BEC* contract, errors in its examination logic caused an *integer-overflow* vulnerability. Since methods such as pattern matching are challenging

**Listing 20** The Dao

```
1 function splitDAO() ... {
2   Transfer(msg.sender, 0, balances[msg.sender]);
3   withdrawRewardFor(msg.sender);
4   // if (reward) throw;
5   // msg.sender.call.value(reward)()
6   balances[msg.sender] = 0;
7   ...
8 }
```

**Listing 21** parity_multisig_bug

```
function initWallet(...) {
  initMultiowned(...);
}
function initMultiowned(...) {
  m_numOwners = _owners.length + 1;
  m_owners[1] = uint(msg.sender);
  m_ownerIndex[uint(msg.sender)] = 1;
}
```

to detect vulnerabilities with complicated logic, SmartFast missed this vulnerability. Thus, the combination of dynamic analysis (e.g., fuzzy testing) and static analysis (e.g., pattern matching and symbolic execution) is a trend in the field of contract audit.

**The Block Parameters Vulnerability Incident for the FOMO3D Contract** In August 2018, *FOMO3D* lost $3 million ETH. Cause of the vulnerability: the function "airdrop()" employs block parameters such as block.timestamp (line 2 in Listing 24) to generate a random seed. Since these parameters can be predicted by the miners, the attackers can conspire with the miners to infer the seed in advance and meet the game victory conditions. However, tools such as SmartCheck yielded false negatives due to the rough detection rules. On the contrary, SmartFast exhaustively considers the features of this vulnerability to detect it successfully.

**Answer to RQ5. Can SmartFast discover contracts with substantial and serious vulnerabilities in public chains such as Ethereum?** From the aforementioned examples, it is clear that SmartFast can find vulnerabilities ignored by other tools, reflecting its detection effectiveness on Ethereum contracts. Moreover, the contract examples in Dataset_2 indirectly confirm the authenticity of the detection conclusions in Section 6.4. In addition, SmartFast is not only a tool for error correction, but also can optimize contracts and reduce unnecessary costs.

## 7 Deep Insights of the Correctness and Effectiveness

### 7.1 Correctness Analysis

SmartIR can almost accurately and unambiguously describe the semantics of the original contract $C$. In other words, considering that a program $\widehat{C}$ represents the contract $C$ in SmartIR, which will hardly destroy the semantics of $C$. Let $B_{\widehat{C}}$ be the set of patterns satisfied by $\widehat{C}$, which can be expressed as $B_{\widehat{C}} = \{b | b \in B \wedge match(b, \widehat{C})\}$, where $B$ is the set of all vulnerability patterns to be checked, and the relationship $match(b, \widehat{C})$ indicates that program $\widehat{C}$ satisfies pattern $b$. From the following inference, it can be concluded that SmartFast can analyze the contract security correctly.

**Lemma 1** *The contract $C$ satisfies a vulnerability pattern $b$ in $B_C$ $\Rightarrow$ The program $\widehat{C}$ satisfies the pattern $\hat{b}$ in $B_{\widehat{C}}$.*

**Listing 22** parity_multisig_bug_2

```
function initWallet(...) only {
  initMultiowned(...);
}
function initMultiowned(...) only {
  ...
}
modifier only { if (m_numOwners > 0) throw; _; }
```

```
mapping (address => uint256) balances;
function transferProxy(uint256 _value, uint256 _feeSmt,...) public ...{
  if(balances[_from]<_feeSmt+_value) revert();
  Transfer(_from, msg.sender, _feeSmt);
  ...
}
```

**Listing 23** SMT

*Proof* The contracts' formal description mainly changes the representation of the programming language, barely the contract original semantics. Also, pattern matching designed for vulnerabilities inspects the semantics of the contract expression, instead of the language. In other words, the effect of pattern matching is related to the semantics checked by the patterns, but not the representation form. Moreover, the patterns in sets $B_{\widehat{C}}$ and $B_C$ correspond one-to-one, that is, there is a mapping function $f(x)$ to make $\{\widehat{b} = f(b)|\widehat{b} \in B_{\widehat{C}}\}$ hold. Thus, the lemma is proved. Nonetheless, it cannot be ignored that when patterns are defined inaccurately, it may cause false positives and false negatives. Fortunately, from Remarks 1∼2, it is known that the patterns and SmartIR of SmartFast are superior to other tools (e.g., Slither). □

