Detecting Smart Contract State-Inconsistency Bugs via Flow Divergence and Multiplex Symbolic Execution

YINXI LIU, Rochester Institute of Technology, USA
WEI MENG, Chinese University of Hong Kong, China
YINQIAN ZHANG, Southern University of Science and Technology, China

Ethereum smart contracts determine state transition results not only by the previous states, but also by a mutable global state consisting of storage variables. This has resulted in state-inconsistency bugs, which grant an attacker the ability to modify contract states either through recursive function calls to a contract (reentrancy), or by exploiting transaction order dependence (TOD). Current studies have determined that identifying data races on global storage variables can capture *all* state-inconsistency bugs. Nevertheless, eliminating false positives poses a significant challenge, given the extensive number of execution paths that could potentially cause a data race.

For simplicity, existing research considers a data race to be *vulnerable* as long as the variable involved could have *inconsistent values* under *different execution orders*. However, such a data race could be *benign* when the inconsistent value does not affect any critical computation or decision-making process in the program. Besides, the data race could also be *infeasible* when there is no valid state in the contract that allows the execution of both orders.

In this paper, we aim to appreciably reduce these false positives without introducing false negatives. We present DivertScan, a precise framework to detect exploitable state-inconsistency bugs in smart contracts. We first introduce the use of *flow divergence* to check where the involved variable may flow to. This allows DivertScan to precisely infer the potential effects of a data race and determine whether it can be exploited for inducing unexpected program behaviors. We also propose *multiplex symbolic execution* to examine different execution orders in one time of solving. This helps DivertScan to determine whether a common starting state could potentially exist. To address the scalability issue in symbolic execution, DivertScan utilizes an overapproximated pre-checking and a selective exploration strategy. As a result, it only needs to explore a limited state space.

DivertScan significantly outperformed state-of-the-art tools by improving the precision rate by 20.72% to 74.93% while introducing no false negatives. It also identified five exploitable real-world vulnerabilities that other tools missed. The detected vulnerabilities could potentially lead to a loss of up to $68.2M, based on trading records and rate limits.

CCS Concepts: • **Security and privacy Software security engineering**.

Additional Key Words and Phrases: Smart Contract; Program Analysis; Vulnerability Detection

ACM Reference Format:
Yinxi Liu, Wei Meng, and Yinqian Zhang. 2025. Detecting Smart Contract State-Inconsistency Bugs via Flow Divergence and Multiplex Symbolic Execution. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE002 (July 2025), 22 pages. https://doi.org/10.1145/3715712

Authors' Contact Information: Yinxi Liu, Rochester Institute of Technology, Rochester, USA, yinxi@mail.rit.edu; Wei Meng, Chinese University of Hong Kong, Hong Kong, China, wei@cse.cuhk.edu.hk; Yinqian Zhang, Southern University of Science and Technology, Shenzhen, China, yinqianz@acm.org.

ACM 2994-970X/2025/7-ARTFSE002
https://doi.org/10.1145/3715712

1 Introduction

The introduction of stateful smart contracts has brought about new challenges and vulnerabilities. The execution of stateful smart contracts is not solely dependent on input parameters, but also on some global storage variables. These variables can be altered by the simultaneous execution of multiple non-atomic operations, potentially leading to inconsistent results. Such non-atomic operations can be observed at different levels in the smart contract. For example, in complex interaction patterns like reentrancy and recursive calls, or at the transaction level where it is uncertain which transactions are chosen for a given block by the miners, or even above the transaction level, such as cross-blockchain messaging.

These vulnerabilities are known as state-inconsistency bugs [6]. A state-inconsistency bug involves a pair of operations; these start from the same initial state, yet produce varying final states under diverse schedules that can be manipulated by the attacker. This definition necessitates the presence of a data race[1], which can be readily identified by analyzing access patterns to the same variable [34]. Existing works have proposed various access patterns like Stale Read [6, 12, 31, 39, 45] or Destructive Write [6, 12].

Although analyzing access patterns guarantees the capture of all possible data races, it is non-trivial to determine if they are vulnerable, *i.e.*, include direct manifestations of state-inconsistency bugs. In the code snippet below, the global variable `totalAmount` can be updated by the `set_amount` function and used in the other functions. Simultaneously invoking `set_amount` and another function in two transactions could potentially create a data race, causing the other function to view *inconsistent values* of `totalAmount`, *i.e.*, values from before and after the execution of `set_amount`. However, only `withdraw_all` contains a state-inconsistency bug, while the `debugging` and `safe_withdraw` functions are benign.

```
function set_amount(uint amount) external {
    if (!under_withdraw){
        totalAmount = amount;
        under_withdraw = true;
    }
}
function withdraw_all(address dst) external {
    if (totalAmount > 0)
        makeTransfer(totalAmount, dst);
} // <- Vulnerable
function debugging() external {
    console.log(totalAmount);
} // <- Benign
function safe_withdraw(address dst) external {
    if (under_withdraw && totalAmount > 0){
        makeTransfer(totalAmount, dst);
        under_withdraw = false;
    }
} // <- Infeasible
```

Listing 1. A simplified example demonstrating the vulnerable or benign conditions of the same data race.

The challenges of determining whether a data race is vulnerable to being viewed as a state-inconsistency bug come from two perspectives. First, it is difficult to determine the *exploitablility* of a data race, as it necessitates precisely evaluating the affected program behavior. While the practical impact of a data race hinges on the specific system involved, a necessary condition for a data race to be vulnerable is that the manipulation of its associated variables' value causes *flow divergence*, *i.e.*, a new blockchain transaction flow takes the place of a scheduled one, resulting in irreversible financial losses. Existing works prepare ground truth by manually inspecting whether

[1]That is, when multiple threads in a process access a shared memory location, with at least one thread performing a write operation.

an attacker's vulnerable schedule diverts the control or Ether flow of another benign schedule [6, 33]. Data races that do not result in flow divergence cannot be exploitable as such, and are generally considered benign. In `withdraw_all`, the data race is considered vulnerable as it may be exploited to enable an unexpected transfer. While the system resets `totalAmount` from 100 to zero by invoking `set_amount(0)`, an attacker can swiftly move 100 by calling the `withdraw_all` function with a higher gas price. However, if the variable is solely used for debugging purposes in the `debugging` function as statistical information, the attacker would receive nothing under the same manipulation. Currently, despite flow divergence being established as a criterion for preparing ground truth, no prior research has created automated methods for in-depth analysis. This lack has led to the false labeling of `debugging` as a vulnerability even though there's no possibility of exploiting it for malicious intent.

Second, efficiently determining the *feasibility* of a data race is challenging, as it requires searching for a state that can enable concurrent execution of different schedules from a vast state space. For simplicity, existing work only verifies the feasibility of different schedules without ensuring they originate from the same initial state [6]. This leads to `safe_withdraw` being misclassified as a vulnerability, as the execution of both `(set_amount, safe_withdraw)` and `(safe_withdraw , set_amount)` is feasible, *i.e.*, under conditions where `under_withdraw` is true for the former and false for the latter. However, no state exists where the attacker has the option to switch between these different schedules. Specifically, when `under_withdraw` is true and `set_amount` is being invoked, the attacker cannot execute `(safe_withdraw, set_amount)` by setting a higher gas price for `safe_withdraw`, compared to what he can do via `withdraw_all`, as the execution of `safe_withdraw` requires `under_withdraw` to be opposite in value.

To address these challenges and effectively eliminate the relevant false positives, we introduce DivertScan, a precise framework that efficiently detects exploitable state-inconsistency bugs in smart contracts. DivertScan first eliminates benign data races by checking if they could result in *flow divergence*. It identifies dependent state transitions and constructs a State Dependency Graph (SDG), which facilitates efficient analysis of not only the associated global storage variables of a data race, but also the control and data flows it impacts in subsequent transactions. This allows DivertScan to filter cases like `debugging` in which altering a variable does not enable flow divergence. DivertScan then applies *multiplex symbolic execution* to eliminate infeasible data races. Unlike current solutions that only verify the feasibility of individual schedules, it can filter cases like `safe_withdraw` where two feasible schedules cannot possess a common starting state. It addresses the scalability issue in traditional symbolic execution approaches by selectively exploring a limited number of paths. It also prioritizes paths based on the estimated likelihood of leading to a true positive.

