SmartState: Detecting State-Reverting Vulnerabilities in Smart Contracts via Fine-Grained State-Dependency Analysis

Zeqin Liao
Sun Yat-sen University
Guangzhou, China
liaozq8@mail2.sysu.edu.cn

Sicheng Hao
Sun Yat-sen University
Guangzhou, China
haosch@mail2.sysu.edu.cn

Yuhong Nan*
Sun Yat-sen University
Guangzhou, China
nanyh@mail.sysu.edu.cn

Zibin Zheng
Sun Yat-sen University
Guangzhou, China
zhzibin@mail.sysu.edu.cn

ABSTRACT

Smart contracts written in Solidity are widely used in different blockchain platforms such as Ethereum, TRON and BNB Chain. One of the unique designs in Solidity smart contracts is its state-reverting mechanism for error handling and access control. Unfortunately, a number of recent security incidents showed that adversaries also utilize this mechanism to manipulate critical states of smart contracts, and hence, bring security consequences such as illegal profit-gain and Deny-of-Service (DoS). In this paper, we call such vulnerabilities as the State-reverting Vulnerability (SRV). Automatically identifying SRVs poses unique challenges, as it requires an in-depth analysis and understanding of the state-dependency relations in smart contracts.

This paper presents SmartState, a new framework for detecting state-reverting vulnerability in Solidity smart contracts via fine-grained state-dependency analysis. SmartState integrates a set of novel mechanisms to ensure its effectiveness. Particularly, SmartState extracts state dependencies from both contract bytecode and historical transactions. Both of them are critical for inferring dependencies related to SRVs. Further, SmartState models the generic patterns of SRVs (i.e., profit-gain and DoS) as SRV indicators, and hence effectively identify SRVs based on the constructed state-dependency graph. To evaluate SmartState, we manually annotated a ground-truth dataset which contains 91 SRVs in the real world. Evaluation results showed that SmartState achieves a precision of 87.23% and a recall of 89.13%. In addition, SmartState successfully identifies 406 new SRVs from 47,351 real-world smart contracts. 11 of these SRVs are from popular smart contracts with high transaction amounts (i.e., top 2000). In total, our reported SRVs affect a total amount of digital assets worth 428,600 USD.

*corresponding author

CCS CONCEPTS

• **Software and its engineering** → **Software creation and management**; • **Software creation and management** → **Software verification and validation**;

KEYWORDS

bug finding, smart contract, static analysis, state dependency

1 INTRODUCTION

Smart contracts are a specific type of program running on the blockchain system. Solidity, one of the most popular programming languages for writing smart contracts, is widely used in major blockchain platforms such as Ethereum [4], TRON [9], and BNB Chain [2]. Smart contracts are now supporting a wide range of Decentralized applications (DApps) in blockchain, such as decentralized finance (DeFi), decentralized gaming (GameFi), and Non-Fungible-Token (NFT). Since most blockchain systems hold a considerable value of digital assets, the security of smart contracts is of great importance to both smart contract owners (e.g., DApp developers) and users. For example, in the well-known DAO attack [31], the attacker exploited a smart contract vulnerability (i.e., reentrancy) and caused an economic loss of 60 million USD.

State-reverting and its security implications. In Solidity, state-reverting is a specific mechanism to support error handling and access control [7]. More specifically, if an unsatisfied condition meets in the middle of a transaction, state-reverting allows the state variables in a smart contract to rollback to their original states. Unfortunately, a number of recent security incidents [1, 6] showed that, smart contracts might contain vulnerable code patterns when implementing state-reverting, allowing adversaries to utilize this mechanism and manipulate certain critical states by fabricating transaction errors. Such attacks can cause severe consequences to the victim smart contracts, such as illegal profit-gain [23] and

Deny-of-Service (DoS) [18]. In our research, we refer to such vulnerabilities in smart contracts as the State-reverting Vulnerability (SRV).

Given the severe impact of SRVs, there has been limited research aiming at SRVs. Specifically, EOSAFE [23] and WASAI [13] focus on detecting the rollback vulnerability caused by state-reverting through symbolic analysis and fuzzing, respectively. Besides, eTainter [19] and Madmax [20] detect Deny-of-Service (DoS) vulnerabilities caused by state-reverting based on abnormal gas consumption. These approaches can only cover a subtype of SRV (i.e., profit-gain or DoS, see more discussion on Section 2.2). In addition, both EOSAFE [23] and WASAI [13] are designed for smart contracts in the WASM language. Since their detection heuristics rely on language-specific features (i.e., the inline mechanism in WASM), these approaches are not applicable to Solidity smart contracts.

Our work. In this paper, to fill the above research gap, we propose SmartState, a new framework for detecting state-reverting vulnerability in Solidity smart contracts. To the best of our knowledge, SmartState is the first of its kind to support SRV detection in a generic manner at the bytecode level.

The key challenge in this research is to construct a fine-grained state-dependency graph, which is mandatory for determining which state could be affected or manipulated by an adversary from the attack surfaces (e.g., function calls from other contracts). However, state-of-the-art approaches can only infer limited state dependency information, which is not sufficient for detecting SRVs. For example, SAILFISH [11] only covers certain explicit dependencies (i.e., state Read and Write (R&W) dependency and control flow dependency) for detecting state inconsistency bugs in smart contracts.

To effectively detect SRVs, SmartState identifies two types of state-dependency relations which are missed by prior research, namely, assertion-related state dependency (ASD) and temporal-order state dependency (TSD). Here, ASD refers to explicit dependencies related to statements such as `assert`, `revert`, and `require` for access control, while TSD refers to dependencies affected by transactions (i.e., function invocations) which is unique to smart contracts. To this end, SmartState infers such important dependencies by analyzing both the smart contract bytecode, as well as their corresponding historical transactions (see Section 4.1 and 4.2 for more details). In this way, SmartState combines the extracted dependencies and constructs a fine-grained state-dependency graph for SRV detection. Further, SmartState models the generic patterns of SRVs (i.e., profit-gain and DoS) as SRV indicators. For example, SmartState detects whether a critical state in the SDG lacks appropriate access control checks. Therefore, it effectively identifies SRVs based on the constructed state-dependency graph.

To evaluate SmartState, we have constructed a manually-labeled dataset which consists of 91 SRVs collected from public reports and manual investigation over the top-popular smart contracts. Our evaluation results showed that SmartState achieves a precision of 87.23% and recall of 89.13%, indicating its high effectiveness in detecting SRVs. In addition, by running SmartState over 2,000 popular smart contracts in the real world, our research successfully identified 11 new SRVs which have not been identified by previous research. The total assets affected by these SRVs reached 428,600 USD.

In summary, this paper makes the following contributions.

- We propose SmartState, a novel framework for detecting state-reverting vulnerabilities based on static analysis. To the best of our knowledge, SmartState is the first of its kind to detect SRVs in a generic manner.
- We propose a set of new mechanisms (i.e., assertion-related dependency and transaction-order dependency) to construct fine-grained state dependency graph in smart contracts.
- We perform extensive evaluation to show the effectiveness of SmartState. In addition, by performing a large-scale study over 47,351 smart contracts in the wild, SmartState identified 406 new SRVs in the real world.
- We release the artifact of SmartState, as well as the manual-labeled SRV dataset [1].

2 BACKGROUND AND MOTIVATION

In this section, we first lay out some basic background knowledge about smart contracts and contract states. Then, we present the problem statement and the motivation of our research.

