

SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds

Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna
UC Santa Barbara
{priyanka, dipanjan, yanju, yufeng, chris, vigna}@cs.ucsb.edu

Abstract—This paper presents SAILFISH, a scalable system for automatically finding state-inconsistency bugs in smart contracts. To make the analysis tractable, we introduce a hybrid approach that includes (i) a light-weight exploration phase that dramatically reduces the number of instructions to analyze, and (ii) a precise refinement phase based on symbolic evaluation guided by our novel value-summary analysis, which generates extra constraints to over-approximate the side effects of whole-program execution, thereby ensuring the precision of the symbolic evaluation. We developed a prototype of SAILFISH and evaluated its ability to detect two state-inconsistency flaws, i.e., reentrancy and transaction order dependence (TOD) in Ethereum smart contracts. Further, we present detection rules for other kinds of smart contract flaws that SAILFISH can be extended to detect.

Our experiments demonstrate the efficiency of our hybrid approach as well as the benefit of the value summary analysis. In particular, we show that SAILFISH outperforms five state-of-the-art smart contract analyzers (SECURITY, MYTHRIL, OYENTE, SEREUM and VANDAL) in terms of performance, and precision. In total, SAILFISH discovered 47 previously unknown vulnerable smart contracts out of 89,853 smart contracts from ETHERSCAN.

I. INTRODUCTION

Smart contracts are programs running on top of the Ethereum blockchain. Due to the convenience of high-level programming languages like SOLIDITY and the security guarantees from the underlying consensus protocol, smart contracts have seen widespread adoption, with over 45 million instances covering financial products, online gaming, real estate, and logistics. Consequently, a vulnerability in a contract can lead to tremendous losses, as demonstrated by recent attacks. For instance, the notorious “TheDAO” reentrancy attack led to a financial loss of about $50M in 2016. Furthermore, in recent years, several other reentrancy attacks, e.g., Uniswap, Burgerswap, Lendf.me, resulted in multimillion dollar losses. To make things worse, smart contracts are immutable—once deployed, the design of the consensus protocol makes it particularly difficult to fix bugs. Since smart contracts are not easily upgradable, auditing the contract’s source pre-deployment, and deploying a bug-free contract is even more important than in the case of traditional software.

In this paper, we present a scalable technique to detect state-inconsistency (SI) bugs—a class of vulnerabilities that enables an attacker to manipulate the global state, i.e., the storage variables of a contract, by tampering with either the order of execution of multiple transactions (transaction order dependence (TOD)), or the control-flow inside a single transaction (reentrancy). In those attacks, an attacker can tamper with the critical storage variables that transitively have an influence on money transactions through data or control dependency. Though “TheDAO” is the most well-known attack of this kind, through an offline analysis of the historical on-chain data, researchers have uncovered several instances of past attacks that leveraged state-inconsistency vulnerabilities.

While there are existing tools for detecting vulnerabilities due to state-inconsistency bugs, they either aggressively over-approximate the execution of a smart contract, and report false alarms, or they precisely enumerate concrete or symbolic traces of the entire smart contract, and hence, cannot scale to large contracts with many paths. Dynamic tools scale well, but can detect a state-inconsistency bug only when the evidence of an active attack is present. Moreover, existing tools adopt a syntax-directed pattern matching that may miss bugs due to incomplete support for potential attack patterns.

A static analyzer for state-inconsistency bugs is crucial for pre-deployment auditing of smart contracts, but designing such a tool comes with its unique set of challenges. For example, a smart contract exposes public methods as interfaces to interact with the outside world. Each of these methods are entry points to the contract code, and can potentially alter the persistent state of the contract by writing to the storage variables. An attacker can invoke any method(s), any number of times, in any arbitrary order—each invocation potentially impacting the overall contract state. Since different contracts can communicate with each other through public methods, it is even harder to detect a cross-function attack where the attacker can stitch calls to multiple public methods to launch an attack. Though SEREUM and ECFCHECKER detect cross-function attacks, they are dynamic tools that reason about one single execution. However, statically detecting state-inconsistency bugs boils down to reasoning about the entire contract control and data flows, over multiple executions. This presents significant scalability challenges, mentioned in prior work.

This paper presents SAILFISH, a highly scalable tool that is aimed at automatically identifying state-inconsistency bugs in smart contracts. To tackle the scalability issue associated with statically analyzing a contract, SAILFISH adopts a hybrid approach that combines a light-weight EXPLORE phase, followed by a REFINE phase guided by our novel value-summary analysis, which constrains the scope of storage variables. Our EXPLORE phase dramatically reduces the number of relevant
instructions to reason about, while the value-summary analysis in the **Refine** phase further improves performance while maintaining the precision of symbolic evaluation. Given a smart contract, **SAILFISH** first introduces an **Explore** phase that converts the contract into a **storage dependency graph** (SDG) \(G\). This graph summarizes the side effects of the execution of a contract on storage variables in terms of read-write dependencies. State-inconsistency vulnerabilities are modeled as graph queries over the SDG structure. A vulnerability query returns either an empty result—meaning that the contract is not vulnerable, or a potentially vulnerable subgraph \(g\) inside \(G\) that matches the query. In the second case, there are two possibilities: either the contract is indeed vulnerable, or \(g\) is a false alarm due to the over-approximation of the static analysis.

To prune potential false alarms, **SAILFISH** leverages a **Refine** phase based on symbolic evaluation. However, a conservative symbolic executor would initialize the storage variables as *unconstrained*, which would, in turn, hurt the tool’s ability to prune many infeasible paths. To address this issue, **SAILFISH** incorporates a light-weight value-summary analysis that summarizes the value constraints of the storage variables, which are used as the pre-conditions of the symbolic evaluation. Unlike classic summary-based approaches [28], [30], [22] that compute summaries path-by-path, which results in full summaries (that encode all bounded paths through a procedure), leading to scalability problems due to the exponential growth with procedure size, our value-summary analysis summarizes *all paths* through a finite (loop-free) procedure, and it produces compact (polynomially-sized) summaries. As we will show later in the evaluation, value-summary analysis not only enables **SAILFISH** to refute more false positives but also scale much better to large contracts compared to a classic summary-based symbolic evaluation strategy.

We evaluated **SAILFISH** on the entire data set from **Etherscan** [11] (89,853 contracts), and showed that our tool is efficient and effective in detecting state-inconsistency bugs. **SAILFISH** significantly outperforms all five state-of-the-art smart contract analyzers we evaluated against, in the number of reported false positives and false negatives. For example, **SAILFISH** took only 30.79 seconds to analyze a smart contract, which is 31 times faster than **Mythril** [19], and six orders of magnitude faster than **Security** [46].

In summary, this paper makes the following contributions:

- We identify and formally define state-inconsistency (Section [III]) that covers a wide range of vulnerabilities in smart contracts, including a new reentrancy attack pattern that has not been investigated in the previous literature.
- We reduce state-inconsistency detection to *hazardous access* queries over a unified, compact graph representation (called a **storage dependency graph** (SDG)), which encodes the high-level semantics of smart contracts over global states. (Section [IV])
- We propose a novel **value-summary analysis** that efficiently computes conditional constraints over storage variables, which enables **SAILFISH** to significantly reduce more false positives compared to a classic summary-based symbolic evaluation.
- We perform a systematic evaluation of **SAILFISH** on the entire data set from **Etherscan**. Not only does **SAILFISH** outperform state-of-the-art smart contract analyzers in terms of both run-time and precision, but also is able to uncover 47 zero-day vulnerabilities (out of 195 contracts that we could manually analyze) not detected by any other tool. (Section [VII])
- In the spirit of open science, we pledge to release both the tool and the experimental data to further future research.

**II. Background**

This section introduces the notion of the state of a smart contract, and provides a brief overview of the vulnerabilities leading to an inconsistent state during a contract’s execution.

**Smart contract.** Smart contracts are written in high-level languages like **Solidity**, **Vyper**, *etc.*, and are compiled down to the **EVM** (**Ethereum Virtual Machine**) bytecode. Storage variables, which hold their values across function calls and transactions, are stored in the **storage** region. The **EVM** runs on the **mining** nodes and has access to the underlying blockchain that stores the contract’s persistent state. The **EVM** is a stack-based virtual machine that executes the contract’s bytecode at the expense of **gas**, a fee that rewards the miners for the computing resources spent in executing the transactions.

**Contract state.** The **Ethereum blockchain** is a transaction-based state machine [14] with transactions transitioning the machine from one state to the other. We define the contract state \(S\) as a tuple \(S = (V, B)\), where \(V = \{V_1, V_2, V_3, ..., V_n\}\) is the set of all the state variables of a contract, and \(B\) is its balance. The state is persistent, and stored on-chain.

**State inconsistency (SI).** There are several sources of non-determinism [47] during the execution of a smart contract on the **Ethereum** network. For example, the state of the blockchain can change between the time a transaction is scheduled, and the time when it is executed. Two transactions are not guaranteed to be processed in the order in which they got scheduled. Miners can choose to prioritize transactions that incentivize them the most. Additionally, an external function call originated from the contract can transfer control to a malicious actor who can now subvert the control and data-flow by re-entering the contract in any public method in the same transaction, even before the execution of the original external call completes. In the absence of any non-determinism, the contract methods and transactions would be sequenced in-order. We refer to this as the **linear** execution, which serves as the ground truth of our notion of **correctness** in this work. Similarly, we call the out-of-order, non-deterministic sequencing as permitted by the network as **non-linear (NL)** execution. If the non-linear scheduling of transactions yields a state that diverges from the one produced by the linear execution, we consider the resulting state to be **inconsistent**. It is worth noting that our notion of inconsistency is in line with the read-write hazards observed in canonical database systems, and control-flow races in parallel programs.
Reentrancy. If a contract \( A \) calls another contract \( B \), the Ethereum protocol allows \( B \) to call back to any public/external method \( m \) of \( A \) in the same transaction before even finishing the original invocation. An attack happens when \( B \) reenters \( A \) in an inconsistent state before \( A \) gets the chance to update its internal state in the original call due to this non-linear flow of execution. Launching an attack executes operations that consume gas. Though, SOLIDITY tries to prevent such attacks by limiting the gas stipend to 2,300 when the call is made through send and transfer APIs, the call opcode puts no such restriction—thereby making the attack possible.