We demonstrate the effectiveness of DivertScan through comprehensive evaluation and testing. Our results showcase how the framework can precisely identify state inconsistency bugs in a diverse range of smart contracts. DivertScan has significantly surpassed the performance of state-of-the-art tools, enhancing the precision rate by 20.72% to 74.93% without generating any false negatives. It also managed to detect five exploitable vulnerabilities in real-world applications that were overlooked by other tools. The detected vulnerabilities could potentially result in a loss of up to $68.2 million, according to trading records and rate limits.

In summary, this paper makes the following contributions:

- It offers a refined solution for determining the exploitability and feasibility of data races through flow divergence and multiplex symbolic execution.
- It introduces DivertScan, a tool for precisely detecting state-inconsistency bugs in stateful smart contracts.

- It addresses scalability challenges by selectively exploring paths and prioritizing them based on the likelihood of true positives.
- It shows that DivertScan outperforms existing solutions through comprehensive evaluation and testing.

2 Background and Problem Statement

In this section, we introduce the background of our research (§2.1), review the limitation of existing literature (§2.2), and finally discuss our research goals and the research challenges (§2.3).

2.1 Consistency Requirement in Stateful Smart Contract

Smart contracts are programs that operate under the Byzantine Fault Tolerant distributed ledger consensus protocol. This protocol essentially ensures all parties involved in a transaction agree on the outcomes, signifying that all computations on a blockchain are consistent [42]. This points towards the necessity of consistency within the ecosystem.

Stateless contracts, due to their dependence solely on input parameters, naturally maintain the consistency of their results. In contrast, stateful smart contracts introduce global storage variables. Some of these variables, which we refer to as state variables, can significantly influence the control flow of the contract execution. In order to maintain consistency in stateful smart contracts, it is crucial to ensure that all changes to the state variables are atomic. Otherwise, non-atomic concurrent changes can lead to inconsistent states. While stateful smart contracts guarantee that method calls are executed sequentially without distractions, non-atomic behavior can still be observed at other levels in the blockchain [40]. For example, when one contract calls another contract, the callee can modify the caller's state, leading to reentrancy vulnerabilities [8]. Besides, the order of transactions in a block is not determined at the moment of execution. This can result in potential unsynchronized concurrent accesses to the state variables that would create inconsistent results [31]. Non-atomic behavior can also be observed at levels above the transaction level [32, 50].

Problem Scope. In this paper, we focus on detecting state inconsistency bugs in smart contracts exploitable by manipulating executing schedules. A smart contract can manifest data races when two or more operations attempt to access the same global data within close temporal proximity. Even though the operations are ultimately executed in a serialized order, the order may be different from the original sequence, leading to potential race conditions and inconsistent data. Some vulnerable data races can lead to unexpected changes in the contract state. Specifically, vulnerable data races occur when operations that should be atomic interfere with each other, leading to inconsistent program execution results. Although there are other factors that can contribute to inconsistent results, such as uncaught exceptions, this paper specifically focuses on the exploitation of vulnerable data races.

2.2 Existing Works and Their Limitations

In the existing body of research, researchers have proposed various means to detect data races by inspecting different patterns of simultaneous access to global data [6, 12, 31, 39, 45]. However, as discussed above, they lack the ability to accurately determine whether these data races can be exploited to cause vulnerable changes in program behavior. Such a problem occurs for different patterns.

Variable Access Pattens. Previous works considered a data race to be vulnerable as long as it involves updating a global storage variable [6, 9, 33, 45]. However, they cannot assess whether the updated global storage variable leads to unwanted program behavior. As a result, they might erroneously conclude that the data race caused by function calls f:`CALL set_amount` and f'':`CALL debugging` is vulnerable because it may create inconsistent view of `totalAmount`, even though the inconsistent view may not have any negative impact on the system. This is due to their lack of consideration of the subsequent control and data flows related to `totalAmount`, which are necessary to assess the exploitability of a data race.

Function Race Conditions. Given that data races are initiated by function calls, existing works have developed patterns to capture race conditions between two function invocations. For example, some earlier works consider reentrancy as the ability to re-enter the callee [8, 31, 37]. Recent work has proposed a more generalized pattern of function race condition by examining the feasibility of executing two function calls under different schedules [6]. However, this solution is imprecise as it does not guarantee the existence of a program state that enables both schedules. This consequently leaves room for the generation of false positives. In particular, it would erroneously deduce that a data race might occur between function calls f:`CALL set_amount` and f':`CALL safe_withdraw`, as either (f, f') or (f', f) is possible.

2.3 Research Challenges

We encounter the following challenges when trying to address the limitations of prior works and eliminate false positives.

Eliminating Benign Data Races. The first challenge is to precisely evaluate the exploitability of a data race and eliminate the benign ones. This requires us to infer the subsequent control and data flows related to a data race and conduct a refined examination. We propose a novel *flow divergence analysis* in response to this challenge, and only report data races that might cause the current or a follow-up transaction to fail to execute or to engage a divergent conditional branch.

Eliminating Infeasible Race Conditions. We need to ensure that there are real program states and that can facilitate function race conditions. Determining whether two separate function calls starting from the same state can lead to race conditions is challenging. This is because the execution process cannot be determined statically, and we can only reach a conclusion by dynamically exploring the possible state spaces. We propose *multiplex symbolic execution* to concurrently explore the execution paths created by two function calls.

Scalable Dynamic Analysis. The second challenge is to develop a scalable dynamic analysis. Even though it is theoretically possible to search for satisfying states by traversing the entire state space, examining a significant number of possible states would be infeasible in practice. Therefore, to achieve good scalability, we should limit the search space to a limited scope. We optimize the search space by incorporating a selective exploration strategy into our symbolic execution.

3 Methodology Overview

First, we introduce the key heuristics employed in our detection task (§3.1). Next, we describe the integration of these checks within a hybrid program analysis framework (§3.2).

3.1 Detection Heuristics

Our overall logic of detecting state inconsistency bugs is comprised of two phases: First, we develop patterns to detect non-atomic operations that may cause function race conditions. Second, we check whether each detected race condition potentially leads to flow divergence. We incorporate race condition checks and flow divergence analysis into the process into the diagram in Figure 1 to illustrate the process.

Given an operation pair (f, f'), we conclude whether it manifests state inconsistency bugs by running the following two checks:

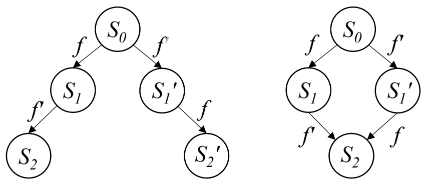
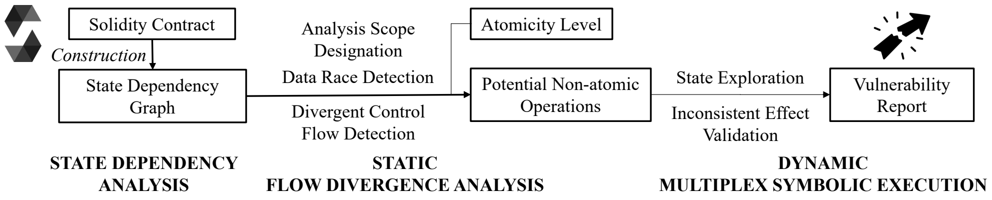


Fig. 1. The Rules for Detecting State Inconsistency Bugs.



Solidity Contract
Construction
State Dependency Graph
Analysis Scope Designation
Data Race Detection
Divergent Control Flow Detection
Atomicity Level
Potential Non-atomic Operations
State Exploration
Inconsistent Effect Validation
Vulnerability Report
STATE DEPENDENCY ANALYSIS
STATIC FLOW DIVERGENCE ANALYSIS
DYNAMIC MULTIPLEX SYMBOLIC EXECUTION

Fig. 2. An architecture overview of DivertScan.

- Whether there exists a state S_0 that enables the successful execution of both f and f' to access a shared global storage variable.
- Whether the resulting states S_2 and S_2' may reside under different conditional branches.

If the first rule evaluates to false, there is no state inconsistency bug. If the second rule evaluates to false (as shown in the left subdiagram), there exists a state inconsistency bug; otherwise (as shown in the right subdiagram), there is no state inconsistency bug.