**Theorem 1** (Correctness) *The program $\widehat{C}$ has no vulnerabilities on patterns set $B_{\widehat{C}} \Rightarrow$ The contract $C$ is safe for $B_C$.*

*Proof* From Lemma 1, if the program $\widehat{C}$ does not satisfy each pattern $\widehat{b} \in B_{\widehat{C}}$, it can be concluded that $b$ is not satisfied with the contract $C$, i.e., $C$ is safe for pattern set $B_C$. So far, the theorem has been proved. □

## 7.2 Effectiveness Analysis

Assuming that the vulnerability pattern set $B_1$ for other tools such as Slither, and the vulnerability pattern set $B_2$ for SmartFast. Since two conditions are required to detect a vulnerability, i.e., the search area contains the vulnerability area (the probability is recorded as $P_{in}$) and the vulnerability can be detected in the vulnerability area (the probability is recorded as $P_{find}$), so the probability of detecting the vulnerability $class_1$ ($class\,(b) = class_1$, where $class\,(\cdot)$ represents the vulnerability category of the patterns) can be expressed as:

$$P(match(b, \widehat{C})) = P_{in} \times P_{find} \tag{3}$$

From Remark 1, SmartFast can detect more kinds of vulnerabilities and accurately describes each security pattern. Thus, $|class\,(B_1)| \le |class\,(B_2)|$ and $P_{find-B_1} \le P_{find-B_2}$. Moreover, combined with the description in Remark 2, SmartIR can express the original semantics more wholly and exactly than the IRs such as SlithIR, which increases the possibility of finding vulnerabilities, namely $P_{in-B_1} \le P_{in-B_2}$. Then $P(match(b_1, \widehat{C})) \le P(match(b_2, \widehat{C}))$ is obtained, where $class\,(b_1 \in B_1) = class\,(b_2 \in B_2) = class_1$. Thus,

```
function airdrop() ... {
  uint256 seed = uint256(keccak256(abi.encodePacked((block.timestamp).add...;
  if((seed -((seed/1000)*1000))<...)
    return(true);
  ...
}
```

**Listing 24** FOMO3D

compared with SOTA tools (e.g., Slither), SmartFast can detect the contract security accurately by optimizing the IR and the security patterns.

*Remark 3* (Effectiveness) Compared with the SOTA tools such as Slither and SmartCheck, SmartFast can better detect the security of contract $C$.

## 8 Discussion

**The Advantages of** SmartFast As illustrated in Section 6, each tool has its own characteristics and applicable scenarios. For example, pattern matching-based tools (e.g., SmartFast and Slither) can detect more vulnerabilities than symbolic execution-based tools (e.g., Mythril). This benefits from the convenient formulation of vulnerability rules in pattern matching-based analysis tools. Moreover, compared to SOTA tools, SmartFast has superior vulnerability detection capabilities and can efficiently detect vulnerabilities of contracts deployed on Ethereum (including vulnerability incident contracts). Also, it can analyze more contracts and resist code obfuscation methods that disrupt contextual logic. These benefit from its unified vulnerability severity evaluation mechanism, robust IRs, perfect security patterns, and accurate pattern verification methods. In addition, the whole process from contract input to report output has no manual intervention, which is convenient for users.

**The Limitation of** SmartFast Since the limitations of pattern matching, SmartFast cannot give the input that caused the vulnerabilities to occur. As described in Section 6.6 on *integer-overflow* vulnerability, although SmartFast uses taint analysis and data dependencies to simulate the dynamic execution process of contracts, it still fails to achieve input solving and vulnerability verification. While tools based on symbolic execution (e.g., Oyente) and fuzzy testing (e.g., SMARTIAN) can obtain the specific vulnerability “input” by accessing Z3 solver (Corporation 2020) and fuzzers (Choi et al. 2021). Moreover, similar to other static analysis tools (e.g., SmartCheck), the vulnerability detection effect of SmartFast depends on the accuracy of the defined patterns, thus requiring further investigation for unknown vulnerabilities. From these aspects, it can be concluded that pattern matching-based tools are suitable for comprehensive detection, while tools based on dynamic analysis and symbolic execution are good at accurate detection.