2.1 Smart Contract and Contract State

Smart contracts are a specific type of program running on the blockchain. At present, most smart contracts are written in Solidity [7]. With the Solidity source code, a smart contract is then compiled into the bytecode and executed on the Virtual Machine (VM) of different blockchain platforms (e.g., Etherum [4], TRON [9], and BNB Chain [2]). Similar to other program languages such as Python and Java, smart contracts consist of a set of functions and variables. Functions can be invoked through calls from smart contracts and user accounts. Such a function invocation is also known as *transactions*.

Contract states are persistent data stored and accessed via global variables (i.e., state variables) in smart contracts. A transaction (function invocation) actually changes the state(s) of related smart contracts, and this process is permanently recorded on blockchain [7]. Due to the limited storage space on the blockchain, smart contracts use state variables to store critical data, such as the owner's address, users' token balance, etc. Therefore, if an adversary could manipulate these critical states, it may bring severe negative impacts (e.g., financial losses) to the contract owner. For example, in the well-known Fomo3D attack, the attacker leverages a *DoS* vulnerability to stop the purchase of other contract users (i.e., roll back the modification on the state variable that represents purchase), which caused an economic loss of 43 million USD [1].

State-reverting mechanism. State-reverting is a unique mechanism for error handling and access control in smart contracts. If an unsatisfied condition meets in the middle of a transaction, all contract states related to this transaction can be rolled back to their previous values before the transaction. Such a state-reverting mechanism is essential for the scenario of smart contracts, as it ensures the atomicity of transactions in smart contracts. In smart contracts, state-reverting can be implemented by invoking the assertion statement (i.e., `require`, `assert`, and `revert`). For example, a smart contract with *"require(tx.origin = msg.sender)"* indicates that

---

[1]https://github.com/InPlusLab/SmartState

```solidity
1  contract TokenGame {
2    mapping(address => uint256) public SheepToken;
3    mapping(address => uint256) public WolfToken;
4    function MintToken(address account) public {
5      uint256 seed= (random () >> 245) % 10;
6      //A random number determines gambling results
7      if ( seed != 0) {
8        SheepToken[account]++;}
9      else{
10       //The state variable manipulated by attacker
11       WolfToken[account]++;}
12   }
13 }
14
15 contract Attacker {
16   function onlyWolf(TokenGame target, tokenId) public{
17     unit256 Before = WolfToken.balanceOf(address(this));
18     target.MintToken(tokenId);
19     unit256 After = WolfToken.balanceOf(address(this));
20     // Reverting the unexpected gambling result
21     require(After > Before); }
22 }
```

Figure 1: An example of state-reverting vulnerability and how attacker exploits it for profit-gain.

the function can only be invoked by an external owned account (EOA) [5], if the caller of the function is not an EOA (e.g., contract account), all other state changes made in this transaction will be rolled back.

2.2 Problem Statement

In this paper, our research focuses on vulnerabilities related to the state-reverting mechanism. We call this type of vulnerability as the State-reverting vulnerability (SRV).

State-reverting vulnerability (SRV). Recent studies and reports [23] showed that the state-reverting mechanism is frequently utilized by attackers and causes severe attacks and security incidents. More specifically, in these attacks, the adversary uses the state-reverting mechanism and makes the transaction fall back, if the result of the transaction is not as expected (e.g., benefits the attacker). State-reverting attacks are more prevalent in GameFi markets, affecting game fairness and bringing heavy financial losses to contract owners [13] or GameFi users.

Motivating example. Figure 1 shows an example of smart contract with state-reverting vulnerability which affects game fairness, as well as how the attacker obtains illegal profits by manipulating contract states. In this example, the state variables, *SheepToken* and *WolfToken* are two types of tokens with different values (i.e., *WolfToken* is more expensive than sheep). A random number (seed) determines the gambling result, with a 90% chance to get a *SheepToken* and a 10% chance to get the *WolfToken* (line 7-11). Unfortunately, due to the lack of appropriate access control, an adversary can arbitrarily check the balance of these tokens before and after the gambling game (line 17-19). To maximize gain in the game, the adversary uses the require statement (line 21) to revert the whole transaction (i.e., *onlyWolf*), if he is rewarded with the low-value *SheepToken*. In this way, the attacker ensures he always gets the *WolfToken* and brakes the game fairness with more profits.
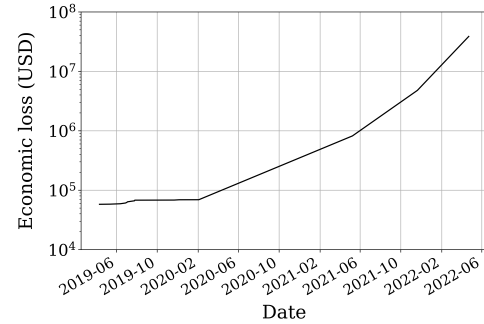

Figure 2: Summarized Economic loss caused by state-reverting vulnerability in recent three years. The sources of the collected incidents are summarized in [8].

Figure 2 summarizes the accumulated financial losses caused by state-reverting attacks in recent three years. The sources of such statistics and corresponding incidents are listed in [8]. As can be seen, the economic loss caused by SRVs is increasing rapidly and has reached 38.69 million USD as of April 2022.

Prior research and their limitations. Despite reported attack incidents caused by SRVs [1], there have been limited prior works on identifying SRV in advance and further eliminating such losses. To the best of our knowledge, the most closely related work could be EOSafe [23] and WASAI [13], which mainly detect the rollback attack based on symbolic execution and fuzzing, respectively. However, since both frameworks do not consider important state dependencies in smart contracts, they can only detect a sub-type of state-reverting related attacks (i.e., rollback which causes profit-gain). Similarly, eTainter [19] and Madmax [20] propose a framework to detect DoS vulnerabilities which are also caused by state-reverting. Moreover, both EOSafe [23] and WASAI [13], are specifically designed for smart contracts in WASM language [10]. Due to the fundamental differences between Solidity contracts and WASM contracts, these frameworks can not be applied to detecting SRVs in Solidity. For example, to detect rollback vulnerability, EOSafe [23] relies on locating specific statement *send_inline* for invoking an inline action in WASM smart contracts. However, in Solidity there are no such inline actions as well as the *send_inline* keyword.

2.3 Scope of Our Work

SmartState is designed to be a generic framework for detecting SRV vulnerabilities in Solidity smart contracts. The assumption of SmartState is similar to other smart contract vulnerability detection frameworks, such as SmartDagger [29] and Slither [18]. More specifically, SmartState statically analyzes smart contract bytecode and precisely reports whether a given smart contract contains state-reverting vulnerabilities. Since SmartState performs analysis at the bytecode level rather than source code, it is applicable for a number of security vetting scenarios such as large-scale third-party auditing, self-security inspection, etc. Compared to other research [13, 19, 23] which only focuses on specific blockchain platforms (e.g., eTainter for Ethereum and EOSafe for EOSIO [3]), SmartState can benefit more blockchain ecosystem that based on Solidity smart contracts (e.g., Etherum [4], Tron [9], and BNB Chain [2]).

```
1 contract TokenGame {
2   mapping(address => uint256) public SheepToken;
3   mapping(address => uint256) public WolfToken;
4   mapping(address => uint256) public Earning;
5   ...
6   function MintToken(address account) public {
7     uint256 seed= (random () >> 245) % 10;
8     if ( seed != 0) {
9       SheepToken[account]++;}
10    else{
11      WolfToken[account]++;} }
12
13  function Withdraw(address account,unit amount) public{
14    require(SheepToken[account]>0||WolfToken[account]>0);
15    tranferForm(address(this), account, amount); }
16
17  function PlaytoEarn(address account,unit tokenId)
          public{
18    if(isWolf(tokenTraits[tokenId]))
19      Earning[account]=Earning[account]*(2-Rate); }
20  ...
21 }
```

Figure 3: The example for vulnerable smart contract with two types of state dependency.