In Figure 1a the withdraw method transfers Ethers to a user if their account balance permits, and then updates the account accordingly. From the external call at Line 4, a malicious user (attacker) can reenter the withdraw method of the Bank contract. It makes Line 3 operate on a stale value of the account balance, which was supposed to be updated at Line 5 in the original call. Repeated calls to the Bank contract can drain it out of Ethers, because the sanity check on the account balance at Line 3 never fails. One such infamous attack, dubbed “TheDAO” [6], siphoned out over USD $50 million worth of Ether from a crowd-sourced contract in 2016.

Though the example presented above depicts a typical reentrancy attack scenario, such attacks can occur in more convoluted setting, e.g., cross-function, create-based, and delegate-based, as studied in prior work [41]. A cross-function attack spans across multiple functions. For example, a function \( f_1 \) in the victim contract \( A \) issues an untrusted external call, which transfers the control over to the attacker \( B \). In turn, \( B \) reenters \( A \), but through a different function \( f_2 \). A delegate-based attack happens when the victim contract \( A \) delegates the control to another contract \( C \), where contract \( C \) issues an untrusted external call. In case of a create-based attack, the victim contract \( A \) creates a new child contract \( C \), which issues an untrusted external call inside its constructor.

Transaction Order Dependence (TOD). Every Ethereum transaction specifies the upper-limit of the gas amount one is willing to spend on that transaction. Miners choose the ones offering the most incentive for their mining work, thereby inevitably making the transactions offering lower gas starve for an indefinite amount of time. By the time a transaction \( T_1 \) (scheduled at time \( t_1 \)) is picked up by a miner, the network and the contract states might change due to another transaction \( T_2 \) (scheduled at time \( t_2 \)) getting executed beforehand, though \( t_1 < t_2 \) (non-linear execution). This is known as Transaction Order Dependence (TOD) [5].

A. Identifying the root causes of SI vulnerabilities

By manually analyzing prior instances of reentrancy and TOD bugs—two popular SI vulnerabilities (Section III), and the warnings emitted by the existing automated analysis tools [41], [19], [46], [36], we observe that an SI vulnerability occurs when the following preconditions are met: (i) two method executions, or transactions—both referred to as threads (\( t_h \))—operate on the same storage state, and (ii) either of the two happens—(a) Stale Read (SR): The attacker thread \( t_a \) diverts the flow of execution to read a stale value from storage(\( v \)) before the victim thread \( t_v \) gets the chance to legitimately update the same in its flow of execution. The reentrancy vulnerability presented in Figure 1a is the result of a stale read. (b) Destructive Write (DW): The attacker thread \( t_a \) diverts the flow of execution to preemptively write to storage(\( v \)) before the victim thread \( t_v \) gets the chance to legitimately read the same in its flow of execution. The TOD vulnerability presented in Figure 1b is the result of a destructive write.

While the SR pattern is well-studied in the existing literature [45], [41], [55], [24], and detected by the respective tools with varied degree of accuracy, the reentrancy attack induced by the DW pattern has never been explored by the academic research community. Due to its conservative strategy of flagging any state access following an external call without considering if it creates an inconsistent state, MYTHRIL raises alarms for a super-set of DW patterns, leading to a high number of false positives. In this work, we not only identify the root causes of SI vulnerabilities, but also unify the detection of both the patterns with the notion of hazardous access (Section III).

B. Running examples

Example 1. The contract in Figure 2 is vulnerable to reentrancy due to destructive write. It allows for the splitting of funds held in the payer’s account between two payees – a and b. For a payer with id id, updateSplits records the fraction (%) of her fund to be sent to the first payer in splits[id] (Line 5). In turn, splitFunds transfers splits[id] fraction of the payer’s total fund to payee a, and the remaining to payee b. Assuming that the payer with id = 0 is the attacker, she executes the following sequence of calls in a transaction with automatically detecting them, how state-of-the-art techniques fail to tackle those challenges, and our solution.

III. Motivation

This section introduces motivating examples of state-inconsistency (SI) vulnerabilities, the challenges associated...
Fig. 2: The attacker reenters updateSplit from the external call at Line 16 and and sets \( \text{splits}[id] = 0 \). This enables the attacker to transfer all the funds again to \( b \).

- (1) calls updateSplit(0,100) to set payee a’s split to 100\% (Line 5); (2) calls splitFunds(0) to transfer her entire fund to payee a (Line 16); (3) from the fallback function, reenters updateSplit(0,0) to set payee a’s split to 0\% (Line 5); (4) returns to splitFunds where her entire fund is again transferred (Line 19) to payee b. Consequently, the attacker is able to trick the contract into double-spending the amount of Ethers held in the payer’s account.

Example 2. The contract in Figure 3 is non-vulnerable (safe). The withdrawBalance method allows the caller to withdraw funds from her account. The storage variable userBalance is updated (Line 10) after the external call (Line 9). In absence of the mutex, the contract could contain a reentrancy bug due to the delayed update. However, the mutex is set to true when the function is entered the first time. If an attacker attempts to reenter withdrawBalance from her fallback function, the check at Line 4 will foil such an attempt. Also, the transfer method adjusts the account balances of a sender and a receiver, and is not reentrant due to the same reason (mutex guard).

C. State of the vulnerability analyses

In light of the examples above, we outline the key challenges encountered by the state-of-the-art techniques, i.e., SECURIFY [46], VANDAL [24], MYTHRIL [19], OYENTE [36], and SEREUM [41] that find state-inconsistency (SI) vulnerabilities. Table II summarizes our observations.

Cross-function attack. The public methods in a smart contract act as independent entry points. Instead of reentering the same function, as in case of the traditional reentrancy attack, in a cross-function attack, the attacker can reenter the contract through any public function. Detecting cross-function vulnerabilities poses a significantly harder challenge than single-function reentrancy, because every external call can jump back to any public method—leading to an explosion in the search space due to the large number of potential call targets.

Unfortunately, most of the state of the art techniques cannot detect cross-function attacks. For example, the No Write After Call (NW) strategy of SECURIFY identifies a storage variable write (SSTORE) following a CALL operation as a potential violation. MYTHRIL adopts a similar policy, except it also warns when a state variable is read after an external call. Both VANDAL and OYENTE check if a CALL instruction at a program point can be reached by a recursive call to the enclosing function. In all four tools, reentrancy is modeled after The DAO [6] attack, and therefore scoped within a single function. Since the attack demonstrated in Example 1 spans across both the updateSplit and splitFunds methods, detecting such an attack is out of scope for these tools. Coincidentally, the last three tools raise alarms here for a wrong reason, due to the over-approximation in their detection strategies. SEREUM is a run-time bug detector that detects cross-function attacks. When a transaction returns from an external call, SEREUM write-locks all the storage variables that influenced control-flow decisions in any previous invocation of the contract during the external call. If a locked variable is re-written going forward, an attack is detected. SEREUM fails to detect the attack in Example 1 (Figure 2), because it would not set any lock due to the absence of any control-flow state variable.

Our solution: To mitigate the state-explosion issue inherent in static techniques, SAILFISH performs a taint analysis from the arguments of a public method to the CALL instructions to consider only those external calls where the destination can be controlled by an attacker. Also, we keep our analysis tractable by analyzing public functions in pairs, instead of modeling an arbitrarily long call-chain required to synthesize exploits.

Hazardous access. Most tools apply a conservative policy, and report a read/write from/to a state variable following an external call as a possible reentrancy attack. Since this pattern alone is not sufficient to lead the contract to an inconsis-

```solidity
// Step 1: Enter when mutex is false
// Step 4: Early return, since mutex is true
function withdrawBalance(uint amount) public {
    // Step 1: Enter when mutex is false
    if (mutex == false) {
        // Step 2: Mutex true prevents re-entry
        mutex = true;
        if (userBalance[msg.sender] > amount) {
            // Step 3: Attempt to reenter
            msg.sender.call.value(amount)(""");
            userBalance[msg.sender] -= amount;
        } else {
            // Step 4: Early return, since mutex is true
            mutex = false;
        }
    }
    // Step 5: Set split of a (id = 0) to 0(\%)
    splits[id] = 0;
    // Step 6: Enter external call (id = 0)
    a.transfer(userBalance[msg.sender] * (100 - splits[id]) / 100);
    // Step 7: Return, set userBalance a = balance(msg.sender)
    userBalance[msg.sender] = userBalance[msg.sender] - amount;
    mutex = false;
}
```

---

1A recent extension [16] of SEREUM adds support for unconditional reentrancy attacks by tracking data-flow dependencies. However, they only track data-flows from storage variables to the parameters of calls. As a result, even with this extension, SEREUM would fail to detect the attack in Example 1.
Similarly, offline detection tent state, they generate a large number of false positives. Example 1 (Figure 2) without the updateSplit method is not vulnerable, since splits[id] cannot be modified any more. However, MYTHRIL, OYENTE, and VANDAL flag the modified example as vulnerable, due to the conservative detection strategies they adopt, as discussed before.