3.2 Hybrid Program Analysis

We have discussed in §2.2 that existing approaches, regardless of the type of patterns they inspect, suffer from a high false positive rate. This is because they consider only different forms of data access patterns and do not take into account the specific program states that enable function race conditions and lead to flow divergence. The difficulty in identifying those vulnerable program states lies not only in developing corresponding methodologies for precisely describing their characteristics but also in solving the problem of state explosion while considering the possible control flows.

To address this challenge, we propose a hybrid analysis framework. First, we design a static analysis to identify non-atomic access patterns that could potentially lead to flow divergence in a state dependency graph representation. Second, we reduce the problem of inconsistent state searching to incorporate constraint solving for two paths (created by invoking two non-atomic operations) on the same set of symbolic input, and thus search for a feasible pair of paths using multiplex symbolic execution. To improve the efficiency of our multiplex symbolic execution, we leverage a selective path exploration strategy guided by the identified non-atomic access patterns. Additionally, we design a specific search strategy to determine the order in which the non-atomic pairs should be verified.

4 DivertScan

In this section, we present DivertScan, a hybrid system based on the methodologies in the last section to detect exploitable state inconsistency bugs in smart contracts. The workflow of DivertScan is shown in Figure 2. DivertScan first constructs a state dependency graph from the Solidity contract source code for further analysis (§4.1). Next, it performs static control and data flow analysis based on the graph and produces a set of operation pairs that may not be atomic

(§4.2). It then selectively executes multiplex symbolic execution to verify whether these operation pairs are vulnerable (§4.3).

4.1 State Dependency Analysis

DivertScan builds a state dependency graph (SDG) to analyze the conditions of transitions among the states and find the inconsistent states caused by divergent control flows on a non-atomic access pattern (§4.2). These inconsistent states and the non-atomic operations that cause them are used as input in our selective multiplex symbolic execution engine (§4.3).

While it's possible to build SDG from bytecode using disassembly tools, contract complexity and optimization settings during compilation can affect the accuracy and completeness of the resulting control and data flow graphs. In this study, we assume that the source code is available and disregard these complexities.

A program state in an Ethereum smart contract includes both global storage variables and stack information. While constructing each SDG node as a program state is comprehensive, where each node represents an individual statement or procedure, we do not need all this information to determine flow divergence. To preserve only the necessary information for determining control flow divergence, we construct an SDG at the basic block level. This is because the control flow divergence is naturally indicated by different basic blocks, and the instructions inside a basic block follow an uninterrupted execution sequence. We correlate each program state s to a basic block b by first checking the EVM stack used by the contract for reaching the current basic block. Based on this information, we can identify a set of basic blocks B related to the current stack frame. We then consider the program counter s_{pc} to help correlate s with a basic block $b \in B$ in which s_{pc} resides. Control flow divergence naturally indicates different basic blocks, as instructions inside a basic block follow an uninterrupted execution sequence. Even though two distinct control flows may pass through the same basic block, they must also pass a conditional statement that leads to two other distinct blocks. Thus, we can identify control flow divergence if there are two states correlated to different basic blocks under the same conditional statement.

4.1.1 SDG Construction Rules. We provide below the rules and constitute of our SDG, which is designed at a fixed level that accounts for conditional logic both within and across functions.

Nodes. Nodes in an SDG represent basic blocks, including the corresponding statements.

Edges. An edge represents either the direct control flow dependency between the connected nodes or the data flow dependency between statements in the nodes.

ICFG Facts. In contrast to building traditional function-level control flow graphs (CFGs), which only include intraprocedural direct control flow dependencies between basic blocks, constructing an SDG necessitates interprocedural information. Therefore, we combine individual function-level CFGs with a global call graph (CG) to generate an interprocedural control flow graph (ICFG).

SDG Construction. We construct the SDG by performing data flow analysis on top of the ICFG. Using the ICFG as a basis, we incorporate data flow dependencies and remove basic blocks that lack any direct control dependencies or data flow dependencies to create an SDG. To consider data flow dependencies that impact the control flow, we perform backward data flow analysis starting from each conditional statement. This analysis allows us to gather the variables that influence condition values. Next, we search for statements that assign new values to these variables. We then create data flow dependencies between the nodes containing these statements and the nodes containing the affected conditional statement.

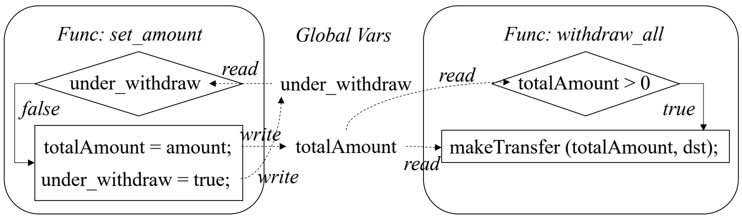


Fig. 3. A State Dependency Graph Example.

Reachability Analysis. The operations that an attacker can manipulate should start with a public function. We consider a function to be public by filtering a set of attributes that affect user-accessibility, *e.g.*, filters onlyAdmin, onlyDAO, internal, initializer, *etc.*. These attributes originated from our manual analysis of Solidity documentation and various projects, and while we do not suggest they are exhaustive, we did not observe false negatives arising from unrecognized attributes in our experiments (§5). Furthermore, an internal function may be accessible to the public through a public function in another contract. To ensure comprehensive analysis, we conduct static cross-contract invocation analysis to identify and classify these internal functions as publicly available. Specifically, when the callee contract is explicitly initialized and invoked within the source code of the caller contract, we perform backward data dependence analysis with the callee contract and establish the dependency. This information is later utilized to filter operation pairs in which both operations are not publicly accessible. We also take into account data races in the same function that is invoked by two different contracts, *e.g.*, when f_1 in Contract A and f_2 in Contract B, invoke a shared function f_0 in Contract C. The function f_0 is atomic and does not have any state inconsistency problems, making it considered safe by existing tools. However, there is a vulnerability that arises due to the undetermined order of the two invocations of f_0. In this case, function f_0 modifies a key variable v. The vulnerability occurs because the order in which f_1 and f_2 invoke f_0 is not determined. This lack of determinism enables inconsistent results of the value of v.

4.1.2 *Cross-Functional Data Flow Construction.*

While each program state encapsulates a distinctly different execution context derived from a specific contract function, we attempt to design a comprehensive static SDG that combines all functions within a contract and additionally correlates contracts that refer to each other statically. Figure 3 demonstrates how the SDG incorporates conditional logic both within individual functions and across multiple functions. This illustration also applies to scenarios involving multiple contracts. The function `set_amount` includes a conditional statement that accesses the global variable `under_withdraw`, and it modifies the global variables `totalAmount` and `under_withdraw` when `under_withdraw` is set to false. Meanwhile, the function `withdraw_all` retrieves data from `totalAmount`. The diagram allows us to ascertain that `totalAmount` acts as a conduit for data flow between `set_amount` and `withdraw_all`, specifically when `under_withdraw` is false.

4.1.3 *Optimizing Control and Data Flow Dependency.*

We employ code snippets to demonstrate scenarios where we remove basic blocks lacking direct control or data flow dependencies, thereby producing a streamlined SDG.

In the below example, the `calculateDiscount` function performs a purely computational operation to calculate the discounted price. It does not read or write to any external variables, and the function is marked as `pure`, indicating that it has no side effects on the contract's state. Therefore, we can conclude that the basic blocks within this function do not have any direct control or data flow dependencies to the contract's global state.

```
function calculateDiscount(uint price, uint discountPercentage) public pure returns (uint)
     {
    uint discount = (price * discountPercentage) / 100;
    return price - discount;
}
```

Listing 2. An example of purely computational logic.

In the below example, the `deposit` function initializes and cleans up some variables (`startTime` and `endTime`) that are not used in any other code locations of the contract. These basic blocks can be safely eliminated, as they do not have any direct control or data flow dependencies with the contract's global state.

```
function deposit(uint amount) public {
    require(amount > 0, "Amount must be greater than 0");
    // Initialization
    uint256 startTime = block.timestamp;
    uint256 endTime = startTime + 1 days;
    // Deposit
    ...
    // Cleanup
    startTime = 0;
    endTime = 0;
}
```

Listing 3. An example of redundant initialization and cleanup code.

4.2 Static Flow Divergence Analysis

The objective of our static flow divergence analysis is to identify potential non-atomic operations from the SDG at a given atomicity level. To achieve this, we follow a series of steps. First, we designate different levels of atomicity for checking non-atomic accesses to the storage variables (§4.2.1). Next, we perform non-atomic access pattern detection to identify pairs of current data accesses to the specified variables (§4.2.2). Finally, by conducting divergent control flow detection, we filter out data races that do not result in control flow divergence (§4.2.3). This helps reduce the number of false positives in our report.