**The Improvement of** SmartFast The theorem proving method serves as another branch in contract auditing, which uses mathematical logic to ascertain whether the contract satisfies the properties (e.g., maintaining the total amount of Token issued). It can verify the deeper properties of the contracts, while neglecting the detection of specific vulnerabilities (e.g., reentrancy). Moreover, with the rise of technologies such as fuzzy testing and artificial intelligence (AI), SmartFast has ushered in many optimization opportunities. For instance, the learning of the AI model may replace the manual definition of security patterns. To this end, we will investigate in the future how to leverage the advantages of these tools and technologies to remedy the shortcomings of SmartFast, which can make SmartFast more precise and robust (i.e., resistance to advanced code obfuscation methods). In addition, as part of maintaining the security of the contract, making improvements after detecting vulnerabilities can better protect users’ privacy and property security. For instance, derived from the advantages of static analysis (i.e., the detected vulnerabilities have specific features), SmartFast can perform customized repairs for vulnerabilities (e.g., the deletion, modification, and addition of contract code) according to their operating mechanism (Krupp and Rossow 2018).

**The Application Prospect of SmartFast** As security-as-a-service businesses are emerging, smart contract security and the contracting of these businesses is an interesting problem space. Similar to Slither, SmartFast can work with Remix to provide security audit services for the Ethereum contract developers. Also, it can serve as an extension plug-in (e.g., a component of the Blockchain Development Kit) for development tools such as Visual Studio Code. Besides, the pattern matching method can integrate the features of Webshell Backdoor to analyze the security of website scripts (e.g., JSP and PHP).

## 9 Related Work

### 9.1 Dynamic Analysis

In 2016, Hirai first proposed using Isabelle to determine whether there are vulnerabilities in the logic code. Grishchenko et al. (2018c) and Hildenbrandt et al. (2018) further used F*s framework and K framework to transform EVM into a formal tool. Recently, Jiao et al. (2020) described the executable semantics of contract source code in the K framework. However, these tools are difficult to describe vulnerabilities, which brings challenges to vulnerability analysis. Moreover, these analysis tools require users to manually provide specifications or invariants, whose automation needs to be improved. Recently, Nguyen et al. (2020) proposed an adaptive fuzzer called sFuzz to detect limited vulnerabilities (e.g., *timestamp*). SMARTIAN (Choi et al. 2021) and ILF (He et al. 2019) used techniques such as machine learning and dynamic data-flow analysis to generate high-quality datasets for fuzzy testing, which can help fuzzer explore the deep contract paths with complex conditions. Although these tools detect fewer vulnerabilities than SmartFast, analyzing features of these tools (e.g., execution process) can help us further optimize SmartFast.

### 9.2 Static Analysis

**Detection Based on EVM Bytecode** Permenev et al. (2020) used predicate abstraction and symbolic execution engine to verify the contract security properties such as time. Other tools based on symbolic execution include ETHBMC (Frank et al. 2020), teEther (Krupp and Rossow 2018), BeosinVaaS (Beosin 2020), Oyente (Luu et al. 2016), etc. Schneidewind et al. (2020) used Horn clauses to describe the semantics of EVM bytecodes, and further proposed a static analyzer named eThor. Similar tools based on EVM bytecode analysis include EtherTrust (Grishchenko et al. 2018a), SODA (Chen et al. 2020), Securify (Tsankov et al. 2018), etc. However, Grishchenko et al. (2018b) stated that the tools had lost part of the original semantics during the code conversion.

**Detection Based on Solidity Source Code** NeuCheck (Lu et al. 2019) constructed a contract syntax diagram based on the contract source code, and searched for vulnerability patterns in the diagram to find the corresponding vulnerabilities. However, they did not give a case for restoring Solidity semantics, nor give formal security attributes. Similarly, ZEUS (Kalra et al. 2018) takes the Solidity code as an input and abstracts the code into IR (i.e., LLVM). However, Torres et al. (2018) and Grishchenko et al. (2018b) noted that ZEUS had an unsound conversion process. Similar tools include Slither (Feist et al. 2019), SmartCheck

(Tikhomirov et al. 2018), etc. Compared with these tools, SmartIR can restore more contract semantics, enabling SmartFast to realize the contract analysis with fully automatic, superior precision and recall.