Our solution: We distinguish between benign and vulnerable reentrancies, i.e., reentrancy as a feature vs. a bug. We only consider reentrancy to be vulnerable if it can be leveraged to induce a state-inconsistency (SI). Precisely, if two operations (a) operate on the same state variable, (b) are reachable from public methods, and (c) at least one is a write—we call these two operations a hazardous access pair. The notion of hazardous access unify both Stale Read (SR), and Destructive Write (DW). SAILFISH performs a light-weight static analysis to detect such hazardous accesses. Since the modified Example 1 (without the updateSplit) presented above does not contain any hazardous access pair, we do not flag it as vulnerable.

Scalability. Any SOLIDITY method marked as either public or external can be called by an external entity any number of times in any arbitrary order—which translates to an unbounded search space during static reasoning. SECURIFY relies on a Datalog-based data-flow analysis, which might fail to reach a fixed point in a reasonable amount of time, as the size of the contract grows. MYTHRIL and OYENTE are symbolic-execution-based tools that share the common problems suffered by any symbolic engine.

Our solution: In SAILFISH, the symbolic verifier validates a program path involving hazardous accesses. Unfortunately, the path could access state variables that are likely to be used elsewhere in the contract. It would be very expensive for a symbolic checker to perform a whole-contract analysis required to precisely model those state variables. We augment the verifier with a value summary that over-approximates the side-effects of the public methods on the state variables across all executions. This results in an inexpensive symbolic evaluation that conservatively prunes false positives.

Offline bug detection. Once deployed, a contract becomes immutable. Therefore, it is important to be able to detect bugs prior to the deployment. However, offline (static) approaches come with their unique challenges. Unlike an online (dynamic) tool that detects an ongoing attack in just one execution, a static tool needs to reason about all possible combinations of the contract’s public methods while analyzing SI issues. As a static approach, SAILFISH needs to tackle all these challenges.

### Table I: Comparison of smart-contract bug-finding tools.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SECURIFY</td>
<td>☐</td>
<td>☐</td>
<td>☑</td>
<td>☐</td>
</tr>
<tr>
<td>VANDAL</td>
<td>☑</td>
<td>☐</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>MYTHRIL</td>
<td>☐</td>
<td>☑</td>
<td>☑</td>
<td>☐</td>
</tr>
<tr>
<td>OYENTE</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>SERREEM</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>SAILFISH</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
</tbody>
</table>

- Full ☑ Partial ☐ No support. Cr.: Cross-function, Haz.: Hazardous access, Scl.: Scalability, Off.: Offline detection.

D. SAILFISH overview

This section provides an overview (Figure 4) of SAILFISH which consists of the EXPLORER and the REFINER modules.

**Explorer.** From a contract’s source, SAILFISH statically builds a storage dependency graph (SDG) (Section IV-B) which over-approximates the read-write accesses (Section IV-A) on the storage variables along all possible execution paths. State-inconsistency (SI) vulnerabilities are modeled as graph queries over the SDG. If the query results in an empty set, the contract is certainly non-vulnerable. Otherwise, we generate a counter-example which is subject to further validation by the REFINER.

**Example 1** Example 1 (Figure 2) contains a reentrancy bug that spans across two functions. The attacker is able to create an SI by leveraging hazardous accesses—splits[id] influences (read) the argument of the external call at Line 16 in splitFunds, and it is set (write) at Line 5 in updateSplit. The counter-example returned by the EXPLORER is 11 → 12 → 16 → 3 → 5. Similarly, in Example 2 (Figure 3), when withdrawBalance is composed with transfer to model a cross-function attack, SAILFISH detects the write at Line 10, and the read at Line 19 as hazardous. Corresponding counter-example is 4 ... 9 → 17 ... 19. In both the cases, the EXPLORER detects a potential SI, so conservatively they are flagged as possibly vulnerable. However, this is incorrect for Example 2. Thus, we require an additional step to refine the initial results.

**Refiner.** Although the counter-examples obtained from the EXPLORER span across only two public functions P₁ and P₂, the path conditions in the counter-examples may involve state variables that can be operated on by the public methods P* other than those two. For example, in case of reentrancy, the attacker can alter the contract state by invoking P* after the external call-site—which makes reentry to P₂ possible. To alleviate this issue, we perform a contract-wide value-summary analysis that computes the necessary pre-conditions to set the values of storage variables. The symbolic verifier consults the value summary when evaluating the path constraints.

**Example 2** In Example 2 (Figure 3), the REFINER would conservatively assume the mutex to be unconstrained after the external call at Line 9 in absence of a value summary – which would make the path condition feasible. However, the summary (Section IV) informs the symbolic checker that all the possible program flows require the mutex already to be false, in order to set the mutex to false again. Since the pre-condition conflicts with the program-state δ = {mutex → true} (set by Line 6), SAILFISH refutes the possibility of the presence of a reentrancy, thereby pruning the false warning.
IV. EXPLORER: LIGHTWEIGHT EXPLORATION OVER SDG

This section defines hazardous access, a necessary condition for SI bugs. In order to check for the existence of such accesses, we introduce storage dependency graph (SDG), a graph abstraction that captures the control and data flow relations between the storage variables and the critical program instructions, e.g., control-flow deciding, and state-changing operations of a smart contract. Finally, we model the detection of such accesses as queries over SDG.

A. Hazardous access

As we already explained in Section II, SI bugs stem from the non-linear execution of the contract. Our idea of hazardous access is inspired by the classical data race problem, where two different execution paths operate on the same storage variable, and at least one operation being a write—conflict with each other. In a smart contract, the execution paths correspond to two executions of either the same, or different function(s)—originated from either the same, or different transaction(s).

Definition 1 Hazardous access. A hazardous access is a tuple $(s_1, s_2, v)$ where $v$ is a storage variable, and either $s_1$, or $s_2$, or both are write operations over $v$.

SAILFISH identifies hazardous access statically by querying the contract’s SDG (defined in the next section), which is a path-condition agnostic data structure. A non-empty query result indicates the existence of a hazardous access. However, these accesses might not indeed be feasible in reality due to conflicting path conditions. The REFINER module (Section V) uses symbolic evaluation to prune such infeasible accesses.

B. Storage dependency graph (SDG)

In a smart contract, the public methods are the entry-points which can be called by an attacker. Each invocation of a public method corresponds to one single execution, whereas two such executions can potentially lead to hazardous access. In order to detect such accesses, SAILFISH builds a storage dependency graph (SDG) $\mathcal{N} = (V, E, \chi)$ that models the non-linear execution flow as if it was subverted by an attacker, and how the subverted flow impacts the global state of the contract. Specifically, the SDG encodes the following information:

Nodes. A node of an SDG represents either a storage variable, or a statement operating on a storage variable. If $V$ is the set of all storage variables of a contract, and $S$ be the statements operating on $V$, the set of nodes $V := \{V \cup S\}$.

Edges. An edge of an SDG represents either the data-flow dependency between a storage variable and a statement, or the relative ordering of statements according to the program control-flow. $\chi(E) \rightarrow \{D, I, W, O\}$ is a labeling function that maps an edge to one of the three types. A directed edge $(u, v)$ from node $u$ to node $v$ is labeled as (a) D: if $u \in V$, $v \in S$, and the statement $v$ is data-dependent on the state variable $u$ (b) W: if $u \in S, v \in V$, and the state variable $v$ is written by the statement $u$ (c) O: if $u \in S, v \in S$, and statement $u$ precedes statement $v$ in the control-flow graph.

Fig. 6: Built-in rules for ICFG related predicates.

Base ICFG facts. The base facts of our inference engine describe the instructions in the application’s inter-procedural control-flow graph (ICFG). In particular, Figure 6 shows the base rules that are derived from a classical ICFG, where $s$, $m$ and $v$ correspond to a statement, method, and variable respectively. Additionally, $\text{owner}(s)$ represents that $s$ can only be executed by contract owners, which enables SAILFISH to model SI attacks precisely. We detail the detection of such statements in Appendix IV-B.

SDG construction. The basic facts generated from the previous step can be leveraged to construct the SDG. As shown in
Fig. 7: Rules for constructing SDG.

Fig. 8: Rules for hazardous access and counter-examples.

Example 3 Consider Example 1 (Figure 2) that demonstrates an SI vulnerability due to both splitFunds and updateSplit methods operating on a state variable splits[id]. Figure 5 models this attack semantics. deposits and splits[id] correspond to the variable nodes in the graph. Line 12 writes to deposits; thus establishing a \( \triangleright \) relation from the instruction to the variable node. Line 16 and Line 19 are data-dependent on both the state variables. Hence, we connect the related nodes with \( \triangleright \) edges. Finally, the instruction nodes are linked together with directed \( \triangleright \) edges following the control-flow. To model the reentrancy attack, we created an edge from the external call node 2 → 4, the entry point of splitFunds. Next, we remove the edge between the external call 2, and its successor 4. Lastly, we add an edge between 6, the exit node of updateSplit, and 3, the following instruction in updateSplit.

Definition 2 Reentrancy. A malicious reentrancy query (Figure 3) can be expressed in terms of hazardous access. In particular, the query is looking for a hazardous access pair \( (s_1, s_2) \) such that both \( s_1 \) and \( s_2 \) are reachable from an external call in the SDG, and none of them are owner only statements.