3 DESIGN OF SMARTSTATE

In this section, we present the high-level design of SmartState. We first highlight the key challenges in detecting SRV. Then, we present how SmartState overcomes these challenges in detail.

As mentioned earlier, the root cause of SRVs is that certain state variables can be affected or manipulated by external calls via the state-reverting mechanism. Therefore, a straightforward idea to identify SRVs has two key steps: (1) Identify state dependencies across different state variables and function invocations, and (2) Identify SRVs based on the impacts of such dependencies.

(1). **Identify state dependencies.** Similar to prior research for identifying state-inconsistency bugs [11], we can construct a state dependency graph that represents such dependency relationships across different smart contracts and function calls. Turning to the example shown in Figure 3, the state variables *SheepToken* and *WolfToken* can be written by the function *MintToken* (line 9, 11) and further accessed by the assert statement (line 14) of the function *Withdraw*. In this case, we say there is a state-dependency relationship between function *MintToken* and *Withdraw* for state variable *SheepToken* and *WolfToken*.

(2). **Identify SRVs**. For a specific security-sensitive state variable, we can check whether there is a valid path (call chain) along the state-dependency graph, allowing an external call to arbitrarily manipulate it. Again, taking the example in Figure 3, for the state variable *balance* modified in *Withdraw* (line 15), it can be actually affected by the adversary due to the dependency relation between *MintToken* and *Withdraw*. To this end, we identified that the contract *TokenGame* is with a state-reverting vulnerability.

3.1 Challenges and Solutions

With the increasing complexity of smart contracts, establishing an effective and complete state-dependency graph (SDG) for SRV

detection is by no means trivial. Previous research [11] performs state-dependency analysis for other purposes (e.g., SAILFISH for detecting state inconsistency bugs), these works only cover certain explicit dependencies such as state R&W relationship and related control flow, which are not sufficient for SRV detection. Our work complements and extends previous research by considering two new types of dependencies: namely, assert-related state dependency (ASD) and temporal-ordered state dependency (TSD), allowing us to detect new vulnerabilities related to state dependency (i.e., SRV).

C1: Extracting assertion-related state dependency (ASD). ASD refers to dependencies related to statements for access control. Solidity uses reserved keywords such as assert, revert, require for this purpose, and some states can be either explicitly (as function parameters), or implicitly (transaction revert) affected by such statements. If state S can be written by function f_A and further accessed by assert statement in another function f_B, we say there is an ASD between function f_A and f_B for state S. In other words, the function f_B can execute only when state S meets the required condition. Otherwise, function f_B fails, and the whole transaction will be reverted. Previous work [30] showed that 82.28% of smart contracts take assertion-related statements for access control. Unlike other explicit dependencies such as state read and write, extracting ASD requires analyzing specific control-flow and data-flow related to assert statements in Solidity. For example, in Figure 3, for state variables *SheepToken* and *WolfToken*, the state-dependency relationship between *MintToken* and *Withdraw* is an assertion-related state dependency, because function *Withdraw* depends on state variables *SheepToken* and *WolfToken* due to the effect of the assertion statement (i.e., control flow), and these state variables depend on function *MintToken* which writes on it (i.e., data flow).

To overcome this challenge, SmartState performs a fine-grained control-flow and data-flow analysis to recover the program logic by analyzing the smart contract bytecode. In this way, based on the semantics of specific assert statements in Solidity, SmartState establishes the dependencies across different functions for different contract states.

C2: Extracting temporal-ordered state dependency (TSD). As mentioned earlier, smart contracts are executed as transactions on the blockchain. TSD refers to the state dependencies caused by the transaction order of multiple transactions in smart contracts. More specifically, since contract users invoke the contract via a specific transaction sequence, the states produced by contract functions at a certain point could be affected by the temporal order of these transactions. For example, as shown in Figure 3, according to the transaction order of this smart contract, we know that the contract user needs to mint the token (i.e., invoking *MintToken*) before playing the game (i.e., invoking *PlaytoEarn*). Therefore, we say there is a temporal order dependency between *MintToken* and *PlaytoEarn*. The importance of TSD has been highlighted by prior works [27, 35, 36], as manipulating the transaction order of specific contracts can cause severe vulnerabilities or financial losses. Unfortunately, the semantics of such temporal order across functions can not be easily recovered at the smart contract bytecode as the transaction order is determined by the application scenarios of a specific smart contract that is related to its functionalities (e.g., a smart contract for gambling).
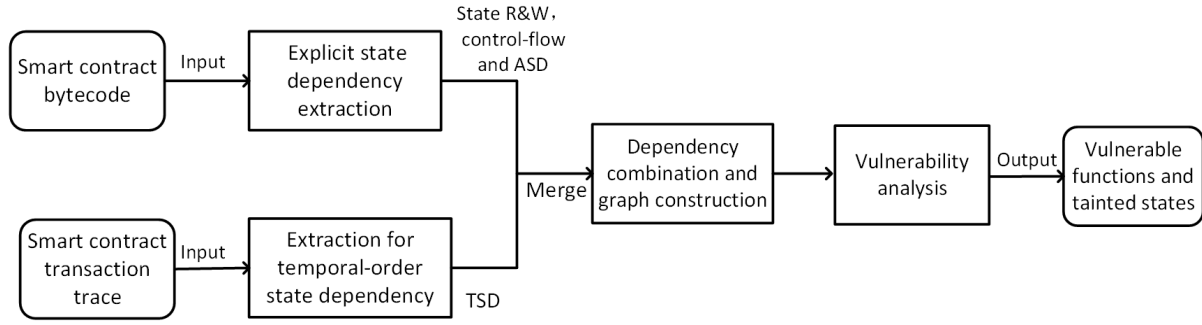
Figure 4: The workflow of SmartState.

To overcome this challenge, SmartState leverages the fact that TSD information is well-preserved on the transaction history of the smart contract, as contract users generally invoke the contract functions according to the specific transaction order. To this end, SmartState leverages a finite state machine to learn the temporal order of smart contract functions from the historical transaction traces, and further extracts the TSD.

C3: SRV identification. The last challenge lies in how to accurately identify SRVs based on the constructed SDG. As mentioned earlier, previous approaches [19, 20] mainly detect SRV-related vulnerabilities based on ad-hoc heuristics such as abnormal gas consumption [20]. These approaches are not generic and can only cover a sub-type of SRV vulnerabilities (e.g., DoS attacks caused by SRV [19]).

Different from prior research, SmartState highlights the root cause of SRVs and uses them as SRV indicators to support vulnerability detection in a generic manner. More specifically, SmartState considers the following two key elements as the SRV indicator: (1) Whether a security-sensitive state is non-deterministic, or it shares dependency relation with other non-deterministic states. For example, as discussed earlier in Figure 1, in the vulnerable function *MintToken*, the number of tokens (e.g., *wolveToken*) can be affected by state variable for storing token values can be affected by the statement that generates a random number. (2) Whether the sensitive state lacks appropriate access control. Based on the state-dependency graph of a particular state, SmartState detects if necessary access control is missing, for example, checking the origin address of a transaction or checking the balance of a specific token). Finally, if an external contract can be invoked and reaches states with the above SRV indicators, we consider the analyzed contract vulnerable.

3.2 Workflow of SmartState

SmartState takes both the smart contract bytecode and its corresponding transaction traces (i.e., history data) as its input, and finally reports the vulnerability as a set of vulnerability traces. A vulnerability trace contains function calls from the vulnerable function to tainted state variable(s) that can be affected by external call(s). Figure 4 shows the workflow of SmartState with the following steps:

S1. **Pre-processing and ASD extraction.** Similar to prior work [29], SmartState first recovers the control flow and data flow of the whole smart contract as the pre-processing step. Then, it identifies ASD from the program logic of smart contracts.