4.2.1 Analysis Scope Designation. In this phase, our main objective is to concretely establish the scope and the corresponding parameters of our analysis. We designate a specific atomicity level, thereby identifying the corresponding level of non-atomic operation pairs. Our objective is to examine if all accesses by pairs of non-atomic operations to individual global storage variables are atomic.

Different levels of non-atomic operations create data races differently. Under each level, we must adopt different approaches to identify pairs of non-atomic operations. By breaking the interleaved non-atomic operations into sub-operations, we only need to consider the data races caused by the ordering of the sub-operations later.

Function Level. In terms of function level, we consider reachable reentrant pattern, *i.e.*, where one method f_0 may involve another f_1 as its callback function and later reenter the original function. For each reachable reentrant pattern, we create a non-atomic function pair from its corresponding functions, *i.e.*, (f_0, f_1). Besides, both f_0 and f_1 should be publicly accessible. Later, we further examine if these pairs can create data races.

Transaction Level. At the transaction level, each operation corresponds to a transaction that invokes a public function. While the execution of each individual transaction is atomic, the order of including different transactions into the block is undetermined. Thus, a non-atomic transaction pair can be formed by creating two transactions t_0 and t_1 that invoke two different public functions f_0 and f_1, respectively. As a result, all combinations of public function calls are within the scope.

Above Transaction Level. The framework can be extended to support above transaction level analysis when users explicitly specify the public functions $F : [f_0, f_1, \ldots, f_{n-1}]$ that should be considered as a single atomic unit. If no specific order is required, we create (f_i, f) for all $i \in (0, n-1)$. Otherwise, we denote a prefix of the ordered list F as F'_l, which are the first l elements in F, for which we create (F'_l, f) for all $l \in (1, n-1)$. We leave the comprehensive study as future work due to the enormous manual effort required to tailor specifications for each case.

4.2.2 Data Race Detection. We detect data races by first identifying pairs of operations that conform to two access patterns: stale reads and destructive writes, and filter operation pairs in which both operations are not publicly accessible. Prior research has shown that these two access patterns can guarantee no false negatives [6]. Based on this, we only filter out cases later that we are certain are either benign or infeasible. This ensures our tool filters out false positives while not introducing false negatives.

A stale read happens when one non-atomic operation tries to update a variable, but another reads the same variable before the update is completed. To identify this pattern in SDG, we check if there is a data flow to the target variable in one operation, and a data flow from the target variable in the other operation. A destructive write occurs when one non-atomic operation tries to read a variable, but another updates the same variable before the read is completed. In SDG, this leads to the same pattern where a data flow from the target variable in one operation, and a data flow to the target variable in the other operation.

In particular, referring to Listing 1, a possible stale read or destructive write may occur when a non-atomic operation that calls `set_amount` attempts to modify the variable `totalAmount`, while another operation which calls `withdraw_all`, `debugging`, or `safe_withdraw` is reading the same variable. This can be pinpointed by monitoring the read and write accesses to the global storage variable `totalAmount` on lines 3, 8, 9, 12, 15, and 16. In addition, a non-atomic operation invoking `set_amount`, along with another operation invoking `safe_withdraw`, would also be flagged as a potentially susceptible operation pair, since both consist of statements that execute read and write operations on the global storage variable `under_withdraw` at lines 2, 4, 15, and 17.

4.2.3 Divergent Control Flow Detection. We carry out divergent control flow detection to eliminate data races that do not lead to control flow divergence. We analyze the SDG data flow information to identify divergent control flow.

Specifically, two types of flow divergence exist in Ethereum smart contracts, as corroborated by the manual inspection rules utilized by previous study to establish a baseline for identifying state inconsistency bugs [6]. The first type is control flow divergence within a single transaction. Specifically, an attacker can exploit a vulnerable operation using multiple techniques, such as invoking it with a high gas price, paying a substantial priority fee, or utilizing private relay networks like Flashbots. These methods can modify the control flow of a scheduled transaction, forcing it to follow a different conditional branch. The other type is data flow divergence which indirectly results in control flow divergence in subsequent transactions. For example, the attacker may contaminate the Ether amount of the contract or the call destination of a benign operation, making a previously scheduled follow-up transaction fail to execute or engage a divergent conditional branch.

All non-atomic access patterns we discussed involve an update statement st_w and a read statement st_r to the same variable v.

Firstly, we check the update details of st_w to see whether the value of v has been made inconsistent. If st_w writes the same value into v as the old value, st_r will read a consistent result even if the non-atomic access pattern is present. Therefore, we need to determine if the old value and the new value can be different. In our static analysis, we only filter cases where st_w actually copies the old value. We will filter the other cases using symbolic execution in our later dynamic analysis.

Secondly, we examine whether the occurrences of v in st_r can potentially result in inconsistent states by flowing into a conditional statement and influencing its branch selection. If this is the case, there is a chance that the non-atomic access patterns may impact the behavior of the program. Specifically, we investigate in the SDG whether there is a flow from st_r to a conditional statement, and we continue to check for different branch selections in our dynamic analysis.

We consider the cases where a single variable flows into multiple variables that influence conditional branching. We leave race conditions created by multiple state variables as future work.

Based on these rules, we can easily sift out the data race created by `set_amount` and `debugging`. This is because the read statement in `debugging` directly outputs the inconsistent value to the console. On the contrary, the read statement within both `withdraw_all` and `safe_withdraw` employs this inconsistent value to determine the permissibility of a subsequent transfer operation. This could potentially make them susceptible to being used to enable previously impossible transfers.

4.3 Dynamic Multiplex Symbolic Execution

From the prior static control flow analysis, we show the existence of data races that may lead to flow divergence. However, these data races might not be feasible in reality due to conflicting path conditions. Additionally, they may not lead to different branch selections, and do not result in flow divergence. In our dynamic multiplex symbolic execution, we aim to prune such infeasible cases in two steps. First, we initiate a state exploration to identify a valid initial state from which the operations could potentially generate feasible data races (§4.3.1). Second, we conduct inconsistent effect validation to verify whether the resulting states, obtained by executing two non-atomic operations from the start state in a different order, are inconsistent (§4.3.2).

4.3.1 State Exploration. Given two non-atomic operations (or sub-operations) f and f', we aim to find a starting state s_0 that allows the completion of both operations. Specifically, our state exploration uncovers a state s_0 that facilitates the two operations causing the data races to reach their respective vulnerable read/write statements st_w and st_r (§4.2.3). Since s_0 represents the initial state, we assume that it operates under an empty EVM stack with the program counter set to zero. Consequently, we initialize the global storage variables in V_0 with symbolic values, which we intend to resolve through symbolic execution.

Suppose P_f^{st} contains all possible paths that can invoke f and reach the statement st. Intuitively, we can first find P_f^{st} and $P_{f'}^{st'}$, and then examine if there exist $p \in P_f^{st}$ and $p' \in P_{f'}^{st'}$ such that they can originate from the same state without conflicting constraint requirements. The challenge of this approach is that there may be too many potential execution paths in P_f^{st} and $P_{f'}^{st'}$. Exploring all these paths and considering all their combinations could result in a large search space.

Over-approximated Analysis. To improve efficiency, we first run a simpler analysis by over-approximating the path constraints for reaching st. Specifically, we collect the constraints for satisfying any path in P_f^{st} and $P_{f'}^{st'}$, respectively, and name them as C_f and $C_{f'}$. From these constraints, we select the ones that all paths share, denoted as C_f^{approx} and $C_{f'}^{approx}$. We collect all the data flows to C_f^{approx} and $C_{f'}^{approx}$, and attempt to deduce the variables in the constraints by using symbolic input variables based on the data flows. For variables that do not have a complete data flow from the input variables, we assign them a new symbolic value directly for constraint solving. We then use a constraint solver to solve the expression $C_f^{approx} \ \& \ C_{f'}^{approx}$. If the constraint solving fails, it indicates a false positive. Specifically, we can collect C_f^{approx} = `(!under_withdraw)` for `set_amount` and $C_{f'}^{approx}$ =`(under_withdraw && totalAmount > 0)` for `safe_withdraw`. Since they contain conflicting conditions `!under_withdraw` and `under_withdraw`, the resolution of their conjunction results in a false outcome. In general, the over-approximated analysis sends a significantly simplified expression to the solver, which can lead to early failure in many cases (§5.3). Consequently, we can effectively filter out many false positives.