To detect delegate-based reentrancy attacks, where the delegatecall destination is tainted, we treat delegatecall in the same way as the extcall in Figure 3. For untainted delegatecall destination, if the source code of the delegated contract is available (Appendix IV-A), SAILFISH constructs an SDG that combines both the contracts. If neither the source, nor the address of the delegated contract is available, SAILFISH ensures soundness by treating a delegatecall as an unsafe external call. For create-based attacks, since the source code of the child contract is a part of the parent contract, SAILFISH builds the SDG by combining both the creator (parent) and the created (child) contracts. Subsequently, SAILFISH leverages the existing queries in Figure 2 on the combined SDG.

Example 4 When run on the SDG in Figure 3 (Example 1), the query returns the tuple (3, 5), because they both operate on the state variable splits, and belong to distinct public methods, viz., splitFunds and updateSplit respectively.

Definition 3 Transaction Order Dependency (TOD). As explained in Section 11, TOD happens when Ether transfer is affected by re-ordering transactions. A hazardous pair \( (s_1, s_2) \) forms a TOD if the following conditions hold: 1) an external call is reachable from either \( s_1 \) or \( s_2 \), and 2) the amount of Ether sent by the external call is greater than zero.

SAILFISH supports all three TOD patterns supported by SECUREIFY 40—(i) TOD Transfer specifies that the pre-condition of an Ether transfer, e.g., a condition \( c \) guarding the transfer, is influenced by transaction ordering, (ii) TOD Amount indicates that the amount \( a \) of Ether transfer is dependent on transaction ordering, and (iii) TOD Receiver defines that the external call destination \( e \) is influenced by the transaction ordering. To detect these attacks, SAILFISH reasons if \( c \), or \( a \), or \( e \) is data-flow dependent on some storage, and the statements corresponding to those three are involved in forming a hazardous pair.

Counter-example generation. If a query over the SDG returns \( \perp \) (empty), then the contract is safe because the SDG models the state inconsistency in the contract. On the other hand, if the query returns a list of pairs \( \langle s_1, s_2 \rangle \), SAILFISH performs a refinement step to determine if those pairs are indeed feasible. Since the original output pairs (i.e., \( \langle s_1, s_2 \rangle \)) can not be directly consumed by the symbolic execution engine, SAILFISH leverages the cex-rule in Figure 8 to compute the minimum
inter-procedural control-flow graph $G$ that contains statements $s_1, s_2$, and the relevant external call $s'$. In the base case, the $\text{cut}$-rule includes edges between entry points and their successors that can transitively reach $s_1, s_2$, or $s'$. In the inductive case, for every node $s_1$ that is already in the graph, we recursively include its successors that can also reach $s_1, s_2$, or $s'$.

Example 5 SAILFISH extracts the graph slice starting from the root (not shown in Figure 5) of the SDG to node $s$. The slicing algorithms extracts the following sub-graph (root) $\mapsto 2 \mapsto 4 \mapsto 5 \mapsto 3$, maps all the SDG nodes to the corresponding nodes in the ICFG, and computes the final path slice which the REFINER runs on.

Detecting other classes of attacks. Though SAILFISH is geared towards detecting SI attacks, it can be extended to detect a wide variety of security flaws that require reasoning about the contract’s state. In particular, our SDG captures both state-changing operations, e.g., self-destruct, external call, child contract creation, storage write, etc., and control-flow deciding operations in a contract. Therefore, by querying over the SDG, SAILFISH can reason about the reachability of such an instruction in any execution of the contract. For example, in case of suicidal attacks $i$, a self-destruct operation in the contract lacks appropriate guard condition; thus enabling an attacker destroy the contract, e.g., the Parity attacks $i$. $i$. We discuss the generality of our approach by presenting the detection rules for other attack patterns in Appendix I.

V. REFINER: SYMBOLIC EVALUATION WITH VALUE SUMMARY

As explained in Section IV if the EXPLORER module reports an alarm, then there are two possibilities: either the contract is indeed vulnerable, or the current counter-example (i.e., subgraph generated by the rules in Figure 5) is infeasible. Thus, SAILFISH proceeds to refine the subgraph by leveraging symbolic evaluation (Section V-B). However, as we show later in the evaluation, a naive symbolic evaluation whose storage variables are completely unconstrained will raise several false positives. To address this challenge, the REFINER module in SAILFISH leverages a light-weight value summary analysis (Section V-A) that output the potential symbolic values of each storage variable under different constraints, which will be used as the pre-condition of the symbolic evaluation (Section V-B).

A. Value summary analysis (VSA)

The goal of the value summary analysis (VSA) is to compute the interval for each storage variable $v$. For instance, if $v \in [1, 10]$, then we can infer that $v$ is an integer between 1 and 10. In other words, the value summary of a storage variable can also be viewed as a contract invariant that holds through the life-cycle of a smart contract. In reality, there are many different ways for computing value summaries and the key challenge is to achieve a good trade-off between precision and scalability. For the example in Figure 3, a naive and scalable VSA will ignore the control flows and conclude that the summary of $\text{mutex}$ is $\top$ (either $\text{true}$ or $\text{false}$), which will be useless for the following symbolic evaluation since $\text{mutex}$ is unconstrained. On the other hand, we could also obtain very precise summaries by symbolically executing the whole contract and generating all possible values and their corresponding path conditions. This approach will be computationally intensive. In the end, we design a light-weight VSA shown in Figure 5. The key intuition is to first start with a precise abstract domain that captures concrete values and their corresponding path conditions, and then gradually sacrifice the precision in the context of statements that are difficult or expensive to reason about, such as loops, return values of external calls, updates over nested data structures, etc.

To formalize our rules for VSA, we introduce a simplified language in Figure 9. In particular, a contract $P$ consists of (a) a list of public functions $\vec{F}$ (private functions are inline), (b) a value environment $\delta$ that maps variables or program identifiers to concrete or symbolic values, and (c) a path environment $\pi$ that maps a location $\text{loc}$ to its path constraint $C$. It is a boolean value encoding the branch decisions taken to reach the current state. Moreover, each function $\mathcal{F}$ consists of arguments, return values, and a list of statements containing loops, branches, and sequential statements, etc. Our expressions $e$ include common features in SOLIDITY such as storage access, struct initialization, function invocations, and arithmetic expressions, etc. Finally, we introduce a $\text{havoc}$ operator to make those variables in hard-to-analyze statements unconstrained, e.g., $\text{havoc}(s)$ changes each variable in $s$ to $\top$ (completely unconstrained).

Figure 10 shows a representative subset of the inference rules for computing the summary. A program state consists of the value environment $\delta$ and the path condition $\pi$. A rule $\langle e, \delta, \pi \rangle \mapsto \langle v, \delta', \pi' \rangle$ says that a successful execution of $e$ in the program state $\langle \delta, \pi \rangle$ results in value $v$ and the state $\langle \delta', \pi' \rangle$.

Bootstrapping. The value summary procedure starts with the “contract” rule that sequentially generates the value summary for each public function $\mathcal{F}_i$ (all non-public methods are inline). The output value environment $\delta'$ contains the value summary for all storage variables. More precisely, for each storage variable $s$, $\delta'$ maps it to a set of pairs $\langle v, \pi \rangle$ where $v$ is the value of $s$ under the constraint $\pi$. Similarly, to generate the value summary for each function $\mathcal{F}_i$, SAILFISH applies the “Func” rule to visit every statement $s_i$ inside method $\mathcal{F}_i$.

Expression. There are several rules to compute the rules for different expressions $e$. In particular, if $e$ is a constant $c$, the value summary for $e$ is $c$ itself. If $e$ is an argument of a public function $\mathcal{F}_i$, whose values are completely under the control of an attacker, the “Argument” rule will havoc $e$ and assume that its value can be any value of a particular type.
In particular, the rule first computes the symbolic expression \( v_0 \) for the branch condition \( e_0 \). If \( e_0 \) is evaluated to \( \text{true} \), then the rule continues with the \( \text{then} \) branch \( e_1 \) and computes its value summary \( v_1 \). Otherwise, the rule goes with the \( \text{else} \) branch \( e_2 \) and obtains its value summary \( v_2 \). Finally, if the branch condition \( e_0 \) is a symbolic variable whose concrete value cannot be determined, then our value summary will include both \( v_1 \) and \( v_2 \) together with their path conditions. Note that in all cases, the path environment \( \pi' \) needs to be updated by conjoining the original \( \pi \) with the corresponding path conditions that are taken by different branches.

**Example 6** Recall in Fig 3 the EXPLORER reports a false alarm due to the over-approximation of the SDG. By applying the rules in Fig 10, we can generate the following value summary for variable \( \text{mutex} \): \( \{ \text{false}, \text{mutex} = \text{false} \}, \{ \text{true}, \text{mutex} = \text{false} \} \). In other words, after invoking any sequence of public functions, \( \text{mutex} \) can be updated to \( \text{true} \) or \( \text{false} \) if pre-condition \( \text{mutex} = \text{false} \) holds.

### B. Symbolic evaluation

Based on the rules in Figure 8, if the contract contains a pair of statements \( (s_1, s_2) \) that match our state-inconsistency query (e.g., reentrancy), the EXPLORER module (Section IV) returns a subgraph \( G \) (of the original ICFG) that contains statement \( s_1 \) and \( s_2 \). In that sense, checking whether the contract indeed contains the state-inconsistency bug boils down to a standard reachability problem in \( G \): does there exist a valid path \( \pi \) that satisfies the following conditions: 1) \( \pi \) starts from an entry point \( v_0 \) of a public method, and 2) following \( \pi \) will visit \( s_1 \) and \( s_2 \), sequentially. Due to the over-approximated nature of our SDG that ignores all path conditions, a valid path in SDG does not always map to a feasible execution path in the original ICFG. As a result, we have to symbolically evaluate \( G \) and confirm whether \( \pi \) is indeed feasible.