S2. **Temporal-ordered state dependency extraction.** In the second step, SmartState analyzes historical transaction traces and extracts the TSD.

S3. **Dependency combination and graph construction.** Then, SmartState merges four types of dependency, i.e., the state R&W dependency, control flow, ASD, and TSD, to generate the state dependency graph (SDG) for SRV detection.

S4. **Vulnerability detection.** Lastly, based on the constructed SDG, SmartState identifies all the SRV indicators and finds out vulnerability traces with vulnerability-specific rules.

4 APPROACH DETAILS

Now we elaborate on the details of each step in SmartState.

4.1 Pre-processing and ASD Extraction

Pre-processing. SmartState utilizes *SmartDagger* [29], a state-of-the-art static analysis tool to recover the control flow and data flow from the bytecode of a given smart contract. Since SmartDagger is designed for detecting cross-contract vulnerabilities, it can construct a more complete control flow and data flow for cross-functions (contracts) invocations, compared to other similar tools (e.g., Mythril [15], Slither [18]). Specifically, *SmartDagger* decompiles the bytecode of the smart contract to generate the intermediate representation (IR). Then, based on the IR and function invocation information, it outputs the constructed the control-flow and data-flow graph.

ASD extraction. After recovering the program logic, SmartState identifies the ASD from the extracted control flow and data flow. Here, SmartState first leverages a similar approach as in SAIL-FISH [11] to extract the basic read and write (R&W) dependency for state variables. In addition, SmartState extracts ASD with the following heuristics:

- A function M_r reads the state variable S_d as a condition within the assertion statement (i.e., revert, assert, require);
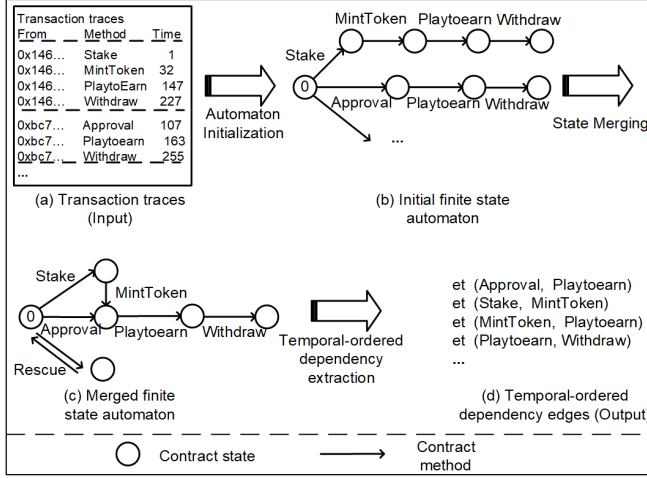
Figure 5: Details of the finite state machine.

Figure 6: Examples for illustrating transaction equivalence and transaction subsumption.

- There is another function M_w which writes on the same state variable S_d.

If the above two conditions are satisfied, SmartState determines that function M_r has an ASD on function M_s. To this end, it adds a directed edge $e_r(M_r, M_w)$ to denote this dependency. Finally, all ASD edges of the smart contract can be denoted as a set of directed edges $E_r = \{e_r(M_r, M_w)|M_r, M_w \in M\}$.

4.2 TSD Extraction

SmartState extracts the TSD from the historical transaction traces of smart contracts. Take the transaction traces in Figure 5 (a) as an example. The user "0x146..." executes a group of transactions in line with the specific order (i.e., *MintToken*, *PlaytoEarn*). Assuming that such transaction information is sufficiently effective, we can identify that *PlaytoEarn* has a TSD on *MintToken*. Obviously, practical transaction order is more complex. It is a demand for an effective approach to learning about the TSD from transaction traces.

SmartState leverages a finite state machine (FSM) [28] to extract TSD. A finite state machine is a mathematical model which can accurately describe the state transitions of specific subjects [17]. In our case, FSM is used to represent transitions of contract states during the execution of transactions. More specifically, each node represents a specific state of the smart contract with a set of state variables, each edge refers to a specific state transition caused by a transaction execution, and the value of each edge is the function invoked by the transaction. We denote this FSM as the following tuple (S, s_0, Tr, M, T) where

- S represents the set of contract states.
- $s_0 \in S$ represents the initial state of the smart contract.
- T_r represents the set of transactions in the smart contract.
- M represents the set of values that correspond to the functions invoked by the transactions.
- T represents the transition relation $T : S \times T_r \rightarrow S$

Note that each state transition is caused by the transaction that executes on the predecessor state and finally reaches the successor state, denoted as $T : S \times Tr \rightarrow S$.
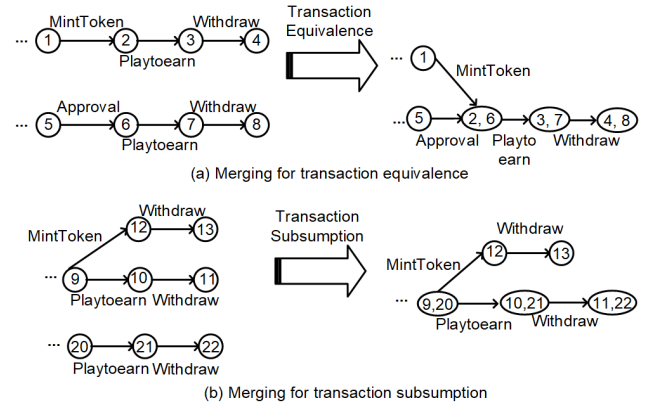
FSM construction. Below we introduce how the FSM works for TSD extraction. The input of FSM is a set of transaction traces. Specifically, each trace represents the transaction sequence that is invoked by a specific contract account, and each transaction contains information about the address of the user, invoking function, and invoking clock. The output of FSM is TSD edges, also denoted as a set of directed edges $E_t = \{e_t(M_{t1}, M_{t2})|M_{t1}, M_{t2} \in M\}$. For instance, Figure 5 (a) and (d) show an example of the transaction traces, and the finally extracted TSD edges, respectively.

The FSM generates such output through the following steps:

(1). **Initialization.** As shown in Figure 5 (b), SmartState first combines all transaction traces into a tree-shaped machine. In the machine, each trace corresponds to a single branch of the tree. Besides, the machine is labeled with relative functions that correspond to the transactions in the traces.

(2). **State merging.** Then, SmartState merges the states of the initial FSM via analyzing the transaction equivalence and transaction subsumption of different states. In this way, the FSM can be simplified into a sufficiently compact model. For instance, Figure 5 (b) and (c) show the FSM update before and after the merging process.

(3). **TSD extraction.** Finally, SmartState extracts the TSD from the merged FSM. Specifically, SmartState determines the TSD according to the temporal order of state transition in the FSM. For example, the temporal-ordered state dependency edges in Figure 5 (d).

State Merging. SmartState merges all the equivalent states and subsumed states in the initial FSM to shrink its size. This process is done in a way similar to methods proposed by prior works [17, 28]. We take two additional examples shown in Figure 6 to illustrate this process. As can be seen from part (a), transactions s_2 and s_6 are equivalent because the share the same future functions (i.e, $\{Playtoearn, Withdraw\}$). Therefore, transactions $[s_2, s_6]$, $[s_3, s_7]$, and $[s_4, s_8]$ are merged. Similarly, the process of transaction subsumption is presented in Figure 6(b).
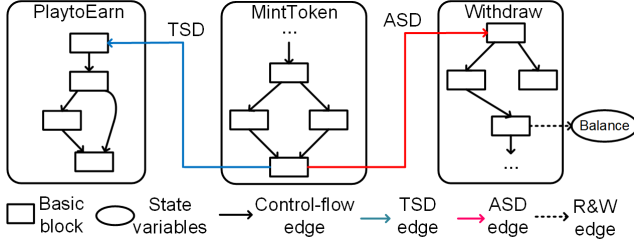
Figure 7: The process of fine-grained SDG construction for the running example described in Figure 3.