Selective Multiplex Symbolic Execution. After passing the static checking, we conduct the selective multiplex symbolic execution to precisely verify the existence of a data race. Unlike prior multiplex symbolic execution, which generates several inputs at one time to explore various paths that diverge at the conditionals within a single program, our multiplex symbolic execution generates one input that satisfies two paths, each incorporating the same functions but executed in different orders. In general, it deduces and solves the expression $C_f \ \& \ C_{f'}$ by solving the constraints between the two paths and generating a set of values for global storage variables V_0 for the initial state s_0. However, the full symbolic equation $C_f \ \& \ C_{f'}$ cannot be derived unless all state spaces formed by all possible paths in P_f^{st} and $P_{f'}^{st'}$ are explored. Given the risk of path explosion during complete exploration, we suggest a selective symbolic execution algorithm that seeks to find a valid solution for the symbolic equation $C_f \ \& \ C_{f'}$ early in the process while exploring paths in P_f^{st} and $P_{f'}^{st'}$ progressively. Algorithm 1 illustrates the entire process. The algorithm accepts as input the initial path constraint to solve, the SDG, and a constraint solver. It outputs the result, which contains a valid solution for the original expression from the solver, or is NaN representing the solution does not exist.

We first initialize the set of constraints for individual operations C_f and $C_{f'}$ by exp (line 1), *i.e.*, $C_f = (p_0| \dots |p_m), C_{f'} = (q_0| \dots |q_n)$, in which $p_0 \dots p_m$ and $q_0 \dots q_n$ are the constraints for individual paths. In the next step, we check if C_f and $C_{f'}$ can be further divided into expressions connected by $\&$ (line 2), *e.g.*,, `(!under_withdraw)` $\&$ `under_withdraw` $\&$ `(totalAmount > 0)`. By doing this, we can quickly identify false positives if any of its prefixes are false, *i.e.*, `(!under_withdraw)` $\&$ `under_withdraw` $= false$. Specifically, if all paths in P_f^t must go through a specific code location l to reach t, we can split C_f into two subexpressions. The first subexpression contains all path constraints that can reach l, while the second subexpression contains all path constraints that start from l and end at t. If the first subexpression fails, meaning that there is no feasible path to reach l, we can promptly exit the execution loop and report a failure (line 19-21). In the following `for` loop, we progressively extend the subexpression to check until it equals the original expression (line 3, line 22-27).

Inside each loop, we check the feasibility of individual path combinations (line 4-18). We start by selecting one feasible path for each operation (line 4). This approach allows us to check one minimum constraint unit at a time, which speeds up the constraint-solving process without adding to the overall burden. We follow the following priorities in our selection: First, we prioritize paths with a prefix that has already been verified in prior iterations of the `for` loop. The symbolic execution engine can efficiently explore verified prefixes by reusing the prior symbolic execution context. We only consider an unverified prefix if all the verified ones have been exhausted. Second, we prioritize path constraints that involve the same storage variables, *e.g.*, `under_withdraw`. This indicates the presence of conflicting constraints on the same variable. Prioritizing these paths helps filter out such cases quickly. Finally, we prioritize shorter paths with fewer constraint units. This helps avoid attempting to solve complex problems when simpler ones have already failed.

After path selection, we initially determine whether results from previous iterations exist that meet the conditions of the current iteration (lines 5-7). In particular, if every path in P_f^t must pass

Algorithm 1 DivertScan's selective symbolic execution loop.

```
Input: The original expression to solve $exp$, the SDG $G$, and the constraint solver $solver$.
Output: The result $res$: global storage variables $V_0$ in the initial state.
1: $C_f, C_{f'} \leftarrow initializeBy(exp); last\_res \leftarrow \{\}$.
2: $C_f : [C_f^0, C_f^1, \dots], C_{f'} : [C_{f'}^0, C_{f'}^1, \dots] \leftarrow divide(C_f, C_{f'}, SDG)$.
3: for all $C_f^0 \in C_f, C_{f'}^0 \in C_{f'}$ do
4:     while Exist unchecked path combinations $p, q \leftarrow select(C_f^0, C_{f'}^0, SDG)$ do
5:         if $last\_res$ satisfies $p \ \& \ q$ then
6:             $res \leftarrow last\_res$. // Reuse the result from the last iteration.
7:         else
8:             Operate the SE engine to explore $p$ and $q$, respectively.
               // Try to first solve the prefix.
9:             if $solver(p)$ or $solver(q)$ fails then
10:                Remove the failed path from selection and continue.
11:            end if
12:            $res \leftarrow solver(p \ \& \ q)$.
13:        end if
14:        $last\_res \leftarrow res$.
15:        if $res$ is non-empty then
16:            break.
17:        end if
18:    end while
19:    if $res$ is empty then
20:        Report a failure.
21:    else
22:        if $C_f^0 = C_f, C_{f'}^0 = C_{f'}$ then
23:            Report a success.
24:        else
25:            Update $C_f^0 = C_f^0 \ \& \ C_f^1, C_{f'}^0 = C_{f'}^0 \ \& \ C_{f'}^1$.
26:            Remove $C_f^1, C_{f'}^1$; Reset $res$.
27:        end if
28:    end if
29: end for
```

through a specific code location l to get to t, and a valid result that reaches l has already been obtained, we then assess whether this result can be directly applied to reach t. This approach helps maintain cost-effectiveness by leveraging earlier solver results to potentially expedite success while expanding symbolic expressions.

If there is no valid previous result, we operate the symbolic execution engine to explore the selected paths only (line 8). If a path constraint fails, we remove all path constraints that include this sub-constraint from the selection (line 9-11). This helps us reduce the search space. Next, we attempt to solve the expression formed by the subexpression (line 12). If the result is non-empty, we can break the `while` loop and further attempt to solve a longer subexpression (line 15-17).

In general, we perform the following optimizations to enable selective symbolic execution:

Early success. If we find a solution that satisfies both path constraints, we can terminate the execution early and consider it a success.

Early failure. Through progressive validation of subexpressions, we can exit early with failure. If any of the subexpressions fails, there is no need to continue further.

These heuristics reduce the volume of conditions the constraint solver has to process by focusing on the placement of conditionals in paths, an aspect that is not considered by a naive solver. Our evaluation in §5.3 empirically demonstrates this improvement, revealing that we can feed the solver fewer conditions for 27 of the 36 filtered cases during the state exploration phase.

4.3.2 Inconsistent Effect Validation. The objective of our inconsistent effect validation is to verify by executing two non-atomic operations from the start state in different order whether the resulting states are inconsistent. This includes how the affected variable in the data race later flows into a conditional statement either within the execution of these two operations or in a subsequent one.

Specifically, in §4.2.3, we mentioned that all data races involve a read statement s_2 flowing into a conditional statement c. First, we will try the concrete result produced by the last phase of state exploration. We will check if different orders of these operations, starting from the state specified by the prior constraint solving result, would cause c to take different branches. This simple validation process takes a short amount of time. If it is successful, we can directly report the vulnerability using this result. If it fails, we will need to further investigate how the different values of s_2 may lead to distinct branches.

Similar to our prior analysis in §4.3.1, we first collect all the necessary path constraints for the different branches of c based on different invocation orders. Specifically, we collect $C_{(f,f',c_T)}$ and $C_{(f',f,c_F)}$, where c_T and c_F represent taking the true and false branches of c, respectively. Next, we can apply the previously over-approximated static analysis to overapproximate $C_{(f,f',c_T)}$ & $C_{(f',f,c_F)}$ and filter some cases. If the over-approximated static validation is successful, we apply selective symbolic execution to further validate $C_{(f,f',c_T)}$ & $C_{(f',f,c_F)}$. We demonstrate in §5.3 that the over-approximated analysis allows fewer conditions to be fed into the constraint solver for 13 out of the 15 filtered cases during the inconsistent effect validation phase.

5 Evaluation

We implemented our static analysis using Slither and performed symbolic execution using Mythril, with Z3 as the constraint solver for solving constraints on symbolic global storage variables. We extensively evaluate the effectiveness of DivertScan by conducting the following experiments.