In addition, some tools (e.g., Rodler et al. 2019 and Liu et al. 2018) focus on a single vulnerability (e.g., reentry and arithmetic bugs). These works are helpful for SmartFast to optimize the detection of these vulnerabilities.

## 10 Conclusion and Outlook

In this paper, we first proposed a vulnerability assessment model to unify the measurement criteria. The evaluation of the tools indicates that most pattern matching-based tools exhibit a superior vulnerability coverage rate over the symbolic execution-based tools. However, since constrained IR and imperfect security patterns in pattern matching-based tools, it remains a challenge to discover vulnerability more accurately and comprehensively. To remedy these deficiencies, we designed SmartIR and pattern verification methods, which constitute SmartFast. The evaluation results demonstrate its efficiency (only took 11.5 seconds for 121KB Ethereum contract). Compared with SOTA tools, SmartFast has a higher precision rate (98.43%) and a lower false negatives (14.88%). Also, SmartFast can analyze most contracts deployed on Ethereum practically (99.6%) and resist code obfuscation methods that disrupt contextual logic, which shows its superior robustness. Furthermore, it can discover vulnerability codes accurately in major vulnerability incident contracts. For instance, a *reentrancy-eth* vulnerability for the "The Dao" security incident. Finally, we discussed the limitations of SmartFast and the in-depth integration with technologies such as symbolic execution, which inspires the future development of automated contract analysis tools.

**Acknowledgements** This work was supported by the National Key R&D Program of China under Grant No.2021YFB2700603, National Natural Science Foundation of China under Grant No.62072487 and No.62172405, and Beijing Natural Science Foundation under Grant No.M21036.

## Declarations

**Conflict of 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

Beosin (2020) Beosin: Blockchain security one-stop service. [EB/OL]. https://beosin.com/#/. Accessed 1 May 2021

Blockchain C (2018) Bamboo: a morphing smart contract language. [EB/OL]. https://github.com/cornellblockchain/bamboo. Accessed 1 May 2021

Bocek T, Stiller B (2018) Smart contracts–blockchains in the wings. In: Digital marketplaces unleashed. Springer, pp 169–184

Chen T, Cao R, Li T, Luo X, Gu G, Zhang Y, Liao Z, Zhu H, Chen G, He Z, Tang Y, Lin X, Zhang X (2020) SODA: a generic online detection framework for smart contracts. In: NDSS. The Internet Society

Choi J, Kim D, Kim S, Grieco G, Groce A, Cha SK (2021) SMARTIAN: enhancing smart contract fuzzing with static and dynamic data-flow analyses. In: ASE. IEEE, pp 227–239

Corporation M (2020) The z3 theorem prover. [EB/OL]. https://github.com/Z3Prover/z3. Accessed 1 May 2021

DappHub (2019) Formal verification of multicollateral dai in the k framework. [EB/OL]. https://github.com/dapphub/k-dss/. 1 Accessed May 2021

Durieux T, Ferreira JF, Abreu R, Cruz P (2020) Empirical review of automated analysis tools on 47, 587 ethereum smart contracts. In: ICSE. ACM, pp 530–541

Etherscan (2017) Contracts with verified source codes only. [EB/OL]. https://etherscan.io/contractsVerified. Accessed 1 May 2021

Feist J, Grieco G, Groce A (2019) Slither: a static analysis framework for smart contracts. In: WETSEB@ICSE. IEEE/ACM, pp 8–15

Foundation E (2020) The solidity contract-oriented programming language. [EB/OL]. https://github.com/ethereum/solidity. Accessed 1 May 2021

Frank J, Aschermann C, Holz T (2020) ETHBMC: a bounded model checker for smart contracts. In: USENIX Security symposium. USENIX Association, pp 2757–2774

Grishchenko I, Maffei M, Schneidewind C (2018a) Ethertrust: sound static analysis of ethereum bytecode. Technische Universität Wien. Tech Rep