A naive symbolic evaluation strategy is to evaluate \( G \) by precisely following its control flows while assuming that all storage variables are completely unconstrained (T). With this assumption, as our ablation study shows (Figure 11), SAILFISH fails to refute a significant amount of false alarms. So, the key question that we need to address is: How can we symbolically check the reachability of \( G \) while soundly constraining the range of storage variables without losing too much precision? This is where VSA comes into play. Our symbolic evaluation takes the output of value summary into account. The implementation details are discussed in Appendix IV-C.

**Example 7** Let us revisit the example in Fig 3 and illustrate how to leverage symbolic evaluation to refute the false alarm.

---

\[
\mu(b, v_1, v_2) = \begin{cases} 
  v_1 & \text{if } b = \text{true} \\
  v_2 & \text{if } b = \text{false} \\
  \{ (b, v_1), (\neg b, v_2) \} & \text{Otherwise}
\end{cases}
\]
Step 1: By applying the value-summary analysis (VSA) discussed in Section V.A to the transfer function in Figure 3, \textsc{Sailfish} generates the summary for storage variable mutex: mutex = {false, mutex = false}, (true, mutex = false) Here, we omit the summary of other storage variables (e.g., userBalance) for simplicity.

Step 2: Now, by applying the symbolic checker in Fig 1 on the withdrawBalance function for the first time, \textsc{Sailfish} generates the following path condition $\pi$: $\text{mutex} == \text{false} \land \text{userBalance[msg.sender]} > \text{amount}$ as well as the following program state $\delta$ before invoking the external call at Line 9: $\delta = \{\text{mutex} \rightarrow \text{true}, ...\}$

Step 3: After step 2, the current program state $\delta$ indicates that the value of mutex is true. Based on the value summary output in Step 1, we know that even if the attacker invokes zero or multiple public function calls before re-entering the withdrawBalance method, the value of mutex will still be true. In that case, although the attack can enter the withdrawBalance method again by invoking another transaction through the callback mechanism in \textsc{Solidity}, it is impossible for the attacker to “re-enter” the then-branch at Line 6 which can trigger the external call at Line 9 again. Thus, \textsc{Sailfish} discards the reentrancy report as false positive.

VI. IMPLEMENTATION

\textbf{Explorer}. It is a light-weight static analysis that lifts the smart contract to an SDG. The analysis is built on top of the \textsc{Slither} [26] framework that lifts \textsc{Solidity} source code to its intermediate representation called \textsc{Slithir}.

\textbf{Refiner}. \textsc{Sailfish} leverages \textsc{Rosette} [45] to symbolically check the feasibility of the counter-examples. \textsc{Rosette} provides support for symbolic evaluation. \textsc{Rosette} programs use assertions and symbolic values to formulate queries about program behavior, which are then solved with off-the-shelf SMT solvers. \textsc{Sailfish} uses (solve expr) query that searches for a binding of symbolic variables to concrete values that satisfies the assertions encountered during the symbolic evaluation of the program expression expr.

VII. EVALUATION

In this section, we describe a series of experiments that are designed to answer the following research questions: RQ1. How effective is \textsc{Sailfish} compared to the existing smart contracts analyzers with respect to vulnerability detection? RQ2. How scalable is \textsc{Sailfish} compared to the existing smart contracts analyzers? RQ3. How effective is the \textsc{Refine} phase in pruning false alarms? RQ4. How effective is the value-summary analysis in \textsc{Sailfish}?

A. Experimental setup

\textbf{Dataset}. We have crawled the source code of all 91,921 contracts from Etherscan [11], which cover a period until October 31, 2020. We excluded 2,068 contracts that either require very old versions (<0.3.x) of the \textsc{Solidity} compiler, or were developed using the \textsc{Vyper} framework. As a result, our evaluation dataset consists of 89,853 \textsc{Solidity} smart contracts. Further, to gain a better understanding of how each tool scales as the size of the contract increases, we have divided the entire dataset, which we refer to as full dataset, into three mutually-exclusive sub-datasets based on the number of lines of source code—small (0, 500), medium (500, 1000), and large (1000, $\infty$) datasets consisting of 73,433, 11,730, and 4,690 contracts, respectively. We report performance metrics individually for all three datasets.

\textbf{Analysis setup}. We ran our analysis on a Celery v4.4.4 [17] cluster consisting of six identical machines running Ubuntu 18.04.3 Server, each equipped with Intel(R) Xeon(R) CPU E5-2690 v2@3.00 GHz processor (40 core) and 256 GB memory.

\textbf{Analysis of real-world contracts}. We evaluated \textsc{Sailfish} against four other static analysis tools, viz., \textsc{Securify} [46], \textsc{Vandal} [24], \textsc{Mythril} [19], \textsc{Oyente} [30], and one dynamic analysis tool, viz., \textsc{Sereum} [41]—capable of finding either reentrancy, or TOD, or both. Given the influx of smart contract related research in recent years, we have carefully chosen a representative subset of the available tools that employ a broad range of minimally overlapping techniques for bug detection. \textsc{Smartcheck} [44] and \textsc{Slither} [26] were omitted because their reentrancy detection patterns are identical to \textsc{Securify}‘s NW (No Write After Ext. Call) signature.

We ran all the static analysis tools, including \textsc{Sailfish}, on the full dataset under the analysis configuration detailed earlier. If a tool supports both reentrancy and TOD bug types, it was configured to detect both. We summarize the results of the analyses in Table 11. For each of the analysis tools and analyzed contracts, we record one of the four possible outcomes—(a) safe: no vulnerability was detected (b) unsafe: a potential state-inconsistency bug was detected (c) timeout: the analysis failed to converge within the time budget (20 minutes) (d) error: the analysis aborted due to infrastructure issues, e.g., unsupported \textsc{Solidity} version, or a framework bug, etc. For example, the latest \textsc{Solidity} version at the time of writing is 0.8.3, while \textsc{Oyente} supports only up to version 0.4.19.

B. Vulnerability detection

In this section, we report the fraction (%) of safe, unsafe (warnings), and timed-out contracts reported by each tool with respect to the total number of contracts successfully analyzed by that tool, excluding the “error” cases.

\textbf{Comparison against other tools}. \textsc{Securify}, \textsc{Mythril}, \textsc{Oyente}, \textsc{Vandal}, and \textsc{Sailfish} report potential reentrancy in 7.10%, 4.18%, 0.99%, 52.27%, and 2.40% of the contracts. Though all five static analysis tools detect reentrancy bugs, TOD detection is supported by only three tools, i.e., \textsc{Securify}, \textsc{Oyente}, and \textsc{Sailfish} which raise potential TOD warnings in 21.37%, 12.77%, and 8.74% of the contracts.

\textsc{Mythril}, being a symbolic execution based tool, demonstrates obvious scalability issues: It timed out for 66.84% of the contracts. Though \textsc{Oyente} is based on symbolic execution as well, it is difficult to properly assess its scalability. The reason is that \textsc{Oyente} failed to analyze most of the contracts in our dataset due to unsupported \textsc{Solidity} version, which explains the low rate of warnings that \textsc{Oyente}
<table>
<thead>
<tr>
<th>Bug</th>
<th>Tool</th>
<th>Safe</th>
<th>Unsafe</th>
<th>Timeout</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reentrancy</td>
<td>SECURIFY</td>
<td>72.149</td>
<td>6.321</td>
<td>10.581</td>
<td>802</td>
</tr>
<tr>
<td></td>
<td>VANDAL</td>
<td>40.607</td>
<td>45.971</td>
<td>1.373</td>
<td>1.902</td>
</tr>
<tr>
<td></td>
<td>MYTHRIL</td>
<td>25.705</td>
<td>3.708</td>
<td>59.296</td>
<td>1.144</td>
</tr>
<tr>
<td></td>
<td>OYENTE</td>
<td>26.924</td>
<td>269</td>
<td>0</td>
<td>62.660</td>
</tr>
<tr>
<td></td>
<td>SYLPHIS</td>
<td>83.171</td>
<td>2.076</td>
<td>1.211</td>
<td>3.395</td>
</tr>
<tr>
<td>TOD</td>
<td>SECURIFY</td>
<td>59.439</td>
<td>19.031</td>
<td>10.581</td>
<td>802</td>
</tr>
<tr>
<td></td>
<td>OYENTE</td>
<td>23.721</td>
<td>3.472</td>
<td>0</td>
<td>62.660</td>
</tr>
<tr>
<td></td>
<td>SAILFISH</td>
<td>77.692</td>
<td>7.555</td>
<td>1.211</td>
<td>3.395</td>
</tr>
</tbody>
</table>

**TABLE II: Comparison of bug finding abilities of tools**

Unlike symbolic execution, static analysis seems to scale well. **SECURIFY** timed-out for only 11.88% of the contracts, which is significantly lower than that of **MYTHRIL**. When we investigated the reason for **SECURIFY** timing out, it appeared that the Datalog-based data-flow analysis (that **SECURIFY** relies on) fails to reach to a fixed-point for larger contracts. **VANDAL**’s static analysis is inexpensive and shows good scalability, but suffers from poor precision. In fact, **VANDAL** flags as many as 52.27% of all contracts as vulnerable to reentrancy—which makes **VANDAL** reports hard to triage due to the overwhelming amount of warnings. **VANDAL** timed out for the least (1.56%) number of contracts. Interestingly, **SECURIFY** generates fewer reentrancy warnings than **MYTHRIL**. This can be attributed to the fact that the **NW** policy of **SECURIFY** considers a write after an external call as vulnerable, while **MYTHRIL** conservatively warns about both read and write. However, **SAILFISH** strikes a balance between both scalability and precision as it timed-out only for 1.40% of the contracts, and generates the fewest alarms.