First, we demonstrate that DivertScan effectively and accurately detects exploitable state inconsistency bugs in smart contracts resulting from data races (§5.2). In particular, DivertScan has a significantly lower false positive rate compared to state-of-the-art detection tools, without introducing any false negatives (§5.2.1). Besides, DivertScan produces a significantly higher recall rate compared to state-of-the-art tools in practice, due to its special design in cross-contract analysis and up-to-date support of Solidity versions (§5.2.2). We thoroughly discuss the contribution of our control-flow divergence rule to the precision of our tool through a component-wise analysis (§5.3).

5.1 Dataset and Setup

We use two datasets for evaluation. First, to demonstrate the efficiency and accuracy of DivertScan in detecting exploitable state inconsistency bugs in real-world scenarios, we create a dataset by crawling projects and their associated contracts from immunefi [25]. We will refer to this dataset as **[D1]** in subsequent discussions. Immunefi is a bug bounty platform that hosts well-audited code, which may still contain vulnerabilities that are difficult to detect. In contrast to on-chain data that may include poorly tested testnet contracts, the contracts we collected only contain hard-to-audit issues that require a new detection tool. Additionally, the scale of on-chain data is too large to analyze comprehensively, as there is no ground truth and manual verification is necessary. Previous studies like Sailfish [6] that use on-chain data have had to randomly select a subset of the data and perform manual analysis to confirm true positives. We filter out closed projects and projects that do not host their code on etherscan networks. In the end, we obtained a more extensive dataset that includes 174 actively maintained projects across 11 etherscan networks, comprising a total of 4,502 contracts.

Since the above crafted dataset does not naturally provide ground truth, analyzing the false negatives is particularly challenging without heavy manual effort. Therefore, we only evaluate the precision of the tools on it. We conduct manual analysis on the reported cases to provide better insights into the results. The analysis aims to determine if the reported cases are true positives. We follow the rules established in prior work [6]. Specifically, we check if we could manually identify a pair of operations (f, f') where the data race between f and f' can directly divert the control flow of f' or indirectly divert the control flow of another operation f''.

To further evaluate the recall rate, we use the vulnerability benchmark collected by Zhang *et al.* [52]. We will refer to this dataset as **[D2]** in subsequent discussions. This benchmark consists of 513 real-world vulnerable contract groups. Each contract group contains one or more contracts and is associated with a vulnerability that has been exploited in Ethereum history. Although the dataset does not guarantee the inclusion of all vulnerabilities in its containing contracts, it does provide a relatively large number of vulnerabilities with ground truth. This allows us to soundly evaluate the recall of our tool against state-of-the-art tools.

We perform our experiment on an x86_64 server with four 24-core 2.10GHz Intel Xeon Platinum 8160 CPUs running Debian 9. We set the timeout for each tool to 20 minutes per contract, following prior work [6]. Additionally, we set the timeout for each individual SMT solving campaign to 15 minutes. We also set an internal 20-minute timeout for DivertScan. If it passes the static checking in state exploration, it will only report failure if the later validation phases explicitly report failure. If the later validation phase is not completed within the timeout, DivertScan will report a success with the over-approximated result. This approach allows DivertScan to filter out false positives while avoiding the filtering of true positives, which may occur due to the heavy constraint solving load required to generate valid inputs.

5.2 Detecting State Inconsistency Bugs

To demonstrate the superiority of our solution in detecting exploitable state inconsistency bugs, we compare our tool, namely DivertScan, against four state-of-the-art static analysis and symbolic execution techniques. These techniques are Oyente [31], Mythril [12], Securify [45], and Sailfish [6]. Oyente and Mythril are symbolic execution-based tools. Securify is a static analysis tool that uses Datalog rules to infer potential vulnerabilities. Sailfish is a hybrid tool that combines rule-based static analysis with symbolic execution to detect data races. While there are other recent works related to vulnerability detection in smart contracts [11, 21, 41], they aim for different objectives and do not concentrate on identifying state inconsistency bugs. We set all the tools at the same atomicity levels, *i.e.*, allowing them to detect both reentry and transaction ordering attacks. For each atomicity level, we only execute each tool on every testing contract once. There is no need for repetition, as all tools are designed to provide a consistent report for each contract. As DivertScan reports pairs of vulnerable operations, we consider a vulnerability to be successfully detected if the reported pair matches the one in the ground truth. However, for the baselines, they can only report a single operation that may be affected by other operations. Therefore, we consider a vulnerability to exist if the reported operation matches any of the operations in the ground truth pair. Due to the technical challenges involved in updating all baseline tools to support the latest language features, we opted to conduct a refined comparative analysis later between DivertScan and each tool, focusing on the contracts that both successfully analyzed.

5.2.1 Precise Real-world Detection. We present the detailed results of detecting real-world state inconsistency bugs in **[D1]**.

Table 1. The overall evaluation of detecting state inconsistency bugs.

Tool	TP	Report	Precision (%)	Analyzed	Timeout
DivertScan	89	117	76.07	4,325	67
Oyente	56	168	33.33	1,362	0
Mythril	2	182	1.10	1,467	2973
Securify	78	457	17.07	3,933	532
Sailfish	31	56	55.35	1,484	18

General Comparison. Table 1 presents the general results of detecting state inconsistency bugs. Oyente, Mythril, Securify, and Sailfish report 168, 182, 457, 56 potential vulnerabilities, respectively. However, their true positive counts are only 56, 2, 78, 31, resulting in precision rates of 33.33%, 1.1%, 17.07%, and 55.35%, respectively. On the other hand, DivertScan only reports a total of 117 potential vulnerabilities but has the highest precision rate among all the tools, at 76.07%. It identifies 89 true positives, including 78 transaction ordering bugs and 11 reentry bugs. In particular, these 89 true positives encompass all the true positives that every other tool combined could identify, *i.e.*, no true positive is overlooked by DivertScan but reported by another tool.

False Positives. We analyze the false positives reported by each tool to summarize their main limitations. Oyente introduces false positives mainly due to its over approximated modeling of reentry attack by the occurrences of a reentrant call. Compared to Oyente, DivertScan filters out 93.8% (105 out of 112) of its false positives. Securify models with a more precise pattern, but it does not consider whether the data race is guarded by conditionals that cannot be satisfied. Compared to Securify, DivertScan filters out 94.2% (357 out of 379) of its false positives. Mythril incorrectly reports a lot of read-only behavior without any inconsistency. Compared to Mythril, DivertScan filters out 98.9% (178 out of 180) of its false positives. Sailfish does not consider whether a shared state exists and whether the identified access pair can lead to divergent control flow. DivertScan filters out 60% (15 out of 25) of its false positives.

Although DivertScan employs the highest precision rate, it would still have false positives due to the lack of consideration of specific contract semantics. The below example showcases a false positive that all existing tools (including DivertScan) would report. In this example, there is no conditional statement that guards the two operations that update `allowances` in the two functions, respectively. All the tools would consider this as a pattern of a data race. However, the vulnerability can only exist when `address(this) = msg.sender`. It implicitly requires the contract to call itself. To rule out this false positive, a more in-depth analysis of the semantics of contract variables in relation to contract components is required.

```
function deposit() external {
        uint _amount = balances[address(this)];
        allowances[address(this)][address(ib)] = _amount;
    }

function approve(address spender, uint amount) external returns (bool) {
        allowances[msg.sender][spender] = amount;
    }
```

Listing 4. A simplified example that can only be exploited when a contract calls itself.

False Negatives. As previously demonstrated, DivertScan is not designed to identify state inconsistency bugs that are theoretically undetectable by other tools, as existing methods already ensure the detection of all potential data races by analyzing access patterns §1. Instead, DivertScan emphasizes on reducing the false positive rate of existing tools without generating new false negatives. However, in practice, we found Oyente and Sailfish face difficulties in analyzing most contracts due to unsupported Solidity versions. Oyente successfully analyzes only 1,362 contracts, while Sailfish analyzes 1,484 contracts. Consequently, they only report a portion of the total true positives, *i.e.*, Oyente detects 52 transaction ordering bugs and four reentrancy bugs; Sailfish detects 26 transaction ordering bugs and five reentry bugs. At the same time, Security detects 73 transaction ordering bugs and five reentry bugs. Mythril only identifies two reentry bugs and fails to detect any transaction ordering bugs, mainly due to the incomplete state exploration it faces in most cases.