4.3 Dependency Graph Construction

SmartState merges the previously extracted control flow, state R&W dependency, ASD, and TSD together to construct the fine-grained state-dependency graph (SDG), to facilitate SRV detection in a more effective manner. Below we give more details about the SDG and its construction process.

Data structure. The SDG constructed by SmartState is denoted as a triple $G_s = (N_s, E_s, X)$. Specifically, SDG encodes the following information: (1) The nodes in SDG are a set of state variables and basic block nodes representing program operations. Here, S denotes the set of state variables, B denotes the set of basic block nodes. We include basic blocks as SDG nodes because basic blocks provides important dependency information related to state variables (i.e., function invocation). Therefore, we say $N_s := \{S \cup B\}$. (2) The edges in SDG are a set of control-flow edges, state R&W dependency edges, ASD edges, and TSD edges. $X(E_s) \rightarrow \{C, R\&W, ASD, TSD\}$ is a labeling function that maps an edge to one of the four types.

SDG construction. SmartState constructs SDG by incrementally adding ASD edges, TSD edges, and state R&W dependency edges to the control flow graph. For each ASD, SmartState searches the control-flow graph and finds out the source and target basic blocks, and connects them with a directed edge. Note that if function A has an ASD on function B, the source of such ASD is the end site (i.e., the last statement such as *return*) of function B, and the target of ASD is the start cite of function A. After that, SmartState adds the directed edge between the source and target for each ASD. Similarly, SmartState adds the TSD edges to the graph in the same manner. For state R&W dependency edges, SmartState searches for basic block nodes that read or write the state variables and adds the directed edge between these basic blocks and state variables. Figure 7 shows an example of the constructed SDG. As can be seen, the ASD edges and TSD edges are labeled with different colors.

4.4 Vulnerability Detection

SRV indicator. Unlike prior works that detect SRV by identifying the side effect of a specific sub-scenario (e.g., identifying gas sufficiency), SmartState takes a set of generic rules for locating vulnerable functions according to the root cause of SRV. More specifically, for each function in smart contracts, SmartState identify it as an SRV indicator (i.e., vulnerable function) only if (1) the state variables can produce uncertainty or have state dependency with other uncertainty states and (2) along the state dependency path, there

is lack of correct access control. We formalize the generic rules as vulnerable indicator as follows.

$$isUncertainty(var_{state}) \land isLackOf(C_{acc}) \text{ for each } f \quad (1)$$

In this formula, f is the function in the smart contract, var_{state} represents the state variable in the function f, and C_{acc} refers to the correct access control condition. Particularly, the former condition is scenario-specific, and the latter condition is generic to all scenarios of SRV. Further, based on the above formula, table 1 summarizes the detailed rules for analyzing two sub-scenarios of SRV covered by SmartState.

Table 1: Vulnerability indicator rules for locating two sub-scenarios of SRV

Vulnerability type	Vulnerability indicator rule
R1-Profit-gain attack	$isRandomness(var_{state}) \land isLackOf(C_{acc})$
R2-DoS attack	$(isInLoop(externalcall)$ $\land isModified(var_{state})) \land isLackOf(C_{acc})$

With the above-illustrated SRV indicator, SmartState takes three key steps to detect SRV.

Step-1: Finding SRV indicator(s). Firstly, SmartState screens for the SRV indicators on the SDG based on our proposed generic rules. This process is modeled as a process of graph query to locate the vulnerable function from SDG. For example, for a profit-gain attack, SmartState inspects whether there exist statements that randomly modify a balance variable (e.g., whether statements with the EVM instruction SSTORE is in an execution lock depending on randomness) as well as the lack of access control for forbidding arbitrary external calls.

Step-2: Finding the entry trace of external contract. After identifying the SRV indicators, SmartState searches for the entry trace for each SRV indicator and utilizes taint analysis to inspect whether it can be accessed by an external attacker. To model this execution flow, SmartState makes the taint propagate from the entry point (e.g., public function) of the contract, and observes whether the taint can propagate to the vulnerability indicators.

Step-3: Finding tainted state variable(s) affected by state-reverting. After finding the entry trace, SmartState continues to perform the forward taint propagation on the SDG and computes the tainted state variables. Apparently, these state variables can be affected by the subverted flow of the vulnerability. Finally, all the tainted functions and state variables are reported as the vulnerable trace that describes the SRV.

To perform taint propagation, the taint sources can be divide into two types: the parameters passed by contract callers and parameters of public functions. The taint sinks of SmartState consist of either those external calls, or state variables of the smart contracts (including the SRV indicators). More detailed information regarding the taint sources and sinks are summarized in Table 2.

Again, we take the running example shown in Figure 8 as an instance to show this process. For this smart contract, SmartState searches for the SRV indicators on the SDG. SmartState identifies that the function *MintToken* satisfies Step-1 because *MintToken* utilizes a random number (seed) which determines the modification

Table 2: EVM instructions defined as taint sources and taint sinks by SmartState.

	Type	EVM Instruction or Keyword or Statement
Source	(1) Parameter passed by contract invoker	CALLDATALOAD, CALLDATACOPY, CALLER, ORIGIN, CALLVALUE, CALLDATASIZE
	(2) Parameter of public function	Public, External
Sink	(1) External calls	CALL, CALLCODE, STATICCALL, DELEGATECALL
	(2) State variables	SSTORE, BALANCE, ADDRESS, SRV indicators

on state variable (i.e., SheepToken and Wolfoken) without access control. Then, SmartState performs the Step-2, and identifies that *MintToken* is a public function that can be accessed by an external attacker. Lastly, SmartState performs the Step-3 by screening for state variables tainted by the vulnerability through data flow analysis. As a result, SmartState reports the vulnerability traces in contract *TokenGame* as (1) $MintToken \rightarrow \{SheepToken, WolfToken\}$, (2) $MintToken \rightarrow Playtoearn \rightarrow \{Earning\}$, (3) $MintToken \rightarrow Withdraw \rightarrow \{Balance\}$.

```
1 contract TokenGame {
2   mapping(address => uint256) public SheepToken;
3   mapping(address => uint256) public WolfToken;
4   mapping(address => uint256) public Earning;
5   ...                                    Entry point of
6   function MintToken(address account) public {  external
7     uint256 seed= (random () >> 245) % 10;       contract
8     if ( seed != 0) {
9       SheepToken[account]++;}           SRV indicator
10    else{
11      WolfToken[account]++;} }
12
13  function Withdraw(address account,unit amount) public{
14    require(SheepToken[account]>0||WolfToken[account]>0);
15    tranferForm(address(this), account, amount); }
16                                          Tainted sink
17  function PlaytoEarn(address account,unit tokenId)
        public{
18    if(isWolf(tokenTraits[tokenId]))
19      Earning[account]=Earning[account]*(2-Rate); } }
20  ...                                     Tainted sink
21 }
```

Figure 8: The process of vulnerability detection for the motivating example.

5 EVALUATION

In this section, we first present our experimental setup and the two datasets for evaluation (the manual-labeled SRV dataset and the large-scale smart contracts dataset). Then, we show the evaluation results of SmartState in terms of false positives and false negatives over the manual-labeled SRV dataset. Lastly, we discuss the results of our large-scale analysis and identified new SRVs in the wild.