**Ground truth determination.** In order to be able to provide better insights into the results, we performed manual analysis on a randomly sampled subset of 750 contracts ranging up to 3,000 lines of code, out of a total of 6,581 contracts successfully analyzed by all five static analysis tools, without any timeout or error.\(^3\) We prepared the ground truth by manually inspecting the contracts for reentrancy and TOD bugs using the following criteria: (a) **Reentrancy:** The untrusted external call allows the attacker to re-enter the contract, thus making it possible to operate on an inconsistent internal state. Our notion of reentrancy is stricter than some of the previous work,\(^4\) which only considers the possibility of being able to re-enter the calling contract. The latter definition encompasses legitimate (benign) reentrancy scenarios,\(^5\) e.g., ones that arise due to withdrawal pattern in **SOLIDITY**. (b) **TOD:** A front-running transaction can divert the control-flow, or alter the Ether-flow, e.g., Ether amount, call destination, etc., of a previously scheduled transaction.

In the end, manual analysis identified 26 and 110 contracts with reentrancy and TOD vulnerabilities, respectively. We then ran each tool on this dataset, and report the number of correct (TP), incorrect (FP), and missed (FN) detection by each tool in Table II-D. For both reentrancy and TOD, **SAILFISH** detected all the vulnerabilities (TP) with zero missed detection (FN), while

<table>
<thead>
<tr>
<th>Tool</th>
<th>Reentrancy</th>
<th>TOD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TP</td>
<td>FP</td>
</tr>
<tr>
<td>SECURIFY</td>
<td>9</td>
<td>163</td>
</tr>
<tr>
<td>VANDAL</td>
<td>26</td>
<td>626</td>
</tr>
<tr>
<td>MYTHRIL</td>
<td>7</td>
<td>334</td>
</tr>
<tr>
<td>OYENTE</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>SAILFISH</td>
<td>26</td>
<td>11</td>
</tr>
</tbody>
</table>

**TABLE III: Manual determination of the ground truth**

maintaining the lowest false positive (FP) rate. We discuss about the FPs and FNs of the tools in the subsequent sections.

**False positive analysis.** While reasoning about the false positives generated by different tools for the reentrancy bug, we observe that both **VANDAL** and **OYENTE** consider every external call to be re-entrant if it can be reached in a recursive call to the calling contract. However, a reentrant call is benign unless it operates on an inconsistent state of the contract. **SECURIFY** considers **SOLIDITY** send and transfer APIs as external calls, and raises violation alerts. Since the gas limit (2,300) for these APIs is inadequate to mount a reentrancy attack, we refrain from modeling these APIs in our analysis. Additionally, **SECURIFY** failed to identify whether a function containing the external call is access-protected, e.g., it contains the msg.sender == owner check, which prohibits anyone else but only the contract owner from entering the function. For both the cases above, though the **EXPLORER** detected such functions as potentially unsafe, the benefit of symbolic evaluation became evident as the **REFINER** eliminated these alerts in the subsequent phase. **MYTHRIL** detects a state variable read after an external call as malicious reentrancy. However, if that particular variable is not written in any other function, that deems the read safe. Since **SAILFISH** looks for hazardous access as a pre-requisite of reentrancy, it does not raise a warning there. However, **SAILFISH** incurs false positives due to imprecise static taint analysis. A real-world case study of such a false positive is presented in Appendix II-D.

To detect TOD attacks, **SECURIFY** checks for writes to a storage variable that influences an Ether-sending external call. We observed that several contracts flagged by **SECURIFY** have storage writes inside the contract’s constructor. Hence, such writes can only happen once during contract creation. Moreover, several contracts flagged by **SECURIFY** have both storage variable writes, and the Ether sending external call inside methods which are guarded by predicates like require(msg.sender == owner)—limiting access to these methods only to the contract owner. Therefore, these methods cannot be leveraged to launch a TOD attack. **SAILFISH** prunes the former case during the **EXPLORE** phase itself. For the latter, **SAILFISH** leverages the **REFINE** phase, where it finds no difference in the satisfiability of two different symbolic evaluation traces. In Appendix II-D, we present a real-world case where both **SECURIFY** and **SAILFISH** incur a false positive due to insufficient reasoning of contract semantics.

**False negative analysis.** **SECURIFY** missed valid reentrancy bugs because it considers only Ether sending call instructions. In reality, any call can be leveraged to trigger reentrancy by transferring control to the attacker if its destination is tainted. To consider this scenario, **SAILFISH** carries out a taint analysis.

---

\(^3\) We believe that the size of the dataset is in line with the previous work.\(^4\)\(^5\).
to determine external calls with tainted destinations. Additionally, Securify missed reentrancy bugs due to lack of support for destructive write (DW), and delegate-based patterns. False negatives incurred by Mythril are due to its incomplete state space exploration within specified timeout. Our manual analysis did not observe any missed detection by Sailfish.

Finding zero-day bugs using Sailfish. In order to demonstrate that Sailfish is capable of finding zero-day vulnerabilities, we first identified the contracts flagged only by Sailfish, but no other tool. Out of total 401 reentrancy-only and 721 TOD-only contracts, we manually selected 88 and 107 contracts, respectively. We limited our selection effort only to contracts which contain at most 500 lines of code, and are relatively easier to reason about in a reasonable time budget. Our manual analysis confirms 47 contracts are exploitable (not just vulnerable)—meaning that they can be leveraged by an attacker to accomplish a malicious goal, e.g., draining Ether, or corrupting application-specific metadata, thereby driving the contract to an unintended state. We present a few vulnerable patterns discovered by Sailfish, and their impacts in Appendix II-A and Appendix II-C, respectively.

Comparison against Sereum. Since Sereum is not publicly available, we could only compare Sailfish on the contracts in their released dataset. Sereum [41] flagged total 16 contracts for potential reentrancy attacks, of which 6 had their sources available in the EtherScan, and therefore, could be analyzed by Sailfish. Four out of those 6 contracts were developed for old Solidity versions (<0.3.x)—not supported by our framework. We ported those contracts to a supported Solidity version (0.4.14) by making minor syntactic changes which is not related to their functionality. According to Sereum, of those 6 contracts, only one (TheDAO) was a true vulnerability, while five others were its false alarms. While Sailfish correctly detects TheDAO as unsafe, it raises a false alarm for another contract (CCRB) due to imprecise modeling of untrusted external call.

**RQ1:** Sailfish emits the fewest warnings in the full dataset, and finds 47 zero-day vulnerabilities. On our manual analysis dataset, Sailfish detects all the vulnerabilities with the lowest false positive rate.

### C. Performance analysis

Table IV reports the average analysis times for each of the small, medium, and large datasets along with the full dataset. As the data shows, the analysis time increases with the size of the dataset for all the tools. Vandal [24] is the fastest analysis across all the four datasets with an average analysis time of 30.68 seconds with highest emitted warnings (52.27%).

**TABLE IV:** Analysis times (in seconds) on four datasets.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
<th>Full</th>
</tr>
</thead>
<tbody>
<tr>
<td>Securify</td>
<td>85.51</td>
<td>642.22</td>
<td>823.48</td>
<td>196.52</td>
</tr>
<tr>
<td>Vandal</td>
<td>16.35</td>
<td>74.77</td>
<td>177.70</td>
<td>30.68</td>
</tr>
<tr>
<td>Mythril</td>
<td>917.99</td>
<td>1,046.80</td>
<td>1,037.77</td>
<td>941.04</td>
</tr>
<tr>
<td>Oyente</td>
<td>148.35</td>
<td>521.16</td>
<td>675.05</td>
<td>183.45</td>
</tr>
<tr>
<td>Sailfish</td>
<td>9.80</td>
<td>80.78</td>
<td>246.89</td>
<td>30.79</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of warnings</th>
<th>St+HV</th>
<th>St+VS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2833</td>
<td>5667</td>
<td>8500</td>
</tr>
<tr>
<td>3391</td>
<td>14485</td>
<td>2436</td>
</tr>
<tr>
<td>10560</td>
<td>2076</td>
<td>7555</td>
</tr>
</tbody>
</table>

**Fig. 11:** Ablation study showing the effectiveness of value-summary analysis for reentrancy and TOD detection.

Securify [46] is approximately 6x more expensive than Vandal over the entire dataset. The average analysis time of Mythril [19] is remarkably high (941.04 seconds), which correlates with its high number of time-out cases (66.84%). In fact, Mythril’s analysis time even for the small dataset is as high as 917.99 seconds. However, another symbolic execution based tool Oyente [36] has average analysis time close to 19% to that of Mythril, as it fails to analyze most of the medium to large contracts due to the unsupported Solidity version. The analysis time of Sailfish over the entire dataset is as low as 30.79 seconds with mean analysis times of 9.80, 80.78, and 246.89 seconds for small, medium, and large ones, respectively. The mean static analysis time is 21.74 seconds as compared to the symbolic evaluation phase, which takes 39.22 seconds. The value summary computation has a mean analysis time of 0.06 seconds.

**RQ2:** While the analysis time of Sailfish is comparable to that of Vandal, it is 6, 31, and 6 times faster than Securify, Mythril, and Oyente, respectively.