Scalability. Mythril exhibits scalability issues, experiencing 2,973 timeouts and being able to analyze only 1,467 out of 4,502 contracts. On average, it takes 937.24 seconds to analyze each contract. In contrast, the static analysis tool Securify demonstrates better scalability. It encounters only 532 timeouts due to failures in its Datalog-based data-flow analysis and spends an average of 183.48 seconds on each contract. Oyente and Sailfish appear to be more efficient, spending an average of 164.37 and 29.34 seconds, respectively, on analyzing each contract. DivertScan addresses scalability issues through its over-approximated analysis and selective exploration strategy. It fails to analyze only 67 cases due to timeouts in the over-approximated analysis. Additionally, 34 cases trigger an internal timeout after passing the over-approximated analysis. For the remaining cases, DivertScan spends an average of 39.25 seconds on analysis. The detailed analysis time for each phase is discussed in §5.3.

Exploitability and Impact. Although all the vulnerabilities we confirm can cause flow divergence, their exploitability and impact depend on the nature of their corresponding variables and control flow. We consider a vulnerability to be *exploitable* if it has an economic impact, under which we identify two types of corresponding violations. The first type is a vulnerability that affects the transaction result of Ether, NFT, or other assets. The second type is a vulnerability that affects metadata related to a general rate of price calculation. Using these criteria, DivertScan detects 21 exploitable vulnerabilities, including five that all the other tools failed to detect. Seven vulnerabilities belong to the first type, and 14 vulnerabilities belong to the second type. We responsibly disclosed the newly detected bugs to the relevant developers.

The impact of these exploitable vulnerabilities may be influenced by the following conditions. First, it hinges on the value of the compromised assets within the vulnerable transaction. We found historically, these vulnerabilities could amass up to $68.2M, according to trading records and rate limits. Second, it depends on whether the inconsistent effect is persistent or not. Specifically, the inconsistent state may exist for a short time window before transitioning to a consistent state. For example, the correct variable value may be stored at another location and written back after some regular executions. Four out of the 21 exploitable vulnerabilities have a non-persistent effect. Additionally, developers might purposefully create exploitability within a small fund to encourage users to compete over higher gas prices and priority fees, thereby enhancing the overall liquidity of the application. We have observed two such cases. Finally, the impact could also be determined by the degree to which the metadata is affected, or the chance of such an attack being established. In practice, developers may choose not to implement these fixes or may implement them incomprehensively, as they believe the exploit is not trivial to set up (**??**). A more in-depth and comprehensive analysis of the affecting factors will be conducted in future work.

5.2.2 Reproducing Known Attacks. Table 2 presents the results for reproducing the known state inconsistency bugs in **[D2]** and comparing the recall rate. Although the benchmark dataset only contains transaction ordering attacks without reentry attacks, we find that there is no need to demonstrate a high recall over reentry attacks, as the well-analyzed reentrancy pattern can easily cover all of them.

DivertScan successfully analyzes 490 contracts and reports 304 true positives. This result further supports the fact that DivertScan is a practical tool for detecting exploitable state inconsistency

Table 2. The overall evaluation of reproducing known state inconsistency bugs.

Tool	TP	Recall Rate (%)	Analyzed	Timeout
DivertScan	304	59.26	490	7
Oyente	0	0	77	0
Mythril	8	1.56	167	332
Securify	33	6.43	405	53
Sailfish	0	0	84	2

Table 3. Filtered cases in each step.

Step	# of Filter	Per.(%)	Time(s)
Data Race Detection	-	-	9.42
Divergent Control Flow Detection	53	23.98	3.84
State Exploration			
Over-approximated Analysis	27	16.07	20.72
Complete Symbolic Execution	9	6.38	94.23
Inconsistent Effect Validation			
Over-approximated Analysis	13	9.85	31.53
Complete Symbolic Execution	2	1.68	104.74

bugs. The false negatives of DivertScan mainly arise from the incomplete modeling of cross-contract calls. Although we mentioned in §4.1 that we conduct cross-contract invocation analysis to identify internal functions that can be publicly invoked by another contract, the current analysis is incomplete as it does not consider implicit invocations, *e.g.*, those made by low-level contract calls. A more comprehensive analysis could potentially improve its performance, which we leave as future work.

At the same time, other techniques do not perform cross-contract analysis and result in a high number of false negatives. Besides, as the dataset only contains recent attacks in the Ethereum block range 13,000,000-13,800,000, Oyente and Sailfish fail to analyze about 85% cases due to incompatible support for recent Solidity versions. As a result, Oyente analyzes only 77 cases and Sailfish analyzes 84 cases, with no true positives detected by either tool. DivertScan detects 21 and 26 true positives among their analyzed cases, respectively. Mythril is able to analyze more contracts, specifically 167 out of 513, and reports eight true positives. Its performance is generally limited by incomplete state space exploration within the specified time-out. Securify successfully analyzes 405 contracts and reports 33 true positives. However, it has false negatives mainly because it only considers Ether sending call instructions and supports stale reads without considering other access patterns.

5.3 Component-wise Analysis

The goal of our component-wise analysis is to demonstrate the contribution of each analysis phase to the precision of our approach. We present in Table 3 the number of filtered cases, along with the average analysis time for each step. This provides more detailed data for the evaluation described in Section §5.2.1.

In this table, the first column describes the steps we take in our analysis. For state exploration and inconsistent effect validation, we both employ an over-approximated checking using a constraint solver prior to running the complete selective symbolic execution. The second column describes the number of false positives filtered in the corresponding steps. The third column describes the percentage of filtering with respect to all reported cases from the last step. The final column describes the average time taken for each analysis phase.

In general, the filtering does not introduce any false negatives but only eliminates false positives. Before applying any control-flow-related filtering, our precise data race detection already contributes to achieving a 40.27% precision rate. Our static divergent control flow detection filters 23.98% of the false positives while taking only 3.84 seconds on average. DivertScan already achieves a precision of 63.12% after the over-approximated checking of state exploration, which only takes an average of 20.72 seconds. DivertScan faces the largest percentage of internal timeouts (72.85%) in the selective symbolic execution of state exploration, which helps filter nine false positives. The inconsistent effect validation also filters 13 and two false positives, respectively, with an average execution time of 31.53 and 104.74 seconds. We discover that using an over-approximated validation method allows us to effectively identify and eliminate most false positives, with only a minimal increase in processing time. However, selective symbolic execution also helps filter some cases. We also discover that conducting the over-approximated analysis of inconsistent effect validation before the selective symbolic execution of state exploration does not result in the detection of additional true positives.

6 Limitations and Future Work

While DivertScan aims to reduce false positives by flow divergence, it does not consider certain specified semantics in Solidity smart contracts, resulting in non-exploitable vulnerabilities.

For instance, certain functions may have special usage in practice, even if they seem to be publicly accessible in the source code, *e.g.*, these functions can only be used within other functions that override them. There are also specific attributes that limit the scope and usage of functions, such as `ValidNFTIndex` and `whenNoPaused`. Besides, smart contracts implicitly limit the situation where `msg.sender=self.address`, *i.e.*, when the contract calls itself, making vulnerabilities that require such a condition unlikely to occur. Functions may also implicitly be designed to be executed only once at the beginning of the entire contract and cannot be used for an attack. For example, `flag = false; initialize() {if (!flag) flag = true;}` is an initiator that cannot be accessed after the contract starts running. It is also possible that some special data races are not identified, such as when two operations write to different variable names that actually refer to the same memory location. However, we did not find such cases in our evaluation. Our current implementation also does not support the analysis of cross-contract invocations that use delegatecall to invoke a dynamically determined address.

Although it is possible to encode all these cases as specific rules in our implementation, a comprehensive analysis is needed for a reliable solution, which we leave as future work.

7 Related Work

Reentry Attacks. Oyente [31], Vandal [8], and Perez *et al.* [37] consider reentrancy as the ability to re-enter the calling function. They can only detect single-function reentrancy and would also report numerous benign reentrancy scenarios [39]. Securify [45] flags any state update after an external call as a bug, without considering if it actually causes an inconsistent state. As a result, it is prone to a high rate of false positives. SeRIF [9] detects reentrancy attacks by identifying trusted-untrusted computation scenarios and enforcing secure information flow. A set of contract-specific invariants is required for the analysis, which makes it difficult to apply to large-scale vulnerability detection.