5.1 Implementation and Evaluation Setup

We implement SmartState with around 3,400 Line-of-Code in Python 3.8.10. All the experiments in our evaluation are conducted on an Ubuntu 20.04 server that is equipped with one Intel i9-10980XE CPU (3.0GHz), one RTX3090 GPU, and 250 GB RAM.

Dataset and ground-truth establishment. We utilize the following two datasets for our evaluation experiment.

Manual-labeled SRV Dataset (D_{srv}). This dataset constructs the ground truth for evaluating the effectiveness of SmartState. More specifically, we manually collected and annotated a total number of 91 SRVs from 47 smart contracts[2]. Particularly, 27 SRVs belong to *profit-gain* and the other 64 SRVs belong to *DoS*. Among this dataset, 17 SRVs (from 11 smart contracts) are collected based on public-reported attack incidents. To the best of our knowledge, this is the most comprehensive collection of SRVs from the public sources. We manually locate the related SRVs by reviewing such reports. The rest 74 SRVs (from 36 smart contracts) are manually collected by inspecting financial-related contracts (e.g., Wallet and Gambling) from popular DApps. To avoid bias, three researchers participated in the annotation process. Every researcher separately performs the annotation. Only the vulnerability agreed by all three researchers are confirmed as a valid SRV. We selectively explored financial applications for building D_{srv}, as their SRVs may cause severe impact (e.g., direct money losses). However, this does not mean SmartState is specifically tailored or biased to financial applications, as the patterns of SRVs are generic to all contracts.

Large-scale Dataset (D_{large}). To show the effectiveness of SmartState in finding SRVs in the real world, we utilize the second dataset, which contains 47,351 smart contracts. The dataset is an open dataset that is proposed in the well-known empirical study [16].

Evaluation Metrics. We summarize the following research questions (RQs) to evaluate SmartState.

RQ1. How effective is SmartState in terms of detecting the state-reverting vulnerability?

RQ2. For state dependency analysis, how do the extracted ASD and TSD help for detecting SRV?

RQ3. How does SmartState perform compared to other state-of-the-art mechanisms in terms of detecting SRV?

RQ4. Can SmartState detect SRV from real-world smart contracts?

Table 3: Overall effectiveness for SmartState on the Manual-labelled SRV Dataset (D_{srv}).

Attack exploits SRV	Precision			Recall		
	TP	FP	rate	TP	FN	rate
Profit-gain attack	24	5	82.76%	24	4	85.71%
DoS attack	58	7	89.23%	58	6	90.63%
Total	82	12	87.23%	82	10	89.13%

---

[2]each smart contract has a unique address in Ethereum

Table 4: Comparison results between SmartState and the other two baselines over the Manual-labeled SRV Dataset (D_{srv}).

Approach	SmartState w/o ASD and TSD			SmartState w/o TSD			SmartState		
	TP	FN	recall	TP	FN	recall	TP	FN	recall
Profit-gain attack	12	16	42.86%	18	10	64.28%	24	4	85.71%
Dos attack	42	22	65.63%	53	11	82.81%	58	6	90.63%
Total	54	38	58.70%	71	21	77.17%	82	10	89.13%

5.2 Effectiveness of SmartState

To answer RQ1, we ran SmartState on the manual-labeled SRV Dataset (D_{srv}) to evaluate its precision and recall. For example, we give the same time budget (i.e., 10-mins timeout) for every smart contract in the dataset. The precision (false positive) and recall (false negatives) are computed by manually inspecting the reported results and comparing these results with the ground-truth data of D_{srv} (i.e., 91 vulnerabilities in the 47 contracts).

Table 3 presents the overall results. As can be seen, SmartState achieves a high recall (i.e., 89.13%) and precision (i.e., 87.23%).

False positives and false negatives. We manually inspect the false positives and false negatives introduced by SmartState. Among 12 false positives, our further inspection shows that most of them are caused by the limitation of control flow analysis produced by SmartDagger (i.e., the existing analyzer used in SmartState). For example, SmartState reports the false results of vulnerable functions due to the fact that SmartDagger fails to locate the function borders. To address these false positives, SmartState can be improved by integrating a more advanced bytecode analyzer for recovering function boarders. For the 10 false negatives, most of them are because they rely on third-party data which is not controlled by the blockchain platform (e.g., a pseudo-random number that relies on oracle). In fact, the problem can not be addressed by any static analysis approach like ours, as they require third-party data outside the blockchain.

5.3 Effectiveness of ASD and TSD

To answer RQ2, we evaluate the effectiveness of individual components in SmartState, i.e., ASD and TSD. As mentioned earlier in Section 4.1 and 4.2, ASD and TSD are the two important advantages possessed by SmartState, which ensure the soundness of vulnerability analysis (i.e., avoiding false negatives). For instance, SmartState can utilize such advantages to perform more taint tracking so that it can identify more vulnerability traces, in contrast to those approaches without considering ASD and TSD. Hence, the effectiveness of ASD and TSD is reflected on the recall rate. We compare the current design with two baseline approaches. More specifically, SmartState without considering ASD&TSD, as well as SmartState without considering TSD. We ran these baseline frameworks on D_{srv}.

Compare to SmartState without TSD. We evaluate the effectiveness of TSD by comparing SmartState to the baseline approach without considering TSD. The column 2 and 3 in table 4 show the recall rate for this comparison. Due to ignoring the TSD, the total recall rate of the baseline approach without considering TSD is only 77.17%, the recall rates of both *Profit-gain attack* and *Dos attack* drop obviously. Particularly, the recall rates of *Profit-gain attack*

drop more rapidly than that of *DoS attack*. In contrast, SmartState maintains a good performance (over 85%) in terms of two types of scenarios. In conclusion, the extracted TSD effectively helps SmartState improve the recall for SRV detection. Particularly, the extracted TSD is more important for the detection of *Profit-gain attack* than *DoS attack*.

Compare to SmartState without TSD and ASD. After that, we further evaluate the effectiveness of ASD by performing the comparison between SmartState and the other baseline approach without considering ASD and TSD. The columns 1 and 3 in table 4 shows the recall rate for this comparison. Due to ignoring ASD and TSD, the total recall rate of this baseline approach is only 58.70%, the recall rates of *Profit-gain attack* and *DoS attack* drop more rapidly. And this baseline approach produces more false negative (i.e., a total of 17 new false negatives). To sum up, the extracted ASD also helps SmartState improve the recall for SRV detection significantly.

Further, we manually inspected every false negative reported by two baseline approaches. The inspected results show that 17 (i.e., 44.74%) of the 38 false negative results can be avoided by analyzing ASD, and 11 (i.e., 28.95%) of the 38 false negative results can be avoided by analyzing the TSD, which are missed by these baseline approaches. For instance, Figure 9 shows an example of false negatives, which can be avoided by SmartState with the help of analyzing ASD. For this case, the baseline without ASD and TSD can only report that function *redeem* contains a DoS attack vulnerability, as it cannot identify the ASD between functions *redeem* and *transferFrom*. Nevertheless, if we analyze such state dependency, we can find that function *redeem* is a vulnerability indicator, and it can taint the function *transferFrom*. SmartState avoids such false negatives, as it extracts the ASD edge and consequently finds all the vulnerabilities alongside it.