### D. Ablation study

**Benefit of value-summary analysis:** To gain a better understanding of the benefits of the symbolic evaluation (Refine) and the value-summary analysis (VSA), we performed an ablation study by configuring Sailfish in three distinct modes: (a) static-only (SO), only the Explorer runs, and (b) static + havoc (St+HV), the Refiner runs, but it havoc’s all the state variables after the external call. (c) static + value summary (St+VS), the Refiner runs, and it is supplied with the value summary facts that the Explorer computes. Figure 11 shows the number of warnings emitted by Sailfish in each of the configurations. In SO mode, the Explore phase generates 3,391 reentrancy and 14,485 TOD warnings, which accounts for 3.92% and 16.75% of the contracts, respectively. Subsequently, St+HV mode brings down the number of reentrancy and TOD warnings to 2,436 and 10,560, which is a 28.16% and 27.10% reduction with respect to the SO baseline. Lastly, by leveraging value summary, Sailfish generates 2,076 reentrancy and 7,555 TOD warnings in St+VS mode, which is a 14.78% and 28.46% improvement over St+HV configuration. This experiment demonstrates that our symbolic
evaluation and value-summary analysis are indeed effective to prune false positives. Appendix III presents a real-world case study showing the advantage of value-summary analysis.

RQ3: Our symbolic evaluation guided by VSA plays a key role in achieving high precision and scalability.

Fig. 12: Relative speedup due to value summary over a path-by-path function summary based REFINE phase.

RQ4: Our novel value summary analysis is significantly faster than a classic summary-based analysis.

VIII. LIMITATIONS

Source-code dependency. Although SAILFISH is built on top of the SLITHER framework, which requires access to the source code, we do not rely on any rich semantic information from the contract source to aid our analysis. In fact, our choice of source code was motivated by our intention to build SAILFISH as a tool for developers, while enabling easier debugging and introspection as a side-effect. Our techniques are not tied to source code, and could be applied directly to bytecode by porting the analysis on top of a contract decompiler that supports variable and CFG recovery.

IX. RELATED WORK

Static analysis. Static analysis tools such as SECURIFY, MADMAX, ZEUS, SMARTCHECK, and SLITHER have been developed to detect specific vulnerabilities in smart contracts. Due to their reliance on bug patterns, they tend to over-approximate program states, which can cause false positives and missed detection of bugs. To mitigate this issue, in this work, we identified two complementary causes of SI bugs—Stale read and Destructive write. While the former is more precise than the patterns found in the previous work, the latter, which is not explored in the literature, plays a role in the missed detection of bugs (Section III). Unlike SAILFISH, which focuses on SI bugs, MADMAX uses a logic-based paradigm to target gas focused vulnerabilities. SECURIFY first computes control and data-flow facts, and then checks for compliance and violation signatures. SLITHER uses data-flow analysis to detect bug patterns scoped within a single function. The bugs identified by these tools are either local in nature, or they refrain from doing any path-sensitive reasoning—leading to spurious alarms. To alleviate this issue, SAILFISH introduces the REFINE phase that prunes significant numbers of false alarms.

Symbolic execution. MYTHRIL, OYENTE, ETHBMC, SMARTSCOPE, and MANTICORE rely on symbolic execution to explore the state-space of the contract. ETHBMC is a symbolic, bounded model checker that models EVM transactions as state transitions. TEEHER generates constraints along a critical path having attacker-controlled instructions. All these tools suffer from the limitation of traditional symbolic execution, e.g., path explosion, and do not scale well. Instead of solely relying on symbolic evaluation, SAILFISH uses the same only for validation. Since whole-contract symbolic execution is expensive, we resort to under-constrained symbolic execution aided by conditional value-summary analysis that over-approximates the preconditions required to set the state variables to certain values across all executions.

Dynamic analysis. While SEREUM and SODA perform run-time checks within the context of a modified EVM, TXSPECTOR performs a post-mortem analysis of transactions. ECFCHECKER detects if the execution of a smart contract is effectively callback-free (ECF), i.e., it checks if two execution traces, with and without callbacks, are equivalent—a property that holds for a contract not vulnerable to reentrancy attacks SAILFISH generalizes the semantics of ECF with the notion of hazardous access for SI attacks. Thus, SAILFISH is not restricted to reentrancy, instead, can express all properties that are caused by state inconsistencies. Dynamic analysis tools rely on manually-written test oracles to detect violations in response to inputs generated according to blackbox or greybox strategies. Though precise, dynamic analysis tools suffer from lack of coverage, which is not an issue for static analysis tools, such as SAILFISH.
X. CONCLUSION

We propose SAILFISH, a scalable hybrid tool for automatically identifying SI bugs in smart contracts. SAILFISH combines light-weight exploration phase followed by symbolic evaluation aided by our novel VSA. On the ETHERSCAN dataset, SAILFISH significantly outperforms state-of-the-art analyzers in terms of precision, and performance, identifying 47 previously unknown vulnerable (and exploitable) contracts.

REFERENCES


we still need to verify whether the write to the storage enables
variable, and which can be modified in the contract. However,
respectively.

The query searches for a hazardous access pair
⟨SDG query (user can first call init(),
the contract by calling selfdestruct(.)
SELFDESTRUCT attack. Ethereum uses the
resulted in making more than 155 M
Ether (around 155 M
USD inaccessible—is an example of one such real-world
attack. Ethereum uses the SELFDESTRUCT opcode to destroy
the contract from the blockchain. Contract owners use this to
manage the lifecycle of a contract. A vulnerable contract lacks
in necessary access control, which enables an attacker kill the
contract by calling selfdestruct.

Conditional suicide. Figure 13 presents a scenario where
the contract owner forgot to implement a permission check in the init() method. Though the contract checks
(Line 9) if initialized is unset before executing selfdestruct(msg.sender) (Line 13), an unauthorized user
can first call init() to set initialized to 1, and then invoke run() to kill the contract. Figure 14 presents the
SDG query (cond_suicide(s1, s2, s3)) to detect suicidal attacks. The query searches for a hazardous access pair ⟨s1, s2⟩, such
that selfdestruct(sd) is reachable from s1 in the SDG. In our example, s1 and s2 are the instructions at Line 9 and Line 5,
respectively.

If the query result is not empty, intuitively it means that
the selfdestruct(.) operation depends on some storage
variable, and which can be modified in the contract. However,
we still need to verify whether the write to the storage enables
an attacker reach the selfdestruct(.) statement. To validate that, SAILFISH invokes the REFINER module.

Unconditional suicide. In this case, a smart contract does not implement any check at all before executing selfdestruct(.). The EXPLORER module alone is enough to detect such attacks precisely, as it does not require any further path-based reasoning. The uncond_suicide(sd) query in Figure 14 detects such attacks.

\[
\text{cond_suicide}(s_1, s_2, s_3) :\quad \text{selfdestruct}(s_d), \text{reach}(s_1, s_d), \text{depend}(s_1, v), \text{hazard}(s_1, s_2, v).
\]

\[
\text{uncond_suicide}(s_d) :\quad \text{selfdestruct}(s_d), \neg \text{cond_suicide}(\_\_, \_\_, s_d).
\]

\[
\text{eth_withdrawal}(s_1, s_2, e) :\quad \text{extcall}(e, cv), \text{reach}(s_1, e), \text{depend}(s_1, v), \text{hazard}(s_1, s_2, v), cv > 0.
\]

\[
\text{generic}(s_1, s_2) :\quad \text{reach}(s_1, s_d), \text{hazard}(s_1, s_2, \_\_).
\]

Fig. 13: An example of conditional suicide vulnerability where the selfdestruct(.) (Line 13) operation can be triggered by an attacker.

Fig. 14: Rules for various attacks. Counter example generation is similar to Figure 8.

B. Unprotected Ether withdrawal.

Unprotected Ether withdrawal [3] causes an unauthorized user to withdraw Ether from the contract’s account. The first Parity incident [2] which enabled an attacker to steal over 150,000 worth of Ether, can be attributed to this attack. This vulnerability can arise due to insufficient permission check while updating a critical variable, e.g., owner, of the contract. In Figure 15, withdrawAll() correctly checks if msg.sender is the owner of the contract (onlyOwner modifier). However, newOwner() fails to check if msg.sender is authorized to update the owner variable—allowing the owner to be updated by anyone. In an attack, an attacker can first set herself as the owner by calling newOwner(), and then call withdrawAll() to transfer all the funds. The SDG query eth_withdrawal(s1, s2, e) in Figure 14 detects unprotected Ether withdrawal attack, where extcall(e,_) is reachable from s1 in the SDG, and ⟨s1, s2⟩ is a hazardous access pair. This also requires further validation by the REFINER if the query result is not empty.

```solidity
contract SuicideFeasible {
    uint256 private initialized = 0;

    function init() public {
        initialized = 1;
    }

    function run(uint256 input) {
        if (initialized == 0) {
            return;
        }

        selfdestruct(msg.sender);
    }

    function run(uint256 input) {
        if (initialized == 0) {
            return;
        }

        selfdestruct(msg.sender);
    }
}
```
Fig. 15: An example of unprotected Ether withdrawal vulnerability where the transfer (Line 18) operation can be triggered by an attacker.

C. Generic detection rules.

In addition to these specific attacks, SAILFISH can be used to test the reachability of any critical instruction present in the SDG. Assume, \( s_{id} \) is an instruction in the SDG with label id, SAILFISH can reason if \( s_{id} \) is reachable in any execution of the contract by an unauthorized user. Figure 14 presents the query \( \text{generic}(s_1, s_2) \) for generic reachability testing.