Transaction-ordering attacks. Researchers have made efforts to identify smart contract code patterns that lead to transaction-ordering attacks. These include Transaction Order Dependency (TOD) [31], event ordering bugs [28], and state inconsistency bugs [6]. While DivertScan shares the concept of TOD with Securify [45] and Sailfish [6], Oyente [31] marks a contract as vulnerable to TOD if two traces have different Ether flows. Torres *et al.* [44], Perez and Livshits [37], and Qin *et al.* [38] conduct measurement studies on front-running attacks on the blockchain. Daian *et al.* [14] introduce the concept of Maximal Extractable Value as a metric for measuring attacker profits in manipulating transaction orders.

Atomicity Violation. Several works have discussed the general atomicity violation problem in smart contracts. Sergey and Hobor [40] examine smart contracts as shared objects in concurrent transaction invocations. Zakhary *et al.* [50] and Lys *et al.* [32] discusses atomicity violations in cross-blockchain scenarios.

Program Analysis on Smart Contract. Durieux *et al.* [15] and Ghaleb *et al.* [20] conduct empirical reviews of automated analysis tools on smart contracts. Static analysis tools are used to detect specific vulnerabilities. Madmax [22] focuses on gas-related vulnerabilities using a logic-based approach. Securify [45] and Slither [16] use datalog-based static analysis to check compliance and violation signatures. NPChecker [46] combines taint analysis and model-checking techniques to find read-write hazards. Other tools include Zeus [27], Smartcheck [43], Ethainter [7], Clairvoyance [49], SmartDagger [30]. These tools typically over-approximate program states and lack path-sensitive reasoning, resulting in a high number of false positives. Some symbolic execution tools [12, 17–19, 31, 36, 41] explore the state-space of the contract and suffer from path explosion and poor scalability, others [6, 19, 21, 29] only apply symbolic execution for validation. Dynamic analysis tool Ethracer [28] generates concrete transactions using SMT solvers and executes them in different orders to check for races between transactions. Sereum and SODA perform run-time checks within the context of a modified EVM [10, 39]. TXSPECTOR performs a post-mortem analysis of transactions [51]. ECFChecker detects if the execution of a smart contract is callback-free [23]. Other fuzz testing tools rely on manually written test oracles to detect vulnerabilities [11, 13, 24, 26, 35, 47, 48].

8 Conclusion

State inconsistency problems can severely break the correct functionality of smart contracts. In this paper, we present DivertScan, a hybrid approach to detect non-atomic operations in smart contracts that can create vulnerable data races. We model state inconsistency bugs by flow divergence, which enables DivertScan to statically identify potentially vulnerable operation pairs. DivertScan then carries out multiplex symbolic execution to explore a state from which the operations can create vulnerable data races. We conducted a comprehensive evaluation of DivertScan on benchmark datasets and real-world applications. DivertScan significantly outperforms the state-of-the-art detection tools and can identify previously unknown vulnerabilities.

References

[1] 2020. *Proceedings of the 29th USENIX Security Symposium (Security).* Virtual Event.

[2] 2020. *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE).* Melbourne, Australia.

[3] 2020. *Proceedings of the 42nd International Conference on Software Engineering (ICSE).* Seoul, Korea.

[4] 2021. *Proceedings of the 30th USENIX Security Symposium (Security).* Virtual Event.

[5] 2022. *Proceedings of the 43nd IEEE Symposium on Security and Privacy (Oakland).* San Francisco, CA, USA.

[6] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. Sailfish: Vetting smart contract state-inconsistency bugs in seconds, See [5].

[7] 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 (PLDI).* London, UK.

[8] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981* (2018).

[9] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C Myers. 2021. Compositional security for reentrant applications. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[10] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, et al. 2020. SODA: A Generic Online Detection Framework for Smart Contracts.. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

[11] 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 *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Melbourne, Australia.

[12] Consensys. [n. d.]. Mythril: a security analysis tool for EVM bytecode. https://github.com/Consensys/mythril. [accessed 10/2023].

[13] Crytic. [n. d.]. Echidna: Ethereum smart contract fuzzer. https://github.com/crytic/echidna. [accessed 10/2023].

[14] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2020. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[15] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts, See [3].

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

[17] Yu Feng, Emina Torlak, and Rastislav Bodik. 2019. Precise attack synthesis for smart contracts. *arXiv preprint arXiv:1902.06067* (2019).

[18] Yu Feng, Emina Torlak, and Rastislav Bodík. 2020. Summary-based symbolic evaluation for smart contracts, See [2].

[19] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. {ETHBMC}: A bounded model checker for smart contracts, See [1].

[20] Asem Ghaleb and Karthik Pattabiraman. 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th International Symposium on Software Testing and Analysis (ISSTA)*. Los Angeles, CA, USA.

[21] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2023. AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. (May 2023).

[22] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. (Nov. 2018).

[23] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online detection of effectively callback free objects with applications to smart contracts. (Jan. 2017).

[24] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*. London, UK.

[25] Immunefi. [n. d.]. Immunefi: Web3's leading bug bounty platform. https://immunefi.com/. [accessed 10/2023].

[26] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Montpellier, France.

[27] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts.. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

[28] 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 International Symposium on Software Testing and Analysis (ISSTA)*. Beijing, China.

[29] Johannes Krupp and Christian Rossow. 2018. {teEther}: Gnawing at ethereum to automatically exploit smart contracts. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD, USA.

[30] 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 31th International Symposium on Software Testing and Analysis (ISSTA)*. Online.

[31] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria.

[32] Léonard Lys, Arthur Micoulet, and Maria Potop-Butucaru. 2020. Atomic cross chain swaps via relays and adapters. In *Proceedings of the 3rd Workshop on Cryptocurrencies and Blockchains for Distributed Systems*. 59–64.

[33] Chenyang Ma, Wei Song, and Jeff Huang. 2023. TransRacer: Function Dependence-Guided Transaction Race Detection for Smart Contracts. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 947–959.

[34] Robert HB Netzer and Barton P Miller. 1991. Improving the accuracy of data race detection. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*. 133–144.

[35] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts, See [3].

[36] Trail of Bits. [n. d.]. Manticore: A symbolic execution tool. https://github.com/trailofbits/manticore/. [accessed 10/2023].

[37] Daniel Perez and Benjamin Livshits. 2021. Smart contract vulnerabilities: Vulnerable does not imply exploited, See [4].

[38] Kaihua Qin, Liyi Zhou, and Arthur Gervais. 2022. Quantifying blockchain extractable value: How dark is the forest?, See [5].

[39] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2019. Sereum: Protecting existing smart contracts against re-entrancy attacks. (Feb. 2019).

[40] Ilya Sergey and Aquinas Hobor. 2017. A concurrent perspective on smart contracts. In *Financial Cryptography and Data Security: FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers 21*. Springer, 478–493.

[41] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. {SmarTest}: Effectively hunting vulnerable transaction sequences in smart contracts through language {Model-Guided} symbolic execution, See [4].

[42] Joao Sousa, Alysson Bessani, and Marko Vukolic. 2018. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *2018 48th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 51–58.

[43] 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.

[44] Christof Ferreira Torres, Ramiro Camino, et al. 2021. Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the ethereum blockchain, See [4].

[45] 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 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, Canada.

[46] Shuai Wang, Chengyu Zhang, and Zhendong Su. 2019. Detecting nondeterministic payment bugs in ethereum smart contracts. (Oct. 2019).

[47] Valentin Wüstholz and Maria Christakis. 2020. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Sacramento, CA, USA.

[48] Valentin Wüstholz and Maria Christakis. 2020. Targeted greybox fuzzing with static lookahead analysis, See [3].

[49] 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, See [2].

[50] Victor Zakhary, Divyakant Agrawal, and Amr El Abbadi. 2019. Atomic commitment across blockchains. *arXiv preprint arXiv:1905.02847* (2019).

[51] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. 2020. {TXSPECTOR}: Uncovering attacks in ethereum from transactions, See [1].

[52] Wuqi Zhang, Lili Wei, Shing-Chi Cheung, Yepang Liu, Shuqing Li, Lu Liu, and Michael R Lyu. 2023. Combatting Front-Running in Smart Contracts: Attack Mining, Benchmark Construction and Vulnerability Detector Evaluation. *IEEE Transactions on Software Engineering* (2023).

Received 2024-09-13; accepted 2025-01-14