5.4 Effectiveness of Vulnerability Indicator Analysis

As mentioned earlier in Section 4.4, another advantage processed by SmartState is the vulnerability indicator analysis in the detection, which helps for improving the precision of SRV identification. The effectiveness of vulnerability indicator analysis is reflected in the precision rate. To evaluate the effectiveness of vulnerability indicator analysis, we compare the precision rate of SmartState with state-of-the-art tools (i.e. eTainter [19] Madmax [20] and Slither [18]). As neither of the three prior tools can support identifying *Profit-gain attack* vulnerability, we evaluate the effectiveness of vulnerability indicator analysis by comparing SmartState with three prior tools for identifying *DoS attack* vulnerability. We ran all of these tools on the large-scale dataset (D_{large}) to evaluate their precision.

As shown in table 5, SmartState presents much higher precision (i.e., 84.16%) than the three prior tools. Our further investigation finds that most of the false positives introduced by prior tools can be avoided through our proposed vulnerability indicator analysis. In Figure 10, we present an example to show how SmartState avoids the false positive, which eTainter, Madmax, and Slither report. All the prior tools mistakenly report that function *refund* contains SRV, as function *refund* invoke an external call in a loop (line 7-8). Actually, if we check the access control (line 5), we can find that the function can be only accessed by externally owned accounts rather than contract accounts, which causes function *refund* to be safe. Due to the incomplete detection approaches, prior tools do not identify the access control of the function and consequently report false positive results. In contrast, SmartState can avoid such false positives because it performs multiple checks on function *refund* through the vulnerability indicator analysis and infers that such function has access control and is consequently safe.

Table 5: Precision rate for different tools on the large-scale dataset (D_{large}).

Tool	Precision		
	TP	FP	rate
eTainter	178	215	45.29%
Madmax	19	54	26.03%
Slither	151	2,555	7.08%
SmartState	574	108	84.16%

5.5 Large-scale Analysis for Finding SRVs

To answer RQ4, we evaluate the performance of SmartState by running SmartState on the large-scale dataset (D_{large}) for SRV detection. Our domain experts manually inspect these results via majority voting and confirm that SmartState successfully identifies 406 new SRVs from 47,351 real-world smart contracts. Specifically, SmartState reports 771 warnings (including 651 TPs and 120 FPs confirmed manually). 245 of 651 SRVs can be detected by SOTA tools (i.e., Madmax [20], Slither [18] and eTainter [19]). Therefore, SmartState reports 406 (651-245) new SRVs. Particularly, we rank the large-scale dataset (D_{large}) according to their number of transactions and intercept the smart contracts of the top 2000 transaction number as the "popular smart contracts". We found that 11 SRVs exist in the popular smart contracts. Further, these 11 SRV affects a total asset of 428,600 USD as of manuscript submission. Below we discuss two case studies for illustration.

Case study 1. at *0xEB834ae72B30866af20a6ce5440Fa598BfAd3a42*. The smart contract is from the GameFi *WolfGame*, which ranks 83 in the top 1000 market-value GameFi. Unfortunately, this smart contract contains a SRV. The vulnerability indicator of such SRV is the function *claimSheepFromBarn*, as it leverages a random number to determine the punishment result of free of tax behavior but without forbidding the invocation of an external contract. Hence, the function *claimSheepFromBarn* can be attacked by malicious players via invoking the specific call chain (i.e., from *claimManyFromBarnAnd-Pack* to *claimSheepFromBarn*). For this case, SmartState effectively identifies the vulnerable function and utilizes the state dependency

```
1 contract BsktToken {
2   //DoS indicator
3   function redeem (uint256 baseUnits) external {
4     for (uint256 i = 0; i < tokens.length; i++) {
5       uint256 amount = baseUnits;
6       //state variable tainted by DoS
7       require(erc20.transfer(msg.sender, amount)); }}
8   //ASD propagate
9   function transferFrom(address _from, address _to,
          uint256 _value) public {
10    require(_value <= balances[_from]);
11    balances[_from] = balances[_from].sub(_value);
12    //state variable tainted by DoS (cross-method)
13    balances[_to] = balances[_to].add(_value);}
```

Figure 9: An example of false negative reported by SmartState w/o ASD and TSD, but effectively eliminated by SmartState

```
1 contract Lotto {
2   address[] internal playerPool;
3   function refund() public payable {
4     // access control, only EOA account can access
5     require(tx.origin = msg.sender)
6     require(playerPool.length > 0);
7     for (uint i = 0; i < playerPool.length; i++) {
8       playerPool[i].transfer(100 finney);}
9     playerPool.length = 0; }
```

Figure 10: An example of false positive reported by eTainter and Madmax, which is avoided by SmartState.

between function *claimSheepFromBarn* and function *mint* to determine that state variable *balance* is manipulated by the SRV.

Case study 2. at *0x222222de1914c2b303504e3b035cf46b11fcfc6c*. As of manuscript submission, this smart contract possesses an asset of 157.41 ETH (i.e., 248,396 USD) and involves 15,465 transactions. Unfortunately, the smart contract contains the SRV. In this case, the vulnerability indicator is the function *pay_royaltie* because it invokes an external transfer in a loop but misses forbidding the invocation of an external contract. If one of the users refuses the money and makes the transaction fail through an assertion statement, the whole loop fails and locks all the rewards in the contract. For this case, SmartState effectively locates the vulnerability indicator and reports that state variable *withdrawl* and *balance* is tainted by the SRV.

5.6 Discussion and Limitation

To sum up, SmartState demonstrates its advantages for detecting SRV. (1) As shown in the evaluation, SmartState can effectively identify SRV for smart contracts, which a few prior works can support. (2) SmartState can precisely find out the root cause (i.e., indicator) of SRV and identify the state variables tainted by SRV by analyzing the state dependency of smart contracts. Hence, SmartState can locate the vulnerability more precisely, in contrast to those approaches devoted to finding out the side effect (i.e., Gas insufficiency) of vulnerability, so it is sufficiently effective and practical. All of the developers, participants, and third-party authorities can leverage SmartState to inspect the security of smart contracts.

SmartState can be further improved in the following aspects: (1) At present, SmartState can support identifying two major types of SRV, and we can further widen its detection scopes so that it can support more newly emerging types of vulnerabilities. (2) To further improve its effectiveness, SmartState can further utilize more precise and robust tools instead of SmartDagger for its program logic recovery analysis.

Below we analyze the soundness and completeness of every design employed in SmartState. Firstly, the explicit dependency construction is limited by program logic recovery (i.e., provided by SmartDagger), resulting in imprecise information and incompleteness. Secondly, the TSD extraction is limited by the diversity of transaction history data, as finite state machine cannot extract all the TSD, it may introduce unsoundness. Thirdly, SDG construction and vulnerability analysis are complete and sound, as it neither introduces false information nor misses valid information.

6 RELATED WORK

Vulnerability detection for smart contract. At present, many program analysis tools have been proposed for detecting vulnerabilities in smart contracts. Similar to traditional program analysis, these tools can be divided into static analysis tools and dynamic analysis tools. Specifically, the static analysis tools include Oyente [31], Osiris [38], Zeus [26], Sailfish [11], SmartDagger [29], EOSafe [23], Ethainter [12], Clairvoyance [40], MadMax [20], Manian[33], Securify [39] and so on. Other tools such as ContractFuzzer [25], Sereum [34], EOSFuzzer [24], Echidna [21], echidna-parade [22], sFuzz [32], TXSPECTOR [41], SMARTIAN [14], WASAI [13] and RLF [36] are based on dynamic testing or analysis. Nonetheless, all these tools are insufficiently effective in detecting SRVs as they do not consider the fine-grained state dependency for vulnerability detection. When the recent work (i.e., Sailfish [11]) focuses on detecting state-inconsistency via analyzing the state dependency, the SDG constructed by SailFish can only cover a subset of state dependency (i.e., control flow and state R&W dependency), which cannot cover the state dependency encountered by SRV detection.