Grishchenko I, Maffei M, Schneidewind C (2018b) Foundations and tools for the static analysis of ethereum smart contracts. In: CAV (1), vol 10981. Springer. Lecture Notes in Computer Science, pp 51–78

Grishchenko I, Maffei M, Schneidewind C (2018c) A semantic framework for the security analysis of ethereum smart contracts. In: POST. Lecture Notes in Computer Science, vol 10804. Springer, pp 243–269

He J, Balunovic M, Ambroladze N, Tsankov P, Vechev MT (2019) Learning to fuzz from symbolic execution with application to smart contracts. In: CCS. ACM, pp 531–548

Hildenbrandt E, Saxena M, Rodrigues N, Zhu X, Daian P, Guth D, Moore BM, Park D, Zhang Y, Stefanescu A, Rosu G (2018) KEVM: a complete formal semantics of the ethereum virtual machine. In: CSF. IEEE Computer Society, pp 204–217

Jiao J, Kan S, Lin S, Sanán D, Liu Y, Sun J (2020) Semantic understanding of smart contracts: Executable operational semantics of solidity. In: IEEE S&P. IEEE, pp 1695–1712

Kalra S, Goel S, Dhawan M, Sharma S (2018) ZEUS: analyzing safety of smart contracts. In: NDSS. The Internet Society

Kasampalis T, Guth D, Moore B, Serbanuta T, Serbanuta V, Filaretti D, Rosu G, Johnson R (2018) Iele: an intermediate-level blockchain language designed and implemented using formal semantics. Tech. rep.

Krupp J, Rossow C (2018) Teether: gnawing at ethereum to automatically exploit smart contracts. In: USENIX security symposium. USENIX Association, pp 1317–1333

Liu C, Liu H, Cao Z, Chen Z, Chen B, Roscoe B (2018) Reguard: finding reentrancy bugs in smart contracts. In: ICSE. ACM, pp 65–68

Lu N, Wang B, Zhang Y, Shi W, Esposito C (2019) Neucheck: a more practical ethereum smart contract security analysis tool. Softw: Pract Exp

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

Nguyen TD, Pham LH, Sun J, Lin Y, Minh QT (2020) sfuzz: an efficient adaptive fuzzer for solidity smart contracts. In: ICSE. ACM, pp 778–788

Nipkow T, Paulson LC, Wenzel M (2283) Isabelle/HOL—a proof assistant for higher-order logic. In: Lecture Notes in Computer Science. Springer

Permenev A, Dimitrov D, Tsankov P, Drachsler-Cohen D, Vechev MT (2020) Verx: safety verification of smart contracts. In: IEEE symposium on security and privacy. IEEE, pp 1661–1677

Reis JS, Crocker PA, de Sousa SM (2020) Tezla, an intermediate representation for static analysis of michelson smart contracts. In: FMBC@CAV, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, OASIcs, vol 84, pp 4:1–4:12

Rodler M, Li W, Karame GO, Davi L (2019) Sereum: protecting existing smart contracts against re-entrancy attacks. In: NDSS. The Internet Society

Schneidewind C, Grishchenko I, Scherer M, Maffei M (2020) Ethor: practical and provably sound static analysis of ethereum smart contracts. In: CCS. ACM, pp 621–640

Sergey I, Hobor A (2017) A concurrent perspective on smart contracts. In: Financial cryptography workshops. Lecture Notes In Computer Science, vol 10323. Springer, pp 478–493

Sergey I, Kumar A, Hobor A (2018) Scilla: a smart contract intermediate-level language. CoRR. arXiv:1801.00687

Software C (2020) Security analysis tool for evm bytecode. [EB/OL]. https://github.com/ConsenSys/mythril. Accessed 1 May 2021

Solidity (2020) Solidity v0.5.0 breaking changes. [EB/OL]. https://docs.soliditylang.org/en/v0.5.0/050-breaking-changes.html. Accessed 1 May 2021

SRI Lab EZ (2020) Securify v2.0. [EB/OL]. https://github.com/eth-sri/securify2. Accessed 1 May 2021

Team V (2020) Vyper documentation. [EB/OL]. https://vyper.readthedocs.io/en/latest/. Accessed 1 May 2021