APPENDIX II
CASE STUDIES

A. Zero-day vulnerabilities

In this section, we present the unique vulnerabilities found by SAILFISH, and not detected by any other tool. We have redacted the code, and masked the program elements for the sake of anonymity and simplicity. The fact that the origin of the smart contracts can’t be traced back in most of the cases makes it hard to report these bugs to the concerned developers. Also, once a contract get deployed, it is difficult to fix any bug due to the immutable nature of the blockchain. Therefore, we do not reveal the contract addresses to prevent potential abuse.

Cross-function reentrancy: Figure 16 presents a simplified real-world contract—vulnerable to cross-function reentrancy attack due to Destructive Write (DW). An attacker can set both item_1.creator (Line 11) and item_1.game (Line 12) to an arbitrary value by invoking funcB(). In funcA(), an amount amt is transferred to item_1.creator through transferFrom—an untrusted external contract call. Therefore, when the external call is underway, the attacker can call funcB() to reset both item_1.creator, and item_1.game. Hence, item_1.creator gets transferred to a different address when Line 6 gets executed.

Delegate-based reentrancy: Figure 17 presents a real-world contract, which is vulnerable to delegate-based reentrancy attack. The contract contains three functions—(a) funcA contains the delegatecall, (b) funcB() allows application data to be modified if the assertion is satisfied, and (c) funcC contains an untrusted external call. A malicious payload can be injected in the _data argument of funcA, which, in turn, invokes funcC() with a tainted destination _to. The receiver at Line 14 is now attacker-controlled, which allows the attacker to re-enter to funcB with _isTokenFallback inconsistently set to true; thus rendering the assertion at Line 8 useless.

B. Advantage of value-summary analysis.

Figure 18 shows a real-world contract that demonstrates the benefit of the value-summary analysis. A modifier in SOLIDITY is an additional piece of code which wraps the execution of a function. Where the underscore (_) is put inside the modifier decides when to execute the original function. In this example, the public function reapFarm is guarded by the modifier nonReentrant, which sets the reentrancy_lock (shortened as L) on entry, and resets it after exit. Due to the hazardous access (Line 14 and 18) detected on workDone, EXPLORER flags this contract as potentially vulnerable. However, the value summary analysis observes that the require clause at Line 7 needs to be satisfied in order to be able to modify the lock variable L, which is encoded as: \( L = \{(false, L = false), (true, L = false)\} \). In other words, there doesn’t exist a program path that sets L to false, if the current value of L is true. While making the external call at Line 16, the program state is \( \delta = \{L \mapsto true, ...\} \), which means that L is true at that program point. Taking both the value summary and the program state into account, the REFINER decides that the corresponding path leading to the potential reentrancy bug is infeasible.
A. Analysis speedup due to value summary

Table V presents the performance improvement achieved by our value-summary analysis with respect to the path-by-path summary. When the REFINE phase is made to consume the path-by-path summary, 23.63% and 37.05% of the medium and large sized contracts which were analyzed successfully in the value summary (VS) mode, timed out in this configuration. While the VS provides 3x speedup to more medium-sized contracts (48.14%) than the large (40.68%) ones, the speedup in the 5x (22.92% (M) vs. 23.86% (L)) and 10x (5.75% (M) vs. 6.59% (L)) categories are comparable in both the size groups. Not only VS provides up to 20x speedup in our experiment, but also 78.34% of the analyses were accelerated at most five orders of magnitude.

D. False positives for reentrancy and TOD

Reentrancy. Figure 20 features a real-world contract where bTken is set inside the constructor. The static taint analysis that SAILFISH performs disregards the fact that Line 5 is guarded by a require clause in the line before; thereby making the variable tainted. Later at Line 9 when the balanceOf method is invoked on bTken, SAILFISH raises a false alarm. In another contract which enables buying and selling of tokens at a higher price, totaling price \( p \) and \( h \), and buy tokens \( n \) at a higher price, totaling price \( p_h \), and making a profit of \( (p_h - p_l) \). We illustrate one more real-world example of a TOD attack in Figure 19.

Fig. 19: Real-world example of a TOD bug.

recordBet() allows a user to place a bet, and then it adds (Line 4) the bet amount to the total balance of the contract. In settleBet(), a user receives a fraction of the total bet amount as the reward amount. Therefore, if two invocations of settleBet() having same bet value race against each other, the front-running one will earn higher reward as the value of totalBlnc[!bet], which reward is calculated on, will also be higher in that case.

Fig. 20: False positive of SAILFISH (Reentrancy).

TOD. Figure 21 presents a real-world donation collection contract, where the contract transfers the collected donations to its recipient of choice. Both SAILFISH and SECURIFY raised TOD warning as the transferred amount, \( i.e., \text{donations} \) at Line 7, can be modified by function pay() at Line 3. Though the amount of Ether withdrawn (donations) is different depending on which of withdrawDonations() and pay() get scheduled first—this doesn’t do any harm as far as the functionality is concerned. In fact, if pay() front-runs withdrawDonations(), the recipient is rewarded with a greater amount of donation. Therefore, this specific scenario does not corresponds to a TOD attack.

Fig. 21: False positive of TOD.

APPENDIX III  
EVALUATION

A. Analysis speedup due to value summary
TABLE VI: Comparison of bug finding abilities of tools only on those contracts successfully analyzed by all of them

<table>
<thead>
<tr>
<th>Bug</th>
<th>Tool</th>
<th>Safe</th>
<th>Unsafe</th>
<th>Timeout</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reentrancy</td>
<td>SAILFISH</td>
<td>6.141</td>
<td>568</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>VANDAL</td>
<td>4.916</td>
<td>1.793</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>MYTHRIL</td>
<td>5.851</td>
<td>858</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>OYENTE</td>
<td>6.652</td>
<td>57</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>SAILFISH</td>
<td>6.633</td>
<td>76</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

B. Vulnerability detection on common success

Table VI presents the number of safe and unsafe contracts with respect to 6,709 contracts successfully analyzed by all tools, excluding both the “timeout” and “error” cases. For reentrancy, SAILFISH, MYTHRIL, VANDAL, OYENTE and SAILFISH marked 8.47%, 12.79%, 26.73%, 0.85%, 1.13% of the contracts as unsafe, respectively. For TOD, SAILFISH marked 8.75% of the contracts as vulnerable. Whereas, SAILFISH and OYENTE raised alarms for 26.71%, and 13.50% of the contracts. Interestingly, OYENTE raises fewer reentrancy warnings than SAILFISH. Our manual analysis (ref. Section VII-B) which was performed on a subset of this dataset, corroborates this observation by showing that OYENTE suffers from false negatives.

APPENDIX IV

TECHNICAL DETAILS

A. Inter-contract analysis

To model inter-contract interaction as precisely as possible, we perform a backward data-flow analysis starting from the destination d of an external call (e.g., call, delegatecall, etc.), which leads to the following three possibilities: (a) d is visible from source, (b) d is set by the owner at run-time, e.g., in the constructor during contract creation. In this case, we further infer d by analyzing existing transactions, e.g., by looking into the arguments of the contract creating transaction, and (c) d is attacker-controlled. While crawling, we build a database from the contract address to its respective source. Hence, for cases (a) and (b) where d is statically known, we incorporate the target contract in our analysis if its source is present in our database. If either the source is not present, or d is tainted (case (c)), we treat such calls as untrusted, thus requiring no further analysis.

B. Detecting owner-only statements

In the context of smart contract, the owner refers to one or more addresses that play certain administrative roles, e.g., contract creation, destruction, etc. Typically, critical functionalities of the contract can only be exercised by the owner. We call the statements that implement such functionalities as owner-only statements. Determining the precise set of owner-only statements in a contract can be challenging as it requires reasoning about complex path conditions. SAILFISH, instead, computes a over-approximate set of owner-only statements during the computation of base ICFG facts. This enables SAILFISH, during the EXPLORE phase, not to consider certain hazardous access pairs that can not be exercised by an attacker.

To start with, SAILFISH initializes the analysis by collecting the set of storage variables (owner-only variables) O defined during the contract creation. Then, the algorithm computes the transitive closure of all the storage variables which have write operations that are control-flow dependent on O. Finally, to compute the set of owner-only statements, SAILFISH collects the statements which have their execution dependent on O.

C. SlithIR interpreter

Motivated by the recent advance in push-button verification for file systems and OS kernels [43], [38], [37], we build our symbolic evaluation engine on top of ROSSETTE, which is a state-of-the-art solver-aid framework that employs a hybrid strategy to combine symbolic execution and bounded model checking [45].

Writing an interpreter. The core of our symbolic engine is an interpreter for SlithIR [26] shown in Figure 1. It is written in the Rosette [45] language whose syntax is based on S-expressions. For example, (+ 1 pc) returns the program counter by 1. A function is defined using expression (define (name args) body ...). From a high level perspective, Figure 1 defines the program state as a structure with pc (i.e., program counter), a hash table for storage, and a vector of registers. The core interpreter takes a program state and a program, and runs in an iterative loop until it encounters a return instruction. For each instruction, the interpret-inst
function updates the program state according to the semantics of that instruction.  

**Lifting to a verifier.** The interpreter in Figure 1 runs as a normal emulator for programs written in SlitherIR when it runs with a concrete state. What is fascinating is that when the interpreter runs on *symbolic state*, ROSETTE automatically lifts the interpreter to a verifier, and executing a program $P$ on the symbolic state encodes *all possible behaviors of $P$*. In particular, ROSETTE symbolically evaluates the interpreter over subgraph $G$ to generate constraints whose satisfiability determines whether there indeed exist a valid path from $V_0$ to $V^*$ in $G$. 