State-reverting vulnerabilities. There have been limited works closely related to SRVs. EOSafe [23] and WASAI [13] focus on detecting the rollback attack (i.e., rollback which causes profit gain)) for smart contract written in WASM language. However, since both of the two frameworks mainly identify the vulnerability via analyzing the heuristic pattern without considering important state dependencies in smart contracts, they can only cover part of the SRV (i.e., profit-gain). Also, they are not sufficiently effective in detecting this sub-type of state-reverting related attacks. While the two key steps (in Section 3) used for detecting SRVs in SmartState are based on heuristics, our proposed SRV indicators are more generic and fundamental. Besides, SmartCheck [37], Slither [18], eTainter [19] and Madmax [20] tend to cover DoS vulnerabilities which are also caused by state-reverting. However, DoS is merely a sub-type of SRV, and the detection in eTainter [19] relies on specific heuristics such as detecting gas insufficiency or exhausting. Such a design is not sufficiently effective and generic to all SRVs (i.e., unauthorized state changes).

7 CONCLUSION

This paper proposes SmartState, a static analysis framework for identifying state-reverting vulnerability in smart contracts. SmartState can be mainly divided into two procedures. Firstly, SmartState extracts the state dependency from the bytecode and historical transactions of smart contracts, and further merges the state dependency as a fine-grained state dependency graph. Secondly, SmartState utilizes a unique vulnerability detection method based on taint analysis guided by our proposed indicator analysis to find out the state-reverting vulnerability on the state dependency graph. We evaluate SmartState over a manually labeled dataset of 47 smart contracts and a large-scale dataset of 47,398 real-world smart contracts. The evaluation result shows that SmartState is effective in detecting state-reverting vulnerability with a high precision of 87.23% and recall of 89.13%. Further, SRVs are prevalent in real-world smart contracts. Particularly, we find that 11 SRVs exist in frequently-used smart contracts, affecting a total asset of 428,600 USD.

REFERENCES

[1] 2018.   How the winner got Fomo3D prize — A Detailed Explanation. Web: https://medium.com/coinmonks/how-the-winner-got-fomo3d-prize-a-detailed-explanation-b30a69b7813f [Accessed: 28-January-2023].
[2] 2023. BNB Chain. Web: https://www.bnbchain.org/ [Accessed: 28-January-2023].
[3] 2023. EOSIO. Web: https://eos.io/ [Accessed: 28-January-2023].
[4] 2023. Ethereum. Web: https://www.ethereum.org/ [Accessed: 28-January-2023].
[5] 2023.   Ethereum accounts.   Web: https://ethereum.org/en/developers/docs/accounts/ [Accessed: 28-January-2023].
[6] 2023. Guide: Cybersecurity for GameFi Projects. Web: https://hacken.io/discover/guide-cybersecurity-for-gamefi-projects/#GameFi_hacks_are_increasing [Accessed: 28-January-2023].
[7] 2023.   Solidity documentation.   Web: https://docs.soliditylang.org/en/v0.8.10/ [Accessed: 28-January-2023].
[8] 2023.   The sources of statistics and corresponding incidents.   Web: https://github.com/InPlusLab/Collection-for-real-world-SRV-attacks [Accessed: 22-May-2023].
[9] 2023. TRON. Web: https://tron.network/ [Accessed: 28-January-2023].
[10] 2023. WASM documentation. Web: https://webassembly.org/getting-started/developers-guide/ [Accessed: 28-January-2023].
[11] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 161–178. https://doi.org/10.1109/SP46214.2022.9833721
[12] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: A smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 454–469. https://doi.org/10.1145/3385412.3385990
[13] Weimin Chen, Zihan Sun, Haoyu Wang, Xiapu Luo, Haipeng Cai, and Lei Wu. 2022. WASAI: uncovering vulnerabilities in Wasm smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 703–715. https://doi.org/10.1145/3533767.3534218
[14] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 227–239. https://doi.org/10.1109/ASE51524.2021.9678888
[15] ConsenSys. 2023. Mythril. Web: https://github.com/ConsenSys/mythril-classic [Accessed: 28-January-2023].

[16] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 530–541. https://doi.org/10.1145/3377811.3380364

[17] Pastore Fabrizio, Micucci Daniela, Guzman Michell, and Mariani Leonardo. 2022. TkT: Automatic Inference of Timed and Extended Pushdown Automata. In *IEEE Transactions on Software Engineering*. 617–636. https://doi.org/10.1109/TSE.2020.2998527

[18] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*. 8–15. https://doi.org/10.1109/WETSEB.2019.00008

[19] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2022. eTainter: detecting gas-related vulnerabilities in smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 728–739. https://doi.org/10.1145/3533767.3534378

[20] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27. https://doi.org/10.1145/3276486

[21] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 557–560. https://doi.org/10.1145/3395363.3404366

[22] Alex Groce and Gustavo Grieco. 2021. echidna-parade: a tool for diverse multicore smart contract fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 658–661. https://doi.org/10.1145/3460319.3469076

[23] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. 2021. EOSAFE: Security Analysis of EOSIO Smart Contracts. In *USENIX Security Symposium*. 1271–1288.

[24] Yuhe Huang, Bo Jiang, and Wing Kwong Chan. 2020. EOSFuzzer: Fuzzing eosio smart contracts for vulnerability detection. In *Proceedings of the 12th Asia-Pacific Symposium on Internetware*. 99–109. https://doi.org/10.1145/3457913.3457920

[25] Bo Jiang, Ye Liu, and WK Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *33rd IEEE/ACM International Conference on Automated Software Engineering*. 259–269. https://doi.org/10.1145/3238147.3238177

[26] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. 1–12.

[27] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. 2019. Exploiting the laws of order in smart contracts. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. 363–373. https://doi.org/10.1145/3293882.3330560

[28] Mariani Leonardo, Pezze Mauro, and Santoro Mauro. 2017. GK-Tail+ An Efficient Approach to Learn Software Models. In *IEEE Transactions on Software Engineering*. 715–738. https://doi.org/10.1109/TSE.2016.2623623

[29] Zeqin Liao, Zibin Zheng, Xiao Chen, and Yuhong Nan. 2022. SmartDagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 752–764. https://doi.org/10.1145/3533767.3534222

[30] Lu Liu, Lili Wei, Wuqi Zhang, Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2021. Characterizing transaction-reverting statements in ethereum smart contracts. In *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 630–641. https://doi.org/10.1109/ASE51524.2021.9678597

[31] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 254–269. https://doi.org/10.1145/2976749.2978309

[32] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 778–788. https://doi.org/10.1145/3377811.3380334

[33] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 653–663. https://doi.org/10.1145/3274694.3274743

[34] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2018. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934* (2018).

[35] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *USENIX Security Symposium*. 1361–1378.

[36] Jianzhong Su, Hong-Ning Dai, Lingjun Zhao, Zibin Zheng, and Xiapu Luo. 2022. Effectively Generating Vulnerable Transaction Sequences in Smart Contracts with Reinforcement Learning-Guided Fuzzing. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. 36:1–36:12. https://doi.org/10.1145/3551349.3560429

[37] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. 9–16. https://doi.org/10.1145/3194113.3194115

[38] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 664–676. https://doi.org/10.1145/3274694.3274737

[39] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 67–82. https://doi.org/10.1145/3243734.3243780

[40] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2020. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *35th IEEE/ACM International Conference on Automated Software Engineering*. 1029–1040. https://doi.org/10.1145/3324884.3416553

[41] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. 2020. TXSPECTOR: Uncovering attacks in ethereum from transactions. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*.