Tezos (2020) Michelson: the language of smart contracts in dune. [EB/OL]. https://www.liquidity-lang.org/doc/reference/michelson.html. Accessed 1 May 2021

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

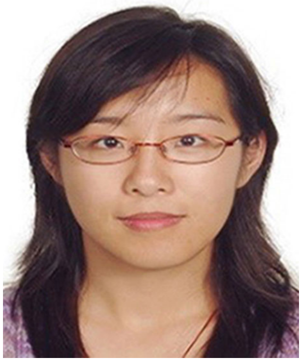
Torres CF, Schütte J, State R (2018) Osiris: hunting for integer bugs in ethereum smart contracts. In: ACSAC. ACM, pp 664–676

Tsankov P, Dan AM, Drachsler-Cohen D, Gervais A, Bünzli F, Vechev MT (2018) Securify: practical security analysis of smart contracts. In: CCS. ACM, pp 67–82

Wood G et al (2014) Ethereum: a secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper 151(2014):1–32

**Zhaoxuan Li** is a Ph.D. student in State Key Laboratory of Information Security (SKLOIS), Institute of Information Engineering (IIE), Chinese Academy of Sciences (CAS), Beijing, China. His research interests include blockchain security, formal methods, and privacy-preserving.

**Siqi Lu** is a lecturer and a Ph.D. student at Information Engineering University, Zhengzhou, China. He obtained M.Sc. in Cryptography from Information Engineering University, Zhengzhou, China, in 2014. His research interests include formal methods, cryptographic protocol, and big data security.



**Rui Zhang** is an associate researcher with SKLOIS, IIE, CAS, China. She received the Ph.D degree in information security from Beijing Jiaotong University, China, in 2011. She was a post-doctor in Institute of Software, CAS from 2011 to 2013. She was a visiting scholar in Georgia Institute of Technology from 2009 to 2010 and 2018 to 2019. She has published more than 40 technical papers in international journals and conference proceedings. Her research interests include blockchain security, security protocol, and applied cryptography.

**Rui Xue** is currently a research professor and vice director with the SKLOIS, IIE, CAS. He is a member of the IEEE, and a member of the ACM. He serves the vice director member of security protocols association in Chinese Association for Cryptologic Research. He has published more than 150 papers in popular journals and international conferences. His research interests include information security and privacy in data and information systems, with a focus on public-key encryption and cryptographic protocols.

**Wenqiu Ma** is an M.D. student in SKLOIS, IIE, CAS, Beijing, China. Her research interests include cryptographic protocols and blockchain security.

**Rujin Liang** is an M.D. student in Information Engineering University, Zhengzhou, China. He obtained B.Sc. in Cryptography from Information Engineering University in 2020. His research interests include formal methods and blockchain security.

**Ziming Zhao** is an M.D. student in Zhejiang University, Hangzhou, China. His research interests include machine learning, traffic identification, and privacy-preserving.

**Sheng Gao** is currently an Associate Professor with the School of Information, Central University of Finance and Economics. He received a Ph.D. degree in computer science and technology from Xidian University in 2014. He is a member of the IEEE. He has published over 30 articles in refereed international journals and conferences. His current research interests include data security, privacy computing, and blockchain technology.

## Affiliations

**Zhaoxuan Li[1,2] · Siqi Lu[3,4] · Rui Zhang[1,2] · Rui Xue[1,2] · Wenqiu Ma[1,2] · Rujin Liang[3,4] · Ziming Zhao[5] · Sheng Gao[6]**

Zhaoxuan Li
lizhaoxuan@iie.ac.cn

Rui Zhang
zhangrui@iie.ac.cn

Rui Xue
xuerui@iie.ac.cn

Wenqiu Ma
mawenqiu@iie.ac.cn

Rujin Liang
coderlrj@163.com

Ziming Zhao
zhaoziming@zju.edu.cn

Sheng Gao
sgao@cufe.edu.cn

1 State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, 100093, China

2 School of Cyber Security, University of Chinese Academy of Sciences, Beijing, 100049, China

3 Information Engineering University, Zhengzhou, 450001, China

4 Henan Key Laboratory of Network Cryptography Technology, Zhengzhou, 450001, China

5 College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, 310027, China

6 School of Information, Central University of Finance and Economics, Beijing, 100081, China